Silicon Graphics® Visual Workstation
OpenGL® Programming Guide
For Windows NT®

Document Number 007-3876-001

Contributors

Written by Allen Bourgoyne, Renée Bornstein, David Yu
Edited by Cindy Kleinfeld, Christina Cary, John Rosasco, Louis Gentry, and Gianpaolo Tommasi
Production by Carlos Miqueo

Contents

About This Guide xiii
Who Should Read This Book xiii
What You Should Already Know xiii
Conventions Used in This Book xiv
Fonts and Styles xiv
Code Examples xiv

1. OpenGL And WGL Extensions 1
Understanding OpenGL Extensions 1
Using OpenGL Extensions 3
Platform Independence and Header Files 3
Conditional Compilation 4
Checking the Extensions String 4
Extension Function Pointers 5
Using WGL Extensions 7
Graphics Software Development Kit (SDK) 8

2. OpenGL Features Accelerated by the Visual Workstations 9
Buffer Management 10
Color Buffer Types 10
Stencil and Depth Buffers 10
Overlay Buffer 11
Selecting Buffers for Read / Write Operations 12
Off-Screen Buffers 12

Rendering Support 14
Blending Modes 14
Texture Support 15
Specular Highlights with Textures 16
Antialiasing 16
Geometry Operations 16
Occlusion Test 17
Optimizations with the Clip Volume Hint 18
Pixel Operations 19
Interlaced Pixel Operations 19
Pixel Formats for Fast Path Transfers 19
Imaging Operations 20
Color Table 22
Convolutions 22
Scale and Bias 22
Color Matrix 22
Histogram and Min-Max 23
Extended Pixel Format Attributes 23
WGL and GL Extension List 23
Extensions by Version and Platform 23

3. Optimizing OpenGL Code for the Visual Workstations 27
General Tuning Tips 27
Making the Best of Blocking Calls 29
Use OpenGL Extensions for Optimal Performance 30
Graphics Hardware Features 31
Conclusion 34

A. Extensions API Descriptions 35
EXT_abgr 35
EXT_bgra 36
EXT_blend_color 37
EXT_blend_minmax 39
EXT_blend_subtract 41
EXT_clip_volume_hint 43
SGI_color_matrix 45
EXT_color_table 47
EXT_compiled_vertex_array 57
EXT_convolution 59
EXT_depth_pass_instrument 72
SGI_texture_lod 76
EXT_histogram 78
SGIX_interlace 88
WGL_EXT_make_current_read 89
EXT_packed_pixels 91
WGL_EXT_pbuffer 93
EXT_secondary_color 97
EXT_separate_specular_color 100
SGI_texture_edge_clamp 102

Glossary 105

Index 107

Examples

Example 1-1 Compile-Time Test for Extension Availability 4
Example 1-2 Run-Time Test for Extension Availability 5
Example 1-3 Run-Time Test for WGL Extension Availability 7
Example 2-1 Creating an Overlay Rendering Context 11
Example 2-2 Using WGL_EXT_make_current_read to Copy Between Two Drawables 12
Example 2-3 Creating and Using Pbuffers 13

Figures

Figure 2-1 Imaging Pipeline Stages with Optimized Paths 21

Tables

Table 2-1 Hardware Accelerated Pixel Formats and Types 19
Table 2-2 Extensions--OpenGL Version and Platform 24
Table 2-3 EXT_color_table 26

About This Guide

With the introduction of the Silicon Graphics 320 and the Silicon Graphics 540 Windows NT Visual Workstations, high-performance graphics features that existed only on UNIX workstations are now available for Windows NT. The Visual Workstations include new graphics features and capabilities; you can take advantage of these through extensions to OpenGL. This book describes the graphics features and capabilities of the Visual Workstations, including the new OpenGL extensions. This book is by no means a complete discussion of OpenGL or the OpenGL Windows NT (WGL) interface.

Who Should Read This Book

This book is for developers of OpenGL software who want information on the OpenGL extensions available on the Silicon Graphics 320 and Silicon Graphics 540 Windows NT Visual Workstations. It explains how you can use these extensions to enhance the capabilities of your applications, render your graphics more quickly and efficiently, and tune your code for optimal performance.

What You Should Already Know

This book assumes you have a working knowledge of basic 3D graphics concepts and techniques. In addition, you should be familiar with the OpenGL API from a programming perspective. You may want to have your OpenGL reference manual with you while reading this book.

Conventions Used in This Book

For brevity, this book refers to the Silicon Graphics 320 and the Silicon Graphics 540 Visual Workstations as “the Visual Workstations.” These are Windows NT workstations.

Fonts and Styles

This book uses the following fonts and styles:

`Fixed-width font` for literal text as you see it on-screen, including code samples.

Italics for filenames and directory paths, a word or phrase in text that has an entry in the glossary, and parameters and variables.

Bold for function names.

Code Examples

This book provides you with many examples of actual code. You can use this code in your applications.

Chapter 1

OpenGL And WGL Extensions

The OpenGL Application Programming Interface (*API*) is designed to be extensible. OpenGL extensions enable applications to take advantage of hardware capabilities that go beyond the basic features supported by all OpenGL implementations. The OpenGL extension mechanism provides a robust way for applications to check for and use the extensions that are available on a system.

The Visual Workstations' current OpenGL implementation conforms to version 1.1 of the OpenGL specification. However, many advanced features supported by the Visual Workstations are accessible only through the use of OpenGL extensions.

The Visual Workstations also support several extensions to *WGL*, the interface that connects OpenGL to the Windows NT windowing system.

This chapter explains how to write portable applications that take advantage of the extensions supported by the Visual Workstations.

Understanding OpenGL Extensions

Every OpenGL extension has an *extension name*. This name is important because it distinguishes a specific extension from other similar extensions and because it determines the *extension string* and *preprocessor symbol* that are associated with the extension. For example:

Extension name: `EXT_vertex_array`

Extension string: `GL_EXT_vertex_array`

Preprocessor symbol: `#define GL_EXT_vertex_array 1`

The first part of an extension name identifies the origin of the extension as well as the likely availability of the extension. This first part is typically a vendor-specific name, such as SGI_ for extensions defined by Silicon Graphics, or WIN_ for extensions defined by Microsoft, and so on. It may also be EXT_ to denote that an extension is supported by multiple vendors:

- `EXT_vertex_array`—defined and supported by multiple vendors
- `SGI_color_matrix`—defined by Silicon Graphics
- `WIN_swap_hint`—defined by Microsoft

In addition to the header files used to define the API for a compiler and the references pages used by programmers to find the parameters for a function, OpenGL has a specification that describes how all of the functions work together to operate on the OpenGL state machine.

Similarly, each OpenGL extension has an extension specification that describes how the extension operates and how it interacts with the rest of OpenGL.

There are several ways that OpenGL can be extended. An extension may use one or more of these, and it is useful to consider which of these are used by any given extension.

- Extend the operation of an existing OpenGL function by defining a new behavior for an existing token.

  SGI_color_matrix defines a new behavior when **GL_COLOR** is passed to **glMatrixMode()**.

- Extend the operation of an existing OpenGL function by defining a new token that adds functionality.

  EXT_bgra defines a new functionality when **GL_BGRA_EXT** is passed as the *format* parameter to **glDrawPixels()**.

- Extend the operation of OpenGL by defining a new function that may accept new or existing tokens.

  EXT_color_table defines a new function **glColorTableEXT()** that accepts a new token **GL_COLOR_TABLE_EXT.**

The new tokens and functions defined by extensions are distinguished from the standard tokens and functions by adding a suffix to their names. This suffix is the same as the first part of the extension name. See the examples above.

It is also useful to consider the ways in which extensions work like core OpenGL features.

- All states that can be specified for an extension can be queried using a **glGet*()** function.
- Most new extension states are affected by **glPushAttrib()** and **glPopAttrib()** calls.

Using OpenGL Extensions

In order to use an extension, application source code must have access to all of the new tokens and functions that are defined by the extension. This is accomplished by including the appropriate header files. There are also conventions that applications should use to test for the availability of an extension at compile time and at run time.

Platform Independence and Header Files

The standard OpenGL constants and function prototypes are defined in the file `<GL/gl.h>`. The Graphics Software Development Kit (SDK) includes the header file `<GL/glSGI.h>`, which defines all of the constants and function prototypes for extensions supported by the Visual Workstations. In addition, the SDK includes a version of the `<GL/gl.h>` header file that contains both the standard definitions and extension definitions in a single file.

There are several ways that these header files can be used:

- Include the standard version of *gl.h* then include vendor-specific header files:

```
#include <GL/gl.h>      // standard header file
#include <GL/glSGI.h>   // SGI definitions for extensions
```

- Include a vendor-specific version of `<GL/gl.h>` that contains all of the definitions for the extensions supported by a particular vendor.
- Include a site or application-specific version of `<GL/gl.h>` that contains all of definitions for extensions supported by multiple vendors.
- Include the standard version of `<GL/gl.h>`, then include a separate site-specific or application-specific header file that contains all of the definitions for extensions supported by multiple vendors.

Consider the following when deciding how to deal with extension header files:

a) Must the application support extensions from multiple vendors?

b) If so, is it acceptable to merge header files from multiple vendors?

c) Can these dependencies be isolated in an application-specific header file?

Conditional Compilation

While the header files contain all of the definitions needed to compile code that uses a particular extension, it is sometimes useful to write application code that can be compiled even when the extension definitions are not available. This can be accomplished by using preprocessor directives to compile the code only when the extension preprocessor symbol is defined.

Example 1-1 Compile-Time Test for Extension Availability

```
#if defined(GL_EXT_vertex_array)
   /* Use EXT_vertex_array if it is available */
   glVertexPointerEXT(3, GL_FLOAT, 0, 3, p);
   glEnable(GL_VERTEX_POINTER_EXT);
   glDrawArraysEXT(GL_TRIANGLES, 0, 3);
#else
   /* otherwise use the standard calls */
   glBegin(GL_TRIANGLES);
   glVertex3fv(&p[0]);
   glVertex3fv(&p[3]);
   glVertex3fv(&p[6]);
   glEnd();
#endif
```

Checking the Extensions String

Conditionally compiling code using the extension preprocessor symbols prevents an extension from being used only when it is not available at *compile-time*. Because an application may be compiled on one system, then run on a different system, it is important to make sure that an extension is supported by the OpenGL rendering context that is actually being used. The OpenGL function **glGetString()** returns a list of all of the extensions supported by a context. This list can be checked to determine extension support at *run time*.

Example 1-2 Run-Time Test for Extension Availability

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

GLboolean supports_EXT_blend_minmax;

GLboolean isSupported(const char *extension_string)
{
   /* get the list of supported extensions */
   const char *p = (const char *)
             glGetString(GL_EXTENSIONS);

   /* search for extension_string in the list */
   while (p = strstr(p, extension_string)) {
          const char *q = p + strlen(extension_string);
   /* must be terminated by <space> or <nul> */
          if (*q == ' ' || *q == '\0') {
              return GL_TRUE;
              }
   /* try to find another match */
          p = q;
   }
   return GL_FALSE;
}
   ...
/* must have a current context to call gl*() functions */
wglMakeCurrent(dc, rc);
supports_EXT_blend_minmax = isSupported("GL_EXT_blend_minmax");
```

Extension Function Pointers

On Windows NT, the functions added by an extension cannot be statically linked. Instead, an application must define a pointer to a function through which calls to the extension function are made. This function pointer must be initialized at run time using the WGL function **wglGetProcAddress()**. As a convenience, a typedef for the appropriate function pointer type is provided instead of a function prototype in the OpenGL header files.

Following is an example of initializing and using extension functions. EXT_blend_minmax defines the new function:

```
void glBlendEquationEXT(GLenum mode);
```

along with the new constants **GL_FUNC_ADD_EXT**, **GL_MIN_EXT**, **GL_MAX_EXT**, and **GL_BLEND_EQUATION_EXT**.

The header definitions for this extension are as follows:

```
/* EXT_blend_minmax */
#ifndef GL_EXT_blend_minmax
#define GL_EXT_blend_minmax                1

#define GL_FUNC_ADD_EXT                    0x8006
#define GL_MIN_EXT                         0x8007
#define GL_MAX_EXT                         0x8008
#define GL_BLEND_EQUATION_EXT              0x8009

typedef
   void (APIENTRY * PFNGLBLENDEQUATIONEXTPROC) (GLenum mode);
#endif /* GL_EXT_blend_minmax */
```

For application code to use the new function, it must declare and initialize a function pointer:

```
#include <windows.h>
#include <string.h>
#include <GL/gl.h>
#include <GL/glSGI.h>

/* declare a function pointer for EXT_blend_minmax */
PFNGLBLENDEQUATIONEXTPROC glBlendEquationEXT;

...
/* must have a current context to call wglGetProcAddress() */
glMakeCurrent(dc, rc);

supports_EXT_blend_minmax = isSupported("GL_EXT_blend_minmax");

if (supports_EXT_blend_minmax)
{
   /* initialize function pointers for EXT_blend_minmax */
   glBlendEquationEXT = (PFNGLBLENDEQUATIONEXTPROC)
       wglGetProcAddress("glBlendEquationEXT");
   ...
   /* use it */
   glBlendEquationEXT(GL_MAX_EXT);

   ...
}
```

Note that the names of the function pointers used can exactly match the name of the functions defined by the extension. Doing this can help with the readability of the code.

Using WGL Extensions

The Visual Workstations also support extensions to WGL. Just like an OpenGL extension, every WGL extension has an extension name, an extension string, a preprocessor symbol, and a specification.

The Graphics SDK provides the header file `<GL/wglSGI.h>`, which contains the definitions for all WGL extensions supported by the Visual Workstations.

Because there is no standard way to determine the list of WGL extensions that are supported at run time, there is one WGL extension that is a prerequisite for all other WGL extensions. The WGL_EXT_extensions_string extension defines the function **wglGetExtensionsStringEXT()**. This function tests for the availability of WGL extensions in the same way the **glGetString()** determines the availability of OpenGL extensions.

Example 1-3 Run-Time Test for WGL Extension Availability

```
#include <windows.h>
#include <string.h>
#include <GL/wglSGI.h>

/* declare function pointers for WGL_EXT_extensions_string */
PFNWGLGETEXTENSIONSSTRINGEXTPROC wglGetExtensionsStringEXT;

GLboolean isSupportedWGL(const char *extension_string)
{
   /* get the list of supported extensions */
   const char *p = (const char *) wglGetExtensionsStringEXT();

   /* search for extension_string in the list */

   while (p = strstr(p, extension_string)) {
          const char *q = p + strlen(extension_string);

         /* must be terminated by <space> or <nul> */
          if (*q == ' ' || *q == '\0') {
              return 1;
              }

        /* try to find another match */
          p = q;

   }
```

```
   return (GL_FALSE);
}

...
/* must have a current context to call wglGetProcAddress() */
glMakeCurrent(dc, rc);


/* initialize function pointers for WGL_EXT_extensions_string */
wglGetExtensionsStringEXT = (PFNWGLGETEXTENSIONSSTRINGEXTPROC)
          wglGetProcAddress("wglGetExtensionsStringEXT");

supports_WGL_EXT_pbuffer = isSupportedWGL("WGL_EXT_pbuffer");

...
```

Graphics Software Development Kit (SDK)

Detailed information on OpenGL extensions for the Visual Workstations is available in the Graphics Software Development Kit (SDK). You can download the SDK free of charge from:

http://www.sgi.com/developers/nt/sdk

Academic, research, corporate and in-house developers creating products for Windows NT based systems, as well as other individuals who are interested in accessing information about developing with Silicon Graphics technologies, can take advantage of the Developer Subscription and Developer On-line membership options. Developer On-line membership is free; there is no charge to download the Graphics SDK or SDK documentation.

You can find detailed information concerning the Silicon Graphics Developer's Program at the Web site listed above, or through the Developer Response Center at (800) 770-3033 or (650) 933-3033, or e-mail devprogram@sgi.com. For Europe, Africa, and the Middle East, please contact (+49-89) 46108-0, or email devprogram@europe.sgi.com.

You can file any bugs on the SDK and documentation at:

https://www.sgi.com/developers/nt/sdk/bugs.htm

Chapter 2

OpenGL Features Accelerated by the Visual Workstations

The Visual Workstations have graphics hardware that supports the standard features of OpenGL and other 3D graphics APIs. These systems also include special features that require extensions to the OpenGL for NT. This chapter discusses both the hardware features and the new extensions:

- "Buffer Management" on page 10
- "Rendering Support" on page 14
- "Geometry Operations" on page 16
- "Pixel Operations" on page 19
- "Imaging Operations" on page 20
- "Extended Pixel Format Attributes" on page 23
- "WGL and GL Extension List" on page 23
- "Extensions by Version and Platform" on page 23

Buffer Management

OpenGL applications can draw to a number of different graphics memory locations, or buffers. The Visual Workstations support several different types of buffers. This support includes new OpenGL extensions for manipulating buffers. This section describes:

- "Color Buffer Types" on page 10
- "Stencil and Depth Buffers" on page 10
- "Overlay Buffer" on page 11
- "Off-Screen Buffers" on page 12
- "Selecting Buffers for Read / Write Operations" on page 12

Color Buffer Types

The Visual Workstations support color buffers with 16 or 32 bits of depth. The 16-bit format is RGBA-5551. The 32 bit format is RGBA-8888. Single and double buffered pixel formats are available for both.

The color buffer depth can be configured using the Display control panel. Because this is a user setting, applications which require a specific color buffer depth should notify the user to update the setting accordingly.

Stencil and Depth Buffers

The Visual Workstations support either 16-bit depth buffers with no stencil bits or 24-bit depth buffers with eight stencil bits.

The stencil and depth buffer allocation can be configured using the Silicon Graphics Settings tab of the Display control panel. Because this is a user setting, applications which require a specific stencil or depth buffer configuration should notify the user to update the setting accordingly.

Overlay Buffer

The Visual Workstations have a single 8-bit, 256-color overlay buffer and use index zero as the fully transparent element in the table.

In order to draw into the overlay buffer, the proper pixel format must be selected from the device context. A pixel format which supports the required overlay attributes may be determined using the functions **DescribePixelFormat()** and **wglDescribeLayerPlane()**. Example 2-1 illustrates a method for creating an overlay rendering context.

Example 2-1 Creating an Overlay Rendering Context

```
HGLRC hRCover; // overlay render context
HDC hDC;       // hardware drawing context

hDC = GetDC(hWnd);
...
// select an overlay pixel format
...
SetPixelFormat(hDC, iPixelFormat);
hRCover = wglCreateLayerContext(hDC, 1);
wglMakeCurrent(hDC, hRCover);
```

In Example 2-1, **wglCreateLayerContext()** creates the overlay drawing context. Then the **wglMakeCurrent()** function makes the overlay window the current window into which OpenGL will draw.

Selecting Buffers for Read/Write Operations

The Visual Workstations have fast buffer-to-buffer copying capabilities. All pixel, texture, and depth buffers can be read or written at very high rates. Additionally, the WGL_EXT_make_current_read extension can be used to copy data between buffers belonging to two different drawables. Example 2-2 shows how to copy data from the color buffer of one window to the color buffer of a different window.

Example 2-2 Using WGL_EXT_make_current_read to Copy Between Two Drawables

```
// Example of make current read.
HDC ReadDC,     // HDC corresponding to the first window
    WriteDC;    // HDC corresponding to the second window
HGLRC hGLRC;
...
wglMakeContextCurrentEXT( WriteDC, ReadDC, hGLRC);
// copy w x h pixels from first to second window
glCopyPixels(0, 0, w, h, GL_COLOR);
```

Off-Screen Buffers

The Visual Workstations support the WGL_EXT_pbuffer extension. This extension can be used to create and render to off-screen pixel buffers (pbuffers). Pbuffers are analogous to regular windows with a few exceptions: they are never visible, they cannot be resized, and they are not accessible by GDI.

The functions **wglCreatePbufferEXT() and wglDestroyPbufferEXT()** are used to create and destroy pbuffers. Pbuffer attributes can be queried using the function **wglQueryPbufferEXT()**. In order to render to a pbuffer an **HDC** must be obtained. Pbuffer **HDC**s are obtained and released using the functions **wglGetPbufferDCEXT()** and **wglReleasePbufferDCEXT()**. Pbuffer **HDC**s are not compatible with the GDI **HDC**s. They can only be used with **wglMakeCurrent()** and **wglMakeContextCurrentEXT()**.

The parent **HDC** for the pbuffer must be a Window **DC**. Pbuffers use frame buffer resources. Your applications must deallocate them in order to free these resources.

Example 2-3 shows an example of using pbuffers.

Example 2-3 Creating and Using Pbuffers

```
HGLRC hGLRC;
HDC hDC;

HPBUFFEREXT hPB;
HDC hPBDC;
int attr[256];
UINT n;
int pixelFormat;

// find a pixel format that supports pbuffers
n = 0;
attr[n++] = WGL_SUPPORT_OPENGL_EXT;
attr[n++] = TRUE;
attr[n++] = WGL_DRAW_TO_PBUFFER_EXT;
attr[n++] = TRUE;
attr[n++] = WGL_RED_BITS_EXT;
attr[n++] = 1;
attr[n++] = WGL_GREEN_BITS_EXT;
attr[n++] = 1;
attr[n++] = WGL_BLUE_BITS_EXT;
attr[n++] = 1;
attr[n++] = WGL_DEPTH_BITS_EXT;
attr[n++] = 1;
attr[n++] = 0;
wglChoosePixelFormatEXT(hDC, (const int *)attr,
                            NULL, 1, &pixelFormat, &n);

// create pbuffer and pbuffer device context
hPB = wglCreatePbufferEXT(hDC, pixelFormat,
                          width, height, NULL);
hPBDC = wglGetPbufferDCEXT(hPB);

...

// render something into the pbuffer
wglMakeContextCurrentEXT(hPBDC, hPBDC, hGLRC);

...

// release pbuffer and pbuffer device context
wglReleasePbufferDCEXT(hPB, hPBDC);
```

Rendering Support

OpenGL supports a variety of methods of drawing or rendering graphics. The Visual Workstations offer a number of hardware features and extensions to OpenGL that relate to rendering graphic primitives. This section describes:

- "Blending Modes" on page 14
- "Texture Support" on page 15
- "Specular Highlights with Textures" on page 16
- "Antialiasing" on page 16

Blending Modes

OpenGL provides a mechanism to include a fourth pixel component in addition to red, green and blue. This fourth component is called "*alpha*." You can use alpha for blending effects. Use the alpha data to specify the transparency of a pixel. Using the blending mode, the source pixel's RGBA values, and the destination pixels RGBA values, you can determine the final color of the blended pixels.

OpenGL supports several blending operations.The Visual Workstations support all of the OpenGL 1.1 blending modes in addition to the following blending extensions:

- blend_color
- blend_minmax
- blend_subtract

Texture Support

The Visual Workstations support texture map sizes up to 4096 x 4096 pixels. Hardware- supported source formats are:

- RGBA-8888
- ABGR-8888
- RGBA-5551
- RGBA-4444
- A8
- I8
- LA-88
- L8

If you want to use any other format, such as YCRCB, it must first be converted to one of these formats. The Visual Workstations provide software facilities for this conversion.

The Visual Workstations utilize their unique architecture to store texture maps in host memory. Users can define how much memory is allocated for use by the graphics subsystem by adjusting the additional gfx memory setting on the Silicon Graphics Settings tab of the Display control panel. If your textures render incorrectly, as white or only partially rendered with some tiles missing, your application may have exceeded the amount of extra memory allocated to the graphics subsystem. Your application should check the OpenGL error state after defining textures with a **glTexImage*D** call. This is particularly important when creating a large number of texture objects with **glBindTexture**. In the event of a texture memory allocation failure within the OpenGL, a GL_OUT_OF_MEMORY message is posted to the error state.

Specular Highlights with Textures

The OpenGL lighting model requires you to apply lighting results to a primitive before you apply any texturing. This can lead to problems when you use specular lighting to add highlights to a model, because the texturing obscures the effect of the highlights. The Visual Workstations support the EXT_separate_specular_color extension that allows you to apply specular lighting after you apply the textures to the geometry.

Antialiasing

The Visual Workstations include hardware that accelerates high-quality line antialiasing in RGBA buffers. Antialiased lines render almost as fast as regular shaded lines as long as stippling is not enabled.

The Visual Workstations render antialiased polygons and triangles through software. This is significantly slower than rendering regular polygons.

Geometry Operations

OpenGL provides a number of basic geometric primitive operations. The Visual Workstations include specialized hardware and software extensions to OpenGL that support advanced geometric operations. This section describes:

- "Occlusion Test" on page 17
- "Optimizations with the Clip Volume Hint" on page 18

Occlusion Test

The Visual Workstations support an occlusion test extension that allows your application to determine if a given set of geometric primitives will be completely occluded when you render them. Your application can pass in any number of geometric primitives, and then read results that indicate if all fragments generated failed the depth test.

Use this feature when rendering a scene with complex geometry. Before rendering the scene, perform the occlusion test on a bounding volume first, and, if the occlusion test passes, don't render any of the complex geometry. Alternatively, you can gather occlusion information about parts of the model, and based on the result of the occlusion test, your application may choose not to render these parts in subsequent frames.

The GL_SGIX_instruments extension allows you to measure various parts of the OpenGL pipeline by sampling various implementation dependent counters. The extension defines routines to start and stop instrumentation, to specify a buffer where results are gathered asynchronously, to determine how many results have been gathered, and to poll for results.

Within this framework, the GL_SGIX_depth_pass_instrument extension allows you to count the number of fragments generated during rasterization that passed the depth test. The counter size is implementation specific (one bit on the Visual Workstations) and is clamped to the maximum value (i.e. it does not wrap).

The format of the returned data from each measurement consists of four words:

- The enum for the occlusion instrument (GL_DEPTH_PASS_INSTRUMENT_SGIX)
- The size in words of the measurement (4)
- The value of the depth pass fragment counter
- The marker passed in by the application

When you perform an occlusion test to gather information, you write the code as if you were actually going to draw the scene's geometry. Since rendering is performed as normal, you should disable writes to the color and depth buffers unless you want test geometry to be rendered in the buffers.

Optimizations with the Clip Volume Hint

During the rendering process, OpenGL applies the current modelview and projection matrices to produce clip coordinates. This transformation defines a viewing volume. The graphics hardware clips geometry that falls outside of this volume so that it is not rendered in the final scene. You can define the viewing volume with several OpenGL commands: **glFrustrum()**, **gluPerspective()** or **glOrtho()**. These functions define the left, right, top, bottom, near, and far clipping planes, which, in turn comprise the viewing volume. Some applications only render geometry that is contained within the defined viewing volume. In these applications, clip tests performed by the graphics hardware against the current viewing volume are redundant. If your application only renders geometry that is contained within the viewing volume, you can disable clip testing by using the Clip Volume Hint extension. Use the **glHint()** command with to disable clip testing by specifying *target* as **GL_CLIP_VOLUME_CLIPPING_HINT_EXT** and mode set to **GL_FASTEST**.

Be sure to use this extension only if your application renders geometry that fits within the viewing volume. If your program renders geometry that falls outside of the viewing volume, the rendering results are undefined, and can possibly cause your application to crash.

Using the Clip Volume Hint extension can give you substantial performance gains, so use the extension when you can. However, take care that geometry is rendered correctly in applications that perform their own viewing volume clipping. Some applications do perform initial viewing volume culling, but they allow end users to manipulate the scene arbitrarily without resetting the viewing volume properly. For example, an end user of such an application could zoom in past the near clipping plane. If the Clip Volume Hint extension is enabled, the zoom could cause the geometry to be rendered incorrectly, or perhaps even crash the application. You must ensure that the viewing volume is redefined whenever necessary.

Pixel Operations

OpenGL allows applications to manipulate data at the pixel level. It provides functions for a number of pixel operations, including reading, writing, copying and altering formats. The Visual Workstations include specialized hardware and software extensions that affect operations with pixels.

This section discusses:

- "Interlaced Pixel Operations" on page 19
- "Pixel Formats for Fast Path Transfers" on page 19

Interlaced Pixel Operations

The Visual Workstations support the GL_SGIX_interlace extension. When enabled, pixel writes fill every other row. Use this interlace extension to support interlaced video data.

Pixel Formats for Fast Path Transfers

The Visual Workstations can handle fast pixel reads and writes of data when the source (or destination for reads) is in one of these formats:

Table 2-1 Hardware Accelerated Pixel Formats and Types

Format	Type
GL_RGBA	GL_UNSIGNED_BYTE GL_UNSIGNED_SHORT_4_4_4_4_EXT GL_UNSIGNED_SHORT_5_5_5_1_EXT
GL_ABGR_EXT	GL_UNSIGNED_BYTE
GL_BGRA_EXT	GL_UNSIGNED_BYTE
GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE
GL_LUMINANCE	GL_UNSIGNED_BYTE
GL_COLORINDEX	GL_UNSIGNED_BYTE

Depth components transfers are hardware accelerated when the following conditions apply:

- The Visual Workstation is configured to use a 24-bit depth buffer and the *type* is **GL_UNSIGNED_INT**
- The Visual Workstation is configured to use a 16-bit depth and the type is **GL_UNSIGNED_SHORT**
- **GL_DEPTH_BIAS** is set to zero
- **GL_DEPTH_SCALE** is set to 1/256 for read operations and 256 for draw operations
- For draw operations: the depth test is enabled, the depth test is set to **GL_ALWAYS, the draw buffer is set to GL_NONE**

The hardware performs subsample packing and unpacking. The graphics hardware handles the integer, fractional, and positive and negative pixel zooms.

The hardware can also perform fast reads on both texture definition and pixel data in these formats.

Imaging Operations

OpenGL does not limit your application to 2D or 3D geometric rendering. The API provides a full set of image manipulation functionality. The Visual Workstations support the full OpenGL imaging pipeline. This “pipeline” enables your application to manipulate image data while applying a number of operations on the image data itself.

While the Visual Workstations support all of the OpenGL imaging operations, and several imaging extensions, not all of these are supported by hardware acceleration. Figure 2-1 describes the optimized paths via hardware.

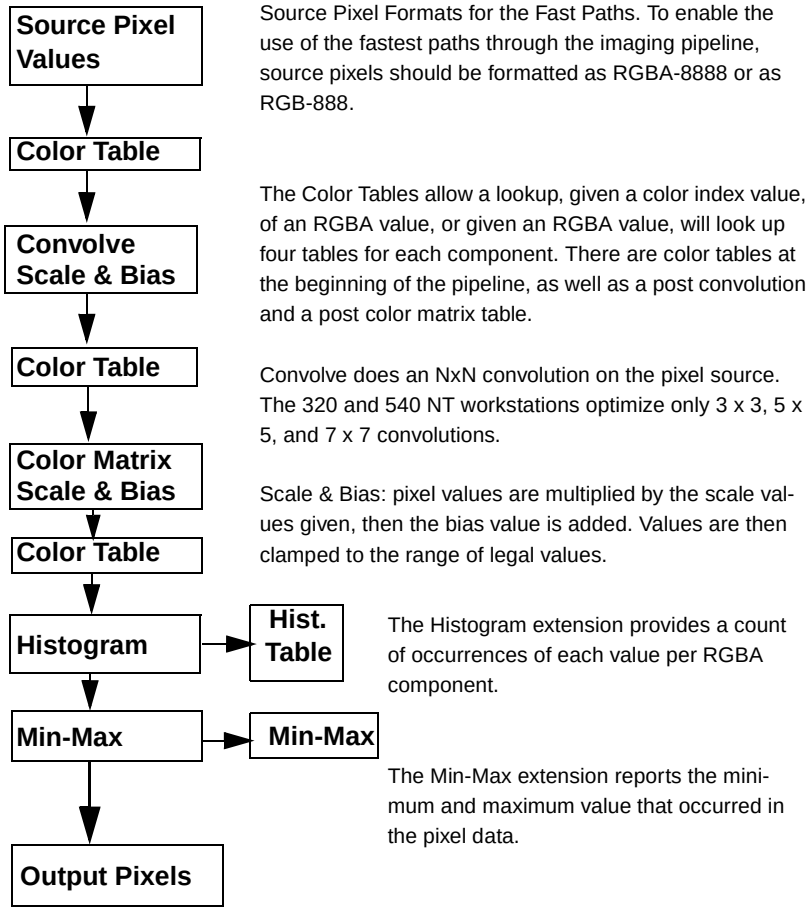

Figure 2-1 Imaging Pipeline Stages with Optimized Paths

This section describes the imaging operations in detail, and notes which ones are not implemented with hardware.

- "Color Table" on page 22
- "Convolutions" on page 22
- "Scale and Bias" on page 22
- "Color Matrix" on page 22
- "Histogram and Min-Max" on page 23

Color Table

There are color tables set at different stages of the imaging pipeline. Each color table allows the R,G,B,A component to serve as an index into a table and to replace the component with the value found in the table.

Color tables are not significantly accelerated on the Visual Workstations.

Convolutions

Convolution kernels of 3 x 3, 5 x 5, and 7 x 7 are optimized. The fast paths are enabled when the kernel values are either all integers or all fractional (between -1.0 and 1.0). The operation runs a bit faster if you don't use the pre- or post-convolve scale and bias.

Scale and Bias

You can use a scale and a bias operation following both the convolve and the color matrix step. If the scale and bias values do not generate numbers that are out of the precision range of normal pixel values (less than 256), the system uses the optimized path.

Color Matrix

The Visual Workstations have a hardware-accelerated color matrix in the pixel path. The hardware retrieves the results from earlier pixel pipeline stages or from pixel data directly from application space, and uses the color matrix. Hardware support for color matrix cannot be used when pixel pipeline stages after the color matrix are enabled for draw operations or when pixel pipeline stages before the color matrix are enabled for read operations.

Histogram and Min-Max

The Visual Workstations have no special hardware devoted to either histogram or min-max.

Extended Pixel Format Attributes

The WGL_EXT_pixel_format extension provides support for additional pixel format attributes that are not supported by the standard pixel format functions. On the Visual Workstations, this extension is used to determine which pixel formats can be used to create pbuffers.

The functions **wglGetPixelFormatAttribivEXT()** and **wglGetPixelFormatAttribfvEXT()** can be used to query the attributes of a pixel format. The function **wglChoosePixelFormatEXT()** is used to select a pixel format from the list of supported formats according to an application specified set of criteria. These functions provide a superset of the functionality provided by the functions **DescribePixelFormat()**, **DescribeLayerPlane()**, and **ChoosePixelFormat()**.

WGL and GL Extension List

This section describes all of the extensions to OpenGL and WGL. You can find complete listings of them in the table below:

Extensions by Version and Platform

Table 2-2 shows a list of extensions listed by the version of OpenGL in which they can be found, and also on which platform you can use them. The abbreviations are:

- EXT = supported by multiple vendors
- SGI = supported on all SGI platforms
- SGIS= supported on a subset of SGI platforms

- SGIX = supported on a few SGI platforms (experimental)
- WGL = extension to the WGL interface
- WIN = Microsoft extensions to the WGL interface
- 1.1 = this extension is part of the specification for OpenGL version 1.1
- 1.2 = this extension is part of the specification for OpenGL version 1.2
- * = this extension is an Silicon Graphics supported extension specific to theVisual Workstations

Detailed descriptions for all of the OpenGL 1.2 and Visual Workstation specific extensions are given in Appendix A. You can go directly to the detailed API description by clicking on the extension name in the table below.

Table 2-2 Extensions--OpenGL Version and Platform

OpenGL/WGL Extension	Specification	Classification
EXT_abgr	*	EXT
EXT_bgra	1.2	EXT
EXT_blend_color	1.2	EXT
EXT_blend_logic_op	1.1	EXT
EXT_blend_minmax	1.2	EXT
EXT_blend_subtract	1.2	EXT
EXT_clip_volume_hint	*	EXT
SGI_color_matrix	1.2	SGI
EXT_color_table	1.2	EXT
EXT_compiled_vertex_array	*	SGI
EXT_convolution	1.2	EXT
EXT_depth_pass_instrument	*	SGIX
EXT_histogram	1.2	EXT
SGIX_interlace	*	SGIX
WGL_EXT_make_current_read	*	WGL
EXT_packed_pixels	1.2	EXT

Table 2-2 Extensions--OpenGL Version and Platform (continued)

WGL_EXT_pbuffer	*	WGL
EXT_polygon_offset	1.1	EXT
EXT_secondary_color	*	EXT
EXT_separate_specular_color	1.2	EXT
EXT_subtexture	1.1	EXT
WIN_swap_hint	1.1	WIN
EXT_texture	1.1	EXT
SGI_texture_edge_clamp	1.2	SGI
SGI_texture_lod	1.2	SGI
EXT_texture_object	1.1	EXT
EXT_vertex_array	1.1	EXT

Chapter 3

Optimizing OpenGL Code for the Visual Workstations

The Visual Workstations are high-performance graphics machines capable of delivering very high levels of graphics, digital media, and raw compute performance. But in the end, it is your application code that determines what level of performance end users realize. This chapter describes how you can write code that takes advantage of the graphics capabilities of these Visual Workstations, how to optimize code so that it utilizes hardware accelerated paths through OpenGL, and how to avoid slower software paths when possible.

General Tuning Tips

This section examines some basic techniques for improving application code performance. It is by no means exhaustive, as applications vary greatly in structure and purpose.

The best way to tune an application is by applying performance analysis tools and techniques to the code in order to identify areas that are computational expensive. This section helps you write OpenGL code that performs well immediately. Later you can apply the fine tuning tools and techniques.

Use Float Format for Vertex, Normal, Color, and Texture Coordinates

When calling the vertex, normal, color, texture coordinates, and matrix definition functions, IEEE single precision floating point is the best format to use. OpenGL automatically converts any other format to single precision floating point. OpenGL geometry hardware and software paths use single precision floating point internally.

Reduce API Call Overhead

In most cases, calls to the OpenGL library result in the execution of only a small amount of software before issuing commands to the graphics hardware. In some cases, the overhead of the function calls to OpenGL is a significant portion of the total code executed with each OpenGL call. So it makes sense to eliminate any unnecessary OpenGL calls wherever possible. Some ways to reduce the amount of overhead necessary to process OpenGL calls are:

- Avoid unnecessary or no-effect calls to OpenGL
- Put as much vertex data between **glBegin()** and **glEnd()** as possible
- Use vertex arrays when possible
- Use display lists when possible

This section discusses these strategies in more detail.

- **Avoid unnecessary calls to OpenGL**

  Every call to the OpenGL library involves some performance cost, so avoid these calls whenever possible. For example, when lighting is disabled, do not pass in normal data that cause calls to the library.

  Avoid state changes to the OpenGL whenever possible. To achieve maximum performance, your application should keep track of the GL state rather than make unnecessary calls to **glEnable**, **glDisable**, **glGet***, or **glPushAttrib** / **glPopAttrib**.

- **Put vertex data between glBegin() and glEnd()**

  The OpenGL library is tuned to process vertex and associated data quickly. It generates internal library software paths that are tuned for the particular geometry types and mode settings. By processing large amounts of vertex and related data between **glBegin()** and **glEnd()** pairs, the performance gains justify the extra set up time required to generate these internal paths.

  In addition, by using many vertices in a connected or “strip” primitive, you can significantly reduce the amount of processing needed to perform the transform, clip check, and lighting calculations. For example, think of a triangle strip. If you use three vertices for one triangle between a **glBegin** / **glEnd** pair, OpenGL processes three vertices to create the triangle. If you have a long triangle strip, it still takes three vertices to process the first triangle, but the system generates additional triangles with only one additional vertex, so the processing cost is reduced by nearly two thirds.

Use vertex arrays when possible

Vertex arrays offer an opportunity for applications to process large amounts of geometry with very few OpenGL API calls. Performance improves since there are fewer OpenGL calls made, reducing the overall cost per vertex and per geometric primitive. The vertex array implementation supports interleaved floating point formats. All of the formats for interleaved arrays are optimized except formats that contain four dimensional texture coordinates.

- **Use display lists when possible**

  Display lists also allow an application to issue large amounts of geometry data with only one OpenGL API call. However, display lists do take up user memory space to store data, so be careful not to consume memory in a wasteful manner when using them.

  The Visual Workstations can automatically processes display lists to see if any clipping extensions can be used. Display lists can also be processed and compiled into formats suitable for further accelerations. The optimal candidates for this kind of acceleration are display lists that are “flat”; that is, they do not call other lists, contain only geometric data, have no matrix operations, and contain no state changes.

  Use display lists whenever possible, but again, take into account memory considerations. Lists that get swapped out by the NT virtual memory manager have to be swapped back in when called, and this reduces performance.

Making the Best of Blocking Calls

With regard to the GL state, the Visual Workstation OpenGL implementation has a combination of asynchronous and synchronous calls. Synchronous calls such as **glDrawPixels**, **glReadPixels**, and **glTexImage** have an implicit finish command that requires all pending GL calls to be executed before these synchronous calls are executed. Synchronous calls return control to the application only when all effects of the command on the GL client and server state, as well as any framebuffer results, are fully realized.

Your application should try to leverage this inherent latency by issuing any asynchronous non-GL calls before calling a synchronous GL call. In doing so, your application may be able to take advantage of any inherent parallelism that may exist.

Use OpenGL Extensions for Optimal Performance

Most OpenGL extensions expose new functionality beyond that of the existing OpenGL specification, but are not limited to new graphics features only. The Visual Workstations offer several extensions that directly improve application software performance. The extensions for Clip Volume Hint, Compiled Vertex Array, and Occlusion Test are examples of such extensions. This section describes these extensions in detail to illustrate how they improve performance.

Clip Volume Hint Extension

Chapter 2 discusses the Clip Volume Hint extension in detail. In review, applications that perform their own geometry culling with respect to the viewing volume can use the Clip Volume Hint extension. If the application culls geometry so that no rendered geometry lies outside of the viewing volume, the Clip Volume Hint call tells the graphics hardware that no additional checks are required. This significantly improves performance.

Depth Pass Instrument Extension

The Occlusion Test Instrument Extension allows an application to determine during the rendering of a scene if, based on what is currently in the depth buffer, a volume of geometry is visible. Your application can use this information to determine if objects are visible in a scene. This way, your application can choose not to draw objects that are not visible. In complex scenes, containing large amounts of geometry, you can greatly improve performance by using the Occlusion Test Instrument Extension.

This extension also allows your application to send a bounding volume to the graphics hardware. The test returns a result that indicates if any pixels of that volume were visible (any pixel's depth test passed). By testing a simple bounding volume first, your application can potentially avoid drawing the more complicated geometry contained in that volume. The cost of the test is the amount of time required to render the test volume. Your application should disable textures, lighting and shading, since they are not relative to the depth test results. The test is non-blocking; you can send the test, and your application continues to execute and use the graphics hardware. Your application needs to supply a buffer so that OpenGL can asynchronously write the results of the occlusion tests to that buffer. The extension provides routines to check the number of results written back and to poll for a specific test result. If your application draws large amounts of geometric data, you should use this extension whenever possible, as it can improve your the performance significantly.

Graphics Hardware Features

This section discusses the Visual Workstations' graphics hardware features. The hardware implements much of the OpenGL functionality. Some operations require software assistance, and a few need complete software implementations.

Features Directly Supported by the Graphics Hardware

Direct hardware-supported features provide the fastest path through the graphics pipeline. Keeping your application code on this *fast path* results in the highest levels of performance available from the graphics hardware. If graphics performance is critical to your application, then you should write your OpenGL code so that it exercises the hardware supported features as much as possible. Direct hardware supported features include:

- Rasterization features supported in hardware
  - Color shading
  - Alpha blending, including all blending extensions
  - Logic ops, including for RGB mode
  - Byte masks (for geometry and pixel operations)
  - Z-buffer and stencil operations
  - Texturing: including nearest, linear, mimap nearest, mimap linear, wrapping, clamping; but not textures with borders
  - Texture perspective correction
  - Fog, but not per-pixel fogging
  - Polygon stipples (only if the pattern is a repeatable 8x8 subpattern of the 32x32 pattern)
  - Secondary color extension support

- Geometry support in hardware
  - Points, lines: independent, strips, and loops;
  - Triangles: strips, fans, and independent
  - Polygon offset (only polygons, not points and lines)
  - Back and front face culling
  - Antialiased RGB lines (not polygons or triangles)
  - Occlusion test (depth pass test)
  - Viewport transform
- Lighting support in hardware
  - Up to 4 infinite light sources accelerated, including 2-sided lighting
  - Normal transforms and normalization
- Pixel operation support in hardware:
  - Fast bitmap rendering
  - Hardware 4x4 color matrix and scale / bias operations
  - All zoom factors: integer, fractional, negative
  - Fogging
  - Alpha blending
  - Z-test and stencil test
  - Byte masking
  - Fast read dest / write source formats: RGBA / ABGR 8888, RGBA 4444, RGB 555, LA 88, L8, I8, CI 88, YCrCb 4:2:2, YCrCb 4:4:4
  - Fast RGB 888 Format supported for writes (without zoom)
  - Interlaced pixels
  - Z-buffer reads / writes with 24 bit data (must disable color plane updates via setting color masks to all 0's or **glDrawBuffer(GL_NONE)**)
  - Texture definitions, with source data in any of the supported pixel formats

- Window and buffer support in hardware
  - Off-screen rendering with pbuffers
  - Fast buffer to buffer pixel copies
  - Fast buffer to texture copies
  - Scissoring

You can get the most performance from your code if you use only these features, thereby taking the fastest path through the graphics hardware pipeline.

Software-Assisted Features

Some operations require software support prior to sending the commands to the graphics hardware. While these features perform extremely well on the Visual Workstations, using them results in slightly lower overall performance, as compared to applications that stay completely on the fast path. However, if your application has a larger demand for quality than for raw speed, you may want to use these features.

Features that require some software support include:

- Local lights & spot lights
- Texgen
- User-defined clip-planes
- Texture matrix
- Color material
- Post texture specular highlight
- Wide stippled lines
- Wide antialiased lines
- Polygon offset when applied to points and lines generated with Polygon Mode
- More than 4 lights
- Simultaneously rendering to the front & back buffers (i.e. glDrawMode(GL_FRONT_AND_BACK))
- Polygon mode not GL_FILL
- Colorindexed lighting

Features That Require Software Implementation

There are a small number of modes that require a software renderer perform the actual OpenGL rendering. Use these features if your application requires better visual realism. However, these features incur significant costs, so use them sparingly if graphics performance is the most important aspect of your application. Features that require software rendering include:

- Polygon stipples that do not fit into an 8x8 repeatable pattern
- Antialiased polygons
- Wide RGB antialiased lines
- Wide antialiased points
- Color indexed antialiased lines

Conclusion

Silicon Graphics continually strives to provide the latest graphics hardware and software to the workstation marketplace. The OpenGL feature set and capabilities of the visual workstations will grow over time. This document will be updated as necessary in order to provide information on any new OpenGL features or any updates to existing OpenGL features. Readers of this guide should visit the Developer's Program website mentioned in Chapter 1 on a regular basis in order to obtain the latest updates to this document.

Appendix A

Extensions API Descriptions

EXT_abgr

The GL_EXT_abgr extension extends the list of host-memory color formats. Specifically, it provides a reverse-order alternative to the RGBA image format.

EXTENSION STRING NAME

GL_EXT_abgr

NEW TOKENS

Accepted by the <format> parameter of glDrawPixels, glGetTexImage, glReadPixels, glTexImage1D, glTexImage2D:

GL_ABGR_EXT

SEE ALSO

glDrawPixels, glReadPixels

EXT_bgra

The GL_EXT_bgra extension extends the list of host-memory color formats. Specifically, it provides a different order alternative to the rgba image format.

EXTENSION STRING NAME

GL_EXT_bgra

PROCEDURE DEFINITIONS

none

NEW TOKENS

Accepted by the <format> parameter of glDrawPixels, glGetTexImage, glReadPixels, glTexImage1D, and glTexImage2D:

GL_BGR_EXT
GL_BGRA_EXT

SEE ALSO

glDrawPixels, glReadPixels

EXT_blend_color

The glBlendColorEXT function sets a constant blend color that can be included in blending equations.

EXTENSION STRING

GL_EXT_blend_color

PROCEDURE DEFINITIONS

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

PARAMETERS

red	Specifies the red color component of the constant blend color
green	Specifies the green color component of the constant blend color
blue	Specifies the blue color component of the constant blend color
alpha	Specifies the alpha component of the constant blend color

NEW TOKENS

Accepted by the <*sfactor*> and <*dfactor*> parameters of glBlendFunc:

GL_CONSTANT_COLOR_EXT
GL_ONE_MINUS_CONSTANT_COLOR_EXT
GL_CONSTANT_ALPHA_EXT
GL_ONE_MINUS_CONSTANT_ALPHA_EXT

Accepted by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

BLEND_COLOR_EXT

DESCRIPTION

The GL_BLEND_COLOR_EXT may be used to calculate the source and destination blending factors. See glBlendFunc for a complete description of the blending operations. Initially the GL_BLEND_COLOR_EXT is set to (0,0,0,0)

NOTES

glBlendColorEXT is part of the EXT_blend_color extension, not part of the core GL command set. If GL_EXT_blend_color is included in the string returned by glGetString, when called with argument GL_EXTENSIONS, extension EXT_blend_color is supported by the connection.

ERRORS

GL_INVALID_OPERATION is generated if glBlendColorEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

ASSOCIATED GETS

glGet* with an argument of GL_BLEND_COLOR_EXT.

EXT_blend_minmax

Blending capabilities are extended by re-specifying the entire blend equation.

EXTENSION STRING

GL_EXT_blend_minmax

PROCEDURE DEFINITIONS

```
void glBlendEquationEXT(enum mode)
```

PARAMETERS

mode One of the blending equation tokens defined below.

NEW TOKENS

Accepted by the <*mode*> parameters of glBlendEquationEXT:

GL_FUNC_ADD_EXT
GL_MIN_EXT
GL_MAX_EXT

Accepted by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

GL_BLEND_EQUATION_EXT

DESCRIPTION

The OpenGL specification defines a single blending equation. The blend equation extension introduces a blend equation mode that is specified by calling **glBlendEquationEXT** with one of three enumerated values. The default value, FUNC_ADD_EXT specifies that the blending equation defined in the OpenGL specification be used. This equation is:

$C' = (Cs * S) + (Cd * D)$

Where *Cs* and *Cd* are the source and destination colors, and *S* and *D* are specified by **glBlendFunc**.

If **glBlendEquationEXT** is called with <*mode*> set to GL_MIN_EXT, the blending equation becomes:

C = min(Cs, Cd)

If **glBlendEquationEXT** is called with <*mode*> set to GL_MAX_EXT, the blending equation becomes:

C= max(Cs, Cd)

In all cases, the blending equation is evaluated separately for each color component.

ERRORS

GL_INVALID_ENUM is generated if **glBlendEquationEXT** is called with a parameter that is not GL_FUNC_ADD_EXT, GL_MIN_EXT, or GL_MAX_EXT.

GL_INVALID_OPERATION is generated if **glBlendEquationEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

ASSOCIATED GETS

Calling **glGetIntegerv** with an argument of GL_BLEND_EQUATION_EXT returns the current blending mode.

SEE ALSO

blendSubtract

EXT_blend_subtract

Blending capabilities are extended by re-specifying the entire blend equation.

EXTENSION STRING

GL_EXT_blend_subtract

PROCEDURE DEFINITIONS

```
void glBlendEquationEXT(enum mode)
```

PARAMETERS

mode One of the blending equation tokens defined below.

NEW TOKENS

Accepted by the <*mode*> parameters of glBlendEquationEXT:

GL_FUNC_SUBRACT_EXT
GL_FUNC_REVERSE_SUBTRACT_EXT

Accepted by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

GL_BLEND_EQUATION_EXT

DESCRIPTION

Two additional blending equations are specified using the interface defined by EXT_blend_minmax. These equations are similar to the default blending equation, but produce the difference of its left and right hand sides, rather than the sum. Image differences are useful in many image processing applications. If the glBlendEquationEXT is called with <mode> set to GL_FUNC_SUBTRACT_EXT, the blending equation becomes:

$C' = (Cs * S) - (Cd * D)$

Where *Cs* and *Cd* are the source and destination colors, and *S* and *D* are specified by **glBlendFunc**.

If **glBlendEquationEXT** is called with <*mode*> set to GL_FUNC_REVERSE_SUBTRACT_EXT, the blending equation becomes:

*C' = (Cd * D) - (Cs * S)*

In all cases, the blending equation is evaluated separately for each color component.

ERRORS

GL_INVALID_ENUM is generated if **glBlendEquationEXT** is called with a parameter that is not GL_FUNC_ADD_EXT, GL_MIN_EXT, GL_MAX_EXT, GL_FUNC_SUBTRACT_EXT, GL_FUNC_REVERSE_SUBTRACT_EXT.

GL_INVALID_OPERATION is generated if **glBlendEquationEXT** is executed between the execution of **glBegin** and the corresponding execution of **glEnd**.

ASSOCIATED GETS

Calling **glGetIntegerv** with an argument of GL_BLEND_EQUATION_EXT returns the current blending mode.

SEE ALSO

blendMinMax

EXT_clip_volume_hint

The clip volume hint enables applications that perform clipping of geometry to the viewing volume internally to direct the graphics hardware that volume clip checks are not necessary for the geometry being passed to it.

NAME STRINGS

GL_EXT_clip_volume_hint

NEW TOKENS

Accepted by the <target> parameter of glHint:

GL_CLIP_VOLUME_HINT_EXT

DESCRIPTION

The clip volume hint extension provides a mechanism for applications to disable hardware clipping tests against the currently defined viewing volume.

To enable the clip volume hint:

```
void glHint(GL_CLIP_VOLUME_HINT_EXT, GL_FASTEST);
```

To disable the clip volume hint:

```
void glHint(GL_CLIP_VOLUME_HINT_EXT, GL_DONT_CARE);
```

If the application code passes down geometry data that has not been clipped with respect to the current viewing volume, the results are undefined.

NOTES

Special consideration must be given to geometry that lies extremely close to the clipping planes. Depth buffer values are 24-bit values. Applications that utilize 32 or 64-bit floating point values for extreme precision may experience precision problems as higher precision values may not be noticeable when converted to 24-bit precision. Application code that use local or spot lighting, user defined clip planes, vertices with four coordinates or any other OpenGL mode that forces the

transform to eye coordinates to perform any operations will not benefit from this extension as the graphics hardware will not permit it to be enabled under these conditions.

ERRORS

GL_INVALID_ENUM is generated if either parameter is an invalid value.

GL_INVALID_OPERATION is generated if **glHint** is called between the execution of a **glBegin** and the corresponding **glEnd** function.

SEE ALSO

The *OpenGL Programming Guide for Silicon Graphics Windows NT Visual Workstations* available on the Silicon Graphics Inc. Graphics Software Developer's Kit (SDK).

SGI_color_matrix

The color matrix extension adds a color matrix stack to the pixel transfer path.

EXTENSION STRING

GL_SGI_color_matrix

NEW TOKENS

Accepted by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

GL_COLOR_MATRIX_SGI
GL_COLOR_MATRIX_STACK_DEPTH_SGI
GL_MAX_COLOR_MATRIX_STACK_DEPTH_SGI

Accepted by the <pname> parameter of glPixelTransfer*, and by the <pname> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

GL_POST_COLOR_MATRIX_RED_SCALE_SGI
GL_POST_COLOR_MATRIX_GREEN_SCALE_SGI
GL_POST_COLOR_MATRIX_BLUE_SCALE_SGI
GL_POST_COLOR_MATRIX_ALPHA_SCALE_SGI
GL_POST_COLOR_MATRIX_RED_BIAS_SGI
GL_POST_COLOR_MATRIX_GREEN_BIAS_SGI
GL_POST_COLOR_MATRIX_BLUE_BIAS_SGI
GL_POST_COLOR_MATRIX_ALPHA_BIAS_SGI

DESCRIPTION

This extension adds a 4x4 matrix stack to the pixel transfer path. The matrix operates on RGBA pixel groups using the equation:

C’ = MC

where

| R |
C = | G |
| B |
| A |

and M is the 4x4 matrix on the top of the color matrix stack. After the matrix multiplication, each resulting color component is scaled and biased by a programmed amount. Color matrix multiplication follows the convolution (and the scale and bias that are associated with the convolution).

ERRORS

None

ASSOCIATED GETS

The following glGet* calls are supported:

glGet Command	Argument	Returned Data
glGetFloatv	GL_COLOR_MATRIX_SGI	current color matrix
glGetIntegerv	GL_COLOR_MATRIX_STACK_DEPTH_SGI	depth of matrix stack
glGetFloatv	GL_POST_COLOR_MATRIX_RED_SCALE_SGI	red scale value
glGetFloatv	GL_POST_COLOR_MATRIX_GREEN_SCALE_SGI	green scale value
glGetFloatv	GL_POST_COLOR_MATRIX_BLUE_SCALE_SGI	blue scale value
glGetFloatv	GL_POST_COLOR_MATRIX_ALPHA_SCALE_SGI	alpha scale value
glGetFloatv	GL_POST_COLOR_MATRIX_RED_BIAS_SGI	red bias value
glGetFloatv	GL_POST_COLOR_MATRIX_GREEN_BIAS_SGI	green bias value
glGetFloatv	GL_POST_COLOR_MATRIX_BLUE_BIAS_SGI	blue bias value
glGetFloatv	GL_POST_COLOR_MATRIX_ALPHA_BIAS_SGI	alpha bias value
glGetIntegerv	GL_MAX_COLOR_MATRIX_STACK_DEPTH_SGI	maximum color stack value

EXT_color_table

This extension defines a new RGBA-format color lookup mechanism. It does not replace the color lookups defined by the OpenGL specification, but rather provides additional lookups with different operation. The key difference is that the new lookup tables are treated as 1-dimensional images with internal formats, like texture images and convolution filter images. From this follows the fact that the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format ALPHA modifies only the A component of each pixel group, leaving the R, G, and B components unmodified.

EXTENSION STRING

GL_EXT_color_table

PROCEDURE DEFINITIONS

```
void glColorTableEXT(enum target
             enum internalformat,
             sizei width,
             enum format,
             enum type,
             const void *table);

void glCopyColorTableEXT(enum target,
             enum internalformat,
             int x,
             int y,
             sizei width);

void glColorTableParameterivEXT(enum target,
             enum pname,
             const int *params);

void glGetColorTableEXT(enum target,
             enum format,
             enum type,
             void * table);

void glGetColorTableParamterivEXT(enum target,
             enum pname,
             int * table);
```

```
void glGetColorTableParamterfvEXT(enum target,
             enum pname,
             float * params);
```

PARAMETERS

glColorTableEXT

target	Type of color table defined by one of the new color table type tokens defined below
internal format	One of the values accepted by the <components> parameter of TexImage2D, or 1, 2, 3, or 4.
width	Number of entries in the color lookup table
format	One of the supported pixel format enum types
type	Data type of table entries
table	up table

glCopyColorTableEXT

target	Type of color table defined by one of the new color table type tokens defined below
internal format	One of the values accepted by the <components> parameter of TexImage2D, or 1, 2, 3, or 4.
x, y	Lower left window coordinate of image data
width	Width in pixels of image data

glColorTableParameterivEXT

target	Type of color table defined by one of the new color table type tokens defined below
pname	Specifies color table parameters defined below
params	Data for the corresponding parameter types listed in the pname parameter

glColorTableParameterfvEXT

target	Type of color table defined by one of the new color table type tokens defined below
pname	Specifies color table parameters defined below
params	Data for the corresponding parameter types listed in the pname parameter

glGetColorTableEXT

target	Type of color table defined by one of the new color table type tokens defined below
format	One of the supported pixel format enum types
type	Data type of table entries
table	Buffer for color lookup table

glGetColorTableParameterivEXT

target	Type of color table defined by one of the new color table type tokens defined below
pname	Specifies color table parameters defined below
params	Buffer to hold returned values for the corresponding parameter types listed in the pname parameter

glGetColorTableParameterfvEXT

target	Type of color table defined by one of the new color table type tokens defined below
pname	Specifies color table parameters defined below
params	Buffer to hold returned values for the corresponding parameter types listed in the pname parameter

NEW TOKENS

Accepted by the <*cap*> parameters of **glEnable, glDisable**, and **glIsEnabled**, by the <*pname*> parameter of **glGetBooleanv**, **glGetIntegerv**, **glGetFloatv**, and **glGetDoublev**, and by the <*target*> parameter of **glColorTableEXT**, **glCopyColorTableEXT**, **glColorTableParameterivEXT**, **glColorTableParameterfvEXT**, **glGetColorTableEXT**, **glGetColorTableParameteriv**, and **glGetColorTableParameterfvEXT**:

GL_COLOR_TABLE_EXT
GL_POST_CONVOLUTION_COLOR_TABLE_EXT
GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT

Accepted by the <*target*> parameter of **glColorTableEXT**, **glGetColorTableivEXT**, and **glGetColorTableParameterfvEXT**:

GL_PROXY_COLOR_TABLE_EXT
GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT
GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXT

Accepted by the <*pname*> parameter of **glColorTableParameterivEXT**, **glColorTableParameterfvEXT**, **glGetColorTableParameterivEXT**, and **glGetColorTableParameterfvEXT**:

GL_COLOR_TABLE_SCALE_EXT
GL_COLOR_TABLE_BIAS_EXT

Accepted by the <*pname*> parameter of **glGetColorTableParameterivEXT**, and **glGetColorTableParameterfvEXT**:

GL_COLOR_TABLE_FORMAT_EXT
GL_COLOR_TABLE_WIDTH_EXT
GL_COLOR_TABLE_RED_SIZE_EXT
GL_COLOR_TABLE_GREEN_SIZE_EXT
GL_COLOR_TABLE_BLUE_SIZE_EXT
GL_COLOR_TABLE_ALPHA_SIZE_EXT
GL_COLOR_TABLE_LUMINANCE_SIZE_EXT
GL_COLOR_TABLE_INTENSITY_SIZE_EXT

DESCRIPTION

A color lookup table is specified using the **glColorTableEXT** command. Its <*target*> parameter must be **GL_COLOR_TABLE_EXT**, **GL_POST_CONVOLUTION_COLOR_TABLE_EXT**, or **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT** if a color lookup table is to be specified. (Optional target values **GL_PROXY_COLOR_TABLE_EXT**, **GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT**, and **GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXT** are described below.) Its <*width*> parameter specifies the number of entries in the color lookup table, and its <*internalformat*> parameter specifies the format of each table entry.

If no error results from the execution of **glColorTableEXT**, the specified color lookup table is redefined to have <*width*> entries, each with the specified internal format. The entries are indexed as zero through N-1, where N is the <*width*> of the table. The values in the previous color lookup table, if any, are lost. The new values are specified by the contents of the 1-dimensional image pointed to by <*table*>, whose memory format and data type are specified by <*format*> and <*type*>. The specified image is extracted from memory and processed just as if **glDrawPixels** were called, stopping after the final expansion to RGBA step. The R, G, B, and A components of each pixel are then scaled by the four **GL_COLOR_TABLE_SCALE_EXT** parameters, then biased by the four **GL_COLOR_TABLE_BIAS_EXT** parameters. The R, G, B, and A values are then clamped to [0,1]. The scale and bias parameters are specified by calling **glColorTableParameterivEXT** or **glColorTableParameterfvEXT** with <*target*> specifying one of the three color tables (**GL_COLOR_TABLE_EXT, GL_POST_CONVOLUTION_COLOR_TABLE_EXT,** or **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT**),<*pname*> **GL_COLOR_TABLE_SCALE_EXT** or **GL_COLOR_TABLE_BIAS_EXT,** and <*params*> pointing to a vector of four values: red, green, blue, and alpha, in that order.

If EXT_copy_texture is supported, color tables can also be defined using image data in the framebuffer. **glCopyColorTableEXT** accepts image data from a <*width*> pixel wide by one pixel high color buffer region whose lower-left pixel has window coordinates <*x*>,<*y*>. If any pixels within this region are outside the window that is associated with the OpenGL context, the values obtained for those pixels are undefined.

These pixel values are obtained from the framebuffer exactly as if **glReadPixels** had been called with <*format*> set to RGBA, with processing continuing through Conversion of RGBA values. At this point all pixel component values are treated

exactly as if **glColorTableEXT** had been called, beginning with the scaling of the color components by **GL_COLOR_TABLE_SCALE_EXT**.

Three lookup tables exist at different points along the pixel processing path. **GL_COLOR_TABLE_EXT** is located immediately after the subsection "Index Lookup", and immediately prior to the convolution operation. **GL_POST_CONVOLUTION_COLOR_TABLE_EXT** is located immediately after the convolution operation (including its scale and bias operations), and immediately prior to the color matrix operation. Finally, **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT** is located immediately after the color matrix operation (including its scale and bias operations) and immediately prior to the histogram operation. Color tables are enabled and disabled by calling **glEnable** and **glDisable** with color table name passed as the <*cap*> parameter. Color table lookup is performed only for RGBA groups, though these groups may have been specified as color indexes and converted to RGBA by index-to-RGBA pixel map table. When enabled, a color lookup table is applied to all RGBA pixel groups, regardless of the command that they are associated with. These commands are **glDrawPixels**, **glCopyPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, and **glGetTexImage**.

Alternate sets of partial color lookup table state are defined for the proxy tables **GL_PROXY_COLOR_TABLE_EXT,**
GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT, and
GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXTWIDTH_EXT.
Specifically**, GL_COLOR_TABLE_FORMAT_EXT,**
GL_COLOR_TABLE_WIDTH_EXTSIZE_EXT,
GL_COLOR_TABLE_RED_SIZE_EXT,
GL_COLOR_TABLE_GREEN_SIZE_EXT,
GL_COLOR_TABLE_BLUE_SIZE_EXT,
GL_COLOR_TABLE_ALPHA_SIZE_EXT,
GL_COLOR_TABLE_LUMINANCE_SIZE_EXT, and
GL_COLOR_TABLE_INTENSITY_SIZE_EXT are maintained for proxy tables. When **glColorTableEXT** is called with <*target*> set to one of these proxy values, these proxy state values are always recomputed and updated, even if the color table is too large to actually be defined. If the color table is too large, all of these state variables are set to zero. If the color table could be accommodated by **glColorTableEXT** called with <*target*> set to the corresponding non-proxy target, these values are set as though that target were being defined. (**GL_COLOR_TABLE_EXT** is the non-proxy target corresponding to proxy target **GL_PROXY_COLOR_TABLE_EXT**, for example.) All of these state values can be queried with **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** with <*target*> set to the appropriate proxy

target. Calling **glColorTableEXT** with a proxy <*target*> has no effect on the image or state of any actual color table.

There is no image associated with any of the proxy targets. Therefore **GL_PROXY_COLOR_TABLE_EXT**, **GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT**, and **GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXT** cannot be used as color tables, and their images must never be queried using **glGetColorTableEXT**.

Integer and floating point query functions **glGetColorTableParameterivEXT** and **glGetColorTableParameterfvEXT** are provided. The <*target*> parameter must be **GL_COLOR_TABLE_EXT**,

GL_POST_CONVOLUTION_COLOR_TABLE_EXT, GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT, GL_PROXY_COLOR_TABLE_EXT, GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT, or **GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXT**. The <*pname*> parameter is one of **GL_COLOR_TABLE_SCALE_EXT, COLOR_TABLE_BIAS_EXT, GL_COLOR_TABLE_FORMAT_EXT**, **GL_COLOR_TABLE_WIDTH_EXT, GL_COLOR_TABLE_RED_SIZE_EXT, GL_COLOR_TABLE_GREEN_SIZE_EXT, GL_COLOR_TABLE_BLUE_SIZE_EXT**, **GL_COLOR_TABLE_ALPHA_SIZE_EXT**, **GL_COLOR_TABLE_LUMINANCE_SIZE_EXT,** or **GL_COLOR_TABLE_INTENSITY_SIZE_EXT**. The value of the specified parameter is returned in <*params*>.

The current contents of a color table are queried using **glGetColorTableEXT**. The <*target*> parameter must be **GL_COLOR_TABLE_EXT, GL_POST_CONVOLUTION_COLOR_TABLE_EXT,** or **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT**. The <*format*> parameter must be one of **RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE**, or **LUMINANCE_ALPHA**. The <*type*> parameter must be **UNSIGNED_BYTE, BYTE, UNSIGNED_SHORT, SHORT, UNSIGNED_INT, INT**, or **FLOAT**. The 1-dimensional color table image is returned to the <*table*> parameter. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **glReadPixels** are performed. Color components that are requested in the specified <*format*>, but which are not included in the internal format of the color lookup table, are returned as zero.

ERRORS

INVALID_ENUM is generated if **glColorTableEXT** parameter <*target*> is not **GL_COLOR_TABLE_EXT**, **GL_POST_CONVOLUTION_COLOR_TABLE_EXT**, **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT**, **GL_PROXY_COLOR_TABLE_EXT**,**GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT**, or **GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXT**.

INVALID_ENUM is generated if **glColorTableEXT** parameter <*internalformat*> is not **ALPHA, RGB, RGBA, LUMINANCE, LUMINANCE_ALPHA**, or one of the tokens defined by EXT_texture.

INVALID_VALUE is generated if **glColorTableEXT** parameter <*width*> is not zero or a non-negative power of two.

INVALID_ENUM is generated if **glColorTableEXT** parameter <*format*> is not **RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE,** or **LUMINANCE_ALPHA.**

INVALID_ENUM is generated if **glColorTableEXT** parameter <*type*> is not **BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT,** or **FLOAT**.

INVALID_ENUM is generated if **glCopyColorTableEXT** parameter <*target*> is not **GL_COLOR_TABLE_EXT**, **GL_POST_CONVOLUTION_COLOR_TABLE_EXT**, or **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT**.

INVALID_ENUM is generated if **glCopyColorTableEXT** parameter <*internalformat*> is not **ALPHA**, **RGB, RGBA, LUMINANCE, *LUMINANCE_ALPHA,*** or one of the tokens defined by EXT_texture.

INVALID_VALUE is generated if **glCopyColorTableEXT** parameter <*width*> is not zero or a non-negative power of two

INVALID_ENUM is generated if **glGetColorTableEXT** parameter <*target*> is not **GL_COLOR_TABLE_EXT, GL_POST_CONVOLUTION_COLOR_TABLE_EXT,** or **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT**.

INVALID_ENUM is generated if **glGetColorTableEXT** parameter <*format*> is not **RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE,** or **LUMINANCE_ALPHA.**

INVALID_ENUM is generated if **glGetColorTableEXT** parameter <*type*> is not **BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT,** or **FLOAT**.

INVALID_ENUM is generated if **glColorTableParameterivEXT** or **glColorTableParameterfvEXT** parameter <*target*> is not **GL_COLOR_TABLE_EXT, GL_POST_CONVOLUTION_COLOR_TABLE_EXT,** or **GL_POST_C0LOR_MATRIX_COLOR_TABLE_EXT.**

INVALID_ENUM is generated if **glColorTableParameterivEXT** or **glColorTableParameterfvEXT** parameter <*pname*> is not **GL_COLOR_TABLE_SCALE_EXT** or **GL_COLOR_TABLE_BIAS_EXT**.

INVALID_ENUM is generated if **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** parameter <*target*> is not **GL_COLOR_TABLE_EXT, GL_POST_CONVOLUTION_COLOR_TABLE_EXT, GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT, GL_PROXY_COLOR_TABLE_EXT**, **GL_PROXY_POST_CONVOLUTION_COLOR_TABLE_EXT**, or **GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE_EXT**.

INVALID_ENUM is generated if **glGetColorTableParameterivEXT** or**glGetColorTableParameterfvEXT** parameter <*pname*> is not **GL_COLOR_TABLE_SCALE_EXT, GL_COLOR_TABLE_BIAS_EXT**, **GL_COLOR_TABLE_FORMAT_EXT**, **GL_COLOR_TABLE_WIDTH_EXT**, **GL_COLOR_TABLE_RED_SIZE_EXT**, **GL_COLOR_TABLE_GREEN_SIZE_EXT**, **GL_COLOR_TABLE_BLUE_SIZE_EXT**, **GL_COLOR_TABLE_ALPHA_SIZE_EXT**, **GL_COLOR_TABLE_LUMINANCE_SIZE_EXT**, or **GL_COLOR_TABLE_INTENSITY_SIZE_EXT**.

INVALID_ENUM is also generated if **glGetColorTableParameterivEXT** or **glGetColorTableParameterfvEXT** parameter <*target*> specifies a proxy table

target and <*pname*> specifies a piece of state which is not maintained for proxy tables.

TABLE_TOO_LARGE_EXT is generated if the color table requested by **glColorTableEXT** or **glCopyColorTableEXT** is larger than can be supported by the implementation, and the <*target*> parameter is **GL_COLOR_TABLE_EXT, GL_POST_CONVOLUTION_COLOR_TABLE_EXT,** or **GL_POST_COLOR_MATRIX_COLOR_TABLE_EXT**.

INVALID_OPERATION is generated if **glColorTableEXT**, **glCopyColorTableEXT**, **glColorTableParameterivEXT, glColorTableParameterfvEXT, glGetColorTableEXT, glGetColorTableParameterivEXT,** or **glGetColorTableParameterfvEXT** is called between execution of **glBegin** and the corresponding execution of **glEnd**.

EXT_compiled_vertex_array

This extension defines an interface which allows static vertex array data to be cached or pre-compiled for more efficient rendering.

EXTENSION STRING

GL_EXT_compiled_vertex_array

PROCEDURE DEFINITIONS

```
void glLockArraysEXT(int first, sizei count);

void glUnlockArraysEXT(void);
```

PARAMETERS

glLockArraysEXT

first	Index of first element in the array to process.
count	number of elements in the array

NEW TOKENS

Accepted by the <*pname*> parameter of **glGetBooleanv**, **glGetIntegerv, glGetFloatv** and **glGetDoublev**:

GL_ARRAY_ELEMENT_LOCK_FIRST_EXT
GL_ARRAY_ELEMENT_LOCK_COUNT_EXT

DESCRIPTION

The currently enabled vertex arrays can be locked with the command **glLockArraysEXT**. The the vertex arrays are locked, OpenGL can compile the array data or the transformed results of array data associated with the currently enabled vertex arrays. The vertex arrays are unlocked by the **glUnlockArraysEXT** command.

Between **glLockArraysEXT** and **glUnlockArraysEXT**, the application should make sure that none of the array data in the range of elements specified by <*first*>

and <*count*> are changed. Changes to the array data between the execution of **glLockArraysEXT** and **glUnlockArraysEXT** commands may affect calls to **glDrawArrays**, **glArrayElement**, or **glDrawElements** commands in non-sequential ways.

While using a compiled vertex array, references to array elements by the commands **glDrawArrays**, **glArrayElement**, or **glDrawElements** which are outside of the range specified by <*first*> and <*count*> are undefined.

ERRORS

INVALID_VALUE is generated if **glLockArraysEXT** parameter <*first*> is less than zero

INVALID_VALUE is generated if **glLockArraysEXT** parameter <*count*> is less than or equal to zero.

INVALID_OPERATION is generated if **glLockArraysEXT** is called between execution of **glLockArraysEXT** and the corresponding **glUnlockArraysEXT**.

INVALID_OPERATION is generated if **glUnlockArraysEXT** is called without a corresponding previous execution of **glLockArraysEXT**.

INVALID_OPERATION is generated if **glLockArraysEXT** or **glUnlockArraysEXT** is called between execution of **glBegin** and the corresponding execution of **glEnd**.

ASSOCIATED GETS

Calling **glGetIntegerv** with an argument of **GL_ARRAY_ELEMENT_LOCK_FIRST_EXT** returns the value of the **glLockArraysEXT** <*first*> parameter.

Calling **glGetIntegerv** with an argument of **GL_ARRAY_ELEMENT_LOCK_COUNT_EXT** returns the value of the **glLockArraysEXT** <*count*> parameter.

SEE ALSO

vertexArrays (OpenGL 1.1)

EXT_convolution

This extension defines 1 and 2 dimensional convolution operations at a fixed location in the pixel transfer process.

EXTENSION STRING

GL_EXT_convolution

PROCEDURE DEFINITIONS

```
void glConvolutionFilter1DEXT(enum target,
              enum internalformat,
              sizei width,
              enum format,
              enum type,
              const void* image);

void glConvolutionFilter2DEXT(enum target,
              enum internalformat,
              sizei width,
              sizei height,
              enum format,
              enum type,
              const void* image);

void glCopyConvolutionFilter1DEXT(enum target,
              enum internalformat,
              int x,
              int y,
              sizei width);

void glCopyConvolutionFilter2DEXT(enum target,
              enum internalformat,
              int x,
              int y,
              sizei width,
              sizei height);

void glGetConvolutionFilterEXT(enum target,
              enum format,
              enum type,
              void* image);
```

```
void glSeparableFilter2DEXT(enum target,
             enum internalformat,
             sizei width,
             sizei height,
             enum format,
             enum type,
             const void* row,
             const void* column);

void glGetSeparableFilterEXT(enum target,
             enum format,
             enum type,
             void* row,
             void* column,
             void* span);

void glConvolutionParameteriEXT(enum target,
             enum pname,
             int param);

void glConvolutionParameterivEXT(enum target,
             enum pname,
             const int* params);

void glConvolutionParameterfEXT(enum target,
             enum pname,
             float param);

void glConvolutionParameterfvEXT(enum target,
             enum pname,
             const float* params);

void glGetConvolutionParameterivEXT(enum target,
             enum pname,
             int* params);

void glGetConvolutionParameterfvEXT(enum target,
             enum pname,
             float* params);
```

PARAMETERS

glConvolutionFilter1DEXT
glConvolutionFilter2DEXT

target	One of the convolution tokens listed below, either GL_CONVOLUTION_1D_EXT or GL_CONVOLUTION_2D_EXT.
internalformat	Specifies the format of the image that will be retained after processing. It must be one of ALPHA, LUMINANCE, LUMINANCE_ALPHA, INTENSITY_EXT, RGB, or RGBA.
width	Width of the image data
height	(glConvolutionFilter2DEXT only) Height of the image data
format	Format of the image data, one of the supported pixel formats such as GL_RGBA
type	Data type of the image data, one of the supported data types such as GL_INT or GL_FLOAT
image	Image data. For glConvolutionFilter1DEXT, this must point to a 1-dimensional image, for glConvolutionFilter2DEXT, a 2-dimensional image.

glCopyConvolutionFilter1DEXT
glCopyConvolutionFilter2DEXT

target	One of the convolution tokens listed below, either GL_CONVOLUTION_1D_EXT or GL_CONVOLUTION_2D_EXT.
internalformat	Specifies the format of the image that will be retained after processing. It must be one of ALPHA, LUMINANCE, LUMINANCE_ALPHA, INTENSITY_EXT, RGB, or RGBA.
x, y	Specifies a color buffer region whose left-most pixel has a window coordinate of x and y. Pixels within this region that are outside of the window associated with the current OpenGL context contain values that are undefined.
width	Width of the image data in pixels
height	(glCopyConvolutionFilter2DEXT only) Height of image data in pixels

glGetConvolutionFilterEXT

target	must be GL_CONVOLUTION_1D_EXT OR GL_CONVOLUTION_2D_EXT
format	must be one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA
type	must be UNSIGNED_BYTE, BYTE, UNSIGNED_SHORT, SHORT, UNSIGNED_INT, INT, or FLOAT
image	The 1-dimensional or 2-dimensional image returned

glSeparableFilter2DEXT
glGetSeparableFilterEXT

target	Must be GL_SEPARABLE_2D_EXT
internalformat	Specifies the formats of tow 1-dimensional images that will be retained; it must be one of GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY, GL_RGB, or GL_RGBA.
width, height	(glSeparableFilter2DEXT only) Used in conjunction with the row and column parameters. Width is the width of the row image. The height value is the width of the column image.
format	Format of image data, must be one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_ABGR_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA
type	Data type, must be GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.
row, column	Pointers to 1-dimensional images, the row image is <width> pixels wide, the column image is <height> pixels wide.
span	(glGetSeparableFilterEXT) No changes are made to any location related to *span*

glConvolutionParameteriEXT
glConvolutionParameterivEXT
glConvolutionParameterfEXT
glConvolutionParameterfvEXT

target	must be one of GL_CONVOLUTION_1D_EXT, GL_CONVOLUTION_2D_EXT, or GL_SEPARABLE_2D
pname	Convolution parameter to set, one of the tokens defined below
params	Convolution parameter values that correspond to the value of pname

glGetConvolutionParameterivEXT
glConvolutionParameterfvEXT

target	must be one of GL_CONVOLUTION_1D_EXT, GL_CONVOLUTION_2D_EXT, or GL_SEPARABLE_2D
pname	Convolution parameter to obtain the value of, one of the tokens defined below
params	Pointer to list of returned values

NEW TOKENS

Accepted by the <*cap*> parameter of glEnable, glDisable, and glIsEnabled, by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev, and by the <*target*> parameter of glConvolutionFilter1DEXT, glCopyConvolutionFilter1DEXT, glGetConvolutionFilterEXT, glConvolutionParameteriEXT, glConvolutionParameterfEXT, glConvolutionParameterivEXT, glConvolutionParameterfvEXT, glGetConvolutionParameterivEXT, and glGetConvolutionParameterfvEXT:

GL_CONVOLUTION_1D_EXT

Accepted by the <*cap*> parameter of glEnable, glDisable, and glIsEnabled, by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev, and by the <*target*> parameter of glConvolutionFilter2DEXT, glCopyConvolutionFilter2DEXT, glGetConvolutionFilterEXT, glConvolutionParameteriEXT, glConvolutionParameterfEXT, glConvolutionParameterivEXT, glConvolutionParameterfvEXT, glGetConvolutionParameterivEXT, and glGetConvolutionParameterfvEXT:

GL_CONVOLUTION_2D_EXT

Accepted by the <*cap*> parameter of glEnable, glDisable, and glIsEnabled, by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev, and by the <*target*> parameter of glSeparableFilter2D, glSeparableFilter2D, glGetSeparableFilter, glConvolutionParameteri, glConvolutionParameterf, glConvolutionParameteriv, glConvolutionParameterfv, glGetConvolutionParameteriv, and glGetConvolutionParameterfv:

GL_SEPARABLE_2D_EXT

Accepted by the <*pname*> parameter of glConvolutionParameteri, glConvolutionParameterf, glConvolutionParameteriv, glConvolutionParameterfv, glGetConvolutionParameteriv, and glGetConvolutionParameterfv:

GL_CONVOLUTION_BORDER_MODE_EXT

Accepted by the <*pname*> parameter of glConvolutionParameteriv, glConvolutionParameterfv, glGetConvolutionParameteriv, and glGetConvolutionParameterfv:

GL_CONVOLUTION_FILTER_SCALE_EXT
GL_CONVOLUTION_FILTER_BIAS_EXT

Accepted by the <*param*> parameter of glConvolutionParameteri, and glConvolutionParameterf, and by the <*params*> parameter of glConvolutionParameteriv and glConvolutionParameterfv, when the <*pname*> parameter is:

GL_REDUCE_EXT

Accepted by the <*pname*> parameter of glGetConvolutionParameteriv and glGetConvolutionParameterfv:

GL_CONVOLUTION_FORMAT_EXT
GL_CONVOLUTION_WIDTH_EXT
GL_CONVOLUTION_HEIGHT_EXT
GL_MAX_CONVOLUTION_WIDTH_EXT
GL_MAX_CONVOLUTION_HEIGHT_EXT

Accepted by the <*pname*> parameter of glPixelTransferi, glPixelTransferf, and by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

GL_POST_CONVOLUTION_RED_SCALE_EXT
GL_POST_CONVOLUTION_GREEN_SCALE_EXT
GL_POST_CONVOLUTION_BLUE_SCALE_EXT
GL_POST_CONVOLUTION_ALPHA_SCALE_EXT
GL_POST_CONVOLUTION_RED_BIAS_EXT
GL_POST_CONVOLUTION_GREEN_BIAS_EXT
GL_POST_CONVOLUTION_BLUE_BIAS_EXT
GL_POST_CONVOLUTION_ALPHA_BIAS_EXT

DESCRIPTION

A 1-dimensional convolution filter is defined using glConvolutionFilter1DEXT. Its <target> parameter must be GL_CONVOLUTION_1D_EXT. Its <internalformat>, <format>, and <type> parameters have identical semantics and accept the same values as do their 2D counterparts. <image> must point to a 1-dimensional image, however. The 1-dimensional image is taken from memory and processed as if glConvolutionFilter2DEXT were called with a height of 1, except that it is scaled and biased by the two 1D vectors, rather than the 2D vectors. (The 1D scale and bias vectors are specified using glConvolutionParameterivEXT with <target> GL_CONVOLUTION_1D_EXT and <pname> GL_CONVOLUTION_FILTER_SCALE_EXT or GL_CONVOLUTION_FILTER_BIAS_EXT.) The 1-dimensional image is formed with coordinates i such that i increases from left to right, starting at zero. Image location i is specified by the ith pixel, counting from zero. The error GL_INVALID_VALUE is generated if <width> is greater than the maximum supported value, which is queried using glGetConvolutionParameterivEXT, setting <target> to GL_CONVOLUTION_1D_EXT and <pname> to GL_MAX_CONVOLUTION_WIDTH_EXT. One and 2-dimensional filters can also be defined using image data in the framebuffer. Rather than accepting image data from memory, they copy image data from the color buffer specified by the current glReadBuffer mode. glCopyConvolutionFilter2DEXT accepts image data from a <width> pixel wide by <height> pixel high color buffer region whose lower-left pixel has window coordinates <x>,<y>. If any pixels within this region are outside the window that is associated with the OpenGL context, the values obtained for those pixels are undefined.

These pixel values are obtained from the framebuffer exactly as if glReadPixels had been called with <format> set to GL_RGBA, with processing continuing through

Conversion of RGBA values. At this point all pixel component values are treated exactly as if glConvolutionFilter2DEXT had been called, beginning with the scaling of the color components by GL_CONVOLUTION_FILTER_SCALE_EXT. Pixel ordering is such that lower X screen coordinates correspond to lower i filter image coordinates, and lower Y screen coordinates correspond to lower j filter image coordinates. The semantics and accepted values of the <target> and <internalformat> parameters are exactly equivalent to their glConvolutionFilter2DEXT counterparts.

glCopyConvolutionFilter1DEXT accepts image data from a <width> pixel wide by 1 pixel high color buffer region whose left-most pixel has window coordinates <x>,<y>. If any pixels within this region are outside the window that is associated with the GL context, the values obtained for those pixels are undefined. The pixels are processed just as those of glCopyConvolutionFilter2DEXT are, and they define a filter image such that lower X window coordinates correspond to lower i filter image coordinates. The semantics and accepted values of the <target> and <internalformat> parameters are exactly equivalent to their glConvolutionFilter1DEXT counterparts. Special facilities are provided for the definition of 2-dimensional separable filters -- filters whose image can be represented as the product of two 1-dimensional images, rather than as full 2-dimensional images. A 2-dimensional convolution filter is specified using glSeparableFilter2DEXT. Its <target> parameter must be GL_SEPARABLE_2D_EXT. Its <internalformat> parameter specifies the formats of two 1-dimensional images that will be retained; it must be one of GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY, GL_RGB, or GL_RGBA. <row> and <column> point to two 1-dimensional images in memory. The <row> image is <width> pixels wide, and is defined by <format> and <type>. The <column> image is <height> pixels wide, and is also defined by <format> and <type>. The two images are extracted from memory and processed just as if glConvolutionFilter1DEXT were called separately for each, with the resulting retained images replacing the current 2D separable filter images, except that each image is scaled and biased using the 2D separable scale and bias vectors. (These vectors are specified using glConvolutionParameterivEXT with <target> GL_SEPARABLE_2D_EXT and <pname> GL_CONVOLUTION_FILTER_SCALE_EXT or GL_CONVOLUTION_FILTER_BIAS_EXT.)

If GL_CONVOLUTION_1D_EXT is enabled, the 1-dimensional convolution filter is applied only to the image passed to glTexImage1D, and to 1-dimensional textures queried by glGetTexImage. If GL_CONVOLUTION_2D_EXT is enabled, the 2-dimensional convolution filter is applied only to the 2D images passed to glDrawPixels, glCopyPixels, glReadPixels, glTexImage2D, glTexSubImage2D,

glCopyTexImage2D, glCopyTexSubImage2D, and glGetTexImage, and to 2-dimensional images queried by glGetTexImage. If GL_SEPARABLE_2D_EXT is enabled, and GL_CONVOLUTION_2D_EXT is disabled, the separable 2-dimensional convolution filter is applied only to these same images.

The convolution operation is defined differently for each of the three convolution filters. In the following equations the

SUM{}{}equation

notation indicate the sum of the equation evaluated for all combinations of conditions indicated within the sets of curly brackets. The variables Wf and Hf refer to the dimensions of the convolution filter. The variables Ws and Hs refer to the dimensions of the source pixel image. The convolution equations are:

1-dimensional filter:

C[i] = SUM{n = 0 through Wf-1}

Cs[i+n] * Cf[n]

2-dimensional filter:

C[i,j] = SUM{n = 0 through Wf-1}

{m = 0 through Hf-1}

Cs[i+n,j+m] * Cf[n,m]

2-dimensional separable filter:

C[i,j] = SUM{n = 0 through Wf-1}

{m = 0 through Hf-1}

Cs[i+n,j+m] * Crow[n] * Ccolumn[m]

If Wf of a 1-dimensional filter is zero, then C[i] is always set to zero. Likewise, if either Wf or Hf of a 2-dimensional filter is zero, then C[i,j] is always set to zero.

The convolution border mode for a specific convolution filter is specified using glConvolutionParameteriEXT with the <target> parameter set to the name of the filter, the <pname> parameter set

to GL_CONVOLUTION_BORDER_MODE_EXT, and <param> set to GL_REDUCE_EXT. (This extension defines only a single border mode.) The width and height of source images convolved with border mode GL_REDUCE_EXT are

reduced by Wf-1 and Hf-1, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a convolution with border mode GL_REDUCE_EXT are zero through Ws-Wf in width, and zero through Hs-Hf in height. In cases where errors can result from the specification of invalid image dimensions, it is these resulting dimensions that are tested, not the dimensions of the source image. (A specific example is glTexImage1D and glTexImage2D, which specify constraints for image dimensions. Even if glTexImage1D or glTexImage2D is called with a null pixel pointer, the dimensions of the resulting texture image are those that would result from the convolution of the specified image.) If a convolution operation is performed, the resulting image is scaled and biased by parameters specified using the glPixelTransfer command. These operations are:

```
red = red * GL_POST_CONVOLUTION_RED_SCALE_EXT +
GL_POST_CONVOLUTION_RED_BIAS_EXT

green = green * GL_POST_CONVOLUTION_GREEN_SCALE_EXT +
GL_POST_CONVOLUTION_GREEN_BIAS_EXT

blue = blue * GL_POST_CONVOLUTION_BLUE_SCALE_EXT +
GL_POST_CONVOLUTION_BLUE_BIAS_EXT

alpha = alpha * GL_POST_CONVOLUTION_ALPHA_SCALE_EXT +
GL_POST_CONVOLUTION_ALPHA_BIAS_EXT
```

If no convolution operation is performed, the scale and bias are not performed either.

The current contents of a convolution filter image are queried using glGetConvolutionFilterEXT. <target> must be GL_CONVOLUTION_1D_EXT or GL_CONVOLUTION_2D_EXT. <format> must be one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_ABGR_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA. <type> must be GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT. The 1-dimensional or 2-dimensional image is returned to <image>. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to glReadPixels are performed.

The current contents of a separable filter image are queried using glGetSeparableFilterEXT. <target> must be GL_SEPARABLE_2D_EXT. <format> and <type> accept the same values as do the corresponding parameters of

glGetConvolutionFilterEXT. The row and column images are returned to <row> and <column> respectively. No change is made to any location related to . Pixel processing and component mapping are identical to those of glGetConvolutionFilterEXT.

ERRORS

GL_INVALID_ENUM is generated if glCovolutionFilter1DEXT parameter <target> is not GL_CONVOLUTION_1D_EXT.

GL_INVALID_ENUM is generated if glCovolutionFilter2DEXT parameter <target> is not GL_CONVOLUTION_2D_EXT.

GL_INVALID_ENUM is generated if glCopyConvolutionFilter1DEXT parameter <target> is not GL_CONVOLUTION_1D_EXT.

GL_INVALID_ENUM is generated if glCopyConvolutionFilter2DEXT parameter <target> is not GL_CONVOLUTION_2D_EXT.

GL_INVALID_ENUM is generated if glSeparableFilter2DEXT parameter <target> is not GL_SEPARABLE_2D_EXT.

GL_INVALID_ENUM is generated if glGetConvolutionFilterEXT parameter <target> is not GL_CONVOLUTION_1D_EXT or GL_CONVOLUTION_2D_EXT.

GL_INVALID_ENUM is generated if glGetSeparableFilterEXT parameter <target> is not GL_SEPARABLE_2D_EXT.

GL_INVALID_ENUM is generated if glCovolutionParameteriEXT, glCovolutionParameterfEXT, glCovolutionParameterivEXT, glCovolutionParameterfvEXT, glGetConvolutionParameterivEXT, or glGetConvolutionParameterfvEXT parameter <target> is not GL_CONVOLUTION_1D_EXT, GL_CONVOLUTION_2D_EXT, or GL_SEPARABLE_2D_EXT.

GL_INVALID_ENUM is generated if glCovolutionFilter1DEXT, glCovolutionFilter2DEXT, glCopyConvolutionFilter1DEXT, glCopyConvolutionFilter2DEXT, or glSeparableFilter2DEXT parameter <internalformat> is not GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY, GL_RGB, or GL_RGBA.

GL_INVALID_VALUE is generated if glCovolutionFilter1DEXT parameter <width> is less than zero, or greater than GL_MAX_CONVOLUTION_WIDTH_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_CONVOLUTION_1D_EXT.

GL_INVALID_VALUE is generated if glCovolutionFilter2DEXT parameter <width> is less than zero, or greater than GL_MAX_CONVOLUTION_WIDTH_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_CONVOLUTION_2D_EXT.

GL_INVALID_VALUE is generated if glCopyConvolutionFilter1DEXT parameter <width> is less than zero, or greater than GL_MAX_CONVOLUTION_WIDTH_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_CONVOLUTION_1D_EXT.

GL_INVALID_VALUE is generated if glCopyConvolutionFilter2DEXT parameter <width> is less than zero, or greater than GL_MAX_CONVOLUTION_WIDTH_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_CONVOLUTION_2D_EXT.

GL_INVALID_VALUE is generated if glSeparableFilter2DEXT parameter <width> is less than zero, or greater than GL_MAX_CONVOLUTION_WIDTH_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_SEPARABLE_2D_EXT.

GL_INVALID_VALUE is generated if glCovolutionFilter2DEXT parameter <height> is less than zero, or greater than GL_MAX_CONVOLUTION_HEIGHT_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_CONVOLUTION_2D_EXT.

GL_INVALID_VALUE is generated if glCopyConvolutionFilter2DEXT parameter <height> is less than zero, or greater than GL_MAX_CONVOLUTION_HEIGHT_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_CONVOLUTION_2D_EXT.

GL_INVALID_VALUE is generated if glSeparableFilter2DEXT parameter <height> is less than zero, or greater than GL_MAX_CONVOLUTION_HEIGHT_EXT as queried using glGetConvolutionParameterivEXT with <target> GL_SEPARABLE_2D_EXT.

GL_INVALID_OPERATION is generated if glCovolutionFilter1DEXT, glCovolutionFilter2DEXT, glCopyConvolutionFilter1D,

glCopyConvolutionFilter2D, glGetConvolutionFilterEXT, glSeparableFilter2DEXT, glGetSeparableFilterEXT, glCovolutionParameteriEXT, glCovolutionParameterivEXT, glCovolutionParameterfEXT, glCovolutionParameterfvEXT, glGetCovolutionParameterivEXT, or glGetCovolutionParameterfvEXT is executed between execution of glBegin and the corresponding execution of glEnd.

ASSOCIATED GETS

See details of glGetConvolutionParameter*EXT, glGetConvolutionFilterEXT, and glGetSeparableFilterEXT routines above.

SEE ALSO

glEnable, glPixelTransfer, glReadPixels

EXT_depth_pass_instrument

This extension provides applications with the capability to perform tests against data written to the depth buffer.

EXTENSION STRING

GL_SGIX_depth_pass_instrument

PROCEDURE DEFINITIONS

```
void glInstrumentsBufferSGIX(sizei size, int *buf);
void glStartInstrumentsSGIX(void)
void glStopInstrumentsSGIX(int marker)
void glReadInstrumentsSGIX(int marker)
int glPollInstrumentsSGIX(int *markerp)
int glGetInstrumentsSGIX(void)
```

PARAMETERS

glInstrumentsBufferSGIX

size	Size of the instrument buffer.
buf	Buffer that will hold instrument results

glStopInstrumentsSGIX, glReadInstrumentsSGIX

marker	Marker value that will be written into the instrument buffer

glPollInstrumentsSGIX

markerp	Pointer that will have an instrument marker value written to it

NEW TOKENS

Accepted by the <*cap*> parameters of glEnable, glDisable, glIsEnable:

GL_DEPTH_PASS_INSTRUMENT_SGIX

Accepted by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

GL_DEPTH_PASS_INSTRUMENT_COUNTERS_SGIX
GL_DEPTH_PASS_INSTRUMENT_MAX_SGIX

DESCRIPTION

This extension defines an instrument that specifies a counter of the number of fragments which passed the depth test during rasterization. Once the GL_DEPTH_PASS_INSTRUMENT_SGIX is enabled via a called to glStartInstrumentSGIX, a counter or counters of the number of fragments which have passed the depth test or which have been drawn with the depth test is disabled is maintained. The number of counters can be queried using the token GL_DEPTH_PASS_INSTRUMENT_COUNTERS_SGIX. For each fragment which passes the depth test or is drawn with depth testing disabled, one counter is incremented by one. If the increment would have caused the counter to go beyond is maximum representable value, the count is clamped to the maximum. The maximum value of the counters may be queried using DEPTH_PASS_INSTRUMENT_MAX_SGIX. The counter values are unsigned, so DEPTH_PASS_INSTRUMENT_MAX_SGIX may be as high as the maximum value of an unsigned integer. Instrument responses are formatted as follows:

- GL_DEPTH_PASS_INSTRUMENT_SGIX enum
- number of int's in the response
- counter value
- marker value

Application code will set the instrument return buffer with the glInstrumentsBufferSGIX function, the buffer should be of a size suitable to contain the return information noted above. A subsequent call to glStartInstrumentsSGIX will start gathering of depth test data. After geometry is drawn, a call to glStopInstrumentsSGIX will stop additional data from being gathered. The glStopInstrumentsSGIX call includes the <marker> parameter, this marker will be stored with the instrument data. The glPollInstrumentsSGIX call will be used to retrieve marker information. If a new measurement has appeared in the buffer since the last call to glPollInstrumentsSGIX, a value of "1" will be returned, otherwise zero is returned. Once the desired marker has been found in the instrument data, the instrument buffer can be searched to determine the status of the results for each depth test. The following code fragment illustrates this procedure:

```
#define MARKER      0x0ffffffd
#define OCCLUSION_INSTRUMENT_BUF_SIZE   32

struct occlusionInfo_s
{
   GLenum enableEnum;
   GLint  measurementSize;
   GLint  result;
   GLint  marker;
} occlusionInstrumentBuf[OCCLUSION_INSTRUMENT_BUF_SIZE];

glInstrumentsBufferSGIX(OCCLUSION_INSTRUMENT_BUF_SIZE,
      (int *) occlusionInstrumentBuf);

glStartInstrumentsSGIX();

...
// drawing of some geometry
...

glStopInstrumentsSGIX(MARKER);

while(!(r = glPollInstrumentsSGIX(&marker)) ||
      (marker != MARKER)) ;

 if(r == -1) printf("Instrument buffer overflowed\n");

 glGetIntegerv(GL_INSTRUMENT_MEASUREMENTS_SGIX,
      &measurements);

 for(i = 0; i < measurements; i++)
 {
   if(occlusionInstrumentBuf[i].result == GL_TRUE)
      printf("Geometry Occluded\n");
   else
      printf("Geometry Visible\n");
 }
```

ERRORS

GL_INVALID_OPERATION is generated if **glStartInstrumentsSGIX** is executed between the execution twice without an intervening **glStopInstrumentsSGIX**. Similarly, this error is generated if a **glStopInstrumentsSGIX** is executed without a prior **glStartInstrumentsSGIX** call.

GL_INVALID_OPERATION is generated if any calls to **glStartInstrumentsSGIX**, **glStopInstrumentsSGIX**, or **glReadInstrumentsSGIX** without a prior successful call to **glInstrumentsBufferSGIX**.

GL_INVALID_OPERATION is generated if **glStartInstrumentsSGIX**, **glStopInstrumentsSGIX**, or **glReadInstrumentsSGIX** are called between **glBegin** and the corresponding **glEnd**.

GL_INVALID_VALUE is generated if **glInstrumentsBufferSGIX** is called with a negative <*size*> value.

ASSOCIATED GETS

Calling **glGet*** with an argument of **GL_INSTRUMENT_MEASURMENTS** returns the number of measurements taken.

SGI_texture_lod

This extension imposes two constraints related to the texture level of detail parameter LOD.

EXTENSION STRING

GL_SGI_texture_lod

PROCEDURE DEFINITIONS

none

NEW TOKENS

Accepted by the <*pname*> parameter of glTexParameteri, glTexParameterf, glTexParameteriv, glTexParameterfv, glGetTexParameteriv, and glGetTexParameterfv:

GL_TEXTURE_MIN_LOD_SGI
GL_TEXTURE_MAX_LOD_SGI
GL_TEXTURE_BASE_LEVEL_SGI
GL_TEXTURE_MAX_LEVEL_SGI

DESCRIPTION

This extension imposes two constraints related to the texture level of detail parameter LOD, which is represented by the Greek character lambda in the GL Specification. One constraint clamps LOD to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the LOD parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

Note: because the shape of the mipmap array is always determined by the dimensions of the level 0 array, this array must be loaded for mipmapping to be active. If the level 0 array is specified with a null image pointer, however, no actual data transfer will take place. And a sufficiently tuned implementation might not even allocate space for a level 0 array so specified until true image data were presented.

ERRORS

INVALID_VALUE is generated if an attempt is made to set GL_TEXTURE_BASE_LEVEL_SGI or GL_TEXTURE_MAX_LEVEL_SGI to a negative value.

ASSOCIATED GETS

Calling **glTexParameterfv** with an argument of GL_TEXTURE_MIN_LOD_SGI or GL_TEXTURE_MAX_LOD_SGI returns the current min and max LOD values respectively.

Calling **glTexParameteriv** with an argument of GL_TEXTURE_BASE_LEVEL_SGI or GL_TEXTURE_MAX_LEVEL_SGI returns the current base and max LOD level values respectively.

SEE ALSO

glTexParameter*, glTexEnv*

EXT_histogram

This extension defines pixel operations that count occurrences of specific color component values (histogram) and that track the minimum and maximum color component values (minmax). An optional mode allows pixel data to be discarded after the histogram and / or minmax operations are completed. Otherwise the pixel data continues on to the next operation unaffected.

EXTENSION STRING

GL_EXT_histogram

PROCEDURE DEFINITIONS

```
void glHistogramEXT(enum target,
             sizei width,
             enum internalformat,
             boolean sink);

void glResetHistogramEXT(enum target);

void glGetHistogramEXT(enum target,
             boolean reset,
             enum format,
             enum type,
             void* values);

void glGetHistogramParameterivEXT(enum target,
             enum pname,
             int* params);

void glGetHistogramParameterfvEXT(enum target,
             enum pname,
             float* params);

void glMinmaxEXT(enum target,
             enum internalformat,
             boolean sink);

void glResetMinmaxEXT(enum target);
```

```
void GetMinmaxEXT(enum target,
             boolean reset,
             enum format,
             enum type,
             void* values);

void GetMinmaxParameterivEXT(enum target,
             enum pname,
             int* params);

void GetMinmaxParameterfvEXT(enum target,
             float* params);
```

PARAMETERS

glHistogramEXT

target parameters are to be set. Must be one of GL_HISTOGRAM_EXT or GL_PROXY_HISTOGRAM_EXT.

width The number of entries in the histogram table. Must be a power of 2.

internalformat The format of entries in the histogram table. Must be one of 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_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, or GL_RGBA16.

sink If GL_TRUE, pixels will be consumed by the histogram process and no drawing or texture loading will take place. If GL_FALSE, pixels will proceed to the minmax process after histogramming.

glResetHistogramEXT

target must be GL_HISTOGRAM_EXT

glGetHistogramEXT

target Must be GL_HISTOGRAM_EXT.

reset If GL_TRUE, each component counter that is actually returned is reset to zero. (Other counters are unaffected.) If GL_FALSE, none of the counters in the histogram table is modified.

format The format of values to be returned in values. Must be one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_ABGR_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA.

type The type of values to be returned in values. Must be one of GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2_EXT, GL_UNSIGNED_SHORT_4_4_4_4_EXT, GL_UNSIGNED_SHORT_5_5_5_1_EXT, GL_UNSIGNED_INT_8_8_8_8_EXT, or GL_UNSIGNED_INT_10_10_10_2_EXT.

glGetHistogramParameterivEXT
glGetHistogramParameterfvEXT

target Must be one of GL_HISTOGRAM_EXT or GL_PROXY_HISTOGRAM_EXT.

pname The name of the parameter to be retrieved. Must be one of GL_HISTOGRAM_WIDTH_EXT, GL_HISTOGRAM_FORMAT_EXT, GL_HISTOGRAM_RED_SIZE_EXT, GL_HISTOGRAM_GREEN_SIZE_EXT, GL_HISTOGRAM_BLUE_SIZE_EXT, GL_HISTOGRAM_ALPHA_SIZE_EXT, GL_HISTOGRAM_LUMINANCE_SIZE_EXT, or GL_HISTOGRAM_SINK_EXT.

params Pointer to storage for the returned values.

glMinMaxEXT

target The minmax table whose parameters are to be set. Must be GL_MINMAX_EXT.

internalformat The format of entries in the minmax table. Must be one of GL_ALPHA, GL_ALPHA4_EXT, GL_ALPHA8_EXT, GL_ALPHA12_EXT, GL_ALPHA16_EXT, GL_LUMINANCE, GL_LUMINANCE4_EXT, GL_LUMINANCE8_EXT, GL_LUMINANCE12_EXT, GL_LUMINANCE16_EXT, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4_EXT, GL_LUMINANCE6_ALPHA2_EXT, GL_LUMINANCE8_ALPHA8_EXT, GL_LUMINANCE12_ALPHA4_EXT, GL_LUMINANCE12_ALPHA12_EXT, GL_LUMINANCE16_ALPHA16_EXT, GL_RGB, GL_RGB2_EXT, GL_RGB4_EXT, GL_RGB5_EXT, GL_RGB8_EXT, GL_RGB10_EXT, GL_RGB12_EXT, GL_RGB16_EXT, GL_RGBA, GL_RGBA2_EXT, GL_RGBA4_EXT, GL_RGB5_A1_EXT,GL_RGBA8_EXT, GL_RGB10_A2_EXT, GL_RGBA12_EXT, or GL_RGBA16_EXT.

sink If GL_TRUE, pixels will be consumed by the minmax process and no drawing or texture loading will take place. If GL_FALSE, pixels will proceed to the final conversion process after minmax.

glResetMinmaxEXT

enum Must be GL_MINMAX_EXT.

glGetMinmaxEXT

target Must be GL_MINMAX_EXT.

reset If GL_TRUE, all entries in the minmax table that are actually returned are reset to their initial values. (Other entries are unaltered.) If GL_FALSE, the minmax table is unaltered.

format The format of the data to be returned in values. Must be one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_ABGR_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA.

type The type of the data to be returned in values. Must be one of GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2_EXT, GL_UNSIGNED_SHORT_4_4_4_4_EXT, GL_UNSIGNED_SHORT_5_5_5_1_EXT, GL_UNSIGNED_INT_8_8_8_8_EXT, or GL_UNSIGNED_INT_10_10_10_2_EXT.

values A pointer to storage for the returned values.

glGetMinmaxParameterfvEXT
glGetMinmaxParameterivEXT

target Must be GL_MINMAX_EXT.

pname The parameter to be retrieved. Must be one of GL_MINMAX_FORMAT_EXT or GL_MINMAX_SINK_EXT.

params A pointer to storage for the retrieved parameters.

NEW TOKENS

Accepted by the <cap> parameter of glEnable, glDisable, and glIsEnabled, by the <pname> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev, and by the <target> parameter of glHistogramEXT, glResetHistogramEXT, glGetHistogramEXT, glGetHistogramParameterivEXT, and glGetHistogramParameterfvEXT:

GL_HISTOGRAM_EXT

Accepted by the <target> parameter of glHistogramEXT, glGetHistogramParameterivEXT, and glGetHistogramParameterfvEXT:

GL_PROXY_HISTOGRAM_EXT

Accepted by the <*pname*> parameter of glGetHistogramParameterivEXT and glGetHistogramParameterfvEXT:

GL_HISTOGRAM_WIDTH_EXT
GL_HISTOGRAM_FORMAT_EXT
GL_HISTOGRAM_RED_SIZE_EXT
GL_HISTOGRAM_GREEN_SIZE_EXT
GL_HISTOGRAM_BLUE_SIZE_EXT

GL_HISTOGRAM_ALPHA_SIZE_EXT
GL_HISTOGRAM_LUMINANCE_SIZE_EXT
GL_HISTOGRAM_SINK_EXT

Accepted by the <*cap*> parameter of glEnable, glDisable, and glIsEnabled, by the <pname> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev, and by the <target> parameter of glMinmaxEXT, glResetMinmaxEXT, glGetMinmaxEXT, glGetMinmaxParameterivEXT, and glGetMinmaxParameterfvEXT:

MINMAX_EXT

Accepted by the <*pname*> parameter of glGetMinmaxParameterivEXT and glGetMinmaxParameterfvEXT:

GL_MINMAX_FORMAT_EXT
GL_MINMAX_SINK_EXT

DESCRIPTION

The glHistogramEXT function defines a histogram table. Histogramming is enabled and disabled with calls to glEnable and glDisable. When GL_HISTOGRAM_EXT is enabled, RGBA color components are converted to histogram table indices by clamping to the range [0,1], multiplying by the width of the histogram table, and rounding to the nearest integer. The table entries selected by the RGBA indices are then incremented.When target is GL_HISTOGRAM_EXT, glHistogramEXT redefines the current histogram table to have width entries of the format specified by

internalformat. The entries are indexed 0 through width-1, and all entries are initialized to zero. The values in the previous histogram table, if any, are lost. If sink is GL_TRUE, then pixels are discarded after histogramming; no further processing of the pixels takes place, and no drawing, texture loading, or pixel readback will result.

When target is GL_PROXY_HISTOGRAM_EXT, glHistogramEXT computes all state information as if the histogram table was to be redefined, but does not actually define the new table. If the requested histogram table is too large to be supported, then the state information will be set to zero. This provides a way to determine if a histogram table with the given parameters can be supported. after histogramming; no further processing of the pixels takes place, and no drawing, texture loading, or pixel readback will result.

The histogram state information may be queried by calling glGetHistogramParameterEXT with a target of GL_HISTOGRAM_EXT (to obtain information for the current histogram table) or GL_PROXY_HISTOGRAM_EXT (to obtain information from the most recent proxy request) and one of the previously described tokens for the pname argument.

Calls to glResetHistogramEXT resets all the elements of the current histogram table to zero.

Calls to glGetHistogramEXT return the current histogram table as a one-dimensional image with the same width as the histogram. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to 1D images are honored. Color components that are requested in the specified format, but which are not included in the internal format of the histogram, are returned as zero.

The glMinmaxEXT function defines the minmax table. Minmax is part of the OpenGL pixel transfer process. It takes place immediately after histogramming. Minmax is performed only for RGBA

pixels (though these may be specified originally as color indices and converted to RGBA by index table lookup). Minmax is enabled with glEnable and disabled with glDisable. glMinmaxEXT redefines the current minmax table to have entries of the format specified by internalformat. The maximum element is initialized with the smallest possible component values, and the minimum element is initialized with the largest possible component values. The values in the previous minmax table, if any, are lost. If sink is GL_TRUE, then pixels are discarded after minmax; no further processing of the pixels takes place, and no drawing, texture loading, or pixel readback will result.The minmax state information may be queried by calling glGetMinmaxParameter*EXT with a target of GL_MINMAX_EXT and one of the minmax tokens previously described for the <*pname*> argument.

The glResetMinmaxEXTfunction resets the elements of the current minmax table to their initial values: the ``maximum'' element receives the minimum possible component values, and the ``minimum'' element receives the maximum possible component values.

The glGetMinmaxEXT function returns the accumulated minimum and maximum pixel values (computed on a per-component basis) in a one-dimensional image of width 2. The first set of return values are the minima, and the second set of return values are the maxima. The format of the return values is determined by format, and their type is determined by type.

No pixel transfer operations are performed on the return values, but pixel storage modes that are applicable to 1-dimensional images are performed. Color components that are requested in the specified format, but which are not included in the internal format of the minmax table, are returned as zero. If <*reset*> is GL_TRUE, the minmax table entries corresponding to the return values are reset to their initial values. Minimum and maximum values that are not returned are not modified, even if reset is GL_TRUE.

ERRORS

glHistogramEXT calls can generate the following errors:

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

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

GL_INVALID_ENUM is generated if internalformat is not one of the allowable values.

GL_TABLE_TOO_LARGE_EXT is generated if target is GL_HISTOGRAM_EXT and the histogram table specified is too large for the implementation.

GL_INVALID_OPERATION is generated if glHistogramEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

glResetHistogramEXT calls can generate the following errors:

GL_INVALID_ENUM is generated if target is not GL_HISTOGRAM_EXT.

GL_INVALID_OPERATION is generated if glResetHistogramEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

Calls to glGetHistogramEXT can generate the following errors:

GL_INVALID_ENUM is generated if target is not GL_HISTOGRAM_EXT.

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

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

GL_INVALID_OPERATION is generated if glGetHistogramEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

If type is set to GL_UNSIGNED_BYTE_3_3_2_EXT, GL_UNSIGNED_SHORT_4_4_4_4_EXT, GL_UNSIGNED_SHORT_5_5_5_1_EXT, GL_UNSIGNED_INT_8_8_8_8_EXT, or GL_UNSIGNED_INT_10_10_10_2_EXT and the EXT_packed_pixels extension is not supported then a GL_INVALID_ENUM error is generated.

glMinMaxEXT calls can generate the following errors:

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

GL_INVALID_ENUM is generated if internalformat is not one of the allowable values.

GL_INVALID_OPERATION is generated if glMinmaxEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

glResetMinmaxEXT calls can generate the following errors:

GL_INVALID_ENUM is generated if target is not GL_MINMAX_EXT.

GL_INVALID_OPERATION is generated if glResetMinmaxEXT is executed between the execution of glBegin and the corresponding execution of glEnd.

glGetMinmaxEXT calls can generate the following errors:

GL_INVALID_ENUM is generated if target is not GL_MINMAX_EXT.

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

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

GL_INVALID_OPERATION is generated if glGetMinmaxEXT is executed between the execution of glBegin and the corresponding execution of glEnd. If type is set to GL_UNSIGNED_BYTE_3_3_2_EXT, GL_UNSIGNED_SHORT_4_4_4_4_EXT, GL_UNSIGNED_SHORT_5_5_5_1_EXT, GL_UNSIGNED_INT_8_8_8_8_EXT, or GL_UNSIGNED_INT_10_10_10_2_EXT and the EXT_packed_pixels extension is not supported then a GL_INVALID_ENUM error is generated.

SGIX_interlace

This extension provides a way to interlace rows of pixels when rasterizing pixel rectangles, and loading texture images.

EXTENSION STRING

GL_SGIX_interlace

NEW TOKENS

Accepted by the <*cap*> parameter of glEnable, glDisable, glIsEnabled, and by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, glGetDoublev:

GL_INTERLACE_SGIX

DESCRIPTION

In this context, interlacing means skipping over rows of pixels or texels in the destination. This is useful for dealing with video data since a single frame of video is typically composed from two images or fields: one image specifying the data for even rows of the frame and the other image specifying the data for odd rows of the frame.

ASSOCIATED GETS

Calling **glIsEnabled** with an argument of GL_INTERLACE_SGIX returns the current state of the interlace mode

WGL_EXT_make_current_read

EXTENSION STRING

WGL_EXT_make_current_read

DESCRIPTION

The wglMakeContextCurrentEXT function defines a way to set different draw (write) and read buffers. It is used for transferring pixels between drawables. Pbuffers must use wglMakeContextCurrentEXT instead of wglMakeCurrent. The wglGetCurrentReadDCEXT provides a mechanism for retrieving the device context of the current read target.

PROCEDURE DEFINITION

```
HDC wglGetCurrentReadDCEXT(void);
BOOL wglMakeContextCurrentEXT(
             HDC hWriteDC,
             HDC hReadDC,
             HGLRC hGLRC);
```

PARAMETERS

wglMakeContextCurrentEXT

hWriteDC	Handle to a device context. Subsequent OpenGL calls made by the calling thread that write data will write to this device context.
hReadDC	Handle to a device context. Subsequent OpenGL calls made by the calling thread that read data will read from this device context.
hGLRC	Handle to an OpenGL rendering context that this function sets as the calling thread's rendering context.

RETURN VALUES

When the **wglMakeContextCurrentEXT** function succeeds, the return value is **TRUE**, otherwise the return value is **FALSE**. The **wglMakeContextCurrentEXT** function will fail if *hWriteDC* or *hReadDC* are not valid DCs or if *hGLRC* is not a valid context. The function wglGetCurrentReadDCEXT returns the HDC of the current read context.

REMARKS

For NT 4.0, there is no way to extend the WGL interface. Applications wishing to use the **wglMakeContextCurrentEXT** function must look up the address of this function by using the **wglGetProcAddress** function. Since this is not an OpenGL extension, support will not be indicated in the *GL_EXTENSIONS* string returned by **glGetString**.

SEE ALSO

wglCreatePbufferEXT

EXT_packed_pixels

This extension provides support for packed pixels in host memory.

EXTENSION STRING

GL_EXT_packed_pixels

PROCEDURE DEFINITIONS

none

NEW TOKENS

Accepted by the <*type*> parameter of glDrawPixels, glReadPixels, glTexImage1D, glTexImage2D, glGetTexImage, glTexSubImage1DEXT, glTexSubImage2DEXT, glGetHistogramEXT, glGetMinmaxEXT, glConvolutionFilter1DEXT, glConvolutionFilter2DEXT, glGetConvolutionFilterEXT, glSeparableFilter2DEXT, glGetSeparableFilterEXT, glColorTableEXT, and glGetColorTableEXT:

GL_UNSIGNED_BYTE_3_3_2_EXT
GL_UNSIGNED_SHORT_4_4_4_4_EXT
GL_UNSIGNED_SHORT_5_5_5_1_EXT
GL_UNSIGNED_INT_8_8_8_8_EXT
GL_UNSIGNED_INT_10_10_10_2_EXT

DESCRIPTION

A packed pixel is represented entirely by one unsigned byte, one unsigned short, or one unsigned integer. The fields with the packed pixel are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes, including GL_PACK_SKIP_PIXELS, GL_PACK_ROW_LENGTH, GL_PACK_SKIP_ROWS, GL_PACK_IMAGE_HEIGHT_EXT, GL_PACK_SKIP_IMAGES_EXT, GL_PACK_SWAP_BYTES, GL_PACK_ALIGNMENT, and their unpacking counterparts all work correctly with packed pixels.

ERRORS

GL_INVALID_OPERATION errors will be generated if data type mismatches occur, such as specifying a 3 component format with 4 component data.

SEE ALSO

glDrawPixels, glReadPixels

WGL_EXT_pbuffer

NAME

WGL_EXT_pbuffer

DESCRIPTION

This extension defines pixel buffers (wglPbuffers or Pbuffer for short). These wglPbuffers are additional non-visible rendering buffers for an OpenGL renderer. The format of the color buffers and type and size of the any associated ancillary buffers for a wglPbuffer can only be described with a PIXELFORMATDESCRIPTOR.

NAME STRINGS

WGL_EXT_pbuffer

PROCEDURE DEFINITIONS

```
HPBUFFEREXT wglCreatePbufferEXT(
             HDC hDC,
             int iPixelFormat,
             int iWidth,
             int iHeight,
             const int *piAttriblist);

HDC wglGetPbufferDCEXT(HPBUFFEREXT hPbuffer);

int wglReleasePbufferDCEXT(
             HPBUFFEREXT hPbuffer
             HDC hDC);

BOOL wglDestroyPbufferEXT(HPBUFFEREXT hPbuffer);

BOOL wglQueryPbufferEXT(
             HPBUFFEREXT hPbuffer,
             int iAttribute,
             int *piValue);
```

PARAMETERS

wglCreatePbufferEXT

hDC	Handle to a device context for the parent of the device context to be created by this function.
iPixelFormat	Pixel format for the Pbuffer, id returned from a ChoosePixelFormat() command.
iWidth	Width of the Pbuffer in pixels.
iHeight	Height of the Pbuffer in pixels
piAtrributelist	pixel attribute list (NULL value unless using extended pixel formats)

wglGetPbufferDCEXT

hPbuffer	Pbuffer device context, created by a previous call to wglCreatePbufferEXT

wglReleasePbufferDCEXT

hPbuffer	Pbuffer device context, created by a previous call to wglCreatePbufferEXT
hDC	hardware device context the pbuffer was created from

wglDestroyPbufferEXT

hPbuffer	Handle to device context of Pbuffer to be destroyed

wglQueryPbufferEXT

hPbuffer	Handle to device of context of the pbuffer that is to be queried
iAttribute	Attribute that the calling thread wants information about. Valid values for this field are: WGL_PB_WIDTH_SGIX WGL_PB_HEIGHT_SGIX
iValue	Address to return query information into

DESCRIPTION

The **wglCreatePbufferEXT** function creates a Pbuffer and returns a handle to the pbuffer device context, HPBUFFEREXT. The size of the Pbuffer is determined by the values of the width and height parameters. If the function fails to create a wglPbuffer, an error is generated and 0 is returned. The piAttribList parameter is used for pixel information for to describe an extended pixel format. See the *Silicon Graphics Visual Workstation OpenGL Programming Guide for Windows NT* for more details on extended pixel formats.

The **wglDestroyPbufferEXT** function destroys the wglPbuffer as specified by the *pbuf* parameter. No values are returned, but an error is generated if the h*Pbuffer* parameter is not a valid pbuffer device context.

The **wglQueryPbufferEXT** function retrieves the dimensions of the allocated pbuffer. If the *hPbuffer* parameter is not a valid pbuffer device context, then an error is generated.

The **wglGetPbufferDCEXT** function returns the hardware device context HDC, used to create the *pbuffer* device context. An error is generated if the hPbuffer parameter is not a valid pbuffer context.

The **wglReleasePbufferDCEXT** function releases resources associated with the *hPbuffer* device context.An error is generated if *hPbuffer* is not a valid pbuffer context.

NOTES

For NT 4.0, there is no way to extend the WGL interface. Applications wishing to use the **wglCreatePbufferEXT, wglDestroyPbufferEXT, wglGetPbufferDCEXT, wglReleasePbufferEXT,** or **wglQueryPbufferEXT** functions must look up the address of this function by using the **wglGetProcAddress** function. Since these are not OpenGL extensions, support will not be indicated in the *GL_EXTENSIONS* string returned by **glGetString**. Applications should try to access the function pointer to these routines. If the function is not supported, 0 will be returned by **wglGetProcAddress**.

A pbuffer device context may not be passed to **wglMakeCurrent**, **wglCreateContext**, or **wglCreateLaterContext**. A pbuffer HDC can only be made current using **wglMakeContextCurrentEXT**.

SEE ALSO

wglMakeContextCurrentEXT, the *Silicon Graphics Visual Workstation OpenGL Programming Guide for Windows NT*

EXT_secondary_color

This extension allows specifying the RGB components of the secondary color used in the Color Sum stage, instead of using the default (0,0,0,0) color. It applies only in RGBA mode and when LIGHTING is disabled.

EXTENSION STRING

GL_EXT_secondary_color

PROCEDURE DEFINITIONS

```
void glSecondaryColor3bEXT(GLbyte red, GLbyte green, GLbyte blue)
void glSecondaryColor3bvEXT(const GLbyte *v)

void glSecondaryColor3sEXT(GLshort red,
GLshort green, GLshort blue);
void glSecondaryColor3svEXT(const GLshort *v);

void glSecondaryColor3iEXT(GLint red, GLint green, GLint blue);
void glSecondaryColor3ivEXT(const GLint *v);

void glSecondaryColor3fEXT(GLfloat red,
             GLfloat green, GLfloat blue);
void glSecondaryColor3fvEXT(const GLfloat *v);

void glSecondaryColor3dEXT(GLdouble red,
             GLdouble green, GLdouble blue);
void glSecondaryColor3dvEXT(const *GLdouble v);

void glSecondaryColor3ubEXT(GLubyte red,
             GLubyte green, GLubyte blue);
void glSecondaryColor3ubvEXT(const GLubyte *v);

void glSecondaryColor3usEXT(GLushort red,
             GLushort green, GLushort blue);

void glSecondaryColor3usvEXT(const GLushort *v);

void glSecondaryColor3uiEXT(GLuint red,
             GLuint green, GLuint blue);
void glSecondaryColor3uivEXT(const GLuint *v);
```

```
void glSecondaryColorPointerEXT(int size, enum type, sizei stride,
             void *pointer))
```

PARAMETERS

glSecondaryColor3*EXT

*red, green, blue, *v*
: Red, green, and blue or an array pointer of color component values.

glSecondaryColorPointerEXT

size
: Size of color entry, must be 3.

type
: Type of color data, one of BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE.

stride
: Stride between RGB values in the color data

pointer
: Pointer to color values

NEW TOKENS

Accepted by the <*cap*> parameter of glEnable, glDisable, and glIsEnabled, and by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

COLOR_SUM_EXT

Accepted by the <*pname*> parameter of glGetBooleanv, glGetIntegerv, glGetFloatv, and glGetDoublev:

CURRENT_SECONDARY_COLOR_EXT
SECONDARY_COLOR_ARRAY_SIZE_EXT
SECONDARY_COLOR_ARRAY_TYPE_EXT
SECONDARY_COLOR_ARRAY_STRIDE_EXT

Accepted by the <*pname*> parameter of glGetPointerv:

SECONDARY_COLOR_ARRAY_POINTER_EXT

Accepted by the <*array*> parameter of glEnableClientState and glDisableClientState:

SECONDARY_COLOR_ARRAY_EXT

DESCRIPTION

The various glSecondaryColor3*EXT commands set the secondary color component values. These values will then be used in the color sum stage of color processing. The color sum stage is described in detail in the OpenGL 1.2 specification (available from www.opengl.org).

ERRORS

GL_INVALID_ENUM is generated if **glSecondaryColorPointerEXT** is called with a parameter <*type*> that is not BYTE, UNSIGNED BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE.

GL_INVALID_VALUE is generated if **glSecondaryColorPointerEXT** parameter <*size*> is not 3.

GL_INVALID_VALUE is generated if **glSecondaryColorPointerEXT** parameter <*stride*> is negative.

ASSOCIATED GETS

Calling **glIsEnabled** with an argument of GL_SECONDARY_COLOR_ARRAY_EXT indicates if secondary color is active. Calls to glGetIntegerv and glGetFloatv with the GL_SECONDARY_COLOR_EXT value returns the current secondary color values. Calls to glGetIntegerv with the arguments GL_SECONDARY_COLOR_ARRAY_SIZE_EXT, GL_SECONDARY_COLOR_ARRAY_STRIDE, or GL_SECONDARY_COLOR_ARRAY_TYPE_EXT, returns the secondary color array size, array stride, or array type. Calling glGetPointerv with GL_SECONDARY_COLOR_ARRAY_POINTER_EXT returns a pointer to the color array.

SEE ALSO

The OpenGL 1.2 specification.

EXT_separate_specular_color

The original OpenGL lighting model requires you to apply lighting results to a primitive before you apply any texturing. This can lead to problems when you use specular lighting to add highlights to a model, because the texturing can obscure the effect of the highlights.

The `separate_specular_color` extension that allows you to apply specular lighting after you apply the textures to the geometry.

EXTENSION STRING

GL_EXT_separate_specular_color

PROCEDURE DEFINITIONS

none

NEW TOKENS

Accepted by <*pname*> parameter of glLightModel*:

GL_LIGHT_MODEL_COLOR_CONTROL_EXT

Accepted as a <*param*> parameter of glLightModel*:

GL_SEPARATE_SPECULAR_COLOR_EXT

DESCRIPTION

The separate specular color extension is enabled by modifying the standard OpenGL lighting model. Application codes can do this a follows:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL_EXT,
                GL_SEPARATE_SPECULAR_COLOR_EXT);
```

ERRORS

GL_INVALID_ENUM is generated if pname is not an accepted value.

GL_INVALID_OPERATION is generated if glLightModel is executed between the execution of glBegin and the corresponding execution of glEnd.

ASSOCIATED GETS

Calling **glGet** with an argument of GL_LIGHT_MODEL_COLOR_CONTROL_EXT returns the current mode.

SEE ALSO

glLightModel

SGI_texture_edge_clamp

This extension defines a new texture clamping algorithm. This extension clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel.

EXTENSION STRING

GL_SGI_texture_edge_clamp

PROCEDURE DEFINITIONS

none

NEW TOKENS

Accepted by the <*param*> parameter of glTexParameteri and glTexParameterf, and by the <*params*> parameter of glTexParameteriv and glTexParameterfv, when their <*pname*> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT:

GL_CLAMP_TO_EDGE_SGI

DESCRIPTION

The base OpenGL provides clamping such that the texture coordinates are limited to exactly the range [0,1]. When a texture coordinate is clamped using this algorithm, the texture sampling filter straddles the edge of the texture image, taking 1/2 its sample values from within the texture image, and the other 1/2 from the texture border. It is sometimes desirable to clamp a texture without requiring a border, and without using the constant border color.

This extension defines a new texture clamping algorithm. GL_CLAMP_TO_EDGE_SGI clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. When used with a NEAREST or a LINEAR filter, the color returned when clamping is derived only from texels at the edge of the texture image. When used with FILTER4 filters, the filter operations of GL_CLAMP_TO_EDGE_SGI are defined but don't result in a nice

clamp-to-edge color. GL_CLAMP_TO_EDGE_SGI is supported by 1, 2-dimensional textures only.

CLAMP_TO_EDGE_SGIS texture clamping is specified by calling glTexParameteri with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT, and <param> set to GL_CLAMP_TO_EDGE_SGI.

Let [min,max] be the range of a clamped texture coordinate, and let N be the size of the 1D or 2D texture image in the direction of clamping. Then in all cases

max = 1 - min

because the clamping is always symmetric about the [0,1] mapped range of a texture coordinate. When used with NEAREST or LINEAR filters, GL_CLAMP_TO_EDGE_SGI defines a minimum clamping value of

min = 1 / 2*N

ERRORS

See glTexParameter*

SEE ALSO

glTexParameter*

Glossary

alpha

A fourth pixel component (in addition to red, green and blue) used for blending effects.

API

Application programming interface.

Drawable

Any buffer that either the GDI or OpenGL can render to or read from.

GDI

indows Graphics Device Interface. Microsoft's set of routines for drawing text and graphics to windows.

fast path

Data path through the OpenGL with the highest bandwidth. Typically, all stages are hardware-accelerated.

OpenGL

The standard graphics library used by most of the industry today.

pbuffer

High-performance off-screen pixel buffers in which GL commands can be rendered.

SDK

Software developer kit.

WGL

Windows extensions for OpenGL. The interface between OpenGL and the Windows NT windowing system.

Index

A

API calls
  reducing use of, 30

B

Buffers, 10
  I/O, 12
  Off screen, 12
  Overlay buffer, 11
  RGB types, 10
  Stencil and Z, 10

C

Clip Volume Hint Extension, 32
Compiling Without Extensions, 4

D

Depth Pass (Occlusion) Test Instrument Extension, 32
Display Lists, 31

E

Extending Pixel Formats, 23
Extension Names, 1 – 3
Extensions
  Clip Volume Hint, 32
  Compiling Without, 4
  Complete list, 23
  Depth Pass (Occlusion) Test Instrument, 32
  Names, 1 – 3
  Overview, 1 – 8
  Using, 3 – 8
    OpenGL, 3 – 6
    WGL, 7 – 8
  Versions and Platforms, 23 – 26
  WGL explanation, 23

F

Floating point format
  using for optimization, 29

G

Geometry, 16
  Clip volume hint optimizations, 18
  Occlusion test, 17
glBegin and glEnd pairs, 30
Graphics Hardware Features, 33 – 36
  Directly supported, 33 – 35
  Software implementation required, 36
  Software-assisted, 35

H

Header Files, 3

I

Imaging, 20 – 23
  Color matrix, 22
  Color table, 22
  Convolutions, 22
  Histogram and min-max, 23
  Pipeline stages, 20
  Scale and bias, 22

O

OpenGL
  Using Extensions, 3 – 6
OpenGL calls
  avoiding, 30
  avoiding with display lists, 31
  avoiding with vertex arrays, 31
  using glBegin and glEnd pairs for performance, 30
Optimizations, 29 – 33
  General tuning tips, 29 – 31
    avoiding unnecessary OpenGL calls, 30
    reducing API calls, 30
    using floating point format, 29
  Using OpenGL Extensions, 32

P

Pipeline Stages With Optimized Paths, 21
Pixel, 19 , 23
  Formats for fast paths, 19
  Interlaced pixel operations, 19
Pixel Formats
  Extending, 23

R

Rendering, 14 – 16
  Anti-aliasing, 16
  Blending mode, 14
  Specular highlights, 16
  Texture support, 15

S

SDK, 8
Software Developer Kit (SDK), 8
Software-assisted features, 35

U

Using Extensions, 3 – 8
  OpenGL, 3 – 6
  WGL, 7 – 8

V

Vertex arrays, 31

W

WGL
  Using Extensions, 7 – 8
WGL Extensions, 23
wglCreatePBufferSGIX(), 12
wglMakeCurrent(), 11