OpenGL Implementation Guide

for HP-UX 11.x

Manufacturing Part Number : B2355-IE002

Edition E0206

Contents

1 overview of OpenGL

OpenGL is a hardware-independent Application Programming Interface (API) that provides an interface to graphics operations. HP's implementation of OpenGL converts API commands to graphical images via hardware and/or software functionality.

introduction

The OpenGL interface consists of a set of commands that allow applications to define and manipulate three-dimensional objects. The commands include:

- Geometric primitive definitions
- Viewing operations
- Lighting specifications
- Primitive attributes
- Pipeline control
- Rasterization control

OpenGL has been implemented on a large number of vendor platforms where the graphics hardware supports a wide range of capabilities (for example, frame buffer only devices, fully accelerated devices, devices without frame buffer, etc.).

For more information on OpenGL, refer to these documents, published by Addison-Wesley and shipped with HP's implementation of OpenGL:

- *OpenGL Programming Guide*
  Instruction on programming in OpenGL, offered in a tutorial format.
- *OpenGL Reference Manual*
  A reference that contains details on all standard OpenGL functions, as well as utility (GLU) functions and X-windows (GLX) functions.
- *OpenGL Programming for the X Window System*
  Instructions on interfacing OpenGL with the X Window system.

the OpenGL product

This section provides information about HP's implementation of the OpenGL product, as well as information about the standard OpenGL product.

hp's implementation of OpenGL

Topics covered in this section are:

- HP's implementation of the OpenGL libraries
- Supported graphics devices
- Supported visuals
- Visual support for other graphics devices
- Buffer sharing between multiple processes

hp's implementation of the OpenGL libraries

The OpenGL product does not support archived libraries.

HP's implementation of OpenGL provides the following libraries:

- libGL.sl
  OpenGL shared library
- libGLU.sl
  OpenGL utilities library

There are two sets of libraries, one for 32-bit and one for 64-bit. The 32-bit path is:

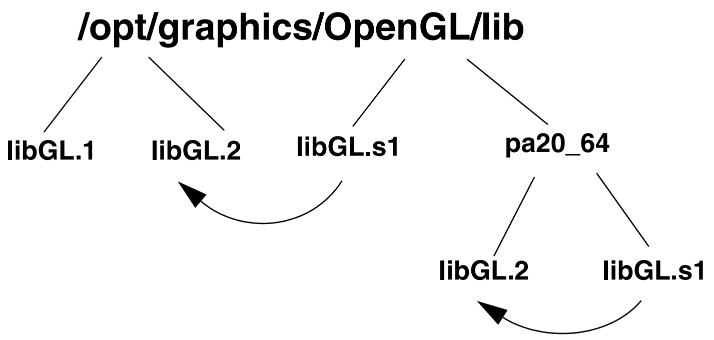
```
/opt/graphics/OpenGL/lib
```

The 64-bit libraries are in a subdirectory:

```
/opt/graphics/OpenGL/lib/pa20_64
```

The following graphic depicts the organization of these libraries, which follows the HP-UX standard for 64-bit libraries.

The arrows in the graphic represent symbolic links.



In the library directory, you will see various versions. For example:

- libGL.1 is a 10.20 compatible library for applications which were built on 10.20.
- libGL.2 is the library which is for applications built on 11.x (this is why libGL.sl is a symbolic link to libGL.2).

There is only one set of libraries in 11.x for 64-bit. These libraries are version number 2.

The other libraries you will see in these directories are all drivers for specific graphics devices. Device drivers, which have names of the form libdd* are loaded automatically at execution time, contingent upon device type. The user program does not specifically link in a device driver.

supported graphics devices

These are the graphics devices that support OpenGL:

- HP Visualize fxe
- HP Visualize fx-5
- HP Visualize fx-10
- HP Fire GL-UX
- ATI FireGL X1
- ATI FireGL T2
- ATI FireGL X3

supported operating systems

OpenGL is supported on PA-RISC 2.0 systems running the 64-bit version of HP-UX 11.0 and 11i v1 (11.11).

supported visuals

In this section, each visual table will have a graphics device associated with it. For information on visual support for graphics devices not in the above list, read the subsequent section “Visual Support for Other Graphics Devices.”

Table 1-1 Visual Table for HP Visualize fxe

X Visual Information			OpenGL GLX Information														
Class	**Depth**	**Color Map Size**	**Buffer Size**	**Overlay=1 or Image=0**	**RGBA=1 or Index=0**	**Double Buffer**	**# Aux. Buffers**	**Color Buffer**					**Stencil**	**Accumulation Buffer**			
								R	**G**	**B**	**A**	**Z**		**R**	**G**	**B**	**A**
PseudoColor	8	256	8	1	0	0	0	0	0	0	0	0	0	0	0	0	0
PseudoColor	8	256	8	0	0	0	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	8	256	8	0	0	1	0	0	0	0	0	24	4	0	0	0	0
TrueColor	24	256	24	0	1	0	0	8	8	8	0	24	4	16	16	16	0
TrueColor	24	256	24	0	1	1	0	8	8	8	0	24	4	16	16	16	0

Table 1-2 Visual Table for HP Visualize fx-5 / fx-10

X Visual Information			OpenGL GLX Information														
Class	**Depth**	**Color Map Size**	**Buffer Size**	**Overlay=1 or Image=0**	**RGBA=1 or Index=0**	**Double Buffer**	**# Aux. Buffers**	**Color Buffer**					**Stencil**	**Accumulation Buffer**			
								R	**G**	**B**	**A**	**Z**		**R**	**G**	**B**	**A**
PseudoColor	8	256	8	1	0	0	0	0	0	0	0	0	0	0	0	0	0
PseudoColor	8	256	8	0	0	0	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	8	256	8	0	0	1	0	0	0	0	0	24	4	0	0	0	0
TrueColor	24	256	24	0	1	0	0	8	8	8	8	24	4	16	16	16	16
TrueColor	24	256	24	0	1	1	0	8	8	8	8	24	4	16	16	16	16
TrueColor	24	256	24	0	1	0	0	8	8	8	8	24	4	16	16	16	0
TrueColor	24	256	24	0	1	1	0	8	8	8	8	24	4	16	16	16	0

Table 1-3 Visual Table for HP Fire GL-UX

X Visual Information			OpenGL GLX Information														
Class	**Depth**	**Color Map Size**	**Buffer Size**	**Overlay=1 or Image=0**	**RGBA=1 or Index=0**	**Double Buffer**	**# Aux. Buffers**	**Color Buffer**					**Stencil**	**Accumulation Buffer**			
								R	**G**	**B**	**A**	**Z**		**R**	**G**	**B**	**A**
PseudoColor	8	256	8	1	0	0	0	0	0	0	0	0	0	0	0	0	0
PseudoColor	8	256	8	1	0	1	0	0	0	0	0	0	0	0	0	0	0
TrueColor	24	256	24	0	1	0	0	8	8	8	0	24	4	16	16	16	0
TrueColor	24	256	24	0	1	1	0	8	8	8	0	24	4	16	16	16	0

Table 1-3 Visual Table for HP Fire GL-UX (Continued)

X Visual Information			OpenGL GLX Information															
Class	Depth	Color Map Size	Buffer Size	Overlay=1 or Image=0	RGBA=1 or Index=0	Double Buffer	# Aux. Buffers	Color Buffer					Stencil	Accumulation Buffer			
								R	G	B	A	Z		R	G	B	A
TrueColor	24	256	32	0	1	0	0	8	8	8	8	24	4	16	16	16	16
TrueColor	24	256	32	0	1	1	0	8	8	8	8	24	4	16	16	16	16

Table 1-4 Visual Table for ATI Fire GL T2/X1/X3

X Visual Information			OpenGL GLX Information														
Class	Depth	Color Map Size	Buffer Size	Overlay=1 or Image=0	RGBA=1 or Index=0	Double Buffer	# Aux. Buffers	Color Buffer					Stencil	Accumulation Buffer			
								R	G	B	A	Z		R	G	B	A
PseudoColor	8	256	8	1	0	0	0	0	0	0	0	0	0	0	0	0	0
PseudoColor	8	256	8	1	0	1	0	0	0	0	0	0	0	0	0	0	0
TrueColor	24	256	32	0	1	0	0	8	8	8	8	24	8	16	16	16	16
TrueColor	24	256	32	0	1	1	0	8	8	8	8	24	8	16	16	16	16

stereo visual support for Visualize fx-5, fx-10, and Fire GL-UX

When a monitor is configured in a stereo capable mode, HP Visualize fx-5, fx-10 and Fire GL-UX will have the following additional stereo visuals available. For more information on OpenGL stereo, read the section "Running HP's Implementation of the OpenGL Stereo Application," found in Chapter 3 of this document.

Table 1-5 Stereo Visual Support for HP Visualize fx-5 and fx-10

X Visual Information			OpenGL GLX Information																
Class	Depth	Color Map Size	Buffer Size	Overlay= 1 or Image=0	RGBA= 1 or Index= 0	Double Buffer	Stereo	# Aux. Buffers	Color Buffer					Stencil	Accumulation Buffer			
									R	G	B	A	Z		R	G	B	A
PseudoColor	8	255	8	0	0	a	1	0	0	0	0	0	24	4	0	0	0	0
TrueColor	24	256	24	0	1	a	1	0	8	8	8	0	24	4	16	16	16	0
TrueColor	24	256	24	0	1	a	1	0	8	8	8	8	24	4	16	16	16	16

a. Depth- and stencil buffers are only allocated for image-plane visuals.

Table 1-6 Stereo Visual Support for HP Fire GL-UX

X Visual Information			**OpenGL GLX Information**															
Class	**Depth**	**Color Map Size**	**Buffer Size**	**Overlay =1 or Image=0**	**RGBA= 1 or Index=0**	**Double Buffer**	**Stereo**	**# Aux. Buffers**	**Color Buffer**					**Stencil**	**Accumulation Buffer**			
									R	**G**	**B**	**A**	**Z**		**R**	**G**	**B**	**A**
TrueColor	24	256	24	0	1	1	1	0	8	8	8	0	24	4	16	16	16	0
TrueColor	24	256	32	0	1	1	1	0	8	8	8	8	24	4	16	16	16	16

Table 1-7 Stereo Mode Visual Support for ATI Fire X1/X3

X Visual Information			**OpenGL GLX Information**															
Class	**Depth**	**Color Map Size**	**Buffer Size**	**Overlay=1 or Image=0**	**RGBA=1 or Index=0**	**Double Buffer**	**Stereo**	**# Aux. Buffers**	**Color Buffer**					**Stencil**	**Accumulation Buffer**			
									R	**G**	**B**	**A**	**Z**		**R**	**G**	**B**	**A**
PseudoColor	8	256	8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PseudoColor	8	256	8	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
TrueColor	24	256	32	0	1	0	0	0	8	8	8	8	24	8	16	16	16	16
TrueColor	24	256	32	0	1	0	1	0	8	8	8	8	24	8	16	16	16	16
TrueColor	24	256	32	0	1	1	0	0	8	8	8	8	24	8	16	16	16	16
TrueColor	24	256	32	0	1	1	1	0	8	8	8	8	24	8	16	16	16	16

visual support for other graphics devices

The OpenGL product can be used with devices that have no hardware OpenGL support using the Virtual Memory Driver (VMD) in Virtual GLX mode (VGL). In addition, VMD allows you to use many X11 drawables (local or remote) as "virtual devices" for three-dimensional graphics with OpenGL. This includes rendering to X terminals and other non-GLX extended X servers.

Table 1-8 Visuals Table for VMD

X Visual Information			**OpenGL GLX Information**														
Class	**Depth**	**Color Map Size**	**Buffer Size**	**Overlay=1 or Image=0**	**RGBA=1 or Index=0**	**Double Buffer**	**# Aux. Buffers**	**Color Buffer**					**Stencil**	**Accumulation Buffer**			
								R	**G**	**B**	**A**	**Z**[a]		**R**	**G**	**B**	**A**
PseudoColor	8	256	8	0	0	[b]	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	8	256	8	0	0	[b]	0	0	0	0	0	0	0	0	0	0	0
TrueColor	24	256	24	0	1	[b]	0	8	8	8	[c]	24	4	16	16	16	16
DirectColor	24	256	24	0	1	[b]	0	8	8	8	[c]	24	4	16	16	16	16

a. Depth- and stencil buffers are only allocated for image-plane visuals.

b. Double buffering is set to True (1) if the X visual supports the X double-buffering extension (DBE).

c. Alpha will only work correctly on 12- and 24-bit TrueColor and DirectColor visuals when the X server does not use the high-order nibble/byte in the X visual. Also, note that when alpha is present, Buffer Size will be 16 for the 12-bit visuals and 32 for the 24-bit visuals.

buffer sharing between multiple processes and threads

In the OpenGL implementation, all drawable buffers that are allocated in virtual memory are not sharable among multiple processes. As an example, on a HP Visualize fx-5 configuration, the accumulation buffer for a drawable resides in virtual memory (VM) and therefore, each OpenGL process rendering to the same drawable through a direct rendering context, will have its own separate copy of the accumulation buffer. For more information on hardware and software buffer configurations for OpenGL devices, see Tables 1-1 through 1-8 in the Supported Visuals section of this chapter.

True buffer sharing between multiple processes can be accomplished by utilizing indirect rendering contexts. In this case, rendering on behalf of all GLX clients is performed by the X server OpenGL daemon process, and there is only one set of virtual memory buffers per drawable.

Within a single process, multiple threads will share virtual memory buffers (both rendering and accumulation buffers) by default. GLX-compliant concurrent rendering into these buffers is supported. It is the responsibility of the application to synchronize buffer access or partition the rendering buffer amongst individual threads, if desired.

SIGCHLD and the GRM daemon

The Graphics Resource Manager daemon (grmd) is started when the X11 server is started. In normal operation, an OpenGL application will not start the daemon, and as a result grmd will not be affected by the SIGCHLD manipulation that occurs as part of that start-up. However, if grmd dies for some reason, the graphics libraries will restart grmd whenever they need shared memory. An example of where this can occur is during calls to glXCreateContext or glXMakeCurrent.

threads support

threads support in November, 1999 11. ACE release Starting with the *HP-UX 11.0 Additional Core Enhancements (ACE) (November, 1999)* release, HP OpenGL will support Level 1b threads. This means HP OpenGL can be used in a threaded application, but OpenGL graphics calls must be restricted to a single thread, which remains the same thread for the duration of the process. This is not the same as OpenGL calls being made in one thread at a time. Other threads can be used for computations, etc.

Using OpenGL graphics in a Kernel threaded application requires that the application link with libpthread.sl (not the archived version, libpthread.a).

OpenGL libraries are not cancel safe or fork safe.

A context can only be made current in the dedicated graphics thread.

multiple graphics threads support in June, 2000 11. ACE OpenGL Starting with the June, 2000 11.ACE OpenGL release, OpenGL will Support Level 2 threads. This means HP OpenGL can be used in threaded applications, and more than one thread can use OpenGL.

Using OpenGL graphics in a Kernel threaded application requires that the application link with libpthread.sl (not the archived version, libpthread.a).

OpenGL libraries are not cancel safe or fork safe.

A given context can only be current in one thread at a time.

additional documentation For more information on using threads, see the following documentation:

- The **devresource.hp.com** web site (search for "Threads and Multiprocessing")
- *The OpenGL Programming Guide*
- *The OpenGL Reference Manual*
- *Threadtime* by S. Norton and M. Dipasquale

64-bit programming

Starting with the HP-UX 11.0 Additional Core Enhancements (ACE) (November, 1999) release, HP OpenGL will support 64-bit programming. Applications using 64-bit computing are supported on SPUs with 64-bit capabilities only; they are not supported on 32-bit SPUs.

For information on porting your application to take advantage of 64-bit capabilities, see the **devresource.hp.com** web site. Search for "64-bit Computing."

64-bit OpenGL allows “large data space” because the pointers are now 64-bit. But, the OpenGL data types themselves are the same as the 32-bit library. For example, GLint is a 32-bit integer, not a 64-bit long.

All 64-bit OpenGL libraries are located in `/opt/graphics/OpenGL/lib/pa20_64`. The following sample compile and link lines may help you to build your application once it has been ported to take advantage of 64-bit capabilities:

Sample 32-bit compile and link:

```
cc -g -Aa -D_HPUX_SOURCE -z -I/opt/graphics/OpenGL/include\
-I/usr/include/X11R6 -o cube.32 cube.c
-L/opt/graphics/OpenGL/lib\
-L/usr/lib/X11R6 -ldld -lGLU -lGL -lXHP11 -lXext -lX11 -lm
```

Sample 64-bit compile and link (for 11.x only):

```
cc -g -Aa +DA2.0W -D_HPUX_SOURCE -z
-I/opt/graphics/OpenGL/include -I/usr/include/X11R6 -o cube.64
cube.c -L/opt/graphics/OpenGL/lib/pa20_64
-L/usr/lib/X11R6/pa20_64 -L/usr/lib/pa20_64  -L/usr/lib -ldld\
-lGLU -lGL -lXHP11 -lXext -lX11 -lm
```

using libGL in 64-bit together with the +compat linker option Because of a limitation in the 64-bit linker, if the +compat linker option is used, -lc must appear in the link order before -lGL. Otherwise, a segmentation violation will occur when running the linked program.

The following partial compile line shows the relevant order:

```
 cc +DA2.OW prog.c -Wl,+compat
-L/opt/graphics/OpenGL/lib/pa20_64 -lc -lGL
```

When not using -Wl,+compat, the link order should have -lGL before -lc. By default, cc implicitly links in -lc as the last library in a link. Without Wl,+compat, a partial compile line is:

```
 cc +DA2.OW prog.c -L/opt/graphics/OpenGL/lib/pa20_64 -lGL -lc
```

or

```
 cc +DA2.OW prog.c -L/opt/graphics/OpenGL/lib/pa20_64 -lGL
```

SLS support

When the display is in a multi-display configuration using the XServer Single Logical Screen (SLS) extension, OpenGL can and will render to windows on or spanning any of the SLS displays. This rendering is done at some loss of performance. For full single display performance, define the HPOGL_SLS_LOCK_WINDOW environment variable before executing the program. The define value should be the display number where the window will reside. When the window is on this display, full performance can be had; when it is on other displays, the window will be blank. For more information see the XServer documentation of SLS.

the standard OpenGL product

This section covers the following topics:

- The OpenGL Utilities Library (GLU)
- Input and Output Routines
- The OpenGL Extensions for the X Window System (GLX)

the OpenGL Utilities Library (GLU)

The OpenGL Utilities Library (GLU) provides a useful set of drawing routines that perform such tasks as:

- Generating texture coordinates
- Transforming coordinates
- Tessellating polygons
- Rendering surfaces
- Providing descriptions of curves and surfaces (NURBS)
- Handling errors

For a detailed description of these routines, refer to the Reference section or the *OpenGL Reference Manual*.

input and output routines

OpenGL was designed to be independent of operating systems and window systems, therefore, it does not have commands that perform such tasks as reading events from a keyboard or mouse, or opening windows. To obtain these capabilities, you will need to use X Windows routines (those whose names start with “glX”).

the OpenGL extensions for the X Window system (GLX)

The OpenGL Extensions to the X Window System (GLX) provide routines for:

- Choosing a visual
- Managing the OpenGL rendering context

- Off-screen rendering
- Double-buffering
- Using X fonts

For a detailed description of these routines, refer to the Reference section or the OpenGL Reference Manual.

mixing of OpenGL and Xlib

The OpenGL implementation conforms to the specification definition for mixing of Xlib and OpenGL rendering to the same drawable. The following points should be considered when mixing Xlib and OpenGL:

- OpenGL and Xlib renderers are implemented through separate pipelines and control streams, thus, rendering synchronization must be performed as necessary by the user's application via the GLX `glXWaitX()` and `glXWaitGL()` function calls.
- Xlib rendering does not affect the Z-buffer, so rendering in X and then OpenGL would result in the OpenGL rendering replacing the Xlib rendering. This is true if the last OpenGL rendering to the Z-buffer at that location resulted in the depth test passing.

Note that mixing Xlib rendering with OpenGL rendering as well as with VMD, when using alpha buffers, can produce unexpected side effects and should be avoided.

Gamma correction

Gamma correction is used to alter hardware colormaps to compensate for the non-linearities in the phosphor brightness of monitors. Gamma correction can be used to improve the “ropy” or modulated appearance of antialiased lines. Gamma correction is also used to improve the appearance of shaded graphics images, as well as scanned photographic images that have not already been gamma corrected.

OpenGL extensions

The extensions listed in this section are extensions that Hewlett-Packard has created; that is, in addition to those standard functions described in the *OpenGL Programming Guide*, *OpenGL Reference Manual*, and *OpenGL Programming for the X Window System*.

visibility test extensions

HP supports extensions for visibility testing and occlusion culling.

See the on-line *Reference Manual* for information on HP's visibility test extensions, `glVisibilityBufferHP` and `glNextVisibilityTestHP`.

occlusion extension

This occlusion culling extension defines a mechanism whereby an application can determine the non-visibility of some set of geometry based on whether an encompassing set of geometry is non-visible. In general, this feature does not guarantee that the target geometry is visible when the test fails, but is accurate with regard to non-visibility.

Typical usage of this feature would include testing the bounding boxes of complex objects for visibility. If the bounding box is not visible, then it is known that the object is not visible and need not be rendered.

occlusion culling code fragments

The following is a sample code segment that shows a simple usage of occlusion culling.

```
/* Turn off writes to depth and color buffers */
glDepthMask(GL_FALSE);
glColorMask (GL_FALSE, GL_FALSE, GL_FALSE);
/* Enable Occlusion Culling test */
glEnable(GL_OCCLUSION_TEST_HP);
for (i=0; i < numParts; i++) {
/* Render your favorite bounding box */
renderBoundingBox(i);
/* If bounding box is visible, render part */
glGetBooleanv(GL_OCCLUSION_RESULT_HP, &result);
if (result) {
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
renderPart(i);
glDepthMask(GL_FALSE);
glColorMask (GL_FALSE, GL_FALSE, GL_FALSE);
                           }
                      }
/* Disable Occlusion Culling test */
glDisable(GL_OCCLUSION_TEST_HP);
/* Turn on writes to depth and color buffers */
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
```

The key idea behind occlusion culling is that the bounding box is much simpler (i.e., fewer vertices) than the part itself. Occlusion culling provides a quick means to test non-visibility of a part by testing its bounding box.

It should also be noted that this occlusion culling functionality is very useful for viewing frustum culling. If a part's bounding box is not visible for any reason (not just because it's occluded in the Z-buffer) this test will give correct results.

To maximize the probability that an object is occluded by other objects in a scene, the database should be sorted and rendered from front to back. Also, the database may be sorted hierarchically such that the outer objects are rendered first and the inner are rendered last. An example would be rendering the body of an automobile first and the engine and transmission last. In this way, the engine would not be rendered due to the bounding box test indicating that the engine is not visible.

Table 1-9 Enumerated Types for Occlusion

Extended Area	**Enumerated Type**	**Description**
Enable.Disable/IsEnabled	GL_OCCLUSION_TEST_HP Default: Disabled	pname variable.
Get*	GL_OCCLUSION_TEST_RESULT_HP Default: Zero (0)	pname variable

For related information, see the functions: glGet, glEnable, glDisable, and glIsEnabled.

GL_HP_supersample extension

This supersample extension defines a mechanism for enabling and disabling a full scene anti-aliasing method. It is supported on Visualize fx-10b, FireGL-UX and FireGL T2/X1/X3 graphics hardware. On the FireGL hardware the X Server must be configured to enable this capability in order for the extension to be present.

On the FireGL-UX graphics hardware, this extension is enabled by using SAM to adjust the Display->X Server Configuration->Screen Options set the "FSAA" option to a supersample resolution (eg. 2.0x2.0) which results in four samples per screen pixel.

On the FireGL T2/X1/X3 this extension is enabled in a similar manner by using SAM to adjust the Display->X Server Configuration->Screen Options set the "FSAAScale" option to a super sample resolution (e.g. 2, 4, or 6) which results in the number of samples per screen pixel. The actual number of samples used will depend on the availability of video memory to implement the supersample function and will vary based on screen resolutions and other functionality that has been also enabled such as stereo display and offscreen pixmaps.

This extension can be enabled either via the OpenGL API using the glEnable(GL_SUPERSAMPLE_HP) call or via the environment variable HPOGL_FORCE_SCENEAA=1 (when in direct rendering mode). The call glDisable(GL_SUPERSAMPLE_HP) can be used to disable this mode. These calls need to be made in the same OpenGL rendering context that is to be supersampled.

rendering details

This section provides the details for several of HP's rendering capabilities. These rendering capabilities range from the way HP implements its default visuals to the way HP deals with the decomposition of concave quadrilaterals.

default visuals

Instead of placing the default visual in the deepest image buffer, HP puts the default visual in the overlay planes. This behavior can be modified using SAM to adjust the Display->X Server Configuration->Screen Options.

EXP and EXP2 fogging

The Virtual Memory Driver's implementation of fog applies fog per fragment. Hardware devices implement EXP and EXP2 fog per fragment and linear fog per vertex.

bow-tie quadrilaterals

A quadrilateral has four vertices that are coplanar. When this quadrilateral is twisted and you look at a front view of it on the display, there appears to be a fifth vertex. This fifth vertex which is not a true vertex will have no attributes, therefore, the color at what appears to be the intersection of two lines will in most cases be different from what is expected. HP treats the two parts of the bow tie as two separate triangles that have attributes assigned to their vertices. This special rendering process takes care of the color problem at the non-existent fifth vertex. To learn how other implementations of OpenGL deal with bow-tie quadrilaterals, read the section "Describing Points, Lines, and Polygons" in Chapter 2 of the OpenGL Programming Guide.

decomposition of concave quadrilaterals

HP determines whether the concave quadrilateral will become front-facing or back-facing prior to dividing the quadrilateral into triangles. HP then divides the surface into two triangles between vertices zero and two or one and three depending on the vertex causing concavity.

vertices outside of a begin/end pair

HP's implementation of this specification is indeterminate as defined by the OpenGL standard.

index mode dithering

If dithering is enabled in indexed visuals, 2D functions such as glDrawPixels() and glBitmap() will not be dithered.

environment variables

Here is a list of environment variables used by HP's implementation of OpenGL.

`HPOGL_ALLOW_LOCAL_INDIRECT_CONTEXTS`

This variable may be set if a need arises to really create a local indirect context. By default, if an indirect context is requested for a local HP display connection, a direct context will be created instead because the performance will be much better.

`HPOGL_ENABLE_MIT_SHMEM`

When rendering locally using the VM Driver, this variable allows the server and client to look at the rendering buffer at the same time. This variable has no effect through DHA. It merely eliminates the data transfer for XPutImage() that is done by VMD. This only offers a performance improvement on simple wireframes. Under most circumstances, it does not provide any performance improvements.

`HPOGL_FORCE_VGL`

This variable can be set to force HP's Virtual GL (VGL) rendering mode using VMD. This differs from HPOGL_FORCE_VMD in that the GLX Visual list and other GLX extension information is not retrieved from the GLX Server extension, but is rather synthesized from standard X Visual information and the capabilities known to exist in VMD.

`HPOGL_FORCE_VMD`

This variable forces clients to render through the VMD. This variable can be used as a temporary fix and/or a diagnostic. You should set this variable when a rendering defect in the hardware device driver is suspected. When this variable is set, rendering speed will slow down. If rendering is identical in both hardware and software, then this may indicate a problem in the application code.

`HPOGL_LIB_PATH`

This variable can be used to load OpenGL driver libraries from a directory outside the standard LIB_PATH. This variable should be set to the actual directory the libraries are in, with or without a trailing'/'.

`HPOGL_LIGHTING_SPACE` (HP Visualize fx family only)

. This variable allows the user to specify the coordinate space to be used for lighting. By default, HP's implementation of the OpenGL will select the lighting space. Possible values are:

```
HPOGL_LIGHTING_SPACE=OC
HPOGL_LIGHTING_SPACE=EC
```

where OC = Object Coordinates and EC = Eye Coordinates. For details on the lighting space, see the sections "Lighting Space" and "Optimization of Lighting" found in Chapter 5. This option is available only for the HP Visualize fx family of graphics devices.

`HPOGL_TXTR_SHMEM_THRESHOLD` (HP Visualize fx family only)

This variable sets a fence for the use of process memory vs. shared memory. Any 2D or 3D texture that has a size greater than or equal to the threshold set is stored in shared memory. The initial value is set to 10241024 bytes. This variable should be set to the byte size desired for shared memory usage. This option is available only for the HP Visualize fx family of graphics devices.

new environment variables as of release 1.05

The performance of HP OpenGL double buffering has been improved for Release 1.05 of HP's implementation of OpenGL 1.1. With this performance enhancement, there is a low probability that minor image tearing may be visible during buffer-swap operations. The majority of OpenGL applications will see no difference in the visual quality of double buffering. However, this tearing may be noticed by some OpenGL applications that render simultaneously to multiple windows. Two environment variables can be set in an application environment to control whether or not the new faster buffer swapping method is in effect:

`HPOGL_DSM_ENABLE_FAST_BUFFER_SWAP`

When set to any non-NULL value, the new or faster double buffering method will be used. (This is the default behavior and does not need to be set except to override glHint as discussed below)

`HPOGL_DSM_DISABLE_FAST_BUFFER_SWAP`

When set to any non-NULL value, the old or slower double buffering method will be used.

Additionally, an application can programmatically switch between the slower and faster double buffering methods using the following new glHint calls:

```
glHint(GL_BUFFER_SWAP_MODE_HINT_HP, GL_FASTEST);
```

Switches to the faster double buffering method.

```
glHint(GL_BUFFER_SWAP_MODE_HINT_HP, GL_NICEST);
```

Switches to the slower double buffering method.

Note that setting either HPOGL_DSM_ENABLE_FAST_BUFFER_SWAP or HPOGL_DSM_DISABLE_FAST_BUFFER_SWAP in the application environment will disable and thus override the behavior of the new glHint calls.

2 installation and setup

For HP-UX 11.X, the box containing the “HP-UX 11.0 Install and Core OS” CD-ROM will also hold a second CD-ROM entitled “HP-UX 11.0 Core Operating Systems Options.” The OpenGL run-time and developer’s products are both available on the HP-UX 11.0 Core Operating Systems Options CD-ROM in the “Graphics and Technical Computing Software” bundle (B6268AA).

verification instructions

This section provides you with the necessary information for determining if your OpenGL product has been installed.

is your system software preloaded with instant ignition?

Your workstation is preloaded with software, which may include OpenGL, if it was ordered with the Instant Ignition option. A label attached to the workstation in its shipping carton confirms the workstation is preloaded:

Important

This product contains preloaded software.
Do not initialize internal hard disk drive.

verify that OpenGL is on your workstation

To verify that OpenGL is installed correctly on your system, execute:

```
/usr/sbin/swlist -l product
```

This will give you a list of all of the products on the system, and in that product list you will see lines similar to the following if HP OpenGL has been installed on your system.

```
OpenGLDevKit   B.11.00.20 HP-UX OpenGL 3D Graphics API
Developer's Kit
OpenGLRunTime  B.11.00.20 HP-UX OpenGL Run Time
Environment
```

If OpenGL is not preloaded, you will need to install it by following the steps in the subsequent sections.

installing OpenGL

Installing the software involves the following steps:

1. Install OpenGL.
2. Check log file.
3. Verify the product.

Each step is described on the subsequent sections.

1. install OpenGL

For 11.00 and 11.11, OpenGL is bundled with the HP-UX Core Operating Systems Option CD-ROM in the “Graphics and Technical Computing Software” bundle B6268AA.

If your system is Instantly Ignited, your OpenGL product is already installed. To verify that the OpenGL developer’s programming environment has been installed on your system, read the section “Verify that OpenGL is on Your Workstation” above.

If OpenGL is installed, you are done with the section. If OpenGL is not installed, execute this command (as root):

```
   /usr/sbin/swinstall
```

...and follow the installation instructions provided in the document Managing HP-UX Software with SD-UX. OpenGLDevKit is the product to install.

The OpenGL development and runtime environment product includes the filesets shown in Table 2-1. To list these filesets, execute this command:

```
   /usr/sbin/swlist -l fileset [OpenGLDevKit |
OpenGLRunTime]
```

Table 2-1 **OpenGL Development Environment Filesets for 11.0 and 11.11**

OpenGLDevKit Fileset	Contains
OPENGL–CONTRIB	Contributed or unsupported program files
OPENGL–PRG	Files necessary for the OpenGL programming environment
OPENGL–WEBDOC	Online documentation files
OPENGL–64–CONTRB	Contributed or unsupported program files
OPENGL–64–EXPL	64-bit example programs

Table 2-2 **OpenGL Runtime Environment Filesets for 11.0 and 11.11**

OpenGLRuntime Fileset	Contains
OPENGL–DEMO	OpenGL Demonstration Programs
OPENGL–RUN	OpenGL Run Time Support Files
OPENGL–SHLIBS	OpenGL Shared Libraries
OPENGL–64–DEMO	OpenGL Demonstration Programs
OPENGL–64–RUN	OpenGL Run Time Support File
OPENGL–64–SHLB	OpenGL Shared Libraries

2. check log file

Once you have completed the installation process, look at the contents of the file `/var/adm/sw/swinstall.log`. This file lists the filesets loaded, the customize scripts that ran during the installation process, and informative messages. Error messages that resulted from attempts to write across an NFS mount point may appear in this file and, if present, may be ignored.

3. verify the product

Here are three methods for determining if you have correctly installed OpenGL on your system.

- Run the program

  ```
  /opt/graphics/OpenGL/demos/verify_install
  ```

  If OpenGL has been correctly installed on your system, running verify_install will cause a window containing a 3D rendering of the text “OpenGL” to open on your monitor.
- Run any of the demos located in the directory:

  ```
  /opt/graphics/OpenGL/examples
  ```

  This directory is installed with the `OPENGL-EXAMPLE` fileset.
- Compile, link and run one of your existing OpenGL programs.

The `README` file in the `examples` directory contains instructions on how to set up and run the examples.

Example programs from the OpenGL Programming Guide are installed in the directory:

```
/opt/graphics/OpenGL/contrib/glut_samples
```

which also contains a `README` file.

the OpenGL file structure

The OpenGL file structure is compliant with the file structure of the 11.x file systems. Here is a list of files and directories that are a part of the OpenGL file structure.

`/opt/graphics/OpenGL/contrib/libwidget`

This directory contains a Motif widget library and source code.

`/opt/graphics/OpenGL/include/GL`

This directory contains header files needed for OpenGL development.

`/opt/graphics/OpenGL/contrib/glut_samples`

This directory contains example OpenGL programs that are referenced in the *OpenGL Programming Guide*, Second Edition published by Addison-Wesley.

`/opt/graphics/OpenGL/contrib/libglut`

This directory contains Mark Kilgard's OpenGL Utility Toolkit (GLUT). This is a window-system-independent toolkit for writing simple OpenGL programs.

`/opt/graphics/OpenGL/lib`

This directory contains the following run-time shared libraries:

- `libGLU.sl`
- `libGL.sl`

`/usr/lib/X11/Xserver/brokers/extensions/Glx.1`
`/usr/lib/X11/Xserver/modules/extensions/HP/glx.1`

These are libraries for the GLX extension to X windows. This directory also contains other run-time libraries including device drivers.

The location of the run-time shared libraries is:

```
/opt/graphics/openGL/lib
```

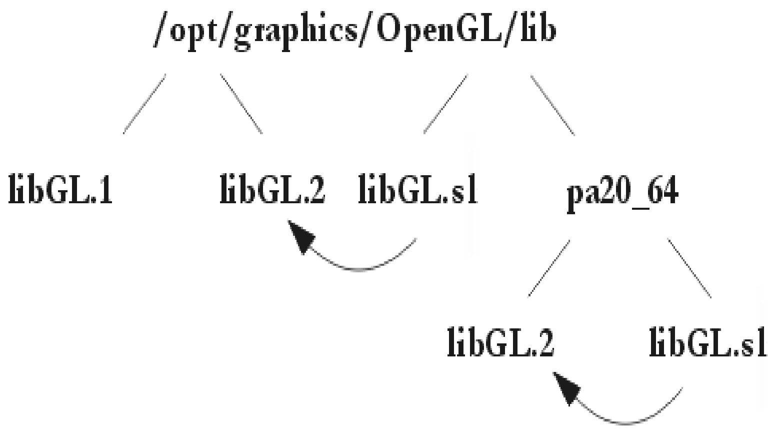
For 11.0 there are two sets of libraries, one for 64-bit and one for 32-bit.

```
/opt/graphics/OpenGL/lib
```

The 64-bit libraries are in a subdirectory:

```
/opt/graphics/OpenGL/lib/pa20_64
```

The following graphic depicts the organization of these libraries, which follows the HP-UX standard for 64-bit libraries.



In the library directory, you will see various versions. For example:

- `libGL.1` is a 10.20 compatible library for applications which were built on 10.20
- `libGL.2` is the library which is for applications built on 11.x (this is why `libGL.sl` is a symbolic link to `libGL.2`)

There is only one version of libraries for 64-bit. These libraries are version numbered.

The other libraries you will see in these directories are all drivers for specific graphics devices. All `libdd`... are loaded automatically at execution time, contingent upon device type.

3 running OpenGL programs

This chapter gives a description of the Virtual GLX mode, Virtual Memory Driver (VMD), and support of threaded applications.

virtual GLX (VGL) mode

Virtual GLX (VGL) defines a special transparent mode within hp's implementation of OpenGL that allows an hp client to render through OpenGL to X servers and/or X terminals that do not support OpenGL or the X server extension for GLX.

This mode is implemented by emulating the X server extension within the OpenGL API client-side library and using the hp Virtual Memory Driver (VMD) to perform Xlib rendering.

VGL provides flexibility for OpenGL users, but does not provide the same level of performance as is available to servers supporting GLX.

visual support for the VGL mode

In VGL mode, the visual capabilities incorporated in `glXChooseVisual()` and `glXGetConfig()` are synthesized from the list of X Visuals supported on the target X Server and the capabilities of the Virtual Memory Driver (VMD). Table 1-5 in Chapter 1 lists the X Visuals that are supported through the OpenGL Extension to the X Window System (GLX) in the Virtual GLX (VGL) mode.

special considerations

When you are in the VGL mode, you will notice the following differences between it and the GLX mode.

- VGL deals with X servers that do not support replicated X visuals that provide extended GLX capabilities. This results in a GLX visual list that is synthesized from available X visuals. This list is assigned the maximum set of capabilities supported by the Virtual Memory Driver (VMD) for each particular visual. For example, if a visual is found to be supported by the Double-Buffered Extension (DBE), then it will be reported as having the capability of doing double-buffering. Note that there will not be a counterpart for the GLX visual with the same type and depth that is single buffered. Such visuals are locked to either single buffer or double buffer mode, based upon the first access. In VGL, to utilize one visual for both single and double buffered operation, two separate X display connections must be opened.

- OpenGL and Xlib rendering when mixed and sent to the same drawable in VGL mode may behave differently than if a GLX capable X server were used. This is because in VGL mode OpenGL rendering is not strictly bounded by the limits of primitives rendered as is the case when a GLX server is used. In fact, rendering a single GLX primitive can result in repainting the entire drawable. This means that in the VGL mode it may not be safe to rely upon the fact that Xlib and OpenGL render to different regions of the drawable. The best way to avoid this issue is to always perform Xlib rendering after OpenGL rendering.
- The `glReadPixels` routine when used in the VGL mode will return only pixel data rendered via OpenGL. Xlib rendering will not be included.
- Because of the way VMD works (rendering to a VM buffer and then displaying the images through X11 protocol), it will behave a bit differently than hardware devices. In particular, since VMD renders to VM buffers, changes to the X11 window will not appear until a buffer swap or a `glFlush`/`glFinish`.
- A call to `glXSwapBuffers` is the only approved way to achieve double buffering for VGL visuals. Note that calls made to `XdbeSwapBuffers` will not work correctly.
- A call can be made to:

  ```
  Bool hpglXDisplayIsVGL(Display *dpy, int screen)
  ```

  to determine if a particular display connection is operating in VGL mode. The return value is "True" if `dpy` is VGL; otherwise, the value returned is "False." This is an hp function that is not available on other implementations of OpenGL.

running hp's implementation of the OpenGL stereo application

Following are the steps required to run hp's implementation of OpenGL "stereo in a window" mode:

1. Find out if your monitor is currently configured in a mode that supports stereo. This can be done by running the command:

   ```
   export DISPLAY=myhost:x.y
   /opt/graphics/OpenGL/contrib/xglinfo/xglinfo
   ```

   The output from xglinfo lists the OpenGL capabilities of the specified X Display, and includes all GLX visuals that are supported. If one or more of the listed GLX visuals are marked as stereo capable, then you can proceed to step three.

2. If none of the GLX visuals support stereo, you will need to re-configure your monitor to a configuration that supports stereo. For the Visualize fx family of graphics cards this is a single step process of re-configuring. For the HP FireGL-UX and ATI FireGL X1/X3 this is a two step process.

   Note that you can use the "Monitor Configuration" component of SAM to re-configure you monitor, or you can execute the following command:

   ```
   /opt/graphics/common/bin/setmon graphics device
   ```

   Note that for Visualize fx cards, *graphics device* is a name such as "`/dev/crt`" that is included on the Screen line in the `/etc/X11/X*screens` file for the X Server that you want to configure for stereo. For FireGL cards graphics device is a name such as "/dev/gvid" that is included in the DeviceFile line of the /etc/X11/XF86Config file.

   The setmon command is interactive and will present you with the possible monitor configurations allowable for the specified device. You should select one of the configurations that is listed by setmon as stereo capable. If none of the configurations indicate stereo capability, then your graphics device cannot be used for OpenGL stereo rendering.

After successfully re-configuring your monitor, the X Server will be restarted.

If you are configuring a Visualize fx display proceed to you can verify the availability of GLX stereo visuals now by running the xglinfo command again.

If you are configuring a FireGL graphics device an additional step is required. Run "sam" to configure stereo mode in the /etc/X11/XF86Config file. Select the "Display" area and the "X Server Configuration" then select the display device icon. Using the Actions menu select "Modify Screen Options". This dialog box will the allow you to select and set to "true" the "Stereo" option (for FireGL X1/X3) or "Qbs" (e.g. Quad-buffer stereo) option (for the FireGL-UX) . Once this option is set and the configuration is saved the X server can be restarted and will be in Stereo mode.

3. To select one of the stereo capable GLX visuals through OpenGL, the `GLX_STEREO` enumerated type should be passed to either `glXChooseVisual()` or `glXGetConfig()`. Once a stereo visual has been selected, it can be used to create a stereo window, and `glDrawBuffer()` can then be called to utilize both the right and left buffers for rendering stereo images.

4 compiling and linking programs

This chapter provides information for including header files in your program, linking shared libraries, compiling 32-bit and 64-bit applications for OpenGL and OpenGL procedure calls.

overview

Table 4-1 contains a list of the subdirectories in the directory

```
/opt/graphics/OpenGL
```

These subdirectories contain header files and libraries which may be used when compiling and linking your programs.

Table 4-1

Subdirectory	**This Directory Contains…**
`include/GL`	Header files needed for OpenGL development.
`lib`	32-bit run-time shared libraries.
`lib/pa20_64`	64-bit run-time shared libraries.
`lbin`	Run-time executables.
`doc`	OpenGL documentation including reference pages.
`contrib/libwidget`	A Motif widget library and source code.
`contrib/libglut`	Utilities found in the OpenGL Utility Toolkit as mentioned in the OpenGL Programming for the X Window System manual.
`contrib/xglinfo`	Utility to print display and visual information for OpenGL with the X Window system.
`demos`	Sample OpenGL programs, including source code.

including header files

Most OpenGL programs and applications that only use the standard OpenGL data types, definitions, and function declarations, need only include the header file gl.h under the `/opt/graphics/OpenGL/include/GL` directory. Use the following syntax:

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

Still other header files may be needed by your program, depending on your application. For example, in order to use the OpenGL extension to X Windows (GLX) you must include `glx.h`, as shown below.

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

Instructions for including various additional header files are usually provided with the `README` file that accompanies a utility or function. The `README` also includes instructions for using or operating the utilities.

Your header file declarations at the beginning of your program should look similar to this:

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <GL/gl.h>
#include <GL/glx.h>
```

linking shared libraries

OpenGL is supported on workstations using shared libraries that must be linked with the application program.

When you compile your OpenGL programs, you must link the application with the OpenGL library `libGL`. Notice that the OpenGL library is dependent on the hp X extensions library (`libXext`).

An ANSI C compile line will typically look similar to this:

```
cc -g -Aa -D_hpUX_SOURCE -z \
-I/opt/graphics/OpenGL/include \
-I/usr/include/X11R6 -o cube.32 cube.c \
-L/opt/graphics/OpenGL/lib \
-L/usr/lib/X11R6 -ldld -lGLU -lGL \
 -lXhp11 -lXext -lX11 -lm
```

To compile your application using ANSI C, you can also use the `cc` command with either of the command line options `+Aa` or `+Ae`.

If you are going to compile your application using HP's ANSI C++ compiler, use the `aCC` compiler.

See the *HP Graphics Administration Guide* for more information on compiling.

This table summarizes the shared libraries and X11 directories that are linked on the command line example above.

Table 4-2 **Shared Libraries**

Library	**Description**
`libGL`	OpenGL routines
`libX11`	X11 routines
`libXext`	HP X11 extensions

compiling 32-bit and 64-bit applications for OpenGL

The following sample compile and link lines may help you to build your application once it has been ported to take advantage of 64-bit capabilities.

Sample 32-bit compile and link:

```
cc -g -Aa -D_<ABBR>hp</ABBR>UX_SOURCE -z \
-I/opt/graphics/OpenGL/include \
-I/usr/include/X11R6 -o cube.32 cube.c \
-L/opt/graphics/OpenGL/lib \
-L/usr/lib/X11R6 -ldld -lGLU -lGL \
-lXhp11 -lXext -lX11 -lm
```

Sample 64-bit compile and link (for 11.x only):

```
cc -g -Aa +DA2.0W -D_hpUX_SOURCE -z \
-I/opt/graphics/OpenGL/include \
-I/usr/include/X11R6 -o cube.64 cube.c \
-L/opt/graphics/OpenGL/lib/pa20_64 \
-L/usr/lib/X11R6/pa20_64 -L/usr/lib/pa20_64 \
-L/usr/lib -ldld \
-lGLU -lGL -lXhp11 -lXext -lX11 -lm
```

OpenGL procedure calls

In order to facilitate maximum performance, the OpenGL library uses a unique procedure calling convention. This convention is supported only by the HP C and C++ compilers.

If you get a large number of “`Undefined pragma`” messages (for example, `Undefined pragma ìhp_PLT_CALLî ignored`) when compiling an OpenGL application, you are most likely using a compiler that does not support this calling convention. To get an appropriate HP C or C++ compiler, you will need to contact your local HP Sales Representative.

You must also include the gl.h header file supplied with HP’s implementation of OpenGL in any source code that makes OpenGL calls. If you have unresolved OpenGL symbols (for example, “`Unsatisfied symbol glVertex3f`”) when linking your application, make sure that the correct `gl.h` file is being included in all your source files. Any `gl.h` files from other vendors or other sources will not work with HP’s implementation of OpenGL.

5 programming hints

The topics covered in this chapter are intended to give you some helpful programming hints as you begin to develop your OpenGL applications. Note that these hints are specific to hp's implementation of OpenGL. For further information on OpenGL programming hints that are not hp specific, see Appendix G in the *OpenGL Programming Guide* and section 6.6 “Maximizing OpenGL Performance” in the *OpenGL Programming for the X Window System* manual.

The programming hints in this chapter are covered in these sections:

- OpenGL Correctness Hints
- OpenGL Performance Hints

OpenGL correctness hints

Hints provided in this section are intended to help you correctly use HP's implementation of OpenGL.

4D values

When specifying 4D values, such as vertices, light positions, etc, if possible supply a w value that is not near the floating point limits of `MINFLOAT` or `MAXFLOAT`. Using w values near the floating point limits increases the likelihood of floating point precision errors in calculations such as lighting, transformations, and perspective division.

Also, performance will be best when 4D positions are normalized such that w is 1.0.

For best accuracy and performance, if you want to specify some 4D position like (0.0, 0.0, 5e10, 1.5e38), instead use the equivalent normalized position (0.0, 0.0, 3.33e-28, 1.0).

On HP Visualize fx devices only, if a light position must be specified with a w value that is near the floating point limits, consider setting

```
HPOGL_LIGHTING_SPACE=EC
```

to ensure that lighting occurs in Eye Space. This will eliminate an extra transformation of the light position, giving the best possible solution.

texture coordinates

When using non-orthographic projection, keep in mind the texture coordinates will be divided by w as an intermediate calculation. HP's implementation of OpenGL estimates that for VMD, the texture coordinates used in perspective projections will have only five significant digits of precision. Therefore, when you have texturing close to a window edge and the decomposition of the primitive causes the vertices to have very closely-spaced texture coordinates after perspective projection, you may see loss of texturing precision. This loss of precision may make the texture primitive seem locally smeared.

OpenGL performance hints

Hints provided in this section are intended to help improve your applications performance when using HP's implementation of OpenGL.

display list performance

The topics covered here are areas where you can gain substantial improvements in program performance when using OpenGL display lists. Here is a list of the topics that are covered:

- geometric primitives
- GL_COMPILE_AND_EXECUTE mode
- textures
- state changes and their effects on display lists
- regular primitive data.

geometric primitives

Geometric primitives will typically be faster in a display list than by using immediate mode. Each display list should have numerous primitives to ensure good performance. As a general rule, larger primitives will be faster than smaller ones. Performance gains here can be dramatic. For example, it is possible that a single `GL_TRIANGLES` primitive with 20 or so triangles will render three times faster than 20 `GL_TRIANGLES` primitives with a single triangle in each one.

GL_COMPILE_AND_EXECUTE mode

Due to the pre-processing of the display list, and execution performance enhancements, creating a display list using the `GL_COMPILE_AND_EXECUTE` mode will reduce program performance. If you need to improve your programs performance, do not create a display list using the `GL_COMPILE_AND_EXECUTE` mode. You will find that it is easier and faster to create the display list using the `GL_COMPILE` mode, and then execute the list after it is created.

draw array set extensions

`glDrawArraySethp` is a Hewlett-Packard OpenGL 1.1 extension to vertex arrays which provides a high-speed mechanism for rendering multiple primitives. Use of `glDrawArraySethp` will be easy for applications which currently store geometry in vertex arrays and use multiple calls to `glDrawArrays` for rendering primitives from the arrays.

`glDrawArraySethp` is especially useful when multiple connected primitives, such as `GL_LINE_STRIP`, `GL_TRIANGLE_STRIP`, etc., are consecutively drawn from a vertex array. However all OpenGL primitive types are supported.

Since only `glDrawArray` calls are made while rendering the vertex array set, primitive attributes, such as material colors, must be established for the entire array set or changed per vertex. If OpenGL library calls other than `glDrawArrays` are required during rendering the set to properly draw the array set, then `glDrawArraySethp` is not appropriate.

benefits of glDrawArraySethp: `glDrawArrays` and `glDrawArraySethp` provide basically the same programmatic benefits, that is reduced function calls and less user code. Note that `glDrawArraySethp`'s major benefit is performance. `glDrawArraySethp` provides from 10%-55% performance advantage over using `glDrawArrays` alone.

The amount of performance benefit depends upon several factors, including the number of primitives in the set, the length of the primitives in the set, and maximum rendering speed of the graphics device.

To achieve optimum `glDrawArraySethp` rendering performance, group as many primitives in each set as possible.

using glDrawArraySethp: To use `glDrawArraySethp`, the current vertex array must be set and enabled. This is done using `glNormalPointer`, `glVertexPointer`, `glEnableClientState`, etc., or `glInterleavedArrays`. After the vertex array is established and enabled, `glDrawArraySethp` may be used.

The C declaration of `glDrawArraySethp` is:

```
void glDrawArraySethp(GLenum mode,
 const GLint* list, GLsizei count);
```

`where:`

mode	Specifies the primitive or primitives that will be created from the vertices. Ten symbolic constants are accepted: GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON.
list	A sequence of starting indices in the enabled arrays. Each index is the start of a primitive in the set. The final value in the sequence indexes the end of the set. The total number of indices is count+1.
count	The number of primitives in the set to render.

When glDrawArraySethp is called, it iterates over count + 1 vertex array indices from list. For 0 HP uses list[i+1] - list[i] sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element list[i].

Calling glDrawArraySethp(mode, list, count) is functionally equivalent to:

```
for (i = 0; i < count; i++)
     glDrawArrays(mode, list[i], list[i+1]- list[i]);
```

textures

If calls to glTexImage are put into a display list, they may be cached. In general, for best performance, use Array Sets and Arrays, Display, and Immediate Mode (Vertex API). Note that if you are going to use the same texture multiple times, you may gain better performance if you put the texture in a display list. Another solution would be to use texture objects. Since 3D textures can potentially become very large, they are not cached.

state changes and their effects on display lists

If there are several state changes in a row, it is possible, in some circumstances, for the display list to optimize them.

It is more efficient to put a state change before a glBegin, than after it. For example, this is always more efficient:

```
glColor3f(1,2,3);
glBegin(GL_TRIANGLES);
glVertex3f(...);
```

```
... many more vertices...
glEnd();
```

than this:

```
glBegin(GL_TRIANGLES);
glColor3f(1,2,3);
glVertex3f(...);
... many more vertices...
glEnd();
```

For performance efficiency avoid `glMaterial` state changes, especially within a `glBegin`/`glEnd` pair.

regular primitive data

If the vertex data that you give to a display list is regular (that is, every vertex has the same data associated with it), it is possible for the display list to optimize the primitive much more effectively than if the data is not regular.

For example if you wanted to give only a single normal for each face in a `GL_TRIANGLES` primitive, the most intuitive way to get the best performance would look like this:

```
glBegin(GL_TRIANGLES);
glNormal3fv(&v1);
glVertex3fv(&p1); glVertex3fv(&p2);  glVertex3fv(&p3);
glNormal3fv(&v2);
glVertex3fv(&p4); glVertex3fv(&p5); glVertex3fv(&p6);
...
glEnd();
```

In immediate mode, this would give you the best performance. However, if you are putting these calls into a display list, you will get much better performance by duplicating the normal for each vertex, thereby giving regular data to the display list:

```
glBegin(GL_TRIANGLES);
glNormal3fv(&v1); glVertex3fv(&p1);
glNormal3fv(&v1); glVertex3fv(&p2);
glNormal3fv(&v1); glVertex3fv(&p3);
glNormal3fv(&v2); glVertex3fv(&p4);
```

```
glNormal3fv(&v2); glVertex3fv(&p5);
glNormal3fv(&v2); glVertex3fv(&p6);
...
glEnd();
```

The reason this is faster is the display list can optimize this type of primitive into a single, very efficient structure. The small cost of adding extra data is offset by this optimization.

Performance is increased by maximizing the number of vertices per Begin/End pair. If your vertex data in memory is organized in a linear, rather than a random manner, performance is enhanced by taking advantage of vertex pre-fetch. It is most efficient to use 32-bit float data, which avoids the need to convert data.

texture downloading performance

This section includes some helpful hints for improving the performance of your program when downloading textures.

- If you are downloading MIP maps, always begin with the base level (level 0) first.
- If it is possible, you should use texture objects to store and bind textures.
- If you are doing dynamic downloading of texture maps, you will get better performance by replacing the current texture with a texture of the same width, height, border size, and format. This should be done instead of deleting the old texture and creating a new one.

selection performance

To increase the performance of selection (`glRenderMode GL_SELECTION`) it is recommended that the following capabilities be disabled before entering the selection mode.

```
GL_TEXTURE_*
GL_TEXTURE_GEN_*
GL_FOG
GL_LIGHTING
```

state change

OpenGL state setting commands can be classified into two different categories. The first category is **vertex-data commands**. These are the calls that can occur between a `glBegin`/`glEnd` pair:

```
glVertex
glColor
glIndex
glNormal
glEdgeFlag
glMaterial
glTexCoord
```

The processing of these calls is very fast. Restructuring a program to eliminate some vertex data commands will not significantly improve performance.

The second category is **modal state-setting** commands, or sometimes referred to as "mode changes." These are the commands that:

- Turn on/off capabilities,
- Change attribute settings for capabilities,
- Define lights,
- Change matrices,
- etc.

These calls cannot occur between a `glBegin`/`glEnd` pair. Examples of such commands are:

```
glEnable(GL_LIGHTING);
glFogf(GL_FOG_MODE, GL_LINEAR);
glLightf(..);
glLoadMatrixf(..);
```

Changes to the modal state are significantly more expensive to process than simple vertex-data commands. Also, application performance can be optimized by grouping modal-state changes, and by minimizing the number of modal-state changes:

- Grouping your state changes together (that is, several modal state changes at one time), and then rendering primitives, will provide better performance than doing the modal state changes one by one and intermixing them with primitives.
- Grouping primitives that require the same modal state together to minimize modal state changes. For example, if only part of a scene's primitives are lighted, draw all the lighted primitives, then turn off lighting and draw all the unlighted primitives, rather than enabling/disabling lighting many times.

Some states negatively impact performance, such as two-sided lighting, polygon mode `GL_LINE`, and wide lines.

optimization of lighting

HP's implementation of OpenGL optimizes the lighting case such that the performance degradation from one light to two or more lights is linear. Lighting performance does not degrade noticeably when you enable a second light. In addition, the `GL_SHININESS` material parameter is not particularly expensive to change.

occlusion culling

The proper use of HP's occlusion culling extension can dramatically improve rendering performance. This extension defines a mechanism for determining the non-visibility of complex geometry based on the non-visibility of a bounding geometry. This feature can greatly reduce the amount of geometry processing and rendering required by an application, thereby, increasing the applications performance. For more information on occlusion culling, see the section "Occlusion Extension" found in Chapter 1.

high frame rate applications

To achieve maximum performance for buffer swap and clear operations across the entire family of Visualize and OEM graphics devices, avoid the following:

- Scissor rectangle smaller than the window size
- Non-trivial plane masks, i.e., any value other than all zeros or all ones

- Single buffered applications
- Alpha planes
- Depth 8 visuals
- Stereo
- Gradient backgrounds

rescaling normals

When normal rescaling is enabled, a new operation is added to the transformation of the normal vector into eye coordinates. The normal vector is rescaled after it is multiplied by the inverse modelview matrix and before it is normalized.

The rescale factor is chosen so that in many cases, normal vectors with unit length in object coordinates will not need to be normalized as they are transformed into eye coordinates.

As of Release 1.05 of HP's implementation of OpenGL 1.1, the `GL_RESCALE_NORMAL_EXT` token is supported. It is accepted by the `<cap>` parameter of `glEnable`, `glDisable`, and `glIsEnabled`, and by the `<pname>` parameter of `glGetBooleanv`, `glGetIntegerv`, `glGetFloatv`, and `glGetDoublev`.

Normals that have unit length when sent to the GL, have their length changed by the inverse of the scaling factor after transformation by the model-view inverse matrix when the model-view matrix represents a uniform scale. If rescaling is enabled, then normals specified with the Normal command are rescaled after transformation by the Modelview Inverse.

Normals sent to the GL may or may not have unit length. In addition, the length of the normals after transformation might be altered due to transformation by the model-view inverse matrix. If normalization is enabled, then normals specified with the `glNormal3` command are normalized after transformation by the model-view inverse matrix and after rescaling if rescaling is enabled. Normalization and rescaling are controlled with `glEnable` and `glDisable` with the target equal to `NORMALIZE` or `RESCALE_NORMAL`. This requires two bits of state. The initial state is for normals not to be normalized or rescaled.

Therefore, if the modelview matrix is `M`, the transformed plane equation is:

$$\begin{pmatrix} n'_x & n'_y & n'_z & q \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z & q \end{pmatrix} \times M^{-1}$$

the rescaled normal is

$$\begin{pmatrix} n''_x & n''_y & n''_z \end{pmatrix} = f \times \begin{pmatrix} n'_x & n'_y & n'_z \end{pmatrix}$$

and the fully transformed normal is

$$\frac{1}{\sqrt{(n''_x)^2+(n''_y)^2+(n''_z)^2}} \begin{pmatrix} n''_x \\ n''_y \\ n''_z \end{pmatrix}$$

If rescaling is disabled, f is 1, otherwise f is computed as follows:

Let m_{ij} denote the matrix element in row *i* and column *j* of M^{-1}, numbering the topmost row of the matrix as row 1, and the left most column as column 1. Then

$$f = \frac{1}{\sqrt{(m_{31})^2+(m_{32})^2+(m_{33})^2}}$$

Alternatively, an implementation may chose to normalize the normal instead of rescaling the normal. Then

$$f = \frac{1}{\sqrt{(n'_x)^2+(n'_y)^2+(n'_z)^2}}$$

If normalization is disabled, then the square root in equation 2.1 is replaced with 1; otherwise, it is calculated as dictated by the OpenGL Spec. If both normalize and rescale are enabled, HP's implementation skips the rescale and does only the normalize.