HP's Implementation of OpenGL

HP 9000 Workstations

HEWLETT
PACKARD

HP Part No. B6196-90001
Printed in USA E0797

Edition 1

FINAL TRIM SIZE : 7.5 in x 9.0 in

Notices

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.

July 1997 . . . Edition 1. This manual is valid for the July 1997 Workstation ACE for HP-UX 10.20 on all HP 9000 workstations.

0

Chapter 0: Preface

Document Conventions

Below is a list of the typographical conventions used in this document:

`mknod /usr/include`	Verbatim computer literals are in computer font. Text in this style is letter-for-letter verbatim and, depending on the context, should be typed in exactly as specified, or is named exactly as specified.
In *every* case ...	Emphasized words are in italic type.
... device is a **freen** ...	New terms being introduced are in bold-faced type.
... the ⟨*device_id*⟩ ...	Conceptual values are in italic type, enclosed in angle brackets. These items are not verbatim values, but are descriptors of the *type* of item it is, and the user should replace the conceptual item with whatever value is appropriate for the context.

Contents

Figures

Tables

1

Overview of OpenGL

Introduction

OpenGL is a hardware-independent Application Programming Interface (API) that provides an interface to graphics operations. It is HP's implementation of OpenGL that converts API commands to graphical images via hardware and/or software functionality. The 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, here is a list of manuals that are published by Addison-Wesley and shipped with HP's implementation of OpenGL.

- *OpenGL Programming Guide* teaches you how to program in OpenGL.
- *OpenGL Reference Manual* is a reference that describes OpenGL functions.
- *OpenGL Programming for the X Window System* teaches you how to use 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

HP's implementation of OpenGL provides the following libraries:

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

The OpenGL product does not support archived libraries.

Supported Graphics Devices

This section covers the graphics devices and visuals that are supported by the OpenGL product. Here is a list to the graphics devices that are supported:

- HP VISUALIZE-FX^2
- HP VISUALIZE-FX^4
- HP VISUALIZE-FX^6

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-FX^2

X Visual Information			OpenGL GLX Information														
Class	Depth	Color Map Size	Bfr Size	Ovrly=1 or Img=0	RGBA=1 or Idx=0	Dbl Bfr	# Aux Bfrs	Color Buffer				Z	Stencil	Accum. Buffer			
								R	G	B	A			R	G	B	A
PseudoColor	8	255	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
PseudoColor	12[1]	4096	12	0	0	0	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	12[1]	4096	12	0	0	1	0	0	0	0	0	24	4	0	0	0	0
TrueColor	12	16	12	0	1	1	0	4	4	4	0	24	4	16	16	16	0
TrueColor	24	256	24	0	1	0	0	8	8	8	0	24	4	16	16	16	0

1. The 12-bit PseudoColor visuals are not present by default. They can be enabled by invoking the "X Server Configuration" component under SAM, or by manually adding the enable 12-bit PseudoColor visual option to your `/etc/X11/X*Screens` file as documented in the *Graphics Administration Guide*.

Table 1-2. Visual Table for HP VISUALIZE-FX^4

X Visual Information			OpenGL GLX Information														
Class	Depth	Color Map Size	Bfr Size	Ovrly=1 or Img=0	RGBA=1 or Idx=0	Dbl Bfr	# Aux Bfrs	Color Buffer				Z	Stencil	Accum. Buffer			
								R	G	B	A			R	G	B	A
PseudoColor	8	255	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
PseudoColor	12[1]	4096	12	0	0	0	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	12[1]	4096	12	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

1. The 12-bit PseudoColor visuals are not present by default. They can be enabled by invoking the "X Server Configuration" component under SAM, or by manually adding the enable 12-bit PseudoColor visual option to your `/etc/X11/X*Screens` file as documented in the *Graphics Administration Guide*.

Table 1-3. Visual Table for HP VISUALIZE-FX^6

X Visual Information			OpenGL GLX Information														
Class	Depth	Color Map Size	Bfr Size	Ovrly=1 or Img=0	RGBA=1 or Idx=0	Dbl Bfr	# Aux Bfrs	Color Buffer				Z	Sten cil	Accum. Buffer			
								R	G	B	A			R	G	B	A
PseudoColor	8	255	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
PseudoColor	12[1]	4096	12	0	0	0	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	12[1]	4096	12	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

1. The 12-bit PseudoColor visuals are not present by default. They can be enabled by invoking the "X Server Configuration" component under SAM, or by manually adding the enable 12-bit PseudoColor visual option to your `/etc/X11/X*Screens` file as documented in the *Graphics Administration Guide*.

Stereo Visual Support for VISUALIZE-FX^4 and VISUALIZE-FX^6

When a monitor is configured in a stereo capable mode, HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 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 1-4. Stereo Visual Support for HP VISUALIZE-FX ^{4} and HP VISUALIZE-FX ^{6}

X Visual Information			OpenGL GLX Information															
Class	Depth	Color Map Size	Bfr Size	Ovrly=1 or Img=0	RGBA=1 or Idx=0	Dbl Bfr	Stereo	# Aux Bfrs	Color Buffer				Z	Stencil	Accum. Buffer			
									R	G	B	A			R	G	B	A
PseudoColor	8	256	8	0	0	0	1	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	8	256	8	0	0	1	1	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	12	4096	12	0	0	0	1	0	0	0	0	0	24	4	0	0	0	0
PseudoColor	12	4096	12	0	0	1	1	0	0	0	0	0	24	7	0	0	0	0
TrueColor	12	16	12	0	1	1	1	0	4	4	4	0	24	4	16	16	16	0
TrueColor	12	16	12	0	1	0	1	0	4	4	4	[1]	24	4	16	16	16	[1]

1. Alpha planes are only available on the HP Visualize-FX ^{6}.

Visual Support for Other Graphics Devices

The OpenGL product can be used with the VISUALIZE-FX family of devices as well as the VISUALIZE-EG device 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 1-5. Visuals Table for VMD

X Visual Information			OpenGL GLX Information															
Class	Depth	Color Map Size	Bfr Size	Ovrly=1 or Img=0	RGBA=1 or Idx=0	Dbl Bfr	# Aux Bfrs	Color Buffer				Z[3]	Stencil[3]	Accum. Buffer			
								R	G	B	A			R	G	B	A
PseudoColor	4	16	4	0	0	[1]	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
PseudoColor	8	255	8	1	0	[1]	0	0	0	0	0	0	0	0	0	0	0
TrueColor	8	256	8	0	1	[1]	0	3	3	2	0	24	4	16	16	16	0
PseudoColor	12	4096	12	0	0	[1]	0	0	0	0	0	24	4	0	0	0	0
TrueColor	12	16	12	0	1	[1]	0	4	4	4	[2]	24	4	16	16	16	16
DirectColor	12	16	12	0	1	[1]	0	4	4	4	[2]	24	4	16	16	16	16
TrueColor	24	256	24	0	1	[1]	0	8	8	8	[2]	24	4	16	16	16	16
DirectColor	24	256	24	0	1	[1]	0	8	8	8	[2]	24	4	16	16	16	16

1. Double buffering is set to True (1) if the X visual supports the X Double Buffering Extension (DBE).
2. Alpha will only work correctly on 12- and 24-bit TrueColor and DirectColor visuals when the X server does not use the high order nybble/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.
3. Depth and stencil buffers are only allocated for image plane visuals.

Buffer Sharing between Multiple Processes

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^4 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 1-5 in 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.

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`.

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, read 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.

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, read 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.

For details on this feature, read the section "Gamma Correction" found in Chapter 7 of the *Graphics Administration Guide*.

OpenGL Extensions

The extensions listed in this section are in addition to those described in the *OpenGL Programming Guide*, *OpenGL Reference Manual*, and *OpenGL Programming for the X Window System*.

Clamp Border and Clamp Edge Extensions

Texture clamp extensions provide techniques to either clamp to the edge texels or border texels even when using a filter that uses linear filtering. Nearest filtering always just selects one of the texels from the texture map. But when using linear filtering, whether from the minification or magnification filters, the filtered texels can be an average of texels from both the texture map and the texture border. To only use the texture map texels when clamping, use the clamp edge extension. To allow the selection of border texels when clamping, use the clamp border extension.

Table 1-6. Clamp Border and Clamp Edge Extensions

Extended Area	Enumerated Types	Description
Wrap Modes	`GL_CLAMP_TO_BORDER_EXT` default: `GL_REPEAT`	When this enumerated type is passed into `glTexParameter`, it will clamp to the border of the mip level.
Wrap Modes	`GL_CLAMP_TO_EDGE_EXT` default: `GL_REPEAT`	When this enumerated type is passed into `glTexParameter`, it will clamp to the edge of the mip level.

To use clamp border extension, substitute `GL_CLAMP_TO_BORDER_EXT` for the `param` in `glTexParameter`. To use clamp edge extension, substitute `GL_CLAMP_TO_EDGE_EXT` for the `param` in `glTexParameter`.

Code fragments and results:

```
float BorderColor[4];
BorderColor[0] = 0.0; /* Red */
BorderColor[1] = 0.0; /* Green */
BorderColor[2] = 1.0; /* Blue */
BorderColor[3] = 1.0; /* Alpha */
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, BorderColor);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

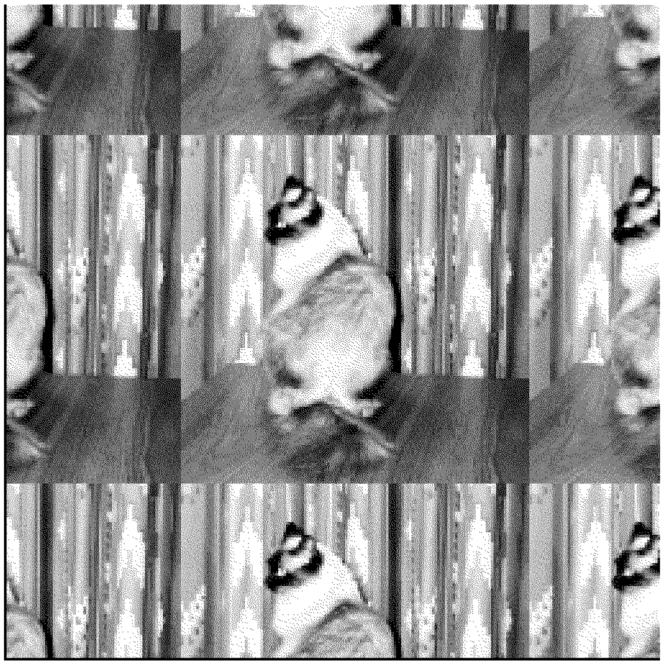


Figure 1-1. Repeat Wrap Mode

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

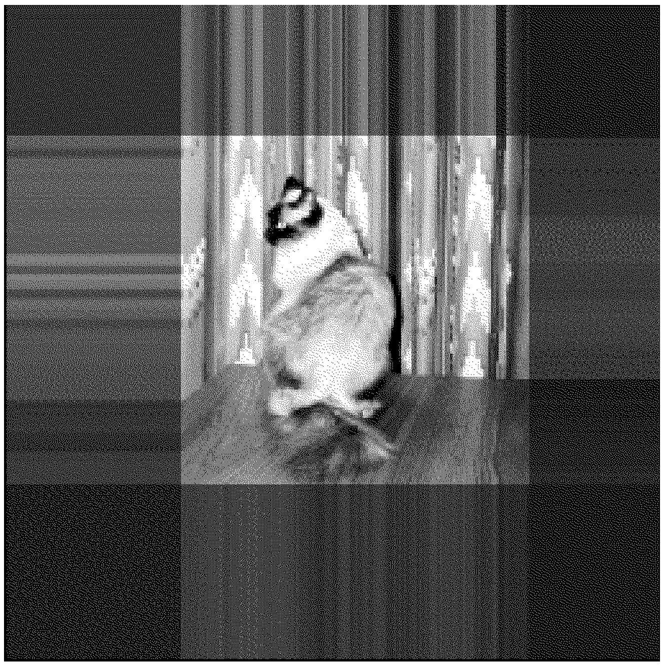


Figure 1-2. Clamp Wrap Mode

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE_EXT);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE_EXT);
```

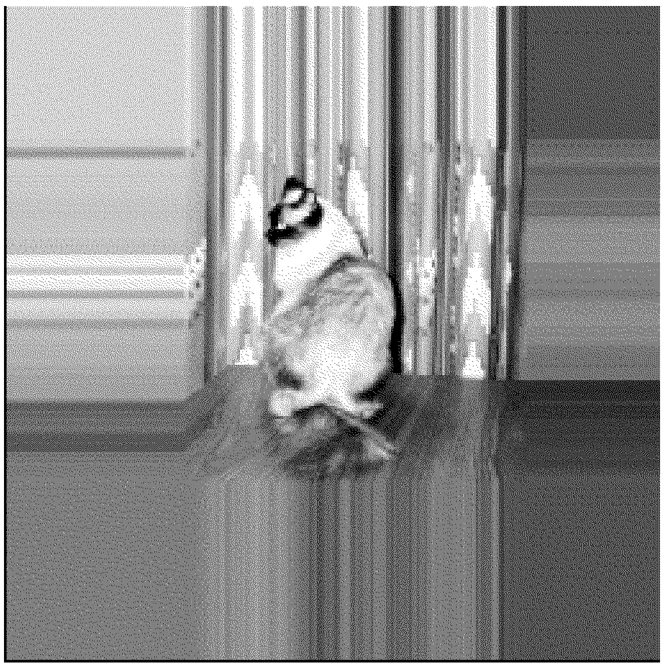


Figure 1-3. Clamp to Edge Wrap Mode

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER_EXT);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER_EXT);
```

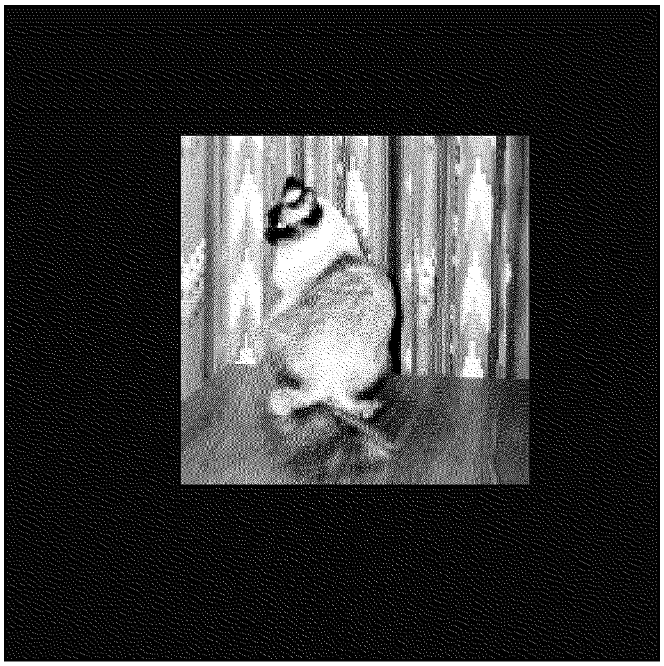


Figure 1-4. Clamp to Border Wrap Mode

For related information, see the function `glTexParameter`.

3D Texture Extension

The 3D texture extension is useful for volumetric rendering of solid surfaces such as a marble vase or for rendering images where geometric alignment is important, such as an MRI medical image. For this extension, the texture maps have width and height as they did for 2D and also an additional depth dimension not included in 2D. The third coordinate forms a right handed coordinate system which is illustrated in Figure 1-5.

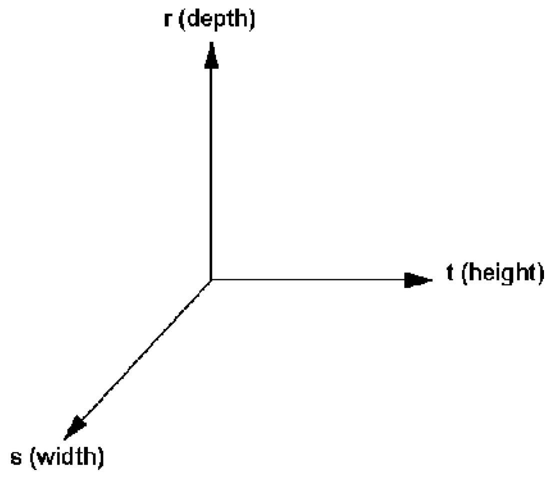


Figure 1-5. Right Handed Coordinate System for 3D Texturing

Each mipmap level consists of a block of data, see Figure 1-6. Each mipmap level of the texture map is treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a 2-dimensional image. So each mip level is a $(2^m + 2b) \times (2^n + 2b) \times (2^l + 2b)$ block where b is a border width of either 0 or 1, and m, n and l are non-negative integers.

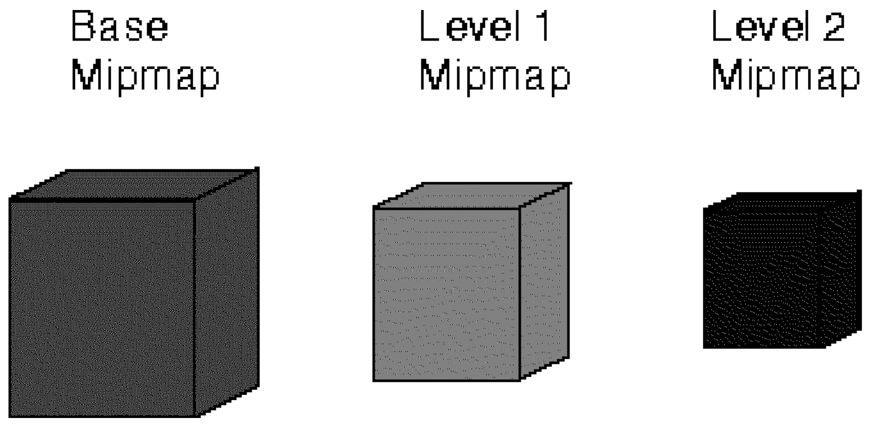


Figure 1-6. Each Mipmap is a Block in 3D Texturing

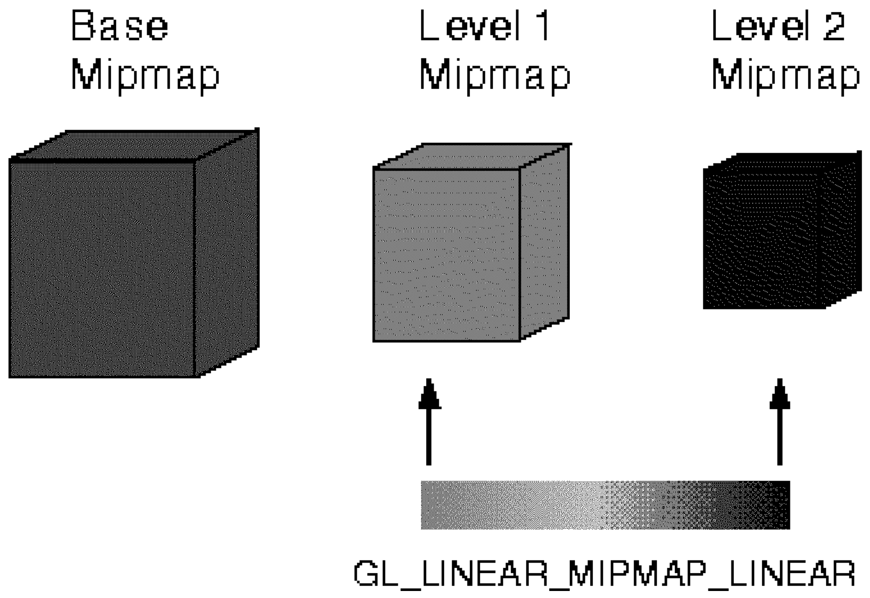


Figure 1-7. GL_LINEAR_MIPMAP_LINEAR Filtering May Use Two Blocks

Table 1-7. Enumerated Types for 3D Texturing

Extended Area	Enumerated Types	Description
Pixel Storage	`GL_[UN]PACK_IMAGE_HEIGHT_EXT` default: 0 for each	The height of the image from which the texture is created; it supercedes the value of the height passed into `glTexImage3DEXT`.
Pixel Storage	`GL_[UN]PACK_SKIP_IMAGES_EXT` default: 0 for each	The initial skip of contiguous rectangles of the texture.
Texture Wrap Modes	`GL_TEXTURE_WRAP_R_EXT` default: `GL_REPEAT`	The wrap mode applied to the `r` texture coordinate
Enable/ Disable	`GL_TEXTURE_3D_EXT` default: Disabled	The method to enable/disable 3D texturing.
Get Formats	`GL_MAX_3D_TEXTURE_SIZE_EXT`, `GL_TEXTURE_BINDING_3D_EXT` default: N/A	The maximum size of the 3D texture allowed; bind query.
Proxy	`GL_PROXY_TEXTURE_3D_EXT` default: N/A	The proxy texture that can be used to query the configurations.

Steps for 3D Texturing Programming

To use the 3D texture extension (see sample program below), do the following steps.

1. Enable the 3D texture extension using `glEnable(GL_TEXTURE_3D_EXT)`,
2. Create a 3D texture using `glTexImage3DEXT`
3. Specify or generate the `s`, `t` and `r` texture coordinates using `glTexGen` or `glTexCoord3*`
4. Specify other parameters such as filters just as you would for 2D texturing, but use `GL_TEXTURE_3D_EXT` for the target.

3D Texture Program Fragments

This program draws four layers in the base mipmap level, and a diagonal slice through the base mipmap level.

```
/* Allocate texture levels separately, then concat to get a 3D texture. */

GLubyte texture1[TEXTURE_WIDTH][TEXTURE_HEIGHT][4];
GLubyte texture2[TEXTURE_WIDTH][TEXTURE_HEIGHT][4];
GLubyte texture3[TEXTURE_WIDTH][TEXTURE_HEIGHT][4];
GLubyte texture4[TEXTURE_WIDTH][TEXTURE_HEIGHT][4];
GLubyte textureConcat[TEXTURE_DEPTH][TEXTURE_WIDTH][TEXTURE_HEIGHT][4];

/* The checkerPattern procedure fills a texture with width, height with
   a period of Checker_period alternating between firstColor and
   secondColor, Texture should be declared prior to calling checkerPattern. */

static void checkerPattern(int width, int height, GLubyte *firstColor,
                           GLubyte *secondColor, GLubyte *texture,
                           int Checker_period) {

int texelX, texelY;
int index, fromIndex;
GLubyte *p = texture;
index = 0;

for (texelY = 0; texelY < height; texelY++) {
    for (texelX = 0; texelX < width; texelX++) {
        if (((texelX/Checker_period) % 2) ^ ((texelY/Checker_period) % 2)) {
            *p++ = firstColor[0]; /* red */
            *p++ = firstColor[1]; /* green */
            *p++ = firstColor[2]; /* blue */
            *p++ = firstColor[3]; /* alpha */
        } else {
            *p++ = secondColor[0]; /* red */
            *p++ = secondColor[1]; /* green */
            *p++ = secondColor[2]; /* blue */
            *p++ = secondColor[3]; /* alpha */
        }
    }
}
```

```
GLubyte blackRGBA[] = {0.0, 0.0, 0.0, 255.0};
GLubyte whiteRGBA[] = {255.0, 255.0, 255.0, 255.0};
GLubyte redRGBA[] = {255.0, 0.0, 0.0, 255.0};
GLubyte greenRGBA[] = {0.0, 255.0, 0.0, 255.0};
GLubyte blueRGBA[] = {0.0, 0.0, 255.0, 255.0};
GLubyte yellowRGBA[]= {255.0, 255.0, 0.0, 255.0};
GLubyte purpleRGBA[]= {255.0, 0.0, 255.0, 255.0};
GLubyte cyanRGBA[]= {0.0, 255.0, 255.0, 255.0};
GLubyte greyRGBA[] = {125.0, 125.0, 125.0, 255.0};

main (int argc, char *argv[]) {
/* Open window for displaying */
  Put your favorite code here to open an window and perform perspective setup

glEnable(GL_TEXTURE_3D_EXT);
checkerPattern( TEXTURE_WIDTH, TEXTURE_HEIGHT, blueRGBA, whiteRGBA,
  &texture1[0][0][0], 4);
checkerPattern( TEXTURE_WIDTH, TEXTURE_HEIGHT, redRGBA, yellowRGBA,
  &texture2[0][0][0], 4);
checkerPattern( TEXTURE_WIDTH, TEXTURE_HEIGHT, greenRGBA, blackRGBA,
  &texture3[0][0][0], 4);
checkerPattern( TEXTURE_WIDTH, TEXTURE_HEIGHT, purpleRGBA, cyanRGBA,
  &texture4[0][0][0], 4);
/* create a 3D texture, textureConcat, which has a different checker
   pattern at each depth */
memcpy(&textureConcat[0][0][0], texture1, sizeof(texture1));
memcpy(&textureConcat[1][0][0], texture2, sizeof(texture2));
memcpy(&textureConcat[2][0][0], texture3, sizeof(texture3));
memcpy(&textureConcat[3][0][0], texture4, sizeof(texture4));

glTexParameterf(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexParameterf(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R_EXT, GL_CLAMP);
glTexParameterf(GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage3DEXT(GL_TEXTURE_3D_EXT, 0, GL_RGBA, TEXTURE_WIDTH, TEXTURE_HEIGHT,
  TEXTURE_DEPTH, 0, GL_RGBA, GL_UNSIGNED_BYTE, &textureConcat);
/* Fill a quad with depth of r = 0.125, passed into glTexCoord for every
   vertex. */
   Add your quad code here

/* Fill a quad with depth of r = 0.375, passed into glTexCoord for
   every vertex. */ Add your quad code here

/* Fill a quad with depth of r = 0.625, passed into glTexCoord for
   every vertex. */ Add your quad code here

/* Fill a quad with depth of r = 0.875, passed into glTexCoord for
   every vertex. */ Add your quad code here

/* Now get a slice across the quad. Heres some quad code for a
   sample. Make sure you have appropriate viewing perspectives. */

glBegin(GL_QUADS);
  glNormal3f(0., 0., 1.);
  glTexCoord3f(0.0, 0.0, 0.0);
  glVertex3f(0.5, 0.5, 0.);
  glNormal3f(0., 0., 1.);
  glTexCoord3f(0.0, 1.0, 0.0);
  glVertex3f(0.5, 62.5, 0.);
  glNormal3f(0., 0., 1.);
  glTexCoord3f(1.0, 1.0, 1.0);
  glVertex3f(62.5, 62.5, 0.);
  glNormal3f(0., 0., 1.);
  glTexCoord3f(1.0, 0.0, 1.0);
  glVertex3f(62.5, 0.5, 0.);
glEnd();
}
```

The results of code fragments are shown in Figure 1-8. This figure shows four layers in the base mipmap level, and a diagonal slice through the base mipmap level.

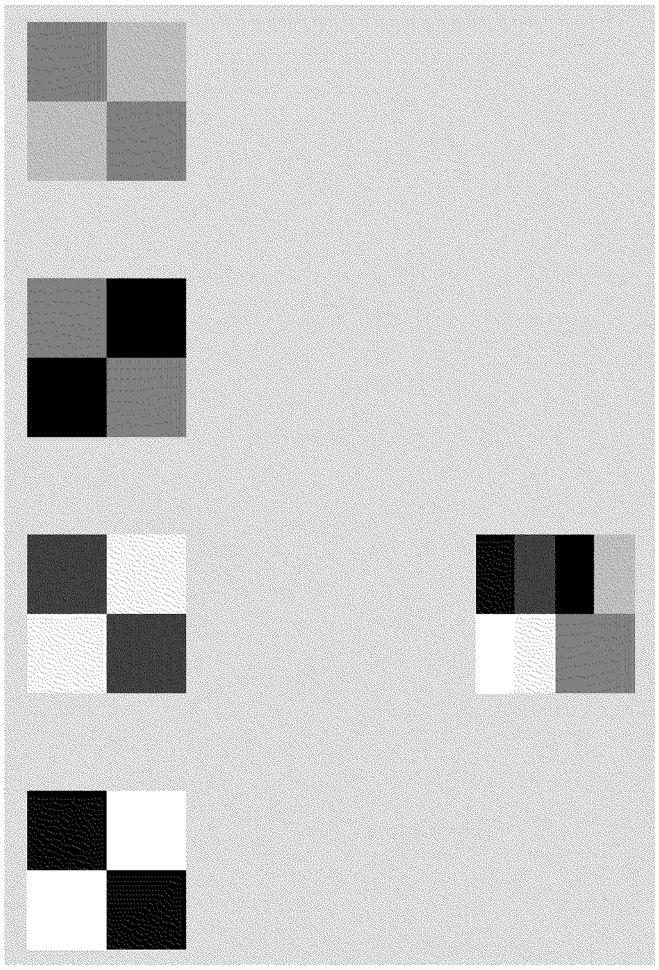


Figure 1-8. Results from the 3D Texture Program Fragments

For more information on 3D texture, see the functions: `glTexImage3DEXT`, `glTexSubImage3DEXT`, `glCopyTexSubImage3DEXT`, `glEnable`, `glDisable`.

Shadow and Depth Extensions

The texture depth extension provides a depth texture format. This is needed to use the shadow texture extension. The shadow texture extension is used to compare the texture's `r` components against the corresponding texel value. Each texel is compared using user specified comparison rules. If the comparison rule passes, the fragments `alpha` value will be set to one by the shadow texture extension. If the comparison rule fails, the fragment's `alpha` value will be set to zero. The `alpha` test can then mask out shadowed areas using the `alpha` values.

When using this extension, set minification and magnification filter to either `GL_NEAREST` or `GL_LINEAR`. Mipmap minification filters of `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, and `GL_LINEAR_MIPMAP_LINEAR` will give indeterminate results.

Table 1-8.
Enumerated Types for Shadow and Depth Texture Extension

Extended Area	Enumerated Types	Description
Texture Formats	`GL_DEPTH_COMPONENT`, `GL_DEPTH_COMPONENT16_EXT`, `GL_DEPTH_COMPONENT24_EXT`, `GL_DEPTH_COMPONENT32_EXT` default: N/A	Texel formats which are useful when using shadow texturing.
Texture Parameter	`GL_TEXTURE_COMPARE_EXT`, default: `GL_FALSE`	Enables comparison to the `r` coordinate when set to true.
Texture Parameter	`GL_TEXTURE_DEPTH_EXT` default: `GL_FALSE`	Used to query if you have depth extension available.
Texture Parameter	`GL_TEXTURE_COMPARE_OPERATOR_EXT`, `GL_LEQUAL_R_EXT`, `GL_GEQUAL_R_EXT` default: `GL_TEXTURE_LEQUAL_R_EXT`	Sets the particular type of comparison with the `r` texture coordinate.

Steps for Shadow Texturing

1. Set the `GL_TEXTURE_COMPARE_EXT` to `GL_TRUE` in `glTexParameter`
2. Set the `GL_TEXTURE_COMPARE_OPERATOR_EXT` to either `GL_TEXTURE_LEQUAL_R_EXT` or `GL_TEXTURE_GEQUAL_R_EXT` using `glTexParameter`.
3. Use `glTexImage` with `GL_DEPTH_COMPONENT` to fill the texture with the image which will be compared with the r coordinate.
4. Use `glTexCoord3*` to set the `s`, `t`, and `r` texture coordinates at the vertices of the object to be rendered.

Shadow Texturing Program

This program renders a simple quadrilateral using the shadow texture extension and the alpha test.

```
/* Put unusual includes */

#define TEXTURE_WIDTH 256
#define TEXTURE_HEIGHT 256
GLubyte texture[TEXTURE_WIDTH][TEXTURE_HEIGHT][1];

static void checkerPattern(int width,
                           int height,
                           int Checker_period)
{
    int texelX, texelY;
    int index ;
    index = 0;

    for (texelY = 0; texelY < height; texelY++) {
        for (texelX = 0; texelX < width; texelX++) {
            if (((texelX/Checker_period) % 2)^((texelY/Checker_period) % 2)) {
                texture[texelX][texelY][0] = (GLubyte) 0;   /* depth */
            } else {
                texture[texelX][texelY][0] = (GLubyte) 255;   /* depth */
            }
        }
    }
}

main code fragment

/* INSERT your favorite window create and map code here */

/* Set up transforms */
glMatrixMode(GL_PROJECTION);
glLoadIdentity ();
glOrtho (-10, 130., -10., 130., 1., -1.);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity ();

glEnable(GL_DEPTH_TEST);
glEnable(GL_ALPHA_TEST);
glEnable(GL_TEXTURE_2D);
glDepthFunc(GL_LEQUAL);
glAlphaFunc(GL_GREATER, 0.5);
```

```
glClearDepth(1.0);
width = TEXTURE_WIDTH;
height = TEXTURE_HEIGHT;
checkerPattern( width,height, 64);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_EXT, GL_TRUE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_OPERATOR_EXT,
  GL_TEXTURE_LEQUAL_R_EXT);

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT16_EXT, width,
             height, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE,
             &texture);

/* Render a red (background) and blue (primitive) checker pattern  */
glClearColor(1.0, 0.0, 0.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glColor3f(0.0, 0.0, 1.0);
glBegin(GL_QUADS);
  glTexCoord3d(0.0, 0.0, 0.0);
  glVertex3f(0.5, 0.5, 0.);
  glTexCoord3d(0.0, 1.0, 1.0);
  glVertex3f(0.5, 106.5, 0.);
  glTexCoord3d(1.0, 1.0, 1.0);
  glVertex3f(106.5, 106.5, 0.);
  glTexCoord3d(1.0, 0.0, 0.0);
  glVertex3f(106.5, 0.5, 0.);
glEnd();
```

Figure 1-9 shows the results from executing the program.

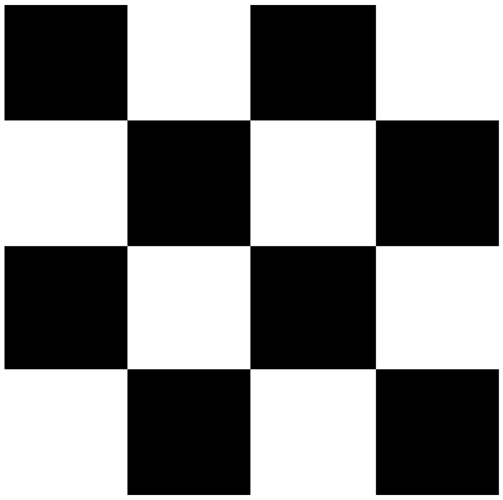


Figure 1-9. Results from Shadow Texturing

For related information, see the function `glTexParameter`.

Texture Lighting Extension

The texture lighting extension defines a mechanism for applications to request that color originating from specular lighting be added to the fragment color after texture application. This is referred to as `preLight` texturing.

Table 1-9. Enumerated Types for Pre-Light Texturing

Extended area	Enumerated Types	Description
Texture Environ-ment	`GL_TEXTURE_LIGHTING_MODE_HP`, `GL_TEXTURE_PRE_SPECULAR_HP`, `GL_TEXTURE_POST_SPECULAR_HP` default: N/A	`pname` and `param` parameters for `glTexEnv`.

Procedure for preLight Texturing

You need to add the following `preLight` texturing code fragments to the normal texturing program that also has lighting.

```
glTexEnv[if](GL_TEXTURE_ENV, GL_TEXTURE_LIGHTING_MODE_HP,
             GL_TEXTURE_PRE_SPECULAR_HP);
```

or

```
GLfloat appMode=GL_TEXTURE_PRE_SPECULAR_HP; glTexEnvf(GL_TEXTURE_ENV,
             GL_TEXTURE_LIGHTING_MODE_HP, &appMode);
```

The results from using `preLight` texturing are given in Figure 1-10. Note that the top image is without prelight texturing, and the bottom is with `preLight` texturing. The left half of each image is the untextured specular-lighted image, and the right half of each image uses `GL_REPLACE` texturing.

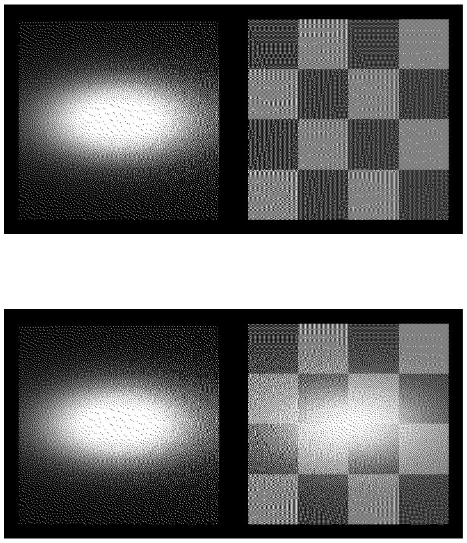


Figure 1-10. Results from Prelight Texturing

For related information, see the function `glTexEnv`.

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 also very useful for viewing frustum culling. If a part's bounding box is not visible for any reason (not just 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 1-10. Enumerated Types for Occlusion

Extended Area	Enumerated Types	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`.

Texture Autogen Mipmap Extension

The autogen mipmap extension introduces a side effect to the modification of the base level texture map. When enabled, any change to the base-level texture map will cause the computation of a complete mipmap for that base level. The internal formats and border widths of the derived mipmap will match those of the base map, and the dimensions of the derived mipmap follow the requirements set forth in OpenGL for a valid mipmap. A simple 2×2 box filter is used to generate the mipmap levels.

Table 1-11. Enumerated Types for Occlusion

Extended Area	Enumerated Types	Description
Texture Parameter	`GL_GENERATE_MIPMAP_EXT` default: `GL_FALSE`	Enables autogen mipmap.

To use the autogen mipmap extension, set `GL_GENERATE_MIPMAP_EXT` to `GL_TRUE` in `glTexParameter`. For example, here is a code fragment that uses this extension:

```
glTexParameter[if](GL_TEXTURE_2D, GL_GENERATE_MIPMAP_EXT, GL_TRUE);
```

For related information on this extension, see the function `glTexParameter`.

X Window Extensions for HP's Implementation of OpenGL

HP's implementation of OpenGL includes two GLX extensions that deal with extended GLX visual information that is not included in the OpenGL 1.1 Standard. These extensions are both supported by HP's implementation of the OpenGL API library, but prior to using them,

`glXQueryExtensionsString` should be called to verify that the extensions are supported on the target display.

GLX Visual Information Extension

The `GLX_EXT_visual_info` extension provides additional GLX visual information and enhanced control of GLX visual selection. The enumerated types listed below can be passed to either `glXChooseVisual`, or `glXGetConfig` to specify or inquire the visual type or transparency capabilities.

Table 1-12. Enumerated Types for GLX Visual Information

Extended Area	Enumerated Types	Description
Visual Type	`GLX_TRUE_COLOR_EXT`, `GLX_DIRECT_COLOR_EXT`, `GLX_PSEUDO_COLOR_EXT`, `GLX_STATIC_COLOR_EXT`[1], `GLX_GRAY_SCALE_EXT`[1], `GLX_STATIC_GRAY_EXT`[1] default: N/A	Values associated with the `GLX_X_VISUAL_TYPE_EXT` enumerated type.
Visual Trans-parency Capabilities	`GLX_NONE_EXT`, `GLX_TRANSPARENT_RGB_EXT`[1], `GLX_TRANSPARENT_INDEX_EXT` default: `GLX_NONE_EXT`	Values associated with the `GLX_TRANSPARENT_TYPE_EXT` enumerated type.
1. These enumerated types are supported through the GLX client-side API library, but there are currently no HP X Server GLX VIsuals with these capabilities. They can still be used to query any Server and will operate properly if connected to a non-HP server with GLX support for these visual capabilities.		

The enumerated types listed below can be used only through `glXGetConfig` when it is known that the GLX visual being queried supports transparency or in other words, has a `GLX_TRANSPARENT_TYPE_EXT` property other than `GLX_NONE_EXT`.

Table 1-13. Enumerated Types for GLX Visual Transparency

Extended Area	Enumerated Types	Description
Transparency Index for PseudoColor Visuals	`GLX_TRANSPARENT_INDEX_VALUE_EXT` default: N/A	Returns the Pixel Index for the transparent color in a `GLX_TRANSPARENT_INDEX_EXT` visual.
Transparency Values for RGBA Visuals	`GLX_TRANSPARENT_RED_VALUE_EXT`, `GLX_TRANSPARENT_GREEN_VALUE_EXT`, `GLX_TRANSPARENT_BLUE_VALUE_EXT`, `GLX_TRANSPARENT_ALPHA_VALUE_EXT` default: N/A	Returns the RGBA data values for the transparent color in a `GLX_TRANSPARENT_RGB_EXT` type GLX visual (Not supported on HP Servers).

GLX_EXT_visual_info Program Fragments

Note that both of the following segments assume that the `GLX_EXT_visual_info` extension exists for `dpy`, which is a pre-existing display connection to an X Server.

Here is a sample code segment that forces selection only of a TrueColor visual.

```
Display *dpy;
XVisualInfo *vInfo;
int attrList[] = {GL_USE_GL,
                  GLX_X_VISUAL_TYPE_EXT,
                  GLX_TRUE_COLOR_EXT,
                  None};

vinfo = glXChooseVisual(dpy, XDefaultScreen(dpy), &attrList);
```

The following sample is a code segment that selects an overlay visual with index transparency, and then obtains the Pixel index for the transparent color.

```
Display *dpy;
XVisualInfo *visInfo;
int transparentPixel;
int attrList[] = {GL_USE_GL,
                  GLX_LEVEL, 1,
                  GLX_TRANSPARENT_TYPE_EXT,
                  GLX_TRANSPARENT_INDEX_EXT,
                  None};

visInfo = glXChooseVisual(dpy, XDefaultScreen(dpy), &attrList);
if (visInfo != NULL) {
    glXGetConfig(dpy, visInfo, GLX_TRANSPARENT_INDEX_VALUE_EXT,
                 &transparentPixel);
}
```

GLX Visual Rating Extension

The `GLX_EXT_visual_rating` extension provides additional GLX visual information which applies rating properties to GLX visuals. The enumerated types listed below can be passed to either `glXChooseVisual`, or `glXGetConfig` to specify or inquire visual rating information.

Table 1-14. Enumerated Types for GLX Visual Rating

Extended Area	**Enumerated Types**	**Description**
Visual Rating	`GLX_NONE_EXT`, `GLX_SLOW_VISUAL_EXT`, `GLX_NON_CONFORMANT_VISUAL_EXT` default: N/A	Values associated with the `GLX_VISUAL_CAVEAT_EXT` enumerated type.

Note that all current HP GLX visuals are rated as `GLX_NONE_EXT`. This extension is implemented for possible future visual support and for use with non-HP servers. Coding to use the `GLX_EXT_visual_rating` extension is similar to the segments listed above for the `GLX_EXT_visual_info` extension.

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.

EXP and EXP2 Fogging

The Virtual Memory Driver's implementation of fog applies fog per fragment. Hardware devices implement EXP and EXP^2 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_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_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`
  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` equals Object Coordinates and `EC` equals Eye Coordinates. For details on the lighting space, see the sections "Lighting Space" and "Optimization of Lighting" found in Chapter 5.
- `HPOGL_TXTR_SHMEM_THRESHOLD`
  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 ot 1024×1024 bytes. This variable should be set to the byte size desired for shared memory usage.

- `HPOGL_ALLOW_LOCAL_INDIRECT_CONTEXTS`
  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. This variable may be set if a need arises to really create a local indirect context.

- `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.

2 Installation and Setup

Introduction

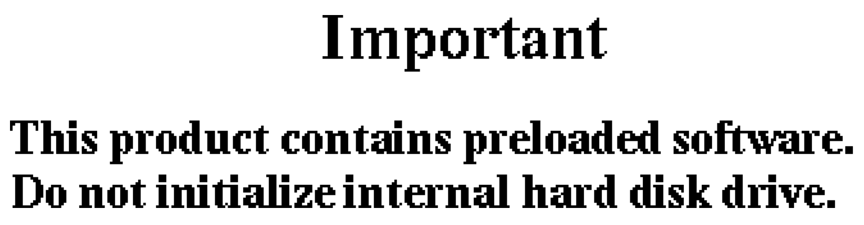
If you are setting up a new workstation, all software is preloaded for you if you purchased the Instant Ignition option. A subsequent section will explain how to determine if OpenGL has been installed. If you did not order Instant Ignition, then you will need to install the OpenGL filesets from the C/ANSI C Developer's Toolkit.

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 yellow 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.10.20     HP-UX OpenGL Developer's Kit
B6196AA          B.10.20     HP-UX 700 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. Read this entire procedure
2. Install Workstation ACE for HP-UX 10.20 (July 1997) patch bundle
3. Install OpenGL
4. Check log file
5. Verify the product.

Each step is described on the subsequent sections.

1. Read this entire procedure

Read all of this procedure to ensure the proper installation of your OpenGL product.

2. Install HP-UX 10.20 and the Workstation ACE for HP-UX 10.20 (July 1997) patch bundle

Before installing OpenGL, install HP-UX 10.20 and the Workstation ACE for HP-UX 10.20 (July 1997) patch bundle shipped with your developer's toolkit.

3. Install OpenGL

Once you have installed HP-UX 10.20, you can install the OpenGL programming environment. This programming environment is bundled with the C/ANSI C Developer's product. 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 environment product includes the filesets shown in Table 2-1. To list these filesets, execute this command:

```
/usr/sbin/swlist -l fileset OpenGLDevKit
```

Table 2-1. OpenGL Development Environment Filesets

OpenGL Fileset	Contains
`OPENGL-CONTRIB`	Contributed or unsupported program files
`OPENGL-EXAMPLE`	Example program source code
`OPENGL-PRG`	Files necessary for the OpenGL programming environment
`OPENGL-WEBDOC`	Online documentation files

4. 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.

5. 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 `/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 `/opt/graphics/OpenGL/contrib/glut_samples` directory, which also contains a `README` file.

The OpenGL File Structure

The OpenGL file structure is compliant with the file structure of the HP-UX 10.*x* file system. 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), which 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`
  - `libddvisxgl.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.

FINAL TRIM SIZE : 7.5 in x 9.0 in

3 Running OpenGL Programs

Introduction

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. If a request is made for a single buffered visual, a double-buffered visual will match the request, but that visual will only be available for single-buffered rendering unless a new display connection is opened to the VGL display.
- 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.
- 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)
  ```

  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.

The Virtual Memory Driver (VMD)

Instead of rendering OpenGL graphics to a dedicated graphics display subsystem, VMD is designed to render these images to a virtual-memory frame buffer and send these images to an X11 drawable using standard X11 protocol.

Because HP VMD uses the X11 protocol to display the images, this targeted drawable may be local or remote. This may include rendering to X terminals, older HP devices, or a personal computer. The only requirement is that the output is directed to an X11 drawable. (See Chapter 1 for a list of supported VMD configurations) VMD is also the driver used to render to GLX pixmaps.

When a GLX context is created for rendering three-dimensional graphics using OpenGL, GLX first checks to see if the X server supports the GLX extension. If it does not, the Virtual Memory Driver will be used. GLX examines the available list of X visuals and decides which ones can be software extended to be GLX visuals (see the supported visuals list). Buffers are allocated in virtual memory for the OpenGL color and ancillary buffers. When the application issues a `glFlush()`, `glFinish()`, or a `glXSwapBuffers()` call, the contents of the corresponding virtual-memory color buffers are sent to the X11 window using X protocol.

Double buffering for VMD is implemented using the X11 Double-Buffering Extension (DBE). Double-buffered visuals are not available for HP OpenGL rendering with VMD on X servers that do not support DBE.

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`.

Resource usage needs to be taken into consideration as well. VM buffers are allocated for all of the OpenGL color and ancillary buffers. Color buffers are allocated when the context is created. Other buffers (depth, stencil, accumulation) are allocated at first use. These buffers can be quite large.

For example, consider an X11 window 750 pixels wide and 600 pixels high. The size of each VM color buffer for an 8-bit visual is:

750 pixels $\times$ 600 pixels $\times$ 1 byte/pixel = 450,000 bytes

Consider that an OpenGL application may use two color buffers (for double buffering), a 32-bit depth/stencil buffer, and a 48-bit accumulation buffer. The size of the virtual memory required then becomes 5,400,000 bytes. In addition, the amount of virtual memory required is correspondingly larger for 12-bit and 24-bit color buffers.

Running HP's Implementation of the OpenGL Stereo Application

With HP's implementation of OpenGL and the VISUALIZE-FX family of graphics devices, it is now possible to run HP's implementation of OpenGL "stereo in a window" mode. Unlike previous HP stereo implementations, "stereo in a window" affects only OpenGL windows that have been created with "stereo capable" `GLX` visuals. The remainder of the X11 screen is rendered in non-stereo mode without any flickering or color artifacts.

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. 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 *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. 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, and you can verify the availability of GLX stereo visuals by running the `xglinfo` command again.

3. To select one of the stereo capable `GLX` visuals through OpenGL, the `GLX_STEREO` enumerated value 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

Introduction

Table 4-1 contains a list of the subdirectories in the /opt/graphics/OpenGL directory. These subdirectories contain header files and libraries which may be used when compiling and linking your programs. They also include helpful sample source code.

Table 4-1. OpenGL Directories and their Content

Subdirectory	**This Directory Contains** . . .
`include/GL`	Header files needed for OpenGL development.
`lib`	Several run-time shared libraries.
`lbin`	Run-time executables.
`doc`	OpenGL documentation including reference pages.
`contrib/libwidget`	A Motif widget library and source code.
`contrib/glut_samples`	Example OpenGL programs that are referenced in the *OpenGL Programming Guide*.
`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.
`contrib/glw_samples`	Source code for Motif widget sample programs.

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 the 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:

```
c89 program.c \
  -I/opt/graphics/OpenGL/include \
  -L/opt/graphics/OpenGL/lib \
  -lXext -lGL -lX11
```

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

If you are going to compile your application using HP's ANSI C++ compiler or cfront C++ compiler, use one of the following commands:

`aCC`	—used to compile an ANSI C++ application.
`CC +a1`	—used to compile a cfront C++ application.

See the *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

OpenGL Procedure Calls

In order to facilitate maximum performance, the OpenGL library uses a unique procedure calling convention. This convention is currently supported only by the HP C and C++ compilers which are available in conjunction with the Workstation ACE for HP-UX 10.20 (July 1997) OpenGL developers package.

If you get a large number of "`Undefined pragma`" messages (for example, `Undefined pragma "HP_PLT_CALL" ignored`) when compiling an OpenGL application, you are most likely using a compiler that does not support this new 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

Introduction

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*.

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).

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 if put in a display list. 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 3-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.

Textures

If calls to `glTexImage` are put into a display list, they may be cached. 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();
```

Regular Primitive Data

If the vertex data that you give to a display list is regular (i.e. 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(&v);
glVertex3fv(&p1); glVertex3fv(&p2); glVertex3fv(&p3);
glNormal3fv(&v);
glVertex3fv(&p1); glVertex3fv(&p2); glVertex3fv(&p3);
...
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(&v); glVertex3fv(&p1);
glNormal3fv(&v); glVertex3fv(&p2);
glNormal3fv(&v); glVertex3fv(&p3);
...
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.

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 to 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.

Lighting Space

OpenGL specifies that lighting operations should be done in Eye Coordinte space. However, if the model-view matrix is isotropic, equivalent lighting calculations can be performed in Object Coordinate space, by transforming stored light positions to Object Coordinates. If there are many vertices between model-view matrix changes, Object-Coordinate space lighting is faster than Eye Coordinate space lighting since the transformation of vertices and normals from Object- to Eye Coordinates can be skipped.

Whether or not Object Coordinate lighting is faster than Eye Coordinate lighting depends on the command mode (immediate mode vs. execution of a display list or vertex array) as well as the number of vertices between model-view matrix changes.

The selection of a lighting space occurs at the start of the next primitive (`glBegin` or vertex array) after any GL calls that could affect the choice of lighting space. The choice of lighting space can be affected by those GL calls that:

- Change Object Coordinates to Eye Coordinates (model-view matrix)
- Turn on/off fog
- Turn on/off generation of spherical texture coordinates.

If the model-view matrix is anisotropic, lighting must be done in Eye Coordinates. Lighting will also be done in Eye Coordinates when fogging and spherical-texture-coordinate generation are done in Eye Coordinates.

If none of the above conditions which force Eye Coordinate lighting are true, then HP's implementation of OpenGL chooses the lighting space depending on how OpenGL commands are being executed at the time a choice must be made. If commands are being executed in immediate mode, eye space lighting is chosen. If commands are being executed from a display list or if a vertex array is being executed, object space lighting is chosen.

Eye-space lighting works well when commands are executed in immediate mode, and object-space lighting works well when:

- There are many (8 or more) vertices between changes to light definitions or to the model-view matrix.
- A display list or vertex array is used.

You can override the above lighting space selection rules by setting the environment variable `HPOGL_LIGHTING_SPACE`. To set this environment variable, execute the following command:

```
export HPOGL_LIGHTING_SPACE=EC
```

when any of the following are true:

- The application uses display lists or vertex arrays, but makes frequent changes to the model-view matrix or to light definitions (using `glLight`).
- The application uses display lists or vertex arrays, but frequently turns fogging or spherical-texture-coordinate generation on/off.
- The application uses 4D data (for example, vertices, light positions) and the `w` values are near the floating point limits. See the section below on 4D values for more information.

It is appropriate to use

```
export HPOGL_LIGHTING_SPACE=OC
```

when:

- There are many (eight or more) vertices between light changes or model-view matrix changes.
- Display lists or vertex arrays are used extensively.

and any of the following are true:

- There is a lot of switching between immediate mode and display-list execution (or vertex arrays) while lighting is on, and intermixed with state changes that affect choice of lighting space (these were listed above).
- Display lists are used predominantly, but the first `glBegin` after commands that affect choice of lighting space (which were listed above) is an immediate mode command.

For example, in a scenario like the following:

```
(Load a model View Matrix)
(Define lights)
(Enable lights and lighting)

glBegin(..); /* Some simple immediate-mode rendering.
                Lighting space gets chosen at this point. */
                 ...
glEnd();

(Execution of many display lists or vertex arrays, with no
 changes to the model-view matrix or light definitions)
```

Here the default lighting space chosen at the time of the first `glBegin` is eye-space lighting. The display lists could have benefited from setting `HPOGL_LIGHTING_SPACE=OC`.

When tuning an application, first use just the default lighting-space selection (do not set `HPOGL_LIGHTING_SPACE`). If the application matches the conditions listed above that indicate the need for setting `HPOGL_LIGHTING_SPACE`, then experiment with setting the environment variable.

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.

A

Function Reference

This appendix contains the reference pages for the three extension functions that Hewlett-Packard defined for its implementation of OpenGL. The three functions are:

- `glCopyTexSubImage3DEXT`
- `glTexImage3DEXT`
- `glTexSubImage3DEXT`

For a paper copy of the standard OpenGL functions supported in HP's implementation of OpenGL, see the *OpenGL Reference Manual* (the "Blue Book"). For an on-line version of *all* of the reference pages—the standard ones included in the Blue Book plus HP's extensions—see the browsable OpenGL documentation under `/opt/graphics/OpenGL/doc/Web/`.

glCopyTexSubImage3DEXT

`glCopyTexSubImage3DEXT`: copy pixels into a 3D texture subimage.

C Specification

```
void glCopyTexSubImage3DEXT(
    GLenum   target,
    GLint    level,
    GLint    xoffset,
    GLint    yoffset,
    GLint    zoffset,
    GLint    x,
    GLint    y,
    GLsizei  width,
    GLsizei  height)
```

Parameters

target	The target texture. Must be `GL_TEXTURE_3D_EXT`.
level	The level-of-detail number. Level 0 is the base image level, and level n is the nth mipmap reduction image.
xoffset	Texel offset in the X direction within the texture array.
yoffset	Texel offset in the Y direction within the texture array.
zoffset	Texel offset in the Z direction within the texture array.
x	The X coordinate of the lower-left corner of the pixel rectangle to be transferred to the texture array.
y	The Y coordinate of the lower-left corner of the pixel rectangle to be transferred to the texture array.
width	The width of the texture subimage.
height	The height of the texture subimage.

Description

`glCopyTexSubImage3DEXT` replaces a rectangular portion of a three-dimensional texture image with pixels from the current `GL_READ_BUFFER` (rather than from main memory, as is the case for `glTexSubImage3DEXT`).

The screen-aligned pixel rectangle with lower-left corner at (x, y) having width *width* and height *height* replaces the rectangular area of the S-T slice located at *zoffset* with X indices *xoffset* through $\mathit{xoffset}+\mathit{width}-1$, inclusive, and Y indices *yoffset* through $\mathit{yoffset}+\mathit{height}-1$, inclusive.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

The pixels in the rectangle are processed exactly as if `glCopyPixels` had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0, 1] and then converted to the texture's internal format for storage in the texel array.

If any of the pixels within the specified rectangle of the current `GL_READ_BUFFER` are outside the read window associated with the current rendering context, then the values obtained for those pixels are undefined.

Notes

`glCopyTexSubImage3DEXT` is part of the `EXT_copy_texture` extension.

Errors

- `GL_INVALID_ENUM` is generated when *target* is not one of the allowable values.
- `GL_INVALID_VALUE` is generated if *level* is less than zero or greater than $\log_2 max$, where *max* is the returned value of `GL_MAX_TEXTURE_SIZE`.
- `GL_INVALID_VALUE` is generated if *xoffset* $<$ −`TEXTURE_BORDER`,
  (*xoffset*+*width*) $>$ (`TEXTURE_WIDTH`−`TEXTURE_BORDER`),
  yoffset $<$ −`TEXTURE_BORDER`, or if
  zoffset $<$ −`TEXTURE_BORDER`, where `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, and `TEXTURE_BORDER` are the state values of the texture image being modified, and interlace is 1 if `GL_INTERLACE_SGIX` is disabled, and 2 otherwise. Note that `TEXTURE_WIDTH` and `TEXTURE_HEIGHT` include twice the border width.

- `GL_INVALID_VALUE` is generated if *width* or *height* is negative.
- `GL_INVALID_OPERATION` is generated when the texture array has not been defined by a previous `glTexImage3D` (or equivalent) operation.
- `GL_INVALID_OPERATION` is generated if `glCopyTexSubImage3DEXT` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.

Associated Gets

`glGetTexImage`

See Also

`glTexImage3D`,
`glTexSubImage3DEXT`,
`glCopyPixels`.

glTexImage3DEXT

`glTexImage3DEXT`: Specify a three-dimensional texture image.

C Specification

```
void glTexImage3DEXT(
    GLenum      target,
    GLint       level,
    GLenum      internalformat,
    GLsizei     width,
    GLsizei     height,
    GLsizei     depth,
    GLint       border,
    GLenum      format,
    GLenum      type,
    const       GLvoid *pixels)
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_3D_EXT` or `GL_PROXY_TEXTURE_3D_EXT`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.

Specifies the number of color components in the texture. Must be 1, 2, 3, or 4, or one of the following symbolic constants:

`GL_ALPHA`, `GL_ALPHA4`,
`GL_ALPHA8`, `GL_ALPHA12`,
`GL_ALPHA16`, `GL_LUMINANCE`,
`GL_LUMINANCE4`, `GL_LUMINANCE8`,
`GL_LUMINANCE12`, `GL_LUMINANCE16`,
`GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`,
`GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`,
`GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`,
`GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`,
`GL_INTENSITY4`, `GL_INTENSITY8`,

GL_INTENSITY12, GL_INTENSITY16,
GL_R3_G3_B2, GL_RGB,
GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10,
GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2,
GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2,
GL_RGBA12, or GL_RGBA16.

Additionally, if the extension GL_EXT_shadow is supported, may be one of the symbolic constants GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_EXT, GL_DEPTH_COMPONENT24_EXT, or GL_DEPTH_COMPONENT32_EXT.

width	Specifies the width of the texture image. Must be $2^n + 2 \times border$ for some integer n.
height	Specifies the height of the texture image. Must be $2^m + 2 \times border$ for some integer m.
depth	Specifies the depth of the texture image. Must be $2^l + 2 \times border$ for some integer l.
border	Specifies the width of the border. Must be either 0 or 1.
format	Specifies the format of the pixel data. The following symbolic values are accepted: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA. If the extension GL_EXT_shadow is supported, the symbolic value GL_DEPTH_COMPONENT is also accepted.
type	Specifies the data type of the pixel data. The following symbolic values are accepted: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT.
pixels	Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. Three-dimensional texturing is enabled and disabled using glEnable and glDisable with argument `GL_TEXTURE_3D_EXT`.

Texture images are defined with `glTexImage3DEXT`. The arguments describe the parameters of the texture image, such as height, width, depth, width of the border, level-of-detail number (see `glTexParameter`), and the internal resolution and format used to store the image. The last three arguments describe the way the image is represented in memory, and they are identical to the pixel formats used for `glDrawPixels`.

If target is `GL_PROXY_TEXTURE_3D_EXT` no data is read from pixels, but all of the texture image state is recalculated, checked for consistency, and checked against the implementation's capabilities. If the implementation cannot handle a texture of the requested texture size, it will set all of the texture image states to 0 (`GL_TEXTURE_WIDTH`, `GL_TEXTURE_HEIGHT`, `GL_TEXTURE_BORDER`, `GL_TEXTURE_COMPONENTS`), but no error will be generated.

If target is `GL_TEXTURE_3D_EXT`, data is read from pixels as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on type. These values are grouped into sets of one, two, three, or four values, depending on format, to form elements.

The first element corresponds to the lower-left-rear corner of the texture volume. Subsequent elements progress left-to-right through the remaining texels in the lowest-rear row of the texture volume, then in successively higher rows of the rear 2D slice of the texture volume, then in successively closer 2D slices of the texture volume. The final element corresponds to the upper-right-front corner of the texture volume.

Each element of pixels is converted to an RGBA element according to

`GL_COLOR_INDEX`	Each element is a single value, a color index. It is converted to fixed point (with an unspecified number of zero bits to the right of the binary point), shifted left or right depending on the value and sign of `GL_INDEX_SHIFT`, and added to `GL_INDEX_OFFSET` (see `glPixelTransfer`). The resulting index is converted to a set of color components using the `GL_PIXEL_MAP_I_TO_R`,

	GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A tables, and clamped to the range [0, 1].
GL_RED	Each element is a single red component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 for alpha.
GL_GREEN	Each element is a single green component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 for alpha.
GL_BLUE	Each element is a single blue component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 for alpha.
GL_ALPHA	Each element is a single alpha component. It is converted to floating-point and assembled into an RGBA element by attaching 0.0 for red, green, and blue.
GL_RGB	Each element is an RGB triple. It is converted to floating-point and assembled into an RGBA element by attaching 1.0 for alpha (see glPixelTransfer).
GL_RGBA, GL_ABGR_EXT	Each element contains all four components; for GL_RGBA, the red component is first, followed by green, then blue, and then alpha; for GL_ABGR_EXT the order is alpha, blue, green, and then red.
GL_LUMINANCE	Each element is a single luminance value. It is converted to floating-point, then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1.0 for alpha.
GL_LUMINANCE_ALPHA	Each element is a luminance/alpha pair. It is converted to floating-point, then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue.

Please refer to the glDrawPixels reference page for a description of the acceptable values for the type parameter.

An application may desire that the texture be stored at a certain resolution, or that it be stored in a certain format. This resolution and format can be requested by *internalformat*, but the implementation may not support that resolution (the formats of `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, and `GL_RGBA` must be supported). When a resolution and storage format is specified, the implementation will update the texture state to provide the best match to the requested resolution. The `GL_PROXY_TEXTURE_3D_EXT` target can be used to try a resolution and format. The implementation will compute its best match for the requested storage resolution and format; this state can then be queried using `glGetTexLevelParameter`.

A one-component texture image uses only the red component of the RGBA color extracted from pixels. A two-component texture image uses the R and A values. A three-component texture image uses the R, G, and B values. A four-component texture image uses all of the RGBA components.

Notes

Texturing has no effect in color index mode.

The texture image can be represented by the same data formats and types as the pixels in a glDrawPixels command, except that formats `GL_STENCIL_INDEX` and `GL_DEPTH_COMPONENT` cannot be used, and type `GL_BITMAP` cannot be used. `glPixelStore` and `glPixelTransfer` modes affect texture images in exactly the way they affect `glDrawPixels`.

A texture image with zero height, width, or depth indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

`glTexImage3DEXT` is part of the `EXT_texture3d` extension.

Errors

- `GL_INVALID_ENUM` is generated when *target* is not an accepted value.
- `GL_INVALID_ENUM` is generated when *format* is not an accepted value.
- `GL_INVALID_ENUM` is generated when *type* is not an accepted value.
- `GL_INVALID_VALUE` is generated if *level* is less than zero or greater than $\log_2 max$, where *max* is the returned value of `GL_MAX_3D_TEXTURE_SIZE_EXT`.
- `GL_INVALID_VALUE` is generated if *internalformat* is not an accepted value.

- `GL_INVALID_VALUE` is generated if *width*, *height*, or *depth* is less than zero or greater than `GL_MAX_3D_TEXTURE_SIZE_EXT`, when *width*, *height*, or *depth* cannot be represented as $2^k + 2 \times border$ for some integer k.
- `GL_INVALID_VALUE` is generated if border is not 0 or 1.
- `GL_INVALID_OPERATION` is generated if `glTexImage3DEXT` is executed between the execution of `glBegin` and the corresponding execution of `glEnd`.
- `GL_TEXTURE_TOO_LARGE_EXT` is generated if the implementation cannot accomodate a texture of the size requested.

Associated Gets

`glGetTexImage`

`glIsEnabled` with argument `GL_TEXTURE_3D_EXT`

See Also

`glDrawPixels`,
`glFog`,
`glPixelStore`,
`glPixelTransfer`,
`glTexEnv`,
`glTexGen`,
`glTexImage1D`,
`glTexImage2D`,
`glTexParameter`.

glTexSubImage3DEXT

`glTexSubImage3DEXT`: specify a three-dimensional texture subimage.

C Specification

```
void glTexSubImage3DEXT(
    GLenum     target,
    GLint      level,
    GLint      xoffset,
    GLint      yoffset,
    GLint      zoffset,
    GLsizei    width,
    GLsizei    height,
    GLsizei    depth,
    GLenum     format,
    GLenum     type,
    const      GLvoid *pixels)
```

Parameters

target	Specifies the target texture. Must be `GL_TEXTURE_3D_EXT`.
level	Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
xoffset	Specifies a texel offset in the X direction within the texture array.
yoffset	Specifies a texel offset in the Y direction within the texture array.
zoffset	Specifies a texel offset in the Z direction within the texture array.
width	Specifies the width of the texture subimage.
height	Specifies the height of the texture subimage.
depth	Specifies the depth of the texture subimage.
format	Specifies the format of the pixel data. The following symbolic values are accepted: `GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`. If the

extension `GL_EXT_shadow` is supported, the symbolic value `GL_DEPTH_COMPONENT` is also accepted.

type Specifies the data type of the pixel data. The following symbolic values are accepted: `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, and `GL_FLOAT`.

pixels Specifies a pointer to the image data in memory.

Description

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. Three-dimensional texturing is enabled and disabled using `glEnable` and `glDisable` with argument `GL_TEXTURE_3D_EXT`.

`glTexSubImage3DEXT` redefines a contiguous subregion of an existing three-dimensional texture image. The texels referenced by *pixels* replace the portion of the existing texture array with X indices *xoffset* and $xoffset+width-1$, inclusive, Y indices *yoffset* and $yoffset+height-1$, inclusive, and Z indices *zoffset* and $zoffset+depth-1$, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width, height or depth, but such a specification has no effect.

Notes

Texturing has no effect in color index mode.

`glPixelStore` and `glPixelTransfer` modes affect texture images in exactly the way they affect `glDrawPixels`.

Errors

- `GL_INVALID_ENUM` is generated when *target* is not `GL_TEXTURE_3D_EXT`.
- `GL_INVALID_OPERATION` is generated when the texture array has not been defined by a previous `glTexImage3D` operation.
- `GL_INVALID_VALUE` is generated if *level* is less than zero or greater than $\log_2 max$, where *max* is the returned value of `GL_MAX_3D_TEXTURE_SIZE_EXT`.
- `GL_INVALID_VALUE` is generated if $xoffset <$ −TEXTURE_BORDER, $(xoffset+width) >$ (TEXTURE_WIDTH−TEXTURE_BORDER),

$yoffset < -$TEXTURE_BORDER, $zoffset < -$TEXTURE_BORDER, or $(zoffset+depth) >$ (TEXTURE_DEPTH_EXT$-$TEXTURE_BORDER), where TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH_EXT and TEXTURE_BORDER are the state values of the texture image being modified. Note that TEXTURE_WIDTH, TEXTURE_HEIGHT and TEXTURE_DEPTH_EXT include twice the border width.

- GL_INVALID_ENUM is generated when *format* is not an accepted format constant.
- GL_INVALID_ENUM is generated when *type* is not a type constant.
- GL_INVALID_ENUM is generated if *type* is GL_BITMAP and *format* is not GL_COLOR_INDEX.
- GL_INVALID_OPERATION is generated if glTexSubImage3DEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

Associated Gets

glGetTexImage

glIsEnabled with argument GL_TEXTURE_3D_EXT

See Also

glDrawPixels,
glFog,
glPixelStore,
glPixelTransfer,
glTexEnv,
glTexGen,
glTexImage3D,
glTexParameter.