NVIDIA OpenGL Extension Specifications

NVIDIA Corporation

Mark J. Kilgard, *editor*
mjk@nvidia.com

May 21, 2001

Copyright NVIDIA Corporation, 1999, 2000, 2001.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation as designated in the document.

Other OpenGL extension specifications can be found at:

http://oss.sgi.com/projects/ogl-sample/registry/

Table of Contents

Table of NVIDIA OpenGL Extension Support 4
ARB_imaging 6
ARB_multisample 7
ARB_multitexture 18
ARB_texture_border_clamp 19
ARB_texture_compression 25
ARB_texture_cube_map 48
ARB_texture_env_add 62
ARB_texture_env_combine 65
ARB_texture_env_dot3 73
ARB_transpose_matrix 76
EXT_abgr 81
EXT_bgra 84
EXT_blend_color 86
EXT_blend_minmax 89
EXT_blend_subtract 92
EXT_compiled_vertex_array 95
EXT_draw_range_elements 98
EXT_fog_coord 101
EXT_packed_pixels 108
EXT_paletted_texture 117
EXT_point_parameters 125
EXT_rescale_normal 130
EXT_secondary_color 133
EXT_separate_specular_color 141
EXT_shared_texture_palette 146
EXT_stencil_wrap 149
EXT_texture_compression_s3tc 151
EXT_texture3D 159
EXT_texture_cube_map 169
EXT_texture_edge_clamp 170
EXT_texture_env_add 173
EXT_texture_env_combine 176
EXT_texture_env_dot3 182
EXT_texture_filter_anisotropic 185
EXT_texture_lod_bias 191
EXT_texture_object 196
EXT_vertex_array 204
EXT_vertex_weighting 216
IBM_texture_mirrored_repeat 227
NV_blend_square 230
NV_evaluators 233
NV_fence 249
NV_fog_distance 258
NV_light_max_exponent 262
NV_multisample_filter_hint 265
NV_packed_depth_stencil 269
NV_register_combiners 277
NV_register_combiners2 305
NV_texgen_emboss 311
NV_texgen_reflection 317
NV_texture_compression_vtc 320
NV_texture_env_combine4 325
NV_texture_rectangle 330
NV_texture_shader 343
NV_texture_shader2 401
NV_vertex_array_range 412
NV_vertex_array_range2 425
NV_vertex_program 428
SGIS_generate_mipmap 506
SGIS_texture_lod 510
SGIX_depth_texture 517
SGIX_shadow 520
WGL_ARB_buffer_region 524
WGL_ARB_extensions_string 530
WGL_ARB_pbuffer 533
WGL_ARB_pixel_format 540
WGL_EXT_swap_control 553

Table of NVIDIA OpenGL Extension Support

Extension	RIVA 128 family	RIVA TNT family	NV1x family	NV2x family	Notes
ARB_imaging		R10	R10	X	
ARB_multisample				X	
ARB_multitexture		X	X	X	
ARB_texture_border_clamp				X	
ARB_texture_compression			X	X	
ARB_texture_cube_map			X	X	
ARB_texture_env_add		X	X	X	
ARB_texture_env_combine			X	X	
ARB_texture_env_dot3			X	X	
ARB_transpose_matrix		X	X	X	
EXT_abgr	X	X	X	X	
EXT_bgra	X	X	X	X	1.2 functionality
EXT_blend_color			X	X	ARB_imaging
EXT_blend_minmax			X	X	ARB_imaging
EXT_blend_subtract			X	X	ARB_imaging
EXT_compiled_vertex_array		X	X	X	
EXT_draw_range_elements		X	X	X	1.2 functionality
EXT_fog_coord		X	X	X	
EXT_packed_pixels	X	X	X	X	1.2 functionality
EXT_paletted_texture			X	X	
EXT_point_parameters	X	X	X	X	
EXT_rescale_normal			X	X	1.2 functionality
EXT_secondary_color		X	X	X	
EXT_separate_specular_color		X	X	X	1.2 functionality
EXT_shared_texture_palette			X	X	
EXT_stencil_wrap	X	X	X	X	
EXT_texture_compression_s3tc			X	X	
EXT_texture3D		sw	sw	X	1.2 functionality
EXT_texture_cube_map			X	X	use ARB version
EXT_texture_edge_clamp		X	X	X	1.2 functionality
EXT_texture_env_add		X	X	X	see ARB version
EXT_texture_env_dot3			X	X	see ARB version
EXT_texture_env_combine		X	X	X	see ARB version
EXT_texture_filter_anisotropic			X	X	
EXT_texture_lod_bias		R10	X	X	
EXT_texture_object	X	X	X	X	1.1 functionality
EXT_vertex_array	X	X	X	X	1.1 functionality
EXT_vertex_weighting		X	X	X	
KTX_buffer_region	X	X	X	X	
IBM_texture_mirrored_repeat		X	X	X	
NV_blend_square		X	X	X	
NV_evaluators		R10	R10	X	
NV_fence			X	X	
NV_fog_distance		X	X	X	
NV_light_max_exponent		X	X	X	
NV_multisample_filter_hint				X	
NV_register_combiners			X	X	
NV_register_combiners2			em	X	
NV_texgen_emboss			X		
NV_texgen_reflection	X	X	X	X	
NV_texture_compression_vtc					
NV_texture_env_combine4		X	X	X	
NV_texture_rectangle			X	X	
NV_texture_shader			em	X	
NV_texture_shader2					
NV_vertex_array_range			X	X	
NV_vertex_array_range2			R10	R10	
NV_vertex_program			R10	X	
SGIS_generate_mipmap			R10	X	
SGIS_multitexture		X	X	X	use ARB version
SGIS_texture_lod			X	X	1.2 functionality
SGIX_depth_texture			em	X	
SGIX_shadow			em	X	
WGL_ARB_buffer_region		X	X	X	Win32
WGL_ARB_extensions_string		X	X	X	Win32
WGL_ARB_pixel_format		R10	R10	X	Win32
WGL_ARB_pbuffer		R10	R10	X	Win32
WGL_EXT_extensions_string		X	X	X	Win32
WGL_EXT_swap_control		X	X	X	Win32
WIN_swap_hint	X	X	X	X	Win32

Key for table entries:

X = *supported*

sw = *supported by software rasterization (expect poor performance)*

em = *like sw, but only supported when "NV20 emulate" mode is enabled using Release 10*

R10 = *introduced in the Release 10 OpenGL driver (not supported by earlier drivers)*

Warning: The extension support columns are based on the latest & greatest NVIDIA driver release (unless otherwise noted). Check your GL_EXTENSIONS string with glGetString at run-time to determine the specific supported extensions for a particular driver version.

Name

ARB_imaging

Name Strings

GL_ARB_imaging

NOTE: This extension does not have its own specification document, since it has been included in the OpenGL 1.2.1 Specification (downloadable from www.opengl.org). Please refer to the 1.2.1 Specification for more information.

Name

ARB_multisample

Name Strings

GL_ARB_multisample
GLX_ARB_multisample
WGL_ARB_multisample

Status

Approved by ARB on 12/8/1999.
GLX protocol must still be defined.

Version

Last Modified Date: December 15, 1999
Author Revision: 0.5

Based on: SGIS_Multisample Specification
Date: 1994/11/22 Revision: 1.14

Number

ARB Extension #5

Dependencies

WGL_EXT_extensions_string is required.
WGL_EXT_pixel_format is required.

Overview

This extension provides a mechanism to antialias all GL primitives: points, lines, polygons, bitmaps, and images. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes depth and stencil information, the depth and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. When the framebuffer includes a multisample buffer, it does not also include separate depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers (left/right, front/back, and aux) do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only points or lines are being rendered, the "smooth" antialiasing mechanism provided by the base GL may result in a higher quality image. This extension is

designed to allow multisample and smooth antialiasing techniques to be alternated during the rendering of a single scene.

IP Status

TBD

Issues

1. Multiple passes have been taken out. Is this acceptable?

   RESOLUTION: Yes. This can be added back with an additional extension if needed.

2. Would SampleAlphaARB be a better name for the function SampleMaskARB? If so, the name SAMPLE_MASK_ARB should also be changed to SAMPLE_ALPHA_ARB.

   RESOLUTION: Names containing "mask" were changed to use "coverage" instead.

3. Should the SampleCoverageARB function be changed to allow blending between more than two objects?

   RESOLUTION: Not addressed by this extension. An additional extension has been proposed that allows a coverage range for each object. The coverage range is a min and max value that can be used to blend multiple objects at different level-of-detail fading. The SampleCoverageARB function will layer on this new extension.

New Procedures and Functions

```
void SampleCoverageARB(clampf value,
                       boolean invert);
```

New Tokens

Accepted by the <attribList> parameter of glXChooseVisual, and by the <attrib> parameter of glXGetConfig:

GLX_SAMPLE_BUFFERS_ARB	100000
GLX_SAMPLES_ARB	100001

Accepted by the <piAttributes> parameter of wglGetPixelFormatAttribivEXT, wglGetPixelFormatAttribfvEXT, and the <piAttribIList> and <pfAttribIList> of wglChoosePixelFormatEXT:

WGL_SAMPLE_BUFFERS_ARB	0x2041
WGL_SAMPLES_ARB	0x2042

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MULTISAMPLE_ARB	0x809D
SAMPLE_ALPHA_TO_COVERAGE_ARB	0x809E
SAMPLE_ALPHA_TO_ONE_ARB	0x809F
SAMPLE_COVERAGE_ARB	0x80A0

Accepted by the <mask> parameter of PushAttrib:

MULTISAMPLE_BIT_ARB	0x20000000

Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv, and GetFloatv:

SAMPLE_BUFFERS_ARB	0x80A8
SAMPLES_ARB	0x80A9
SAMPLE_COVERAGE_VALUE_ARB	0x80AA
SAMPLE_COVERAGE_INVERT_ARB	0x80AB

Additions to Chapter 2 of the 1.2.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2.1 Specification (Rasterization)

If SAMPLE_BUFFERS_ARB is a value of one, the rasterization of all GL primitives is changed, and is referred to as multisample rasterization. Otherwise, primitive rasterization operates as it is described in the GL specification, and is referred to as single-sample rasterization. The value of SAMPLE_BUFFERS_ARB is an implementation dependent constant, and is queried by calling GetIntegerv with <pname> set to SAMPLE_BUFFERS_ARB. This value is the same as GLX_SAMPLE_BUFFERS_ARB or WGL_SAMPLE_BUFFERS_ARB for the visual or pixel format associated with the context.

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with SAMPLES_ARB bits. The value of SAMPLES_ARB is an implementation-dependent constant, and is queried by calling GetIntegerv with <pname> set to SAMPLES_ARB. Second, each fragment includes SAMPLES_ARB depth values, instead of the single depth value that is maintained in single-sample rendering mode. Each pixel fragment thus consists of integer x and y grid coordinates, a color, SAMPLES_ARB depth values, texture coordinates, and a coverage value with a maximum of SAMPLES_ARB bits.

The behavior of multisample rasterization is a function of MULTISAMPLE_ARB, which is enabled and disabled by calling Enable or Disable, with <cap> set to MULTISAMPLE_ARB. Its value is queried using IsEnabled, with <cap> set to MULTISAMPLE_ARB.

If MULTISAMPLE_ARB is disabled, multisample rasterization of all primitives is equivalent to single-sample rasterization, except that the fragment coverage value is set to full coverage. The depth values may all be set to the single value that would have

been assigned by single-sample rasterization, or they may be assigned as described below for multisample rasterization.

If MULTISAMPLE_ARB is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has SAMPLES_ARB locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel may be located inside or outside of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement described in section 3.1 is relaxed for all enabled multisample rendering, because the sample locations may be a function of pixel location.

It is not possible to query the actual sample locations of a pixel.

Point Multisample Rasterization
[Insert before section 3.3.1]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (POINT_SMOOTH) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the region lying within the circle having diameter equal to the current point width and centered at the point's (Xw,Yw). Coverage bits that correspond to sample points that intersect the circular region are 1, other coverage bits are 0. All depth values of the fragment are assigned the depth value of the point being rasterized. Other data associated with each fragment are the data associated with the point being rasterized.

Point size range and number of gradations are equivalent to those supported for antialiased points.

Line Multisample Rasterization
[Insert before section 3.4.3]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (LINE_SMOOTH) is enabled or disabled. Line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the rectangular region that is described in the Antialiasing section of 3.4.2 (Other Line Segment Features). If line stippling is enabled, the rectangular region is subdivided into adjacent unit-length rectangles, with some rectangles eliminated according to the procedure given under Line Stipple, where "fragment" is replaced by "rectangle".

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each depth value is produced by substituting the corresponding sample location into equation 3.1, then using the result to evaluate equation 3.3. The data associated with each fragment are otherwise computed by evaluating equation 3.1 at the fragment center, then substituting into equation 3.2.

Line width range and number of gradations are equivalent to those supported for antialiased lines.

Polygon Multisample Rasterization
[Insert before section 3.5.6]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (POLYGON_SMOOTH) is enabled or disabled. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 3.5.1, including the special treatment for sample points that lie on a polygon boundary edge. If a polygon is culled, based on its orientation and the CullFace mode, then no fragments are produced during rasterization. Fragments are culled by the polygon stipple just as they are for aliased and antialiased polygons.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each depth value is produced by substituting the corresponding sample location into the barycentric equations described in section 3.5.1, using the approximation to equation 3.4 that omits w components. The data associated with each fragment are otherwise computed by barycentric evaluation using the fragment's center point.

The rasterization described above applies only to the FILL state of PolygonMode. For POINT and LINE, the rasterizations described in the Point Multisample Rasterization and the Line Multisample Rasterization sections apply.

Pixel Rectangle Multisample Rasterization
[Insert before section 3.6.5]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then pixel rectangles are rasterized using the following algorithm. Let (Xrp,Yrp) be the current raster position. (If the current raster position is invalid, then DrawPixels is ignored.) If a particular group (index or components) is the nth in a row and belongs to the mth row, consider the region in window coordinates bounded by the rectangle with corners

```
  (Xrp + Zx*n, Yrp + Zy*m)
```

and

```
  (Xrp + Zx*(n+1), Yrp + Zy*(m+1))
```

where Zx and Zy are the pixel zoom factors specified by PixelZoom,

and may each be either positive or negative. A fragment representing group n,m is produced for each framebuffer pixel with one or more sample points that lie inside, or on the bottom or left boundary, of this rectangle. Each fragment so produced takes its associated data from the group and from the current raster position, in a manner consistent with the discussion in the Conversion to Fragments subsection of section 3.6.4 of the GL specification. All depth sample values are assigned the same value, taken either from the group (if it is a depth component group) or from the current raster position (if it is not).

A single pixel rectangle will generate multiple, perhaps very many fragments for the same framebuffer pixel, depending on the pixel zoom factors.

Bitmap Multisample Rasterization
[Insert at the end section 3.7]

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is a value of one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower-left corner at (Xrp,Yrp), and its upper right corner at (Xrp+w,Yrp+h), where w and h are the width and height of the bitmap. Rectangles in this array are eliminated if the corresponding bit in the bitmap is zero, and are retained otherwise. Bitmap rasterization produces a fragment for each framebuffer pixel with one or more sample points either inside or on the bottom or left edge of a retained rectangle.

Coverage bits that correspond to sample points either inside or on the bottom or left edge of a retained rectangle are 1, other coverage bits are 0. The associated data for each fragment are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated exactly as it is in the single-sample rasterization case.

Additions to Chapter 4 of the 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Multisample Fragment Operations
[Insert after section 4.1.2]

This step modifies fragment alpha and coverage values based on the values of SAMPLE_ALPHA_TO_COVERAGE_ARB, SAMPLE_ALPHA_TO_ONE_ARB, SAMPLE_COVERAGE_ARB, SAMPLE_COVERAGE_VALUE_ARB, and SAMPLE_COVERAGE_INVERT_ARB. No changes to the fragment alpha or coverage values are made at this step if MULTISAMPLE_ARB is disabled, or if SAMPLE_BUFFERS_ARB is not a value of one.

SAMPLE_ALPHA_TO_COVERAGE_ARB, SAMPLE_ALPHA_TO_ONE_ARB, and SAMPLE_COVERAGE_ARB are enabled and disabled by calling Enable and Disable with <cap> specified as one of the three token values. All three values are queried by calling IsEnabled, with <cap> set to the desired token value. If SAMPLE_ALPHA_TO_COVERAGE_ARB is enabled, the fragment alpha value is used to generate a temporary coverage value, which is then ANDed with the fragment coverage

value. Otherwise the fragment coverage value is unchanged at this point.

This specification does not require a specific algorithm for converting an alpha value to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the alpha value, with all 1's corresponding to the maximum alpha value, and all 0's corresponding to an alpha value of 0. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if SAMPLE_ALPHA_TO_ONE_ARB is enabled, fragment alpha is replaced by the maximum representable alpha value. Otherwise, fragment alpha value is not changed.

Finally, if SAMPLE_COVERAGE_ARB is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of SAMPLE_COVERAGE_VALUE_ARB. The function need not be identical, but it must have the same properties of proportionality and invariance. If SAMPLE_COVERAGE_INVERT_ARB is TRUE, the temporary coverage is inverted (all bit values are inverted) before it is ANDed with the fragment coverage.

The values of SAMPLE_COVERAGE_VALUE_ARB and SAMPLE_COVERAGE_INVERT_ARB are specified simultaneously by calling SampleCoverageARB, with <value> set to the desired coverage value, and <invert> set to TRUE or FALSE. <value> is clamped to [0,1] before being stored as SAMPLE_COVERAGE_VALUE_ARB.
SAMPLE_COVERAGE_VALUE_ARB is queried by calling GetFloatv with <pname> set to SAMPLE_COVERAGE_VALUE_ARB.
SAMPLE_COVERAGE_INVERT_ARB is queried by calling GetBooleanv with <pname> set to SAMPLE_COVERAGE_INVERT_ARB.

Multisample Fragment Operations
[Insert after section 4.1.8]

If the DrawBuffers mode is NONE, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

If MULTISAMPLE_ARB is enabled, and SAMPLE_BUFFERS_ARB is one, the stencil test, depth test, blending, and dithering operations are performed for each pixel sample, rather than just once for each fragment. Failure of the stencil or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample buffer (to be described in a following section). The contents of the color buffers are not modified at this point.

Stencil, depth, blending, and dithering operations are performed for a pixel sample only if that sample's fragment coverage bit is

a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample. Depth operations use the fragment depth value that is specific for each sample. The single fragment color value is used for all sample operations, however, as is the current stencil value.

If MULTISAMPLE_ARB is disabled, and SAMPLE_BUFFERS_ARB is one, the fragment may be treated exactly as described above, with optimization possible because the fragment coverage must be set to full coverage. Further optimization is allowed, however. An implementation may choose to identify a centermost sample, and to perform stencil and depth tests on only that sample. Regardless of the outcome of the stencil test, all multisample buffer stencil sample values are set to the appropriate new stencil value. If the depth test passes, all multisample buffer depth sample values are set to the depth of the fragment's centermost sample's depth value, and all multisample buffer color sample values are set to the color value of the incoming fragment. Otherwise, no change is made to any multisample buffer color or depth value.

After all operations have been completed on the multisample buffer, the color sample values are combined to produce a single color value, and that value is written into each color buffer that is currently enabled, based on the DrawBuffers mode. An implementation may defer the writing of the color buffer until a later time, but the state of the framebuffer must behave as if the color buffer was updated as each fragment was processed. The method of combination is not specified, though a simple average computed independently for each color component is recommended.

Fine Control of Multisample Buffer Updates
[Insert at the end of section 4.2.2]

When SAMPLE_BUFFERS_ARB is one, ColorMask, DepthMask, and StencilMask control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by DrawBuffers.

Clearing the Multisample Buffer
[Insert as a subsection for section 4.2.3]

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the Clear mask bit COLOR_BUFFER_BIT and the DrawBuffers mode. If the DrawBuffers mode is NONE, the color samples of the multisample buffer cannot be cleared.

Clear mask bits DEPTH_BUFFER_BIT and STENCIL_BUFFER_BIT indicate that the depth and stencil samples of the multisample buffer are to be cleared. If Clear mask bit DEPTH_BUFFER_BIT is specified, and if the DrawBuffers mode is not NONE, then the multisample depth buffer samples are cleared. Likewise, if Clear mask bit STENCIL_BUFFER_BIT is specified, and if the DrawBuffers mode is not NONE, then the multisample stencil buffer is cleared.

Reading Pixels
[These changes are made to the text in section 4.3.2, following the subheading Obtaining Pixels from the Framebuffer.]

Follow the sentence "If there is no depth buffer, the error INVALID_OPERATION occurs." with: If there is a multisample buffer (SAMPLE_BUFFERS_ARB is 1) then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

Follow the sentence "if there is no stencil buffer, the error INVALID_OPERATION occurs." with: If there is a multisample buffer, then values are obtained from the stencil samples in this buffer. It is recommended that the stencil value of the centermost sample be used, though implementations may choose any function of the stencil sample values at each pixel.

[This extension makes no change to the way that color values are obtained from the framebuffer.]

Additions to Chapter 5 of the 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

An additional group of state variables, MULTISAMPLE_BIT_ARB, is defined by this extension. When PushAttrib is called with bit MULTISAMPLE_BIT_ARB set, the multisample group of state variables is pushed onto the attribute stack. When PopAttrib is called, these state variables are restored to their previous values if they were pushed. Some multisample state is included in the ENABLE_BIT group as well. In order to avoid incompatibility with GL implementations that do not support SGIS_multisample, ALL_ATTRIB_BITS does not include MULTISAMPLE_BIT_ARB.

Additions to the GLX Specification

The parameter GLX_SAMPLE_BUFFERS_ARB is added to glXGetConfig. When queried, by calling glXGetConfig with <attrib> set to GLX_SAMPLE_BUFFERS_ARB, it returns the number of multisample buffers included in the visual. For a normal visual, the return value is zero. A return value of one indicates that a single multisample buffer is available. The number of samples per pixel is queried by calling glXGetConfig with <attrib> set to GLX_SAMPLES_ARB. It is understood that the number of color, depth, and stencil bits per sample in the multisample buffer are as specified by the GLX_*_SIZE parameters. It is also understood that there are no single-sample depth or stencil buffers associated with this visual -- the only depth and stencil buffers are those in the multisample buffer. GLX_SAMPLES_ARB is zero if GLX_SAMPLE_BUFFERS_ARB is zero.

glXChooseVisual accepts GLX_SAMPLE_BUFFERS_ARB in <attribList>, followed by the minimum number of multisample buffers that can be accepted. Visuals with the smallest number of multisample buffers that meets or exceeds the specified minimum number are preferred. Currently operation with more than one multisample buffer is undefined, so the returned value will be either zero or one.

glXChooseVisual accepts GLX_SAMPLES_ARB in <attribList>, followed by the minimum number of samples that can be accepted in the multisample buffer. Visuals with the smallest number of samples that meets or exceeds the specified minimum number are preferred.

If the color samples in the multisample buffer store fewer bits than are stored in the color buffers, this fact will not be reported accurately. Presumably a compression scheme is being employed, and is expected to maintain an aggregate resolution equal to that of the color buffers.

GLX Protocol

TBD

Additions to the WGL Specification

The parameter WGL_SAMPLE_BUFFERS_ARB is added to wglGetPixelFormatAttrib*v. When queried, by calling wglGetPixelFormatAttrib*v with <piAttributes> set to WGL_SAMPLE_BUFFERS_ARB, it returns the number of multisample buffers included in the pixel format. For a normal pixel format, the return value is zero. A return value of one indicates that a single multisample buffer is available. The number of samples per pixel is queried by calling wglGetPixelFormatAttrib*v with <piAttributes> set to WGL_SAMPLES_ARB. It is understood that the number of color, depth, and stencil bits per sample in the multisample buffer are as specified by the WGL_*_SIZE parameters. It is also understood that there are no single-sample depth or stencil buffers associated with this visual -- the only depth and stencil buffers are those in the multisample buffer. WGL_SAMPLES_ARB is zero if WGL_SAMPLE_BUFFERS_ARB is zero.

wglChoosePixelFormatEXT accepts WGL_SAMPLE_BUFFERS_ARB in <piAttribIList> and <pfAttribIList> with the corresponding value set to the minimum number of multisample buffers that can be accepted. Pixel formats with the smallest number of multisample buffers that meets or exceeds the specified minimum number are preferred. Currently operation with more than one multisample buffer is undefined, so the returned value will be either zero or one.

If the color samples in the multisample buffer store fewer bits than are stored in the color buffers, this fact will not be reported accurately. Presumably a compression scheme is being employed, and is expected to maintain an aggregate resolution equal to that of the color buffers.

Errors

INVALID_OPERATION is generated if SampleCoverageARB is called between the execution of Begin and the execution of the corresponding End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
MULTISAMPLE_ARB	IsEnabled	B	TRUE	multisample/enable
SAMPLE_ALPHA_TO_COVERAGE_ARB	IsEnabled	B	FALSE	multisample/enable
SAMPLE_ALPHA_TO_ONE_ARB	IsEnabled	B	FALSE	multisample/enable
SAMPLE_COVERAGE_ARB	IsEnabled	B	FALSE	multisample/enable
SAMPLE_COVERAGE_VALUE_ARB	GetFloatv	R+	1	multisample
SAMPLE_COVERAGE_INVERT_ARB	GetBooleanv	B	FALSE	multisample

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
SAMPLE_BUFFERS_ARB	GetIntegerv	Z+	0
SAMPLES_ARB	GetIntegerv	Z+	0

Conformance Testing

TBD

Revision History

09/20/1999 0.1
- First ARB draft based on the original SGI draft.

10/1/1999 0.2
- Added query for the number of passes.

11/8/1999 0.3
- Fixed numerous typos reported by E&S.

12/7/1999 0.4
- Removed the multiple pass feature.
- Resolved the working group issues at the ARB meeting.
- Added language that stated that SAMPLE_BUFFERS_ARB is the same value as either GLX_SAMPLE_BUFFERS_ARB or WGL_SAMPLE_BUFFERS_ARB.

12/15/1999 0.5
- Added back in the statement about ALL_ATTRIB_BITS not including MULTISAMPLE_BIT_ARB.

Name Strings

ARB_multitexture

Name Strings

GL_ARB_multitexture

Status

Complete. Approved by ARB on 9/15/1998

NOTE: This extension no longer has its own specification document, since it has been included in the OpenGL 1.2.1 Specification (downloadable from www.opengl.org). Please refer to the 1.2.1 Specification for more information.

Name

ARB_texture_border_clamp

Name Strings

GL_ARB_texture_border_clamp

Status

DRAFT VERSION ONLY -- FOR OPENGL ARB CONSIDERATION AT 6/2000 MEETING

Version

0.3, 2 June 2000 (Alternate Formulation)

Number

!!! To be assigned when added to registry

Dependencies

OpenGL 1.0 is required.

This extension is written against the OpenGL 1.2.1 Specification.

This extension is based on and intended to replace GL_SGIS_texture_border_clamp.

Overview

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 for a texture to be clamped to the border color, rather than to an average of the border and edge colors.

This extension defines an additional texture clamping algorithm. CLAMP_TO_BORDER_ARB clamps out-of-bounds texture accesses at all mipmap levels and LINEAR filters return only the color of the border texels.

IP Status

No known IP issues.

Issues

(1) This specification could be written to clamp the s, t, and r values to -1/2^N, 1+1/2^N in the manner similar to the formulation of CLAMP_TO_EDGE in the OpenGL 1.2.1 specification. Such a formulation does not work properly in the presence of cubic and anisotropic filters. While specifications for such filters could correct this problem, should this specification take care of it instead?

UNRESOLVED: Yes. This specification formulates texture filtering in a manner consistent with other filters. In addition, the formulation of texture clamping for CLAMP_TO_EDGE and CLAMP_TO_BORDER in the SGIS_texture_border_clamp extensions are not very straightforward.

New Procedures and Functions

None.

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R:

CLAMP_TO_BORDER_ARB	0x812D

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify Table 3.17, p. 124, editing only the following lines:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER_ARB
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER_ARB
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER_ARB

Delete Sections 3.8.4 (Texture Wrap Modes), 3.8.5 (Texture Minification), and 3.8.6 (Texture Magnification). Replace with a single section (3.8.4, Texture Filtering).

Begin with single-paragraph introduction copied from first paragraph of old Section 3.8.5 (p.125)

Add minor subsection of new Section 3.8.4, Coordinate Clamping

If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R is CLAMP, the GL clamps the s, t, or r coordinates, respectively to the range [0,1]. Otherwise, the s, t, or r coordinates are unmodified.

Add minor subsection of new Section 3.8.4, Scale Factor and Level of Detail

Copy text from the beginning with the corresponding subsection of old Section 3.8.5 (p. 125) through the end of the first paragraph on p. 127, removing the first paragraph from page 126 ("If lambda(x,y) is less than...").

The GL selects a texture filter using the computed value of lambda. The

GL always selects the magnification filter (given by the value of TEXTURE_MAG_FILTER) if lambda is less than zero. It also selects the magnification filter if lambda is less than 0.5, the magnification filter is LINEAR, and the minification filter (given by the value of TEXTURE_MIN_FILTER) is NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise, the GL selects the minification filter.

Add minor subsection of new Section 3.8.4, Mipmapping

Copy text from the beginning of the corresponding subsection of the old Section 3.8.5 (p. 129) through the end of the second paragraph on p. 130 (ending with "if either value is negative"). Replace references to TEXTURE_MIN_FILTER with "the texture filter").

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. The GL uses the texture filter to generate filtered results from one or multiple mipmap arrays. If multiple mipmap arrays are used, the filtered results for each mipmap array are combined to yield a final filtered texture value.

If the texture filter is NEAREST or LINEAR, the GL uses the mipmap array specified by TEXTURE_BASE_LEVEL.

If the texture filter is NEAREST_MIPMAP_NEAREST, or LINEAR_MIPMAP_NEAREST, the GL uses the level d mipmap array, where

```
d = ceil(b + lambda + 1/2) - 1,
```

and d is clamped to the range [TEXTURE_BASE_LEVEL, q].

If the texture filter is NEAREST_MIPMAP_LINEAR, or LINEAR_MIPMAP_LINEAR, the GL uses the level d1 and d2 mipmap arrays, where

```
d1 = floor(b + lambda),
d2 = d1 + 1,
```

and d1 and d2 are both clamped to the range [TEXTURE_BASE_LEVEL, q]. When combining the filtered results from the two mipmap arrays, the GL uses the weights w1 and w2, where

```
w1 = 1 - (lambda - floor(lambda)), and
w2 = 1 - w1.
```

Add minor subsection of new Section 3.8.4, Sample Generation

For each mipmap array used to produce a final texture value, the GL generates one or multiple samples. Each sample consists of a set of coordinates and a weight used to combine the samples. As in the computation of lambda, the GL scales the fragment's (s,t,r) texture coordinates to produce a (u,v,w) coordinate, where

```
    u = s * width_t,
    v = t * height_t,
    w = r * depth_t,
```

where width_t, height_t, and depth_t are the width, height, and depth of the mipmap array, excluding any texture borders.

When the texture filter is NEAREST, NEAREST_MIPMAP_NEAREST, or NEAREST_MIPMAP_LINEAR, a single sample is generated. Let

```
i = floor(u),
j = floor(v), and
k = floor(w).
```

The coordinates of the sample are (i) for a one-dimensional texture, (i,j) for a two-dimensional texture, and (i,j,k) for a three-dimensional texture. The weight for the single sample is always 1.0.

When the texture filter is LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, multiple samples are generated. Let

```
i0 = floor(u - 1/2),
i1 = i0 + 1,
j0 = floor(v - 1/2),
j1 = j0 + 1,
k0 = floor(w - 1/2), and
k1 = k0 + 1.
```

For a one-dimensional texture, the GL generates two samples with coordinates (i0) and (i1). For a two-dimensional texture, the GL generates four samples with coordinates (i0,j0), (i1,j0), (i0,j1), and (i1,j1). For a three-dimensional texture, the GL generates eight samples with coordinates (i0,j0,k0), (i1,j0,k0), (i0,j1,k0), (i1,j1,k0), (i0,j0,k1), (i1,j0,k1), (i0,j1,k1), and (i1,j1,k1).

To generate sample weights, let

```
wi0 = 1 - ((u - 1/2) - i0),
wi1 = 1 - wi0,
wj0 = 1 - ((v - 1/2) - j0),
wj1 = 1 - wj0,
wk0 = 1 - ((w - 1/2) - k0), and
wk1 = 1 - wk0.
```

For a one-dimensional texture, the weights of the two samples are wi0 and wi1, respectively. For a two-dimensional texture, the weights of the four samples are wi0*wj0, wi1*wj0, wi0*wj1, and wi1*wj1, respectively. For a three-dimensional texture, the weights of the eight samples are wi0*wj0*wk0, wi1*wj0*wk0, wi0*wj1*wk0, wi1*wj1*wk0, wi0*wj0*wk1, wi1*wj0*wk1, wi0*wj1*wk1, and wi1*wj1*wk1, respectively.

Add minor subsection of new Section 3.8.4, Sample Coordinate Processing

For each sample, the sample coordinates may fall outside the extents of the mipmap array, and may need be modified according to the texture wrap modes. The texture wrap modes are used to generate new coordinates (i'),

```
(i',j'), or (i',j',k'), where

    i' = { i mod width_t,  TEXTURE_WRAP_S is REPEAT,
         { 0,              TEXTURE_WRAP_S is CLAMP_TO_EDGE, i<0,
         { width_t - 1,    TEXTURE_WRAP_S is CLAMP_TO_EDGE, i>=width_t,
         { i,              otherwise.

    j' = { j mod height_t, TEXTURE_WRAP_T is REPEAT,
         { 0,              TEXTURE_WRAP_T is CLAMP_TO_EDGE, j<0,
         { height_t - 1,   TEXTURE_WRAP_T is CLAMP_TO_EDGE, j>=height_t,
         { j,              otherwise.

    k' = { k mod depth_t,  TEXTURE_WRAP_R is REPEAT,
         { 0,              TEXTURE_WRAP_R is CLAMP_TO_EDGE, k<0,
         { depth_t - 1,    TEXTURE_WRAP_R is CLAMP_TO_EDGE, k>=depth_t,
         { k,              otherwise.
```

Out-of-range samples in CLAMP and CLAMP_TO_BORDER modes will require clamping, but are not modified here. They are accounted for in the sample lookup section below.

Add minor subsection of new Section 3.8.4, Sample Lookup

For each sample, a texture sample color, tau_sample, is generated by using the texel at location (i'), (i',j'), or (i',j',k') in the corresponding mipmap array. If the sample coordinate falls outside the range of texels in the mipmap array, the border color given by the current value of the TEXTURE_BORDER_COLOR is used instead. A sample coordinate is outside the range of texels in a mipmap if i'<-b_s, i'>width_t+b_s, j'<-b_s, j'>height_t+b_s, k'<-b_s, or k'>depth_t+b_s), where b_s is the value of TEXTURE_BORDER for the mipmap. If the texture border color is used, the RGBA values of the TEXTURE_BORDER_COLOR are interpreted to match the texture's internal format in a manner consistent with table 3.15.

Add minor subsection of new Section 3.8.4, Sample Filtering

The texture sample colors for each mipmap array are filtered to generate a single texture mipmap color, tau_mipmap, given by

$$\tau_{mipmap} = \sum_{samples} (\tau_{sample} * w_{sample}).$$

Each component of the texture is summed separately over all the samples, using the weights and texture sample colors of each sample.

If multiple mipmaps are used, the tau_mipmap values are filtered to yield a final texture color, tau, given by

$$\tau = \sum_{mipmaps} (\tau_{mipmap} * w_{mipmap}).$$

Again, each component of the texture is summed separately over all the mipmap arrays, using the weights and texture mipmap colors of each mipmap array.

If a single mipmap is used, the final texture color is given by the single tau_mipmap value.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

None.

New State

Only the type information changes for these parameters.

(table 6.13, p. 203)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_WRAP_S	3+ x Z4	GetTexParameter	REPEAT	Texture wrap	3.8	texture
TEXTURE_WRAP_T	3+ x Z4	GetTexParameter	REPEAT	Texture wrap	3.8	texture
TEXTURE_WRAP_R	3+ x Z4	GetTexParameter	REPEAT	Texture wrap	3.8	texture

Revision History

0.3, 06/02/2000 prbrown1: Rewrote texture filtering sections to interact properly with higher-order filters and also to be more easily understood.

0.2, 05/23/2000 prbrown1: Removed dependency on SGIS_texture_filter4 per ARB guidelines.

0.1, 05/02/2000 prbrown1: Initial revision -- mostly stolen from GL_SGIS_texture_border_clamp.

Name

ARB_texture_compression

Name Strings

GL_ARB_texture_compression

Status

FINAL VERSION -- APPROVED BY OPENGL ARB, 3/16/2000.

Version

Final 1.03, 23 May 2000 (supersedes Final 1.0, 24 March 2000 -
contains a few minor fixes documented in
the Revision History below).

Number

ARB Extension #12

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 1.2.1 Specification.

This extension is written against the GLX Extensions for OpenGL
Specification (Version 1.3).

Depends on GL_ARB_texture_cube_map, as cube maps may be stored in
compressed form.

Overview

Compressing texture images can reduce texture memory utilization and
improve performance when rendering textured primitives. This extension
allows OpenGL applications to use compressed texture images by providing:

(1) A framework upon which extensions providing specific compressed
image formats can be built.

(2) A set of generic compressed internal formats that allow
applications to specify that texture images should be stored in
compressed form without needing to code for specific compression
formats.

An application can define compressed texture images by providing a texture
image stored in a specific compressed image format. This extension does
not define any specific compressed image formats, but it does provide the
mechanisms necessary to enable other extensions that do.

An application can also define compressed texture images by providing an
uncompressed texture image but specifying a compressed internal format.
In this case, the GL will automatically compress the texture image using
the appropriate image format. Compressed internal formats can either be

specific (as above) or generic. Generic compressed internal formats are not actual image formats, but are instead mapped into one of the specific compressed formats provided by the GL (or to an uncompressed base internal format if no appropriate compressed format is available). Generic compressed internal formats allow applications to use texture compression without needing to code to any particular compression algorithm. Generic compressed formats allow the use of texture compression across a wide range of platforms with differing compression algorithms and also allow future GL implementations to substitute improved compression methods transparently.

Compressed texture images can be obtained from the GL in uncompressed form by calling GetTexImage and in compressed form by calling GetCompressedTexImageARB. Queried compressed images can be saved and later reused by calling CompressedTexImage[123]DARB. Pre-compressed texture images do not need to be processed by the GL and should significantly improve texture loading performance relative to uncompressed images.

This extension does not define specific compressed image formats (e.g., S3TC, FXT1), nor does it provide means to encode or decode such images. To support images in a specific compressed format, a hardware vendor would:

(1) Provide a new extension defininig specific compressed <internalformat> and <format> tokens for TexImage[123]D, TexSubImage[123]D, CopyTexImage[12]D, CompressedTexImage[123]DARB, CompressedTexSubImage[123]DARB, and GetCompressedTexImageARB calls.

(2) Specify the encoding of compressed images of that specific format.

(3) Specify a method for deriving the size of compressed images of that specific format, using the <internalformat>, <width>, <height>, <depth> parameters, and (if necessary) the compressed image itself.

IP Status

No known intellectual property issues on this general extension.

Specific compression algorithms used to implement this extension (and any other specific texture compression extensions) may be protected and require licensing agreements.

Issues

(1) Should we define additional internal formats that strongly tie an underlying compression algorithm to the format?

RESOLVED: Not here. Explicit compressed formats will be provided by other extensions built on top of this one.

(2) Should we provide additional compression state that gives more control on the level/quality of compression? If so, how?

RESOLVED: Yes, as a hint. Could have also been implemented as a [0.0, 1.0] floating-point TexParameter "quality" state variable (such as the JPEG quality scale found in many apps). This control will affect only

the speed (and quality) with which a driver compresses incoming images, but will not affect the compressed image format selected by the driver.

As the spec is currently formulated, the requirement that quality control not affect compression format selection could have been relaxed by loosening the invariance requirements (so that the quality control can affect the choice of internal format). The risk was the potential for subtle mipmap consistency issues if the hint changes.

(3) Most current compression algorithms handle primarily RGB and RGBA images. Does it make sense having generic compressed formats for alpha, intensity, luminance, and luminance-alpha?

RESOLVED: Yes. It is conceivable that some or all of these formats may be compressed. Implementations not having compression algorithms for these formats can simply choose not to compress and use the appropriate base internal format instead.

(4) Full GetTexImage support requires that the renderer decompress the whole image. Should this extra implementation burden be imposed on the renderer?

RESOLVED: Yes, returning the uncompressed image is a useful feature for evaluating the quality of the compressed image. A decompression engine may also be required for a number of other areas, including software rasterization.

(5) Full TexSubImage support may require that the renderer decompress portions of the image (or perhaps the whole image), do a merge, and then recompress. Even if this were done, portions of the image outside the "modified" area may also be modified due to lossy compression. Should this extra implementation burden be imposed on the renderer?

RESOLVED: No. To avoid the complications involved with modifying a compressed texture image, only the lower-left corner may be modified by TexSubImage. In addition, after calling TexSubImage, the "unmodified" portion of the image is left undefined. An INVALID_OPERATION error results from any other TexSubImage calls.

This behavior allows for the use of compressed images whose dimensions are not powers of two, which TexImage will not accept. The recommended sequence of calls for defining such images is to first call TexImage with a NULL <data> pointer and the image size parameters padded out to the next power of two, and then call CompressedTexSubImageARB or TexSubImage with <xoffset>, <yoffset>, and <zoffset> parameters of zero and the compressed data pointed to by <data>. This behavior also allows TexSubImage to be used as a light-weight replacement of TexImage, where only the image contents are modified.

Certain compressed formats may allow a wider variety of edits -- their specifications will document the restrictions under which these edits are permitted. it is impossible to document such restrictions for unknown generic formats. It is desirable to keep the behavior of generic formats and the specific formats they map to as consistent as possible.

(6) What do the return values of the component sizes (RED_BITS, GREEN_BITS, ...) give for compressed textures? Compressed proxy textures?

RESOLVED: Some behavior has to be defined. For both normal and proxy textures, we return the bit depths of an uncompressed sized image that would most closely match the quality of the compression algorithm for an "average" texture image. Since compressed image quality is highly data dependent, the actual compressed image quality may be better or worse than the renderer's best guess at the best matching sized internal format. To implement this feature in a driver, it is expected that an error analysis would be done on a set of representative images, and the resultant "equivalent bit depths" would be hardwired constants.

(7) What should GetTexLevelParameter with TEXTURE_COMPRESSED_IMAGE_SIZE_ARB return for existing uncompressed formats? For proxy textures?

RESOLVED: For both, an INVALID_OPERATION error results. The actual image to be compressed is not available for proxies, so actually compressing the specified image is not an option.

For uncompressed internal formats, we could return the actual amount of memory taken by the texture image. Such a mechanism might be useful as a metric of "how much space does this texture image take". It's not particularly useful for an application based texture management scheme, since there is no information available indicating the amount of available memory. In addition, because of implementation-dependent hardware constraints, the amount of texture memory consumed by a texture object is not necessarily equal to the sum of the memory consumed by each of its mipmaps. The OpenGL ARB decided against adopting this behavior when this specification was approved.

(8) What about texture borders?

RESOLVED: Not a problem for generic compressed formats since a base internal format can be used if borders are not supported in the compressed image format. Borders may pose problems for specific compression extensions, and compressed textures with borders might well be disallowed by those extensions.

(9) Should certain pixel operations be disallowed for compressed texture internal formats (e.g., PixelStorage, PixelTransfer)? What about byte swapping?

RESOLVED: For uncompressed source images, all pixel storage and pixel transfer modes will be applied prior to compression. For compressed source images, all pixel storage and transfer modes will be ignored. The encoding of compressed images should be specified as a byte stream that matches the disk file format defined for the corresponding image type.

(10) Should functionality be provided to allow applications to save compressed images to disk and reuse them in subsequent runs without programming to specific formats? If so, how?

RESOLVED: Yes. This can be done without knowledge of specific compression formats in the following manner:

* Call TexImage with an uncompressed image and a generic compressed internal format. The texture image will be compressed by the GL, if possible.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_COMPRESSED_ARB to determine if the GL was able to store the image in compressed form.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_INTERNAL_FORMAT to determine the specific compressed image format in which the image is stored.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB to determine the size (in ubytes) of the compressed image that will be returned by the GL. Allocate a buffer of at least this size.

* Call GetCompressedTexImageARB. The GL will write the compressed texture image into the allocated buffer.

* Save the returned compressed image to disk, along with the associated width, height, depth, border parameters and the returned values of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB and TEXTURE_INTERNAL_FORMAT.

* Load the compressed image and its parameters, and call CompressedTexImage_[123]DARB to use the compressed image. The value of TEXTURE_INTERNAL_FORMAT should be used as <internalFormat> and the value of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB should be used as <imageSize>.

The saved images will be valid as long as they are used on a device supporting the returned <internalFormat> parameter. If the saved images are used on a device that does not support the compressed internal format, an INVALID_ENUM error would be generated by the call to CompressedTexImage_[123]D because of the unknown format.

Note also that to reliably determine if the GL will compress an image without actually compressing it, an application need only define a proxy texture image and query TEXTURE_COMPRESSED_ARB as above.

(11) Without knowing of the compressed image format, there is no convenient way for the client-side GLX library or tracing tools to ascertain the size of a compressed texture image when sending a TexImage1D, TexImage2D, or TexImage3D packet or interpret pixel storage modes. To complicate matters further, it is possible to create both indirect (that might not understand an image format) and direct rendering contexts (that might understand an image format) on the same renderer. How should this be solved?

RESOLVED: A separate set of CompressedTexImage and CompressedTexSubImage calls has been created that allows libraries to pass compressed images along to the renderer without needing to understand their specific image formats or how to interpret pixel storage modes.

(12) Are the CompressedTexImage[123]DARB entry points really needed?

RESOLVED: Yes. To robustly support images of unknown format, specific compressed entry points are required. While the extension does not support images in a completely unspecified format (early drafts did), having a separate call means that GLX and tools such as GLS (stream encoder) do not need intimate knowledge of every compressed image format. Having separate calls also cleanly solves the problem where pixel storage and pixel transfer operations apply if and only if the source image is uncompressed.

(13) Is variable-ratio compression supported?

RESOLVED: Yes. Fixed-ratio compression is currently the predominant texture compression format, but this spec should not preclude the use of other compression schemes.

(14) Should the <imageSize> parameter be validated on CompressedTexImage calls?

RESOLVED: Yes. Enforcement overhead is generally trivial. Without enforcement, an application could specify incorrect image sizes but notice them only when run on an indirect renderer, causing portability problems. There is also a reliability issue with respect to the GLX environment -- if the compressed image size provided by the user is less than the required image size, the GLX server may run off the end of the image and access invalid memory. A size check may thus be desirable to prevent server crashes (even though that could be considered an "undefined" result).

While enforcing correct <imageSize> parameters is trivial for current compressed internal formats, it might not be reasonable on others (particular variable-ratio compression formats). For such formats, this restriction should be overridden in the spec defining the formats. The <imageSize> check was made mandatory only in the final draft approved at the March 2000 OpenGL ARB meeting.

(15) Should TexImage calls fall back to uncompressed image formats when <internalformat> is a specific compressed format but its use in combination with other parameter values passed is not supported by the renderer?

RESOLVED: Yes. Advantages: Works in exactly the same way as generic formats, meaning no extra code/error checking. Inherent limitations of TexImage on specific formats should be documented in their specs and observed by their users. One simple query can detect fallback cases. Disadvantages: Silent fallback to a format not requested by the user.

(16) Should the texture format invariance requirements disallow scanning of the image data to select a compression method? What about for a base (uncompressed) internal format?

RESOLVED: The primary issue is mipmap consistency. The 1.2.1 spec defines a set of mipmaps as consistent if all are specified using the same internal format. However, it doesn't require that all mipmaps are allocated using the same format -- the renderer is responsible for ensuring mipmap consistency if it selects different formats for

different images. There is no reason to disallow scanning for base internal formats; the renderer is responsible for doing the right thing.

The selection of a specific compressed internal format is different. It must be independent of the the image data because the GL treats the texture image as though it were specified using the specific compressed internal format chosen by the renderer.

(17) Should functionality be provided to enumerate the specific compressed formats supported by the renderer? If so, how and what will it accomplish?

RESOLVED: Yes. A glGet* query is added to return the number of compressed internal formats supported by the renderer and the <internalformat> tokens for each. These tokens can subsequently be used as <internalformat> parameters for normal TexImage calls and the new CompressedTexImage calls.

Providing an internal format enumeration allows applications to weigh the suitability of the various compression methods provided to it by the renderer without needing specific knowledge of the formats. Applications can query the component sizes (see issue 6) to determine the base format and approximate precision. Applications can directly evaluate image compression quality by having the renderer generate compressed texture images (using the returned <internalformat> values) and return them in uncompressed form using GetTexImage. Applications should also be aware that the use of the internal formats returned by this query is subject to the restrictions imposed by the specification defining them. The use of proxy textures allows the application to determine if a specific set of TexImage parameters is supported for a given internal format.

The renderer should enumerate all supported compression formats EXCEPT those that operate fundamentally differently from a normal uncompressed format. For example, the DirectX DXT1 compression format is fundamentally an RGB format, but it has a "transparent" encoding where the red, green, and blue component values are forced to zero, regardless of their original (uncompressed) values. Since such formats may have caveats that must be understood before being used, they should not be enumerated by this query.

This allows for forward compatibility -- an application can exploit compression techniques provided by future renderers.

(18) Should the separate GetCompressedTexImageARB function exist, or is GetTexImage with special <format> and/or <type> parameters sufficient?

RESOLVED: Provide a separate GetCompressedTexImageARB function. The primary rationale is for GLX indirect rendering. The client GetTexImage would require information to determine if an image is uncompressed (and should be decoded using pixel storage state) or compressed (pixel storage ignored). In addition, if the image is compressed, the actual image size would be required, but the only image size that could be inferred from the GLX protocol is padded out to a multiple of four bytes. A separate call is the cleanest solution to both issues.

New Procedures and Functions

```
    void CompressedTexImage3DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 sizei height, sizei depth,
                                 int border, sizei imageSize,
                                 const void *data);
    void CompressedTexImage2DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 sizei height, int border,
                                 sizei imageSize, const void *data);
    void CompressedTexImage1DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 int border, sizei imageSize,
                                 const void *data);
    void CompressedTexSubImage3DARB(enum target, int level,
                                    int xoffset, int yoffset,
                                    int zoffset, sizei width,
                                    sizei height, sizei depth,
                                    enum format, sizei imageSize,
                                    const void *data);
    void CompressedTexSubImage2DARB(enum target, int level,
                                    int xoffset, int yoffset,
                                    sizei width, sizei height,
                                    enum format, sizei imageSize,
                                    const void *data);
    void CompressedTexSubImage1DARB(enum target, int level,
                                    int xoffset, sizei width,
                                    enum format, sizei imageSize,
                                    const void *data);
    void GetCompressedTexImageARB(enum target, int lod,
                                  void *img);
```

New Tokens

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, and CopyTexImage2D:

COMPRESSED_ALPHA_ARB	0x84E9
COMPRESSED_LUMINANCE_ARB	0x84EA
COMPRESSED_LUMINANCE_ALPHA_ARB	0x84EB
COMPRESSED_INTENSITY_ARB	0x84EC
COMPRESSED_RGB_ARB	0x84ED
COMPRESSED_RGBA_ARB	0x84EE

Accepted by the <target> parameter of Hint and the <value> parameter of GetIntegerv, GetBooleanv, GetFloatv, and GetDoublev:

TEXTURE_COMPRESSION_HINT_ARB	0x84EF

Accepted by the <value> parameter of GetTexLevelParameter:

TEXTURE_COMPRESSED_IMAGE_SIZE_ARB	0x86A0
TEXTURE_COMPRESSED_ARB	0x86A1

Accepted by the <value> parameter of GetIntegerv, GetBooleanv, GetFloatv, and GetDoublev:

NUM_COMPRESSED_TEXTURE_FORMATS_ARB	0x86A2
COMPRESSED_TEXTURE_FORMATS_ARB	0x86A3

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify **Section 3.8.1, Texture Image Specification (p.113)**

(p.113, modify 3rd paragraph) <internalformat> may be specified as one of the six base internal format symbolic constants listed in table 3.15, as one of the sized internal format symbolic constants listed in table 3.16, as one of the specific compressed internal format symbolic constants listed in table 3.16.1, or as one of the six generic compressed internal format symbolic constants listed in table 3.16.2.

(p.113, add after 3rd paragraph)

The ARB_texture_compression specification provides no specific compressed internal formats but does provide a mechanism to obtain the enums for such formats provided by other specifications. If the ARB_texture_compression extension is supported, the number of specific compressed internal format symbolic constants supported by the renderer can be obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS_ARB. The set of specific compressed internal format symbolic constants supported by the renderer can be obtained by querying the value of COMPRESSED_TEXTURE_FORMATS_ARB. The only symbolic constants returned by this query are those suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

Generic compressed internal formats are never used directly as the internal formats of texture images. If <internalformat> is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL's choosing with the same base internal format. If no specific compressed format is available, <internalformat> is instead replaced by the corresponding base internal format. If <internalformat> is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), <internalformat> is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

(p.113, modify 4th paragraph) ... If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

(p.113, 5th paragraph) A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any TexImage3D, TexImage2D, or TexImage1D (see below) parameter (except

<target>, but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed image format may not be affected by the <data> parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.8.7.

Add Table 3.16.1: Specific Compressed Internal Formats

Compressed Internal Format	Base Internal Format
none provided here -- defined by dependent extensions	

Add Table 3.16.2: Generic Compressed Internal Formats

Generic Compressed Internal Format	Base Internal Format
COMPRESSED_ALPHA_ARB	ALPHA
COMPRESSED_LUMINANCE_ARB	LUMINANCE
COMPRESSED_LUMINANCE_ALPHA_ARB	LUMINANCE_ALPHA
COMPRESSED_INTENSITY_ARB	INTENSITY
COMPRESSED_RGB_ARB	RGB
COMPRESSED_RGBA_ARB	RGBA

Modify **Section 3.8.2, Alternate Image Specification**

(add to end of TexSubImage discussion, p.123)

Texture images with compressed internal formats may be stored in such a way that it is not possible to edit an image with subimage commands without having to decompress and recompress the texture image being edited. Even if the image were edited in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary edits to texture images with compressed internal formats. Calling TexSubImage3D, CopyTexSubImage3D, TexSubImage2D, CopyTexSubImage2D, TexSubImage1D, or CopyTexSubImage1D will result in an INVALID_OPERATION error if <xoffset>, <yoffset>, or <zoffset> is not equal to -b_s (border). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily edited.

(add new subsection at end of section, p.123)

Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format. The ARB_texture_compression extension defines no such formats, but provides the mechanisms for other extensions that do.

The commands

```
  void CompressedTexImage1DARB(enum target, int level,
                               enum internalformat, sizei width,
                               int border, sizei imageSize,
                               const void *data);
  void CompressedTexImage2DARB(enum target, int level,
                               enum internalformat, sizei width,
                               sizei height, int border,
                               sizei imageSize, const void *data);
  void CompressedTexImage3DARB(enum target, int level,
                               enum internalformat, sizei width,
                               sizei height, sizei depth,
                               int border, sizei imageSize,
                               const void *data);
```

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The <target>, <level>, <internalformat>, <width>, <height>, <depth>, and <border> parameters have the same meaning as in TexImage1D, TexImage2D, and TexImage3D. <data> points to compressed image data stored in the compressed image format corresponding to <internalformat>. Since this extension provides no specific image formats, using any of the six generic compressed internal formats as <internalformat> will result in an INVALID_ENUM error.

For all other compressed internal formats, the compressed image will be decoded according to the specification defining the <internalformat> token. Compressed texture images are treated as an array of <imageSize> ubytes beginning at address <data>. All pixel storage and pixel transfer modes are ignored when decoding a compressed texture image. If the <imageSize> parameter is not consistent with the format, dimensions, and contents of the compressed image, an INVALID_VALUE error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Specific compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might be supported only for 2D textures or may not allow non-zero <border> values. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB will not result in an INVALID_OPERATION error if the following restrictions are satisfied:

* <data> points to a compressed texture image returned by GetCompressedTexImageARB (Section 6.1.4).

* <target>, <level>, and <internalformat> match the <target>, <level> and <format> parameters provided to the GetCompressedTexImageARB call returning <data>.

```
    * <width>, <height>, <depth>, <border>, <internalformat>, and
      <imageSize> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT,
      TEXTURE_DEPTH, TEXTURE_BORDER, TEXTURE_INTERNAL_FORMAT, and
      TEXTURE_COMPRESSED_IMAGE_SIZE_ARB for image level <level> in effect at
      the time of the GetCompressedTexImageARB call returning <data>.

This guarantee applies not just to images returned by
GetCompressedTexImageARB, but also to any other properly encoded
compressed texture image of the same size and format.


The commands

  void CompressedTexSubImage1DARB(enum target, int level,
                                  int xoffset, sizei width,
                                  enum format, sizei imageSize,
                                  const void *data);
  void CompressedTexSubImage2DARB(enum target, int level,
                                  int xoffset, int yoffset,
                                  sizei width, sizei height,
                                  enum format, sizei imageSize,
                                  const void *data);
  void CompressedTexSubImage3DARB(enum target, int level,
                                  int xoffset, int yoffset,
                                  int zoffset, sizei width,
                                  sizei height, sizei depth,
                                  enum format, sizei imageSize,
                                  const void *data);


respecify only a rectangular region of an existing texture array, with
incoming data stored in a known compressed image format.  The <target>,
<level>, <xoffset>, <yoffset>, <zoffset>, <width>, <height>, and <depth>
parameters have the same meaning as in TexSubImage1D, TexSubImage2D, and
TexSubImage3D.  <data> points to compressed image data stored in the
compressed image format corresponding to <format>.  Since this extension
provides no specific image formats, using any of these six generic
compressed internal formats as <format> will result in an INVALID_ENUM
error.

The image pointed to by <data> and the <imageSize> parameter are
interpreted as though they were provided to CompressedTexImage1DARB,
CompressedTexImage2DARB, and CompressedTexImage3DARB.  These commands do
not provide for image format conversion, so an INVALID_OPERATION error
results if <format> does not match the internal format of the texture
image being modified.  If the <imageSize> parameter is not consistent with
the format, dimensions, and contents of the compressed image (too little
or too much data), an INVALID_VALUE error results.

As with CompressedTexImage calls, compressed internal formats may have
additional restrictions on the use of the compressed image specification
calls or parameters.  Any such restrictions will be documented in the
specification defining the compressed internal format; violating these
restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be
invariant, meaning that if the GL accepts and stores a texture image in
```

compressed form, providing the same image to CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, CompressedTexSubImage3DARB will not result in an INVALID_OPERATION error if the following restrictions are satisfied:

* <data> points to a compressed texture image returned by GetCompressedTexImageARB (Section 6.1.4).

* <target>, <level>, and <format> match the <target>, <level> and <format> parameters provided to the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <format>, and <imageSize> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE_ARB for image level <level> in effect at the time of the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <format> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_INTERNAL_FORMAT currently in effect for image level <level>.

* <xoffset>, <yoffset>, and <zoffset> are all "-<b>", where <b> is the value of TEXTURE_BORDER currently in effect for image level <level>.

This guarantee applies not just to images returned by GetCompressedTexImageARB, but also to any other properly encoded compressed texture image of the same size.

Calling CompressedTexSubImage3D, CompressedTexSubImage2D, or CompressedTexSubImage1D will result in an INVALID_OPERATION error if <xoffset>, <yoffset>, or <zoffset> is not equal to -b_s (border), or if <width>, <height>, and <depth> do not match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, or TEXTURE_DEPTH, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily edited.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

Modify **Section 5.6, Hints** (p.180)

(p.180, modify first paragraph)

...; FOG_HINT, indicating whether fog calculations are done per pixel or per vertex; and TEXTURE_COMPRESSION_HINT_ARB, indicating the desired quality and performance of compressing texture images.

For the texture compression hint, a <hint> of FASTEST indicates that texture images should be compressed as quickly as possible, while NICEST indicates that the texture images be compressed with as little image degradation as possible. FASTEST should be used for one-time texture compression, and NICEST should be used if the compression results are to

be retrieved by GetCompressedTexImageARB (Section 6.1.4) for reuse.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

Modify **Section 6.1.3, Enumerated Queries** (p.183)

(p.183, modify next-to-last paragraph)

For texture images with uncompressed internal formats, queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. For texture images with a compressed internal format, the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation's compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations. ...

(p.183, add to end of next-to-last paragraph)

TEXTURE_COMPRESSED_IMAGE_SIZE_ARB returns the size (in ubytes) of the compressed texture image that would be returned by GetCompressedTexImageARB (Section 6.1.4). Querying TEXTURE_COMPRESSED_IMAGE_SIZE_ARB is not allowed on texture images with an uncompressed internal format or on proxy targets and will result in an INVALID_OPERATION error if attempted.

Modify **Section 6.1.4, Texture Queries** (p.184)

(add immediately after the GetTexImage section and before the IsTexture section)

The command

```
void GetCompressedTexImageARB(enum target, int lod,
                              void *img);
```

is used to obtain texture images stored in compressed form. The parameters <target>, <lod>, and <img> are interpreted in the same manner as in GetTexImage. When called, GetCompressedTexImageARB writes TEXTURE_COMPRESSED_IMAGE_SIZE_ARB ubytes of compressed image data to the memory pointed to by <img>. The compressed image data is formatted according to the specification defining INTERNAL_FORMAT. All pixel storage and pixel transfer modes are ignored when returning a compressed texture image.

Calling GetCompressedTexImageARB with an <lod> value less than zero or greater than the maximum allowable causes an INVALID_VALUE error. Calling GetCompressedTexImageARB with a texture image stored with an uncompressed internal format causes an INVALID_OPERATION error.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

(Add after GetTexImage to Section 2.2.2 of the GLX 1.3 encoding spec, p.74)

GetCompressedTexImageARB

1	CARD8	opcode (X assigned)
1	160	GLX opcode
2	4	request length
4	GLX_CONTEXT_TAG	context tag
4	ENUM	target
4	INT32	level

-->

1	1	Reply
1	1	unused
2	CARD16	sequence number
4	n	reply length
8		unused
4	INT32	compressed image size (in bytes) -- should be between 4n-3 and 4n
12		unused
4*n	LISTofBYTE	teximage

Note that n may be zero, indicating that a GL error occurred.

Since pixel storage modes do not apply to compressed texture images, teximage is simply an array of bytes. The client library will ignore pixel storage modes and should copy only <compressed image size> bytes, regardless of the value of <reply length>.

(Add to end of Section 2.3 of the GLX 1.3 encoding spec, p.147)

CompressedTexImage1DARB

2	32+n+p	rendering command length
2	214	rendering command opcode
4	ENUM	target
4	INT32	level
4	ENUM	internalformat
4	INT32	width
4		unused
4	INT32	border
n	LISTofBYTE	image
4	INT32	imageSize
p		unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields are expanded to 4 bytes each.

4	36+n+p	rendering command length
4	214	rendering command opcode

CompressedTexImage2DARB

2	32+n+p	rendering command length
2	215	rendering command opcode
4	ENUM	target
4	INT32	level
4	ENUM	internalformat
4	INT32	width
4	INT32	height
4	INT32	border
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields are expanded to 4 bytes each.

4	36+n+p	rendering command length
4	215	rendering command opcode

```
CompressedTexImage3DARB

    2       36+n+p          rendering command length
    2       216             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       INT32           internalformat
    4       INT32           width
    4       INT32           height
    4       INT32           depth
    4       INT32           border
    4       INT32           imageSize
    n       LISTofBYTE      image
    p                       unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command
opcode and command length fields are expanded to 4 bytes each.

    4       36+n+p          rendering command length
    4       216             rendering command opcode

CompressedTexSubImage1DARB

    2       36+n+p          rendering command length
    2       217             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       INT32           xoffset
    4                       unused
    4       INT32           width
    4                       unused
    4       ENUM            format
    4       INT32           imageSize
    n       LISTofBYTE      image
    p                       unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command
opcode and command length fields are expanded to 4 bytes each.

    4       40+n+p          rendering command length
    4       217             rendering command opcode
```

CompressedTexSubImage2DARB

2	36+n+p	rendering command length
2	218	rendering command opcode
4	ENUM	target
4	INT32	level
4	INT32	xoffset
4	INT32	yoffset
4	INT32	width
4	INT32	height
4	ENUM	format
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields are expanded to 4 bytes each.

4	40+n+p	rendering command length
4	218	rendering command opcode

CompressedTexSubImage3DARB

2	44+n+p	rendering command length
2	219	rendering command opcode
4	ENUM	target
4	INT32	level
4	INT32	xoffset
4	INT32	yoffset
4	INT32	zoffset
4	INT32	width
4	INT32	height
4	INT32	depth
4	ENUM	format
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields are expanded to 4 bytes each.

4	48+n+p	rendering command length
4	219	rendering command opcode

Errors

Errors for compressed TexImage and TexSubImage calls specific to compression:

INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D, TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or CopyTexSubImage3D if the internal format of the texture image is compressed and <xoffset>, <yoffset>, or <zoffset> does not equal -b, where b is value of TEXTURE_BORDER.

INVALID_VALUE is generated by CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if the entire texture image is not being edited: if <xoffset>, <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is less than w+b, <yoffset> + <height> is less than h+b, or <zoffset> + <depth> is less than d+b, where b is the value of TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.

INVALID_ENUM is generated by CompressedTexImage1DARB, CompressedTexImage2DARB, or CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, if <internalformat> is any of the six generic compressed internal formats (e.g., COMPRESSED_RGBA_ARB)

INVALID_OPERATION is generated by CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, if any parameter combinations are not supported by the specific compressed internal format. Such invalid combinations are documented in the specification defining the internal format.

INVALID_VALUE is generated by CompressedTexImage1DARB, CompressedTexImage2DARB, or CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, if <imageSize> is not consistent with the format, dimensions, and contents of the specified image. The appropriate value for the <imageSize> parameter is documented in the specification defining the compressed internal format.

Undefined results (including abnormal program termination) are generated by CompressedTexImage1DARB, CompressedTexImage2DARB, or CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, is not encoded in a manner consistent with the specification defining the internal format.

INVALID_OPERATION is generated by CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if <format> does not match the internal format of the texture image being modified.

INVALID_OPERATION is generated by GetTexLevelParameter[if]v if <target> is PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, or PROXY_TEXTURE_3D and <value> is TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.

INVALID_OPERATION is generated by GetTexLevelParameter[if]v if the internal format of the queried texture image is not compressed and <value> is TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.

INVALID_OPERATION is generated by GetCompressedTexImageARB if the internal format of the queried texture image is not compressed.

Errors for compressed TexImage and TexSubImage calls not specific to compression:

INVALID_ENUM is generated by CompressedTexImage3DARB or
CompressedTexSubImage3DARB if <target> is not TEXTURE_3D.

INVALID_ENUM is generated by CompressedTexImage2DARB or
CompressedTexSubImage2DARB if <target> is not TEXTURE_2D,
TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.

INVALID_ENUM is generated by CompressedTexImage1DARB or
CompressedTexSubImage1DARB if <target> is not TEXTURE_1D.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, CompressedTexImage3DARB,
CompressedTexSubImage1DARB, CompressedTexSubImage1DARB, or
CompressedTexSubImage3DARB if <level> is negative.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, CompressedTexImage3DARB,
CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or
CompressedTexSubImage3DARB, if <width>, <height>, or <depth> is negative.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, or CompressedTexImage3DARB if <width>, <height>,
or <depth> can not be represented as 2^k+2 for some integer value k.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, or CompressedTexImage3DARB if <border> is not
zero or one.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, CompressedTexImage3DARB,
CompressedTexSubImage1DARB, CompressedTexSubImage1DARB, or
CompressedTexSubImage3DARB if the call is made between a call to Begin and
the corresponding call to End.

INVALID_VALUE is generated by CompressedTexSubImage1DARB,
CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if <xoffset>,
<yoffset>, or <zoffset> is less than -b, <xoffset> + <width> is greater
than w+b, <yoffset> + <height> is greater than h+b, or <zoffset> + <depth>
is greater than d+b, where b is the value of TEXTURE_BORDER, w is the
value of TEXTURE_WIDTH, h is the value of TEXTURE_HEIGHT, and d is the
value of TEXTURE_DEPTH.

INVALID_VALUE is generated by GetCompressedTexImageARB if <lod> is
negative or greater than the maximum allowable level.

New State

(table 6.12, p.202)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSED_IMAGE_SIZE_ARB	n x Z+	GetTexLevel-Parameter	0	size (in ubytes) of xD compressed texture image i.	3.8	-
TEXTURE_COMPRESSED_ARB	n x B	GetTexLevel-Parameter	FALSE	True if xD image i has a compressed internal format	3.8	-

(table 6.23, p.213)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSION_HINT_ARB	Z_3	GetIntegerv	DONT_CARE	Texture compression quality hint	5.6	hint

(table 6.25, p. 215)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
NUM_COMPRESSED_TEXTURE_FORMATS_ARB	Z	GetIntegerv	0	Number of enumerated compressed texture formats	3.8	-
COMPRESSED_TEXTURE_FORMATS_ARB	0* x Z	GetIntegerv	-	Enumerated compressed texture formats	3.8	-

Revision History

1.03, 05/23/00 prbrown1: Removed stray "None." paragraph in modifications to Chapter 5.

1.02, 05/08/00 prbrown1: Fixed prototype of GetCompressedTexImageARB (no "const" qualifiers) in "New Procedures and Functions" section. Changed <internalformat> parameter of CompressedTexImage functions to be an "enum" instead of an "int". "int" was carried over only on TexImage calls as a 1.0 legacy -- the newer CopyTexImage call takes an "enum".

1.01, 04/11/00 prbrown1: Minor bug fixes to the first published version. Fixed prototypes to match extension spec standards (no "GL" type prefixes). Fixed a couple erroneous function names. Added "const" qualifier to prototypes involving image data not modified by the GL. Added text to indicate that compressed formats apply to texture maps supported by GL_ARB_texture_cube_map.

1.0, 03/24/00 prbrown1: Applied changes approved as part of the extension at the March 2000 ARB meeting, as follows:

* CompressedTexSubImage: Only allowed if the

```
                              entire image is replaced.  Document that this
                              restriction can be relaxed for specific
                              compression extensions.
                            * Renamed TEXTURE_IMAGE_SIZE_ARB to
                              TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.
                            * Querying image size on uncompressed images is
                              now an INVALID_OPERATION error.
                            * INVALID_VALUE error is generated if <imageSize>
                              is inconsistent with the image data.  This
                              restriction may be overridden by specific
                              extensions only if requiring an image size
                              check is unreasonable.
                            * Added documentaion of undefined behavior for
                              CompressedTexImage/SubImage if the image data
                              is encoded in a manner inconsistent with the
                              spec defining the compressed image format.
                            * Fixed issue (16).  Text was truncated.
                            * Modified invariance section.  <data> can not
                              affect the choice of compressed internal
                              format, but can theoretically affect regular
                              component resolution.
                            * Add new function GetCompressedTexImage to deal
                              with subtle GLX issues.
                            * GLX protocol for CompressedTexImage/SubImage
                              and GetCompressedTexImage holds both a padded
                              image size (for GLX data transfer) and actual
                              image size (for packing in user buffers).

                            Minor wording clean-ups.

                            Added enum and GLX opcode values allocated from
                            OpenGL Extensions and GLX registries.

  0.81, 03/07/00 prbrown1: Fixed error documentation for TexSubImage calls
                            of arbitrary alignment (did not document that the
                            internal format had to be compressed).  Removed
                            references to CopyTexImage3D, which doesn't
                            actually exist.

                            Per Kurt Akeley suggestions: (1) Renamed
                            TexImageCompressed to CompressedTexImage to
                            conform with naming conventions, (2) clarified
                            that the main feature distinguishing
                            CompressedTex[Sub]Image calls from normal
                            Tex[Sub]Image calls is compressed input data, (3)
                            added query to explicitly determine whether the
                            internal format of a texture is compressed.

  0.8,  02/23/00 prbrown1: Marked previously unresolved issues as resolved
                            per the ARB working group.  Added docs for errors
                            not specific to compression for the new
                            CompressedTexImage and CompressedTexSubImage
                            calls.  Added queries to enumerate specific
                            compressed texture formats.
  0.76, 02/16/00 prbrown1: Removed "gl" and "GL_" prefixes.
  0.75, 02/07/00 prbrown1: Incorporated feedback from 12/99 ARB meeting
                            and a number of other revisions.
```

```
0.7,  12/03/99 prbrown1: Incorporated comments from public review of 0.2
                         document.
0.2,  10/28/99 prbrown1: Renamed to ARB_texture_compression.  Significant
                         functional changes.
0.11, 10/21/99 prbrown1: Edits suggested by 3dfx.
0.1,  10/19/99 prbrown1: Initial revision.
```

Name

ARB_texture_cube_map

Name Strings

GL_ARB_texture_cube_map

Notice

Copyright OpenGL Architectural Review Board, 1999.

Status

Complete. Approved by ARB on 12/8/1999

Version

Last Modified Date: December 14, 1999

Number

ARB Extension #7

Dependencies

None.

Written based on the wording of the OpenGL 1.2.1 specification but not dependent on it.

Overview

This extension provides a new texture generation scheme for cube map textures. Instead of the current texture providing a 1D, 2D, or 3D lookup into a 1D, 2D, or 3D texture image, the texture is a set of six 2D images representing the faces of a cube. The (s,t,r) texture coordinates are treated as a direction vector emanating from the center of a cube. At texture generation time, the interpolated per-fragment (s,t,r) selects one cube face 2D image based on the largest magnitude coordinate (the major axis). A new 2D (s,t) is calculated by dividing the two other coordinates (the minor axes values) by the major axis value. Then the new (s,t) is used to lookup into the selected 2D texture image face of the cube map.

Unlike a standard 1D, 2D, or 3D texture that have just one target, a cube map texture has six targets, one for each of its six 2D texture image cube faces. All these targets must be consistent, complete, and have equal width and height (ie, square dimensions).

This extension also provides two new texture coordinate generation modes for use in conjunction with cube map texturing. The reflection map mode generates texture coordinates (s,t,r) matching the vertex's eye-space reflection vector. The reflection map mode is useful for environment mapping without the singularity inherent in sphere mapping. The normal map mode generates texture coordinates (s,t,r) matching the vertex's transformed eye-space

normal. The normal map mode is useful for sophisticated cube map texturing-based diffuse lighting models.

The intent of the new texgen functionality is that an application using cube map texturing can use the new texgen modes to automatically generate the reflection or normal vectors used to look up into the cube map texture.

An application note: When using cube mapping with dynamic cube maps (meaning the cube map texture is re-rendered every frame), by keeping the cube map's orientation pointing at the eye position, the texgen-computed reflection or normal vector texture coordinates can be always properly oriented for the cube map. However if the cube map is static (meaning that when view changes, the cube map texture is not updated), the texture matrix must be used to rotate the texgen-computed reflection or normal vector texture coordinates to match the orientation of the cube map. The rotation can be computed based on two vectors: 1) the direction vector from the cube map center to the eye position (both in world coordinates), and 2) the cube map orientation in world coordinates. The axis of rotation is the cross product of these two vectors; the angle of rotation is the arcsin of the dot product of these two vectors.

Issues

Should we place the normal/reflection vector in the (s,t,r) texture coordinates or (s,t,q) coordinates?

RESOLUTION: (s,t,r). Even if hardware uses "q" for the third component, the API should claim to support generation of (s,t,r) and let the texture matrix (through a concatenation with the user-supplied texture matrix) move "r" into "q".

Should the texture coordinate generation functionality for cube mapping be specified as a distinct extension from the actual cube map texturing functionality?

RESOLUTION: NO. Real applications and real implementations of cube mapping will tie the texgen and texture generation functionality together. Applications won't have to query two separate extensions then.

While applications will almost always want to use the texgen functionality for automatically generating the reflection or normal vector as texture coordinates (s,t,r), this extension does permit an application to manually supply the reflection or normal vector through glTexCoord3f explicitly.

Note that the NV_texgen_reflection extension does "unbundle" the texgen functionality from cube maps.

Should you be able to have some texture coordinates computing REFLECTION_MAP_ARB and others not? Same question with NORMAL_MAP_ARB.

RESOLUTION: YES. This is the way that SPHERE_MAP works. It is not clear that this would ever be useful though.

Should something special be said about the handling of the q texture coordinate for this spec?

RESOLUTION: NO. But the following paragraph is useful for implementors concerned about the handling of q.

The REFLECTION_MAP_ARB and NORMAL_MAP_ARB modes are intended to supply reflection and normal vectors for cube map texturing hardware. When these modes are used for cube map texturing, the generated texture coordinates can be thought of as an reflection vector. The value of the q texture coordinate then simply scales the vector but does not change its direction. Because only the vector direction (not the vector magnitude) matters for cube map texturing, implementations are free to leave q undefined when any of the s, t, or r texture coordinates are generated using REFLECTION_MAP_ARB or NORMAL_MAP_ARB.

How should the cube faces be labeled?

RESOLUTION: Match the render man specification's names of "px" (positive X), "nx" (negative x), "py", "ny", "pz", and "nz". There does not actually need to be an "ordering for the faces" (Direct3D 7.0 does number their cube map faces.) For this extension, the symbolic target names (TEXTURE_CUBE_MAP_POSITIVE_X_ARB, etc) is sufficient without requiring any specific ordering.

What coordinate system convention should be used? LHS or RHS?

RESOLUTION: The coordinate system is left-handed if you think of yourself within the cube. The coordinate system is right-handed if you think of yourself outside the cube.

This matches the convention of the RenderMan interface. If you look at Figure 12.8 (page 265) in "The RenderMan Companion", think of the cube being folded up with the observer inside the cube. Then the coordinate system convention is left-handed.

The spec just linearly interpolates the reflection vectors computed per-vertex across polygons. Is there a problem interpolating reflection vectors in this way?

Probably. The better approach would be to interpolate the eye vector and normal vector over the polygon and perform the reflection vector computation on a per-fragment basis. Not doing so is likely to lead to artifacts because angular changes in the normal vector result in twice as large a change in the reflection vector as normal vector changes. The effect is likely to be reflections that become glancing reflections too fast over the surface of the polygon.

Note that this is an issue for REFLECTION_MAP_ARB, but not NORMAL_MAP_ARB.

What happens if an (s,t,q) is passed to cube map generation that is close to (0,0,0), ie. a degenerate direction vector?

RESOLUTION: Leave undefined what happens in this case (but

may not lead to GL interruption or termination).

Note that a vector close to (0,0,0) may be generated as a result of the per-fragment interpolation of (s,t,r) between vertices.

Do we need a distinct proxy texture mechanism for cube map textures?

RESOLUTION: YES. Cube map textures take up six times the memory as a conventional 2D image texture so proxy 2D texture determinations won't be of value for a cube map texture. Cube maps need their own proxy target.

Should we require the 2D texture image width and height to be identical (ie, square only)?

RESOLUTION: YES. This limitation is quite a reasonable limitation and DirectX 7 has the same limitation.

This restriction is enforced by generating an INVALID_VALUE when calling TexImage2D or CopyTexImage2D with a non-equal width and height.

Some consideration was given to enforcing the "squarness" constraint as a texture consistency constraint. This is confusing however since the squareness is known up-front at texture image specification time so it seems confusing to silently report the usage error as a texture consistency issue.

Texture consistency still says that all the level 0 textures of all six faces must have the same square size.

If some combination of 1D, 2D, 3D, and cube map texturing is enabled, which really operates?

RESOLUTION: Cube map texturing. In OpenGL 1.2, 3D takes priority over 2D takes priority over 1D. Cube mapping should take priority over all conventional n-dimensional texturing schemes.

Does anything need to be said about combining cube mapping with multitexture?

RESOLUTION: NO. Cube mapping should be available on all texture units. The hardware should fully orthogonal in its handling of cube map textures.

Does it make sense to support borders for cube map textures.

Actually, it does. It would be nice if the texture border pixels match the appropriate texels from the edges of the other cube map faces that they junction with. For this reason, we'll leave the texture border capability implicitly supported.

How does mipmap level-of-detail selection work for cube map textures?

The existing spec's language about LOD selection is fine.

Should the implementation dependent value for the maximum texture size for a cube map be the same as MAX_TEXTURE_SIZE?

RESOLUTION: NO. OpenGL 1.2 has a different MAX_3D_TEXTURE_SIZE for 3D textures, and cube maps should take six times more space than a 2D texture map of the same width & height. The implementation dependent MAX_CUBE_MAP_TEXTURE_SIZE_ARB constant should be used for cube maps then.

Note that the proxy cube map texture provides a better way to find out the maximum cube map texture size supported since the proxy mechanism can take into account the internal format, etc.

In section 3.8.10 when the "largest magnitude coordinate direction" is choosen, what happens if two or more of the coordinates (rx,ry,rz) have the identical magnitude?

RESOLUTION: Implementations can define their own rule to choose the largest magnitude coordinate direction whne two or more of the coordinates have the identical magnitude. The only restriction is that the rule must be deterministic and depend only on (rx,ry,rz).

In practice, (s,t,r) is interpolated across polygons so the cases where |s|==|t|, etc. are pretty arbitary (the equality depends on interpolation precision). This extension could mandate a particular rule, but that seems heavy-handed and there is no good reason that multiple vendors should be forced to implement the same rule.

Should there be limits on the supported border modes for cube maps?

RESOLUTION: NO. The specificiation is written so that cube map texturing proceeds just like conventional 2D texture mapping once the face determination is made.

Therefore, all OpenGL texture wrap modes should be supported though some modes are clearly inappropriate for cube maps. The WRAP mode is almost certainly incorrect for cube maps. Likewise, the CLAMP mode without a texture border is almost certainly incorrect for cube maps. CLAMP when a texture border is present and CLAMP_TO_EDGE are both reasonably suited for cube maps. Ideally, CLAMP with a texture border works best if the cube map edges can be replicated in the approriate texture borders of adjacent cube map faces. In practice, CLAMP_TO_EDGE works reasonably well in most circumstances.

Perhaps another extension could support a special cube map wrap mode that automatically wraps individual texel fetches to the appropriate adjacent cube map face. The benefit from such a mode is small and the implementation complexity is involved so this wrap mode should not be required for a basic cube map texture extension.

How is mipmap LOD selection handled for cube map textures?

RESOLUTION: The specification is written so that cube map texturing proceeds just like conventional 2D texture mapping once the face determination is made.

Thereforce, the partial differentials in Section 3.8.5 (page 126) should be evaluated for the u and v parameters based on the post-face determination s and t.

In Section 2.10.3 "Normal Transformation", there are several versions of the eye-space normal vector to choose from. Which one should the NORMAL_MAP_ARB texgen mode use?

RESOLUTION: nf. The nf vector is the final normal, post-rescale normal and post-normalize. In practice, the rescale normal and normalize operations do not change the direction of the vector so the choice of which version of transformed normal is used is not important for cube maps.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

NORMAL_MAP_ARB	0x8511
REFLECTION_MAP_ARB	0x8512

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain NORMAL_MAP_ARB or REFLECTION_MAP_ARB.

Accepted by the <cap> parameter of Enable, Disable, IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindTexture, GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameteri, TexParameterfv, and TexParameteriv:

TEXTURE_CUBE_MAP_ARB	0x8513

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

TEXTURE_BINDING_CUBE_MAP_ARB	0x8514

Accepted by the <target> parameter of GetTexImage,
GetTexLevelParameteriv, GetTexLevelParameterfv, TexImage2D,
CopyTexImage2D, TexSubImage2D, and CopySubTexImage2D:

TEXTURE_CUBE_MAP_POSITIVE_X_ARB	0x8515
TEXTURE_CUBE_MAP_NEGATIVE_X_ARB	0x8516
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB	0x8517
TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB	0x8518
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB	0x8519
TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB	0x851A

Accepted by the <target> parameter of GetTexLevelParameteriv,
GetTexLevelParameterfv, GetTexParameteriv, and TexImage2D:

PROXY_TEXTURE_CUBE_MAP_ARB	0x851B

Accepted by the <pname> parameter of GetBooleanv, GetDoublev,
GetIntegerv, and GetFloatv:

MAX_CUBE_MAP_TEXTURE_SIZE_ARB	0x851C

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph (page 37) to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to
or <param> is an integer that is one of the symbolic constants
OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP_ARB, or
NORMAL_MAP_ARB."

Add these paragraphs after the 4th paragraph (page 38):

"If TEXTURE_GEN_MODE indicates REFLECTION_MAP_ARB, compute the
reflection vector r as described for the SPHERE_MAP mode. Then the
value assigned to an s coordinate (the first TexGen argument value
is S) is s = rx; the value assigned to a t coordinate is t = ry;
and the value assigned to a r coordinate is r = rz. Calling TexGen
with a <coord> of Q when <pname> indicates REFLECTION_MAP_ARB
generates the error INVALID_ENUM.

If TEXTURE_GEN_MODE indicates NORMAL_MAP_ARB, compute the normal
vector nf as described in section 2.10.3. Then the value assigned
to an s coordinate (the first TexGen argument value is S) is s =
nfx; the value assigned to a t coordinate is t = nfy; and the
value assigned to a r coordinate is r = nfz. (The values nfx, nfy,
and nfz are the components of nf.) Calling TexGen with a <coord>
of Q when <pname> indicates NORMAL_MAP_ARB generates the error
INVALID_ENUM.

The last paragraph's first sentence (page 38) should be changed to:

"The state required for texture coordinate generation comprises a
five-valued integer for each coordinate indicating coordinate
generation mode, ..."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- **Section 3.6.5 "Pixel Transfer Operations" under "Convolution"**

Change this paragraph (page 103) to say:

... "If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to DrawPixels, CopyPixels, ReadPixels, TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, and CopyTexSubImage3D, and returned by GetTexImage with one of the targets TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB."

-- **Section 3.8.1 "Texture Image Specification"**

Change the second and third to last sentences on page 116 to:

"<target> must be one of TEXTURE_2D for a 2D texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB for a cube map texture. Additionally, <target> can be either PROXY_TEXTURE_2D for a 2D proxy texture or PROXY_TEXTURE_CUBE_MAP_ARB for a cube map proxy texture as discussed in section 3.8.7."

Add the following paragraphs after the first paragraph on page 117:

"A 2D texture consists of a single 2D texture image. A cube map texture is a set of six 2D texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_*_ARB targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map 2D image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X_ARB are used when specifying, updating, or querying one of a cube map's six 2D image, but when enabling cube map texturing or binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular 2D image), the TEXTURE_CUBE_MAP_ARB target is specified.

When the target parameter to TexImage2D is one of the six cube map 2D image targets, the error INVALID_VALUE is generated if the width and height parameters are not equal.

If cube map texturing is enabled at the time a primitive is rasterized and if the set of six targets are not "cube complete", then it is as if texture mapping were disabled. The targets of a cube map texture are "cube complete" if the array 0 of all six targets have identical, positive, and square dimensions, the array 0 of all six targets were specified with the same internalformat, and the array 0 of all six targets have the same border width."

After the 14th paragraph (page 116) add:

"In a similiar fashion, the maximum allowable width and height (they must be the same) of a cube map texture must be at least

2^(k-lod)+2bt for image arrays level 0 through k, where k is the
log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE_ARB."

-- **Section 3.8.2 "Alternate Texture Image Specification Commands"**

Update the second paragraph (page 120) to say:

... "Currently, <target> must be
TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB,
TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB,
TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB,
or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB." ...

Add after the second paragraph (page 120), the following:

"When the target parameter to CopyTexImage2D is one of the six cube
map 2D image targets, the error INVALID_VALUE is generated if the
width and height parameters are not equal."

Update the fourth paragraph (page 121) to say:

... "Currently the target arguments of TexSubImage1D and
CopyTexSubImage1D must be TEXTURE_1D, the <target> arguments of
TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D,
TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB,
and the <target> arguments of TexSubImage3D and CopyTexSubImage3D
must be TEXTURE_3D." ...

-- **Section 3.8.3 "Texture Parameters"**

Change paragraph one (page 124) to say:

... "<target> is the target, either TEXTURE_1D,
TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ...

Add a final paragraph saying:

"Texture parameters for a cube map texture apply to cube map
as a whole; the six distinct 2D texture images use the
texture parameters of the cube map itself.

-- **Section 3.8.5 "Texture Minification" under "Mipmapping"**

Change the first full paragraph on page 130 to:

... "If texturing is enabled for one-, two-, or three-dimensional
texturing but not cube map texturing (and TEXTURE_MIN_FILTER
is one that requires a mipmap) at the time a primitive is
rasterized and if the set of arrays TEXTURE_BASE_LEVEL through q =
min{p,TEXTURE_MAX_LEVEL} is incomplete, based on the dimensions of
array 0, then it is as if texture mapping were disabled."

Follow the first full paragraph on page 130 with:

"If cube map texturing is enabled and TEXTURE_MIN_FILTER is one that requires mipmap levels at the time a primitive is rasterized and if the set of six targets are not "mipmap cube complete", then it is as if texture mapping were disabled. The targets of a cube map texture are "mipmap cube complete" if the six cube map targets are "cube complete" and the set of arrays TEXTURE_BASE_LEVEL through q are not incomplete (as described above)."

-- **Section 3.8.7 "Texture State and Proxy State"**

Change the first sentence of the first paragraph (page 131) to say:

"The state necessary for texture can be divided into two categories. First, there are the nine sets of mipmap arrays (one each for the one-, two-, and three-dimensional texture targets and six for the cube map texture targets) and their number." ...

Change the second paragraph (page 132) to say:

"In addition to the one-, two-, three-dimensional, and the six cube map sets of image arrays, the partially instantiated one-, two-, and three-dimensional and one cube map sets of proxy image arrays are maintained." ...

After the third paragraph (page 132) add:

"The cube map proxy arrays are operated on in the same manner when TexImage2D is executed with the <target> field specified as PROXY_TEXTURE_CUBE_MAP_ARB with the addition that determining that a given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP_ARB indicates that all six of the cube map 2D images are supported. Likewise, if the specified PROXY_TEXTURE_CUBE_MAP_ARB is not supported, none of the six cube map 2D images are supported."

Change the second sentence of the fourth paragraph (page 132) to:

"Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP_ARB cannot be used as textures, and their images must never be queried using GetTexImage." ...

-- **Section 3.8.8 "Texture Objects"**

Change the first sentence of the first paragraph (page 132) to say:

"In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB, named one-, two-, and three-dimensional texture objects and cube map texture objects can be created and operated on." ...

Change the second paragraph (page 132) to say:

"A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ... "If the new texture object is bound to TEXTURE_1D, TEXTURE_2D,

TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, it remains a one-, two-, three-dimensional, or cube map texture until it is deleted."

Change the third paragraph (page 133) to say:

"BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB."

Change paragraph five (page 133) to say:

"In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP have one-dimensional, two-dimensional, three-dimensional, and cube map state vectors associated with them respectively." ... "The initial, one-dimensional, two-dimensional, three-dimensional, and cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTUER_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB respectively while 0 is bound to the corresponding targets."

Change paragraph six (page 133) to say:

... "If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB is deleted, it is as though BindTexture has been executed with the same <target> and <texture> zero." ...

-- Section 3.8.10 "Texture Application"

Replace the beginning sentences of the first paragraph (page 138) with:

"Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB to enable the one-dimensional, two-dimensional, three-dimensional, or cube map texturing respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-, two-, or one-dimensional textures is enabled, then cube map texturing is used. If texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality.

However, when cube map texturing is enabled, the rules are more complicated. For cube map texturing, the (s,t,r) texture coordinates are treated as a direction vector (rx,ry,rz) emanating from the center of a cube. (The q coordinate can be ignored since it merely scales the vector without affecting the direction.) At texture application time, the interpolated per-fragment (s,t,r) selects one of the cube map face's 2D image based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on (rx,ry,rz). The target column

in the table below explains how the major axis direction maps to the 2D image of a particular cube map target.

major axis direction	target	sc	tc	ma
+rx	TEXTURE_CUBE_MAP_POSITIVE_X_ARB	-rz	-ry	rx
-rx	TEXTURE_CUBE_MAP_NEGATIVE_X_ARB	+rz	-ry	rx
+ry	TEXTURE_CUBE_MAP_POSITIVE_Y_ARB	+rx	+rz	ry
-ry	TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB	+rx	-rz	ry
+rz	TEXTURE_CUBE_MAP_POSITIVE_Z_ARB	+rx	-ry	rz
-rz	TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB	-rx	-ry	rz

Using the sc, tc, and ma determined by the major axis direction as specified in the table above, an updated (s,t) is calculated as follows

$$s = (sc/|ma| + 1) / 2$$
$$t = (tc/|ma| + 1) / 2$$

This new (s,t) is used to find a texture value in the determined face's 2D texture image using the rules given in sections 3.8.5 and 3.8.6." ...

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

-- Section 5.4 "Display Lists"

In the first paragraph (page 179), add PROXY_TEXTURE_CUBE_MAP_ARB to the list of PROXY_* tokens.

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the fourth paragraph (page 183) to say:

"The GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, indicating the currently bound one-dimensional, two-dimensional, three-dimensional, or cube map texture object. For GetTexLevelParameter, <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP_ARB, indicating the one-dimensional texture object, two-dimensional texture object, three-dimensional texture object, or one of the six distinct 2D images making up the cube map texture object or one-dimensional, two-dimensional, three-dimensional, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP_ARB is not a valid <target> parameter for

GetTexLevelParameter because it does not specify a particular cube map face."

-- **Section 6.1.4 "Texture Queries"**

Change the first paragraph (page 184) to read:

... "It is somewhat different from the other get commands; <tex> is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D indicates a one-dimensional texture, TEXTURE_2D indicates a two-dimensional texture, TEXTURE_3D indicates a three-dimensional texture, and TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, and TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB indicate the respective face of a cube map texture.

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <coord> of Q when <pname> indicates REFLECTION_MAP_ARB or NORMAL_MAP_ARB.

INVALID_VALUE is generated when the target parameter to TexImage2D or CopyTexImage2D is one of the six cube map 2D image targets and the width and height parameters are not equal.

New State

(table 6.12, p202) add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_CUBE_MAP_ARB	B	IsEnabled	False	True if cube map texturing is enabled	3.8.10	texture/enable
TEXTURE_BINDING_CUBE_MAP_ARB	Z+	GetIntegerv	0	Texture object for TEXTURE_CUBE_MAP	3.8.8	texture
TEXTURE_CUBE_MAP_POSITIVE_X_ARB	nxI	GetTexImage	see 3.8	positive x face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_X_ARB	nxI	GetTexImage	see 3.8	negative x face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB	nxI	GetTexImage	see 3.8	positive y face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB	nxI	GetTexImage	see 3.8	negative y face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB	nxI	GetTexImage	see 3.8	positive z face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB	nxI	GetTexImage	see 3.8	negative z face cube map texture image at lod i	3.8	-

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture

(the type changes from 4xZ3 to 4xZ5)

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_CUBE_MAP_TEXTURE_SIZE_ARB	Z+	GetIntegerv	16	Maximum cube map texture image dimension	3.8.1	-

Backwards Compatibility

This extension replaces EXT_texture_cube_map. The tokens and name strings now refer to ARB instead of EXT. Enumerant values are unchanged.

Name

ARB_texture_env_add

Name Strings

GL_ARB_texture_env_add

Notice

Copyright OpenGL Architectural Review Board, 1999.

Status

Complete. Approved by ARB on 12/8/1999

Version

Last Modified Date: June 22, 2000
Author Revision: 0.3

Based on: EXT_texture_env_add
Date: 1999/03/22 Revision: 1.1

Number

ARB Extension #6

Dependencies

None

Overview

New texture environment function ADD is supported with the following equation:

$$Cv = \min(1, Cf + Ct)$$

New function may be specified by calling TexEnv with ADD token.

One possible application is to add a specular highlight texture to a Gouraud-shaded primitive to emulate Phong shading, in a single pass.

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnvfi when the <pname> parameter value is GL_TEXTURE_ENV_MODE

ADD

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

The description of TEXTURE_ENV_MODE in the first paragraph of section 3.8.9 should be modified as follows:

TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND or ADD;

Table 3.19 is augmented as follows:

Base Internal Format	DECAL tex func	BLEND tex func	ADD tex func
ALPHA	...	...	Rv = Rf
	...	...	Gv = Gf
	...	...	Bv = Bf
	...	...	Av = AfAt
LUMINANCE (or 1)	...	...	Rv = min(1, Rf+Lt)
	...	...	Gv = min(1, Gf+Lt)
	...	...	Bv = min(1, Bf+Lt)
	...	...	Av = Af
LUMINANCE_ALPHA (or 2)	...	...	Rv = min(1, Rf+Lt)
	...	...	Gv = min(1, Gf+Lt)
	...	...	Bv = min(1, Bf+Lt)
	...	...	Av = AfAt
INTENSITY	...	...	Rv = min(1, Rf+It)
	...	...	Gv = min(1, Gf+It)
	...	...	Bv = min(1, Bf+It)
	...	...	Av = min(1, Af+It)
RGB (or 3)	...	...	Rv = min(1, Rf+Rt)
	...	...	Gv = min(1, Gf+Gt)
	...	...	Bv = min(1, Bf+Bt)
	...	...	Av = Af
RGBA (or 4)	...	...	Rv = min(1, Rf+Rt)
	...	...	Gv = min(1, Gf+Gt)
	...	...	Bv = min(1, Bf+Bt)
	...	...	Av = AfAt

Table 3.19: Decal, blend and add texture functions.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

None

Errors

None

New State

The Type of TEXTURE_ENV_MODE in Table F.2 should be changed to

1 * xZ5

New Implementation Dependent State

None

Revision History

11/09/1999 0.1
- First ARB draft based on the original EXT draft.

1/13/2000 0.2
- Added justification to the overview
- Updated to describe modifications to 1.2.1 specification
- Added changes to description of TEXTURE_ENV_MODE parameter to TexEnv{if} and TexEnv{if}v
- Added change to TEXTURE_ENV_MODE type (Z4 -> Z5)

6/22/2000 0.3
- The addition should saturate to 1.

Name

ARB_texture_env_combine

Name Strings

GL_ARB_texture_env_combine

Version

Last modified date: 2001/05/21

Number

ARB Extension #17

Dependencies

This extension is written against the OpenGL 1.2.1 Specification.
OpenGL 1.1 and ARB_multitexture are required for this extension.

Overview

New texture environment function COMBINE_ARB allows programmable
texture combiner operations, including:

REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_ARB	Arg0 + Arg1 - 0.5
SUBTRACT_ARB	Arg0 - Arg1
INTERPOLATE_ARB	Arg0 * (Arg2) + Arg1 * (1-Arg2)

where Arg0, Arg1 and Arg2 are derived from

PRIMARY_COLOR_ARB	primary color of incoming fragment
TEXTURE	texture color of corresponding texture unit
CONSTANT_ARB	texture environment constant color
PREVIOUS_ARB	result of previous texture environment; on texture unit 0, this maps to PRIMARY_COLOR_ARB

In addition, the result may be scaled by 1.0, 2.0 or 4.0.

Issues

1. Should the explicit bias be removed in favor of an implcit bias as
   part of a ADD_SIGNED_ARB function?

   - RESOLVED: Yes. This pre-scale bias is a special case and will
     be treated as such.

2. Should the primary color of the incoming fragment be available to
   all texture environments? Currently it is only available to the
   texture environment of texture unit 0.

   - RESOLVED: Yes. PRIMARY_COLOR_ARB has been added as an input
     source.

3. Should textures from other texture units be allowed as sources?

   - RESOLVED: NO. Even though this adds a lot of flexibility that folks can use today, there is not enough support amonst the ARB participants to add it to the base spec.

4. All of the 1.2 modes except BLEND can be expressed in terms of this extension. Should texture color be allowed as a source for Arg2, so all of the 1.2 modes can be expressed? If so, should all color sources be allowed, to maintain orthogonality?

   - RESOLVED: Yes. This seems to be a reasonable area to expand functionality and remain backwards compatible with the EXT version of the extension.

5. If the texture environment for a given texture unit does not reference the texture object that is bound to that texture unit, does a valid texture object need to be bound that unit?

   - RESOLVED: Yes. Each texture unit implicitly references the texture object that is bound to that unit, regardless of the texture environment function. This may require that applications bind a dummy texture to the texture unit.

6. Should we allow the secondary color to take part in texture blending?

   - RESOLVED: Not in this extension. Secondary color was defined as a specular part of the lit color and does not have associated alpha. In order to do this right, the secondary color extension needs to be fixed first to allow a full featured color and clearly state the interaction of how it interacts with the color sum stage.

7. How exactly is this ARB extension different from the EXT version?

   - RESOLVED:

     1) This extension adds the GL_SUBTRACT_ARB mode

     2) OPERAND2_RGB_ARB can use SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA, and ONE_MINUS_SRC_ALPHA instead of just SRC_ALPHA (NV_texture_env_combine4 already provides this).

     3) OPERAND2_ALPHA_ARB can use SRC_ALPHA and ONE_MINUS_SRC_ALPHA instead of just SRC_ALPHA (NV_texture_env_combine4 already provides this).

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

COMBINE_ARB	0x8570

```
Accepted by the <pname> parameter of TexEnvf, TexEnvi, TexEnvfv,
and TexEnviv when the <target> parameter value is TEXTURE_ENV

    COMBINE_RGB_ARB                                 0x8571
    COMBINE_ALPHA_ARB                               0x8572
    SOURCE0_RGB_ARB                                 0x8580
    SOURCE1_RGB_ARB                                 0x8581
    SOURCE2_RGB_ARB                                 0x8582
    SOURCE0_ALPHA_ARB                               0x8588
    SOURCE1_ALPHA_ARB                               0x8589
    SOURCE2_ALPHA_ARB                               0x858A
    OPERAND0_RGB_ARB                                0x8590
    OPERAND1_RGB_ARB                                0x8591
    OPERAND2_RGB_ARB                                0x8592
    OPERAND0_ALPHA_ARB                              0x8598
    OPERAND1_ALPHA_ARB                              0x8599
    OPERAND2_ALPHA_ARB                              0x859A
    RGB_SCALE_ARB                                   0x8573
    ALPHA_SCALE

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
and TexEnviv when the <pname> parameter value is COMBINE_RGB_ARB
or COMBINE_ALPHA_ARB

    REPLACE
    MODULATE
    ADD
    ADD_SIGNED_ARB                                  0x8574
    INTERPOLATE_ARB                                 0x8575
    SUBTRACT_ARB                                    0x84E7

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
and TexEnviv when the <pname> parameter value is SOURCE0_RGB_ARB,
SOURCE1_RGB_ARB, SOURCE2_RGB_ARB, SOURCE0_ALPHA_ARB,
SOURCE1_ALPHA_ARB, or SOURCE2_ALPHA_ARB

    TEXTURE
    CONSTANT_ARB                                    0x8576
    PRIMARY_COLOR_ARB                               0x8577
    PREVIOUS_ARB                                    0x8578

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
and TexEnviv when the <pname> parameter value is
OPERAND0_RGB_ARB, OPERAND1_RGB_ARB, or OPERAND2_RGB_ARB

    SRC_COLOR
    ONE_MINUS_SRC_COLOR
    SRC_ALPHA
    ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
and TexEnviv when the <pname> parameter value is
OPERAND0_ALPHA_ARB, OPERAND1_ALPHA_ARB, or OPERAND2_ALPHA_ARB

    SRC_ALPHA
    ONE_MINUS_SRC_ALPHA
```

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is RGB_SCALE_ARB or ALPHA_SCALE

1.0
2.0
4.0

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE_ARB, the form of the texture function depends on the values of COMBINE_RGB_ARB and COMBINE_ALPHA_ARB, according to table 3.20. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_ARB and ALPHA_SCALE, respectively. The results are clamped to [0,1].

COMBINE_RGB_ARB	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_ARB	Arg0 + Arg1 - 0.5
INTERPOLATE_ARB	Arg0 * (Arg2) + Arg1 * (1-Arg2)
SUBTRACT_ARB	Arg0 - Arg1

COMBINE_ALPHA_ARB	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_ARB	Arg0 + Arg1 - 0.5
INTERPOLATE_ARB	Arg0 * (Arg2) + Arg1 * (1-Arg2)
SUBTRACT_ARB	Arg0 - Arg1

Table 3.20: COMBINE_ARB texture functions

The arguments Arg0, Arg1 and Arg2 are determined by the values of SOURCE<n>_RGB_ARB, SOURCE<n>_ALPHA_ARB, OPERAND<n>_RGB_ARB and OPERAND<n>_ALPHA_ARB. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. The relationship is described in tables 3.21 and 3.22.

SOURCE<n>_RGB_ARB	OPERAND<n>_RGB_ARB	Argument
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_ARB	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_ARB	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_ARB	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.21: Arguments for COMBINE_RGB_ARB functions

SOURCE<n>_ALPHA_ARB	OPERAND<n>_ALPHA_ARB	Argument
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_ARB	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_ARB	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_ARB	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.22: Arguments for COMBINE_ALPHA_ARB functions

The mapping of texture components to source components is summarized in Table 3.23. In the following table, At, Lt, It, Rt, Gt and Bt are the filtered texel values.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At

Table 3.23: Correspondence of texture components to source components for COMBINE_RGB_ARB and COMBINE_ALPHA_ARB arguments

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to Appendix F of the GL Specification (ARB Extensions)

Inserted after the second paragraph of F.2.12:

If the value of TEXTURE_ENV_MODE is COMBINE_ARB, the texture function associated with a given texture unit is computed using the values specified by SOURCE<n>_RGB_ARB, SOURCE<n>_ALPHA_ARB, OPERAND<n>_RGB_ARB and OPERAND<n>_ALPHA_ARB. If TEXTURE<n>_ARB is specified as SOURCE<n>_RGB_ARB or SOURCE<n>_ALPHA_ARB, the texture value from texture unit <n> will be used in computing the texture function for this texture unit.

Inserted after the third paragraph of F.2.12:

If a texture environment for a given texture unit references a texture unit that is disabled or does not have a valid texture object bound to it, then it is as if texture is disabled for the given texture unit. Every texture unit implicitly references the texture object that is bound to it, regardless of the texture function specified by COMBINE_RGB_ARB or COMBINE_ALPHA_ARB.

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for COMBINE_RGB_ARB or COMBINE_ALPHA_ARB is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_ARB, INTERPOLATE_ARB, or SUBTRACT_ARB

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_ARB, SOURCE1_RGB_ARB, SOURCE2_RGB_ARB, SOURCE0_ALPHA_ARB, SOURCE1_ALPHA_ARB or SOURCE2_ALPHA_ARB is not one of TEXTURE, CONSTANT_ARB, PRIMARY_COLOR_ARB, or PREVIOUS_ARB.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_ARB, OPERAND1_RGB_ARB, or OPERAND2_RGB_ARB is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_ARB, OPERAND1_ALPHA_ARB, or OPERAND2_ALPHA_ARB is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_VALUE is generated if <params> value for RGB_SCALE_ARB or ALPHA_SCALE is not one of 1.0, 2.0, or 4.0.

New State

Get Value	Get Command	Type	Initial Value	Attribute
COMBINE_RGB_ARB	GetTexEnviv	n x Z4	MODULATE	texture
COMBINE_ALPHA_ARB	GetTexEnviv	n x Z4	MODULATE	texture
SOURCE0_RGB_ARB	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_RGB_ARB	GetTexEnviv	n x Z3	PREVIOUS_ARB	texture
SOURCE2_RGB_ARB	GetTexEnviv	n x Z3	CONSTANT_ARB	texture
SOURCE0_ALPHA_ARB	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_ALPHA_ARB	GetTexEnviv	n x Z3	PREVIOUS_ARB	texture
SOURCE2_ALPHA_ARB	GetTexEnviv	n x Z3	CONSTANT_ARB	texture
OPERAND0_RGB_ARB	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND1_RGB_ARB	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND2_RGB_ARB	GetTexEnviv	n x Z1	SRC_ALPHA	texture
OPERAND0_ALPHA_ARB	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND1_ALPHA_ARB	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND2_ALPHA_ARB	GetTexEnviv	n x Z1	SRC_ALPHA	texture
RGB_SCALE_ARB	GetTexEnvfv	n x R3	1.0	texture
ALPHA_SCALE	GetTexEnvfv	n x R3	1.0	texture

New Implementation Dependent State

None

Revision History

01/05/21	mjk	Added ARB versus EXT differences issue
01/00/02	bpoddar	Added original EXT/ARB contributors to the contact list
00/12/13	bpoddar	Added enum value for SUBTRACT_ARB
00/12/06	bpoddar	Moved references to Ct<n> and At<n> to ARB_texture_env_crossbar spec.
00/12/01	bpoddar	Removed TEXTURE<n>_ARB since several companies had problems with this addition in the base spec.
00/11/13	bpoddar	Recreated 6/20 spec with language for dealing with inconsistent textures moved to appendix F.
00/06/20	rhammers	Changed behavior when dealing with references do disabled and inconsistent textures.
00/05/23	rhammers	Cleaned up for first draft of ARB version. Added issue -- TEXTURE with TEXTURE<n>_ARB Added issue .. "upstream" textures Listed get functions with description of enumerants. Added 1.1 and multitexture to dependencies

```
00/05/18  rhammers  First rev of ARB version of the spec. Based on
                    EXT_texture_env_combine.
                    Relaxed restriction on Arg2.
                    Added support for TEXTURE<n>_ARB.
                    Added SUBTRACT_ARB combiner function.
                    do disabled and inconsistent textures.
```

Name

ARB_texture_env_dot3

Name Strings

GL_ARB_texture_env_dot3

Contact

Bimal Poddar, Intel, bimal.poddar@intel.com
Dave Gosselin, ATI Technologies, Inc. (gosselin 'at' ati.com)
Dan Ginsburg, ATI Technologies, Inc. (dginsbur 'at' ati.com)

Status

Complete. Approved by ARB on February 16, 2001.

Version

Last modified date: 2001/05/16

Number

ARB Extension #19

Dependencies

This extension is written against the OpenGL 1.2.1 Specification.
OpenGL 1.1, ARB_multitexture and ARB_texture_env_combine are required
for this extension.

Overview

Adds new operation to the texture combiner operations.

```
DOT3_RGB_ARB                    Arg0 <dotprod> Arg1
DOT3_RGBA_ARB                   Arg0 <dotprod> Arg1
```

where Arg0, Arg1 are specified by <params> parameter of
TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname>
parameter value is SOURCE0_RGB_ARB and SOURCE1_RGB_ARB.

Issues

1. This extension is an ARB version of EXT_texture_env_dot3 which bears
   a copyright by ATI Technologies. Is ATI willing to have the ARB
   go ahead and modify their original spec and use it for the
   ARB extension.

   - RESOLVED: ATI does not have a problem with the copyright issue.

2. The EXT version of the spec does not multiply the output by
   RGB_SCALE_ARB and ALPHA_SCALE_ARB. There is no reason to impose this
   restriction since it makes the scale operations non-orthogonal.
   Should the enum values for the new tokens in this extension should
   be the same as the original EXT version?

- RESOLVED: No.

3. How exactly is this ARB extension different from the EXT version?

- RESOLVED: Scaling by 2.0 and 4.0 is supported by the ARB version, but not the EXT version (as noted above). Note that when DOT3_RGBA_ARB is used, the alpha component result is scaled based on the RGB scale factor rather than the alpha scale factor (the COMBINE_ALPHA_ARB function and scale factor are ignored). The COMBINE_ALPHA_ARB mode is ignored in the EXT version and the previous alpha is passed through; however, the ARB version abides by the COMBINE_ALPHA_ARB setting.

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is COMBINE_RGB_ARB

DOT3_RGB_ARB	0x86AE
DOT3_RGBA_ARB	0x86AF

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to table 3.20 of the ARB_texture_env_combine spec:

COMBINE_RGB_ARB	Texture Function
DOT3_RGB_ARB	4*((Arg0_r - 0.5)*(Arg1_r - 0.5) + (Arg0_g - 0.5)*(Arg1_g - 0.5) + (Arg0_b - 0.5)*(Arg1_b - 0.5)) This value is placed into all three r,g,b components of the output.
DOT3_RGBA_ARB	4*((Arg0_r - 0.5)*(Arg1_r - 0.5) + (Arg0_g - 0.5)*(Arg1_g - 0.5) + (Arg0_b - 0.5)*(Arg1_b - 0.5)) This value is placed into all four r,g,b,a components of the output. Note that the result generated from COMBINE_ALPHA_ARB function is ignored.

Additions to Chapter 4 of the OpenGL 1.2 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for COMBINE_RGB_ARB is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_ARB, INTERPOLATE_ARB, SUBTRACT_ARB, DOT3_RGB_ARB or DOT3_RGBA_ARB.

New State

None

New Implementation Dependent State

None

Revision History

01/05/16	mjk	Dot3 combiner operations not allowed for alpha portion
01/02/02	bpoddar	Added original EXT/ARB contributors to the contact list
00/12/13	bpoddar	Added enum values for DOT3_RGB_ARB and DOT3_RGBA_ARB Added resolution to issue # 1.
00/12/06	bpoddar	Fixed typos - EXT -> ARB, RED_SCALE -> RGB_SCALE
00/12/01	bpoddar	Created an ARB version of the ARB_texture_env_dot3 by breaking up the proposed ARB_texture_env_combine spec.

Name

ARB_transpose_matrix

Name Strings

GL_ARB_transpose_matrix

Status

Complete. Approved by ARB on 12/8/1999

Version

Last Modified Date: January 3, 2000
Author Revision: 1.3

Number

ARB Extension #3

Dependencies

This extensions is written against the OpenGL 1.2 Specification.
May be implemented in any version of OpenGL.

Overview

New functions and tokens are added allowing application matrices stored in row major order rather than column major order to be transferred to the OpenGL implementation. This allows an application to use standard C-language 2-dimensional arrays (m[row][col]) and have the array indices match the expected matrix row and column indexes. These arrays are referred to as transpose matrices since they are the transpose of the standard matrices passed to OpenGL.

This extension adds an interface for transfering data to and from the OpenGL pipeline, it does not change any OpenGL processing or imply any changes in state representation.

IP Status

No IP is believed to be involved.

Issues

* Why do this?

  It's very useful for layered libraries that desire to use two dimensional C arrays as matrices. It avoids having the layered library perform the transpose itself before calling OpenGL since most OpenGL implementations can efficiently perform the transpose while reading the matrix from client memory.

* Why not add a mode?

  It's substantially more confusing and complicated to add a mode.

Simply adding two new entry points saves considerable confusion and avoids having layered libraries need to query the current mode in order to send a matrix with the correct memory layout.

* Why not a utility routine in GLU

It costs some performance. It is believed that most OpenGL implementations can perform the transpose in place with negligble performance penalty.

* Why use the name transpose?

It's sure a lot less confusing than trying to ascribe unambiguous meaning to terms like row and column. It could be matrix_transpose rather than transpose_matrix though.

* Short Transpose to Trans?

New Procedures and Functions

```
void LoadTransposeMatrix{fd}ARB(T m[16]);
void MultTransposeMatrix{fd}ARB(T m[16]);
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev

```
TRANSPOSE_MODELVIEW_MATRIX_ARB    0x84E3
TRANSPOSE_PROJECTION_MATRIX_ARB   0x84E4
TRANSPOSE_TEXTURE_MATRIX_ARB      0x84E5
TRANSPOSE_COLOR_MATRIX_ARB        0x84E6
```

Additions to Chapter 2 of the 1.2 OpenGL Specification (OpenGL Operation)

Add to Section 2.10.2 Matrices <before LoadIdentity>

LoadTransposeMatrixARB takes a 4x4 matrix stored in row-major order as

Let transpose(m,n) be defined as

```
n[0] = m[0];
n[1] = m[4];
n[2] = m[8];
n[3] = m[12];
n[4] = m[1];
n[5] = m[5];
n[6] = m[9];
n[7] = m[13];
n[8] = m[2];
n[9] = m[6];
n[10] = m[10];
n[11] = m[14];
n[12] = m[3];
n[13] = m[7];
n[14] = m[11];
n[15] = m[15];
```

The effect of LoadTransposeMatrixARB(m) is then the same as the effect of the command sequence

```
float n[16];
transpose(m,n)
LoadMatrix(n);
```

The effect of MultTransposeMatrixARB(m) is then the same as the effect of the command sequence

```
float n[16];
transpose(m,n);
MultMatrix(n);
```

Additions to Chapter 3 of the 1.2 OpenGL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 OpenGL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.2 OpenGL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 OpenGL Specification (State and State Requests)

Matrices are queried and returned in their transposed form by calling GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev with <pname> set to TRANSPOSE_MODELVIEW_MATRIX_ARB, TRANSPOSE_PROJECTION_MATRIX_ARB, TRANSPOSE_TEXTURE_MATRIX_ARB, or TRANSPOSE_COLOR_MATRIX_ARB.
The effect of GetFloatv(TRANSPOSE_MODELVIEW_MATRIX_ARB,m) is then the same as the effect of the command sequence

```
    float n[16];
    GetFloatv(MODELVIEW_MATRIX_ARB,n);
    transpose(n,m);
```

Similar results occur for TRANSPOSE_PROJECTION_MATRIX_ARB, TRANSPOSE_TEXTURE_MATRIX_ARB, and TRANSPOSE_COLOR_MATRIX_ARB.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None

Additions to the GLX Specification

None

GLX Protocol

LoadTransposeMatrix and MultTransposeMatrix are layered on top of LoadMatrix and MultMatrix protocol performing client-side translation. The Get commands are passed over the wire as part of the generic Get protocol with no translation required.

Errors

No new errors, but error behavoir is inherited by the commands that the transpose commands are implemented on top of (LoadMatrix, MultMatrix, and Get*).

New State

None

TRANSPOSE_*_MATRIX_ARB refer to the same state as their non-transposed counterparts.

New Implementation Dependent State

None

Revision History

* Revision 1.1 - initial draft (18 Mar 1999)
* Revision 1.2 - changed to use layered specification and ARB affix (23 Nov 1999)
* Revision 1.3 - Minor tweaks to GLX protocol and Errors. (7 Dec 1999)

Conformance Testing

Load and Multiply the modelview matrix (initialized to identity each time) using LoadTransposeMatrixfARB and MultTransposeMatrixfARB with the matrix:

```
    ( 1  2  3  4 )
    ( 5  6  7  8 )
    ( 9 10 11 12 )
    (13 14 15 16 )
```

and get the modelview matrix using TRANSPOSE_MODELVIEW_MATRIX_ARB and validate that the matrix is correct. Get the matrix using MODELVIEW_MATRIX and verify that it is the transpose of the above matrix. Load and Multiply the modelview matrix using LoadMatrixf and MultMatrixf with the above matrix and verify that the correct matrix is on the modelview stack using gets of MODELVIEW_MATRIX and TRANSPOSE_MODELVIEW_MATRIX_ARB.

Name

EXT_abgr

Name Strings

GL_EXT_abgr

Version

$Date: 1995/03/31 04:40:18 $ $Revision: 1.10 $

Number

1

Dependencies

None

Overview

EXT_abgr extends the list of host-memory color formats. Specifically, it provides a reverse-order alternative to image format RGBA. The ABGR component order matches the cpack Iris GL format on big-endian machines.

New Procedures and Functions

None

New Tokens

Accepted by the <format> parameter of DrawPixels, GetTexImage, ReadPixels, TexImage1D, and TexImage2D:

ABGR_EXT	0x8000

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

One entry is added to table 3.5 (DrawPixels and ReadPixels formats). The new table is:

Name	Type	Elements	Target Buffer
COLOR_INDEX	Index	Color Index	Color
STENCIL_INDEX	Index	Stencil value	Stencil
DEPTH_COMPONENT	Component	Depth value	Depth
RED	Component	R	Color
GREEN	Component	G	Color
BLUE	Component	B	Color
ALPHA	Component	A	Color
RGB	Component	R, G, B	Color
RGBA	Component	R, G, B, A	Color
LUMINANCE	Component	Luminance value	Color
LUMINANCE_ALPHA	Component	Luminance value, A	Color
ABGR_EXT	Component	A, B, G, R	Color

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

The new format is added to the discussion of Obtaining Pixels from the Framebuffer. It should read " If the <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained."

The new format is added to the discussion of Index Lookup. It should read "If <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

One entry is added to tables 1 and 5 in the GLX Protocol Specification:

format	encoding
GL_ABGR_EXT	0x8000

Table A.2 is also extended:

format	nelements
GL_ABGR_EXT	4

Errors

None

New State

None

New Implementation Dependent State

None

Name

EXT_bgra

Name Strings

GL_EXT_bgra

Version

Microsoft revision 1.0, May 19, 1997 (drewb)
$Date: 1997/09/22 23:03:13 $ $Revision: 1.1 $

Number

129

Dependencies

None

Overview

EXT_bgra extends the list of host-memory color formats.
Specifically, it provides formats which match the memory layout of
Windows DIBs so that applications can use the same data in both
Windows API calls and OpenGL pixel API calls.

New Procedures and Functions

None

New Tokens

Accepted by the <format> parameter of DrawPixels, GetTexImage,
ReadPixels, TexImage1D, and TexImage2D:

BGR_EXT	0x80E0
BGRA_EXT	0x80E1

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

One entry is added to table 3.5 (DrawPixels and ReadPixels formats).
The new table is:

Name	Type	Elements	Target Buffer
COLOR_INDEX	Index	Color Index	Color
STENCIL_INDEX	Index	Stencil value	Stencil
DEPTH_COMPONENT	Component	Depth value	Depth
RED	Component	R	Color
GREEN	Component	G	Color
BLUE	Component	B	Color
ALPHA	Component	A	Color
RGB	Component	R, G, B	Color
RGBA	Component	R, G, B, A	Color
LUMINANCE	Component	Luminance value	Color
LUMINANCE_ALPHA	Component	Luminance value,A	Color
BGR_EXT	Component	B, G, R	Color
BGRA_EXT	Component	B, G, R, A	Color

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

The new format is added to the discussion of Obtaining Pixels from the Framebuffer. It should read " If the <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR_EXT, BGRA_EXT, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained."

The new format is added to the discussion of Index Lookup. It should read "If <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR_EXT, BGRA_EXT, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Revision History

Original draft, revision 0.9, October 13, 1995 (drewb)
Created
Minor revision, revision 1.0, May 19, 1997 (drewb)
Removed Microsoft Confidential.

Name

EXT_blend_color

Name Strings

GL_EXT_blend_color

Version

$Date: 1995/03/31 04:40:19 $ $Revision: 1.7 $

Number

2

Dependencies

None

Overview

Blending capability is extended by defining a constant color that can be included in blending equations. A typical usage is blending two RGB images. Without the constant blend factor, one image must have an alpha channel with each pixel set to the desired blend factor.

New Procedures and Functions

```
void BlendColorEXT(clampf red,
                   clampf green,
                   clampf blue,
                   clampf alpha);
```

New Tokens

Accepted by the <sfactor> and <dfactor> parameters of BlendFunc:

CONSTANT_COLOR_EXT	0x8001
ONE_MINUS_CONSTANT_COLOR_EXT	0x8002
CONSTANT_ALPHA_EXT	0x8003
ONE_MINUS_CONSTANT_ALPHA_EXT	0x8004

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

BLEND_COLOR_EXT	0x8005

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

The commands that control blending are now BlendFunc and BlendColorEXT. A constant color to be used in the blending equation is specified by BlendColorEXT. The four parameters are clamped to the range [0,1] before being stored. The default value for the constant blending color is (0,0,0,0).

The constant color can be used in both the source and destination blending factors. Four lines are added to table 4.1 and table 4.2:

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
DST_COLOR	(Rd/Kr, Gd/Kg, Bd/Kb, Ad/Ka)	
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd/Kr,Gd/Kg,Bd/Kb,Ad/Ka)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	NEW
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	NEW
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	NEW
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	NEW
SRC_ALPHA_SATURATE	(f, f, f, 1)	

Table 4.1: Values controlling the source blending function and the source blending values they compute. Ka = 2**m - 1, where m is the number of bits in the A color component. Kr, Kg, and Kb are similarly determined by the number of bits in the R, G, and B color components. f = min(As, 1-Ad) / Ka.

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs/Kr, Gs/Kg, Bs/Kb, As/Ka)	
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs/Kr,Gs/Kg,Bs/Kb,As/Ka)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	NEW
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	NEW
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	NEW
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	NEW

Table 4.2: Values controlling the destination blending function and the destination blending values they compute. Ka = 2**m - 1, where m is the number of bits in the A color component. Kr, Kg, and Kb are similarly determined by the number of bits in the R, G, and B color components.

Rc, Gc, Bc, and Ac are the four components of the constant blending color. These blend factors are not scaled by Kr, Kg, Kb, and Ka, because they are already in the range [0,1].

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
BlendColorEXT
    2           20              rendering command length
    2           4096            rendering command opcode
    4           FLOAT32  red
    4           FLOAT32  green
    4           FLOAT32  blue
    4           FLOAT32  alpha
```

Errors

INVALID_OPERATION is generated if BlendColorEXT is called between execution of Begin and the corresponding call to End.

New State

Get Value	Get Command	Initial Type	Value	Attrib
BLEND_COLOR_EXT	GetFloatv	C	(0,0,0,0)	color-buffer

New Implementation Dependent State

None

Name

EXT_blend_minmax

Name Strings

GL_EXT_blend_minmax

Version

$Date: 1995/03/31 04:40:34 $ $Revision: 1.3 $

Number

37

Dependencies

None

Overview

Blending capability is extended by respecifying the entire blend equation. While this document defines only two new equations, the BlendEquationEXT procedure that it defines will be used by subsequent extensions to define additional blending equations.

The two new equations defined by this extension produce the minimum (or maximum) color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

Issues

* I've prefixed the ADD token with FUNC, to indicate that the blend equation includes the parameters specified by BlendFunc. (The min and max equations don't.) Is this necessary? Is it too ugly? Is there a better way to accomplish the same thing?

New Procedures and Functions

void BlendEquationEXT(enum mode);

New Tokens

Accepted by the <mode> parameter of BlendEquationEXT:

FUNC_ADD_EXT	0x8006
MIN_EXT	0x8007
MAX_EXT	0x8008

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

BLEND_EQUATION_EXT	0x8009

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

The GL Specification defines a single blending equation. This extension introduces a blend equation mode that is specified by calling BlendEquationEXT with one of three enumerated values. The default value FUNC_ADD_EXT specifies that the blending equation defined in the GL Specification be used. This equation is

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

     /  1.0 C' > 1.0
C = (
     \   C' C' <= 1.0
```

where Cs and Cd are the source and destination colors, and S and D are as specified by BlendFunc.

If BlendEquationEXT is called with <mode> set to MIN_EXT, the blending equation becomes

```
C = min (Cs, Cd)
```

Finally, if BlendEquationEXT is called with <mode> set to MAX_EXT, the blending equation becomes

```
C = max (Cs, Cd)
```

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

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
    BlendEquationEXT
           2           8               rendering command length
           2           4097            rendering command opcode
           4           ENUM       mode
```

Errors

INVALID_ENUM is generated by BlendEquationEXT if its single parameter is not FUNC_ADD_EXT, MIN_EXT, or MAX_EXT.

INVALID_OPERATION is generated if BlendEquationEXT is executed between the execution of Begin and the corresponding execution to End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
BLEND_EQUATION_EXT	GetIntegerv	Z3	FUNC_ADD_EXT	color-buffer

New Implementation Dependent State

None

Name

EXT_blend_subtract

Name Strings

GL_EXT_blend_subtract

Version

$Date: 1995/03/31 04:40:39 $ $Revision: 1.4 $

Number

38

Dependencies

EXT_blend_minmax affects the definition of this extension

Overview

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.

New Procedures and Functions

None

New Tokens

Accepted by the <mode> parameter of BlendEquationEXT:

FUNC_SUBTRACT_EXT	0x800A
FUNC_REVERSE_SUBTRACT_EXT	0x800B

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Two additional blending equations are defined. If BlendEquationEXT is called with <mode> set to FUNC_SUBTRACT_EXT, the blending equation becomes

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

     /  0.0 C' < 0.0
C = (
     \   C' C' >= 0.0
```

where Cs and Cd are the source and destination colors, and S and D are as specified by BlendFunc.

If BlendEquationEXT is called with <mode> set to FUNC_REVERSE_SUBTRACT_EXT, the blending equation becomes

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

     /  0.0 C' < 0.0
C = (
     \   C' C' >= 0.0
```

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

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on EXT_blend_minmax

If this extension is supported, but EXT_blend_minmax is not, then this extension effectively defines the procedure BlendEquationEXT, its parameter FUNC_ADD_EXT, and the query target BLEND_EQUATION_EXT, as described in EXT_blend_minmax. It is therefore as though EXT_blend_minmax were also supported, except that equations MIN_EXT and MAX_EXT are not supported.

Errors

INVALID_ENUM is generated by BlendEquationEXT if its single parameter is not FUNC_ADD_EXT, MIN_EXT, MAX_EXT, FUNC_SUBTRACT_EXT, or FUNC_REVERSE_SUBTRACT_EXT.

INVALID_OPERATION is generated if BlendEquationEXT is executed between the execution of Begin and the corresponding execution to End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
BLEND_EQUATION_EXT	GetIntegerv	Z5	FUNC_ADD_EXT	color-buffer

New Implementation Dependent State

None

XXX - Not complete yet!!!

Name

EXT_compiled_vertex_array

Name Strings

GL_EXT_compiled_vertex_array

Version

$Date: 1996/11/21 00:52:19 $ $Revision: 1.3 $

Number

97

Dependencies

None

Overview

This extension defines an interface which allows static vertex array data to be cached or pre-compiled for more efficient rendering. This is useful for implementations which can cache the transformed results of array data for reuse by several DrawArrays, ArrayElement, or DrawElements commands. It is also useful for implementations which can transfer array data to fast memory for more efficient processing.

For example, rendering an M by N mesh of quadrilaterals can be accomplished by setting up vertex arrays containing all of the vertexes in the mesh and issuing M DrawElements commands each of which operate on 2 * N vertexes. Each DrawElements command after the first will share N vertexes with the preceding DrawElements command. If the vertex array data is locked while the DrawElements commands are executed, then OpenGL may be able to transform each of these shared vertexes just once.

Issues

* Is compiled_vertex_array the right name for this extension?

* Should there be an implementation defined maximum number of array elements which can be locked at a time (i.e. MAX_LOCKED_ARRAY_SIZE)?

  Probably not, the lock request can always be ignored with no resulting change in functionality if there are insufficent resources, and allowing the GL to define this limit can make things difficult for applications.

* Should there be any restrictions on what state can be changed while the vertex array data is locked?

  Probably not. The GL can check for state changes and invalidate any cached vertex state that may be affected. This is likely to cause a performance hit, so the preferred use will be to not change

state while the vertex array data is locked.

New Procedures and Functions

```
void LockArraysEXT (int first, sizei count)
void UnlockArraysEXT (void)
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

ARRAY_ELEMENT_LOCK_FIRST_EXT	0x81A8
ARRAY_ELEMENT_LOCK_COUNT_EXT	0x81A9

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

After the discussion of InterleavedArrays, add a description of array compiling/locking.

The currently enabled vertex arrays can be locked with the command LockArraysEXT. When the vertex arrays are locked, the GL 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 command UnlockArraysEXT.

Between LockArraysEXT and UnlockArraysEXT the application should ensure 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 LockArraysEXT and UnlockArraysEXT commands may affect calls may affect DrawArrays, ArrayElement, or DrawElements commands in non-sequential ways.

While using a compiled vertex array, references to array elements by the commands DrawArrays, ArrayElement, or DrawElements which are outside of the range specified by <first> and <count> are undefined.

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

LockArraysEXT and UnlockArraysEXT are not complied into display lists but are executed immediately.

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

None

Additions to the GLX Specification

XXX - Not complete yet!!!

GLX Protocol

XXX - Not complete yet!!!

Errors

INVALID_VALUE is generated if LockArrarysEXT parameter <first> is less than zero.

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

INVALID_OPERATION is generated if LockArraysEXT is called between execution of LockArraysEXT and corresponding execution of UnlockArraysEXT.

INVALID_OPERATION is generated if UnlockArraysEXT is called without a corresponding previous execution of LockArraysEXT.

INVALID_OPERATION is generated if LockArraysEXT or UnlockArraysEXT is called between execution of Begin and the corresponding execution of End.

New State

Get Value	Get Command	Type	Initial Value	Attrib
ARRAY_ELEMENT_LOCK_FIRST_EXT	GetIntegerv	Z+	0	client-vertex-array
ARRAY_ELEMENT_LOCK_COUNT_EXT	GetIntegerv	Z+	0	client-vertex-array

New Implementation Dependent State

None

Name

EXT_draw_range_elements

Name Strings

GL_EXT_draw_range_elements

Version

$Date: 1997/5/19

Number

112

Status

Superceded by OpenGL 1.2 functionaltity.
See section 2.8 (page 25) of the OpenGL 1.2.1 specification.

Proposal

Add a new vertex array rendering command:

```
void glDrawRangeElementsEXT(
     GLenum mode,
     GLuint start,
     GLuint end,
     GLsizei count,
     GLenum type,
     const GLvoid *indices
);
```

Add two implementation-dependent limits for describing data size recommendations for glDrawRangeElementsEXT:

```
GL_MAX_ELEMENTS_VERTICES_EXT  0x80E8
GL_MAX_ELEMENTS_INDICES_EXT   0x80E9
```

glDrawRangeElementsEXT is a restricted form of glDrawElements. All vertices referenced by indices must lie between start and end inclusive. Not all vertices between start and end must be referenced, however unreferenced vertices may be sent through some of the vertex pipeline before being discarded, reducing performance from what could be achieved by an optimal index set. Index values which lie outside the range will cause implementation-dependent results.

glDrawRangeElementsEXT may also be further constrained to only operate at maximum performance for limited amounts of data. Implementations may advertise recommended maximum amounts of vertex and index data using the GL_MAX_ELEMENTS_VERTICES_EXT and GL_MAX_ELEMENTS_INDICES_EXT enumerants. If a particular call to glDrawRangeElementsEXT has (end-start+1) greater than GL_MAX_ELEMENTS_VERTICES_EXT or if count is greater than GL_MAX_ELEMENTS_INDICES_EXT then the implementation may be forced to process the data less efficiently than it could have with less data. An implementation which has no effective limits can advertise the maximum

integer value for the two enumerants. An implementation must always process a glDrawRangeElementsEXT call with valid parameters regardless of the amount of data passed in the call.

GL_INVALID_VALUE will be returned if end is less than start. Other errors are as for glDrawElements.

Motivation:
Rendering primitives from indexed vertex lists is a fairly common graphics operation, particularly in modeling applications such as VRML viewers. OpenGL 1.1 added support for the glDrawElements API to allow rendering of primitives by indexing vertex array data.

The specification of glDrawElements does not allow optimal performance for some OpenGL implementations, however. In particular, it has no restrictions on the number of indices given, the number of unique vertices referenced nor a direct indication of the set of unique vertices referenced by the given indices. This forces some OpenGL implementations to walk the index data given, building up a separate list of unique vertex references for later use in the pipeline. Additionally, since some OpenGL implementations have internal limitations on how many vertices they can deal with simultaneously the unbounded nature of glDrawElements requires the implementation to be prepared to segment the input data and do multiple passes. These preprocessing steps can consume a significant amount of time.

Such preprocessing can be done once and stored when building display lists but this only works for objects whose geometry does not change. Applications using morphing objects or other objects that are changing dynamically cannot take advantage of display lists and so must pay the preprocessing penalty on every redraw.

glDrawRangeElementsEXT is designed to avoid the preprocessing steps which may be necessary for glDrawElements. As such it does not have the flexibility of glDrawElements but it is sufficiently functional for a large class of applications to benefit from its use.
glDrawRangeElementsEXT enhances glDrawElements in two ways:
1. The set of unique vertices referenced by the indices is explicitly indicated via the start and end parameters, removing the necessity to determine this through examination of the index data. The implementation is given a contiguous chunk of vertex data that it can immediately begin streaming through the vertex pipeline.
2. Recommended limits on the amount of data to be processed can be indicated by the implementation through GL_MAX_ELEMENTS_VERTICES_EXT and GL_MAX_ELEMENTS_INDICES_EXT. If an application respects these limits it removes the need to split the incoming data into multiple chunks since the maximums can be set to the optimal values for the implementation to handle in one pass.

The first restriction isn't particularly onerous for applications since they can always call glDrawElements in the case where they cannot or do not know whether they can call glDrawRangeElementsEXT. Performance should be at least as good as it was calling glDrawElements alone. The second point isn't really a restriction as glDrawRangeElementsEXT doesn't fail if the data size limits are exceeded.

OpenGL implementation effort is also minimal. For implementations where

glDrawElements performance is not affected by preprocessing glDrawRangeElementsEXT can be implemented simply as a call to glDrawElements and the maximums set to the maximum integer value. For the case where glDrawElements is doing non-trivial preprocessing there is probably already an underlying routine that takes consecutive, nicely sectioned index and vertex chunks that glDrawRangeElementsEXT can plug directly in to.

Design Decisions

The idea of providing a set of vertex indices along with a set of element indices was considered but dropped as it still may require some preprocessing, although there is some reduction in overhead from glDrawElements. The implementation may require internal vertex data to be contiguous, in which case a gather operation would have to be performed with the vertex index list before vertex data could be processed. It is expected that most apps will keep vertex data for particular elements packed consecutively anyway so the added flexibility of a vertex index list would potentially impose overhead with little expected benefit. In the case where a vertex index list really is necessary to avoid performance penalties due to sparse vertex usage glDrawElements should provide performance similar to what such an API would have.

The restriction on maximum data size cannot easily be lifted without potential performance implications. For implementations which have an internal maximum vertex buffer size it would be necessary to break up large data sets into multiple chunks. Splitting indexed data requires walking the indices and gathering those that fall within particular chunks into sets for processing, a time-consuming operation. Splitting the indices themselves is easier but still requires some processing to handle connected primitives that cross a split.

Name

EXT_fog_coord

Name Strings

GL_EXT_fog_coord

Status

Shipping (version 1.6)

Version

$Date: 1999/06/21 19:57:19 $ $Revision: 1.11 $

Number

149

Dependencies

OpenGL 1.1 is required.
The extension is written against the OpenGL 1.2 Specification.

Overview

This extension allows specifying an explicit per-vertex fog coordinate to be used in fog computations, rather than using a fragment depth-based fog equation.

Issues

* Should the specified value be used directly as the fog weighting factor, or in place of the z input to the fog equations?

  As the z input; more flexible and meets ISV requests.

* Do we want vertex array entry points? Interleaved array formats?

  Yes for entry points, no for interleaved formats, following the argument for secondary_color.

* Which scalar types should FogCoord accept? The full range, or just the unsigned and float versions? At the moment it follows Index(), which takes unsigned byte, signed short, signed int, float, and double.

  Since we're now specifying a number which behaves like an eye-space distance, rather than a [0,1] quantity, integer types are less useful. However, restricting the commands to floating point forms only introduces some nonorthogonality.

  Restrict to only float and double, for now.

* Interpolation of the fog coordinate may be perspective-correct or not. Should this be affected by PERSPECTIVE_CORRECTION_HINT,

FOG_HINT, or another to-be-defined hint?

PERSPECTIVE_CORRECTION_HINT; this is already defined to affect all interpolated parameters. Admittedly this is a loss of orthogonality.

* Should the current fog coordinate be queryable?

Yes, but it's not returned by feedback.

* Control the fog coordinate source via an Enable instead of a fog parameter?

No. We might want to add more sources later.

* Should the fog coordinate be restricted to non-negative values?

Perhaps. Eye-coordinate distance of fragments will be non-negative due to clipping. Specifying explicit negative coordinates may result in very large computed f values, although they are defined to be clipped after computation.

* Use existing DEPTH enum instead of FRAGMENT_DEPTH? Change name of FRAGMENT_DEPTH_EXT to FOG_FRAGMENT_DEPTH_EXT?

Use FRAGMENT_DEPTH_EXT; FOG_FRAGMENT_DEPTH_EXT is somewhat misleading, since fragment depth itself has no dependence on fog.

New Procedures and Functions

```
void FogCoord[fd]EXT(T coord)
void FogCoord[fd]vEXT(T coord)
void FogCoordPointerEXT(enum type, sizei stride, void *pointer)
```

New Tokens

Accepted by the <pname> parameter of Fogi and Fogf:

```
FOG_COORDINATE_SOURCE_EXT         0x8450
```

Accepted by the <param> parameter of Fogi and Fogf:

```
FOG_COORDINATE_EXT                0x8451
FRAGMENT_DEPTH_EXT                0x8452
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
CURRENT_FOG_COORDINATE_EXT        0x8453
FOG_COORDINATE_ARRAY_TYPE_EXT     0x8454
FOG_COORDINATE_ARRAY_STRIDE_EXT   0x8455
```

Accepted by the <pname> parameter of GetPointerv:

```
FOG_COORDINATE_ARRAY_POINTER_EXT    0x8456
```

Accepted by the <array> parameter of EnableClientState and DisableClientState:

FOG_COORDINATE_ARRAY_EXT 0x8457

Additions to Chapter 2 of the OpenGL 1.2 Specification (OpenGL Operation)

These changes describe a new current state type, the fog coordinate, and the commands to specify it:

- (2.6, p. 12) Second paragraph changed to:

"Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current fog coordinate may be used in processing each vertex."

- 2.6.3, p. 19) First paragraph changed to

"The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, texture coordinates, and fog coordinates (Vertex, Color, Index, Normal, TexCoord, FogCoord)..."

- (2.7, p. 20) Insert the following paragraph following the third paragraph describing current normals:

" The current fog coodinate is set using
void FogCoord[fd]EXT(T coord)
void FogCoord[fd]vEXT(T coord)."

The last paragraph is changed to read:

"The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, one floating-point value to store the current fog coordinate, four floating-point values to store the current RGBA color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial fog coordinate is zero. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial color index is 1."

- (2.8, p. 21) Added fog coordinate command for vertex arrays:

Change first paragraph to read:

"The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution

of a single GL command. The client may specify up to seven arrays: one each to store edge flags, texture coordinates, fog coordinates, colors, color indices, normals, and vertices. The commands"

Add to functions listed following first paragraph:

void FogCoordPointerEXT(enum type, sizei stride, void *pointer)

Add to table 2.4 (p. 22):

Command	Sizes	Types
FogCoordPointerEXT	1	float,double

Starting with the second paragraph on p. 23, change to add FOG_COORDINATE_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
void EnableClientState(enum array)
void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, FOG_COORDINATE_ARRAY_EXT, COLOR_ARRAY, INDEX_ARRAY, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, fog coordinate, color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, fog coordinate, color, color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, FogCoord<type>v, Color<size><type>v, Index<type>v, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable fog coordinate array for canned interleaved array formats. After the lines

```
DisableClientState(EDGE_FLAG_ARRAY);
DisableClientState(INDEX_ARRAY);
```

insert the line

```
DisableClientState(FOG_COORDINATE_ARRAY_EXT);
```

Substitute "seven" for every occurence of "six" in the final paragraph on p. 27.

- (2.12, p. 41) Add fog coordinate to the current rasterpos state.

  Change the first sentence of the first paragraph to read

  "The state required for the current raster position consists of three window coordinates x_w, y_w, and z_w, a clip coordinate w_c value, an eye coordinate distance, a fog coordinate, a valid bit, and associated data consisting of a color and texture coordinates."

  Change the last paragraph to read

  "The current raster position requires six single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, its eye coordinate distance, and its fog coordinate, a single valid bit, a color (RGBA color and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both (0,0,0,1), the fog coordinate is 0, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value."

- (3.10, p. 139) Change the second and third paragraphs to read

  "This factor f may be computed according to one of three equations:"

  ```
  f = exp(-d*c) (3.24)
  f = exp(-(d*c)^2)   (3.25)
  f = (e-c)/(e-s)     (3.26)
  ```

  If the fog source (as defined below) is FRAGMENT_DEPTH_EXT, then c is the eye-coordinate distance from the eye, (0 0 0 1) in eye coordinates, to the fragment center. If the fog source is FOG_COORDINATE_EXT, then c is the interpolated value of the fog coordinate for this fragment. The equation and the fog source, along with either d or e and s, is specified with

  ```
  void Fog{if}(enum pname, T param);
  void Fog{if}v(enum pname, T params);
  ```

  If <pname> is FOG_MODE, then <param> must be, or <param> must point to an integer that is one of the symbolic constants EXP, EXP2, or LINEAR, in which case equation 3.24, 3.25, or 3.26,, respectively, is selected for the fog calculation (if, when 3.26 is selected, e = s, results are undefined). If <pname> is FOG_COORDINATE_SOURCE_EXT, then <param> is or <params> points to an integer that is one of the symbolic constants FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT. If <pname> is FOG_DENSITY, FOG_START, or FOG_END, then <param> is or <params> points to a value that is d, s, or e, respectively. If d is specified less than zero, the error INVALID_VALUE results."

- (3.10, p. 140) Change the last paragraph preceding section 3.11 to read

  "The state required for fog consists of a three valued integer to select the fog equation, three floating-point values d, e, and s, an RGBA fog color and a fog color index, a two-valued integer to select the fog coordinate source, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_COORDINATE_SOURCE_EXT is FRAGMENT_DEPTH_EXT, FOG_MODE is EXP, d = 1.0, e = 1.0, and s = 0.0; C_f = (0,0,0,0) and i_f=0."

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.2 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

Two new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

FogCoordfvEXT

2	8	rendering command length
2	4124	rendering command opcode
4	FLOAT32	v[0]

FogCoorddvEXT

2	12	rendering command length
2	4125	rendering command opcode
8	FLOAT64	v[0]

Errors

INVALID_ENUM is generated if FogCoordPointerEXT parameter <type> is not FLOAT or DOUBLE.

INVALID_VALUE is generated if FogCoordPointerEXT parameter <stride> is negative.

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_FOG_COORDINATE_EXT	R	GetIntegerv, GetFloatv	0	Current fog coordinate	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_COORDINATE_ARRAY_EXT	B	IsEnabled	False	Fog coord array enable	2.8	vertex-array
FOG_COORDINATE_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of fog coordinate	2.8	vertex-array
FOG_COORDINATE_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between fog coords	2.8	vertex-array
FOG_COORDINATE_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the fog coord array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_COORDINATE_SOURCE_EXT	Z2	GetIntegerv, GetFloatv	FRAGMENT_DEPTH_EXT	Source of fog coordinate for fog calculation	3.10	fog

Revision History

* Revision 1.6 - Functionality complete

* Revision 1.7-1.9 - Fix typos and add fields to bring up to date with the new extension template. No functionality changes.

Name

EXT_packed_pixels

Name Strings

GL_EXT_packed_pixels

Version

$Date: 1997/09/22 23:23:58 $ $Revision: 1.21 $

Number

23

Dependencies

EXT_abgr affects the definition of this extension
EXT_texture3D affects the definition of this extension
EXT_subtexture affects the definition of this extension
EXT_histogram affects the definition of this extension
EXT_convolution affects the definition of this extension
SGI_color_table affects the definition of this extension
SGIS_texture4D affects the definition of this extension
EXT_cmyka affects the definition of this extension

Overview

This extension provides support for packed pixels in host memory. 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 PACK_SKIP_PIXELS, PACK_ROW_LENGTH, PACK_SKIP_ROWS, PACK_IMAGE_HEIGHT_EXT, PACK_SKIP_IMAGES_EXT, PACK_SWAP_BYTES, PACK_ALIGNMENT, and their unpacking counterparts all work correctly with packed pixels.

New Procedures and Functions

None

New Tokens

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, GetHistogramEXT, GetMinmaxEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, GetSeparableFilterEXT, ColorTableSGI, GetColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS:

UNSIGNED_BYTE_3_3_2_EXT	0x8032
UNSIGNED_SHORT_4_4_4_4_EXT	0x8033
UNSIGNED_SHORT_5_5_5_1_EXT	0x8034
UNSIGNED_INT_8_8_8_8_EXT	0x8035
UNSIGNED_INT_10_10_10_2_EXT	0x8036

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

The five tokens defined by this extension are added to Table 3.4:

<type> Parameter Token Value	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	No
BYTE	byte	No
UNSIGNED_SHORT	ushort	No
SHORT	short	No
UNSIGNED_INT	uint	No
INT	int	No
FLOAT	float	No
BITMAP	ubyte	Yes
UNSIGNED_BYTE_3_3_2_EXT	ubyte	Yes
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	Yes
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	Yes
UNSIGNED_INT_8_8_8_8_EXT	uint	Yes
UNSIGNED_INT_10_10_10_2_EXT	uint	Yes

Table 3.4: DrawPixels and ReadPixels <type> parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.3.

[Section 3.6.3 of the GL Specification (Rasterization of Pixel Rectangles) is rewritten as follows:]

3.6.3 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in host memory is diagrammed in Figure 3.7. We describe the stages of this process in the order in which they occur.

Pixels are drawn using

```
void DrawPixels(sizei width,
                sizei height,
                enum format,
                enum type,
                void* data);
```

<format> is a symbolic constant indicating what the values in memory represent. <width> and <height> are the width and height, respectively, of the pixel rectangle to be drawn. <data> is a pointer to the data to be drawn. These data are represented with one of seven GL data types, specified by <type>. The correspondence between the thirteen <type> token values and the GL data types they indicate is given in Table 3.4. If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION occurs. Some additional constraints on the combinations of <format>

and <type> values that are accepted are discussed below.

Unpacking

Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating-point values (GL data type float). These elements are grouped into sets of one, two, three, four, or five values, depending on the <format>, to form a group. Table 3.5 summarizes the format of groups obtained from memory. It also indicates those formats that yield indices and those that yield components.

Format Name	Target Buffer	Element Meaning and Order
COLOR_INDEX	Color	Color index
STENCIL_INDEX	Stencil	Stencil index
DEPTH_COMPONENT	Depth	Depth component
RED	Color	R component
GREEN	Color	G component
BLUE	Color	B component
ALPHA	Color	A component
RGB	Color	R, G, B components
RGBA	Color	R, G, B, A components
ABGR_EXT	Color	A, B, G, R components
CMYK_EXT	Color	Cyan, Magenta, Yellow, Black components
CMYKA_EXT	Color	Cyan, Magenta, Yellow, Black, A components
LUMINANCE	Color	Luminance component
LUMINANCE_ALPHA	Color	Luminance, A components

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If UNPACK_SWAP_BYTES is set to TRUE, however, then the values are interpreted with the bit orderings modified as per the table below. The modified bit orderings are defined only if the GL data type ubyte has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

Element Size	Default Bit Ordering	Modified Bit Ordering
8-bit	[7..0]	[7..0]
16-bit	[15..0]	[7..0] [15..8]
32-bit	[31..0]	[7..0] [15..8] [23..16] [31..24]

Table: Bit ordering modification of elements when UNPACK_SWAP_BYTES is TRUE. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of rows, with the first element of the first group of the first row pointed to by the pointer passed to

DrawPixels. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is <width>; otherwise the number of groups is UNPACK_ROW_LENGTH. If the first element of the first row is at location p in memory, then the location of the first element of the Nth row is

```
p + Nk
```

where N is the row number (counting from zero) and k is defined as

```
     / nl               s >= a
k = <
     \ a/s * ceiling(snl/a)   s < a
```

where n is the number of elements in a group, l is the number of groups in a row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then k = nl for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to DrawPixels is effectively advanced by

```
UNPACK_SKIP_PIXELS * n + UNPACK_SKIP_ROWS * k
```

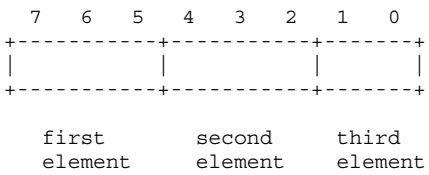
elements. Then <width> groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. <height> sets of <width> groups of values are obtained this way. See Figure 3.8.

Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_INT_8_8_8_8, or UNSIGNED_INT_10_10_10_2 is a special case in which all the elements of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. The number of elements per packed pixel is fixed by the type, and must match the number of elements per group indicated by the <format> parameter. (See the table below.) The error INVALID_OPERATION is generated if a mismatch occurs.

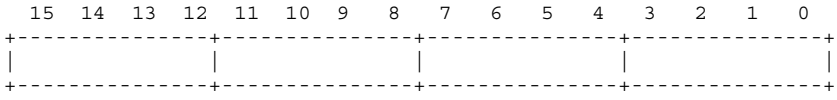
<type> Parameter Token Name	GL Data Type	Number of Elements	Matching Pixel Formats
UNSIGNED_BYTE_3_3_2_EXT	ubyte	3	RGB
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_INT_8_8_8_8_EXT	uint	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_INT_10_10_10_2_EXT	uint	4	RGBA,ABGR_EXT,CMYK_EXT

Bitfield locations of the first, second, third, and fourth elements of each packed pixel type are illustrated in the diagrams below. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed elements are right-justified in the pixel.

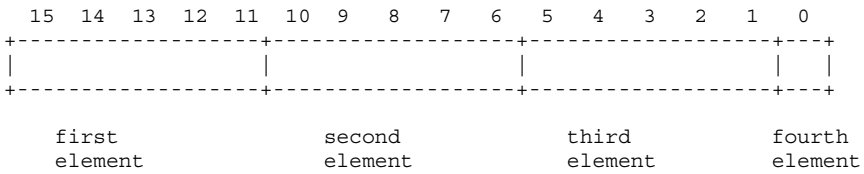
UNSIGNED_BYTE_3_3_2_EXT:



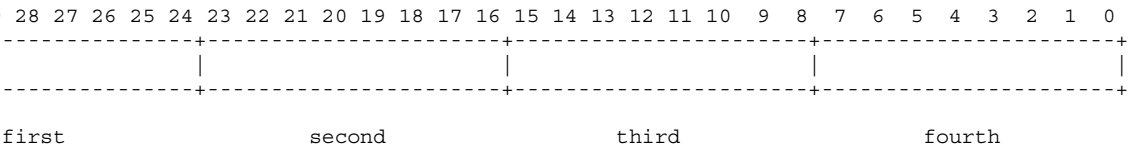
UNSIGNED_SHORT_4_4_4_4_EXT:



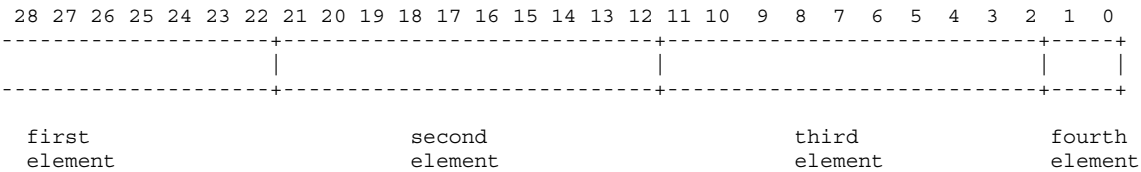
UNSIGNED_SHORT_5_5_5_1_EXT:



UNSIGNED_INT_8_8_8_8_EXT:



UNSIGNED_INT_10_10_10_2_EXT:



The assignment of elements to fields in the packed pixel is as described in the table below:

Format	First Element	Second Element	Third Element	Fourth Element
RGB	red	green	blue	
RGBA	red	green	blue	alpha
ABGR_EXT	alpha	blue	green	red
CMYK_EXT	cyan	magenta	yellow	black

Byte swapping, if enabled, is performed before the elements are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "element" and the number of elements per group is understood to be one.

Calling DrawPixels with a <type> of BITMAP is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to DrawPixels. The first element of the second row is the first bit (again as defined above) of the ubyte at location p+k, where k is computed as

```
k = a * ceiling(nl/8a)
```

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to DrawPixels is effectively advanced by

```
UNPACK_SKIP_ROWS * k
```

ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent <width> 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by k ubytes. <height> sets of <width> elements are obtained this way.

Conversion to floating-point

This step applies only to groups of components. It is not performed on indices. Each element in a group is converted to a floating-point value according to the appropriate formula in Table 2.4 (section 2.12). Unsigned integer bitfields extracted from packed pixels are interpreted using the formula

```
f = c / ((2**N)-1)
```

where c is the value of the bitfield (interpreted as an unsigned integer), N is the number of bits in the bitfield, and the division is performed in floating point.

[End of changes to Section 3.6.3]

If this extension is supported, all commands that accept pixel data also accept packed pixel data. These commands are DrawPixels, TexImage1D, TexImage2D, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, ColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

[Make the following changes to Section 4.3.2 (Reading Pixels):]

Final Conversion

For an index, if the <type> is not FLOAT, final conversion consists of masking the index with the value given in Table 4.6; if the <type> is FLOAT, then the integer index is converted to a GL float data value. For a component, each component is first clamped to [0,1]. Then, the appropriate conversion formula from Table 4.7 is applied to the component.

<type> Parameter	Index Mask
UNSIGNED_BYTE	2**8 - 1
BITMAP	1
BYTE	2**7 - 1
UNSIGNED_SHORT	2**16 - 1
SHORT	2**15 - 1
UNSIGNED_INT	2**32 - 1
INT	2**31 - 1

Table 4.6: Index masks used by ReadPixels. Floating point data are not masked.

<type> Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	c = ((2**8)-1)*f
BYTE	byte	c = (((2**8)-1)*f-1)/2
UNSIGNED_SHORT	ushort	c = ((2**16)-1)*f
SHORT	short	c = (((2**16)-1)*f-1)/2
UNSIGNED_INT	uint	c = ((2**32)-1)*f
INT	int	c = (((2**32)-1)*f-1)/2
FLOAT	float	c = f
UNSIGNED_BYTE_3_3_2_EXT	ubyte	c = ((2**N)-1)*f
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	c = ((2**N)-1)*f
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	c = ((2**N)-1)*f
UNSIGNED_INT_8_8_8_8_EXT	uint	c = ((2**N)-1)*f
UNSIGNED_INT_10_10_10_2_EXT	uint	c = ((2**N)-1)*f

Table 4.7: Reversed component conversions - used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation (f) to a datum of the specified GL data type (c) using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2.) Equations with N as the exponent are performed for each bitfield of the packed data type, with N set to the number of bits in the bitfield.

Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for DrawPixels. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory must where the ith group of the jth row would be taken from for DrawPixels. See Unpacking under section 3.6.3. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_.

[End of changes to Section 4.3.2]

If this extension is supported, all commands that return pixel data also return packed pixel data. These commands are ReadPixels, GetTexImage, GetHistogramEXT, GetMinmaxEXT, GetConvolutionFilterEXT, GetSeparableFilterEXT, and GetColorTableSGI.

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on EXT_abgr

If EXT_abgr is not implemented, then the references to ABGR_EXT in this file are invalid, and should be ignored.

Dependencies on EXT_texture3D

If EXT_texture3D is not implemented, then the references to TexImage3DEXT in this file are invalid, and should be ignored.

Dependencies on EXT_subtexture

If EXT_subtexture is not implemented, then the references to TexSubImage1DEXT, TexSubImage2DEXT, and TexSubImage3DEXT in this file are invalid, and should be ignored.

Dependencies on EXT_histogram

If EXT_histogram is not implemented, then the references to GetHistogramEXT and GetMinmaxEXT in this file are invalid, and should be ignored.

Dependencies on EXT_convolution

If EXT_convolution is not implemented, then the references to ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, and GetSeparableFilterEXT in this file are invalid, and should be ignored.

Dependencies on SGI_color_table

If SGI_color_table is not implemented, then the references to ColorTableSGI and GetColorTableSGI in this file are invalid, and should be ignored.

Dependencies on SGIS_texture4D

If SGIS_texture4D is not implemented, then the references to TexImage4DSGIS and TexSubImage4DSGIS in this file are invalid, and should be ignored.

Dependencies on EXT_cmyka

If EXT_cmyka is not implemented, then the references to CMYK_EXT and CMYKA_EXT in this file are invalid, and should be ignored.

Errors

[For the purpose of this enumeration of errors, GenericPixelFunction represents any OpenGL function that accepts or returns pixel data, using parameters <type> and <format> to define the type and format of that data. Currently these functions are DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, GetHistogramEXT, GetMinmaxEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, GetSeparableFilterEXT, ColorTableSGI, GetColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS.]

INVALID_OPERATION is generated by GenericPixelFunction if its <type> parameter is UNSIGNED_BYTE_3_3_2_EXT and its <format> parameter does not specify three components. Currently the only 3-component format is RGB.

INVALID_OPERATION is generated by GenericPixelFunction if its <type> parameter is UNSIGNED_SHORT_4_4_4_4_EXT, UNSIGNED_SHORT_5_5_5_1_EXT, UNSIGNED_INT_8_8_8_8_EXT, or UNSIGNED_INT_10_10_10_2_EXT and its <format> parameter does not specify four components. Currently the only 4-component formats are RGBA, ABGR_EXT, and CMYK_EXT.

New State

None

New Implementation Dependent State

None

Name

EXT_paletted_texture

Name Strings

GL_EXT_paletted_texture

Version

$Date: 1997/06/12 01:07:42 $ $Revision: 1.2 $

Number

78

Dependencies

GL_EXT_paletted_texture shares routines and enumerants with GL_SGI_color_table with the minor modification that EXT replaces SGI. In all other ways these calls should function in the same manner and the enumerant values should be identical. The portions of GL_SGI_color_table that are used are:

ColorTableSGI, GetColorTableSGI, GetColorTableParameterivSGI, GetColorTableParameterfvSGI.
COLOR_TABLE_FORMAT_SGI, COLOR_TABLE_WIDTH_SGI, COLOR_TABLE_RED_SIZE_SGI, COLOR_TABLE_GREEN_SIZE_SGI, COLOR_TABLE_BLUE_SIZE_SGI, COLOR_TABLE_ALPHA_SIZE_SGI, COLOR_TABLE_LUMINANCE_SIZE_SGI, COLOR_TABLE_INTENSITY_SIZE_SGI.

Portions of GL_SGI_color_table which are not used in GL_EXT_paletted_texture are:

CopyColorTableSGI, ColorTableParameterivSGI, ColorTableParameterfvSGI.
COLOR_TABLE_SGI, POST_CONVOLUTION_COLOR_TABLE_SGI, POST_COLOR_MATRIX_COLOR_TABLE_SGI, PROXY_COLOR_TABLE_SGI, PROXY_POST_CONVOLUTION_COLOR_TABLE_SGI, PROXY_POST_COLOR_MATRIX_COLOR_TABLE_SGI, COLOR_TABLE_SCALE_SGI, COLOR_TABLE_BIAS_SGI.

EXT_paletted_texture can be used in conjunction with EXT_texture3D. EXT_paletted_texture modifies TexImage3DEXT to accept paletted image data and allows TEXTURE_3D_EXT and PROXY_TEXTURE_3D_EXT to be used a targets in the color table routines. If EXT_texture3D is unsupported then references to 3D texture support in this spec are invalid and should be ignored.

Overview

EXT_paletted_texture defines new texture formats and new calls to support the use of paletted textures in OpenGL. A paletted texture is defined by giving both a palette of colors and a set of image data which is composed of indices into the palette. The paletted texture cannot function properly without both pieces of information so it increases the work required to define a texture. This is offset by the fact that the

overall amount of texture data can be reduced dramatically by factoring redundant information out of the logical view of the texture and placing it in the palette.

Paletted textures provide several advantages over full-color textures:

* As mentioned above, the amount of data required to define a texture can be greatly reduced over what would be needed for full-color specification. For example, consider a source texture that has only 256 distinct colors in a 256 by 256 pixel grid. Full-color representation requires three bytes per pixel, taking 192K of texture data. By putting the distinct colors in a palette only eight bits are required per pixel, reducing the 192K to 64K plus 768 bytes for the palette. Now add an alpha channel to the texture. The full-color representation increases by 64K while the paletted version would only increase by 256 bytes. This reduction in space required is particularly important for hardware accelerators where texture space is limited.

* Paletted textures allow easy reuse of texture data for images which require many similar but slightly different colored objects. Consider a driving simulation with heavy traffic on the road. Many of the cars will be similar but with different color schemes. If full-color textures are used a separate texture would be needed for each color scheme, while paletted textures allow the same basic index data to be reused for each car, with a different palette to change the final colors.

* Paletted textures also allow use of all the palette tricks developed for paletted displays. Simple animation can be done, along with strobing, glowing and other palette-cycling effects. All of these techniques can enhance the visual richness of a scene with very little data.

New Procedures and Functions

```
void ColorTableEXT(
 enum target,
 enum internalFormat,
 sizei width,
 enum format,
 enum type,
 const void *data);

void ColorSubTableEXT(
 enum target,
 sizei start,
 sizei count,
 enum format,
 enum type,
 const void *data);

void GetColorTableEXT(
 enum target,
 enum format,
 enum type,
 void *data);
```

```
    void GetColorTableParameterivEXT(
     enum target,
     enum pname,
     int *params);

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

New Tokens

Accepted by the internalformat parameter of TexImage1D, TexImage2D and TexImage3DEXT:

COLOR_INDEX1_EXT	0x80E2
COLOR_INDEX2_EXT	0x80E3
COLOR_INDEX4_EXT	0x80E4
COLOR_INDEX8_EXT	0x80E5
COLOR_INDEX12_EXT	0x80E6
COLOR_INDEX16_EXT	0x80E7

Accepted by the pname parameter of GetColorTableParameterivEXT and GetColorTableParameterfvEXT:

COLOR_TABLE_FORMAT_EXT	0x80D8
COLOR_TABLE_WIDTH_EXT	0x80D9
COLOR_TABLE_RED_SIZE_EXT	0x80DA
COLOR_TABLE_GREEN_SIZE_EXT	0x80DB
COLOR_TABLE_BLUE_SIZE_EXT	0x80DC
COLOR_TABLE_ALPHA_SIZE_EXT	0x80DD
COLOR_TABLE_LUMINANCE_SIZE_EXT	0x80DE
COLOR_TABLE_INTENSITY_SIZE_EXT	0x80DF

Accepted by the value parameter of GetTexLevelParameter{if}v:

TEXTURE_INDEX_SIZE_EXT	0x80ED

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Section 3.6.4, 'Pixel Transfer Operations,' subsection 'Color Index Lookup,'

Point two is modified from 'The groups will be loaded as an image into texture memory' to 'The groups will be loaded as an image into texture memory and the internalformat parameter is not one of the color index formats from table 3.8.'

Section 3.8, 'Texturing,' subsection 'Texture Image Specification' is modified as follows:

The portion of the first paragraph discussing interpretation of format, type and data is split from the portion discussing target, width and height. The target, width and height section now ends with the sentence 'Arguments width and height specify the image's width and height.'

The format, type and data section is moved under a subheader 'Direct Color Texture Formats' and begins with 'If internalformat is not one of the color index formats from table 3.8,' and continues with the existing text through the internalformat discussion.

After that section, a new section 'Paletted Texture Formats' has the text:

> If format is given as COLOR_INDEX then the image data is composed of integer values representing indices into a table of colors rather than colors themselves. If internalformat is given as one of the color index formats from table 3.8 then the texture will be stored internally as indices rather than undergoing index-to-RGBA mapping as would previously have occurred. In this case the only valid values for type are BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT and UNSIGNED_INT.
>
> The image data is unpacked from memory exactly as for a DrawPixels command with format of COLOR_INDEX for a context in color index mode. The data is then stored in an internal format derived from internalformat. In this case the only legal values of internalformat are COLOR_INDEX1_EXT, COLOR_INDEX2_EXT, COLOR_INDEX4_EXT, COLOR_INDEX8_EXT, COLOR_INDEX12_EXT and COLOR_INDEX16_EXT and the internal component resolution is picked according to the index resolution specified by internalformat. Any excess precision in the data is silently truncated to fit in the internal component precision.
>
> An application can determine whether a particular implementation supports a particular paletted format (or any paletted formats at all) by attempting to use the paletted format with a proxy target. TEXTURE_INDEX_SIZE_EXT will be zero if the implementation cannot support the texture as given.
>
> An application can determine an implementation's desired format for a particular paletted texture by making a TexImage call with COLOR_INDEX as the internalformat, in which case target must be a proxy target. After the call the application can query TEXTURE_INTERNAL_FORMAT to determine what internal format the implementation suggests for the texture image parameters. TEXTURE_INDEX_SIZE_EXT can be queried after such a call to determine the suggested index resolution numerically. The index resolution suggested by the implementation does not have to be as large as the input data precision. The resolution may also be zero if the implementation is unable to support any paletted format for the given texture image.

Table 3.8 should be augmented with a column titled 'Index bits.' All existing formats have zero index bits. The following formats are added with zeroes in all existing columns:

Name	Index bits
COLOR_INDEX1_EXT	1
COLOR_INDEX2_EXT	2
COLOR_INDEX4_EXT	4
COLOR_INDEX8_EXT	8
COLOR_INDEX12_EXT	12
COLOR_INDEX16_EXT	16

At the end of the discussion of level the following text should be added:

  All mipmapping levels share the same palette. If levels are created with different precision indices then their internal formats will not match and the texture will be inconsistent, as discussed above.

In the discussion of internalformat for CopyTexImage{12}D, at end of the sentence specifying that 1, 2, 3 and 4 are illegal there should also be a mention that paletted internalformat values are illegal.

At the end of the width, height, format, type and data section under TexSubImage there should be an additional sentence:

  If the target texture has an color index internal format then format may only be COLOR_INDEX.

At the end of the first paragraph describing TexSubImage and CopyTexSubImage the following sentence should be added:

  If the target of a CopyTexSubImage is a paletted texture image then INVALID_OPERATION is returned.

After the Alternate Image Specification Commands section, a new 'Palette Specification Commands' section should be added.

  Paletted textures require palette information to translate indices into full colors. The command

```
void ColorTableEXT(enum target, enum internalformat, sizei width,
      enum format, enum type, const void *data);
```

  is used to specify the format and size of the palette for paletted textures. target specifies which texture is to have its palette changed and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT or PROXY_TEXTURE_3D_EXT. internalformat specifies the desired format and resolution of the palette when in its internal form. internalformat can be any of the non-index values legal for TexImage internalformat although implementations are not required to support palettes of all possible formats. width controls the size of the palette and must be a power of two greater than or equal to one. format and type specify the number of components and type of the data given by data. format can be any of the formats legal for DrawPixels although implementations are not required to support all possible formats. type can be any of the types legal for DrawPixels except GL_BITMAP.

  Data is taken from memory and converted just as if each palette entry were a single pixel of a 1D texture. Pixel unpacking and transfer modes apply just as with texture data. After unpacking and conversion the data is translated into a internal format that matches the given format as closely as possible. An implementation does not, however, have a responsibility to support more than one precision for the base formats.

  If the palette's width is greater than than the range of the color indices in the texture data then some of the palettes entries

will be unused. If the palette's width is less than the range of the color indices in the texture data then the most-significant bits of the texture data are ignored and only the appropriate number of bits of the index are used when accessing the palette.

Specifying a proxy target causes the proxy texture's palette to be resized and its parameters set but no data is transferred or accessed. If an implementation cannot handle the palette data given in the call then the color table width and component resolutions are set to zero.

Portions of the current palette can be replaced with

```
void ColorSubTableEXT(enum target, sizei start, sizei count,
      enum format, enum type, const void *data);
```

target can be any of the non-proxy values legal for ColorTableEXT. start and count control which entries of the palette are changed out of the range allowed by the internal format used for the palette indices. count is silently clamped so that all modified entries all within the legal range. format and type can be any of the values legal for ColorTableEXT. The data is treated as a 1D texture just as in ColorTableEXT.

In the 'Texture State and Proxy State' section the sentence fragment beginning 'six integer values describing the resolutions...' should be changed to refer to seven integer values, with the seventh being the index resolution.

Palette data should be added in as a third category of texture state.

After the discussion of properties, the following should be added:

Next there is the texture palette. All textures have a palette, even if their internal format is not color index. A texture's palette is initially one RGBA element with all four components set to 1.0.

The sentence mentioning that proxies do not have image data or properties should be extended with 'or palettes.'

The sentence beginning 'If the texture array is too large' describing the effects of proxy failure should change to read:

If the implementation is unable to handle the texture image data the proxy width, height, border width and component resolutions are set to zero. This situation can occur when the texture array is too large or an unsupported paletted format was requested.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

In the section on GetTexImage, the sentence saying 'The components are assigned among R, G, B and A according to' should be changed to be

> If the internal format of the texture is not a color index format then the components are assigned among R, G, B, and A according to Table 6.1. Specifying COLOR_INDEX for format in this case will generate the error INVALID_ENUM. If the internal format of the texture is color index then the components are handled in one of two ways depending on the value of format. If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette and the resulting components are assigned among R, G, B, and A according to Table 6.1. If format is COLOR_INDEX then the data is treated as single components and the palette indices are returned. Components are taken starting...

Following the GetTexImage section there should be a new section:

> GetColorTableEXT is used to get the current texture palette.

```
void GetColorTableEXT(enum target, enum format, enum type, void *data);
```

> GetColorTableEXT retrieves the texture palette of the texture given by target. target can be any of the non-proxy targets valid for ColorTableEXT. format and type are interpreted just as for ColorTableEXT. All textures have a palette by default so GetColorTableEXT will always be able to return data even if the internal format of the texture is not a color index format.
>
> Palette parameters can be retrieved using

```
void GetColorTableParameterivEXT(enum target, enum pname, int *params);
void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
```

> target specifies the texture being queried and pname controls which parameter value is returned. Data is returned in the memory pointed to by params.
>
> Querying COLOR_TABLE_FORMAT_EXT returns the internal format requested by the most recent ColorTableEXT call or the default. COLOR_TABLE_WIDTH_EXT returns the width of the current palette. COLOR_TABLE_RED_SIZE_EXT, COLOR_TABLE_GREEN_SIZE_EXT, COLOR_TABLE_BLUE_SIZE_EXT and COLOR_TABLE_ALPHA_SIZE_EXT return the actual size of the components used to store the palette data internally, not the size requested when the palette was defined.

Table 6.11, "Texture Objects" should have a line appended for TEXTURE_INDEX_SIZE_EXT:

```
TEXTURE_INDEX_SIZE_EXT  n x Z+  GetTexLevelParameter 0 xD texture image i's index resolution 3.8 -
```

Revision History

Original draft, revision 0.5, December 20, 1995 (drewb) Created

Minor revisions and clarifications, revision 0.6, January 2, 1996 (drewb)
Replaced all request-for-comment blocks with final text based on implementation.

Minor revisions and clarifications, revision 0.7, Feburary 5, 1996 (drewb)
Specified the state of the palette color information when existing data is replaced by new data.

Clarified behavior of TexPalette on inconsistent textures.

Major changes due to ARB review, revision 0.8, March 1, 1996 (drewb)
Switched from using TexPaletteEXT and GetTexPaletteEXT to using SGI's ColorTableEXT routines. Added ColorSubTableEXT so equivalent functionality is available.

Allowed proxies in all targets.

Changed PALETTE?_EXT values to COLOR_INDEX?_EXT. Added support for one and two bit palettes. Removed PALETTE_INDEX_EXT in favor of COLOR_INDEX.

Decoupled palette size from texture data type. Palette size is controlled only by ColorTableEXT.

Changes due to ARB review, revision 1.0, May 23, 1997 (drewb)
Mentioned texture3D.

Defined TEXTURE_INDEX_SIZE_EXT.

Allowed implementations to return an index size of zero to indicate no support for a particular format.

Allowed usage of GL_COLOR_INDEX as a generic format in proxy queries for determining an optimal index size for a particular texture.

Disallowed CopyTexImage and CopyTexSubImage to paletted formats.

Deleted mention of index transfer operations during GetTexImage with paletted formats.

Name

EXT_point_parameters

Name Strings

GL_EXT_point_parameters

Version

$Date: 1997/08/21 21:26:36 $ $Revision: 1.6 $

Number

54

Dependencies

SGIS_multisample affects the definition of this extension.

Overview

This extension supports additional geometric characteristics of points. It can be used to render particles or tiny light sources, commonly referred as "Light points".

The raster brightness of a point is a function of the point area, point color, point transparency, and the response of the display's electron gun and phosphor. The point area and the point transparency are derived from the point size, currently provided with the <size> parameter of glPointSize.

The primary motivation is to allow the size of a point to be affected by distance attenuation. When distance attenuation has an effect, the final point size decreases as the distance of the point from the eye increases.

The secondary motivation is a mean to control the mapping from the point size to the raster point area and point transparency. This is done in order to increase the dynamic range of the raster brightness of points. In other words, the alpha component of a point may be decreased (and its transparency increased) as its area shrinks below a defined threshold.

This extension defines a derived point size to be closely related to point brightness. The brightness of a point is given by:

$$\text{dist_atten}(d) = \frac{1}{a + b * d + c * d^2}$$

$$\text{brightness}(Pe) = \text{Brightness} * \text{dist_atten}(|Pe|)$$

where 'Pe' is the point in eye coordinates, and 'Brightness' is some initial value proportional to the square of the size provided with glPointSize. Here we simplify the raster brightness to be a function of the rasterized point area and point transparency.

```
             brightness(Pe)                brightness(Pe) >= Threshold_Area
area(Pe) =
            Threshold_Area                 Otherwise

factor(Pe) = brightness(Pe)/Threshold_Area

alpha(Pe) = Alpha * factor(Pe)
```

where 'Alpha' comes with the point color (possibly modified by lighting).

'Threshold_Area' above is in area units. Thus, it is proportional to the square of the threshold provided by the programmer through this extension.

The new point size derivation method applies to all points, while the threshold applies to multisample points only.

Issues

* Does point alpha modification affect the current color ?

No.

* Do we need a special function glGetPointParameterfvEXT, or get by with glGetFloat ?

No.

* If alpha is 0, then we could toss the point before it reaches the fragment stage.

No. This can be achieved with enabling the alpha test with reference of 0 and function of LEQUAL.

* Do we need a disable for applying the threshold ?

The default threshold value is 1.0. It is applied even if the point size is constant.

If the default threshold is not overriden, the area of multisample points with provided constant size of less than 1.0, is mapped to 1.0, while the alpha component is modulated accordingly, to compensate for the larger area. For multisample points this is not a problem, as there are no relevant applications yet. As mentioned above, the threshold does not apply to alias or antialias points.

The alternative is to have a disable of threshold application, and state that threshold (if not disabled) applies to non antialias points only (that is, alias and multisample points).

The behavior without an enable/disable looks fine.

* Future extensions (to the extension)

1. GL_POINT_FADE_ALPHA_CLAMP_EXT

When the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, it might be desired to

clamp the computed alpha to a minimum value, in order to keep the point visible. In this case the formula below change:

```
factor = (derived_size/threshold)^2

                 factor                clamp <= factor
clamped_value =
                 clamp                 factor < clamp

             1.0                       derived_size >= threshold
alpha *=
             clamped_value             Otherwise
```

where clamp is defined by the GL_POINT_FADE_ALPHA_CLAMP_EXT new parameter.

New Procedures and Functions

```
void glPointParameterfEXT ( GLenum pname, GLfloat param );
void glPointParameterfvEXT ( GLenum pname, GLfloat *params );
```

New Tokens

Accepted by the <pname> parameter of glPointParameterfEXT, and the <pname> of glGet:

```
GL_POINT_SIZE_MIN_EXT
GL_POINT_SIZE_MAX_EXT
GL_POINT_FADE_THRESHOLD_SIZE_EXT
```

Accepted by the <pname> parameter of glPointParameterfvEXT, and the <pname> of glGet:

```
GL_POINT_SIZE_MIN_EXT              0x8126
GL_POINT_SIZE_MAX_EXT              0x8127
GL_POINT_FADE_THRESHOLD_SIZE_EXT   0x8128
GL_DISTANCE_ATTENUATION_EXT        0x8129
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

All parameters of the glPointParameterfEXT and glPointParameterfvEXT functions set various values applied to point rendering. The derived point size is defined to be the <size> provided with glPointSize modulated with a distance attenuation factor.

The parameters GL_POINT_SIZE_MIN_EXT and GL_POINT_SIZE_MAX_EXT simply define an upper and lower bounds respectively on the derived point size.

The above parameters affect non multisample points as well as multisample points, while the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, has no effect on non multisample points. If the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, the derived point size is used as the diameter of the rasterized point, and the alpha component is intact. Otherwise, the threshold size is

set to be the diameter of the rasterized point, while the alpha component is modulated accordingly, to compensate for the larger area.

The distance attenuation function coefficients, namely a, b, and c in:

```
                  1
dist_atten(d) = -------------------
                a + b * d + c * d^2
```

are defined by the <pname> parameter GL_DISTANCE_ATTENUATION_EXT of the function glPointParameterfvEXT. By default a = 1, b = 0, and c = 0.

Let 'size' be the point size provided with glPointSize, let 'dist' be the distance of the point from the eye, and let 'threshold' be the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE parameter of glPointParameterfEXT. The derived point size is given by:

```
derived_size = size * sqrt(dist_atten(dist))
```

Note that when default values are used, the above formula reduces to:

```
derived_size = size
```

the diameter of the rasterized point is given by:

```
            derived_size             derived_size >= threshold
diameter =
            threshold                Otherwise
```

The alpha of a point is calculated to allow the fading of points instead of shrinking them past a defined threshold size. The alpha component of the rasterized point is given by:

```
              1                              derived_size >= threshold
alpha *=
              (derived_size/threshold)^2     Otherwise
```

The threshold defined by GL_POINT_FADE_THRESHOLD_SIZE_EXT is not clamped to the minimum and maximum point sizes.

Points do not affect the current color.

This extension doesn't change the feedback or selection behavior of points.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_multisample

If SGIS_multisample is not implemented, then the references to multisample points are invalid, and should be ignored.

Errors

INVALID_ENUM is generated if PointParameterfEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, or GL_POINT_FADE_THRESHOLD_SIZE_EXT.

INVALID_ENUM is generated if PointParameterfvEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, GL_POINT_FADE_THRESHOLD_SIZE_EXT, or GL_DISTANCE_ATTENUATION_EXT

INVALID_VALUE is generated when values are out of range according to:

<pname>	valid range
GL_POINT_SIZE_MIN_EXT	>= 0
GL_POINT_SIZE_MAX_EXT	>= 0
GL_POINT_FADE_THRESHOLD_SIZE_EXT	>= 0

Issues

- should we generate INVALID_VALUE or just clamp?

New State

Get Value	Get Command	Type	Initial Value	Attribute
GL_POINT_SIZE_MIN_EXT	GetFloatv	R	0	point
GL_POINT_SIZE_MAX_EXT	GetFloatv	R	M	point
GL_POINT_FADE_THRESHOLD_SIZE_EXT	GetFloatv	R	1	point
GL_DISTANCE_ATTENUATION_EXT	GetFloatv	3xR	(1,0,0)	point

M is the largest available point size.

New Implementation Dependent State

None

Backwards Compatibility

This extension replaces SGIS_point_parameters. The procedures, tokens, and name strings now refer to EXT instead of SGIS. Enumerant values are unchanged. SGI implementations which previously provided this functionality should support both forms of the extension.

Name

EXT_rescale_normal

Name Strings

GL_EXT_rescale_normal

Version

$Date: 1997/07/02 23:38:17 $ $Revision: 1.7 $

Number

27

Dependencies

None

Overview

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

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

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

RESCALE_NORMAL_EXT	0x803A

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

Section 2.10.3

Finally, we consider how the ModelView transformation state affects normals. Normals are of interest only in eye coordinates, so the rules governing their transformation to other coordinate systems are not examined.

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

the Normal command are rescaled after transformation by the ModelView Inverse.

Normals sent to the GL may or may not have unit length. In addition, the length of the normals after transformation might be altered due to transformation by the model-view inverse matrix. If normalization is enabled, then normals specified with the Normal3 command are normalized after transformation by the model-view inverse matrix and after rescaling if rescaling is enabled. Normalization and rescaling are controlled with

```
    void Enable( enum target);
```

and

```
    void Disable( enum target);
```

with target equal to NORMALIZE or RESCALE_NORMAL. This requires two bits of state. The initial state is for normals not to be normalized or rescaled.

.
.
.

Therefore, if the modelview matrix is M, then the transformed plane equation is

```
 (n_x' n_y' n_z' q') = ((n_x n_y n_z q) * (M^-1)),
```

the rescaled normal is

```
 (n_x" n_y" n_z")  = f * (n_x' n_y' n_z'),
```

and the fully transformed normal is

$$\frac{1}{\sqrt{(n_x")^2 + (n_y")^2 + (n_z")^2}} \begin{pmatrix} n_x" \\ n_y" \\ n_z" \end{pmatrix} \qquad (2.1)$$

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

Let m_ij denote the matrix element in row i and column j of M^-1, numbering the topmost row of the matrix as row 1, and the leftmost column as column 1. Then

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

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

$$f = \frac{1}{\sqrt{(n_x')^2 + (n_y')^2 + (n_z')^2}}$$

If normalization is disabled, then the square root in equation 2.1 is replaced with 1, otherwise . . . .

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

New State

Get Value	Get Command	Type	Initial Value	Attribute
RESCALE_NORMAL_EXT	IsEnabled	B	FALSE	transform/enable

New Implementation Dependent State

None

Name

EXT_secondary_color

Name Strings

GL_EXT_secondary_color

Version

NVIDIA Date: February 22, 2000
$Date: 1999/06/21 19:57:47 $ $Revision: 1.8 $

Number

145

Dependencies

Either EXT_separate_specular_color or OpenGL 1.2 is required, to specify the "Color Sum" stage and other handling of the secondary color. This is written against the 1.2 specification (available from www.opengl.org).

Overview

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.

Issues

* Can we use the secondary alpha as an explicit fog weighting factor?

  ISVs prefer a separate interface (see GL_EXT_fog_coord). The current interface specifies only the RGB elements, leaving the option of a separate extension for SecondaryColor4() entry points open (thus the apparently useless ARRAY_SIZE state entry).

  There is an unpleasant asymmetry with Color3() - one assumes A = 1.0, the other assumes A = 0.0 - but this appears unavoidable given the 1.2 color sum specification language. Alternatively, the color sum language could be rewritten to not sum secondary A.

* What about multiple "color iterators" for use with aggrandized multitexture implementations?

  We may need this eventually, but the secondary color is well defined and a more generic interface doesn't seem justified now.

* Interleaved array formats?

  No. The multiplicative explosion of formats is too great.

* Do we want to be able to query the secondary color value? How does it interact with lighting?

The secondary color is not part of the GL state in the separate_specular_color extension that went into OpenGL 1.2. There, it can't be queried or obtained via feedback.

The secondary_color extension is slightly more general-purpose, so the secondary color is explicitly in the GL state and can be queried - but it's still somewhat limited and can't be obtained via feedback, for example.

New Procedures and Functions

```
void SecondaryColor3[bsifd ubusui]EXT(T components)
void SecondaryColor3[bsifd ubusui]vEXT(T components)
void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                      void *pointer)
```

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

COLOR_SUM_EXT	0x8458

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

CURRENT_SECONDARY_COLOR_EXT	0x8459
SECONDARY_COLOR_ARRAY_SIZE_EXT	0x845A
SECONDARY_COLOR_ARRAY_TYPE_EXT	0x845B
SECONDARY_COLOR_ARRAY_STRIDE_EXT	0x845C

Accepted by the <pname> parameter of GetPointerv:

SECONDARY_COLOR_ARRAY_POINTER_EXT	0x845D

Accepted by the <array> parameter of EnableClientState and DisableClientState:

SECONDARY_COLOR_ARRAY_EXT	0x845E

Additions to Chapter 2 of the 1.2 Draft Specification (OpenGL Operation)

These changes describe a new current state type, the secondary color, and the commands to specify it:

- (2.6, p. 12) Second paragraph changed to:

  "Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current secondary color may be used in processing each vertex."

  Third paragraph, second sentence changed to:

  "These associated colors are either based on the current color and current secondary color, or produced by lighting, depending on

whether or not lighting is enabled."

- 2.6.3, p. 19) First paragraph changed to

"The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, and texture coordinates (Vertex, Color, SecondaryColorEXT, Index, Normal, TexCoord)..."

- (2.7, p. 20) Starting with the fourth paragraph, change to:

"Finally, there are several ways to set the current color and secondary color. The GL stores a current single-valued color index as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in color index mode or RGBA mode. The mode selection is made when the GL is initialized.

The commands to set RGBA colors and secondary colors are:

```
void Color[34][bsifd ubusui](T components)
void Color[34][bsifd ubusui]v(T components)
void SecondaryColor3[bsifd ubusui]EXT(T components)
void SecondaryColor3[bsifd ubusui]vEXT(T components)
```

The color command has two major variants: Color3 and Color4. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.13.)

The secondary color command has only the three value versions. Secondary A is always set to 0.0.

Versions of the Color and SecondaryColorEXT commands that take floating-point values accept values nominally between 0.0 and 1.0...."

The last paragraph is changed to read:

"The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial RGBA secondary color is (R,G,B,A) = (0,0,0,0). The initial color index is 1."

- (2.8, p. 21) Added secondary color command for vertex arrays:

Change first paragraph to read:

"The vertex specification commands described in section 2.7 accept

data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to seven arrays: one each to store edge flags, texture coordinates, colors, secondary colors, color indices, normals, and vertices. The commands"

Add to functions listed following first paragraph:

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

Add to table 2.4 (p. 22):

Command	Sizes	Types
SecondaryColorPointerEXT	3	byte,ubyte,short,ushort, int,uint,float,double

Starting with the second paragraph on p. 23, change to add SECONDARY_COLOR_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
    void EnableClientState(enum array)
    void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY, SECONDARY_COLOR_ARRAY_EXT, INDEX_ARRAY, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, color, secondary color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
    void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, color, secondary color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, Color<size><type>v, SecondaryColor3<type>vEXT, Index<type>v, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable secondary color array for canned interleaved array formats. After the lines

```
    DisableClientState(EDGE_FLAG_ARRAY);
    DisableClientState(INDEX_ARRAY);
```

insert the line

```
    DisableClientState(SECONDARY_COLOR_ARRAY_EXT);
```

Substitute "seven" for every occurence of "six" in the final paragraph on p. 27.

- (2.12, p. 41) Add secondary color to the current rasterpos state.

  Change the last paragraph to read

  "The current raster position requires five single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA color, RGBA secondary color, and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both $(0,0,0,1)$, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is $(1,1,1,1)$, the associated RGBA secondary color is $(0,0,0,0)$, and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color and and secondary color always maintain their initial values."

- (2.13, p. 43) Change second paragraph to acknowledge two colors when lighting is disabled:

  "Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or current color (primary color) and current secondary color are used in further processing. After lighting, RGBA colors are clamped..."

- (Figure 2.8, p. 42) Change to show primary and secondary RGBA colors in both lit and unlit paths.

- (2.13.1, p. 44) Change so that the second paragraph starts:

  "Lighting may be in one of two states:

  1. Lighting Off. In this state, the current color and current secondary color are assigned to the vertex primary color and vertex secondary color, respectively.

  2. ..."

- (2.13.1, p. 48) Change the sentence following equation 2.5 (for spot_i) so that color sum is implicitly enabled when SEPARATE_SPECULAR_COLOR is set:

  "All computations are carried out in eye coordinates. When c_es = SEPARATE_SPECULAR_COLOR, it is as if color sum (see section 3.9) were enabled, regardless of the value of COLOR_SUM_EXT."

- (3.9, p. 136) Change the first paragraph to read

"After texturing, a fragment has two RGBA colors: a primary color c_pri (which texturing, if enabled, may have modified) and a secondary color c_sec.

If color sum is enabled, the components of these two colors are summed to produce a single post-texturing RGBA color c (the A component of the secondary color is always 0). The components of c are then clamped to the range [0,1]. If color sum is disabled, then c_pri is assigned to the post texturing color. Color sum is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant COLOR_SUM_EXT.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

Eight new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

SecondaryColor3bvEXT

2	8	rendering command length
2	4126	rendering command opcode
1	INT8	v[0]
1	INT8	v[1]
1	INT8	v[2]
1		unused

SecondaryColor3svEXT

2	12	rendering command length
2	4127	rendering command opcode
2	INT16	v[0]
2	INT16	v[1]
2	INT16	v[2]
2		unused

SecondaryColor3ivEXT

2	16	rendering command length
2	4128	rendering command opcode
4	INT32	v[0]
4	INT32	v[1]
4	INT32	v[2]

SecondaryColor3fvEXT

2	16	rendering command length
2	4129	rendering command opcode
4	FLOAT32	v[0]
4	FLOAT32	v[1]
4	FLOAT32	v[2]

```
SecondaryColor3dvEXT
    2       28          rendering command length
    2       4130        rendering command opcode
    8       FLOAT64     v[0]
    8       FLOAT64     v[1]
    8       FLOAT64     v[2]

SecondaryColor3ubvEXT
    2       8           rendering command length
    2       4131        rendering command opcode
    1       CARD8       v[0]
    1       CARD8       v[1]
    1       CARD8       v[2]
    1                   unused

SecondaryColor3usvEXT
    2       12          rendering command length
    2       4132        rendering command opcode
    2       CARD16      v[0]
    2       CARD16      v[1]
    2       CARD16      v[2]
    2                   unused

SecondaryColor3uivEXT
    2       16          rendering command length
    2       4133        rendering command opcode
    4       CARD32      v[0]
    4       CARD32      v[1]
    4       CARD32      v[2]
```

Errors

INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <size> is not 3.

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

INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <stride> is negative.

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_SECONDARY_COLOR_EXT	C	GetIntegerv, GetFloatv	(0,0,0,0)	Current secondary color	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SECONDARY_COLOR_ARRAY_EXT	B	IsEnabled	False	Sec. color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE_EXT	Z+	GetIntegerv	3	Sec. colors per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of sec. color components	2.8	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between sec. colors	2.8	vertex-array
SECONDARY_COLOR_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the sec. color array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
COLOR_SUM_EXT	B	IsEnabled	False	True if color sum enabled	3.9	fog/enable

Name

EXT_separate_specular_color

Name Strings

GL_EXT_separate_specular_color

Version

$Date: 1997/10/05 00:16:23 $ $Revision: 1.3 $

Number

144

Dependencies

None

Overview

This extension adds a second color to rasterization when lighting is enabled. Its purpose is to produce textured objects with specular highlights which are the color of the lights. It applies only to rgba lighting.

The two colors are computed at the vertexes. They are both clamped, flat-shaded, clipped, and converted to fixed-point just like the current rgba color (see Figure 2.8). Rasterization interpolates both colors to fragments. If texture is enabled, the first (or primary) color is the input to the texture environment; the fragment color is the sum of the second color and the color resulting from texture application. If texture is not enabled, the fragment color is the sum of the two colors.

A new control to LightModel*, LIGHT_MODEL_COLOR_CONTROL_EXT, manages the values of the two colors. It takes values: SINGLE_COLOR_EXT, a compatibility mode, and SEPARATE_SPECULAR_COLOR_EXT, the object of this extension. In single color mode, the primary color is the current final color and the secondary color is 0.0. In separate specular mode, the primary color is the sum of the ambient, diffuse, and emissive terms of final color and the secondary color is the specular term.

There is much concern that this extension may not be compatible with the future direction of OpenGL with regards to better lighting and shading models. Until those impacts are resolved, serious consideration should be given before adding to the interface specified herein (for example, allowing the user to specify a second input color).

Issues

* Where is emissive included?

  RESOLVED - Emissive is included with the ambient and diffuse

terms. Grouping emissive with specular (the "proper" thing) could be implemented with a new value for the color control.

* Should there be two colors when not lighting or with index lighting?

RESOLVED - The answer is probably yes--there should be two colors when lighting is disabled and there could be an incorporation of two colors with index lighting; but these are beyond the scope of this extension. Further, attempts to accomplish these may not be compatible with the future direction of OpenGL with respect to high quality lighting and shading models.

* What happens when texture is disabled?

RESOLVED - The extension specifies to add the two colors when texture is disabled. This is compatible with the philosophy of "if texture is disabled, this mode does not apply".

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameter of LightModel*, and also by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

LIGHT_MODEL_COLOR_CONTROL_EXT	0x81F8

Accepted by the <param> parameter of LightModel* when <pname> is LIGHT_MODEL_COLOR_CONTROL_EXT:

SINGLE_COLOR_EXT	0x81F9
SEPARATE_SPECULAR_COLOR_EXT	0x81FA

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

- (2.13, p. 40) Rework the second paragraph to acknowledge two colors:

"Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or color is used in further processing (the current color is the primary color and the secondary color is 0). After lighting, colors are clamped..."

- (Figure 2.8, p. 41) Change RGBA to primary RGBA and secondary RGB:

Ideally, there might be an RGB2 underneath RGBA (both places). Alternatively, a note in the caption could clarify that RGBA referred to the primary RGBA and a secondary RGB. (Speaking of the caption, the part about "m is the number of bits an R, G, B, or A component" could be removed as m doesn't appear in the diagram.)

- (2.13.1, p. 42) Rework the opening of this section to not imply a single color:

  In the first sentence, change "a color" to "colors". Rephrase the itemization of the two lighting states to:

  "1. Lighting Off. In this state, the current color is assigned to the vertex primary color. The vertex secondary color is 0.

   2. Lighting On. In this state, the vertex primary and secondary colors are computed from the current lighting parameters."

- (Table 2.7, p.44) Add new entry (at the bottom):

Parameter	Type	Default Value	Description
c_es	enum	SINGLE_COLOR_EXT	controls computation of colors

- (p. 45, top of page) Rephrase the first line and equation:

  "Lighting produces two colors at a vertex: a primary color c_1 and a secondary color c_2. The values of c_1 and c_2 depend on the light model color control, c_es (note: c_es should be in italics and c_1 and c_2 in bold, so this really won't be as confusing as it seems). If c_es = SINGLE_COLOR_EXT, then the equations to compute c_1 and c_2 are (note: the equation for c_1 is the current equation for c):

```
  c_1 = e_cm
      + a_cm * a_cs
      + SUM(att_i * spot_i * (a_cm * a_cli
                            + dot(n, VP_pli) * d_cm * d_cli
                            + f_i * dot(n, h_i)^s_rm * s_cm * s_cli)
  c_2 = 0
```

  If c_es = SEPARATE_SPECULAR_COLOR_EXT, then:

```
  c_1 = e_cm
      + a_cm * a_cs
      + SUM (att_i * spot_i * (a_cm * a_cli
                             + (n dot VP_pli) * d_cm * d_cli)

  c_2 = SUM(att_i * spot_i * (f_i * (n dot h_i)^s_rm * s_cm * s_cli)
```

- (p. 45, second paragraph from bottom) Clarify that A is in the primary color:

  After the sentence "The value of A produced by lighting is the alpha value associated with d_cm", add "A is always associated with the primary color c_1; c_2 has no alpha component."

- (Table 2.8, p. 48) Add a new entry (at the bottom):

Parameter	Name	Number of values
c_es	LIGHT_MODEL_COLOR_CONTROL_EXT	1

- (2.13.6, p. 51) Clarify that both primary and secondary colors are clamped:

  Replace "RGBA" in the first line of the section with "both primary and secondary".

- (2.13.7, p. 52) Clarify what happens to primary and secondary colors when flat shading--reword the first paragraph:

  "A primitive may be flatshaded, meaning that all vertices of the primitive are assigned the same color index or primary and secondary colors. These come from the vertex that spawned the primitive. For a point, these are the colors associated with the point. For a line segment, they are the colors of the second (final) vertex of the segment. For a polygon, they come from a selected vertex depending on how the polygon was generated. Table 2.9 summarizes the possibilities."

- (2.13.8, p. 52) Rework to not imply a single color:

  In the second sentence, change "If the color is" to "Those" and ", it is" to "are". In the first sentence of the next paragraph, change "the color" to "two colors".

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

- (Figure 3.1, p. 55) Add a box between texturing and fog called "color sum".

- (3.8, p. 85) In the first paragraph, second sentence, insert "primary" before RGBA. Insert after this sentence "Texturing does not affect the secondary color."

- (new section before 3.9) Insert new section titled "Color Sum":

  "At the beginning of this stage in RGBA mode, a fragment has two colors: a primary RGBA color (which texture, if enabled, may have modified) and a secondary RGB color. This stage sums the R, G, and B components of these two colors to produce a single RGBA color. If the resulting RGB values exceed 1.0, they are clamped to 1.0.

  In color index mode, a fragment only has a single color index and this stage does nothing."

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

- (5.3, p. 137) Specify that feedback returns the primary color by changing the last sentence of the large paragraph in the middle of the page to:

```
    "The colors returned are the primary colors.  These colors and the
    texture coordinates are those resulting from the clipping operations
    as described in section 2.13.8."
```

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

- (Table 6.9, p. 157) Add:

```
    Get Value - LIGHT_MODEL_COLOR_CONTROL_EXT
    Type - Z2
    Get Cmnd - GetIntegerv
    Initial Value - SINGLE_COLOR_EXT
    Description - color control
    Sec. - (whatever it ends up as)
    Attribute - lighting
```

Additions to the GLX Specification

None.

GLX Protocol

None.

Errors

None.

New State

(see changes to table 6.9)

Name

EXT_shared_texture_palette

Name Strings

GL_EXT_shared_texture_palette

Version

$Date: 1997/09/10 23:23:04 $ $Revision: 1.2 $

Number

141

Dependencies

EXT_paletted_texture is required.

Overview

EXT_shared_texture_palette defines a shared texture palette which may be used in place of the texture object palettes provided by EXT_paletted_texture. This is useful for rapidly changing a palette common to many textures, rather than having to reload the new palette for each texture. The extension acts as a switch, causing all lookups that would normally be done on the texture's palette to instead use the shared palette.

Issues

* Do we want to use a new <target> to ColorTable to specify the shared palette, or can we just infer the new target from the corresponding Enable?

* A future extension of larger scope might define a "texture palette object" and bind these objects to texture objects dynamically, rather than making palettes part of the texture object state as the current EXT_paletted_texture spec does.

* Should there be separate shared palettes for 1D, 2D, and 3D textures?

  Probably not; palette lookups have nothing to do with the dimensionality of the texture. If multiple shared palettes are needed, we should define palette objects.

* There's no proxy mechanism for checking if a shared palette can be defined with the requested parameters. Will it suffice to assume that if a texture palette can be defined, so can a shared palette with the same parameters?

* The changes to the spec are based on changes already made for EXT_paletted_texture, which means that all three documents must be referred to. This is quite difficult to read.

* The changes to section 3.8.6, defining how shared palettes are enabled and disabled, might be better placed in section 3.8.1. However, the underlying EXT_paletted_texture does not appear to modify these sections to define exactly how palette lookups are done, and it's not clear where to put the changes.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, GetDoublev, IsEnabled, Enable, Disable, ColorTableEXT, ColorSubTableEXT, GetColorTableEXT, GetColorTableParameterivEXT, and GetColorTableParameterfd EXT:

SHARED_TEXTURE_PALETTE_EXT 0x81FB

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

Section 3.8, 'Texturing,' subsection 'Texture Image Specification' is modified as follows:

In the Palette Specification Commands section, the sentence beginning 'target specifies which texture is to' should be changed to:

target specifies the texture palette or shared palette to be changed, and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT, PROXY_TEXTURE_3D_EXT, or SHARED_TEXTURE_PALETTE_EXT.

In the 'Texture State and Proxy State' section, the sentence beginning 'A texture's palette is initially...' should be changed to:

There is also a shared palette not associated with any texture, which may override a texture palette. All palettes are initially...

Section 3.8.6, 'Texture Application' is modified by appending the following:

Use of the shared texture palette is enabled or disabled using the generic Enable or Disable commands, respectively, with the symbolic constant SHARED_TEXTURE_PALETTE_EXT.

The required state is one bit indicating whether the shared palette is enabled or disabled. In the initial state, the shared palettes is disabled.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

In the section on GetTexImage, the sentence beginning 'If format is not COLOR_INDEX...' should be changed to:

> If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette, or the shared palette if one is enabled, and the resulting components are assigned among R, G, B, and A according to Table 6.1.

In the GetColorTable section, the first sentence of the second paragraph should be changed to read:

> GetColorTableEXT retrieves the texture palette or shared palette given by target.

The first sentence of the third paragraph should be changed to read:

> Palette parameters can be retrieved using

```
void GetColorTableParameterivEXT(enum target, enum pname, int *params);
void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
```

> target specifies the texture palette or shared palette being queried and pname controls which parameter value is returned.

Additions to the GLX Specification

None

New State

Get Value	Get Command	Type	Initial Value	Attribute
SHARED_TEXTURE_PALETTE_EXT	IsEnabled	B	False	texture/enable

New Implementation Dependent State

None

Name

EXT_stencil_wrap

Name Strings

GL_EXT_stencil_wrap

Version

Date: 11/15/1999 Version 1.2

Number

176

Dependencies

None

Overview

Various algorithms use the stencil buffer to "count" the number of surfaces that a ray passes through. As the ray passes into an object, the stencil buffer is incremented. As the ray passes out of an object, the stencil buffer is decremented.

GL requires that the stencil increment operation clamps to its maximum value. For algorithms that depend on the difference between the sum of the increments and the sum of the decrements, clamping causes an erroneous result.

This extension provides an enable for both maximum and minimum wrapping of stencil values. Instead, the stencil value wraps in both directions.

Two additional stencil operations are specified. These new operations are similiar to the existing INCR and DECR operations, but they wrap their result instead of saturating it. This functionality matches the new stencil operations introduced by DirectX 6.

New Procedures and Functions

None

New Tokens

Accepted by the <mode> parameter of BlendEquation:

INCR_WRAP_EXT	0x8507
DECR_WRAP_EXT	0x8508

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Section 4.1.4 "Stencil Test" (page 144), change the 3rd paragraph to read:

"... The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP_EXT, and DECR_WRAP_EXT. The correspond to keeping the current value, setting it to zero, replacing it with the reference value, incrementing it with saturation, decrementing it with saturation, bitwise inverting it, incrementing it without saturation, and decrementing it without saturation. For purposes of incrementing and decrementing, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation will clamp values at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0 and decrementing 0 results in the maximum representable value. ..."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated by StencilOp if any of its parameters are not KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP_EXT, or DECR_WRAP_EXT.

New State

(table 6.15, page 205)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
STENCIL_FAIL	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_PASS	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer

NOTE: the only change is that Z6 type changes to Z8

New Implementation Dependent State

None

Name

EXT_texture_compression_s3tc

Name Strings

GL_EXT_texture_compression_s3tc

Status

FINAL

Version

1.0, 7 July 2000

Number

198

Dependencies

OpenGL 1.1 is required.

GL_ARB_texture_compression is required.

This extension is written against the OpenGL 1.2.1 Specification.

Overview

This extension provides additional texture compression functionality specific to S3's S3TC format (called DXTC in Microsoft's DirectX API), subject to all the requirements and limitations described by the extension GL_ARB_texture_compression.

This extension supports DXT1, DXT3, and DXT5 texture compression formats. For the DXT1 image format, this specification supports an RGB-only mode and a special RGBA mode with single-bit "transparent" alpha.

IP Status

Contact S3 Incorporated (http://www.s3.com) regarding any intellectual property issues associated with implementing this extension.

WARNING: Vendors able to support S3TC texture compression in Direct3D drivers do not necessarily have the right to use the same functionality in OpenGL.

Issues

(1) Should DXT2 and DXT4 (premultiplied alpha) formats be supported?

RESOLVED: No -- insufficient interest. Supporting DXT2 and DXT4 would require some rework to the TexEnv definition (maybe add a new base internal format RGBA_PREMULTIPLIED_ALPHA) for these formats. Note that the EXT_texture_env_combine extension (which extends normal TexEnv modes) can be used to support textures with premultipled alpha.

(2) Should generic "RGB_S3TC_EXT" and "RGBA_S3TC_EXT" enums be supported or should we use only the DXT<n> enums?

RESOLVED: No. A generic RGBA_S3TC_EXT is problematic because DXT3 and DXT5 are both nominally RGBA (and DXT1 with the 1-bit alpha is

also) yet one format must be chosen up front.

(3) *Should TexSubImage support all block-aligned edits or just the minimal functionality required by the the ARB_texture_compression extension?*

RESOLVED: Allow all valid block-aligned edits.

(4) *A pre-compressed image with a DXT1 format can be used as either an RGB_S3TC_DXT1 or an RGBA_S3TC_DXT1 image. If the image has transparent texels, how are they treated in each format?*

RESOLVED: The renderer has to make sure that an RGB_S3TC_DXT1 format is decoded as RGB (where alpha is effectively one for all texels), while RGBA_S3TC_DXT1 is decoded as RGBA (where alpha is zero for all texels with "transparent" encodings). Otherwise, the formats are identical.

(5) *Is the encoding of the RGB components for DXT1 formats correct in this spec? MSDN documentation does not specify an RGB color for the "transparent" encoding. Is it really black?*

RESOLVED: Yes. The specification for the DXT1 format initially required black, but later changed that requirement to a recommendation. All vendors involved in the definition of this specification support black. In addition, specifying black has a useful behavior.

When blending multiple texels (GL_LINEAR filtering), mixing opaque and transparent samples is problematic. Defining a black color on transparent texels achieves a sensible result that works like a texture with premultiplied alpha. For example, if three opaque white and one transparent sample is being averaged, the result would be a 75% intensity gray (with an alpha of 75%). This is the same result on the color channels as would be obtained using a white color, 75% alpha, and a SRC_ALPHA blend factor.

(6) *Is the encoding of the RGB components for DXT3 and DXT5 formats correct in this spec? MSDN documentation suggests that the RGB blocks for DXT3 and DXT5 are decoded as described the the DXT1 format.*

RESOLVED: Yes -- this appears to be a bug in the MSDN documentation. The specification for the DXT2-DXT5 formats require decoding using the opaque block encoding, regardless of the relative values of "color0" and "color1".

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2DARB and the <format> parameter of CompressedTexSubImage2DARB:

COMPRESSED_RGB_S3TC_DXT1_EXT	0x83F0
COMPRESSED_RGBA_S3TC_DXT1_EXT	0x83F1
COMPRESSED_RGBA_S3TC_DXT3_EXT	0x83F2
COMPRESSED_RGBA_S3TC_DXT5_EXT	0x83F3

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Add to Table 3.16.1: Specific Compressed Internal Formats

Compressed Internal Format	Base Internal Format
COMPRESSED_RGB_S3TC_DXT1_EXT	RGB
COMPRESSED_RGBA_S3TC_DXT1_EXT	RGBA
COMPRESSED_RGBA_S3TC_DXT3_EXT	RGBA
COMPRESSED_RGBA_S3TC_DXT5_EXT	RGBA

Modify **Section 3.8.2, Alternate Image Specification**

(add to end of TexSubImage discussion, p.123 -- after edit from the ARB_texture_compression spec)

If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of the several S3TC compressed texture image formats. Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D or CopyTexSubImage2D parameters are relaxed. TexSubImage2D and CopyTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH, unless <xoffset> and <yoffset> are both zero.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT, unless <xoffset> and <yoffset> are both zero.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an S3TC compressed texture image that does not intersect the area being modifed are preserved during valid TexSubImage2D and CopyTexSubImage2D calls.

Add to Section 3.8.2, Alternate Image Specification (adding to the end of the CompressedTexImage section introduced by the ARB_texture_compression spec)

If <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the compressed texture is stored using one of several S3TC compressed texture image formats. The S3TC texture compression algorithm supports only 2D images without borders. CompressedTexImage1DARB and CompressedTexImage3DARB produce an INVALID_ENUM error if <internalformat> is an S3TC format. CompressedTexImage2DARB will produce an INVALID_OPERATION error if <border> is non-zero.

Add to Section 3.8.2, Alternate Image Specification (adding to the end of the CompressedTexSubImage section introduced by the ARB_texture_compression spec)

If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of the several S3TC compressed texture image formats. Since the S3TC texture compression algorithm supports only 2D images, CompressedTexSubImage1DARB and CompressedTexSubImage3DARB produce

an INVALID_ENUM error if <format> is an S3TC format. Since S3TC images are easily edited along 4x4 texel boundaries, the limitations on CompressedTexSubImage2D are relaxed. CompressedTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an S3TC compressed texture image that does not intersect the area being modifed are preserved during valid TexSubImage2D and CopyTexSubImage2D calls.

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

INVALID_ENUM is generated by CompressedTexImage1DARB or CompressedTexImage3DARB if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT.

INVALID_OPERATION is generated by CompressedTexImage2DARB if if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1DARB or CompressedTexSubImage3DARB if <format> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT.

INVALID_OPERATION is generated by TexSubImage2D CopyTexSubImage2D, or CompressedTexSubImage2D if INTERNAL_FORMAT is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to

TEXTURE_HEIGHT; <xoffset> or <yoffset> is not a multiple of four.

The following restrictions from the ARB_texture_compression specification do not apply to S3TC texture formats, since subimage modification is straightforward as long as the subimage is properly aligned.

DELETE: INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D,
DELETE: TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or
DELETE: CopyTexSubImage3D if the internal format of the texture image is
DELETE: compressed and <xoffset>, <yoffset>, or <zoffset> does not equal
DELETE: -b, where b is value of TEXTURE_BORDER.

DELETE: INVALID_VALUE is generated by CompressedTexSubImage1DARB,
DELETE: CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if the
DELETE: entire texture image is not being edited: if <xoffset>,
DELETE: <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is
DELETE: less than w+b, <yoffset> + <height> is less than h+b, or <zoffset>
DELETE: + <depth> is less than d+b, where b is the value of
DELETE: TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of
DELETE: TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.

See also errors in the GL_ARB_texture_compression specification.

New State

None.

Appendix

S3TC Compressed Texture Image Formats

Compressed texture images stored using the S3TC compressed image formats are represented as a collection of 4x4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4x4 block is treated as a single pixel. If an S3TC image has a width or height less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an S3TC image with a width of <w>, height of <h>, and block size of <blocksize> (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

ceil(<w>/4) * ceil(<h>/4) * blocksize.

When decoding an S3TC image, the block containing the texel at offset (<x>, <y>) begins at an offset (in bytes) relative to the beginning of the image of:

blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).

There are four distinct S3TC image formats:

COMPRESSED_RGB_S3TC_DXT1_EXT: Each 4x4 block of texels consists of 64 bits of RGB image data.

Each RGB image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

c0_lo, c0_hi, c1_lo, c1_hi, bits_0, bits_1, bits_2, bits_3

The 8 bytes of the block are decoded into three quantities:

```
    color0 = c0_lo + c0_hi * 256
    color1 = c1_lo + c1_hi * 256
    bits   = bits_0 + 256 * (bits_1 + 256 * (bits_2 + 256 * bits_3))
```

color0 and color1 are 16-bit unsigned integers that are unpacked to RGB colors RGB0 and RGB1 as though they were 16-bit packed pixels with a <format> of RGB and a type of UNSIGNED_SHORT_5_6_5.

bits is a 32-bit unsigned integer, from which a two-bit control code is extracted for a texel at location (x,y) in the block using:

```
    code(x,y) = bits[2*(4*y+x)+1..2*(4*y+x)+0]
```

where bit 31 is the most significant and bit 0 is the least significant bit.

The RGB color for a texel at location (x,y) in the block is given by:

```
    RGB0,              if color0 > color1 and code(x,y) == 0
    RGB1,              if color0 > color1 and code(x,y) == 1
    (2*RGB0+RGB1)/3,   if color0 > color1 and code(x,y) == 2
    (RGB0+2*RGB1)/3,   if color0 > color1 and code(x,y) == 3

    RGB0,              if color0 <= color1 and code(x,y) == 0
    RGB1,              if color0 <= color1 and code(x,y) == 1
    (RGB0+RGB1)/2,     if color0 <= color1 and code(x,y) == 2
    BLACK,             if color0 <= color1 and code(x,y) == 3
```

Arithmetic operations are done per component, and BLACK refers to an RGB color where red, green, and blue are all zero.

Since this image has an RGB format, there is no alpha component and the image is considered fully opaque.

COMPRESSED_RGBA_S3TC_DXT1_EXT: Each 4x4 block of texels consists of 64 bits of RGB image data and minimal alpha information. The RGB components of a texel are extracted in the same way as COMPRESSED_RGB_S3TC_DXT1_EXT.

The alpha component for a texel at location (x,y) in the block is given by:

```
    0.0,               if color0 <= color1 and code(x,y) == 3
    1.0,               otherwise
```

IMPORTANT: When encoding an RGBA image into a format using 1-bit alpha, any texels with an alpha component less than 0.5 end up with an alpha of 0.0 and any texels with an alpha component greater than or equal to 0.5 end up with an alpha of 1.0. When encoding an RGBA image into the COMPRESSED_RGBA_S3TC_DXT1_EXT format, the resulting red, green, and blue components of any texels with a final alpha of 0.0 will automatically be zero (black). If this behavior is not desired by an application, it should not use COMPRESSED_RGBA_S3TC_DXT1_EXT. This format will never be used when a generic compressed internal format (Table 3.16.2) is specified, although the nearly identical format COMPRESSED_RGB_S3TC_DXT1_EXT (above) may be.

COMPRESSED_RGBA_S3TC_DXT3_EXT: Each 4x4 block of texels consists of 64 bits of uncompressed alpha image data followed by 64 bits of RGB image data.

```
Each RGB image data block is encoded according to the
COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code
bits always use the non-transparent encodings.  In other words, they are
treated as though color0 > color1, regardless of the actual values of
color0 and color1.

Each alpha image data block is encoded as a sequence of 8 bytes, called
(in order of increasing address):

        a0, a1, a2, a3, a4, a5, a6, a7

    The 8 bytes of the block are decoded into one 64-bit integer:

        alpha = a0 + 256 * (a1 + 256 * (a2 + 256 * (a3 + 256 * (a4 +
                    256 * (a5 + 256 * (a6 + 256 * a7))))))

    alpha is a 64-bit unsigned integer, from which a four-bit alpha value
    is extracted for a texel at location (x,y) in the block using:

        alpha(x,y) = bits[4*(4*y+x)+3..4*(4*y+x)+0]

    where bit 63 is the most significant and bit 0 is the least
    significant bit.

    The alpha component for a texel at location (x,y) in the block is
    given by alpha(x,y) / 15.


COMPRESSED_RGBA_S3TC_DXT5_EXT:  Each 4x4 block of texels consists of 64
bits of compressed alpha image data followed by 64 bits of RGB image data.

Each RGB image data block is encoded according to the
COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code
bits always use the non-transparent encodings.  In other words, they are
treated as though color0 > color1, regardless of the actual values of
color0 and color1.

Each alpha image data block is encoded as a sequence of 8 bytes, called
(in order of increasing address):

    alpha0, alpha1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5

    The alpha0 and alpha1 are 8-bit unsigned bytesw converted to alpha
    components by multiplying by 1/255.

    The 6 "bits" bytes of the block are decoded into one 48-bit integer:

      bits = bits_0 + 256 * (bits_1 + 256 * (bits_2 + 256 * (bits_3 +
                      256 * (bits_4 + 256 * bits_5))))

    bits is a 48-bit unsigned integer, from which a three-bit control code
    is extracted for a texel at location (x,y) in the block using:

        code(x,y) = bits[3*(4*y+x)+1..3*(4*y+x)+0]

    where bit 47 is the most significant and bit 0 is the least
    significant bit.

    The alpha component for a texel at location (x,y) in the block is
    given by:

          alpha0,                   code(x,y) == 0
          alpha1,                   code(x,y) == 1
```

```
            (6*alpha0 + 1*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 2
            (5*alpha0 + 2*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 3
            (4*alpha0 + 3*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 4
            (3*alpha0 + 4*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 5
            (2*alpha0 + 5*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 6
            (1*alpha0 + 6*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 7

            (4*alpha0 + 1*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 2
            (3*alpha0 + 2*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 3
            (2*alpha0 + 3*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 4
            (1*alpha0 + 4*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 5
            0.0,                      alpha0 <= alpha1 and code(x,y) == 6
            1.0,                      alpha0 <= alpha1 and code(x,y) == 7
```

Revision History

1.0, 07/07/00 prbrown1: Published final version agreed to by working group members.

0.9, 06/24/00 prbrown1: Documented that block-aligned TexSubImage calls do not modify existing texels outside the modified blocks. Added caveat to allow for a (0,0)-anchored TexSubImage operation of arbitrary size.

0.7, 04/11/00 prbrown1: Added issues on DXT1, DXT3, and DXT5 encodings where the MSDN documentation doesn't match what is really done. Added enum values from the extension registry.

0.4, 03/28/00 prbrown1: Updated to reflect final version of the ARB_texture_compression extension. Allowed block-aligned TexSubImage calls.

0.3, 03/07/00 prbrown1: Resolved issues pertaining to the format of RGB blocks in the DXT3 and DXT5 formats (they don't ever use the "transparent" encoding). Fixed decoding of DXT1 blocks. Pointed out issue of "transparent" texels in DXT1 encodings having different behaviors for RGB and RGBA internal formats.

0.2, 02/23/00 prbrown1: Minor revisions; added several issues.

0.11, 02/17/00 prbrown1: Slight modification to error semantics (INVALID_ENUM instead of INVALID_OPERATION).

0.1, 02/15/00 prbrown1: Initial revisio

Name

EXT_texture3D

Name Strings

GL_EXT_texture3D

Version

$Date: 1996/04/05 19:17:05 $ $Revision: 1.22 $

Number

6

Dependencies

EXT_abgr affects the definition of this extension
EXT_texture is required

Overview

This extension defines 3-dimensional texture mapping. In order to define a 3D texture image conveniently, this extension also defines the in-memory formats for 3D images, and adds pixel storage modes to support them.

One important application of 3D textures is rendering volumes of image data.

New Procedures and Functions

```
void TexImage3DEXT(enum target,
                   int level,
                   enum internalformat,
                   sizei width,
                   sizei height,
                   sizei depth,
                   int border,
                   enum format,
                   enum type,
                   const void* pixels);
```

New Tokens

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <pname> parameter of PixelStore:

PACK_SKIP_IMAGES_EXT	0x806B
PACK_IMAGE_HEIGHT_EXT	0x806C
UNPACK_SKIP_IMAGES_EXT	0x806D
UNPACK_IMAGE_HEIGHT_EXT	0x806E

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of TexImage3DEXT, GetTexImage, GetTexLevelParameteriv, GetTexLevelParameterfv, GetTexParameteriv, and GetTexParameterfv:

TEXTURE_3D_EXT 0x806F

Accepted by the <target> parameter of TexImage3DEXT, GetTexLevelParameteriv, and GetTexLevelParameterfv:

PROXY_TEXTURE_3D_EXT 0x8070

Accepted by the <pname> parameter of GetTexLevelParameteriv and GetTexLevelParameterfv:

TEXTURE_DEPTH_EXT 0x8071

Accepted by the <pname> parameter of TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

TEXTURE_WRAP_R_EXT 0x8072

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MAX_3D_TEXTURE_SIZE_EXT 0x8073

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

The pixel storage modes are augmented to support 3D image formats in memory. Table 3.1 is replaced with the table below:

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	[0, infinity]
UNPACK_SKIP_ROWS	integer	0	[0, infinity]
UNPACK_SKIP_PIXELS	integer	0	[0, infinity]
UNPACK_ALIGNMENT	integer	4	1, 2, 4, 8
UNPACK_IMAGE_HEIGHT_EXT	integer	0	[0, infinity]
UNPACK_SKIP_IMAGES_EXT	integer	0	[0, infinity]

Table 3.1: PixelStore parameters pertaining to one or more of DrawPixels, TexImage1D, TexImage2D, and TexImage3DEXT.

When TexImage3DEXT is called, the groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a 2-dimensional image, whose size and organization are specified by the <width> and <height> parameters to TexImage3DEXT. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images in exactly the manner described in the GL Specification for

2-dimensional images. If the value of UNPACK_IMAGE_HEIGHT_EXT is not positive, then the number of rows in each 2-dimensional image is <height>; otherwise the number of rows is UNPACK_IMAGE_HEIGHT_EXT. Each 2-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a 3-dimensional image builds on the mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. If UNPACK_SKIP_IMAGES_EXT is positive, the pointer is advanced by UNPACK_SKIP_IMAGES_EXT times the number of elements in one 2-dimensional image. Then <depth> 2-dimensional images are processed, each having a subimage extracted in the manner described in the GL Specification for 2-dimensional images.

The selected groups are processed as though they were part of a 2-dimensional image. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a texel as described by Table 3.6 in the EXT_texture extension. Counting from zero, each resulting Nth texel is assigned internal integer coordinates [i,j,k], where

```
i = (N mod width) - border

j = ((N div width) mod height) - border

k = ((N div (width * height)) mod depth) - border
```

and the div operator performs integer division with truncation. Thus the last 2-dimensional image of the 3-dimensional image is indexed with the highest value of k. The dimensions of the 3-dimensional texture image are <width> x <height> x <depth>. Integer values that will represent the base-2 logarithm of these dimensions are n, m, and l, defined such that

```
width = 2**n + (2 * border)

height = 2**m + (2 * border)

depth = 2**l + (2 * border)
```

It is acceptable for an implementation to vary its allocation of internal component resolution based any TexImage3DEXT parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. In particular, allocations must be invariant -- the same allocation must be made each time a texture image is specified with the same parameter values. Provision is made for an application to determine what component resolutions are available without having to fully specify the texture (see below).

Texture Wrap Modes

The additional token value TEXTURE_WRAP_R_EXT is accepted by TexParameteri, TexParameterv, TexParameteriv, and TexParameterfv, causing table 3.7 to be replaced with the table below:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]

Table 3.7: Texture parameters and their values.

If TEXTURE_WRAP_R_EXT is set to REPEAT, then the GL ignores the integer part of R coordinates, using only the fractional part. CLAMP causes R to be clamped to the range [0, 1]. The initial state is for TEXTURE_WRAP_R_EXT to be REPEAT.

Texture Minification

Continuous coordinates s, t, u, and v are defined in figure 3.10 of the GL Specification. To discuss 3-dimensional texture mapping, coordinates r and w are defined similarly. Coordinate w is equal to -border at the "far" edge of the 3D image, understanding the image to be right-handed, with k values increasing toward the viewer. It has value depth+border at the near edge of this volume. Coordinate r has the same direction, but is normalized so that it is 0.0 and 1.0 at the "far" and "near" edges of a borderless volume. If the volume has a border, the 0.0 and 1.0 mappings of r continue to bound the core image.

The formulas for p, used to determine the level of detail, are modified by including dw/dx and dw/dy terms in the obvious ways. Equation 3.7 sums (dw/dx)**2 into the left term, and (dw/dy)**2 into the right term. Equation 3.8 has ((dw/dx * Dx + dw/dy * Dy)**2 added to the two terms under the square root. The requirements for the function f(x,y) become

1. f(x, y) is continuous and monotonically increasing in each of |du/dx|, |du/dy|, |dv/dx|, |dv/dy|, |dw/dx|, and |dw/dy|.

2. Let

```
m_u = max(|du/dx|, |du/dy|)
m_v = max(|dv/dx|, |dv/dy|)
m_w = max(|dw/dx|, |dw/dy|)
```

   Then

```
max(m_u, m_v, m_w) <= f(x, y) <= m_u + m_v + m_w
```

The i and j coordinates of the texel selected for NEAREST filtering are as defined in equations 3.9 and 3.10 of the GL Specification. Coordinate k is computed as

```
         /  floor(w),       r < 1
    k = (
         \  2**l - 1,       r = 1
```

A 2x2x2 cube of texels is selected for LINEAR filtering. The i and j coordinates of these texels are computed as defined in the GL Specification for 2-dimensional images. The k coordinates are computed as

```
          / floor(w - 1/2) mod 2**l,        TEXTURE_WRAP_R_EXT is REPEAT
    k0 = (
          \ floor(w - 1/2),                 TEXTURE_WRAP_R_EXT is CLAMP


          / (k0 + 1) mod 2**l,      TEXTURE_WRAP_R_EXT is REPEAT
    k1 = (
          \ k0 + 1,                 TEXTURE_WRAP_R_EXT is CLAMP
```

Let

```
    A = frac(u - 1/2)
    B = frac(v - 1/2)
    C = frac(w - 1/2)
```

where frac(x) denotes the fractional part of x. Let T[i,j,k] be the texel at location [i,j,k] in the texture image. Then the texture value, T, is found as

```
    T = (1-A) * (1-B) * (1-C) * T[i0,j0,k0] +
          A   * (1-B) * (1-C) * T[i1,j0,k0] +
        (1-A) *   B   * (1-C) * T[i0,j1,k0] +
          A   *   B   * (1-C) * T[i1,j1,k0] +
        (1-A) * (1-B) *   C   * T[i0,j0,k1] +
          A   * (1-B) *   C   * T[i1,j0,k1] +
        (1-A) *   B   *   C   * T[i0,j1,k1] +
          A   *   B   *   C   * T[i1,j1,k1]
```

for a 3-dimensional texture. If any of the selected T[i,j,k] in the above equation refer to a border texel with unspecified value, then the border color given by the current setting of TEXTURE_BORDER_COLOR is used instead of the unspecified value or values.

Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a mipmap. A 3-dimensional mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the texture, excluding is border, has dimensions 2**n x 2**m x 2**l, then there are exactly max(n, m, l) + 1 mipmap arrays. Each subsequent array has dimensions

```
    size(i-1) x size(j-1) x size(k-1)
```

where the dimensions of the previous array are

```
    size(i) x size(j) x size(k)
```

and

```
                /  2**x + 2*border,    x > 0
    size(x) = (
                \  1 + 2*border,       x <= 0
```

Each array in a 3-dimensional mipmap is transmitted to the GL using TexImage3DEXT; the array being set is indicated with the <level> parameter. The rules for completeness of the set of arrays are as described in the GL Specification, augmented in EXT_texture. The rules for mipmap array selection, and for filtering of the two selected arrays, are also as described in the GL Specification. Finally, the rules for texture magnification are also exactly as described in the GL Specification.

Texture Application

3-dimensional texture mapping is enabled and disabled using the generic Enable and Disable commands, with <cap> specified as TEXTURE_3D_EXT. If either or both TEXTURE_1D or TEXTURE_2D are enabled at the same time as TEXTURE_3D_EXT, the 3-dimensional texture is used.

Query support

The proxy texture PROXY_TEXTURE_3D_EXT can be used by applications to query an implementations maximum configurations just as it can be for 1-dimensional and 2-dimensional textures.

Alternate sets of partial per-level texture state are defined for the proxy texture PROXY_TEXTURE_3D_EXT. Specifically, TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH_EXT, TEXTURE_BORDER, TEXTURE_COMPONENTS, TEXTURE_RED_SIZE_EXT, TEXTURE_GREEN_SIZE_EXT, TEXTURE_BLUE_SIZE_EXT, TEXTURE_ALPHA_SIZE_EXT, TEXTURE_LUMINANCE_SIZE_EXT, and TEXTURE_INTENSITY_SIZE_EXT are maintained the the proxy texture. When TexImage3DEXT is called with <target> set to PROXY_TEXTURE_3D_EXT, these proxy state values are always respecified, even if the texture is too large to actually be used. If the texture is too large, all of these state variables are set to zero. If the texture could be accommodated by TexImage3DEXT called with <target> TEXTURE_3D_EXT, these values are set as though TEXTURE_3D_EXT were being defined. All of these state value can be queried with GetTexLevelParameteriv with <target> set to PROXY_TEXTURE_3D_EXT. Calling TexImage3DEXT with <target> PROXY_TEXTURE_3D_EXT has no effect on the actual 3-dimensional texture or its state.

There is no image associated with PROXY_TEXTURE_3D_EXT. Therefore PROXY_TEXTURE_3D_EXT cannot be used as a texture, and its image must never be queried using GetTexImage. (The error INVALID_ENUM results if this is attempted.) Likewise, there is no nonlevel-related state associated with a proxy texture, so calling GetTexParameteriv or GetTexParameterfv with <target> PROXY_TEXTURE_3D_EXT results in the

error INVALID_ENUM.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

TexImage3DEXT with a proxy target is not included in display lists, but is instead executed immediately.

Additions to Chapter 6 of the GL Specification (State and State Requests)

3-dimensional texture images are queried using GetTexImage with its <target> parameter set to TEXTURE_3D_EXT. The assignment of texel component values to the initial R, G, B, and A components of a pixel group is described in EXT_texture. Pixel transfer and pixel storage operations are applied as if the image were 2-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT_EXT and PACK_SKIP_IMAGES_EXT affect the storage of the image into memory. The correspondence of texels to memory locations is as defined for TexImage3DEXT above, substituting PACK* state for UNPACK* state in all occurrences.

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. This command contains pixel data; thus it is sent to the server either as part of a glXRender request or as part of a glXRenderLarge request:

TexImage3DEXT

2	84+n+p	rendering command length
2	4114	rendering command opcode
1	BOOL	swap_bytes
1	BOOL	lsb_first
2		unused
4	CARD32	row_length
4	CARD32	image_height
4	CARD32	image_depth
4	CARD32	skip_rows
4	CARD32	skip_images
4	CARD32	skip_volumes
4	CARD32	skip_pixels
4	CARD32	alignment
4	ENUM	target
4	INT32	level
4	ENUM	internalformat
4	INT32	width
4	INT32	height
4	INT32	depth
4	INT32	size4d
4	INT32	border
4	ENUM	format
4	ENUM	type
4	CARD32	null_image
n	LISTofBYTE	pixels
p		unused, p=pad(n)

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields above are expanded to 4 bytes each:

4	88+n+p	rendering command length
4	4114	rendering command opcode

If <width> < 0, <height> < 0, <depth> < 0, <format> is invalid or <type> is invalid, then the command is erroneous and n=0.

<pixels> is arranged as a sequence of adjacent rectangles. Each rectangle is a 2-dimensional image, whose structure is determined by the image height and the parameters <swap_bytes>, <lsb_first>, <row_length>, <skip_rows>, <skip_pixels>, <alignment>, <width>, <format>, and <type> given in the request. If <image_height> is not positive then the number of rows (i.e., the image height) is <height>; otherwise the number of rows is <image_height>.

<skip_images> allows a sub-volume of the 3-dimensional image to be selected. If <skip_images> is positive, then the pointer is advanced by <skip_images> times the number of elements in one 2-dimensional image. Then <depth> 2-dimensional images are read, each having a subimage extracted in the manner described in Appendix A of the GLX Protocol Specification.

Dependencies on EXT_abgr

If EXT_abgr is supported, the <format> parameter of TexImage3DEXT accepts ABGR_EXT. Otherwise it does not.

Dependencies on EXT_texture

EXT_texture is required. All of the <components> tokens defined by EXT_texture are accepted by the <internalformat> parameter of TexImage3DEXT, with the same semantics that are defined by EXT_texture.

The query and error extensions defined by EXT_texture are extended in this document.

Errors

INVALID_ENUM is generated if <target> is not TEXTURE_3D_EXT or PROXY_TEXTURE_3D_EXT.

INVALID_ENUM is generated if the <target> parameter to GetTexParameteriv, GetTexParameterfv or GetTexImage is PROXY_TEXTURE_3D_EXT.

INVALID_VALUE is generated if <level> is less than zero

INVALID_ENUM is generated if <internalformat> is not ALPHA, RGB, RGBA, LUMINANCE, LUMINANCE_ALPHA, or one of the tokens defined by the EXT_texture extension. (Values 1, 2, 3, and 4 are not accepted as internal formats by TexImage3DEXT).

INVALID_VALUE is generated if <width>, <height>, or <depth> is less than zero, or cannot be represented as 2**k + 2*border for some integer k.

INVALID_VALUE is generated if <border> is not 0 or 1.

INVALID_ENUM is generated if <format> is not COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA (or ABGR_EXT if EXT_abgr is supported).

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

INVALID_OPERATION is generated if TexImage3DEXT is called between execution of Begin and the corresponding execution of End.

TEXTURE_TOO_LARGE_EXT is generated if the texture as specified cannot be accommodated by the implementation. This error will not occur if none of <width>, <height>, or <depth> is greater than MAX_3D_TEXTURE_SIZE_EXT.

New State

Get Value	Get Command	Type	Initial Value	Attribute
UNPACK_SKIP_IMAGES_EXT	GetIntegerv	Z+	0	-
UNPACK_IMAGE_HEIGHT_EXT	GetIntegerv	Z+	0	-
PACK_SKIP_IMAGES_EXT	GetIntegerv	Z+	0	-
PACK_IMAGE_HEIGHT_EXT	GetIntegerv	Z+	0	-
TEXTURE_3D_EXT	IsEnabled	B	FALSE	texture/enable
TEXTURE_WRAP_R_EXT	GetTexParameteriv	1 x Z2	REPEAT	texture
TEXTURE_DEPTH_EXT	GetTexLevelParameteriv	1 x 2 x levels x Z+	0	-

(old state with new type information)

TEXTURE	GetTexImage	3 x 1 x levels x I	null	-
TEXTURE_RED_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_GREEN_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_BLUE_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_ALPHA_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_LUMINANCE_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_INTENSITY_SIZE_EXT	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_WIDTH	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_HEIGHT	GetTexLevelParameteriv	2 x 2 x levels x Z+	0	-
TEXTURE_BORDER	GetTexLevelParameteriv	3 x 2 x levels x Z+	0	-
TEXTURE_COMPONENTS (1D and 2D)	GetTexLevelParameteriv	2 x 2 x levels x Z42	1	-
TEXTURE_COMPONENTS (3D)	GetTexLevelParameteriv	1 x 2 x levels x Z38	LUMINANCE	-
TEXTURE_BORDER_COLOR	GetTexParameteriv	3 x C	0, 0, 0, 0	texture
TEXTURE_MIN_FILTER	GetTexParameteriv	3 x Z6	NEAREST_MIPMAP_LINEAR	texture
TEXTURE_MAG_FILTER	GetTexParameteriv	3 x Z2	LINEAR	texture
TEXTURE_WRAP_S	GetTexParameteriv	3 x Z2	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	2 x Z2	REPEAT	texture

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
MAX_3D_TEXTURE_SIZE_EXT	GetIntegerv	Z+	16

Name

EXT_texture_cube_map

Name Strings

GL_EXT_texture_cube_map

Forward Compatibility

This extension is superceded by the ARB_texture_cube_map extension that is officially sanctioned by the OpenGL Architectural Review Board. Enumerant values for EXT_texture_cube_map and ARB_texture_cube_map are identical. The two extensions are operationally identical; the only difference is the change of identifier from EXT to ARB.

Because the enumerants are identical for the two extensions and because there are no new entry points, an application that detects either the "GL_EXT_texture_cube_map" or "GL_ARB_texture_cube_map" extension name will operate correctly using either extension.

NVIDIA's Release 4 drivers and early versions of NVIDIA's Release 5 drivers advertised the EXT_texture_cube_map without also advertising the ARB_texture_cube_map extension because the ARB version of the extension was not then available. To ensure that your applications operate correctly with these older drivers, NVIDIA recommends that you query for either the EXT_texture_cube_map or ARB_texture_cube_map extension to determine when texture cube map functionality is available. Because the enumerants and functionality is unchanged, programs written to use ARB_texture_cube_map need only recognize EXT_texture_cube_map to operate correctly.

Name

EXT_texture_edge_clamp

Name Strings

GL_EXT_texture_edge_clamp

Version

$Date: 1997/09/22 23:04:01 $ $Revision: 1.1 $

Dependencies

SGIS_texture_filter4 affects the definition of this extension

Overview

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. CLAMP_TO_EDGE_EXT 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 CLAMP_TO_EDGE_EXT are defined but don't result in a nice clamp-to-edge color.

CLAMP_TO_EDGE_EXT is supported by 1, 2, and 3-dimensional textures only.

Issues

* Is the arithmetic for FILTER4 filters correct? Is this the right thing to do?

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R:

CLAMP_TO_EDGE_EXT	0x812F

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_WRAP_R	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS, LINEAR_CLIPMAP_LINEAR_SGIX
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS, LINEAR_LEQUAL_R_SGIS, LINEAR_GEQUAL_R_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD	float	any value
TEXTURE_MAX_LOD	float	any value
TEXTURE_BASE_LEVEL	integer	any non-negative integer
TEXTURE_MAX_LEVEL	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE
TEXTURE_CLIPMAP_OFFSET_SGIX	2 floats	any 2 values

Table 3.7: Texture parameters and their values.

CLAMP_TO_EDGE_EXT texture clamping is specified by calling TexParameteri with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D, <pname> set to TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R, and <param> set to CLAMP_TO_EDGE_EXT.

Let [min,max] be the range of a clamped texture coordinate, and let N be the size of the 1D, 2D, or 3D 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, CLAMP_TO_EDGE_EXT defines a minimum clamping value of

min = 1 / 2*N

When used with FILTER4 filters, CLAMP_TO_EDGE_EXT defines a minimum clamping value of

```
min = 3 / 2*N,          N > 2

min = 1/2               N <= 2
```

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_texture_filter4

If SGIS_texture_filter4 is not implemented, then discussions about the interaction of filter4 texture filters and the clamping function described in this file are invalid, and should be ignored.

Errors

None

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_WRAP_S	GetTexParameteriv	n x Z3	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z3	REPEAT	texture
TEXTURE_WRAP_R	GetTexParameteriv	n x Z3	REPEAT	texture

New Implementation Dependent State

None

Name

EXT_texture_env_add

Name Strings

GL_EXT_texture_env_add

Status

Shipping (version 1.6)

Version

$Date: 1999/03/22 17:28:00 $ $Revision: 1.1 $

Number

185

Dependencies

None

Overview

New texture environment function ADD is supported with the following equation:

Cv = Cf + Ct

New function may be specified by calling TexEnv with ADD token.

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnvfi when the <pname> parameter value is GL_TEXTURE_ENV_MODE

ADD

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

	Texture Environment				
Base Texture Format	REPLACE	MODULATE	BLEND	DECAL	ADD
ALPHA	...	...	...	...	Rv = Rf
	...	...	...	...	Gv = Gf
	...	...	...	...	Bv = Bf
	...	...	...	...	Av = AfAt
LUMINANCE	...	...	...	...	Rv = Rf+Lt
	...	...	...	...	Gv = Gf+Lt
	...	...	...	...	Bv = Bf+Lt
	...	...	...	...	Av = Af
LUMINANCE_ALPHA	...	...	...	...	Rv = Rf+Lt
	...	...	...	...	Gv = Gf+Lt
	...	...	...	...	Bv = Bf+Lt
	...	...	...	...	Av = AfAt
INTENSITY	...	...	...	...	Rv = Rf+It
	...	...	...	...	Gv = Gf+It
	...	...	...	...	Bv = Bf+It
	...	...	...	...	Av = Af+It
RGB	...	...	...	...	Rv = Rf+Rt
	...	...	...	...	Gv = Gf+Gt
	...	...	...	...	Bv = Bf+Bt
	...	...	...	...	Av = Af
RGBA	...	...	...	...	Rv = Rf+Rt
	...	...	...	...	Gv = Gf+Gt
	...	...	...	...	Bv = Bf+Bt
	...	...	...	...	Av = AfAt

Table 3.11: Texture functions.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

None

Errors

None

New State

None

New Implementation Dependent State

None

Name

EXT_texture_env_combine

Name Strings

GL_EXT_texture_env_combine

Version

$Date: 1999/04/02 13:54:17 $ $Revision: 1.7 $

Number

158

Dependencies

SGI_texture_color_table affects the definition of this extension
SGIX_texture_scale_bias affects the definition of this extension

Overview

New texture environment function COMBINE_EXT allows programmable texture combiner operations, including:

```
REPLACE             Arg0
MODULATE            Arg0 * Arg1
ADD                 Arg0 + Arg1
ADD_SIGNED_EXT      Arg0 + Arg1 - 0.5
INTERPOLATE_EXT     Arg0 * (Arg2) + Arg1 * (1-Arg2)
```

where Arg0, Arg1 and Arg2 are derived from

```
PRIMARY_COLOR_EXT    primary color of incoming fragment
TEXTURE              texture color of corresponding texture unit
CONSTANT_EXT         texture environment constant color
PREVIOUS_EXT         result of previous texture environment; on
                     texture unit 0, this maps to PRIMARY_COLOR_EXT
```

and Arg2 is restricted to the alpha component of the corresponding source.

In addition, the result may be scaled by 1.0, 2.0 or 4.0.

Issues

Should the explicit bias be removed in favor of an implcit bias as part of a ADD_SIGNED_EXT function?

- Yes. This pre-scale bias is a special case and will be treated as such.

Should the primary color of the incoming fragment be available to all texture environments? Currently it is only available to the texture environment of texture unit 0.

- Yes, PRIMARY_COLOR_EXT has been added as an input source.

Should textures from other texture units be allowed as sources?

- No, not in the base spec. Too many vendors have expressed concerns about the scalability of such functionality. This can be added as a subsequent extension.

All of the 1.2 modes except BLEND can be expressed in terms of this extension. Should texture color be allowed as a source for Arg2, so all of the 1.2 modes can be expressed? If so, should all color sources be allowed, to maintain orthogonality?

- No, not in the base spec. This can be added as a subsequent extension.

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

COMBINE_EXT	0x8570

Accepted by the <pname> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <target> parameter value is TEXTURE_ENV

COMBINE_RGB_EXT	0x8571
COMBINE_ALPHA_EXT	0x8572
SOURCE0_RGB_EXT	0x8580
SOURCE1_RGB_EXT	0x8581
SOURCE2_RGB_EXT	0x8582
SOURCE0_ALPHA_EXT	0x8588
SOURCE1_ALPHA_EXT	0x8589
SOURCE2_ALPHA_EXT	0x858A
OPERAND0_RGB_EXT	0x8590
OPERAND1_RGB_EXT	0x8591
OPERAND2_RGB_EXT	0x8592
OPERAND0_ALPHA_EXT	0x8598
OPERAND1_ALPHA_EXT	0x8599
OPERAND2_ALPHA_EXT	0x859A
RGB_SCALE_EXT	0x8573
ALPHA_SCALE	

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is COMBINE_RGB_EXT or COMBINE_ALPHA_EXT

REPLACE	
MODULATE	
ADD	
ADD_SIGNED_EXT	0x8574
INTERPOLATE_EXT	0x8575

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT, or SOURCE2_ALPHA_EXT

TEXTURE	
CONSTANT_EXT	0x8576
PRIMARY_COLOR_EXT	0x8577
PREVIOUS_EXT	0x8578

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_RGB_EXT or OPERAND1_RGB_EXT

SRC_COLOR
ONE_MINUS_SRC_COLOR
SRC_ALPHA
ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_ALPHA_EXT or OPERAND1_ALPHA_EXT

SRC_ALPHA
ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND2_RGB_EXT or OPERAND2_ALPHA_EXT

SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is RGB_SCALE_EXT or ALPHA_SCALE

1.0
2.0
4.0

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE_EXT, the form of the texture function depends on the values of COMBINE_RGB_EXT and COMBINE_ALPHA_EXT, according to table 3.20. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1].

COMBINE_RGB_EXT or COMBINE_ALPHA_EXT	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_EXT	Arg0 + Arg1 - 0.5
INTERPOLATE_EXT	Arg0 * (Arg2) + Arg1 * (1-Arg2)

Table 3.20: COMBINE_EXT texture functions

The arguments Arg0, Arg1 and Arg2 are determined by the values of SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. The relationship is described in tables 3.21 and 3.22.

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.21: Arguments for COMBINE_RGB_EXT functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.22: Arguments for COMBINE_ALPHA_EXT functions

The mapping of texture components to source components is summarized in Table 3.23. In the following table, At, Lt, It, Rt, Gt and Bt are the filtered texel values.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At

Table 3.23: Correspondence of texture components to source components for COMBINE_RGB_EXT and COMBINE_ALPHA_EXT arguments

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for COMBINE_RGB_EXT or COMBINE_ALPHA_EXT is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_EXT, or INTERPOLATE_EXT.

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT or SOURCE2_ALPHA_EXT is not one of TEXTURE, CONSTANT_EXT, PRIMARY_COLOR_EXT or PREVIOUS_EXT.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_EXT or OPERAND1_RGB_EXT is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_EXT or OPERAND1_ALPHA_EXT is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND2_RGB_EXT or OPERAND2_ALPHA_EXT is not SRC_ALPHA.

INVALID_VALUE is generated if <params> value for RGB_SCALE_EXT or ALPHA_SCALE is not one of 1.0, 2.0, or 4.0.

Dependencies on SGI_texture_color_table

If SGI_texture_color_table is implemented, the expanded Rt, Gt, Bt, and At values are used directly instead of the expansion described by Table 3.23.

Dependencies on SGIX_texture_scale_bias

If SGIX_texture_scale_bias is implemented, the expanded Rt, Gt, Bt, and At values are used directly instead of the expansion described by Table 3.23.

New State

Get Value	Get Command	Type	Initial Value	Attribute
COMBINE_RGB_EXT	GetTexEnviv	n x Z4	MODULATE	texture
COMBINE_ALPHA_EXT	GetTexEnviv	n x Z4	MODULATE	texture
SOURCE0_RGB_EXT	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_RGB_EXT	GetTexEnviv	n x Z3	PREVIOUS_EXT	texture
SOURCE2_RGB_EXT	GetTexEnviv	n x Z3	CONSTANT_EXT	texture
SOURCE0_ALPHA_EXT	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_ALPHA_EXT	GetTexEnviv	n x Z3	PREVIOUS_EXT	texture
SOURCE2_ALPHA_EXT	GetTexEnviv	n x Z3	CONSTANT_EXT	texture
OPERAND0_RGB_EXT	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND1_RGB_EXT	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND2_RGB_EXT	GetTexEnviv	n x Z1	SRC_ALPHA	texture
OPERAND0_ALPHA_EXT	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND1_ALPHA_EXT	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND2_ALPHA_EXT	GetTexEnviv	n x Z1	SRC_ALPHA	texture
RGB_SCALE_EXT	GetTexEnvfv	n x R3	1.0	texture
ALPHA_SCALE	GetTexEnvfv	n x R3	1.0	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

Because of a hardware limitation, TNT, TNT2, GeForce, and Quadro treat "scale by 4.0" with the COMBINE_RGB_EXT or COMBINE_ALPHA_EXT mode of ADD_SIGNED_EXT as "scale by 2.0".

Name

EXT_texture_env_dot3

Name Strings

EXT_texture_env_dot3

Notice

Copyright ATI Technologies, 2000.

IP Status

None

Version

$Date: 2000/09/28 13:54:17 $ $Revision: 1.2 $

Number

None.

Dependencies

EXT_texture_env_combine is required and is modified by this extension
ARB_multitexture affects the definition of this extension

Overview

Adds new operation to the texture combiner operations.

```
DOT3_RGB_EXT                    Arg0 <dotprod> Arg1
DOT3_RGBA_EXT                   Arg0 <dotprod> Arg1
```

where Arg0, Arg1 are derived from

```
PRIMARY_COLOR_EXT       primary color of incoming fragment
TEXTURE                 texture color of corresponding texture unit
CONSTANT_EXT            texture environment constant color
PREVIOUS_EXT            result of previous texture environment; on
                        texture unit 0, this maps to PRIMARY_COLOR_EXT
```

This operaion can only be performed if SOURCE0_RGB_EXT,
SOURCE1_RGB_EXT are defined.

Issues

None

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is COMBINE_RGB_EXT

DOT3_RGB_EXT	0x8740
DOT3_RGBA_EXT	0x8741

Additions to Chapter 2 of the OpenGL 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.2 Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE_EXT, the form of the texture function depends on the values of COMBINE_RGB_EXT and COMBINE_ALPHA_EXT, according to table 3.20. The RGB and ALPHA results of the texture function are not multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1].

COMBINE_RGB_EXT	Texture Function
DOT3_RGB_EXT	4*((Arg0_r - 0.5)*(Arg1_r - 0.5) + (Arg0_g - 0.5)*(Arg1_g - 0.5) + (Arg0_b - 0.5)*(Arg1_b - 0.5)) This value is placed into all three r,g,b components of the output.
DOT3_RGBA_EXT	4*((Arg0_r - 0.5)*(Arg1_r - 0.5) + (Arg0_g - 0.5)*(Arg1_g - 0.5) + (Arg0_b - 0.5)*(Arg1_b - 0.5)) This value is placed into all four r,g,b,a components of the output.

Table 3.20: COMBINE_EXT texture functions

Additions to Chapter 4 of the OpenGL 1.2 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.2 Specification (State and State Requests)

None

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

None

Errors

Modifications to EXT_texture_env_combine

Dependencies on ARB_multitexture

New State

None

New Implementation Dependent State

None

Revision History

None

Name

EXT_texture_filter_anisotropic

Name Strings

GL_EXT_texture_filter_anisotropic

Notice

Copyright NVIDIA Corporation, 1999.

Version

August 24, 1999

Number

187

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

Texture mapping using OpenGL's existing mipmap texture filtering modes assumes that the projection of the pixel filter footprint into texture space is a square (ie, isotropic). In practice however, the footprint may be long and narrow (ie, anisotropic). Consequently, mipmap filtering severely blurs images on surfaces angled obliquely away from the viewer.

Several approaches exist for improving texture sampling by accounting for the anisotropic nature of the pixel filter footprint into texture space. This extension provides a general mechanism for supporting anisotropic texturing filtering schemes without specifying a particular formulation of anisotropic filtering.

The extension permits the OpenGL application to specify on a per-texture object basis the maximum degree of anisotropy to account for in texture filtering.

Increasing a texture object's maximum degree of anisotropy may improve texture filtering but may also significantly reduce the implementation's texture filtering rate. Implementations are free to clamp the specified degree of anisotropy to the implementation's maximum supported degree of anisotropy.

A texture's maximum degree of anisotropy is specified independent from the texture's minification and magnification filter (as opposed to being supported as an entirely new filtering mode). Implementations are free to use the specified minification and magnification filter to select a particular anisotropic texture filtering scheme. For example, a NEAREST filter with a maximum degree of anisotropy of two could be treated as a 2-tap filter that

accounts for the direction of anisotropy. Implementations are also permitted to ignore the minification or magnification filter and implement the highest quality of anisotropic filtering possible.

Applications seeking the highest quality anisotropic filtering available are advised to request a LINEAR_MIPMAP_LINEAR minification filter, a LINEAR magnification filter, and a large maximum degree of anisotropy.

Issues

Should there be a particular anisotropic texture filtering minification and magnification mode?

RESOLUTION: NO. The maximum degree of anisotropy should control when anisotropic texturing is used. Making this orthogonal to the minification and magnification filtering modes allows these settings to influence the anisotropic scheme used. Yes, such an anisotropic filtering scheme exists in hardware.

What should the minimum value for MAX_TEXTURE_MAX_ANISTROPY_EXT be?

RESOLUTION: 2.0. To support this extension, at least 2 to 1 anisotropy should be supported.

Should an implementation-defined limit for the maximum maximum degree of anisotropy be "get-able"?

RESOLUTION: YES. But you should not assume that a high maximum maximum degree of anisotropy implies anything about texture filtering performance or quality.

Should anything particular be said about anisotropic 3D texture filtering?

Not sure. Does the implementation example shown in the spec for 2D anisotropic texture filtering readily extend to 3D anisotropic texture filtering?

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameterfv, TexParameteri, and TexParameteriv:

TEXTURE_MAX_ANISOTROPY_EXT	0x84FE

Accepted by the <pname> parameters of GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

MAX_TEXTURE_MAX_ANISOTROPY_EXT	0x84FF

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Sections 3.8.3 "Texture Parameters"

Add the following entry to the end of Table 3.17:

Name	Type	Legal Values
TEXTURE_MAX_ANISOTROPY_EXT	float	greater or equal to 1.0

-- Sections 3.8.5 "Texture Minification" and 3.8.6 "Texture Magnification"

After the first paragraph in Section 3.8.5:

"When the texture's value of TEXTURE_MAX_ANISOTROPY_EXT is equal to 1.0, the GL uses an isotropic texture filtering approach as described in this section and Section 3.8.6. However, when the texture's value of TEXTURE_MAX_ANISOTROPY_EXT is greater than 1.0, the GL implementation should use a texture filtering scheme that accounts for a degree of anisotropy up to the smaller of the value of TEXTURE_MAX_ANISTROPY_EXT or the implementation-defined value of MAX_TEXTURE_MAX_ANISTROPY_EXT.

The particular scheme for anisotropic texture filtering is implementation dependent. Additionally, implementations are free to consider the current texture minification and magnification modes to control the specifics of the anisotropic filtering scheme used.

The anisotropic texture filtering scheme may only access mipmap levels if the minification filter is one that requires mipmaps. Additionally, when a minification filter is specified, the anisotropic texture filtering scheme may only access texture mipmap levels between the texture's values for TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL, inclusive. Implementations are also recommended to respect the values of TEXTURE_MAX_LOD and TEXTURE_MIN_LOD to whatever extent the particular anisotropic texture filtering scheme permits this."

The following describes one particular approach to implementing anisotropic texture filtering for the 2D texturing case:

"Anisotropic texture filtering substantially changes Section 3.8.5. Previously a single scale factor P was determined based on the pixel's projection into texture space. Now two scale factors, Px and Py, are computed.

```
  Px = sqrt(dudx^2 + dvdx^2)
  Py = sqrt(dudy^2 + dvdy^2)

  Pmax = max(Px,Py)
  Pmin = min(Px,Py)

  N = min(ceil(Pmax/Pmin),maxAniso);
  Lamda' = log2(Pmax/N)
```

where maxAniso is the smaller of the texture's value of TEXTURE_MAX_ANISOTROPY_EXT or the implementation-defined value of MAX_TEXTURE_MAX_ANISOTROPY_EXT.

It is acceptable for implementation to round 'N' up to the nearest supported sampling rate. For example an implementation may only support power-of-two sampling rates.

It is also acceptable for an implementation to approximate the ideal functions Px and Py with functions Fx and Fy subject to the following conditions:

1. Fx is continuous and monotonically increasing in |du/dx| and |dv/dx|.
   Fy is continuous and monotonically increasing in |du/dy| and |dv/dy|.

2. max(|du/dx|,|dv/dx|} <= Fx <= |du/dx| + |dv/dx|.
   max(|du/dy|,|dv/dy|} <= Fy <= |du/dy| + |dv/dy|.

Instead of a single sample, Tau, at (u,v,Lamda), 'N' locations in the mipmap at LOD Lamda, are sampled within the texture footprint of the pixel. This sum TauAniso is defined using the single sample Tau. When the texture's value of TEXTURE_MAX_ANISOTROPHY_EXT is greater than 1.0, use TauAniso instead of Tau to determine the fragment's texture value.

$$\text{TauAniso} = 1/N \sum_{i=1}^{i=N} \text{Tau}(u(x - 1/2 + i/(N+1), y), v(x - 1/2 + i/(N+1), y)), \quad Px > Py$$

$$\text{TauAniso} = 1/N \sum_{i=1}^{i=N} \text{Tau}(u(x, y - 1/2 + i/(N+1)), v(x, y - 1/2 + i/(N+1))), \quad Py >= Px$$

It is acceptable to approximate the u and v functions with equally spaced samples in texture space at LOD Lamda:

```
                i=N
                ---
TauAniso = 1/N  \ Tau(u(x,y)+dudx(i/(N+1)-1/2), v(x,y)+dvdx(i/(N+1)-1/2)), Px > Py
                /
                ---
                i=1

                i=N
                ---
TauAniso = 1/N  \ Tau(u(x,y)+dudy(i/(N+1)-1/2), v(x,y)+dvdy(i/(N+1)-1/2)), Py >= Px
                /
                ---
                i=1
```

"

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_VALUE is generated when TexParameter is called with <pname> of TEXTURE_MAX_ANISOTROPY_EXT and a <param> value or value of what <params> points to less than 1.0.

New State

(table 6.13, p203) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_MAX_ANISOTROPY_EXT	R	GetTexParameterfv	1.0	Maximum degree of anisotropy	3.8.5	texture

New Implementation State

(table 6.25, p215) add the entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TEXTURE_MAX_ANISOTROPY_EXT	R	GetFloatv	2.0	Limit of maximum degree of anisotropy	3.8.5	-

Name

EXT_texture_lod_bias

Name Strings

GL_EXT_texture_lod_bias

Notice

Copyright NVIDIA Corporation, 1999, 2000.

Version

NVIDIA Date: May 23, 2000
$Id: //sw/main/docs/OpenGL/specs/GL_EXT_texture_lod_bias.txt#11 $

Number

186

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Affects ARB_multitexture.

Overview

OpenGL computes a texture level-of-detail parameter, called lambda in the GL specification, that determines which mipmap levels and their relative mipmap weights for use in mipmapped texture filtering.

This extension provides a means to bias the lambda computation by a constant (signed) value. This bias can provide a way to blur or pseudo-sharpen OpenGL's standard texture filtering.

This blurring or pseudo-sharpening may be useful for special effects (such as depth-of-field effects) or image processing techniques (where the mipmap levels act as pre-downsampled image versions). On some implementations, increasing the texture lod bias may improve texture filtering performance (at the cost of texture bluriness).

The extension mimics functionality found in Direct3D.

Issues

Should the texture LOD bias be settable per-texture object or per-texture stage?

RESOLUTION: Per-texture stage. This matches the Direct3D semantics for texture lod bias. Note that this differs from the semantics of SGI's SGIX_texture_lod_bias extension that has the biases per-texture object.

This also allows the same texture object to be used by two different texture units for different blurring. This is useful for

extrapolating detail between various levels of detail in a mipmapped texture.

For example, you can extrapolate texture detail with ARB_multitexture and EXT_texture_env_combine by computing

(B0 - B2) * 2 + B2

where B0 is a non-biased texture (normal sharpness) and B2 is the same texture but bias by 2 levels-of-detail (fairly blurry). This has the effect of increasing the high-frequency information in the texture. There are immediate Earth Sciences and medical imaging applications for this technique.

Per-texture stage control of the LOD bias is also useful for allowing an application to control overall texture bluriness. This can be used in games to simulate disorientation (note that only textures will blur, not edges). It can also be used to globally control texturing performance. An application may be able to sustain a constant frame rate by avoiding texture fetch stalls by using slightly blurrier textures.

How does EXT_texture_lod_bias differ from SGIX_texture_lod bias?

EXT_texture_lod_bias adds a bias to lambda. The SGIX_texture_lod_bias extension changes the computation of rho (the log2 of which is lambda). The SGIX extension provides separate biases in each texture dimension. The EXT extension does not provide an "directionality" in the LOD control.

Does the texture lod bias occur before or after the TEXTURE_MAX_LOD and TEXTURE_MIN_LOD clamping?

RESOLUTION: BEFORE. This allows the texture lod bias to still be clamped within the max/min lod range.

Does anything special have to be said to keep the biased lambda value from being less than zero or greater than the maximum number of mipmap levels?

RESOLUTION: NO. The existing clamping in the specification handles these situations.

The texture lod bias is specified to be a float. In practice, what sort of range is assumed for the texture lod bias?

RESOLUTION: The MAX_TEXTURE_LOD_BIAS_EXT implementation constant advertises the maximum absolute value of the supported texture lod bias. The value is recommended to be at least the maximum mipmap level supported by the implementation.

The texture lod bias is specified to be a float. In practice, what sort of precision is assumed for the texture lod bias?

RESOLUTION; This is implementation dependent. Presumably, hardware would implement the texture lod bias as a fractional bias

but the exact fractional precision supported is implementation dependent. At least 4 fractional bits is recommended.

New Procedures and Functions

None

New Tokens

Accepted by the <target> parameters of GetTexEnvfv, GetTexEnviv, TexEnvi, TexEnvf, Texenviv, and TexEnvfv:

TEXTURE_FILTER_CONTROL_EXT 0x8500

When the <target> parameter of GetTexEnvfv, GetTexEnviv, TexEnvi, TexEnvf, TexEnviv, and TexEnvfv is TEXTURE_FILTER_CONTROL_EXT, then the value of <pname> may be:

TEXTURE_LOD_BIAS_EXT 0x8501

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MAX_TEXTURE_LOD_BIAS_EXT 0x84FD

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8.5 "Texture Minification"

Change the first formula under "Scale Factor and Level of Detail" to read:

"The choice is governed by a scale factor p(x,y), the level of detail parameter lambda(x,y), defined as

lambda'(x,y) = log2[p(x,y)] + lodBias

where lodBias is the texture unit's (signed) texture lod bias parameter (as described in Section 3.8.9) clamped between the positive and negative values of the implementation defined constant MAX_TEXTURE_LOD_BIAS_EXT."

-- Section 3.8.9 "Texture Environments and Texture Functions"

Change the first paragraph to read:

"The command

```
void TexEnv{if}(enum target, enum pname, T param);
void TexEnv{if}v(enum target, enum pname, T params);
```

sets parameters of the texture environment that specifies how texture values are interepreted when texturing a fragment or sets per-texture unit texture filtering parameters. The possible target parameters are TEXTURE_ENV or TEXTURE_FILTER_CONTROL_EXT. ... When target is

TEXTURE_ENV, the possible environment parameters are TEXTURE_ENV_MODE and TEXTURE_ENV_COLOR. ... When target is TEXTURE_FILTER_CONTROL_EXT, the only possible texture filter parameter is TEXTURE_LOD_BIAS_EXT. TEXTURE_LOD_BIAS_EXT is set to a signed floating point value that is used to bias the level of detail parameter, lambda, as described in Section 3.8.5."

Add a final paragraph at the end of the section:

"The state required for the per-texture unit filtering parameters consists of one floating-point value."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Texture Environments and Texture Functions"

Change the third sentence of the third paragraph to read:

"The env argument to GetTexEnv must be either TEXTURE_ENV or TEXTURE_FILTER_CONTROL_EXT."

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexEnv is called with a <pname> of TEXTURE_FILTER_PARAMETER_EXT and the value of <param> or what is pointed to by <params> is not TEXTURE_LOD_BIAS_EXT.

New State

(table 6.14, p204) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_LOD_BIAS_EXT	R	GetTexEnvfv	0.0	Biases texture level of detail	3.8.9	texture

(When ARB_multitexture is supported, the TEXTURE_LOD_BIAS_EXT state is per-texture unit.)

New Implementation State

(table 6.24, p214) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TEXTURE_LOD_BIAS_EXT	R+	GetFloatv	4.0	Maximum absolute texture lod bias	3.8.9	-

Revision History

6/2/00 - add spec language to allow GetTexEnv to accept TEXTURE_FILTER_CONTROL_EXT.

Name

EXT_texture_object

Name Strings

GL_EXT_texture_object

Version

$Date: 1995/10/03 05:39:56 $ $Revision: 1.27 $

Number

20

Dependencies

EXT_texture3D affects the definition of this extension

Overview

This extension introduces named texture objects. The only way to name a texture in GL 1.0 is by defining it as a single display list. Because display lists cannot be edited, these objects are static. Yet it is important to be able to change the images and parameters of a texture.

Issues

* Should the dimensions of a texture object be static once they are changed from zero? This might simplify the management of texture memory. What about other properties of a texture object?

No.

Reasoning

* Previous proposals overloaded the <target> parameter of many Tex commands with texture object names, as well as the original enumerated values. This proposal eliminated such overloading, choosing instead to require an application to bind a texture object, and then operate on it through the binding reference. If this constraint ultimately proves to be unacceptable, we can always extend the extension with additional binding points for editing and querying only, but if we expect to do this, we might choose to bite the bullet and overload the <target> parameters now.

* Commands to directly set the priority of a texture object and to query the resident status of a texture object are included. I feel that binding a texture object would be an unacceptable burden for these management operations. These commands also allow queries and operations on lists of texture objects, which should improve efficiency.

* GenTexturesEXT does not return a success/failure boolean because it should never fail in practice.

New Procedures and Functions

```
    void GenTexturesEXT(sizei n,
                uint* textures);

    void DeleteTexturesEXT(sizei n,
                   const uint* textures);

    void BindTextureEXT(enum target,
                uint texture);

    void PrioritizeTexturesEXT(sizei n,
                       const uint* textures,
                       const clampf* priorities);

    boolean AreTexturesResidentEXT(sizei n,
                         const uint* textures,
                         boolean* residences);

    boolean IsTextureEXT(uint texture);
```

New Tokens

Accepted by the <pname> parameters of TexParameteri, TexParameterf, TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

```
TEXTURE_PRIORITY_EXT          0x8066
```

Accepted by the <pname> parameters of GetTexParameteriv and GetTexParameterfv:

```
TEXTURE_RESIDENT_EXT          0x8067
```

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
TEXTURE_1D_BINDING_EXT        0x8068
TEXTURE_2D_BINDING_EXT        0x8069
TEXTURE_3D_BINDING_EXT        0x806A
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

Add the following discussion to section 3.8 (Texturing). In addition to the default textures TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT, it is possible to create named 1, 2, and 3-dimensional texture objects. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT. This binding is accomplished by calling BindTextureEXT with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, and <texture> set to the name of the new texture object. When a texture object is bound to a target, the previous binding for

that target is automatically broken.

When a texture object is first bound it takes the dimensionality of its target. Thus, a texture object first bound to TEXTURE_1D is 1-dimensional; a texture object first bound to TEXTURE_2D is 2-dimensional, and a texture object first bound to TEXTURE_3D_EXT is 3-dimensional. The state of a 1-dimensional texture object immediately after it is first bound is equivalent to the state of the default TEXTURE_1D at GL initialization. Likewise, the state of a 2-dimensional or 3-dimensional texture object immediately after it is first bound is equivalent to the state of the default TEXTURE_2D or TEXTURE_3D_EXT at GL initialization. Subsequent bindings of a texture object have no effect on its state. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object to a target of different dimensionality.

While a texture object is bound, GL operations on the target to which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object. If texture mapping of the dimensionality of the target to which a texture object is bound is active, the bound texture object is used.

By default when an OpenGL context is created, TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT have 1, 2, and 3-dimensional textures associated with them. In order that access to these default textures not be lost, this extension treats them as though their names were all zero. Thus the default 1-dimensional texture is operated on, queried, and applied as TEXTURE_1D while zero is bound to TEXTURE_1D. Likewise, the default 2-dimensional texture is operated on, queried, and applied as TEXTURE_2D while zero is bound to TEXTURE_2D, and the default 3-dimensional texture is operated on, queried, and applied as TEXTURE_3D_EXT while zero is bound to TEXTURE_3D_EXT.

Texture objects are deleted by calling DeleteTexturesEXT with <textures> pointing to a list of <n> names of texture object to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is freed. If a texture object that is currently bound is deleted, the binding reverts to zero. DeleteTexturesEXT ignores names that do not correspond to textures objects, including zero.

GenTexturesEXT returns <n> texture object names in <textures>. These names are chosen in an unspecified manner, the only condition being that only names that were not in use immediately prior to the call to GenTexturesEXT are considered. Names returned by GenTexturesEXT are marked as used (so that they are not returned by subsequent calls to GenTexturesEXT), but they are associated with a texture object only after they are first bound (just as if the name were unused).

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently being treated as a part of the working set is said to be resident. AreTexturesResidentEXT returns TRUE if all of the <n> texture objects named in <textures> are resident, FALSE otherwise. If FALSE is returned, the residence of each texture object is returned in <residences>. Otherwise the contents of the <residences> array are not changed. If any of the names in <textures> is not the name of a texture object, FALSE is returned, the

error INVALID_VALUE is generated, and the contents of <residences> are indeterminate. The resident status of a single bound texture object can also be queried by calling GetTexParameteriv or GetTexParameterfv with <target> set to the target to which the texture object is bound, and <pname> set to TEXTURE_RESIDENT_EXT. This is the only way that the resident status of a default texture can be queried.

Applications guide the OpenGL implementation in determining which texture objects should be resident by specifying a priority for each texture object. PrioritizeTexturesEXT sets the priorities of the <n> texture objects in <textures> to the values in <priorities>. Each priority value is clamped to the range [0.0, 1.0] before it is assigned. Zero indicates the lowest priority, and hence the least likelihood of being resident. One indicates the highest priority, and hence the greatest likelihood of being resident. The priority of a single bound texture object can also be changed by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to the target to which the texture object is bound, <pname> set to TEXTURE_PRIORITY_EXT, and <param> or <params> specifying the new priority value (which is clamped to [0.0,1.0] before being assigned). This is the only way that the priority of a default texture can be specified. (PrioritizeTexturesEXT silently ignores attempts to prioritize nontextures, and texture zero.)

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

BindTextureEXT and PrioritizeTexturesEXT are included in display lists. All other commands defined by this extension are not included in display lists.

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

IsTextureEXT returns TRUE if <texture> is the name of a valid texture object. If <texture> is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, IsTextureEXT returns FALSE.

Because the query values of TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT are already defined as booleans indicating whether these textures are enabled or disabled, another mechanism is required to query the binding associated with each of these texture targets. The name of the texture object currently bound to TEXTURE_1D is returned in <params> when GetIntegerv is called with <pname> set to TEXTURE_1D_BINDING_EXT. If no texture object is currently bound to TEXTURE_1D, zero is returned. Likewise, the name of the texture object bound to TEXTURE_2D or TEXTURE_3D_EXT is returned in <params> when GetIntegerv is called with <pname> set to TEXTURE_2D_BINDING_EXT or TEXTURE_3D_BINDING_EXT. If no texture object is currently bound to TEXTURE_2D or to TEXTURE_3D_EXT, zero is returned.

A texture object comprises the image arrays, priority, border color, filter modes, and wrap modes that are associated with that object. More

explicitly, the state list

```
TEXTURE,
TEXTURE_PRIORITY_EXT
TEXTURE_RED_SIZE,
TEXTURE_GREEN_SIZE,
TEXTURE_BLUE_SIZE,
TEXTURE_ALPHA_SIZE,
TEXTURE_LUMINANCE_SIZE,
TEXTURE_INTENSITY_SIZE,
TEXTURE_WIDTH,
TEXTURE_HEIGHT,
TEXTURE_DEPTH_EXT,
TEXTURE_BORDER,
TEXTURE_COMPONENTS,
TEXTURE_BORDER_COLOR,
TEXTURE_MIN_FILTER,
TEXTURE_MAG_FILTER,
TEXTURE_WRAP_S,
TEXTURE_WRAP_T,
TEXTURE_WRAP_R_EXT
```

composes a single texture object.

When PushAttrib is called with TEXTURE_BIT enabled, the priorities, border colors, filter modes, and wrap modes of the currently bound texture objects are pushed, as well as the current texture bindings and enables. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture objects have their priorities, border colors, filter modes, and wrap modes restored to their pushed values.

Additions to the GLX Specification

Texture objects are shared between GLX rendering contexts if and only if the rendering contexts share display lists. No change is made to the GLX API.

GLX Protocol

Six new GL commands are added.

The following rendering command is sent to the server as part of a glXRender request:

```
    BindTextureEXT
        2           12              rendering command length
        2           4117            rendering command opcode
        4           ENUM            target
        4           CARD32          texture
```

The following rendering command can be sent to the server as part of a glXRender request or as part of a glXRenderLarge request:

```
PrioritizeTexturesEXT
        2           8+(n*8)         rendering command length
        2           4118            rendering command opcode
        4           INT32           n
        n*4         LISTofCARD32    textures
        n*4         LISTofFLOAT32   priorities

    If the command is encoded in a glXRenderLarge request, the
    command opcode and command length fields above are expanded to
    4 bytes each:

        4           12+(n*8)        rendering command length
        4           4118            rendering command opcode
```

The remaining commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using either the glXVendorPrivate request or the glXVendorPrivateWithReply request:

```
DeleteTexturesEXT
    1       CARD8            opcode (X assigned)
    1       16               GLX opcode (glXVendorPrivate)
    2       4+n              request length
    4       12               vendor specific opcode
    4       GLX_CONTEXT_TAG  context tag
    4       INT32            n
    n*4     CARD32           textures

   GenTexturesEXT
    1       CARD8            opcode (X assigned)
    1       17               GLX opcode (glXVendorPrivateWithReply)
    2       4                request length
    4       13               vendor specific opcode
    4       GLX_CONTEXT_TAG  context tag
    4       INT32       n
  =>
     1       1               reply
     1                       unused
     2       CARD16           sequence number
     4       n                reply length
     24                       unused
     4*n     LISTofCARD32     textures
```

```
    AreTexturesResidentEXT
     1       CARD8             opcode (X assigned)
     1       17                GLX opcode (glXVendorPrivateWithReply)
     2       4+n               request length
     4       11                vendor specific opcode
     4       GLX_CONTEXT_TAG   context tag
     4       INT32             n
     4*n     LISTofCARD32      textures
   =>
     1       1                 reply
     1                         unused
     2       CARD16            sequence number
     4       (n+p)/4           reply length
     4       BOOL32            return_value
     20                        unused
     n       LISTofBOOL        residences
     p                         unused, p=pad(n)

    IsTextureEXT
     1       CARD8             opcode (X assigned)
     1       17                GLX opcode (glXVendorPrivateWithReply)
     2       4                 request length
     4       14                vendor specific opcode
     4       GLX_CONTEXT_TAG   context tag
     4       CARD32            textures
   =>
     1       1                 reply
     1                         unused
     2       CARD16            sequence number
     4       0                 reply length
     4       BOOL32            return_value
     20                        unused
```

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, then all references to 3D textures in this specification are invalid.

Errors

INVALID_VALUE is generated if GenTexturesEXT parameter <n> is negative.

INVALID_VALUE is generated if DeleteTexturesEXT parameter <n> is negative.

INVALID_ENUM is generated if BindTextureEXT parameter <target> is not TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is TEXTURE_1D, and parameter <texture> is the name of a 2-dimensional or 3-dimensional texture object.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is TEXTURE_2D, and parameter <texture> is the name of a 1-dimensional or 3-dimensional texture object.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is

TEXTURE_3D_EXT, and parameter <texture> is the name of a 1-dimensional or 2-dimensional texture object.

INVALID_VALUE is generated if PrioritizeTexturesEXT parameter <n> negative.

INVALID_VALUE is generated if AreTexturesResidentEXT parameter <n> is negative.

INVALID_VALUE is generated by AreTexturesResidentEXT if any of the names in <textures> is zero, or is not the name of a texture.

INVALID_OPERATION is generated if any of the commands defined in this extension is executed between the execution of Begin and the corresponding execution of End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
TEXTURE_1D	IsEnabled	B	FALSE	texture/enable
TEXTURE_2D	IsEnabled	B	FALSE	texture/enable
TEXTURE_3D_EXT	IsEnabled	B	FALSE	texture/enable
TEXTURE_1D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_2D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_3D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_PRIORITY_EXT	GetTexParameterfv	n x Z+	1	texture
TEXTURE_RESIDENT_EXT	AreTexturesResidentEXT	n x B	unknown	-
TEXTURE	GetTexImage	n x levels x I	null	-
TEXTURE_RED_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_GREEN_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_BLUE_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_ALPHA_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_LUMINANCE_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_INTENSITY_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_WIDTH	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_HEIGHT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_DEPTH_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_4DSIZE_SGIS	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_BORDER	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_COMPONENTS (1D and 2D)	GetTexLevelParameteriv	n x levels x Z42	1	-
TEXTURE_COMPONENTS (3D and 4D)	GetTexLevelParameteriv	n x levels x Z38	LUMINANCE	-
TEXTURE_BORDER_COLOR	GetTexParameteriv	n x C	0, 0, 0, 0	texture
TEXTURE_MIN_FILTER	GetTexParameteriv	n x Z7	NEAREST_MIPMAP_LINEAR	texture
TEXTURE_MAG_FILTER	GetTexParameteriv	n x Z3	LINEAR	texture
TEXTURE_WRAP_S	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_Q_SGIS	GetTexParameteriv	n x Z2	REPEAT	texture

New Implementation Dependent State

None

Name

EXT_vertex_array

Name Strings

GL_EXT_vertex_array

Version

$Date: 1995/10/03 05:39:58 $ $Revision: 1.16 $ FINAL

Number

30

Dependencies

None

Overview

This extension adds the ability to specify multiple geometric primitives with very few subroutine calls. Instead of calling an OpenGL procedure to pass each individual vertex, normal, or color, separate arrays of vertexes, normals, and colors are prespecified, and are used to define a sequence of primitives (all of the same type) when a single call is made to DrawArraysEXT. A stride mechanism is provided so that an application can choose to keep all vertex data staggered in a single array, or sparsely in separate arrays. Single-array storage may optimize performance on some implementations.

This extension also supports the rendering of individual array elements, each specified as an index into the enabled arrays.

Issues

* Should arrays for material parameters be provided? If so, how?

A: No. Let's leave this to a separate extension, and keep this extension lean.

* Should a FORTRAN interface be specified in this document?

* It may not be possible to implement GetPointervEXT in FORTRAN. If not, should we eliminate it from this proposal?

A: Leave it in.

* Should a stride be specified by DrawArraysEXT which, if non-zero, would override the strides specified for the individual arrays? This might improve the efficiency of single-array transfers.

A: No, it's not worth the effort and complexity.

* Should entry points for byte vertexes, byte indexes, and byte texture coordinates be added in this extension?

A: No, do this in a separate extension, which defines byte support
for arrays and for the current procedural interface.

* Should support for meshes (not strips) of rectangles be provided?

A: No. If this is necessary, define a separate quad_mesh extension
that supports both immediate mode and arrays. (Add QUAD_MESH_EXT
as an token accepted by Begin and DrawArraysEXT. Add
QuadMeshLengthEXT to specify the length of the mesh.)

Reasoning

* DrawArraysEXT requires that VERTEX_ARRAY_EXT be enabled so that
future extensions can support evaluation as well as direct
specification of vertex coordinates.

* This extension does not support evaluation. It could be extended
to provide such support by adding arrays of points to be evaluated,
and by adding enables to indicate that the arrays are to be
evaluated. I think we may choose to add an array version of
EvalMesh, rather than extending the operation of DrawArraysEXT,
so I'd rather wait on this one.

* <size> is specified before <type> to match the order of the
information in immediate mode commands, such as Vertex3f.
(first 3, then f)

* It seems reasonable to allow attribute values to be undefined after
DrawArraysEXT executes. This avoids implementation overhead in
the case where an incomplete primitive is specified, and will allow
optimization on multiprocessor systems. I don't expect this to be
a burden to programmers.

* It is not an error to call VertexPointerEXT, NormalPointerEXT,
ColorPointerEXT, IndexPointerEXT, TexCoordPointerEXT,
or EdgeFlagPointerEXT between the execution of Begin and the
corresponding execution of End. Because these commands will
typically be implemented on the client side with no protocol,
testing for between-Begin-End status requires that the client
track this state, or that a round trip be made. Neither is
desirable.

* Arrays are enabled and disabled individually, rather than with a
single mask parameter, for two reasons. First, we have had trouble
allocating bits in masks, so eliminating a mask eliminates potential
trouble down the road. We may eventually require a larger number of
array types than there are bits in a mask. Second, making the
enables into state eliminates a parameter in ArrayElementEXT, and
may allow it to execute more efficiently. Of course this state
model may result in programming errors, but OpenGL is full of such
hazards anyway!

* ArrayElementEXT is provided to support applications that construct
primitives by indexing vertex data, rather than by streaming through
arrays of data in first-to-last order. Because each call specifies
only a single vertex, it is possible for an application to explicitly

specify per-primitive attributes, such as a single normal per individual triangle.

* The <count> parameters are added to the *PointerEXT commands to allow implementations to cache array data, and in particular to cache the transformed results of array data that are rendered repeatedly by ArrayElementEXT. Implementations that do not wish to perform such caching can ignore the <count> parameter.

* The <first> parameter of DrawArraysEXT allows a single set of arrays to be used repeatedly, possibly improving performance.

New Procedures and Functions

```
void ArrayElementEXT(int i);

void DrawArraysEXT(enum mode,
                   int first,
                   sizei count);

void VertexPointerEXT(int size,
                      enum type,
                      sizei stride,
                      sizei count,
                      const void* pointer);

void NormalPointerEXT(enum type,
                      sizei stride,
                      sizei count,
                      const void* pointer);

void ColorPointerEXT(int size,
                     enum type,
                     sizei stride,
                     sizei count,
                     const void* pointer);

void IndexPointerEXT(enum type,
                     sizei stride,
                     sizei count,
                     const void* pointer);

void TexCoordPointerEXT(int size,
                        enum type,
                        sizei stride,
                        sizei count,
                        const void* pointer);

void EdgeFlagPointerEXT(sizei stride,
                        sizei count,
                        const Boolean* pointer);

void GetPointervEXT(enum pname,
                    void** params);
```

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
VERTEX_ARRAY_EXT                0x8074
NORMAL_ARRAY_EXT                0x8075
COLOR_ARRAY_EXT                 0x8076
INDEX_ARRAY_EXT                 0x8077
TEXTURE_COORD_ARRAY_EXT         0x8078
EDGE_FLAG_ARRAY_EXT             0x8079
```

Accepted by the <type> parameter of VertexPointerEXT, NormalPointerEXT, ColorPointerEXT, IndexPointerEXT, and TexCoordPointerEXT:

```
DOUBLE_EXT                      0x140A
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
VERTEX_ARRAY_SIZE_EXT           0x807A
VERTEX_ARRAY_TYPE_EXT           0x807B
VERTEX_ARRAY_STRIDE_EXT         0x807C
VERTEX_ARRAY_COUNT_EXT          0x807D
NORMAL_ARRAY_TYPE_EXT           0x807E
NORMAL_ARRAY_STRIDE_EXT         0x807F
NORMAL_ARRAY_COUNT_EXT          0x8080
COLOR_ARRAY_SIZE_EXT            0x8081
COLOR_ARRAY_TYPE_EXT            0x8082
COLOR_ARRAY_STRIDE_EXT          0x8083
COLOR_ARRAY_COUNT_EXT           0x8084
INDEX_ARRAY_TYPE_EXT            0x8085
INDEX_ARRAY_STRIDE_EXT          0x8086
INDEX_ARRAY_COUNT_EXT           0x8087
TEXTURE_COORD_ARRAY_SIZE_EXT    0x8088
TEXTURE_COORD_ARRAY_TYPE_EXT    0x8089
TEXTURE_COORD_ARRAY_STRIDE_EXT  0x808A
TEXTURE_COORD_ARRAY_COUNT_EXT   0x808B
EDGE_FLAG_ARRAY_STRIDE_EXT      0x808C
EDGE_FLAG_ARRAY_COUNT_EXT       0x808D
```

Accepted by the <pname> parameter of GetPointervEXT:

```
VERTEX_ARRAY_POINTER_EXT        0x808E
NORMAL_ARRAY_POINTER_EXT        0x808F
COLOR_ARRAY_POINTER_EXT         0x8090
INDEX_ARRAY_POINTER_EXT         0x8091
TEXTURE_COORD_ARRAY_POINTER_EXT 0x8092
EDGE_FLAG_ARRAY_POINTER_EXT     0x8093
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

Array Specification

Individual array pointers and associated data are maintained for an array of vertexes, an array of normals, an array of colors, an array

of color indexes, an array of texture coordinates, and an array of edge flags. The data associated with each array specify the data type of the values in the array, the number of values per element in the array (e.g. vertexes of 2, 3, or 4 coordinates), the byte stride from one array element to the next, and the number of elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When an array is specified, the pointer and associated data are saved as client-side state, and static elements may be cached by the implementation. Non-static (dynamic) elements are never accessed until ArrayElementEXT or DrawArraysEXT is issued.

VertexPointerEXT specifies the location and data format of an array of vertex coordinates. <pointer> specifies a pointer to the first coordinate of the first vertex in the array. <type> specifies the data type of each coordinate in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <size> specifies the number of coordinates per vertex, and must be 2, 3, or 4. <stride> specifies the byte offset between pointers to consecutive vertexes. If <stride> is zero, the vertex data are understood to be tightly packed in the array. <count> specifies the number of vertexes, counting from the first, that are static.

NormalPointerEXT specifies the location and data format of an array of normals. <pointer> specifies a pointer to the first coordinate of the first normal in the array. <type> specifies the data type of each coordinate in the array, and must be one of BYTE, SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types byte, short, int, float, and double respectively. It is understood that each normal comprises three coordinates. <stride> specifies the byte offset between pointers to consecutive normals. If <stride> is zero, the normal data are understood to be tightly packed in the array. <count> specifies the number of normals, counting from the first, that are static.

ColorPointerEXT specifies the location and data format of an array of color components. <pointer> specifies a pointer to the first component of the first color element in the array. <type> specifies the data type of each component in the array, and must be one of BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE_EXT, implying GL data types byte, ubyte, short, ushort, int, uint, float, and double respectively. <size> specifies the number of components per color, and must be 3 or 4. <stride> specifies the byte offset between pointers to consecutive colors. If <stride> is zero, the color data are understood to be tightly packed in the array. <count> specifies the number of colors, counting from the first, that are static.

IndexPointerEXT specifies the location and data format of an array of color indexes. <pointer> specifies a pointer to the first index in the array. <type> specifies the data type of each index in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <stride> specifies the byte offset between pointers to consecutive indexes. If <stride> is zero, the index data are understood to be tightly packed

in the array. <count> specifies the number of indexes, counting from the first, that are static.

TexCoordPointerEXT specifies the location and data format of an array of texture coordinates. <pointer> specifies a pointer to the first coordinate of the first element in the array. <type> specifies the data type of each coordinate in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <size> specifies the number of coordinates per element, and must be 1, 2, 3, or 4. <stride> specifies the byte offset between pointers to consecutive elements of coordinates. If <stride> is zero, the coordinate data are understood to be tightly packed in the array. <count> specifies the number of texture coordinate elements, counting from the first, that are static.

EdgeFlagPointerEXT specifies the location and data format of an array of boolean edge flags. <pointer> specifies a pointer to the first flag in the array. <stride> specifies the byte offset between pointers to consecutive edge flags. If <stride> is zero, the edge flag data are understood to be tightly packed in the array. <count> specifies the number of edge flags, counting from the first, that are static.

The table below summarizes the sizes and data types accepted (or understood implicitly) by each of the six pointer-specification commands.

Command	Sizes	Types
VertexPointerEXT	2,3,4	short, int, float, double
NormalPointerEXT	3	byte, short, int, float, double
ColorPointerEXT	3,4	byte, short, int, float, double, ubyte, ushort, uint
IndexPointerEXT	1	short, int, float, double
TexCoordPointerEXT	1,2,3,4	short, int, float, double
EdgeFlagPointerEXT	1	boolean

Rendering the Arrays

By default all the arrays are disabled, meaning that they will not be accessed when either ArrayElementEXT or DrawArraysEXT is called. An individual array is enabled or disabled by calling Enable or Disable with <cap> set to appropriate value, as specified in the table below:

Array Specification Command	Enable Token
VertexPointerEXT	VERTEX_ARRAY_EXT
NormalPointerEXT	NORMAL_ARRAY_EXT
ColorPointerEXT	COLOR_ARRAY_EXT
IndexPointerEXT	INDEX_ARRAY_EXT
TexCoordPointerEXT	TEXTURE_COORD_ARRAY_EXT
EdgeFlagPointerEXT	EDGE_FLAG_ARRAY_EXT

When ArrayElementEXT is called, a single vertex is drawn, using vertex and attribute data taken from location <i> of the enabled arrays. The semantics of ArrayElementEXT are defined in the C-code below:

```
void ArrayElementEXT (int i) {
    byte* p;
    if (NORMAL_ARRAY_EXT) {
      if (normal_stride == 0)
          p = (byte*)normal_pointer + i * 3 * sizeof(normal_type);
      else
          p = (byte*)normal_pointer + i * normal_stride;
      Normal3<normal_type>v ((normal_type*)p);
    }
    if (COLOR_ARRAY_EXT) {
      if (color_stride == 0)
          p = (byte*)color_pointer +
            i * color_size * sizeof(color_type);
      else
          p = (byte*)color_pointer + i * color_stride;
      Color<color_size><color_type>v ((color_type*)p);
    }
    if (INDEX_ARRAY_EXT) {
      if (index_stride == 0)
          p = (byte*)index_pointer + i * sizeof(index_type);
      else
          p = (byte*)index_pointer + i * index_stride;
      Index<index_type>v ((index_type*)p);
    }
    if (TEXTURE_COORD_ARRAY_EXT) {
      if (texcoord_stride == 0)
          p = (byte*)texcoord_pointer +
            i * texcoord_size * sizeof(texcoord_type);
      else
          p = (byte*)texcoord_pointer + i * texcoord_stride;
      TexCoord<texcoord_size><texcoord_type>v ((texcoord_type*)p);
    }
    if (EDGE_FLAG_ARRAY_EXT) {
      if (edgeflag_stride == 0)
          p = (byte*)edgeflag_pointer + i * sizeof(boolean);
      else
          p = (byte*)edgeflag_pointer + i * edgeflag_stride;
      EdgeFlagv ((boolean*)p);
    }
    if (VERTEX_ARRAY_EXT) {
      if (vertex_stride == 0)
          p = (byte*)vertex_pointer +
            i * vertex_size * sizeof(vertex_type);
      else
          p = (byte*)vertex_pointer + i * vertex_stride;
      Vertex<vertex_size><vertex_type>v ((vertex_type*)p);
    }
}
```

ArrayElementEXT executes even if VERTEX_ARRAY_EXT is not enabled. No drawing occurs in this case, but the attributes corresponding to enabled arrays are modified.

When DrawArraysEXT is called, <count> sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with element <first>. <mode> specifies what kind of primitives are constructed, and how the array elements are used to

construct these primitives. Accepted values for <mode> are POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, and POLYGON. If VERTEX_ARRAY_EXT is not enabled, no geometric primitives are generated.

The semantics of DrawArraysEXT are defined in the C-code below:

```
void DrawArraysEXT(enum mode, int first, sizei count) {
    int i;
    if (count < 0)
      /* generate INVALID_VALUE error and abort */
    else {
      Begin (mode);
      for (i=0; i < count; i++)
          ArrayElementEXT(first + i);
      End ();
    }
}
```

The ways in which the execution of DrawArraysEXT differs from the semantics indicated in the pseudo-code above are:

1. Vertex attributes that are modified by DrawArraysEXT have an unspecified value after DrawArraysEXT returns. For example, if COLOR_ARRAY_EXT is enabled, the value of the current color is undefined after DrawArraysEXT executes. Attributes that aren't modified remain well defined.

2. Operation of DrawArraysEXT is atomic with respect to error generation. If an error is generated, no other operations take place.

Although it is not an error to respecify an array between the execution of Begin and the corresponding execution of End, the result of such respecification is undefined. Static array data may be read and cached by the implementation at any time. If static array data are modified by the application, the results of any subsequently issued ArrayElementEXT or DrawArraysEXT commands are undefined.

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

ArrayElementEXT and DrawArraysEXT are included in display lists. When either command is entered into a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client side state, their values affect display lists when the lists are created, not when the lists are executed.

Array specification commands VertexPointerEXT, NormalPointerEXT, ColorPointerEXT, IndexPointerEXT, TexCoordPointerEXT, and EdgeFlagPointerEXT specify client side state, and are therefore not included in display lists. Likewise Enable and Disable, when called with <cap> set to VERTEX_ARRAY_EXT, NORMAL_ARRAY_EXT, COLOR_ARRAY_EXT, INDEX_ARRAY_EXT, TEXTURE_COORD_ARRAY_EXT, or EDGE_FLAG_ARRAY_EXT, are not included in display lists. GetPointervEXT returns state information, and so is not included in display lists.

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

GetPointervEXT returns in <param> the array pointer value specified by <pname>. Accepted values for <pname> are VERTEX_ARRAY_POINTER_EXT, NORMAL_ARRAY_POINTER_EXT, COLOR_ARRAY_POINTER_EXT, INDEX_ARRAY_POINTER_EXT, TEXTURE_COORD_ARRAY_POINTER_EXT, and EDGE_FLAG_ARRAY_POINTER_EXT.

All array data are client side state, and are not saved or restored by PushAttrib and PopAttrib.

Additions to the GLX Specification

None

GLX Protocol

A new rendering command is added; it can be sent to the server as part of a glXRender request or as part of a glXRenderLarge request:

The DrawArraysEXT command consists of three sections, in the following order: (1) header information, (2) a list of array information, containing the type and size of the array values for each enabled array and (3) a list of vertex data. Each element in the list of vertex data contains information for a single vertex taken from the enabled arrays.

DrawArraysEXT

2	16+(12*m)+(s*n)	rendering command length
2	4116	rendering command opcode
4	CARD32	n (number of array elements)
4	CARD32	m (number of enabled arrays)
4	ENUM	mode /* GL_POINTS etc */
12*m	LISTofARRAY_INFO	
s*n	LISTofVERTEX_DATA	

Where s = ns + cs + is + ts + es + vs + np + cp + ip + tp + ep + vp. (See description below, under VERTEX_DATA.) Note that if an array is disabled then no information is sent for it. For example, when the normal array is disabled, there is no ARRAY_INFO record for the normal array and ns and np are both zero.

Note that the list of ARRAY_INFO is unordered: since the ARRAY_INFO record contains the array type, the arrays in the list may be stored in any order. Also, the VERTEX_DATA list is a packed list of vertices. For each vertex, data is retrieved from the enabled arrays, and stored in the list.

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields above are expanded to 4 bytes each:

```
    4           20+(12*m)+(s*n) rendering command length
    4           4116            rendering command opcode

ARRAY_INFO
    4             ENUM                  data type
          0x1400  i=1           BYTE
          0x1401  i=1           UNSIGNED_BYTE
          0x1402  i=2           SHORT
          0x1403  i=2           UNSIGNED_SHORT
          0x1404  i=4           INT
          0x1405  i=4           UNSIGNED_INT
          0x1406  i=4           FLOAT
          0x140A  i=8           DOUBLE_EXT
    4             INT32                 j (number of values in array element)
    4             ENUM                  array type
          0x8074  j=2/3/4       VERTEX_ARRAY_EXT
          0x8075  j=3           NORMAL_ARRAY_EXT
          0x8076  j=3/4         COLOR_ARRAY_EXT
          0x8077  j=1           INDEX_ARRAY_EXT
          0x8078  j=1/2/3/4     TEXTURE_COORD_ARRAY_EXT
          0x8079  j=1           EDGE_FLAG_ARRAY_EXT

 For each array, the size of an array element is i*j. Some arrays
    (e.g., the texture coordinate array) support different data sizes;
    for these arrays, the size, j, is specified when the array is defined.

VERTEX_DATA
 if the normal array is enabled:

    ns            LISTofBYTE          normal array element
    np                                unused, np=pad(ns)

    if the color array is enabled:

    cs            LISTofBYTE          color array element
    cp                                unused, cp=pad(cs)

    if the index array is enabled:

    is            LISTofBYTE          index array element
    ip                                unused, ip=pad(is)

    if the texture coord array is enabled:

    ts            LISTofBYTE          texture coord array element
    tp                                unused, tp=pad(ts)

    if the edge flag array is enabled:

    es            LISTofBYTE          edge flag array element
    ep                                unused, ep=pad(es)

    if the vertex array is enabled:

    vs            LISTofBYTE          vertex array element
    vp                                unused, vp=pad(vs)

 where ns, cs, is, ts, es, vs is the size of the normal, color, index,
 texture, edge and vertex array elements and np, cp, ip, tp, ep, vp is
 the padding for the normal, color, index, texture, edge and vertex array
 elements, respectively.
```

Errors

INVALID_OPERATION is generated if DrawArraysEXT is called between the execution of Begin and the corresponding execution of End.

INVALID_ENUM is generated if DrawArraysEXT parameter <mode> is not POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, or POLYGON.

INVALID_VALUE is generated if DrawArraysEXT parameter <count> is negative.

INVALID_VALUE is generated if VertexPointerEXT parameter <size> is not 2, 3, or 4.

INVALID_ENUM is generated if VertexPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if VertexPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if NormalPointerEXT parameter <type> is not BYTE, SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if NormalPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if ColorPointerEXT parameter <size> is not 3 or 4.

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

INVALID_VALUE is generated if ColorPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if IndexPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if IndexPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if TexCoordPointerEXT parameter <size> is not 1, 2, 3, or 4.

INVALID_ENUM is generated if TexCoordPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if TexCoordPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if EdgeFlagPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if GetPointervEXT parameter <pname> is not VERTEX_ARRAY_POINTER_EXT, NORMAL_ARRAY_POINTER_EXT,

COLOR_ARRAY_POINTER_EXT, INDEX_ARRAY_POINTER_EXT,
TEXTURE_COORD_ARRAY_POINTER_EXT, or EDGE_FLAG_ARRAY_POINTER_EXT.

New State

Get Value	Get Command	Type	Initial Value	Attrib
VERTEX_ARRAY_EXT	IsEnabled	B	False	client
VERTEX_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
VERTEX_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
VERTEX_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
VERTEX_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
VERTEX_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
NORMAL_ARRAY_EXT	IsEnabled	B	False	client
NORMAL_ARRAY_TYPE_EXT	GetIntegerv	Z5	FLOAT	client
NORMAL_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
NORMAL_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
NORMAL_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
COLOR_ARRAY_EXT	IsEnabled	B	False	client
COLOR_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
COLOR_ARRAY_TYPE_EXT	GetIntegerv	Z8	FLOAT	client
COLOR_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
COLOR_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
COLOR_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
INDEX_ARRAY_EXT	IsEnabled	B	False	client
INDEX_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
INDEX_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
INDEX_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
INDEX_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
TEXTURE_COORD_ARRAY_EXT	IsEnabled	B	False	client
TEXTURE_COORD_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
TEXTURE_COORD_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
TEXTURE_COORD_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
TEXTURE_COORD_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
TEXTURE_COORD_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
EDGE_FLAG_ARRAY_EXT	IsEnabled	B	False	client
EDGE_FLAG_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
EDGE_FLAG_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
EDGE_FLAG_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client

New Implementation Dependent State

None

Name

EXT_vertex_weighting

Name Strings

GL_EXT_vertex_weighting

Notice

Copyright NVIDIA Corporation, 1999, 2000.

Status

Shipping (version 1.0)

Version

NVIDIA Date: May 25, 2000

Number

188

Dependencies

None

Written based on the wording of the OpenGL 1.2 specification but not dependent on it.

Overview

The intent of this extension is to provide a means for blending geometry based on two slightly differing modelview matrices. The blending is based on a vertex weighting that can change on a per-vertex basis. This provides a primitive form of skinning.

A second modelview matrix transform is introduced. When vertex weighting is enabled, the incoming vertex object coordinates are transformed by both the primary and secondary modelview matrices; likewise, the incoming normal coordinates are transformed by the inverses of both the primary and secondary modelview matrices. The resulting two position coordinates and two normal coordinates are blended based on the per-vertex vertex weight and then combined by addition. The transformed, weighted, and combined vertex position and normal are then used by OpenGL as the eye-space position and normal for lighting, texture coordinate, generation, clipping, and further vertex transformation.

Issues

Should the extension be written to extend to more than two vertex weights and modelview matrices?

RESOLUTION: NO. Supports only one vertex weight and two modelview matrices. If more than two is useful, that can be handled with

another extension.

Should the weighting factor be GLclampf instead of GLfloat?

RESOLUTION: GLfloat. Though the value of a weighting factors outside the range of zero to one (and even weights that do not add to one) is dubious, there is no reason to limit the implementation to values between zero and one.

Should the weights and modelview matrices be labeled 1 & 2 or 0 & 1?

RESOLUTION: 0 & 1. This is consistent with the way lights and texture units are named in OpenGL. Make GL_MODELVIEW0_EXT be an alias for GL_MODELVIEW. Note that the GL_MODELVIEW0_EXT+1 will not be GL_MODELVIEW1_EXT as is the case with GL_LIGHT0 and GL_LIGHT1.

Should there be a way to simultaneously Rotate, Translate, Scale, LoadMatrix, MultMatrix, etc. the two modelview matrices together?

RESOLUTION: NO. The application must use MatrixMode and repeated calls to keep the matrices in sync if desired.

Should the secondary modelview matrix stack be as deep as the primary matrix stack or can they be different sizes?

RESOLUTION: Must be the SAME size. This wastes a lot of memory that will be probably never be used (the modelview matrix stack must have at least 32 entries), but memory is cheap.

The value returned by MAX_MODELVIEW_STACK_DEPTH applies to both modelview matrices.

Should there be any vertex array support for vertex weights.

RESOLUTION: YES.

Should we have a VertexWeight2fEXT that takes has two weight values?

RESOLUTION: NO. The weights are always vw and 1-vw.

What is the "correct" way to blend matrices, particularly when wo is not one or the modelview matrix is projective?

RESOLUTION: While it may not be 100% correct, the extension blends the vertices based on transforming the object coordinates by both M0 and M1, but the resulting w coordinate comes from simply transforming the object coordinates by M0 and extracting the w.

Another option would be to simply blend the two sets of eye coordinates without any special handling of w. This is harder.

Another option would be to divide by w before blending the two sets of eye coordinates. This is awkward because if the weight is 1.0 with vertex weighting enabled, the result is not the same as disabling vertex weighting since EYE_LINEAR texgen is based of of the non-perspective corrected eye coordinates.

As specified, the normal weighting and combination is performed on unnormalized normals. Would the math work better if the normals were normalized before weighting and combining?

RESOLUTION: Vertex weighting of normals is after the GL_RESCALE_NORMAL step and before the GL_NORMALIZE step.

As specified, feedback and selection should apply vertex weighting if enabled. Yuck, that would mean that we need software code for vertex weighting.

RESOLUTION: YES, it should work with feedback and selection.

Sometimes it would be useful to mirror changes in both modelview matrices. For example, the viewing transforms are likely to be different, just the final modeling transforms would be different. Should there be an API support for mirroring transformations into both matrices?

RESOLUTION: NO. Such support is likely to complicate the matrix management in the OpenGL. Applications can do a Get matrix from modelview0 and then a LoadMatrix into modelview1 manually if they need to mirror things.

I also worry that if we had a mirrored matrix mode, it would double the transform concatenation work if used naively.

Many of the changes to the two modelview matrices will be the same. For example, the initial view transform loaded into each will be the same. Should there be a way to "mirror" changes to both modelview matrices?

RESOLUTION: NO. Mirroring matrix changes would complicate the driver's management of matrices. Also, I am worried that naive users would mirror all transforms and lead to lots of redundant matrix concatenations. The most efficient way to handle the slight differences between the modelview matrices is simply to GetFloat the primary matrix, LoadMatrix the values in the secondary modelview matrix, and then perform the "extra" transform to the secondary modelview matrix.

Ideally, a glCopyMatrix(GLenum src, GLenum dst) type OpenGL command could make this more efficient. There are similiar cases where you want the modelview matrix mirrored in the texture matrix. This is not the extension to solve this minor problem.

The post-vertex weighting normal is unlikely to be normalized. Should this extension automatically enable normalization?

RESOLUTION: NO. Normalization should operate as specified. The user is responsible for enabling GL_RESCALE_NORMAL or GL_NORMALIZE as needed.

You could imagine cases where the application only sent vertex weights of either zero or one and pre-normalized normals so that GL_NORMALIZE would not strictly be required.

Note that the vertex weighting of transformed normals occurs BEFORE normalize and AFTER rescaling. See the issue below for why this can make a difference.

How does vertex weighting interact with OpenGL 1.2's GL_RESCALE_NORMAL enable?

RESOLUTION: Vertex weighting of transformed normals occurs BEFORE normalize and AFTER rescaling.

OpenGL 1.2 permits normal rescaling to behave just like normalize and because normalize immediately follows rescaling, enabling rescaling can be implementied by simply always enabling normalize.

Vertex weighting changes this. If one or both of the modelview matrices has a non-uniform scale, it may be useful to enable rescaling and normalize and this operates differently than simply enabling normalize. The difference is that rescaling occurs before the normal vertex weighting.

An implementation that truly treated rescaling as a normalize would support both a pre-weighting normalize and a post-weighting normalize. Arguably, this is a good thing.

For implementations that perform simply rescaling and not a full normalize to implement rescaling, the rescaling factor can be concatenated into each particular inverse modelview matrix.

New Procedures and Functions

```
void VertexWeightfEXT(float weight);

void VertexWeightfvEXT(float *weight);

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

New Tokens

Accepted by the <target> parameter of Enable:

VERTEX_WEIGHTING_EXT	0x8509

Accepted by the <mode> parameter of MatrixMode:

MODELVIEW0_EXT	0x1700	(alias to MODELVIEW enumerant)
MODELVIEW1_EXT	0x850A	

```
    Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
    GetFloatv, and GetDoublev:

        VERTEX_WEIGHTING_EXT
        MODELVIEW0_EXT
        MODELVIEW1_EXT
        MODELVIEW0_MATRIX_EXT           0x0BA6  (alias to MODELVIEW_MATRIX)
        MODELVIEW1_MATRIX_EXT           0x8506
        CURRENT_VERTEX_WEIGHT_EXT       0x850B
        VERTEX_WEIGHT_ARRAY_EXT         0x850C
        VERTEX_WEIGHT_ARRAY_SIZE_EXT    0x850D
        VERTEX_WEIGHT_ARRAY_TYPE_EXT    0x850E
        VERTEX_WEIGHT_ARRAY_STRIDE_EXT  0x850F
        MODELVIEW0_STACK_DEPTH_EXT      0x0BA3  (alias to MODELVIEW_STACK_DEPTH)
        MODELVIEW1_STACK_DEPTH_EXT      0x8502

     Accepted by the <pname> parameter of GetPointerv:

        VERTEX_WEIGHT_ARRAY_POINTER_EXT     0x8510
```

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

-- **Section 2.6. 2nd paragraph changed:**

"Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current vertex weight may be used in processing each vertex."

-- **Section 2.6. New paragraph after the 3rd paragraph:**

"A vertex weight is associated with each vertex. When vertex weighting is enabled, this weight is used as a blending factor to blend the position and normals transformed by the primary and secondary modelview matrix transforms. The vertex weighting functionality takes place completely in the "vertex / normal transformation" stage of Figure 2.2."

-- **Section 2.6.3. First paragraph changed to**

"The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, and texture coordinates (Vertex, Color, VertexWeightEXT, Index, Normal, TexCoord)..."

-- **Section 2.7. New paragraph after the 4th paragraph:**

"The current vertex weight is set using

```
    void VertexWeightfEXT(float weight);
    void VertexWeightfvEXT(float *weight);
```

This weight is used when vertex weighting is enabled."

-- **Section 2.7. The last paragraph changes from**

"... and one floating-point value to store the current color index."

to:

"... one floating-point number to store the vertex weight, and one floating-point value to store the current color index."

-- **Section 2.8. Change 1st paragraph to say:**

"The client may specify up to seven arrays: one each to store edge flags, texture coordinates, colors, color indices, vertex weights, normals, and vertices. The commands"

Add to functions listed following first paragraph:

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

Add to table 2.4 (p. 22):

Command	Sizes	Types
VertexWeightPointerEXT	1	float

Starting with the second paragraph on p. 23, change to add VERTEX_WEIGHT_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
       void EnableClientState(enum array)
       void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY, INDEX_ARRAY, VERTEX_ARRAY_WEIGHT_EXT, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, color, secondary color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
       void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, color, secondary color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, Color<size><type>v, Index<type>v, VertexWeightfvEXT, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable vertex weight array for canned interleaved array formats. After the lines

```
       DisableClientState(EDGE_FLAG_ARRAY);
       DisableClientState(INDEX_ARRAY);
```

insert the line

```
        DisableClientState(VERTEX_WEIGHT_ARRAY_EXT);
```

Substitute "seven" for every occurrence of "six" in the final paragraph on p. 27.

-- **Section 2.10. Change the sentence:**

"The model-view matrix is applied to these coordinates to yield eye coordinates."

to:

"The primary modelview matrix is applied to these coordinates to yield eye coordinates. When vertex weighting is enabled, a secondary modelview matrix is also applied to the vertex coordinates, the result of the two modelview transformations are weighted by its respective vertex weighting factor and combined by addition to yield the true eye coordinates. Vertex weighting is enabled or disabled using Enable and Disable (see section 2.10.3) with an argument of VERTEX_WEIGHTING_EXT."

Change the 4th paragraph to:

"If vertex weighting is disabled and a vertex in object coordinates is given by (xo yo zo wo)' and the primary model-view matrix is M0, then the vertex's eye coordinates are found as

```
   (xe ye ze we)'  =  M0 (xo yo zo wo)'
```

If vertex weighting is enabled, then the vertex's eye coordinates are found as

```
   (xe0 ye0 ze0 we0)'  =  M0 (xo yo zo wo)'

   (xe1 ye1 ze1 we1)'  =  M1 (xo yo zo wo)'

   (xe,ye,ze)' = vw*(xe0,ye0,ze0)' + (1-vw) * (xe1,ye1,ze1)'

   we = we0
```

where M1 is the secondary modelview matrix and vw is the current vertex weight."

-- **Section 2.10.2 Change the 1st paragraph to say:**

"The projection matrix and the primary and secondary modelview matrices are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
   void MatrixMode(enum mode);
```

which takes one of the four pre-defined constants TEXTURE, MODELVIEW0, MODELVIEW1, or PROJECTION (note that MODELVIEW is an alias for MODELVIEW0). TEXTURE is described later. If the current matrix is MODELVIEW0, then matrix operations apply to the primary

modelview matrix; if MODELVIEW1, then matrix operations apply to the secondary modelview matrix; if PROJECTION, then they apply to the projection matrix."

Change the 9th paragraph to say:

"There is a stack of matrices for each of the matrix modes. For the MODELVIEW0 and MODELVIEW1 modes, the stack is at least 32 (that is, there is a stack of at least 32 modelview matrices). ..."

Change the last paragraph to say:

"The state required to implement transformations consists of a four-valued integer indicating the current matrix mode, a stack of at least two 4x4 matrices for each of PROJECTION and TEXTURE with associated stack pointers, and two stacks of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW0 and MODELVIEW1. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW0."

-- **Section 2.10.3 Change the 2nd and 7th paragraphs to say:**

"For a modelview matrix M, the normal for this matrix is transformed to eye coordinates by:

$$(nx'\ ny'\ nz'\ q') = (nx\ ny\ nz\ q) * M^{-1}$$

where, if (x y z w)' are the associated vertex coordinates, then

$$q = \begin{cases} 0, & w = 0 \\ -\dfrac{(nx\ ny\ nz)\ (x\ y\ z)'}{w}, & w \neq 0 \end{cases} \qquad (2.1)$$

Implementations may choose instead to transform (x y z)' to eye coordinates using

$$(nx'\ ny'\ nz') = (nx\ ny\ nz) * Mu^{-1}$$

Where Mu is the upper leftmost 3x3 matrix taken from M.

Rescale multiplies the transformed normals by a scale factor

$$(\ nx"\ ny"\ nz"\ ) = f\ (nx'\ ny'\ nz')$$

If rescaling is disabled, then f = 1. If rescaling is enabled, then f is computed as (mij denotes the matrix element in row i and column j of M^-1, numbering the topmost row of the matrix as row 1 and the leftmost column as column 1

$$f = \frac{1}{\sqrt{m31^2 + m32^2 + m33^2}}$$

Note that if the normals sent to GL were unit length and the model-view matrix uniformly scales space, the rescale make sthe transformed normals

unit length.

Alternatively, an implementation may chose f as

$$f = \frac{1}{\text{sqrt}(nx'^2 + ny'^2 + nz'^2)}$$

recomputing f for each normal. This makes all non-zero length normals unit length regardless of their input length and the nature of the modelview matrix.

After rescaling, the final transformed normal used in lighting, nf, depends on whether vertex weighting is enabled or not.

When vertex weighting is disabled, nf is computed as

nf = m * (nx"0 ny"0 nz"0)

where (nx"0 ny"0 nz"0) is the normal transformed as described above using the primary modelview matrix for M.

If normalization is enabled m=1. Otherwise

$$m = \frac{1}{\text{sqrt}(nx"0^2 + ny"0^2 + nz"0^2)}$$

However when vertex weighting is enabled, the normal is transformed twice as described above, once by the primary modelview matrix and again by the secondary modelview matrix, weighted using the current per-vertex weight, and normalized. So nf is computed as

nf = m * (nx"w ny"w nz"w)

where nw is the weighting normal computed as

nw = vw * (nx"0 ny"0 nz"0) + (1-vw) * (nx"1 ny"1 nz"1)

where (nx"0 ny"0 nz"0) is the normal transformed as described above using the primary modelview matrix for M, and (nx"1 ny"1 nz"1) is the normal transformed as described above using the secondary modelview matrix for M, and vw is the current pver-vertex weight."

-- **Section 2.12. Changes the 3rd paragraph:**

"The coordinates are treated as if they were specified in a Vertex command. The x, y, z, and w coordinates are transformed by the current primary modelview and perspective matrices. These coordinates, along with current values, are used to generate a color and texture coordinates just as done for a vertex, except that vertex weighting is always treated as if it is disabled."

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
VertexWeightfvEXT
    2  8        rendering command length
    2  4135     rendering command opcode
    4  FLOAT32  weight0
```

To support vertex arrays, the DrawArrays rendering command (sent via a glXRender or glXRenderLarge request) is amended as follows:

The list of arrays listed for the third element in the ARRAY_INFO structure is amended to include:

```
            0x850c         j=1        VERTEX_WEIGHT_ARRAY_EXT
```

The VERTEX_DATA description is amended to include:

```
   If the vertex weight array is enabled:
   ws            LISTofBYTE        vertex weight array element
   wp                              unused, wp=pad(ws)
```

with the following paragraph amended to read:

"where ns, cs, is, ts, es, vs, ws is the size of the normal, color, index, texture, edge, vertex, and vertex weight array elements and np, cp, ip, tp, ep, vp, wp is the padding for the normal, color, index, texture, edge, vertex, and vertex weight array elements, respectively."

Errors

The current vertex weight can be updated at any time. In particular WeightVertexEXT can be called between a call to Begin and the corresponding call to End.

INVALID_VALUE is generated if VertexWeightPointerEXT parameter <size> is not 1.

INVALID_ENUM is generated if VertexWeightPointerEXT parameter <type> is not FLOAT.

INVALID_VALUE is generated if VertexWeightPointerEXT parameter <stride> is negative.

New State

(table 6.5, p196)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_VERTEX_WEIGHT_EXT	F	GetFloatv	1	Current vertex weight	2.8	current

(table 6.6, p197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_WEIGHT_ARRAY_EXT	B	IsEnabled	False	Vertex weight enable	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_SIZE_EXT	Z+	GetIntegerv	1	Weights per vertex	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_TYPE_EXT	Z1	GetIntegerv	FLOAT	Type of weights	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_STRIDE_EXT	Z	GetIntegerv	0	Stride between weights	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to vertex weight array	2.8	vertex-array

(table 6.7, p198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
MODELVIEW0_MATRIX_EXT	32*xM4	GetFloatv	Identity	Primary modelview stack	2.10.2	-
MODELVIEW1_MATRIX_EXT	32*xM4	GetFloatv	Identity	Secondary modelview stack	2.10.2	-
MODELVIEW0_STACK_DEPTH_EXT	Z+	GetIntegerv	1	Primary modelview stack depth	2.10.2	-
MODELVIEW1_STACK_DEPTH_EXT	Z+	GetIntegerv	1	Secondary modelview stack depth	2.10.2	-
MATRIX_MODE	Z4	GetIntegerv	MODELVIEW0	Current matrix mode	2.10.2	transform
VERTEX_WEIGHTING_EXT	B	IsEnabled	False	Vertex weighting on/off	2.10.2	transform/enable

NOTE: MODELVIEW_MATRIX is an alias for MODELVIEW0_MATRIX_EXT
MODELVIEW_STACK_DEPTH is an alias for MODELVIEW0_STACK_DEPTH_EXT

New Implementation Dependent State

None

Revision History

12/16/2000 amended to include GLX protocol for vertex arrays
5/25/2000 added missing MODELVIEW#_MATRIX tokens values

Name

IBM_texture_mirrored_repeat

Name Strings

GL_IBM_texture_mirrored_repeat

Version

$Date: 1999/12/28 01:40:35 $ $Revision: 1.2 $
IBM Id: texture_mirrored_repeat.spec,v 1.5 1998/01/16 18:09:31 pbrown Exp

Number

unassigned

Dependencies

EXT_texture_3D
IBM_texture_edge_clamp

Overview

IBM_texture_mirrored_repeat extends the set of texture wrap modes to include a mode (GL_MIRRORED_REPEAT_IBM) that effectively uses a texture map twice as large at the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching "mirror image".

Issues

* The spec clamps the final (u,v) coordinates to the range [0.5, 2^n-0.5]. This will produce the same effect as trapping a sample of the border texel and using the corresponding edge texel. The choice of technique is purely an implementation detail.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT:

GL_MIRRORED_REPEAT_IBM	0x8370

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Change to Section 3.8 (Subsection "Texture Wrap Modes")

If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT is set to MIRRORED_REPEAT_IBM, the s (or t or r) coordinate is converted to:

```
 s - floor(s),       if floor(s) is even, or
 1 - (s - floor(s)), if floor(s) is odd.
```

Change to Section 3.8.1, Texture Minification

Let:

```
 u(x,y) = 2^n * s(x,y),
 v(x,y) = 2^m * t(x,y), and
 w(x,y) = 2^l * r(x,y).
```

If the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT is set to either MIRRORED_REPEAT_IBM or CLAMP_TO_EDGE_IBM, the resulting u, v, or w coordinates (respectively) are clamped to the range [0.5, 2^n-0.5].

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None.

Errors

None

Dependencies on EXT_texture3D

If EXT_texture3D is not implemented, then the references clamping of 3D textures in this file are invalid, and references to TEXTURE_WRAP_R_EXT should be ignored.

Dependencies on IBM_texture_edge_clamp

If IBM_texture_edge_clamp is not implemented, then the references to CLAMP_TO_EDGE_IBM should be ignored.

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_WRAP_S	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z5	REPEAT	texture

New Implementation Dependent State

None

Name

NV_blend_square

Name Strings

GL_NV_blend_square

Version

Date: 8/7/1999 Version: 1.0

Number

194

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

It is useful to be able to multiply a number by itself in the blending stages -- for example, in certain types of specular lighting effects where a result from a dot product needs to be taken to a high power.

This extension provides four additional blending factors to permit this and other effects: SRC_COLOR and ONE_MINUS_SRC_COLOR for source blending factors, and DST_COLOR and ONE_MINUS_DST_COLOR for destination blending factors.

New Procedures and Functions

None

New Tokens

None

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

Two lines are added to each of tables 4.1 and 4.2:

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs, Gs, Bs, As)	*NEW*
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)	*NEW*
DST_COLOR	(Rd, Gd, Bd, Ad)	
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR	(Rc, Gc, Bc, Ac)	
ONE_MINUS_CONSTANT_COLOR	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	
CONSTANT_ALPHA	(Ac, Ac, Ac, Ac)	
ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	
SRC_ALPHA_SATURATE	(f, f, f, 1)	

Table 4.1: Values controlling the source blending function and the source blending values they compute. f = min(As, 1 - Ad).

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs, Gs, Bs, As)	
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)	
DST_COLOR	(Rd, Gd, Bd, Ad)	*NEW*
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)	*NEW*
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	

Table 4.2: Values controlling the destination blending function and the destination blending values they compute.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

New State

(table 6.15, page 205)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
BLEND_SRC	Z15	GetIntegerv	ONE	4.1.6	color-buffer
BLEND_DST	Z14	GetIntegerv	ZERO	4.1.6	color-buffer

NOTE: the only change is that Z13 changes to Z15 and Z12 changes to Z14

New Implementation Dependent State

None

Name

NV_evaluators

Name Strings

GL_NV_evaluators

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_evaluators.txt#2 $

Number

225

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

Assumes support for the ARB_multitexture extension.

NV_vertex_program affects the definition of this extension.

Overview

OpenGL evaluators provide applications with the capability to specify polynomial or rational curves and surfaces using control points relative to the Bezier basis. The curves and surfaces are then drawn by evaluating the polynomials provided at various values for the u parameter of a curve or the (u,v) parameters of a surface. A tensor product formulation is used for the surfaces.

For various historical reasons, evaluators have not been particularly popular as an interface for drawing curves and surfaces. This extension proposes a new interface for surfaces that provides a number of significant enhancements to the functionality provided by the original OpenGL evaluators.

Many implementations never optimized evaluators, so applications often implemented their own algorithms instead. This extension relaxes some restrictions that make it difficult to optimize evaluators.

Also, new vertex attributes have been added to OpenGL through extensions, including multiple sets of texture coordinates, a secondary color, a fog coordinate, a vertex weight, and others. The extensions which added these vertex attributes never bothered

to update the functionality of evaluators, since they were used so little in the first place. In turn, evaluators have become more and more out of date, making it even less likely that developers will want to use them. Most of the attributes are not a big loss, but support for multiple sets of texture coordinates would be absolutely essential to developers considering the use of evaluators.

OpenGL evaluators only support rectangular patches, not triangular patches. Although triangular patches can be converted into rectangular patches, direct support for triangular patches is likely to be more efficient.

The tessellation algorithm used is too inflexible for most purposes; only the number of rows and columns can be specified. Adjacent patches must then have identical numbers of rows and columns, or severe cracking will occur. Ideally, a number of subdivisions could be specified for all four sides of a rectangular patch and for all three of a triangular patch. This extension goes one step further and allows those numbers to be specified in floating-point, providing a mechanism for smoothly changing the level of detail of the surface.

Meshes evaluated with EvalMesh are required to match up exactly with equivalent meshes evaluated with EvalCoord or EvalPoint. This makes it difficult or impossible to use optimizations such as forward differencing.

Finally, little attention is given to some of the difficult problems that can arise when multiple patches are drawn. Depending on the way evaluators are implemented, and depending on the orientation of edges, numerical accuracy problems can cause cracks to appear between patches with the same boundary control points. This extension makes guarantees that an edge shared between two patches will match up exactly under certain conditions.

Issues

* *Should one-dimensional evaluators be supported?*

  RESOLVED: No. This extension is intended for surfaces only.

* *Should we support triangular patches?*

  RESOLVED: Yes. Otherwise, applications will have to convert them to rectangular patches themselves. We can do this more efficiently.

* *What domain should triangular patches be defined on?*

  RESOLVED: (0,0),(1,0),(0,1).

* *What memory layout should we use for triangular patch control points?*

  RESOLVED: Both a[i][j], where i+j <= n, and a packed format are supported.

* *Is it worth it to have "evaluator objects"?*

  RESOLVED: No. We will leave these out for now.

* *Should we support the original evaluators' ability to specify a map from an arbitrary (u1,v1) to an arbitrary (u2,v2)?*

  RESOLVED: No. Maps will always extend from (0,0) to (1,1) and will always be evaluated from (0,0) to (1,1).

* *Should the new interface support an EvalCoord-like syntax?*

  RESOLVED: No. We are only interested in evaluating an entire mesh at once.

* *Should we support the "mode" parameter to the existing EvalMesh2, which allows the mesh to be tessellated in wireframe or as points?*

  RESOLVED: No. We will leave in the parameter and require that it be FILL, though, to leave room for a future extension.

* *Should there be a new interface to specify control points or should Map2{fd} be reused?*

  RESOLVED: A new interface. There are enough changes compared to the original evaluators that we can't reuse the old interface without causing more problems. For example, the target parameter of Map2{fd} is really a cross of target and index in MapControlPointsNV, and so it ends up creating an excessive number of enumerants.

* *How should grids be specified?*

  RESOLVED: A MapParameter command. This is better than a new MapGrid- style command because it can be extended to include new parameter types.

* *Should there be any rules about the order of generation of primitives within a single patch?*

  RESOLVED: No. The tessellation algorithm itself is not even specified, so it makes no sense to do this. Applications must not depend on the order in which the primitives are drawn.

* *Should the stride for MapControlPointsNV be specified in basic machine units (i.e. unsigned bytes) or in floats/doubles?*

  RESOLVED: ubytes. Most of the rest of OpenGL (vertex arrays, pixel path, etc.) uses ubytes; evaluators are actually inconsistent.

* *How much leeway should implementations be given to choose their own algorithm for tessellation?*

  RESOLVED: The integral tessellation scheme will require a specific tessellation of the boundary edges of the patch, but the interior tessellation is implementation-specific. The fractional

tessellation scheme will only require a minimum number of segments along each edge. In either case, a minimum number of triangles for the entire patch is specified.

* *Should there be rules to ensure that the triangles will be oriented in a consistent fashion?*

  RESOLVED: Yes. This is essential for features such as backface culling to work properly. The rule given ensures that the orientation will be identical to the orientation used for the original evaluators.

* *Should there be a separate MAX_EVAL_ORDER for rational surfaces?*

  RESOLVED: Yes. Rational surfaces require additional calculation to be done by the implementation, especially if AUTO_NORMAL is enabled. Furthermore, the most useful rational surfaces are of low order. For example, all the conic sections are quadratic rational surfaces.

* *Should there be enables similar to AUTO_NORMAL that generate partials of U (tangents), partials of V, and/or binormals?*

  RESOLVED: No. The application is responsible for configuring the evaluators appropriately.

  The auto normal functionality is supported because it is fairly complicated and was already a core part of OpenGL for evaluators. Plus there is already a "normal" vertex attribute for it to automatically generate.

  The partials of U and partials of V are fairly straightforward to evaluate (just take the derivative of the bivariate polynomial in terms of either U or V) plus there is not a particular vertex attribute associated with each of these.

New Procedures and Functions

```
void MapControlPointsNV(enum target, uint index, enum type,
                        sizei ustride, sizei vstride,
                        int uorder, int vorder,
                        boolean packed,
                        const void *points)

void MapParameterivNV(enum target, enum pname, const int *params)
void MapParameterfvNV(enum target, enum pname, const float *params)

void GetMapControlPointsNV(enum target, uint index, enum type,
                           sizei ustride, sizei vstride,
                           boolean packed, void *points)

void GetMapParameterivNV(enum target, enum pname, int *params)
void GetMapParameterfvNV(enum target, enum pname, float *params)
void GetMapAttribParameterivNV(enum target, uint index, enum pname,
                               int *params)
void GetMapAttribParameterfvNV(enum target, uint index, enum pname,
                               float *params)
```

```
    void EvalMapsNV(enum target, enum mode)
```

New Tokens

Accepted by the <target> parameter of MapControlPointsNV, MapParameter[if]vNV, GetMapControlPointsNV, GetMapParameter[if]vNV, GetMapAttribParameter[if]vNV, and EvalMapsNV:

```
    EVAL_2D_NV                          0x86C0
    EVAL_TRIANGULAR_2D_NV               0x86C1
```

Accepted by the <pname> parameter of MapParameter[if]vNV and GetMapParameter[if]vNV:

```
    MAP_TESSELLATION_NV                 0x86C2
```

Accepted by the <pname> parameter of GetMapAttribParameter[if]vNV:

```
    MAP_ATTRIB_U_ORDER_NV               0x86C3
    MAP_ATTRIB_V_ORDER_NV               0x86C4
```

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    EVAL_FRACTIONAL_TESSELLATION_NV     0x86C5

    EVAL_VERTEX_ATTRIB0_NV              0x86C6
    EVAL_VERTEX_ATTRIB1_NV              0x86C7
    EVAL_VERTEX_ATTRIB2_NV              0x86C8
    EVAL_VERTEX_ATTRIB3_NV              0x86C9
    EVAL_VERTEX_ATTRIB4_NV              0x86CA
    EVAL_VERTEX_ATTRIB5_NV              0x86CB
    EVAL_VERTEX_ATTRIB6_NV              0x86CC
    EVAL_VERTEX_ATTRIB7_NV              0x86CD
    EVAL_VERTEX_ATTRIB8_NV              0x86CE
    EVAL_VERTEX_ATTRIB9_NV              0x86CF
    EVAL_VERTEX_ATTRIB10_NV             0x86D0
    EVAL_VERTEX_ATTRIB11_NV             0x86D1
    EVAL_VERTEX_ATTRIB12_NV             0x86D2
    EVAL_VERTEX_ATTRIB13_NV             0x86D3
    EVAL_VERTEX_ATTRIB14_NV             0x86D4
    EVAL_VERTEX_ATTRIB15_NV             0x86D5
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    MAX_MAP_TESSELLATION_NV             0x86D6
    MAX_RATIONAL_EVAL_ORDER_NV          0x86D7
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None.

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

-- NEW Section 5.7 "General Evaluators"

"General evaluators are similar to evaluators in that they can be used to evaluate polynomial and rational mappings, but general evaluators have several new features that the original evaluators do not. First, they support triangular surfaces in addition to (quadrilateral) tensor product surfaces. Second, the tessellation can be varied continuously as well as in integral steps. Finally, general evaluators can evaluate all vertex attributes, not just the vertex, color, normal, and texture coordinates.

Several elements of the original evaluators have been removed in the general evaluators interface. The general evaluators always evaluate four components in parallel, whereas the original evaluators might evaluate between 1 and 4 (see the "k" column in Table 5.1 on page 165). The original evaluators can map on an arbitrary domain and can map grids on an arbitrary region, whereas the general evaluators only use the [0,1] range. Support for 1D evaluators, an EvalCoord-style interface, and the "mode" parameter of EvalMesh* has also been removed from the general evaluators.

The command

```
void MapControlPointsNV(enum target, uint index, enum type,
                        sizei ustride, sizei vstride,
                        int uorder, int vorder, boolean packed,
                        const void *points);
```

specifies control points for a general evaluator map. target is the type of evaluator map and can be either EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. index is the number of the vertex attribute register the map will be used to evaluate for; these are the same indices used in the GL_NV_vertex_program extension. Table X.1 shows the relationship between these indices and the conventional per-vertex attributes for implementations that do not support GL_NV_vertex_program.

Vertex Attribute Conventional Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoordARB(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoordARB(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoordARB(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoordARB(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoordARB(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoordARB(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoordARB(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoordARB(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.1: Aliasing of vertex attributes with conventional per-vertex parameters.

type is either FLOAT or DOUBLE. ustride and vstride are the numbers of basic machine units (typically unsigned bytes) between control points in the u and v directions. uorder and vorder have the same meaning they do in the Map2{fd} command. The error INVALID_VALUE is generated if either uorder or vorder is less than one or greater than MAX_EVAL_ORDER. The error INVALID_OPERATION is generated if target is EVAL_TRIANGULAR_2D_NV and uorder is not equal to vorder.

points is a pointer to an array of control points. If target is EVAL_2D_NV, there are uorder*vorder control points in the array, and if it is EVAL_TRIANGULAR_2D_NV, there are uorder*(uorder+1)/2 points in the array. If packed is FALSE, control point i,j is located

```
(ustride)i + (vstride)j
```

basic machine units from points. If target is EVAL_2D_NV, i ranges from 0 to uorder-1, and j ranges from 0 to vorder-1. If target is EVAL_TRIANGULAR_2D_NV, i and j range from 0 to uorder-1, and i+j must be less than or equal to uorder-1.

If packed is TRUE and target is EVAL_2D_NV, control point i,j is located

```
(ustride)(j*uorder + i)
```

basic machine units from points. If packed is TRUE and target is EVAL_TRIANGULAR_2D_NV, control point i,j is located

```
(ustride)(j*uorder + i - j*(j-1)/2)
```

basic machine units from points.

The error INVALID_OPERATION is generated if index is 0, one of the control points' fourth components is not equal to 1, and either uorder of vorder is greater than MAX_RATIONAL_EVAL_ORDER_NV.

The evaluation of a 2D tensor product map is performed in the same way as for the original evaluators. The exact coordinates produced by the original evaluators may differ from those produced by the general evaluators, since different algorithms may be used.

A triangular map may be evaluated as follows. Let Ri,j be the 4-component vector for control point i,j and n be the degree of the patch (i.e. uorder-1). Then:

$$p_t(u,v) = \sum_{\substack{i,j \geq 0 \\ i+j \leq n}} \binom{n}{i}\binom{n-i}{j} u^i v^j (1-u-v)^{n-i-j} R_{i,j}$$

evaluates the point p_t(u,v) on the triangular patch at parameter values (u,v). (The notation on the left indicates "n choose i" and "n minus i choose j", i.e., binomial coefficients.)

The evaluation of any particular attribute can be enabled or disabled with Enable and Disable using one of the EVAL_VERTEX_ATTRIBi_NV constants.

If AUTO_NORMAL is enabled (see section 5.1), analytically computed normals are evaluated as well. The formula for the normal is the same as the one in section 5.1, except that the magnitude of the normals is undefined. These normals should be renormalized by enabling NORMALIZE, or by normalizing them in a vertex program. The w of the normal vertex attribute will always be 1.

The commands

```
void MapParameter{if}vNV(enum target, enum pname, T params);
```

can be used to specify the level of tessellation to evaluate, where target is EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV and pname is MAP_TESSELLATION_NV. If target is EVAL_2D_NV, params contains the four values [nu0,nu1,nv0,nv1], and if it is EVAL_TRIANGULAR_2D_NV, params contains the three values [n1,n2,n3]. The state for each target is independent of the other. These values are clamped to the range [1.0, MAX_MAP_TESSELLATION_NV].

The use of a fractional tessellation algorithm can be enabled or disabled with Enable and Disable using the EVAL_FRACTIONAL_TESSELLATION_NV constant. The fractional tessellation algorithm allows the tessellation to smoothly morph without popping as the tessellation parameters are varied by small amounts.

The command

```
  void EvalMapsNV(enum target, enum mode);
```

evaluates the currently enabled maps. target is either EVAL_2D_NV or EVAL_TRIANGULAR_2D and specifies which set of maps to evaluate. mode must be FILL. If EVAL_VERTEX_ATTRIB0_NV is not enabled, the error INVALID_OPERATION results.

If EVAL_FRACTIONAL_TESSELLATION_NV is disabled, tensor product maps are evaluated such that the boundaries of the mesh are divided into ceil(nu0) segments on the edge from (0,0) to (1,0), ceil(nu1) segments on the edge from (0,1) to (1,1), ceil(nv0) segments on the edge from (0,0) to (0,1), and ceil(nv1) segments on the edge from (1,0) to (1,1). These segments must be evaluated at equal spacings in (u,v) parameter space.

Triangular maps are evaluated such that the boundary of the mesh from (0,0) to (1,0) has ceil(n1) equally-spaced segments, the boundary from (1,0) to (0,1) has ceil(n2) equally-spaced segments, and the boundary from (0,1) to (0,0) has ceil(n3) equally-spaced segments.

If EVAL_FRACTIONAL_TESSELLATION_NV is enabled, each edge must be tessellated with no fewer the number of segments that would be used in the non- fractional case for any values of the tessellation parameters. Furthermore, the tessellation of each edge must vary smoothly with the parameters; that is, a small change in any or all of the parameters must cause a small change in the tessellation. Whenever a new vertex is introduced into the tessellation, it must be coincident with another vertex, and whenever a vertex is removed, it must have been coincident with a different vertex. The parameter-space position of any vertex must be a continuous function of the tessellation parameters.

The same minimum triangle requirements apply to fractional tessellations as to integral tessellations.

A tensor product patch must always be tessellated with no fewer than

```
    2 * ceil((nu0+nu1)/2) * ceil((nv0+nv1)/2)
```

triangles in total.

A triangular patch must always be tessellated with no fewer than

```
    ceil((n1+n2+n3)/3)^2
```

triangles in total.

If a triangle is formed by evaluating the maps at the three coordinates (u1,v1), (u2,v2), and (u3,v3), it must be true that

```
    (u3-u1)*(v2-v1) - (u2-u1)*(v3-v1) >= 0
```

to ensure that all triangles in a patch have a consistent orientation.

The current value of any vertex attribute for which the evaluation of a map is enabled becomes undefined after an EvalMapsNV command. If AUTO_NORMAL is enabled, the current normal becomes undefined as well.

If AUTO_NORMAL is enabled, the analytically computed normals take precedence over the currently enabled map for vertex attribute 2 (the normal).

To prevent cracks, certain rules must be established for performing the evaluations. The goal of these rules is to ensure that no matter what order control points are specified in and what the tessellation parameters are, so long as the control points on any edge exactly match the control points of an adjacent edge, and so long as the subdivision parameter for that edge is the same for the adjacent patch, there will be no cracking artifacts between the two patches. These requirements are completely independent of numerical precision. In particular, we will require that these shared vertices' positions be equal. Furthermore, there must be no cracking inside the geometry of any patch, and normals must not change in a discontinuous fashion so that there are no discontinuities in lighting or other effects that use the normal.

Let two patches share an edge of equal order (the order of an edge is the order of the patch in that direction for a tensor product patch, or the order of the patch for a triangular patch). Then this edge is said to be consistent if all the vertex control points (vertex attribute 0) are identical on each edge (although they may be specified in the opposite direction, or even in a different coordinate; one may an edge in the u direction, and one may be an edge in the v direction).

If an edge is consistent, and if each of the two patches are tessellated with identical tessellation parameters for that edge, then the vertex coordinates given to vertex processing must be exactly equal for each of the vertices.

The vertex coordinates given to vertex processing for the corner vertices of any patch must be exactly equal to the values of the corner control points of that patch, regardless of the patch's order, type, tessellation parameters, the state of the AUTO_NORMAL or EVAL_FRACTIONAL_TESSELLATION_NV enables, the control points, order, or enable of any other associated map, or any other OpenGL state.

The vertex coordinates and normals given to vertex processing for any vertex of a patch must be exactly equal each time that vertex is evaluated during the tessellation of a patch. Since each vertex is shared between several triangles in the patch, any variation in these coordinates and normals would result in cracks or lighting discontinuities.

The state required for the general evaluators consists of a bit indicating whether fractional tessellation is enabled or disabled, 16 bits indicating whether the evaluation of each vertex attribute is enabled or disabled, four floating-point map tessellation values for tensor product patches, three floating-point map tessellation values for triangular patches, and a map specification for a tensor product patch and a triangular patch for each vertex attribute. A map

specification consists of two integers indicating the order of the map in u and v and a two-dimensional array of vectors of four floating-point values containing the control points for that map. The initial state of fractional tessellation is disabled. The initial state of evaluation of vertex attribute 0 is enabled, and the initial state of evaluation for any other vertex attribute is disabled. The initial value of the tessellation parameters is 1.0. The initial order of each map specification is an order of 1 in both u and v and a single control point of [0,0,0,1]."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- NEW Section 6.1.13 "General Evaluator Queries"

"The commands

```
  void GetMapParameterivNV(enum target, enum pname, int *params);
  void GetMapParameterfvNV(enum target, enum pname, float *params);
```

obtain the parameters for a map target. target may be one of EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. pname must be MAP_TESSELLATION_NV. The map tessellation is placed in params.

The commands

```
  void GetMapAttribParameterivNV(enum target, uint index, enum pname,
                                 int *params);
  void GetMapAttribParameterfvNV(enum target, uint index, enum pname,
                                 float *params);
```

obtain parameters for a single map. target may be one of EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. index is the number of the vertex attribute register the map is used for evaluating. If pname is MAP_ATTRIB_U_ORDER_NV, the u order of the map is placed in params. If pname is MAP_ATTRIB_V_ORDER_NV, the v order of the map is placed in params.

The command

```
  void GetMapControlPointsNV(enum target, uint index, enum type,
                             sizei ustride, sizei vstride, boolean packed,
                             void *points);
```

obtains the control points of a map. target may be one of EVAL_2D_NV or EVAL_TRIANGULAR_2D_NV. index is the number of the vertex attribute register the map is used for evaluating. type is either FLOAT or DOUBLE. ustride and vstride are the numbers of basic machine units (typically unsigned bytes) between control points in the u and v directions. points is a pointer to an array where the control points are to be written. If target is EVAL_2D_NV, there are uorder*vorder control points in the array, and if it is EVAL_TRIANGULAR_2D_NV, there are uorder*(uorder+1)/2 points in the array. If packed is FALSE, control point i,j is located

```
    (ustride)i + (vstride)j
```

basic machine units from points. If packed is TRUE and target is

EVAL_2D_NV, control point i,j is located

    (ustride)(j*uorder + i)

basic machine units from points. If packed is TRUE and target is EVAL_TRIANGULAR_2D_NV, control point i,j is located

    (ustride)(j*uorder + i - j*(j-1)/2)

basic machine units from points. If target is EVAL_2D_NV, i ranges from 0 to uorder-1, and j ranges from 0 to vorder-1. If target is EVAL_TRIANGULAR_2D_NV, i and j range from 0 to uorder-1, and i+j must be less than or equal to uorder-1."

Additions to the GLX Specification

Nine new GL commands are added.

The following three rendering commands are sent to the sever as part of a glXRender request:

MapParameterivNV

2	12+4*n		rendering command length
2	????		rendering command opcode
4	ENUM		target
4	ENUM		pname
	0x86C2		GL_MAP_TESSELLATION_NV
		n=3	if (target == GL_EVAL_TRIANGULAR_2D_NV)
		n=4	if (target == GL_EVAL_2D_NV)
	else	n=0	command is erroneous
4*n	LISTofINT32		params

MapParameterfvNV

2	12+4*n		rendering command length
2	????		rendering command opcode
4	ENUM		target
4	ENUM		pname
	0x86C2		GL_MAP_TESSELLATION_NV
		n=3	if (target == GL_EVAL_TRIANGULAR_2D_NV)
		n=4	if (target == GL_EVAL_2D_NV)
	else	n=0	command is erroneous
4*n	LISTofFLOAT32		params

EvalMapsNV

2	12	rendering command length
2	????	rendering command opcode
4	ENUM	target
4	ENUM	mode

The following rendering command is potentially large and can be sent in a glXRender or glXRenderLarge request:

MapControlPointsNV

2	24+m	rendering command length
2	????	rendering command opcode
4	ENUM	target
4	CARD32	index
4	CARD32	type
4	INT32	uorder
4	INT32	vorder
m	(see below)	points

Determine m from the table below; n depends on the target. If the target is GL_EVAL_2D_NV, then n = uorder*vorder. If the target is GL_EVAL_TRIANGULAR_2D_NV, then n = uorder * (uorder+1)/2. The points data is packed such that when unpacked by the server,

```
the value of ustride is 16 for GL_FLOAT typed data and 32 for
GL_DOUBLE typed data.

 type       encoding of type  type of lists  m (bytes)
 ---------  ----------------  -------------  ---------
 GL_FLOAT   0x1406            LISTofFLOAT32  n*4
 GL_DOUBLE  0x140A            LISTofFLOAT64  n*8

If the command is encoded in a glXRenderLarge request, the command
opcode and command length fields above are expanded to 4 bytes each:

    4           28+m              rendering command length
    4           ????              rendering command opcode
```

The remaining five commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
GetMapParameterivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetMapParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

```
GetMapAttribParameterivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetMapAttribParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

```
GetMapControlPointsNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            type
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m
    4                           unused
    4           CARD32          uorder
    4           CARD32          vorder
    12                          unused

    if type == 0x1406 (GL_FLOAT) and target == 0x86C0
    (GL_EVAL_2D_NV), m = 4*uorder*vorder and the packed control
    points follow assuming ustride = 16

    m*4         LISTofFLOAT32   params

    if type == 0x140A (GL_DOUBLE) and target == 0x86C0
    (GL_EVAL_2D_NV), m = 4*uorder*vorder and the packed control
    points follow asssuming ustride = 32

    m*8         LISTofFLOAT64   params

    if type == 0x1406 (GL_FLOAT) and target == 0x86C1
    (GL_EVAL_TRIANGULAR_2D_NV), m = 4*uorder*(uorder+1)/2 and
    the packed control points follow assuming ustride = 16

    m*4         LISTofFLOAT32   params

    if type == 0x140A (GL_DOUBLE) and target == 0x86C1
    (GL_EVAL_TRIANGULAR_2D_NV), m = 4*uorder*(uorder+1)/2 and
    the packed control points follow asssuming ustride = 32

    m*8         LISTofFLOAT64   params

    otherwise m = 0 and nothing else follows.
```

Errors

The error INVALID_VALUE is generated if MapControlPointsNV, GetMapControlPointsNV, or GetMapAttribParameter{if}v is called where index is greater than 15.

The error INVALID_VALUE is generated if MapControlPointsNV or GetMapControlPointsNV is called where ustride or vstride is negative.

The error INVALID_VALUE is generated if MapControlPointsNV is called where uorder or vorder is less than one or greater than MAX_EVAL_ORDER.

The error INVALID_OPERATION is generated if MapControlPointsNV is called where target is EVAL_TRIANGULAR_2D_NV and uorder is not equal to vorder.

The error INVALID_OPERATION is generated if MapControlPointsNV is called where index is 0, one of the control points' fourth

components is not equal to 1, and either uorder of vorder is greater than MAX_RATIONAL_EVAL_ORDER_NV.

The error INVALID_OPERATION is generated if EvalMapsNV is called where EVAL_VERTEX_ATTRIB0_NV is disabled.

New State

(add to table 6.22, page 212)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
EVAL_FRACTIONAL_TESSELLATION_NV	B	IsEnabled	False	fractional tess. enable	5.7	eval/enable
EVAL_VERTEX_ATTRIBi_NV	Bx16	IsEnabled	True if i=0, false otherwise	attrib eval enable	5.7	eval/enable
EVAL_2D_NV	R4x16x8*x8*	GetMapControlPointsNV	[0,0,0,1]	control points	5.7	-
EVAL_TRIANGULAR_2D_NV	R4x16x8*x8*	GetMapControlPoints	[0,0,0,1]	control points	5.7	-
MAP_TESSELLATION_NV	R4,R3	GetMapParameter*NV	all 1.0	level of tessellation	5.7	eval
MAP_ATTRIB_U_ORDER_NV	Z8*x16x2	GetMapAttribParameter*NV	1	map order in U direction	5.7	-
MAP_ATTRIB_V_ORDER_NV	Z8*x16x2	GetMapAttribParameter*NV	1	map order in V direction	5.7	-

New Implementation Dependent State

(add to table 6.24/6.25, page 214)

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_MAP_TESSELLATION_NV	Z+	GetIntegerv	256	maximum level of tessellation	5.7	-
MAX_RATIONAL_EVAL_ORDER_NV	Z+	GetIntegerv	4	maximum order of rational surfaces	5.7	-

Revision History

none yet

Name

NV_fence

Name Strings

GL_NV_fence

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Shipping as of June 8, 2000 (version 1.0)

Version

April 13, 2001 (version 1.0)
$Id: //sw/main/docs/OpenGL/specs/GL_NV_fence.txt#11 $

Number

222

Dependencies

None

Overview

The goal of this extension is provide a finer granularity of synchronizing GL command completion than offered by standard OpenGL, which offers only two mechanism for synchronization: Flush and Finish. Since Flush merely assures the user that the commands complete in a finite (though undetermined) amount of time, it is, thus, of only modest utility. Finish, on the other hand, stalls CPU execution until all pending GL commands have completed. This extension offers a middle ground - the ability to "finish" a subset of the command stream, and the ability to determine whether a given command has completed or not.

This extension introduces the concept of a "fence" to the OpenGL command stream. Once the fence is inserted into the command stream, it can be queried for a given condition - typically, its completion. Moreover, the application may also request a partial Finish -- that is, all commands prior to the fence will be forced to complete until control is returned to the calling process. These new mechanisms allow for synchronization between the host CPU and the GPU, which may be accessing the same resources (typically memory).

This extension is useful in conjunction with NV_vertex_array_range
to determine when vertex information has been pulled from the
vertex array range. Once a fence has been tested TRUE or finished,
all vertex indices issued before the fence must have been pulled.
This ensures that the vertex data memory corresponding to the issued
vertex indices can be safely modified (assuming no other outstanding
vertex indices are issued subsequent to the fence).

Issues

Do we need an IsFenceNV command?

RESOLUTION: Yes. Not sure who would use this, but it's in there.
Semantics currently follow the texture object definition --
that is, calling IsFenceNV before SetFenceNV will return FALSE.

Are the fences sharable between multiple contexts?

RESOLUTION: No.

Potentially this could change with a subsequent extension.

What other conditions will be supported?

Only ALL_COMPLETED_NV will be supported initially. Future extensions
may wish to implement additional fence conditions.

What is the relative performance of the calls?

Execution of a SetFenceNV is not free, but will not trigger a
Flush or Finish.

Is the TestFenceNV call really necessary? How often would this be used compared to the FinishFenceNV call (which also flushes to ensure this happens in finite time)?

It is conceivable that a user may use TestFenceNV to decide
which portion of memory should be used next without stalling
the CPU. An example of this would be a scenario where a single
AGP buffer is used for both static (unchanged for multiple frames)
and dynamic (changed every frame) data. If the user has written
dynamic data to all banks dedicated to dynamic data, and still
has more dynamic objects to write, the user would first want to
check if the first dynamic object has completed, before writing
into the buffer. If the object has not completed, instead of
stalling the CPU with a FinishFenceNV call, it would possibly
be better to start overwriting static objects instead.

What should happen if TestFenceNV is called for a name before SetFenceNV is called?

We should probably generate an error, and return TRUE.
This follows the semantics for texture object names before
they are bound, in that they acquire their state upon binding.
We will arbitrarily return TRUE for consistency.

What should happen if FinishFenceNV is called for a name before SetFenceNV is called?

RESOLUTION: Generate an INVALID_OPERATION error because the fence id does not exist yet. SetFenceNV must be called to create a fence.

Do we need a mechanism to query which condition a given fence was set with?

RESOLUTION: Yes, use glGetFenceivNV with FENCE_CONDITION_NV.

Should we allow these commands to be compiled within display list? Which ones? How about within Begin/End pairs?

RESOLUTION: DeleteFencesNV, GenFencesNV, TestFenceNV, and IsFenceNV are executed immediately while FinishFenceNV and SetFenceNV are compiled. Do not allow any of these commands within Begin/End pairs.

Can fences be used as a form of performance monitoring?

Yes, with some caveats. By setting and testing or finishing fences, developers can measure the GPU latency for completing GL operations. For example, developers might do the following:

```
start = getCurrentTime();
updateTextures();
glSetFenceNV(TEXTURE_LOAD_FENCE, GL_ALL_COMPLETED_NV);
drawBackground();
glSetFenceNV(DRAW_BACKGROUND_FENCE, GL_ALL_COMPLETED_NV);
drawCharacters();
glSetFenceNV(DRAW_CHARACTERS_FENCE, GL_ALL_COMPLETED_NV);

glFinishFenceNV(TEXTURE_LOAD_FENCE);
textureLoadEnd = getCurrentTime();

glFinishFenceNV(DRAW_BACKGROUND_FENCE);
drawBackgroundEnd = getCurrentTime();

glFinishFenceNV(DRAW_CHARACTERS_FENCE);
drawCharactersEnd = getCurrentTime();

printf("texture load time = %d\n", textureLoadEnd - start);
printf("draw background time = %d\n", drawBackgroundEnd - textureLoadEnd);
printf("draw characters time = %d\n", drawCharacters - drawBackgroundEnd);
```

Note that there is a small amount of overhead associated with inserting each fence into the GL command stream. Each fence causes the GL command stream to momentarily idle (idling the entire GPU pipeline). The significance of this idling should be small if there are a small nuber of fences and large amount of intervening commands.

If the time between two fences is zero or very near zero, it probably means that a GPU-CPU synchronization such as a glFinish probably occurred. A glFinish is an explicit GPU-CPU synchronization, but sometimes implicit GPU-CPU synchronizations are performed by the driver.

New Procedures and Functions

```
    void GenFencesNV(sizei n, uint *fences);

    void DeleteFencesNV(sizei n, const uint *fences);

    void SetFenceNV(uint fence, enum condition);

    boolean TestFenceNV(uint fence);

    void FinishFenceNV(uint fence);

    boolean IsFenceNV(uint fence);

    void GetFenceivNV(uint fence, enum pname, int *params);
```

New Tokens

Accepted by the <condition> parameter of SetFenceNV:

```
    ALL_COMPLETED_NV                    0x84F2
```

Accepted by the <pname> parameter of GetFenceivNV:

```
    FENCE_STATUS_NV                     0x84F3
    FENCE_CONDITION_NV                  0x84F4
```

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists"

"DeleteFencesNV, GenFencesNV, GetFenceivNV, TestFenceNV, and IsFenceNV are not complied into display lists but are executed immediately."

After the discussion of Flush and Finish (Section 5.5) add a description of the fence operations:

"5.X Fences

The command

```
   void SetFenceNV(uint fence, enum condition);
```

sets a fence within the GL command stream, and assigns the fence a status of FALSE and a condition as set by the condition argument. The condition argument must be ALL_COMPLETED_NV. Once the fence's condition is satisfied within the command stream, its state is changed to TRUE. For a condition of ALL_COMPLETED_NV, this is completion of the fence command. No other state is affected by execution of the fence command. A fence's state can be queried by calling the command

```
  boolean TestFenceNV(uint fence);
```

The command

```
  void FinishFenceNV(uint fence);
```

forces all GL commands prior to the fence to satisfy the condition set within SetFenceNV, which, in this spec, is always completion. FinishFenceNV does not return until all effects from these commands on GL client and server state and the framebuffer are fully realized.

The fence must first be created before it can be used. The command

```
  void GenFencesNV(sizei n, uint *fences);
```

returns n previously unused fence names in fences. These names are marked as used, for the purposes of GenFencesNV only, but acquire boolean state only when they have been set.

Fences are deleted by calling

```
  void DeleteFencesNV(sizei n, const uint *fences);
```

fences contains n names of fences to be deleted. After a fence is deleted, it has no state, and its name is again unused. Unused names in fences are silently ignored.

If the fence passed to TestFenceNV or FinishFenceNV is not the name of a fence, the error INVALID_OPERATION is generated. In this case, TestFenceNV will return TRUE, for the sake of consistency.

State must be maintained to indicate which fence integers are currently used or set. In the initial state, no indices are in use. When a fence integer is set, the condition and status of the fence are also maintained. The status is a boolean. The condition is the value last set as the condition by SetFenceNV.

Once the status of a fence has been finished (via FinishFenceNV) or tested and the returned status is TRUE (via either TestFenceNV or GetFenceivNV querying the FENCE_STATUS_NV), the status remains TRUE until the next SetFenceNV of the fence."

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

Insert new section after Section 6.1.10 "Minmax Query"

"6.1.11 Fence Query

The command

```
  boolean IsFenceNV(uint fence);
```

return TRUE if texture is the name of a fence. If fence is not the name of a fence, or if an error condition occurs, IsFenceNV returns FALSE. A name returned by GenFencesNV, but not yet set via SetFenceNV, is not the name of a fence.

The command

```
  void GetFenceivNV(uint fence, enum pname, int *params)
```

obtains the indicated fence state for the specified fence in the array params. pname must be either FENCE_STATUS_NV or FENCE_CONDITION_NV. The INVALID_OPERATION error is generated if the named fence does not exist."

Additions to the GLX Specification

None

GLX Protocol

Seven new GL commands are added.

The following two rendering commands are sent to the sever as part of a glXRender request:

```
SetFenceNV
    2           12              rendering command length
    2           4143            rendering command opcode
    4           CARD32          fence
    4           CARD32          condition

FinishFenceNV
    2           8               rendering command length
    2           4144            rendering command opcode
    4           CARD32          fence
```

The remaining five commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
DeleteFencesNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4+n             request length
    4           1276            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
    n*4         LISTofCARD32    fences
```

```
GenFencesNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           1277            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           n               reply length
    24                          unused
    n*4         LISTofCARD322   fences

IsFenceNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           1278            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           0               reply length
    4           BOOL32          return value
    20                          unused
    1           1               reply

TestFenceNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           1279            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           fence
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           0               reply length
    4           BOOL32          return value
    20                          unused
```

```
    GetFenceivNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           5               request length
        4           1280            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           fence
        4           CARD32          pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m=(n==1?0:n)
        4                           unused
        4           CARD32          n

        if (n=1) this follows:

        4           INT32           params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofINT32     params
```

Note that polling with TestFenceNV when using indirect GLX rendering will be considerably less efficient than using FinishFenceNV because TestFenceNV is an X protocol round-trip while FinishFenceNV synchronizes the GLX command stream without an X protocol round-trip.

Errors

INVALID_VALUE is generated if GenFencesNV parameter <n> is negative.

INVALID_VALUE is generated if DeleteFencesNV parameter <n> is negative.

INVALID_OPERATION is generated if the fence used in TestFenceNV or FinishFenceNV is not the name of a fence.

INVALID_ENUM is generated if the condition used in SetFenceNV is not ALL_COMPLETED_NV.

INVALID_OPERATION is generated if any of the commands defined in this extension is executed between the execution of Begin and the corresponding execution of End.

INVALID_OPERATION is generated if the named fence in GetFenceivNV does not exist.

INVALID_VALUE is generated if DeleteFencesNV or GenFencesNV are called where n is negative.

New State

Table 6.X. Fence Objects.

Get value	Type	Get command	Initial value	Description	Section	Attribute
FENCE_STATUS_NV	B	GetFenceivNV	determined by 1st SetFenceNV	Fence status	5.X	-
FENCE_CONDITION_NV	Z1	GetFenceivNV	determined by 1st SetFenceNV	Fence condition	5.X	-

New Implementation Dependent State

None

GeForce Implementation Details

This section describes implementation-defined limits for GeForce:

SetFenceNV calls are not free. They should be used prudently, and a "good number" of commands should be sent between calls to SetFenceNV. Each fence insertion will cause the GPU's command processing to go momentarily idle. Testing or finishing a fence may require an one or more somewhat expensive uncached reads.

Do not leave a fence untested or unfinished for an extremely large interval of intervening fences. If more than approximately 2 billion (specifically 2^31-1) intervening fences are inserted into the GL command stream before a fence is tested or finished, said fence may indicate an incorrect status. Note that certain GL operations involving display lists, compiled vertex arrays, and textures may insert fences implicitly for internal driver use.

In practice, this limitation is unlikely to be a practical limitation if fences are finished or tested within a few frames of their insertion into the GL command stream.

Revision History

November 13, 2000 - GLX enumerant values assigned

Name

NV_fog_distance

Name Strings

GL_NV_fog_distance

Notice

Copyright NVIDIA Corporation, 1999, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Shipping (version 1.0)

Version

NVIDIA Date: January 18, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_fog_distance.txt#14 $

Number

192

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

Ideally, the fog distance (used to compute the fog factor as described in Section 3.10) should be computed as the per-fragment Euclidean distance to the fragment center from the eye. In practice, implementations "may choose to approximate the eye-coordinate distance from the eye to each fragment center by abs(ze). Further, [the fog factor] f need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are."

This extension provides the application specific control over how OpenGL computes the distance used in computing the fog factor.

The extension supports three fog distance modes: "eye plane absolute", where the fog distance is the absolute planar distance from the eye plane (i.e., OpenGL's standard implementation allowance as cited above); "eye plane", where the fog distance is the signed planar distance from the eye plane; and "eye radial", where the fog distance is computed as a Euclidean distance. In the case of the eye radial fog distance mode, the distance may be computed per-vertex and then interpolated per-fragment.

The intent of this extension is to provide applications with better

control over the tradeoff between performance and fog quality. The "eye planar" modes (signed or absolute) are straightforward to implement with good performance, but scenes are consistently under-fogged at the edges of the field of view. The "eye radial" mode can provide for more accurate fog at the edges of the field of view, but this assumes that either the eye radial fog distance is computed per-fragment, or if the fog distance is computed per-vertex and then interpolated per-fragment, then the scene must be sufficiently tessellated.

Issues

What should the default state be?

IMPLEMENTATION DEPENDENT.

The EYE_PLANE_ABSOLUTE_NV mode is the most consistent with the way most current OpenGL implementations are implemented without this extension, but because this extension provides specific control over a capability that core OpenGL is intentionally lax about, the default fog distance mode is left implementation dependent. We would not want a future OpenGL implementation that supports fast EYE_RADIAL_NV fog distance to be stuck using something less.

Advice: If an implementation can provide fast per-pixel EYE_RADIAL_NV support, then EYE_RADIAL_NV is the ideal default, but if not, then EYE_PLANE_ABSOLUTE_NV is the most reasonable default mode.

How does this extension interact with the EXT_fog_coord extension?

If FOG_COORDINATE_SOURCE_EXT is set to FOG_COORDINATE_EXT, then the fog distance mode is ignored. However, the fog distance mode is used when the FOG_COORDINATE_SOURCE_EXT is set to FRAGMENT_DEPTH_EXT. Essentially, when the EXT_fog_coord functionality is enabled, the fog distance is supplied by the user-supplied fog-coordinate so no automatic fog distance computation is performed.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of Fogf, Fogi, Fogfv, Fogiv, GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

FOG_DISTANCE_MODE_NV	0x855A

When the <pname> parameter of Fogf, Fogi, Foggv, and Fogiv, is FOG_DISTANCE_MODE_NV, then the value of <param> or the value pointed to by <params> may be:

EYE_RADIAL_NV	0x855B
EYE_PLANE	
EYE_PLANE_ABSOLUTE_NV	0x855C

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.10 "Fog"

Add to the end of the 3rd paragraph:

"If pname is FOG_DISTANCE_MODE_NV, then param must be, or params must point to an integer that is one of the symbolic constants EYE_PLANE_ABSOLUTE_NV, EYE_PLANE, or EYE_RADIAL_NV and this symbolic constant determines how the fog distance should be computed."

Replace the 4th paragraph beginning "An implementation may choose to approximate ..." with:

"When the fog distance mode is EYE_PLANE_ABSOLUTE_NV, the fog distance z is approximated by abs(ze) [where ze is the Z component of the fragment's eye position]. When the fog distance mode is EYE_PLANE, the fog distance z is approximated by ze. When the fog distance mode is EYE_RADIAL_NV, the fog distance z is computed as the Euclidean distance from the center of the fragment in eye coordinates to the eye position. Specifically:

```
z  =  sqrt( xe*xe + ye*ye + ze*ze );
```

In the EYE_RADIAL_NV fog distance mode, the Euclidean distance is permitted to be computed per-vertex, and then interpolated per-fragment."

Change the last paragraph to read:

"The state required for fog consists of a three valued integer to select the fog equation, a three valued integer to select the fog distance mode, three floating-point values d, e, and s, and RGBA fog color and a fog color index, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_MODE is EXP, FOG_DISTANCE_NV is implementation defined, d = 1.0, e = 1.0, and s = 0.0; Cf = (0,0,0,0) and if = 0."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when Fog is called with a <pname> of FOG_DISTANCE_MODE_NV and the value of <param> or what is pointed to by <params> is not one of EYE_PLANE_ABSOLUTE_NV, EYE_PLANE, or EYE_RADIAL_NV.

New State

(table 6.8, p198) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_DISTANCE_MODE_NV	Z3	GetIntegerv	implementation dependent	Determines how fog distance is computed	3.10	fog

New Implementation State

None

Name

NV_light_max_exponent

Name Strings

GL_NV_light_max_exponent

Notice

Copyright NVIDIA Corporation, 1999, 2000.

Version

May 20, 1999

Number

189

Dependencies

None

Overview

Default OpenGL does not permit a shininess or spot exponent over 128.0. This extension permits implementations to support and advertise a maximum shininess and spot exponent beyond 128.0.

Note that extremely high exponents for shininess and/or spot light cutoff will require sufficiently high tessellation for acceptable lighting results.

Paul Deifenbach's thesis suggests that higher exponents are necessary to approximate BRDFs with per-vertex ligthing and multiple passes.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MAX_SHININESS_NV	0x8504
MAX_SPOT_EXPONENT_NV	0x8505

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

In Table 2.7, change the srm range entry to read:

"(range: [0.0, value of MAX_SHININESS_NV])"

In Table 2.7, change the srli range entry to read:

"(range: [0.0, value of MAX_SPOT_EXPONENT_NV])"

Add to the end of the second paragraph in Section 2.13.2:

"The values of MAX_SHININESS_NV and MAX_SPOT_EXPONENT_NV are implementation dependent, but must be equal or greater than 128."

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_VALUE is generated by Material if enum is SHININESS and the shininess param is greater than the MAX_SHININESS_NV.

INVALID_VALUE is generated by Material if enum is SPOT_EXPONENT and the shininess param is greater than the MAX_SPOT_EXPONENT_NV.

New State

None.

New Implementation Dependent State

(table 6.24, p214) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_SHININESS_NV	Z+	GetIntegerv	128	Maximum shininess for specular lighting	2.13.2	-
MAX_SPOT_EXPONENT_NV	Z+	GetIntegerv	128	Maximum exponent for spot lights	2.13.2	-

NVIDIA Implementation Details

NVIDIA's Release 4 drivers incorrectly and accidently advertised this extension with an "EXT" prefix instead of an "NV" prefix. Release 5 and later drivers correctly advertise this extension with an "NV" extension.

Revision History

5/20/00 - earlier versions of this specification had the incorrect enumerant values which did not match NVIDIA's driver implementation.

Name

NV_multisample_filter_hint

Name Strings

GL_NV_multisample_filter_hint

Notice

Copyright NVIDIA Corporation, 2001.

IP Status

NVIDIA Proprietary.

Status

Shipping, May 2001

Version

NVIDIA Date: May 16, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_multisample_filter_hint.txt#2 $

Number

??

Dependencies

Written based on the OpenGL 1.2.1 specification.

Requires ARB_multisample.

Overview

OpenGL multisampling typically assumes that the samples of a given pixel are weighted uniformly and averaged to compute the pixel's resolved color. This extension provides a hint that permits implementations to provide an alternative method of resolving the color of multisampled pixels.

As an example of such an alternative method, NVIDIA's GeForce3 GPU provides a technique known as Quincunx filtering. This technique is used in two-sample multisampling, but it blends the pixel's two samples and three additional samples from adjacent pixels. The sample pattern is analogous to the 5 pattern on a die. The quality of this technique is widely regarded as comparable to 4 sample multisampling.

Issues

Is the glHint mechanism the right mechanism to expose this functionality?

RESOLUTION: Yes. Multisample filtering quality is subject to the kinds of variations that the glHint was intended to control.

Arguably, the glHint mechanism only provides two definite settings: GL_FASTEST and GL_NICEST while there may be many different techniques for controlling multisample filtering quality. We expect hardware to support only one or two techniques rather than a multitude of nearly indistinguishable sampling techniques.

When does changing the multisampling filter hint take effect?

RESOLUTION: It may not be until the next swap buffers or glClear operation that the multisample hint actually takes effect. This may be implementation dependent.

What is the meaning of GL_DONT_CARE for the multisample hint?

RESOLUTION: By default, NVIDIA expects to treat GL_DONT_CARE the same as GL_FASTEST. However, the meaning of GL_DONT_CARE for this hint may be subject to a registry (or environment) setting, possibly settable through a control panel.

Does GL_NICEST require Quincunx filtering?

RESOLUTION: No. NVIDIA's GeForce3 Quincunx filtering is one possible technique that may be used to implement the GL_NICEST setting, but future GPUs may use other techniques.

Can the meaning of the multisample hint vary depending on the number of samples of the drawable?

RESOLUTION: Yes.

The following describes how GeForce3 uses the multisample hint:

When using 2-sample multisampling with GeForce3, the multisample filter hint affects multisample filtering as follows: GL_NICEST uses 5-tap Quincunx multisample filtering while GL_FASTEST uses standard even-weighted 2-tap multisample filtering of the pixel's 2 samples.

When using 4-sample multisampling with GeForce3, the multisample filter hint affects multisample filtering as follows: GL_NICEST uses 9-tap 3x3 multisample filtering while GL_FASTEST uses standard even-weighted 4-tap multisample filtering of the pixel's 4 samples.

What is the difference between a "tap" and a "sample"?

In the context of multisample filtering, a sample is a subpixel frame buffer sample containing color, depth, and stencil information. A tap is a source of data for filtering. Typically, samples are filtered by evenly weighting all the samples belonging to a pixel. In this case, the number of taps for the filter is equal to the number of samples for the pixel. In other filtering schemes, the number of taps and samples may not be equal (and potentially not evenly weighted as well). For example, GeForce3's quincunx filtering uses 5 taps even though each pixel has only 2 multisample samples. Three of the five taps source samples outside the pixel's footprint of two samples.

Should the multisample filtering technique be determined by the visual/PFD rather than OpenGL rendering context state?

RESOLUTION: No. The number of multisample samples per pixel that a window has is a property of the visual/PFD, but the filtering technique does not have to be defined up-front at when the pixel format is set.

While not quite consistent with the way ARB_multisample is specified, NVIDIA uses the SwapBuffers operation as a trigger for downsampling multisample sample buffers (other operations such as glReadPixels also trigger downsampling). But a SwapBuffers operation can be requested without a current OpenGL rendering context. What happens when a SwapBuffers operation is performed with no current OpenGL rendering context?

RESOLUTION: The multisample filter hint is treated as GL_DONT_CARE in this case. Applications that want the multisample filter hint to apply to their BufferSwap operation should perform the BufferSwap operation while bound to an OpenGL rendering context.

New Procedures and Functions

None

New Tokens

Accepted by the <target> parameter of Hint and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MULTISAMPLE_FILTER_HINT_NV	0x8534

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

-- Section 5.6 "Hints"

Replace the description of hint targets in the first paragraph with:

"target may be one of PERSPECTIVE_HINT, indicating the desired quality of parameter interpolation; POINT_SMOOTH_HINT, indicating the desired sampling quality of points; LINE_SMOOTH_HINT, indicating the desired sampling quality of lines; POLYGON_SMOOTH_HINT, indicating the desired sampling quality of polygons; FOG_HINT, indicating whether fog calculations are done per pixel or per vertex; and

MULTISAMPLE_FILTER_HINT, indicating the desired quality of multisample filtering. The MULTISAMPLE_FILTER_HINT is ignored if the frame buffer has no multisample samples. When NICEST (or possibly DONT_CARE) multisample filtering is requested and the frame buffer supports multisampling, the multisample filter pattern may involve samples outside the pixel's sample set. The exact NICEST (or possibly DONT_CARE) multisample filtering technique used is implementation dependent and may vary with the number of multisample samples supported."

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX, WGL, and AGL Specification

Add the following to the description of what happens at SwapBuffers time.

"When a SwapBuffers operation is performed by a thread without a current OpenGL rendering context and the target drawable to be swapped is multisampled, any multisample filtering operation that occurs should be done as if the GL_MULTISAMPLE_FILTER_HINT value is set to GL_DONT_CARE."

GLX Protocol

None

Errors

None

New State

(table 6.23, p213) add the following entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
MULTISAMPLE_FILTER_HINT_NV	Z3	GetIntegerv	DONT_CARE	Multisample filter quality hint	5.6	hint

Revision History

None

Name

NV_packed_depth_stencil

Name Strings

GL_NV_packed_depth_stencil

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Version

NVIDIA Date: January 18, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_packed_depth_stencil.txt#6 $

Number

??

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

SGIX_depth_texture affects the definition of this extension.

Overview

Many OpenGL implementations have chosen to interleave the depth and stencil buffers into one buffer, often with 24 bits of depth precision and 8 bits of stencil data. 32 bits is more than is needed for the depth buffer much of the time; a 24-bit depth buffer, on the other hand, requires that reads and writes of depth data be unaligned with respect to power-of-two boundaries. On the other hand, 8 bits of stencil data is more than sufficient for most applications, so it is only natural to pack the two buffers into a single buffer with both depth and stencil data. OpenGL never provides direct access to the buffers, so the OpenGL implementation can provide an interface to applications where it appears the one merged buffer is composed of two logical buffers.

One disadvantage of this scheme is that OpenGL lacks any means by which this packed data can be handled efficiently. For example, when an application reads from the 24-bit depth buffer, using the type GL_UNSIGNED_SHORT will lose 8 bits of data, while GL_UNSIGNED_INT has 8 too many. Both require expensive format conversion operations. A 24-bit format would be no more suitable, because it would also suffer from the unaligned memory accesses that made the standalone 24-bit depth buffer an unattractive proposition in the first place.

Many applications, such as parallel rendering applications, may also wish to draw to or read back from both the depth and stencil buffers at the same time. Currently this requires two separate operations, reducing performance. Since the buffers are interleaved, drawing to or reading from both should be no more expensive than using just one; in some cases, it may even be cheaper.

This extension provides a new data format, GL_DEPTH_STENCIL_NV, that can be used with the glDrawPixels, glReadPixels, and glCopyPixels commands, as well as a packed data type, GL_UNSIGNED_INT_24_8_NV, that is meant to be used with GL_DEPTH_STENCIL_NV. No other formats are supported with GL_DEPTH_STENCIL_NV. If SGIX_depth_texture is supported, GL_DEPTH_STENCIL_NV/GL_UNSIGNED_INT_24_8_NV data can also be used for textures; this provides a more efficient way to supply data for a 24-bit depth texture.

GL_DEPTH_STENCIL_NV data, when passed through the pixel path, undergoes both depth and stencil operations. The depth data is scaled and biased by the current GL_DEPTH_SCALE and GL_DEPTH_BIAS, while the stencil data is shifted and offset by the current GL_INDEX_SHIFT and GL_INDEX_OFFSET. The stencil data is also put through the stencil-to-stencil pixel map.

glDrawPixels of GL_DEPTH_STENCIL_NV data operates similarly to that of GL_STENCIL_INDEX data, bypassing the OpenGL fragment pipeline entirely, unlike the treatment of GL_DEPTH_COMPONENT data. The stencil and depth masks are applied, as are the pixel ownership and scissor tests, but all other operations are skipped.

glReadPixels of GL_DEPTH_STENCIL_NV data reads back a rectangle from both the depth and stencil buffers.

glCopyPixels of GL_DEPTH_STENCIL_NV data copies a rectangle from both the depth and stencil buffers. Like glDrawPixels, it applies both the stencil and depth masks but skips the remainder of the OpenGL fragment pipeline.

glTex[Sub]Image[1,2,3]D of GL_DEPTH_STENCIL_NV data loads depth data into a depth texture. glGetTexImage of GL_DEPTH_STENCIL_NV data can be used to retrieve depth data from a depth texture.

Issues

* Depth data has a format of GL_DEPTH_COMPONENT, and stencil data has a format of GL_STENCIL_INDEX. So shouldn't the enumerant be called GL_DEPTH_COMPONENT_STENCIL_INDEX_NV?

  RESOLVED: No, this is fairly clumsy.

* Should we support CopyPixels?

  RESOLVED: Yes. Right now copying stencil data means masking off just the stencil bits, while copying depth data has strange unintended consequences (fragment operations) and is difficult to implement. It is easy and useful to add CopyPixels support.

* Should we support textures?

  RESOLVED: Yes. 24-bit depth textures have no good format without this extension.

* Should the depth/stencil format support other standard types, like GL_FLOAT or GL_UNSIGNED_INT?

  RESOLVED: No, this extension is designed to be minimalist. Supporting more types gains little because the new types will just require data format conversions. Our goal is to match the native format of the buffer, not add broad new classes of functionality.

* Should the 24/8 format be supported for other formats, such as LUMINANCE_ALPHA? Should we support an 8/24 reversed format as well?

  RESOLVED: No and no, this adds implementation burden and gains us little, if anything.

* Does anything need to be written in the spec on the topic of using GL_DEPTH_STENCIL_NV formats for glTexImage* or glGetTexImage?

  RESOLVED: No. Since the SGIX_depth_texture extension spec was never actually written (the additions to Section 3 are "XXX - lots" and a few brief notes on how it's intended to work), it's impossible to write what would essentially be amendments to that spec.

  However, it is worthwhile to mention here the intended behavior. When downloading into a depth component texture, the stencil indices are ignored, and when retrieving a depth component texture, the stencil indices obtained from the texture are undefined.

* Should anything be said about performance?

  RESOLVED: No, not in the spec. However, common sense should apply. Apps should probably check that GL_DEPTH_BITS is 24 and that GL_STENCIL_BITS is 8 before using either the new DrawPixels or ReadPixels formats. CopyPixels is probably safe regardless of the size of either buffer. The 24/8 format should probably only be used with 24-bit depth textures.

New Procedures and Functions

None.

New Tokens

Accepted by the <format> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, and GetTexImage, and by the <type> parameter of CopyPixels:

DEPTH_STENCIL_NV	0x84F9

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, and GetTexImage:

UNSIGNED_INT_24_8_NV	0x84FA

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Update the first paragraph on page 90 to say:

"... If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL_NV, then the error INVALID_OPERATION occurs. If <type> is BITMAP and <format> is not COLOR_INDEX or STENCIL_INDEX then the error INVALID_ENUM occurs. If <format> is DEPTH_STENCIL_NV and <type> is not UNSIGNED_INT_24_8_NV then the error INVALID_ENUM occurs. Some additional constraints on the combinations of <format> and <type> values that are accepted is discussed below."

Add a row to Table 3.5 (page 91):

type Parameter	GL Type	Special
...	...	...
UNSIGNED_INT_2_10_10_10_REV	uint	Yes
UNSIGNED_INT_24_8_NV	uint	Yes

Add a row to Table 3.6 (page 92):

Format Name	Element Meaning and Order	Target Buffer
...	...	...
DEPTH_COMPONENT	Depth	Depth
DEPTH_STENCIL_NV	Depth and Stencil Index	Depth and Stencil
...	...	...

Add a row to Table 3.8 (page 94):

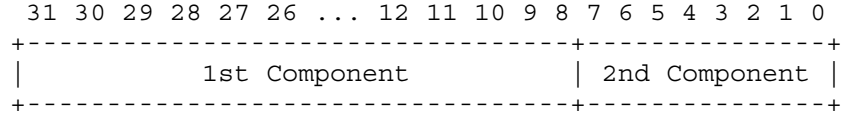
type Parameter	GL Type	Components	Pixel Formats
...	...	...	...
UNSIGNED_INT_2_10_10_10_REV	uint	4	RGBA,BGRA
UNSIGNED_INT_24_8_NV	uint	2	DEPTH_STENCIL_NV

Update the last paragraph on page 93 to say:

"Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, ..., UNSIGNED_INT_2_10_10_10_REV, or UNSIGNED_INT_24_8_NV is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type."

Add the following diagram to Table 3.11 (page 97):

UNSIGNED_INT_24_8_NV



Add a row to Table 3.12 (page 98):

Format	1st	2nd	3rd	4th
...	...	...	...	...
BGRA	blue	green	red	alpha
DEPTH_STENCIL_NV	depth	stencil	N/A	N/A

Add the following paragraph to the end of the section "Conversion to floating-point" (page 99):

"For groups of components that contain both standard components and index elements, such as DEPTH_STENCIL_NV, the index elements are not converted."

Update the last paragraph in the section "Conversion to Fragments" (page 100) to say:

"... Groups arising from DrawPixels with a <format> of STENCIL_INDEX or DEPTH_STENCIL_NV are treated specially and are described in section 4.3.1."

Update the first paragraph of section 3.6.5 "Pixel Transfer Operations" (pages 100-101) to say:

"The GL defines five kinds of pixel groups:

1. RGBA component: Each group comprises four color components: red, green, blue, and alpha.
2. Depth component: Each group comprises a single depth component.
3. Color index: Each group comprises a single color index.
4. Stencil index: Each group comprises a single stencil index.
5. Depth/stencil: Each group comprises a depth component and a stencil index."

Update the first paragraph in the section "Arithmetic on Components" (page 101) to say:

"This step applies only to RGBA component and depth component groups and the depth components in depth/stencil groups. ..."

Update the first paragraph in the section "Arithmetic on Indices" (page 101) to say:

"This step applies only to color index and stencil index groups and the stencil indices in depth/stencil groups. ..."

Update the first paragraph in the section "Stencil Index Lookup" (page 102) to say:

"This step applies only to stencil index groups and the stencil indices in depth/stencil groups. ..."

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

Replace section 4.3.1 "Writing to the Stencil Buffer" (page 156) with the following:

"4.3.1 Writing to the Stencil Buffer or to the Depth and Stencil Buffers

The operation of DrawPixels was described in section 3.6.4, except if the <format> argument was STENCIL_INDEX or DEPTH_STENCIL_NV. In this case, all operations described for DrawPixels take place, but window (x,y) coordinates, each with the corresponding stencil index or depth value and stencil index, are produced in lieu of fragments. Each coordinate-data pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current setting of StencilMask, and if a depth component is present, if the setting of DepthMask is not FALSE, it is also written to the framebuffer; the setting of DepthTest is ignored.

The error INVALID_OPERATION results if there is no stencil buffer, or if the <format> argument was DEPTH_STENCIL_NV, if there is no depth buffer."

Add the following paragraph after the second paragraph of the section "Obtaining Pixels from the Framebuffer" (page 158):

"If the <format> is DEPTH_STENCIL_NV, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, the error INVALID_OPERATION occurs. If the <type> parameter is not UNSIGNED_INT_24_8_NV, the error INVALID_ENUM occurs."

Update the third paragraph on page 159 to say:

"If the GL is in RGBA mode, and <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE_ALPHA, then red, green, blue, and alpha values are obtained from the framebuffer

Update the first sentence of the section "Conversion of RGBA values" (page 159) to say:

"This step applies only if the GL is in RGBA mode, and then only if <format> is neither STENCIL_INDEX, DEPTH_COMPONENT, nor DEPTH_STENCIL_NV."

Update the section "Conversion of Depth values" (page 159) to say:

"This step applies only if <format> is DEPTH_COMPONENT or DEPTH_STENCIL_NV. Each element taken from the depth buffer is taken to be a fixed-point value in [0,1] with m bits, where m is the number of bits in the depth buffer (see section 2.10.1)."

Add a row to Table 4.6 (page 160):

type Parameter	Index Mask
...	...
INT	2^31-1
UNSIGNED_INT_24_8_NV	2^8-1

Add the following paragraph to the end of the section "Final Conversion" (page 160):

"For a depth/stencil pair, first the depth component is clamped to [0,1]. Then the appropriate conversion formula from Table 4.7 is applied to the depth component, while the index is masked by the value given in Table 4.6 or converted to a GL float data type if the <type> is FLOAT."

Add a row to Table 4.7 (page 161):

type Parameter	GL Type	Component Conversion ...
...	...	...
UNSIGNED_INT_2_10_10_10_REV	uint	c = (2^N - 1)f
UNSIGNED_INT_24_8_NV	uint	c = (2^N - 1)f (depth only)

Update the second and third paragraphs of section 4.3.3 (page 162) to say:

"<type> is a symbolic constant that must be one of COLOR, STENCIL, DEPTH, or DEPTH_STENCIL_NV, indicating that the values to be transfered are colors, stencil values, or depth values, respectively. The first four arguments have the same interpretation as the corresponding arguments to ReadPixels.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.6.5, just as if ReadPixels were called with the corresponding arguments. If the <type> is STENCIL, DEPTH, or DEPTH_STENCIL_NV, then it is as if the <format> for ReadPixels were STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL_NV, respectively. If the <type> is COLOR, then if the GL is in RGBA mode, it is as if the <format> were RGBA, while if the GL is in color index mode, it is as if the <format> were COLOR_INDEX."

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

GLX Protocol

None.

Errors

The error INVALID_ENUM is generated if DrawPixels or ReadPixels is called where format is DEPTH_STENCIL_NV and type is not UNSIGNED_INT_24_8_NV.

The error INVALID_OPERATION is generated if DrawPixels or ReadPixels is called where type is UNSIGNED_INT_24_8_NV and format is not DEPTH_STENCIL_NV.

The error INVALID_OPERATION is generated if DrawPixels or ReadPixels is called where format is DEPTH_STENCIL_NV and there is not both a depth buffer and a stencil buffer.

The error INVALID_OPERATION is generated if CopyPixels is called where type is DEPTH_STENCIL_NV and there is not both a depth buffer and a stencil buffer.

New State

None.

Revision History

none yet

Name

NV_register_combiners

Name Strings

GL_NV_register_combiners

Notice

Copyright NVIDIA Corporation, 1999, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Shipping (version 1.4)

Version

NVIDIA Date: January 18, 2001 (version 1.4)
$Id: //sw/main/docs/OpenGL/specs/GL_NV_register_combiners.txt#44 $

Number

191

Dependencies

ARB_multitexture, assuming the value of MAX_ACTIVE_TEXTURES_ARB is at least 2.

Written based on the wording of the OpenGL 1.2 specification with the ARB_multitexture appendix E.

Overview

NVIDIA's next-generation graphics processor and its derivative designs support an extremely configurable mechanism know as "register combiners" for computing fragment colors.

The register combiner mechanism is a significant redesign of NVIDIA's original TNT combiner mechanism as introduced by NVIDIA's RIVA TNT graphics processor. Familiarity with the TNT combiners will help the reader appreciate the greatly enhanced register combiners functionality (see the NV_texture_env_combine4 OpenGL extension specification for this background). The register combiner mechanism has the following enhanced functionality:

The numeric range of combiner computations is from [-1,1] (instead of TNT's [0,1] numeric range),

The set of available combiner inputs is expanded to include the secondary color, fog color, fog factor, and a second combiner constant color (TNT's available combiner inputs consist of only zero, a single combiner constant color, the primary color,

texture 0, texture 1, and, in the case of combiner 1, the result of combiner 0).

Each combiner variable input can be independently scaled and biased into several possible numeric ranges (TNT can only complement combiner inputs).

Each combiner stage computes three distinct outputs (instead TNT's single combiner output).

The output operations include support for computing dot products (TNT has no support for computing dot products).

After each output operation, there is a configurable scale and bias applied (TNT's combiner operations builds in a scale and/or bias into some of its combiner operations).

Each input variable for each combiner stage is fetched from any entry in a combiner register set. Moreover, the outputs of each combiner stage are written into the register set of the subsequent combiner stage (TNT could only use the result from combiner 0 as a possible input to combiner 1; TNT lacks the notion of an input/output register set).

The register combiner mechanism supports at least two general combiner stages and then a special final combiner stage appropriate for applying a color sum and fog computation (TNT provides two simpler combiner stages, and TNT's color sum and fog stages are hard-wired and not subsumed by the combiner mechanism as in register combiners).

The register combiners fit into the OpenGL pipeline as a rasterization processing stage operating in parallel to the traditional OpenGL texture environment, color sum, AND fog application. Enabling this extension bypasses OpenGL's existing texture environment, color sum, and fog application processing and instead use the register combiners. The combiner and texture environment state is orthogonal so modifying combiner state does not change the traditional OpenGL texture environment state and the texture environment state is ignored when combiners are enabled.

OpenGL application developers can use the register combiner mechanism for very sophisticated shading techniques. For example, an approximation of Blinn's bump mapping technique can be achieved with the combiner mechanism. Additionally, multi-pass shading models that require several passes with unextended OpenGL 1.2 functionality can be implemented in several fewer passes with register combiners.

Issues

Should we expose the full register combiners mechanism?

RESOLUTION: NO. We ignore small bits of NV10 hardware functionality. The texture LOD input is ignored. We also ignore the inverts on input to the EF product.

Do we provide full gets for all the combiner state?

RESOLUTION: YES.

Do we parameterize combiner input and output updates to avoid enumerant explosions?

RESOLUTION: YES. To update a combiner stage input variable, you need to specify the <stage>, <portion>, and <variable>. To update a combiner stage output operation, you need to specify the <stage> and <portion>. This does mean that we need to add special Get routines that are likewise parameterized. Hence, GetCombinerInputParameter*, GetCombinerOutputParameter*, and GetFinalCombinerInputParameter*.

Is the register combiner functionality a super-set of the TNT combiner functionality?

Yes, but only in the sense of being a computational super-set. All computations performed with the TNT combiners can be performed with the register combiners, but the sequence of operations necessary to configure an identical computational result can be quite different.

For example, the TNT combiners have an operation that includes a final complement operation. The register combiners can perform range mappings only on inputs, but not on outputs. The register combiners can mimic the TNT operation with a post-operation complement only by taking pains to complement on input any uses of the output in later combiner stages.

What this does mean is that NV10's hardware functionality will permit support for both the NV_register_combiners AND NV_texture_env_combine4 extensions.

Note the existance of an "speclit" input complement bit supported by NV10 (but not accessible through the NV_register_combiners extensions).

Should we say anything about the precision of the combiner computations?

RESOLUTION: NO. The spec is written as if the computations are done on floating point values ranging from -1.0 to 1.0 (clamping is specified where this range is exceeded). The fact that NV10 does the computations as 9-bit signed fixed point is not mentioned in the spec. This permits a future design to support more precision or use a floating pointing representation.

What should the initial combiner state be?

RESOLUTION: See tables NV_register_combiners.4 and NV_register_combiners.5. The default state has one general combiner stage active that modulates the incoming color with texture 0. The final combiner is setup initially to implement OpenGL 1.2's standard color sum and fog stages.

What should happen to the TEXTURE0_ARB and TEXTUER1_ARB inputs if one or both textures are disabled?

RESOLUTION: The value of these inputs is undefined.

What do the TEXTURE0_ARB and TEXTURE1_ARB inputs correspond to? Does the number correspond to the absolute texture unit number or is the number based on how many textures are enabled (ie, TEXTURE_ARB0 would correspond to the 2nd texture unit if the 2nd unit is enabled, but the 1st is disabled).

RESOLUTION: The absolute texture unit.

This should be a lot less confusing to the programmer than having the texture inputs switch textures if texture 0 is disabled.

Note that the proposed hardware actually determines the TEXTURE0 and TEXTURE1 input based on which texture is enabled. This means it is up to the ICD to properly update the combiner state when just one texture is enabled. Since we will already have to do this to track the standard OpenGL texture environment for ARB_multitexture, we can do it for this extension too.

Should the combiners state be PushAttrib/PopAttrib'ed along with the texture state?

RESOLUTION: YES.

Should we advertise the LOD fractional input to the combiners?

RESOLUTION: NO.

There will be a performance impact when two combiner stages are enabled versus just one stage. Should we mention that somewhere?

RESOLUTION: NO. (But it is worth mentioning in this issues section.)

Should the scale and bias for the CombinerOutputNV be indicated by enumerants or specified outright as floats?

RESOLUTION: ENUMERANTS. While some future combiners might support an arbitrary scale & bias specified as floats, NV10 just does the enumerated options.

Should a dot product be computed in parralel with the sum of products?

RESOLUTION: NO. Language has been added ot the CombinerOutputNV discussion saying that if either <abDotProduct> or <cdDotProduct> is true, then <sumOutput> must be GL_DISCARD.

The rationale for this is that we want to minimize the number of adders that are required to ease a transition to floating point.

New Procedures and Functions

```
void CombinerParameterfvNV(GLenum pname,
                           const GLfloat *params);
```

```
void CombinerParameterivNV(GLenum pname,
                           const GLint *params);

void CombinerParameterfNV(GLenum pname,
                          GLfloat param);

void CombinerParameteriNV(GLenum pname,
                          GLint param);

void CombinerInputNV(GLenum stage,
                     GLenum portion,
                     GLenum variable,
                     GLenum input,
                     GLenum mapping,
                     GLenum componentUsage);

void CombinerOutputNV(GLenum stage,
                      GLenum portion,
                      GLenum abOutput,
                      GLenum cdOutput,
                      GLenum sumOutput,
                      GLenum scale,
                      GLenum bias,
                      GLboolean abDotProduct,
                      GLboolean cdDotProduct,
                      GLboolean muxSum);

void FinalCombinerInputNV(GLenum variable,
                          GLenum input,
                          GLenum mapping,
                          GLenum componentUsage);

void GetCombinerInputParameterfvNV(GLenum stage,
                                   GLenum portion,
                                   GLenum variable,
                                   GLenum pname,
                                   GLfloat *params);

void GetCombinerInputParameterivNV(GLenum stage,
                                   GLenum portion,
                                   GLenum variable,
                                   GLenum pname,
                                   GLint *params);

void GetCombinerOutputParameterfvNV(GLenum stage,
                                    GLenum portion,
                                    GLenum pname,
                                    GLfloat *params);

void GetCombinerOutputParameterivNV(GLenum stage,
                                    GLenum portion,
                                    GLenum pname,
                                    GLint *params);
```

```
    void GetFinalCombinerInputParameterfvNV(GLenum variable,
                                            GLenum pname,
                                            GLfloat *params);

    void GetFinalCombinerInputParameterivNV(GLenum variable,
                                            GLenum pname,
                                            GLfloat *params);
```

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

REGISTER_COMBINERS_NV	0x8522

Accepted by the <stage> parameter of CombinerInputNV, CombinerOutputNV, GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, and GetCombinerOutputParameterivNV:

COMBINER0_NV	0x8550
COMBINER1_NV	0x8551
COMBINER2_NV	0x8552
COMBINER3_NV	0x8553
COMBINER4_NV	0x8554
COMBINER5_NV	0x8555
COMBINER6_NV	0x8556
COMBINER7_NV	0x8557

Accepted by the <variable> parameter of CombinerInputNV, GetCombinerInputParameterfvNV, and GetCombinerInputParameterivNV:

VARIABLE_A_NV	0x8523
VARIABLE_B_NV	0x8524
VARIABLE_C_NV	0x8525
VARIABLE_D_NV	0x8526

Accepted by the <variable> parameter of FinalCombinerInputNV, GetFinalCombinerInputParameterfvNV, and GetFinalCombinerInputParameterivNV:

VARIABLE_A_NV	
VARIABLE_B_NV	
VARIABLE_C_NV	
VARIABLE_D_NV	
VARIABLE_E_NV	0x8527
VARIABLE_F_NV	0x8528
VARIABLE_G_NV	0x8529

Accepted by the <input> parameter of CombinerInputNV:

ZERO		(not new)
CONSTANT_COLOR0_NV	0x852A	
CONSTANT_COLOR1_NV	0x852B	
FOG		(not new)
PRIMARY_COLOR_NV	0x852C	
SECONDARY_COLOR_NV	0x852D	
SPARE0_NV	0x852E	
SPARE1_NV	0x852F	
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)

Accepted by the <mapping> parameter of CombinerInputNV:

UNSIGNED_IDENTITY_NV	0x8536
UNSIGNED_INVERT_NV	0x8537
EXPAND_NORMAL_NV	0x8538
EXPAND_NEGATE_NV	0x8539
HALF_BIAS_NORMAL_NV	0x853A
HALF_BIAS_NEGATE_NV	0x853B
SIGNED_IDENTITY_NV	0x853C
SIGNED_NEGATE_NV	0x853D

Accepted by the <input> parameter of FinalCombinerInputNV:

ZERO		(not new)
CONSTANT_COLOR0_NV		
CONSTANT_COLOR1_NV		
FOG		(not new)
PRIMARY_COLOR_NV		
SECONDARY_COLOR_NV		
SPARE0_NV		
SPARE1_NV		
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)
E_TIMES_F_NV	0x8531	
SPARE0_PLUS_SECONDARY_COLOR_NV	0x8532	

Accepted by the <mapping> parameter of FinalCombinerInputNV:

UNSIGNED_IDENTITY_NV
UNSIGNED_INVERT_NV

Accepted by the <scale> parameter of CombinerOutputNV:

NONE		(not new)
SCALE_BY_TWO_NV	0x853E	
SCALE_BY_FOUR_NV	0x853F	
SCALE_BY_ONE_HALF_NV	0x8540	

Accepted by the <bias> parameter of CombinerOutputNV:

NONE		(not new)
BIAS_BY_NEGATIVE_ONE_HALF_NV	0x8541	

Accepted by the <abOutput>, <cdOutput>, and <sumOutput> parameter of CombinerOutputNV:

DISCARD_NV	0x8530	
PRIMARY_COLOR_NV		
SECONDARY_COLOR_NV		
SPARE0_NV		
SPARE1_NV		
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)

Accepted by the <pname> parameter of GetCombinerInputParameterfvNV and GetCombinerInputParameterivNV:

COMBINER_INPUT_NV	0x8542
COMBINER_MAPPING_NV	0x8543
COMBINER_COMPONENT_USAGE_NV	0x8544

Accepted by the <pname> parameter of GetCombinerOutputParameterfvNV and GetCombinerOutputParameterivNV:

COMBINER_AB_DOT_PRODUCT_NV	0x8545
COMBINER_CD_DOT_PRODUCT_NV	0x8546
COMBINER_MUX_SUM_NV	0x8547
COMBINER_SCALE_NV	0x8548
COMBINER_BIAS_NV	0x8549
COMBINER_AB_OUTPUT_NV	0x854A
COMBINER_CD_OUTPUT_NV	0x854B
COMBINER_SUM_OUTPUT_NV	0x854C

Accepted by the <pname> parameter of CombinerParameterfvNV, CombinerParameterivNV, GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

CONSTANT_COLOR0_NV
CONSTANT_COLOR1_NV

Accepted by the <pname> parameter of CombinerParameterfvNV, CombinerParameterivNV, CombinerParameterfNV, CombinerParameteriNV, GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

NUM_GENERAL_COMBINERS_NV	0x854E
COLOR_SUM_CLAMP_NV	0x854F

Accepted by the <pname> parameter of GetFinalCombinerInputParameterfvNV and GetFinalCombinerInputParameterivNV:

COMBINER_INPUT_NV
COMBINER_MAPPING_NV
COMBINER_COMPONENT_USAGE_NV

Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

MAX_GENERAL_COMBINERS_NV	0x854D

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- **Figure 3.1 "Rasterization" (page 58)**

+ Change the "Texturing" block to say "Texture Fetching".

+ Insert a new block between "Texture Fetching" and "Color Sum". Name the new block "Texture Environment Application".

+ Insert a new block after "Texture Fetching". Name the new block "Register Combiners Application".

+ The output of the "Texture Fetching" stage feeds to both "Texture Environment Application" and "Register Combiners Application".

+ The input for "Color Sum" comes from "Texture Environment Application".

+ The output to "Fragments" is switched (controlled by Disable/Enable REGISTER_COMBINERS_NV) between the output of "Fog" and "Register Combiners Application".

Essentially, when register combiners are enabled, the entire standard texture environment application, color sum, and fog blocks are replaced with the single register combiners block. [Note that this is different from how the NV_texture_env_combine4 extension works; that extension controls the texture environment application block, but still uses the standard color sum and fog blocks.]

-- **NEW Section 3.8.12 "Register Combiners Application"**

"In parallel to the texture application, color sum, and fog processes described in sections 3.8.10, 3.9, and 3.10, register combiners provide a means of computing fcoc, the final combiner output color, for each fragment generated by rasterization.

The register combiners consist of two or more general combiner stages arranged in a fixed sequence ordered by each combiner stage's number. An implementation supports a maximum number of general combiners stages, which may be queried by calling GetIntegerv with the symbolic constant MAX_GENERAL_COMBINERS_NV. Implementations must support at least two general combiner stages. The general combiner stages are named COMBINER0_NV, COMBINER1_NV, and so on.

Each general combiner in the sequence receives its inputs and computes its outputs in an identical manner. At the end of the sequence of general combiner stages, there is a final combiner stage that operates in a different manner than the general combiner stages. The general combiner operation is described first, followed by a description of the final combiner operation.

Each combiner stage (the general combiner stages and the final combiner stage) has an associated combiner register set. Each

combiner register set contains <n> RGBA vectors with components ranging from -1.0 to 1.0 where <n> is 8 plus the maximum number of active textures supported (that is, the implementation's value for MAX_ACTIVE_TEXTURES_ARB). The combiner register set entries are listed in the table NV_register_combiners.1.

[Table NV_register_combiners.1]

Register Name	Initial Value	Reference	Output Status
ZERO	0	-	read only
CONSTANT_COLOR0_NV	ccc0	Section 3.8.12.1	read only
CONSTANT_COLOR1_NV	ccc1	Section 3.8.12.1	read only
FOG	Cf	Section 3.10	read only
PRIMARY_COLOR_NV	cpri	Section 2.13.1	read/write
SECONDARY_COLOR_NV	csec	Section 2.13.1	read/write
SPARE0_NV	see below	Section 3.8.12	read/write
SPARE1_NV	undefined	Section 3.8.12	read/write
TEXTURE0_ARB	CT0	Figure E.2	read/write
TEXTURE1_ARB	CT1	Figure E.2	read/write
TEXTURE<i>_ARB	CT<i>	Figure E.2	read/write

The register set of COMBINER0_NV, the first combiner stage, is initialized as described in table NV_register_combiners.1.

The initial value of the alpha portion of register SECONDARY_COLOR_NV is undefined. The initial value of the alpha portion of register SPARE0_NV is the alpha component of texture 0 if texturing is enabled for texture 0; however, the initial value of the RGB portion SPARE0_NV is undefined. The initial value of the SPARE1_NV register is undefined. The initial of registers TEXTURE0_ARB, TEXTURE1_ARB, and TEXTURE<i>_ARB are undefined if texturing is not enabled for textures 0, 1, and <i>, respectively.

The mapping of texture components to components of texture registers is summarized in Table NV_register_combiners.2. In the following table, At, Lt, It, Rt, Gt, Bt, and Dt, are the filtered texel values.

[Table NV_register_combiners.2]: Correspondence of texture components to register components for texture registers.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At
DEPTH_COMPONENT (when TEXTURE_COMPARE_SGIX is false)	0, 0, 0,	Lt
DEPTH_COMPONENT (when TEXTURE_COMPARE_SGIX is true)	Lt, Lt, Lt,	Lt
HILO_NV	0, 0, 0,	0
DSDT_NV	0, 0, 0,	0
DSDT_MAG_NV	0, 0, 0,	0
DSDT_MAG_INTENSITY_NV	0, 0, 0,	It

Note that the ALPHA, DEPTH_COMPONENT, and DSDT_MAG_INTENSITY_NV base internal formats are mapped to components differently than one could infer from the standard texture environment operations with this formats.

3.8.12.1 Combiner Parameters

Combiner parameters are specified by

```
CombinerParameterfvNV(GLenum pname, const GLfloat *params);
CombinerParameterivNV(GLenum pname, const GLint *params);
CombinerParameterfNV(GLenum pname, GLfloat param);
CombinerParameteriNV(GLenum pname, GLint param);
```

<pname> is a symbolic constant indicating which parameter is to be set as described in the table NV_register_combiners.3:

[Table NV_register_combiners.3]

Parameter	Name	Number of values	Type
ccc0	CONSTANT_COLOR0_NV	4	color
ccc1	CONSTANT_COLOR1_NV	4	color
ngc	NUM_GENERAL_COMBINERS_NV	1	positive integer
csc	COLOR_SUM_CLAMP_NV	1	boolean

<params> is a pointer to a group of values to which to set the indicated parameter. <param> is simply the indicated parameter. The number of values pointed to depends on the parameter being set as shown in the table above. Color parameters specified with CombinerParameter*NV are converted to floating-point values (if specified as integers) as indicated by Table 2.6 for signed integers. The floating-point color values are then clamped to the range [0,1].

The values ccc0 and ccc1 named by CONSTANT_COLOR0_NV and

CONSTANT_COLOR1_NV are constant colors available for inputs to the combiner stages. The value ngc named by NUM_GENERAL_COMBINERS_NV is a positive integer indicating how many general combiner stages are active, that is, how many general combiner stages a fragment should be processed by. Setting ngc to a value less than one or greater than the value of MAX_GENERAL_COMBINERS_NV generates an INVALID_VALUE error. The value csc named by COLOR_SUM_CLAMP_NV is a boolean described in section 3.8.12.3.

3.8.12.2 General Combiner Stage Operation

The command

```
CombinerInputNV(GLenum stage,
                GLenum portion,
                GLenum variable,
                GLenum input,
                GLenum mapping,
                GLenum componentUsage);
```

controls the assignment of all the general combiner input variables. For the RGB combiner portion, these are Argb, Brgb, Crgb, and Drgb; and for the combiner alpha portion, these are Aa, Ba, Ca, and Da. The <stage> parameter is a symbolic constant of the form COMBINER<i>_NV, indicating that general combiner stage <i> is to be updated. The constant COMBINER<i>_NV = COMBINER0_NV + <i> where <i> is in the range 0 to <k>-1 and <k> is the implementation dependent value of MAX_COMBINERS_NV. The <portion> parameter may be either RGB or ALPHA and determines whether the RGB color vector or alpha scalar portion of the specified combiner stage is updated. The <variable> parameter may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, or VARIABLE_D_NV and determines which respective variable of the specified combiner stage and combiner stage portion is updated.

The <input>, <mapping>, and <componentUsage> parameters specify the assignment of a value for the input variable indicated by <stage>, <portion>, and <variable>. The <input> parameter may be one of the register names from table NV_register_combiners.1.

The <componentUsage> parameter may be one of RGB, ALPHA, or BLUE.

When the <portion> parameter is RGB, a <componentUsage> parameter of RGB indicates that the RGB portion of the indicated register should be assigned to the RGB portion of the combiner input variable, while an ALPHA <componentUsage> parameter indicates that the alpha portion of the indicated register should be replicated across the RGB portion of the combiner input variable.

When the <portion> parameter is ALPHA, the <componentUsage> parameter of ALPHA indicates that the alpha portion of the indicated register should be assigned to the alpha portion of the combiner input variable, while a BLUE <componentUsage> parameter indicates that the blue component of the indicated register should be assigned to the alpha portion of the combiner input variable.

When the <portion> parameter is ALPHA, a <componentUsage> parameter

of RGB generates an INVALID_OPERATION error. When the <portion> parameter is RGB, a <componentUsage> parameter of BLUE generates an INVALID_OPERATION error.

When the <portion> parameter is ALPHA, an <input> parameter of FOG generates an INVALID_OPERATION error. The alpha component of the fog register is only available in the final combiner. The alpha component of the fog register is the fragment's fog factor when fog is enabled; otherwise, the alpha component of the fog register is one.

Before the value in the register named by <input> is assigned to the specified input variable, a range mapping is performed based on <mapping>. The mapping may be one of the tokens from the table NV_register_combiners.4.

[Table NV_register_combiners.4]

Mapping Name	Mapping Function
UNSIGNED_IDENTITY_NV	max(0.0, e)
UNSIGNED_INVERT_NV	1.0 - min(max(e, 0.0), 1.0)
EXPAND_NORMAL_NV	2.0 * max(0.0, e) - 1.0
EXPAND_NEGATE_NV	-2.0 * max(0.0, e) + 1.0
HALF_BIAS_NORMAL_NV	max(0.0, e) - 0.5
HALF_BIAS_NEGATE_NV	-max(0.0, e) + 0.5
SIGNED_IDENTITY_NV	e
SIGNED_NEGATE_NV	-e

Based on the <mapping> parameter, the mapping function in the table above is evaluated for each element <e> of the input vector before assigning the result to the specified input variable. Note that the mapping for the RGB and alpha portion of each input variable is distinct.

Each general combiner stage computes the following ten expressions based on the values assigned to the variables Argb, Brgb, Crgb, Drgb, Aa, Ba, Ca, and Da as determined by the combiner state set by CombinerInputNV.

["gcc" stands for general combiner computation.]

```
   gcc1rgb = [ Argb[r]*Brgb[r], Argb[g]*Brgb[g], Argb[b]*Brgb[b] ]

   gcc2rgb = [ Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b],
               Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b],
               Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b] ]

   gcc3rgb = [ Crgb[r]*Drgb[r], Crgb[g]*Drgb[g], Crgb[b]*Drgb[b] ]

   gcc4rgb = [ Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b],
               Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b],
               Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b] ]

   gcc5rgb = gcc1rgb + gcc3rgb

   gcc6rgb = gcc1rgb or gcc3rgb                [see below]
```

```
    gcc1a   = Aa * Ba

    gcc2a   = Ca * Da

    gcc3a   = gcc1a + gcc2a

    gcc4a   = gcc1a or gcc2a                    [see below]
```

The computation of gcc6rgb and gcc4a involves a special "or" operation. This operation evaluates to the left-hand operand if the alpha component of the combiner's SPARE0_NV register is less than 0.5; otherwise, the operation evaluates to the right-hand operand.

The command

```
    CombinerOutputNV(GLenum stage,
                     GLenum portion,
                     GLenum abOutput,
                     GLenum cdOutput,
                     GLenum sumOutput,
                     GLenum scale,
                     GLenum bias,
                     GLboolean abDotProduct,
                     GLboolean cdDotProduct,
                     GLboolean muxSum);
```

controls the general combiner output operation including designating the register set locations where results of the general combiner's three computations are written. The <stage> and <portion> parameters take the same values as the respective parameters for CombinerInputNV.

If the <portion> parameter is ALPHA, specifying a non-FALSE value for either of the parameters <abDotProduct> or <cdDotProduct>, generates an INVALID_VALUE error.

If the <abDotProduct> or <cdDotProduct> parameter is non-FALSE, the value of the <sumOutput> parameter must be GL_DISCARD_NV; otherwise, generate an INVALID_OPERATION error.

The <scale> parameter must be one of NONE, SCALE_BY_TWO_NV, SCALE_BY_FOUR_NV, or SCALE_BY_ONE_HALF_NV and specifies the value of the combiner stage's portion scale, either cscalergb or cscalea depending on the <portion> parameter, to 1.0, 2.0, 4.0, or 0.5, respectively.

The <bias> parameter must be either NONE or BIAS_BY_NEGATIVE_ONE_HALF_NV and specifies the value of the combiner stage's portion bias, either cbiasrgb or cbiasa depending on the <portion> parameter, to 0.0 or -0.5, respectively. If <scale> is either SCALE_BY_ONE_HALF_NV or SCALE_BY_FOUR_NV, a <bias> of BIAS_BY_NEGATIVE_ONE_HALF_NV generates an INVALID_OPERATION error.

If the <abDotProduct> parameter is FALSE, then

```
  if <portion> is RGB,     out1rgb = max(min(gcc1rgb + cbiasrgb) * cscalergb, 1), -1)
  if <portion> is ALPHA,   out1a   = max(min((gcc1a + cbiasa) * cscalea, 1), -1)
```

```
otherwise <portion> must be RGB and

   out1rgb = max(min((gcc2rgb + cbiasrgb) * cscalergb, 1), -1)

If the <cdDotProduct> parameter is FALSE, then

   if <portion> is RGB,      out2rgb = max(min((gcc3rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out2a   = max(min((gcc2a + cbiasa) * cscalea, 1), -1)

otherwise <portion> must be RGB so

   out2rgb = max(min((gcc4rgb + cbiasrgb) * cscalergb, 1), -1)

If the <muxSum> parameter is FALSE, then

   if <portion> is RGB,      out3rgb = max(min((gcc5rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out3a   = max(min((gcc3a + cbiasa) * cscalea, 1), -1)

otherwise

   if <portion> is RGB,      out3rgb = max(min((gcc6rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out3a   = max(min((gcc4a + cbiasa) * cscalea, 1), -1)
```

out1rgb, out2rgb, and out3rgb are written to the RGB portion of combiner stage's registers named by <abOutput>, <cdOutput>, and <sumOutput> respectively. out1a, out2a, and out3a are written to the alpha portion of combiner stage's registers named by <abOutput>, <cdOutput>, and <sumOutput> respectively. The parameters <abOutput>, <cdOutput>, and <sumOutput> must be either DISCARD_NV or one of the register names from table NV_register_combiners.1 that has an output status of read/write. If an output is set to DISCARD_NV, that output is not written to any register. The error INVALID_OPERATION is generated if <abOutput>, <cdOutput>, and <sumOutput> do not all name unique register names (though multiple outputs to DISCARD_NV are legal).

When the general combiner stage's register set is written based on the computed outputs, the updated register set is copied to the register set of the subsequent combiner stage in the combiner sequence. Copied undefined values are likewise undefined. The subsequent combiner stage following the last active general combiner stage, indicated by the general combiner stage's number being equal to ngc-1, in the sequence is the final combiner stage. In other words, the number of general combiner stages each fragment is transformed by is determined by the value of NUM_GENERAL_COMBINERS_NV.

3.8.12.3 Final Combiner Stage Operation

The final combiner stage operates differently from the general combiner stages. While a general combiner stage updates its register set and passes the register set to the next combiner stage, the final combiner outputs an RGBA color fcoc, the final combiner output color. The final combiner stage is capable of applying the standard OpenGL color sum and fog operations, but has the configurability to be used for other purposes.

The command

```
FinalCombinerInputNV(GLenum variable,
```

```
                        GLenum input,
                        GLenum mapping,
                        GLenum componentUsage);
```

controls the assignment of all the final combiner input variables. The variables A, B, C, D, E, and F are RGB vectors. The variable G is an alpha scalar. The <variable> parameter may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, VARIABLE_D_NV, VARIABLE_E_NV, VARIABLE_F_NV, and VARIABLE_G_NV, and determines which respective variable of the final combiner stage is updated.

The <input>, <mapping>, and <componentUsage> parameters specify the assignment of a value for the input variable indicated by <variable>.

The <input> parameter may be any one of the register names from table NV_register_combiners.1 or be one of two pseudo-register names, either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV. The value of E_TIMES_F_NV is the product of the value of variable E times the value of variable F. The value of SPARE0_PLUS_SECONDARY_COLOR_NV is the value the SPARE0_NV register mapped using the UNSIGNED_IDENITY_NV input mapping plus the value of the SECONDARY_COLOR_NV register mapped using the UNSIGNED_IDENTITY_NV input mapping. If csc, the color sum clamp, is non-FALSE, the value of SPARE0_PLUS_SECONDARY_COLOR_NV is first clamped to the range [0,1]. The alpha component of E_TIMES_F_NV and SPARE0_PLUS_SECONDARY_COLOR_NV is always zero.

When <variable> is one of VARIABLE_E_NV, VARIABLE_F_NV, or VARIABLE_G_NV and <input> is either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV, generate an INVALID_OPERATION error. When <variable> is VARIABLE_A_NV and <input> is SPARE0_PLUS_SECONDARY_COLOR_NV, generate an INVALID_OPERATION error.

The <componentUsage> parameter may be one of RGB, BLUE, ALPHA (with certain restrictions depending on the <variable> and <input> as described below).

When the <variable> parameter is not VARIABLE_G_NV, a <componentUsage> parameter of RGB indicates that the RGB portion of the indicated register should be assigned to the RGB portion of the combiner input variable, while an ALPHA <componentUsage> parameter indicates that the alpha portion of the indicated register should be replicated across the RGB portion of the combiner input variable.

When the <variable> parameter is VARIABLE_G_NV, a <componentUsage> parameter of ALPHA indicates that the alpha component of the indicated register should be assigned to the alpha portion of the G input variable, while a BLUE <componentUsage> parameter indicates that the blue component of the indicated register should be assigned to the alpha portion of the G input variable.

The INVALID_OPERATION error is generated when <componentUsage> is BLUE and <variable> is not VARIABLE_G_NV. The INVALID_OPERATION error is generated when <componentUsage> is RGB and <variable> is VARIABLE_G_NV.

The INVALID_OPERATION error is generated when both the <input> parameter is either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV and the <componentUsage> parameter is ALPHA or BLUE.

Before the value in the register named by <input> is assigned to the specified input variable, a range mapping is performed based on <mapping>. The mapping may be either UNSIGNED_IDENTITY_NV or UNSIGNED_INVERT_NV and operates as specified in table NV_register_combiners.4.

The final combiner stage computes the following expression based on the values assigned to the variables A, B, C, D, E, F, and G as determined by the combiner state set by FinalCombinerInputNV

```
  fcoc = [ min(ab[r] + iac[r] + D[r], 1.0),
           min(ab[g] + iac[g] + D[g], 1.0),
           min(ab[b] + iac[b] + D[b], 1.0),
           G ]
```

where

```
  ab   = [ A[r]*B[r], A[g]*B[g], A[b]*B[b] ]
  iac  = [ (1.0 -A [r])*C[r], (1.0 - A[g])*C[g], (1.0 - A[b])*C[b] ]
```

3.8.12.4 Required State

The state required for the register combiners is a bit indicating whether register combiners are enabled or disabled, an integer indicating how many general combiners are active, a bit indicating whether or not the color sum clamp to 1 should be performed, two RGBA constant colors, <n> sets of general combiner stage state where <n> is the value of MAX_GENERAL_COMBINERS_NV, and the final combiner stage state. The per-stage general combiner state consists of the RGB input portion state and the alpha input portion state. Each portion (RGB and alpha) of the per-stage general combiner state consists of: four integers indicating the input register for the four variables A, B, C, and D; four integers to indicate each variable's range mapping; four bits to indicate whether to use the alpha component of the input for each variable; a bit indicating whether the AB dot product should be output; a bit indicating whether the CD dot product should be output; a bit indicating whether the sum or mux output should be output; two integers to maintain the output scale and bias enumerants; three integers to maintain the output register set names. The final combiner stage state consists of seven integers to indicate the input register for the seven variables A, B, C, D, E, F, and G; seven integers to indicate each variable's range mapping; and seven bits to indicate whether to use the alpha component of the input for each variable.

The general combiner per-stage state is initialized as described in table NV_register_combiners.5.

[Table NV_register_combiners.5]

Portion	Variable	Input	Component Usage	Mapping
RGB	A	PRIMARY_COLOR_NV	RGB	UNSIGNED_IDENTITY_NV
RGB	B	ZERO	RGB	UNSIGNED_INVERT_NV
RGB	C	ZERO	RGB	UNSIGNED_IDENTITY_NV
RGB	D	ZERO	RGB	UNSIGNED_IDENTITY_NV
alpha	A	PRIMARY_COLOR_NV	ALPHA	UNSIGNED_IDENTITY_NV
alpha	B	ZERO	ALPHA	UNSIGNED_INVERT_NV
alpha	C	ZERO	ALPHA	UNSIGNED_IDENTITY_NV
alpha	D	ZERO	ALPHA	UNSIGNED_IDENTITY_NV

The final combiner stage state is initialized as described in table NV_register_combiners.6.

[Table NV_register_combiners.6]

Variable	Input	Component Usage	Mapping
A	FOG	ALPHA	UNSIGNED_IDENTITY_NV
B	SPARE0_PLUS_SECONDARY_COLOR_NV	RGB	UNSIGNED_IDENTITY_NV
C	FOG	RGB	UNSIGNED_IDENTITY_NV
D	ZERO	RGB	UNSIGNED_IDENTITY_NV
E	ZERO	RGB	UNSIGNED_IDENTITY_NV
F	ZERO	RGB	UNSIGNED_IDENTITY_NV
G	SPARE0_NV	ALPHA	UNSIGNED_IDENTITY_NV"

-- NEW Section 3.8.11 "Antialiasing Application"

Insert the following paragraph BEFORE the section's first paragraph:

"Register combiners are enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant REGISTER_COMBINERS_NV. If the register combiners are enabled (and not in color index mode), the fragment's color value is replaced with fcoc, the final combiner output color, computed in section 3.8.12; otherwise, the fragment's color value is the result of the fog application in section 3.10."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the first two sentences (page 182) to say:

"Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, combiners, etc.) as well as

a symbolic constant. These are"

Add to the bottom of the list of function prototypes (page 183):

```
  void GetCombinerInputParameterfvNV(GLenum stage, GLenum portion,
                                     GLenum variable,
                                     GLenum pname, const GLfloat *params);
  void GetCombinerInputParameterivNV(GLenum stage, GLenum portion,
                                     GLenum variable,
                                     GLenum pname, const GLint *params);
  void GetCombinerOutputParameterfvNV(GLenum stage, GLenum portion,
                                      GLenum pname, const GLfloat *params);
  void GetCombinerOutputParameterivNV(GLenum stage, GLenum portion,
                                      GLenum pname, GLint *params);
  void GetFinalCombinerInputParameterfvNV(GLenum variable, GLenum pname,
                                          const GLfloat *params);
  void GetFinalCombinerInputParameterivNV(GLenum variable, GLenum pname,
                                          const GLfloat *params);
```

Add the following paragraph to the end of the section (page 184):

"The GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, and GetCombinerOutputParameterivNV parameter <stage> may be one of COMBINER0_NV, COMBINER1_NV, and so on, indicating which general combiner stage to query. The GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, and GetCombinerOutputParameterivNV parameter <portion> may be either RGB or ALPHA, indicating which portion of the general combiner stage to query. The GetCombinerInputParameterfvNV and GetCombinerInputParameterivNV parameter <variable> may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, or VARIABLE_D_NV, indicating which variable of the general combiner stage to query. The GetFinalCombinerInputParameterfvNV and GetFinalCombinerInputParameterivNV parameter <variable> may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, VARIABLE_D_NV, VARIABLE_E_NV, VARIABLE_F_NV, or VARIABLE_G_NV."

Additions to the GLX Specification

None.

GLX Protocol

Thirteen new GL commands are added.

The following seven rendering commands are sent to the sever as part of a glXRender request:

CombinerParameterfNV

2	12	rendering command length
2	4136	rendering command opcode
4	ENUM	pname
4	FLOAT32	param

CombinerParameterfvNV

2	8+4*n	rendering command length

```
    2           4137            rendering command opcode
    4           ENUM            pname
                0x852A   n=4    GL_CONSANT_COLOR0_NV
                0x852B   n=4    GL_CONSANT_COLOR1_NV
                0x854E   n=1    GL_NUM_GENERAL_COMBINERS_NV
                0x854F   n=1    GL_COLOR_SUM_CLAMP_NV
                else     n=0
    4*n         LISTofFLOAT32   params

CombinerParameteriNV
    2           12              rendering command length
    2           4138            rendering command opcode
    4           ENUM            pname
    4           INT32           param

CombinerParameterivNV
    2           8+4*n           rendering command length
    2           4139            rendering command opcode
    4           ENUM            pname
                0x852A   n=4    GL_CONSANT_COLOR0_NV
                0x852B   n=4    GL_CONSANT_COLOR1_NV
                0x854E   n=1    GL_NUM_GENERAL_COMBINERS_NV
                0x854F   n=1    GL_COLOR_SUM_CLAMP_NV
                else     n=0
    4*n         LISTofINT32     params

CombinerInputNV
    2           28              rendering command length
    2           4140            rendering command opcode
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            variable
    4           ENUM            input
    4           ENUM            mapping
    4           ENUM            componentUsage

CombinerOutputNV
    2           36              rendering command length
    2           4141            rendering command opcode
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            abOutput
    4           ENUM            cdOutput
    4           ENUM            sumOutput
    4           ENUM            scale
    4           ENUM            bias
    1           BOOL            abDotProduct
    1           BOOL            cdDotProduct
    1           BOOL            muxSum
    1           BOOL            unused
```

```
    FinalCombinerOutputNV
        2           20              rendering command length
        2           4142            rendering command opcode
        4           ENUM            variable
        4           ENUM            input
        4           ENUM            mapping
        4           ENUM            componentUsage
```

The remaining six commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
    GetCombinerInputParameterfvNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           7               request length
        4           1270            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            stage
        4           ENUM            portion
        4           ENUM            variable
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m = (n==1 ? 0 : n)
        4                           unused
        4           CARD32          unused

        if (n=1) this follows:

        4           FLOAT32         params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofFLOAT32   params
```

```
GetCombinerInputParameterivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           7               request length
    4           1271            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            variable
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetCombinerOutputParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1272            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

```
GetCombinerOutputParameterivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1273            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetFinalCombinerInputParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1274            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            variable
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

```
    GetFinalCombinerInputParameterivNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           5               request length
        4           1275            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            variable
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m = (n==1 ? 0 : n)
        4                           unused
        4           CARD32          unused

        if (n=1) this follows:

        4           INT32           params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofINT32     params
```

Errors

INVALID_VALUE is generated when CombinerParameterfvNV or CombinerParameterivNV is called with <pname> set to NUM_GENERAL_COMBINERS and the value pointed to by <params> is less than one or greater or equal to the value of MAX_GENERAL_COMBINERS_NV.

INVALID_OPERATION is generated when CombinerInputNV is called with a <componentUsage> parameter of RGB and a <portion> parameter of ALPHA.

INVALID_OPERATION is generated when CombinerInputNV is called with a <componentUsage> parameter of BLUE and a <portion> parameter of RGB.

INVALID_OPERATION is generated When CombinerInputNV is called with a <componentUsage> parameter of ALPHA and an <input> parameter of FOG.

INVALID_VALUE is generated when CombinerOutputNV is called with a <portion> parameter of ALPHA, but a non-FALSE value for either of the parameters <abDotProduct> or <cdDotProduct>.

INVALID_OPERATION is generated when CombinerOutputNV is called with a <scale> of either SCALE_BY_TWO_NV or SCALE_BY_FOUR_NV and a <bias> of BIAS_BY_NEGATIVE_ONE_HALF_NV.

INVALID_OPERATION is generated when CombinerOutputNV is called such that <abOutput>, <cdOutput>, and <sumOutput> do not all name unique register names (though multiple outputs to DISCARD_NV are legal).

INVALID_OPERATION is generated when FinalCombinerOutputNV
is called where <variable> is one of VARIABLE_E_NV,
VARIABLE_F_NV, or VARIABLE_G_NV and <input> is E_TIMES_F_NV
or SPARE0_PLUS_SECONDARY_COLOR_NV.

INVALID_OPERATION is generated when FinalCombinerOutputNV
is called where <variable> is VARIABLE_A_NV and <input> is
SPARE0_PLUS_SECONDARY_COLOR_NV.

INVALID_OPERATION is generated when FinalCombinerInputNV is called
with VARIABLE_G_NV for <variable> and RGB for <componentUsage>.

INVALID_OPERATION is generated when FinalCombinerInputNV is called
with a value other than VARIABLE_G_NV for <variable> and BLUE for
<componentUsage>.

INVALID_OPERATION is generated when FinalCombinerInputNV is
called where the <input> parameter is either E_TIMES_F_NV or
SPARE0_PLUS_SECONDARY_COLOR_NV and the <componentUsage> parameter
is ALPHA.

INVALID_OPERATION is generated when CombinerOutputNV is called with
either <abDotProduct> or <cdDotProduct> assigned non-FALSE and
<sumOutput> is not GL_DISCARD_NV.

New State

-- (NEW table 6.29, after p217)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
REGISTER_COMBINERS_NV texture/enable	B	IsEnabled	False	register combiners enable	3.8.11	
NUM_GENERAL_COMBINERS_NV	Z+	GetIntegerv	1	number of active combiner stages	3.8.12.1	texture
COLOR_SUM_CLAMP_NV	B	GetBooleanv	True	whether or not SPARE0_PLUS_SECONDARY_COLOR_NV clamps combiner stages	3.8.12.1	texture
CONSTANT_COLOR0_NV	C	GetFloatv	0,0,0,0	combiner constant color zero	3.8.12.1	texture
CONSTANT_COLOR1_NV	C	GetFloatv	0,0,0,0	combiner constant color one	3.8.12.1	texture
COMBINER_INPUT_NV	Z8x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	combiner input variables	3.8.12.2	texture
COMBINER_COMPONENT_USAGE_NV	Z3x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	use alpha for combiner input	3.8.12.2	texture
COMBINER_MAPPING_NV	Z8x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	complement combiner input	3.8.12.2	texture
COMBINER_AB_DOT_PRODUCT_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output AB dot product	3.8.12.3	texture
COMBINER_CD_DOT_PRODUCT_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output CD dot product	3.8.12.3	texture
COMBINER_MUX_SUM_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output mux sum	3.8.12.3	texture
COMBINER_SCALE_NV	Z2x#x2	GetCombinerOutputParameter*NV	NONE	output scale	3.8.12.3	texture
COMBINER_BIAS_NV	Z2x#x2	GetCombinerOutputParameter*NV	NONE	output bias	3.8.12.3	texture
COMBINER_AB_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	DISCARD_NV	AB output register	3.8.12.3	texture
COMBINER_CD_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	DISCARD_NV	CD output register	3.8.12.3	texture
COMBINER_SUM_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	SPARE0_NV	sum output register	3.8.12.3	texture
COMBINER_INPUT_NV	Z10x7	GetFinalCombinerInputParameter*NV	see 3.8.12.4	final combiner input	3.8.12.4	texture
COMBINER_MAPPING_NV	Z2x7	GetFinalCombinerInputParameter*NV	UNSIGNED_IDENTITY_NV	final combiner input mapping	3.8.12.4	texture
COMBINER_COMPONENT_USAGE_NV	Z2x7	GetFinalCombinerInputParameter*NV	see 3.8.12.4	use alpha for final combiner input mapping	3.8.12.4	texture

[where # is the value of MAX_GENERAL_COMBINERS_NV]

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_GENERAL_COMBINERS_NV	Z+	GetIntegerv	2	Maximum num of general combiner stages	3.8.12	-

NVIDIA Implementation Details

The effective range of the RGB portion of the final combiner should be [0,4] if the color sum clamp is false. Exercising this range requires assigning SPARE0_PLUS_SECONDARY_COLOR_NV to the D variable and either B or C or both B and C. In practice this is a very unlikely configuration.

However due to a bug in the GeForce 256 and Quadro hardware, values generated above 2 in the RGB portion of the final combiner will be computed incorrectly. GeForce2 GTS and subsequent NVIDIA GPUs have

fixed this bug.

Revision History

April 4, 2000 - Document that alpha component of the FOG register should be zero when fog is disabled. The Release 4 NVIDIA drivers have a bug where this is not always true (though it often still is). The bug is fixed in the Release 5 NVIDIA drivers.

June 8, 2000 - The alpha component of the FOG register is not available for use until the final combiner. The specification previously incorrectly stated:

> "INVALID_OPERATION is generated When CombinerInputNV is called with a <portion> parameter of ALPHA and an <input> parameter of FOG."

It is actually the <componentUsage> (not the <portion>) that should not be allowed to be ALPHA. The Release 4 NVIDIA drivers implemented the above incorrect error check. The Release 5 (and later) NVIDIA drivers (after June 8, 2000) have fixed this bug and correctly implement the error based on <componentUsage>.

The specification previously did not allow BLUE for the <componentUsage> of the G variable in the final combiner. This is now allowed in the Release 5 (and later) NVIDIA drivers (after June 8, 2000). The Release 4 NVIDIA drivers do not permit BLUE for the <componentUsage> of the G variable and generate an INVALID_OPERATION error if this is attempted. The Release 5 NVIDIA drivers (after June 8, 2000) have fixed this bug and permit BLUE for the <componentUsage> of the G variable.

August 11, 2000 - The "mux" operation was incorrectly documented in previous versions of this specification. The correct mux behave is as follows:

```
spare0_alpha >= 0.5 ? C*D : A*B
```

or

```
spare0_alpha <  0.5 ? A*B : C*D
```

Previous versions of this specification had the mux sense reversed.

October 31, 2000 - The initial general combiner state was misdocumented for the B variable. Previously, Table NV_register_combiners.5 said that the RGB and alpha inputs for B were GL_TEXTURE#_ARB and the RGB and alpha input mappings for B were GL_UNSIGNED_IDENTITY_NV. The table is now updated so that the RGB and alpha inputs for B are GL_ZERO and the RGB and alpha input mappings for B are GL_UNSIGNED_INVERT_NV. The implementation has always behaved in the manner described by the updated specification.

December 13, 2000 - Added a new table NV_register_combiners.2 describing the correspondence of texture components to register components for texture registers. This table is based on the table in the EXT_texture_env_combine extension. The table includes correspondences for HILO, DSDT, DSDT_MAG, DSDT_MAG_INTENSITY, and DEPTH_COMPONENT formatted textures when supported in conjunction with the NV_texture_shader, SGIX_depth_texture, and SGIX_shadow extensions.

Because a new table 2 was inserted, all the tables beyond it are renumbered.

```
Document the behavior of SIGNED_NEGATE_NV in conjunction with shadow
mapping in the "NVIDIA Implementation Details" section.
```

Name

NV_register_combiners2

Name Strings

GL_NV_register_combiners2

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Implemented.

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_register_combiners2.txt#1 $

Number

227

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

Assumes support for the NV_register_combiners extension (version 1.4).

Overview

The NV_register_combiners extension provides a powerful fragment coloring mechanism. This specification extends the register combiners functionality to support more color constant values that are unique for each general combiner stage.

The base register combiners functionality supports only two color constants. These two constants are available in every general combiner stage and in the final combiner.

When many general combiner stages are supported, more than two unique color constants is often required. The obvious way to extend the register combiners is to add several more color constant registers. But adding new unique color constant registers is expensive for hardware implementation because every color constant register must be available as an input to any stage.

In practice however, it is the total set of general combiner stages that requires more color constants, not each and every individual general combiner stage. Each individual general combiner stage typically requires only one or two color constants.

By keeping two color constant registers but making these two registers contain two unique color constant values for each general combiner stage, the hardware expense of supporting multiple color constants is minimized. Additionally, this scheme scales appropriately as more general combiner stages are added.

Issues

How do is compatibility maintained with the original register combiners?

RESOLUTION: Initially, per general combiner stage constants are disabled and the register combiners operate as described in the original NV_register_combiners specification. A distinct "per stage constants" enable exposes this extension's new functionality.

Where do the final combiner color constant values come from?

RESOLUTION: When "per stage constants" is enabled, the final combiner color constants continue to use the constant colors set with glCombinerParaterfvNV.

New Procedures and Functions

```
void CombinerStageParameterfvNV(GLenum stage,
                                GLenum pname,
                                const GLfloat *params);

void GetCombinerStageParameterfvNV(GLenum stage,
                                   GLenum pname,
                                   GLfloat *params);
```

New Tokens

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

PER_STAGE_CONSTANTS_NV	0x8535

Accepted by the <pname> parameter of CombinerStageParameterfvNV and GetCombinerStageParameterfvNV:

CONSTANT_COLOR0_NV	(see NV_register_combiners)
CONSTANT_COLOR1_NV	(see NV_register_combiners)

Accepted by the <stage> parameter of CombinerStageParameterfvNV and GetCombinerStageParameterfvNV:

COMBINER0_NV	(see NV_register_combiners)
COMBINER1_NV	(see NV_register_combiners)
COMBINER2_NV	(see NV_register_combiners)
COMBINER3_NV	(see NV_register_combiners)
COMBINER4_NV	(see NV_register_combiners)
COMBINER5_NV	(see NV_register_combiners)
COMBINER6_NV	(see NV_register_combiners)
COMBINER7_NV	(see NV_register_combiners)

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8.12 "Register Combiners Application"

Add a paragraph immediately before section 3.8.12.1:

"The ccc0 and ccc1 values map to particular constant color values. The ccc0 and ccc1 mappings depend on whether per-stage constants are enabled or not. Per-stage constants are enabled and disabled with the Enable and Disable commands using the symbolic constant PER_STAGE_CONSTANTS_NV.

When per-stage constants are disabled, ccc0 and ccc1 are mapped to the register combiners' global color constant values, gccc0 and gccc1.

When per-stage constants are enabled, ccc0 and ccc1 depend on the combiner stage that inputs the COLOR_CONSTANT0_NV and COLOR_CONSTANT1_NV registers. Each general combiner stage # maps ccc0 and ccc1 to the per-stage values s#ccc0 and s#ccc1 respectively. The final combiner maps ccc0 and ccc1 to the values gccc0 and gccc1 (the same as if per-stage constants are disabled).

gccc0, gccc1, s#ccc0, and s#ccc1 are further described in the following section."

-- Section 3.8.12.1 "Combiner Parameters"

Change Table NV_register_combiners.3 to read "gccc0" instead of "ccc0" and "gccc1" instead of "ccc1".

Change the first sentence of the last paragraph to read:

"The values gccc0 and gccc1 named by CONSTANT_COLOR0_NV and CONSTANT_COLOR1_NV are global constant colors available for inputs to the final combiner stage and, when per-stage constants is disabled, to the general combiner stages."

Add the following after the last paragraph in the section:

"Per-stage combiner parameters are specified by

```
void CombinerStageParameterfvNV(GLenum stage,
                                GLenum pname,
                                const GLfloat *params);
```

The <stage> parameter is a symbolic constant of the form COMBINER<#>_NV, indicating the general combiner stage <#> whose parameter named by <pname> is to be updated. <pname> must be either CONSTANT_COLOR0_NV or CONSTANT_COLOR1_NV. <params> is a pointer to a group of values to which to set the indicated parameter. The parameter names CONSTANT_COLOR0_NV and CONSTANT_COLOR1_NV update the per-stage color constants s#ccc0 and s#ccc1 respectively where # is the number of the specified general combiner stage."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Add to the bottom of the list of function prototypes (page 183):

```
void GetCombinerStageParameterfvNV(GLenum stage,
                                   GLenum pname,
                                   GLfloat *params);
```

Change the first sentence describing the register combiner queries to mention GetCombinerStageParameterfvNV so the sentence reads:

"The GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, GetCombinerOutputParameterivNV, and GetCombinerStageParameterfvNV parameter <stage> may be one of COMBINER0_NV, COMBINER1_NV, and so on, indicating which general combiner stage to query."

Additions to the GLX Specification

None

GLX Protocol

Two new GL commands are added.

The following rendering command is sent to the sever as part of a glXRender request:

```
CombinerParameterfvNV
    2           8+4*n           rendering command length
    2           ????            rendering command opcode
    4           ENUM            pname
                0x852A   n=4    GL_CONSANT_COLOR0_NV
                0x852B   n=4    GL_CONSANT_COLOR1_NV
                else     n=0
    4*n         LISTofFLOAT32   params
```

The remaining command is a non-rendering command. This commands is sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
GetCombinerStageParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            stage
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

Errors

None

New State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PER_STAGE_CONSTANTS_NV	B	IsEnabled	False	enable for	3.8.12	texture/enable
CONSTANT_COLOR0_NV	Cx#	GetCombinerStageParameterfvNV	0,0,0,0	per-stage constant color zero	3.8.12.1	texture
CONSTANT_COLOR1_NV	Cx#	GetCombinerStageParameterfvNV	0,0,0,0	per-stage constant color one	3.8.12.1	texture

[where # is the value of MAX_GENERAL_COMBINERS_NV]

New Implementation State

None

Name

NV_texgen_emboss

Name Strings

GL_NV_texgen_emboss

Notice

Copyright NVIDIA Corporation, 1999, 2001.

IP Status

NVIDIA Proprietary.

Status

Deprecated. Future NVIDIA drivers will NOT support this extension. Developers are strongly encouraged to use NV_vertex_program instead of this extension.

Version

NVIDIA Date: February 20, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texgen_emboss.txt#22 $

Number

193

Dependencies

ARB_multitexture.

Written based on the wording of the OpenGL 1.2 specification and the ARB_multitexture extension.

Overview

This extension provides a new texture coordinate generation mode suitable for multitexture-based embossing (or bump mapping) effects.

Given two texture units, this extension generates the texture coordinates of a second texture unit (an odd-numbered texture unit) as a perturbation of a first texture unit (an even-numbered texture unit one less than the second texture unit). The perturbation is based on the normal, tangent, and light vectors. The normal vector is supplied by glNormal; the light vector is supplied as a direction vector to a specified OpenGL light's position; and the tanget vector is supplied by the second texture unit's current texture coordinate. The perturbation is also scaled by program-supplied scaling constants.

If both texture units are bound to the same texture representing a height field, by subtracting the difference between the resulting two filtered texels, programs can achieve a per-pixel embossing effect.

Issues

Can you do embossing on any texture unit?

NO. Just odd numbered units. This meets a constraint of the proposed hardware implementation, and because embossing takes two texture units anyway, it shouldn't be a real limitation.

Can you just enable one coordinate of a texture unit for embossing?

Yes but NOT REALLY. The texture coordinate generation formula is specified such that only when ALL the coordinates are enabled and are using embossing, do you get the embossing computation. Otherwise, you get undefined values for texture coordinates enabled for texture coordinate generation and setup for embossing.

Does the light specified have to be enabled for embossing to work?

Yes, currently. But perhaps we could require implementations to enable a phantom light (the light colors would be black).

Could the emboss constant just be the reciprocal of the width and height of the texture units texture if that's what the programmer will have it be most of the time?

NO. Too much work and there may be reasons for the programmer to control this.

OpenGL's base texture environment functionality isn't powerful enough to do the subtraction needed for embossing. Where would you get powerful enough texture environment functionality.

Another extension. Try NV_register_combiners.

What is the interpretation of CT?

For the purposes of embossing, CT should be thought of as the vertex's tangent vector. This tangent vector indicates the direction on the "surface" where PCTs is not changing and PCTt is increasing.

Are the CT and PCT variables the user-supplied current texture coordinates?

YES. Except when the texture unit's texture coordinate evaluator is enabled, then CT and PCT use the respective evaluated texture coordinates.

This extension specification's language "Denote as CT the texture unit's current texture coordinates" and "Denote as PCT the previous texture unit's current texture coordinates" refers to the "current texture coordinates" OpenGL state which is the state specified via glTexCoord. Plus the exception for evaluators.

To be explicit, PCT is NOT the result of texgen or the texture matrix. Likewise, CT is NOT the result of texgen or the texture matrix. PCT and CT are the respective texture unit's

evaluated texture coordinate if the vertex is evaluated with texture coordinate evaluation enabled, otherwise if the vertex is generated via vertex arrays with the respective texture coordinate array enabled, the texture coordinate from the texture coordinate array, otherwise the respective current texture coordinate is used.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

EMBOSS_MAP_NV	0x855F

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain EMBOSS_MAP_NV.

Accepted by the <pname> parameters of GetTexGendv, GetTexGenfv, GetTexGeniv, TexGend, TexGendv, TexGenf, TexGenfv, TexGeni, and TexGeniv:

EMBOSS_LIGHT_NV	0x855D
EMBOSS_CONSTANT_NV	0x855E

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, or EMBOSS_MAP_NV."

Add these paragraphs after the 4th paragraph:

"When used with a suitable texture, suitable explicit texture coordinates, a suitable (extended) texture environment, suitable lighting parameters, and suitable embossing parameters, calling TexGen with TEXTURE_GEN_MODE indicating EMBOSS_MAP_NV can simulate the lighting effect of embossing on a polygon. The error INVALID_ENUM occurs when the active texture unit has an even number.

The emboss constant and emboss light parameters for controlling the EMBOSS_MAP_NV mode are specified by calling TexGen with pname set to EMBOSS_CONSTANT_NV and EMBOSS_LIGHT_NV respectively.

When pname is EMBOSS_CONSTANT_NV, param or what params points to is a scalar value. An error INVALID_ENUM occurs if pname is EMBOSS_CONSTANT_NV and coord is R or Q. An error INVALID_ENUM also occurs if pname is EMBOSS_CONSTANT_NV and the active texture unit number is even.

When pname is EMBOSS_LIGHT_NV, param or what params points to is a symbolic constant of the form LIGHTi, indicating that light i is to have the specified parameter set. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and coord is R or Q. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and the active texture unit number is even. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and the value i for LIGHTi is negative or is greater than or equal to the value of MAX_LIGHTS.

If TEXTURE_GEN_MODE indicates EMBOSS_MAP_NV, the generation function for the coordinates S, T, R, and Q is computed as follows.

Denote as L the light direction vector from the vertex's eye position to the position of the light specified by the coordinate's EMBOSS_LIGHT_NV state (the direction vector is computed as described in Section 3.13.1).

Denote as N the current normal after transformation to eye coordinates.

Denote as CT the texture unit's current texture coordinates transformed to eye coordinates by normal transformation (as described in Section 3.10.3) and normalized.

However, if the vertex is evaluated (as described in Section 5.1) and the texture unit's texture coordinate map is enabled, use the texture unit's evaluated texture coordinate to compute CT.

Denote as B the cross product of N and the <s,t,r> vector of CT.

```
Bx = Ny*CTr - CTt*Nz
By = Nz*CTs - CTr*Nx
Bz = Nx*CTt - CTs*Ny
```

Denote as BN the normalized version of the vector B.

```
BNx = Bx / sqrt(Bx*Bx + By*By + Bz*Bz);
BNy = By / sqrt(Bx*Bx + By*By + Bz*Bz);
BNz = Bz / sqrt(Bx*Bx + By*By + Bz*Bz);
```

Denote as T the cross product of B and N.

```
Tx = BNy*Nz - Ny*BNz
Ty = BNz*Nx - Nz*BNx
Tz = BNx*Ny - Nx*BNy
```

Observe that BN and T are orthonormal.

Denote as PCT the previous texture unit's current texture coordinates. If the number of the texture unit for the texture coordinates being generated is n, then the previous texture unit is texture unit number n-1. Note that n is restricted to be odd.

However, if the vertex is evaluated (as described in Section 5.1) and the previous texture unit's texture coordinate map is enabled,

use the previous texture unit's evaluated texture coordinate to compute PCT.

Denote Ks as the S coordinate's EMBOSS_CONSTANT_NV state. Denote Kt as the T coordinate's EMBOSS_CONSTANT_NV state. These constants should typically be set to the reciprocal of the width and height respectively of the texture map used for embossing.

Denote E as follows:

```
Es = PCTs + Ks * (Lx*BNx + Ly*BNy + Lz*BNz) * PCTq
Et = PCTt - Kt * (Lx*Tx + Ly*Ty + Lz*Tz) * PCTq
Er = PCTr
Eq = PCTq
```

Then the value assigned to an s, t, r, and q coordinates are Es, Et, Er, and Eq respectively. However, for this assignment to occur, the following three conditions must be met. First, all the texture coordinate generation modes of all the texture coordinates (S, T, R, and Q) of the texture unit must be set to EMBOSS_MAP_NV. Second, all the texture coordinate generation modes of the texture unit must be enabled. Third, the EMBOSS_LIGHT_NV parameters of coordinates S and T must be identical and the light and lighting must be enabled. If these conditions are not met, the values of all coordinates in the texture unit with the EMBOSS_MAP_NV mode are undefined."

The last paragraph's first sentence should be changed to:

"The state required for texture coordinate generation comprises a five-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR; also, an emboss constant and emboss light are required for each of the four coordinates.... The initial values for emboss constants and emboss lights are 1.0 and LIGHT0 respectively."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <pname> of TEXTURE_GEN_MODE, a <param> value or value of what <params> points to of EMBOSS_MAP_NV, and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_CONSTANT_NV and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_LIGHT_NV and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <coord> of R or Q when <pname> indicates EMBOSS_CONSTANT_NV.

INVALID_ENUM is generated when TexGen is called with a <coord> of R or Q when <pname> indicates EMBOSS_LIGHT_NV.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_LIGHT_NV and the value of i for the parameter LIGHTi is negative or is greater than or equal to the value of MAX_LIGHTS.

New State

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture
EMBOSS_CONSTANT_NV	4xR	GetTexGenfv	1.0	Scaling constant for emboss texgen	2.10.4	texture
EMBOSS_LIGHT_NV	4xZ8*	GetTexGeniv	LIGHT0	Light used for embossing.	2.10.4	texture

When ARB_multitexture is supported, the Type column is per-texture unit.

(the TEXTURE_GEN_MODE type changes from 4xZ3 to 4xZ5)

New Implementation State

None

Revision History

2001/02/20 - Status changed to deprecated.

Name

NV_texgen_reflection

Name Strings

GL_NV_texgen_reflection

Notice

Copyright NVIDIA Corporation, 1999.
NVIDIA Proprietary.

Version

August 24, 1999

Number

179

Dependencies

Written based on the wording of the OpenGL 1.2 specification but not dependent on it.

Overview

This extension provides two new texture coordinate generation modes that are useful texture-based lighting and environment mapping. The reflection map mode generates texture coordinates (s,t,r) matching the vertex's eye-space reflection vector. The reflection map mode is useful for environment mapping without the singularity inherent in sphere mapping. The normal map mode generates texture coordinates (s,t,r) matching the vertex's transformed eye-space normal. The normal map mode is useful for sophisticated cube map texturing-based diffuse lighting models.

Issues

Should we place the normal/reflection vector in the (s,t,r) texture coordinates or (s,t,q) coordinates?

RESOLUTION: (s,t,r). Even if the proposed hardware uses "q" for the third component, the API should claim to support generation of (s,t,r) and let the texture matrix (through a concatenation with the user-supplied texture matrix) move "r" into "q".

Should you be able to have some texture coordinates computing REFLECTION_MAP_NV and others not? Same question with NORMAL_MAP_NV.

RESOLUTION: YES. This is the way that SPHERE_MAP works. It is not clear that this would ever be useful though.

Should something special be said about the handling of the q texture coordinate for this spec?

RESOLUTION: NO. But the following paragraph is useful for implementors concerned about the handling of q.

The REFLECTION_MAP_NV and NORMAL_MAP_NV modes are intended to supply reflection and normal vectors for cube map texturing hardware. When these modes are used for cube map texturing, the generated texture coordinates can be thought of as an reflection vector. The value of the q texture coordinate then simply scales the vector but does not change its direction. Because only the vector direction (not the vector magnitude) matters for cube map texturing, implementations are free to leave q undefined when any of the s, t, or r texture coordinates are generated using REFLECTION_MAP_NV or NORMAL_MAP_NV.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

NORMAL_MAP_NV	0x8511
REFLECTION_MAP_NV	0x8512

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain NORMAL_MAP_NV or REFLECTION_MAP_NV.

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP_NV, or NORMAL_MAP_NV."

Add these paragraphs after the 4th paragraph:

"If TEXTURE_GEN_MODE indicates REFLECTION_MAP_NV, compute the reflection vector r as described for the SPHERE_MAP mode. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = rx; the value assigned to a t coordinate is t = ry; and the value assigned to a r coordinate is r = rz. Calling TexGen with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV generates the error INVALID_ENUM.

If TEXTURE_GEN_MODE indicates NORMAL_MAP_NV, compute the normal vector n' as described in section 2.10.3. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = nfx; the value assigned to a t coordinate is t = nfy; and the value assigned to a r coordinate is r = nfz. (The values nfx, nfy, and nfz are the components of nf.) Calling TexGen with a <coord>

of Q when <pname> indicates REFLECTION_MAP_NV generates the error INVALID_ENUM.

The last paragraph's first sentence should be changed to:

"The state required for texture coordinate generation comprises a five-valued integer for each coordinate indicating coordinate generation mode, ..."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV or NORMAL_MAP_NV.

New State

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture

(the type changes from 4xZ3 to 4xZ5)

New Implementation State

None

Name

NV_texture_compression_vtc

Name Strings

GL_NV_texture_compression_vtc

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_compression_vtc.txt#2 $

Number

228

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

ARB_texture_compression is required.

EXT_texture_compression_s3tc is required.

Overview

This extension adds support for the VTC 3D texture compression formats, which are analogous to the S3TC texture compression formats, with the addition of some retiling in the Z direction. VTC has the same compression ratio as S3TC and uses 4x4x1, 4x4x2, or 4x4x4 blocks.

Issues

* *Should the enumerants' (1) values and (2) names be reused from the S3TC extension?*

  RESOLVED: Yes and yes. There is such a close correspondence between the formats that introducing new values or names would serve no purpose.

* *Should the block alignment restrictions differ in any way from the block alignment restrictions in the S3TC extension?*

  RESOLVED: No, except for the addition of the Z-direction block alignment restriction, which is analogous to the X and Y restrictions.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage3D and CompressedTexImage3DARB and the <format> parameter of CompressedTexSubImage2DARB:

COMPRESSED_RGB_S3TC_DXT1_EXT	0x83F0
COMPRESSED_RGBA_S3TC_DXT1_EXT	0x83F1
COMPRESSED_RGBA_S3TC_DXT3_EXT	0x83F2
COMPRESSED_RGBA_S3TC_DXT5_EXT	0x83F3

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

Modify the paragraph added to the end of the TexSubImage discussion (page 123) by EXT_texture_compression_s3tc to say:

"If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of several S3TC or VTC compressed texture image formats. Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D, TexSubImage3D, CopyTexSubImage2D, and CopyTexSubImage3D parameters are relaxed. These commands will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <depth> is not a multiple of four or equal to TEXTUR_DEPTH.
* <xoffset>, <yoffset>, or <zoffset> is not a multiple of four."

Modify the paragraph added to Section 3.8.2 "Alternate Image Specification" at the end of the CompressedTexImage section by EXT_texture_compression_s3tc to say:

"If <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the compressed texture is stored using one of several S3TC or VTC compressed texture image formats. The S3TC texture compression algorithm supports only 2D images without borders, while the VTC texture compression algorithm supports only 3D images without borders. CompressedTexImage1DARB produces an INVALID_ENUM error if <internalformat> is an S3TC/VTC format. CompressedTexImage2DARB and CompressedTexImage3DARB will produce an INVALID_OPERATION error if <border> is non-zero."

Modify the paragraph added to Section 3.8.2 "Alternate Image Specification" at the end of the CompressedTexSubImage section by EXT_texture_compression_s3tc to say:

"If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of several S3TC or VTC compressed texture image formats. Since these algorithms support only 2D and 3D images, CompressedTexSubImage1DARB produces an INVALID_ENUM error if <format> is an S3TC/VTC format. Since S3TC/VTC images are easily edited along 4x4 and 4x4x4 texel boundaries, the limitations on CompressedTexSubImage2D and CompressedTexSubImage3D are relaxed. CompressedTexSubImage2D and CompressedTexSubImage3D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
* <depth> is not a multiple of four or equal to TEXTUR_DEPTH.
* <xoffset>, <yoffset>, or <zoffset> is not a multiple of four."

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None.

GLX Protocol

None.

Errors

The INVALID_ENUM error that was generated by CompressedTexImage3DARB if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT no longer occurs.

INVALID_OPERATION is generated by CompressedTexImage3DARB if if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and <border> is not equal to zero.

The INVALID_ENUM error that was generated by CompressedTexSubImage3DARB if <format> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT no

longer occurs.

INVALID_OPERATION is generated by TexSubImage3D, CopyTexSubImage3D, or CompressedTexSubImage3D if INTERNAL_FORMAT is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <depth> is not a multiple of four or equal to TEXTURE_DEPTH; <xoffset>, <yoffset>, or <zoffset> is not a multiple of four.

See also errors in the GL_ARB_texture_compression and GL_EXT_texture_compression_s3tc specifications.

New State

None.

Appendix

VTC Compressed Texture Image Formats

Each VTC compression format is similar to a corresponding S3TC compression format, but where an S3TC block encodes a 4x4 block of texels, a VTC block encodes a 4x4x1, 4x4x2, or 4x4x4 block of texels. If the depth of the image is four or greater, 4x4x4 blocks are used, and if the depth is 1 or 2, 4x4x1 or 4x4x2 blocks are used.

The size in bytes of a VTC image with dimensions w, h, and d is:

```
ceil(w/4) * ceil(h/4) * d * blocksize,
```

where blocksize is the size of an analogous 4x4 S3TC block and is either 8 or 16 bytes.

The block containing a texel at location (x,y,z) starts at an offset inside the image of:

```
blocksize * min(d,4) * (floor(x/4) +
                        ceil(w/4) * (floor(y/4) +
                                     ceil(h/4) * floor(z/4)))
```

bytes.

A 4x4x1 block of each of the four formats is stored in exactly the same way that a 4x4 block of the analogous S3TC format is stored.

A 4x4x2 or 4x4x4 block is stored as two or four consecutive 4x4 blocks of the analogous S3TC format, one for each layer inside the block. For example, a 4x4x2 DXT1 block consists of 16 bytes in total. The first 8 bytes encode the texels at locations (0,0,0) through (3,3,0), and the second 8 bytes encode the texels at locations (0,0,1) through (3,3,1).

For definitions of the S3TC formats, please refer to the EXT_texture_compression_s3tc specification.

Revision History

none yet

Name

NV_texture_env_combine4

Name Strings

GL_NV_texture_env_combine4

Notice

Copyright NVIDIA Corporation, 1999, 2000, 2001.

IP Status

NVIDIA Proprietary.

Version

NVIDIA Date: January 18, 2001
$Date: 1999/06/21 13:54:17 $ $Revision: 1.2 $
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_env_combine4.txt#13 $

Number

195

Dependencies

EXT_texture_env_combine is required and is modified by this extension
ARB_multitexture affects the definition of this extension

Overview

New texture environment function COMBINE4_NV allows programmable
texture combiner operations, including

ADD	Arg0 * Arg1 + Arg2 * Arg3
ADD_SIGNED_EXT	Arg0 * Arg1 + Arg2 * Arg3 - 0.5

where Arg0, Arg1, Arg2 and Arg3 are derived from

ZERO	the value 0
PRIMARY_COLOR_EXT	primary color of incoming fragment
TEXTURE	texture color of corresponding texture unit
CONSTANT_EXT	texture environment constant color
PREVIOUS_EXT	result of previous texture environment; on texture unit 0, this maps to PRIMARY_COLOR_EXT
TEXTURE<n>_ARB	texture color of the <n>th texture unit

In addition, the result may be scaled by 1.0, 2.0 or 4.0.

Issues

None

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

COMBINE4_NV	0x8503

Accepted by the <pname> parameter of GetTexEnvfv, GetTexEnviv, TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <target> parameter value is TEXTURE_ENV

SOURCE3_RGB_NV	0x8583
SOURCE3_ALPHA_NV	0x858B
OPERAND3_RGB_NV	0x8593
OPERAND3_ALPHA_NV	0x859B

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE3_RGB_NV, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT, SOURCE2_ALPHA_EXT, or SOURCE3_ALPHA_NV

ZERO
TEXTURE<n>_ARB

where <n> is in the range 0 to NUMBER_OF_TEXTURE_UNITS_ARB-1.

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_RGB_EXT, OPERAND1_RGB_EXT, OPERAND2_RGB_EXT or OPERAND3_RGB_NV

SRC_COLOR
ONE_MINUS_SRC_COLOR
SRC_ALPHA
ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_ALPHA_EXT, OPERAND1_ALPHA_EXT, OPERAND2_ALPHA_EXT, or OPERAND3_ALPHA_NV

SRC_ALPHA
ONE_MINUS_SRC_ALPHA

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE4_NV, the form of the texture function depends on the values of COMBINE_RGB_EXT and

COMBINE_ALPHA_EXT, according to table 3.21. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1]. If the value of COMBINE_RGB_EXT or COMBINE_ALPHA_EXT is not one of the listed values, the result is undefined.

COMBINE_RGB_EXT or COMBINE_ALPHA_EXT	Texture Function
ADD	Arg0 * Arg1 + Arg2 * Arg3
ADD_SIGNED_EXT	Arg0 * Arg1 + Arg2 * Arg3 - 0.5

Table 3.21: COMBINE4_NV texture functions

The arguments Arg0, Arg1, Arg2 and Arg3 are determined by the values of SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. Ct<n> and At<n> are the filtered texture RGB and alpha values from the texture bound to the <n>th texture unit. If the <n>th texture unit is disabled, the value of each component is 1. The relationship is described in tables 3.22 and 3.23.

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
ZERO	SRC_COLOR	0
	ONE_MINUS_SRC_COLOR	1
	SRC_ALPHA	0
	ONE_MINUS_SRC_ALPHA	1
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)
TEXTURE<n>_ARB	SRC_COLOR	Ct<n>
	ONE_MINUS_SRC_COLOR	(1-Ct<n>)
	SRC_ALPHA	At<n>
	ONE_MINUS_SRC_ALPHA	(1-At<n>)

Table 3.22: Arguments for COMBINE_RGB_EXT functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
ZERO	SRC_ALPHA	0
	ONE_MINUS_SRC_ALPHA	1
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)
TEXTURE<n>_ARB	SRC_ALPHA	At<n>
	ONE_MINUS_SRC_ALPHA	(1-At<n>)

Table 3.23: Arguments for COMBINE_ALPHA_EXT functions

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

Additions to Chapter 6 of the GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE3_RGB_NV, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT, SOURCE2_ALPHA_EXT or SOURCE3_ALPHA_NV is not one of ZERO, TEXTURE, CONSTANT_EXT, PRIMARY_COLOR_EXT, PREVIOUS_EXT or TEXTURE<n>_ARB, where <n> is in the range 0 to NUMBER_OF_TEXTURE_UNITS_ARB-1.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_EXT, OPERAND1_RGB_EXT, OPERAND2_RGB_EXT or OPERAND3_RGB_NV is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_EXT OPERAND1_ALPHA_EXT, OPERAND2_ALPHA_EXT, or OPERAND3_ALPHA_NV is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

Modifications to EXT_texture_env_combine

This extension relaxes the restrictions on SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT for use with EXT_texture_env_combine. All params specified by Table 3.22 and Table 3.23 are valid.

Dependencies on ARB_multitexture

If ARB_multitexture is not implemented, all references to TEXTURE<n>_ARB and NUMBER_OF_TEXTURE_UNITS_ARB are deleted.

New State

Get Value	Get Command	Type	Initial Value	Attribute
SOURCE3_RGB_NV	GetTexEnviv	n x Z5+n	ZERO	texture
SOURCE3_ALPHA_NV	GetTexEnviv	n x Z5+n	ZERO	texture
OPERAND3_RGB_NV	GetTexEnviv	n x Z2	ONE_MINUS_SRC_COLOR	texture
OPERAND3_ALPHA_NV	GetTexEnviv	n x Z2	ONE_MINUS_SRC_ALPHA	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

Because of a hardware limitation, TNT, TNT2, GeForce, and Quadro treat "scale by 4.0" with the COMBINE_RGB_EXT or COMBINE_ALPHA_EXT mode of ADD_SIGNED_EXT as "scale by 2.0".

Name

NV_texture_rectangle

Name Strings

GL_NV_texture_rectangle

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Implemented in NVIDIA's Release 10 drivers.

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_rectangle.txt#2 $

Number

229

Dependencies

Written based on the OpenGL 1.2.1 specification including ARB_texture_cube_map wording.

IBM_mirrored_repeat affects the definition of this extension.

ARB_texture_border_clamp affects the definition of this extension.

EXT_paletted_texture affects the definition of this extension.

This extension affects the definition of the NV_texture_shader extension.

Overview

OpenGL texturing is limited to images with power-of-two dimensions and an optional 1-texel border. NV_texture_rectangle extension adds a new texture target that supports 2D textures without requiring power-of-two dimensions.

Non-power-of-two dimensioned textures are useful for storing video images that do not have power-of-two dimensions. Re-sampling artifacts are avoided and less texture memory may be required by using non-power-of-two dimensioned textures. Non-power-of-two dimensioned textures are also useful for shadow maps and window-space texturing.

However, non-power-of-two dimensioned (NPOTD) textures have

limitations that do not apply to power-of-two dimensioned (POT) textures. NPOTD textures may not use mipmap filtering; POTD textures support both mipmapped and non-mipmapped filtering. NPOTD textures support only the GL_CLAMP, GL_CLAMP_TO_EDGE, and GL_CLAMP_TO_BORDER_ARB wrap modes; POTD textures support GL_CLAMP_TO_EDGE, GL_REPEAT, GL_CLAMP, GL_MIRRORED_REPEAT_IBM, and GL_CLAMP_TO_BORDER. NPOTD textures do not support an optional 1-texel border; POTD textures do support an optional 1-texel border.

POTD textures are accessed by non-normalized texture coordinates. So instead of thinking of the texture image lying in a [0..1]x[0..1] range, the NPOTD texture image lies in a [0..w]x[0..h] range.

This extension adds a new texture target and related state (proxy, binding, max texture size).

Issues

Should rectangular textures simply be an extension to the 2D texture target that allows non-power-of-two widths and heights?

RESOLUTION: No. The rectangular texture is an entirely new texture target type called GL_TEXTURE_RECTANGLE_NV. This is because while the texture rectangle target relaxes the power-of-two dimensions requirements of the texture 2D target, it also has limitations such as the absence of both mipmapping and the GL_REPEAT and GL_MIRRORED_REPEAT_IBM wrap modes. Additionally, rectangular textures do not use [0..1] normalized texture coordinates.

How is the image of a rectangular texture specified?

RESOLUTION: Using the standard OpenGL API for specifying a 2D texture image: glTexImage2D, glSubTexImage2D, glCopyTexImage2D, and glCopySubTexImage2D. The target for these commands is GL_TEXTURE_RECTANGLE_NV though.

This is similar to how the ARB_texture_cube_map extension uses the 2D texture image specification API though with its own texture target.

Should 3D textures be allowed to be NPOTD?

RESOLUTION: No. That should be left to another extension.

Should cube map textures be allowed to be NPOTD?

RESOLUTION: No. Probably not particularly interesting for cube maps. If it becomes important, another extension should provide NPOTD cube maps.

Should 1D textures be allowed to be NPOTD?

RESOLUTION: No. Rectangular textures are always considered 2D by this extension. You can always simulate a 1D NPOTD textures by using a 2D Wx1 or 1xH dimensioned rectangular texture.

Should anything be said about performance?

RESOLUTION: No, but developers should not be surprised if conventional POTD textures will render slightly faster than NPOTD textures. This is particularly likely to be true when NPOTD textures are minified leading to texture cache thrashing.

How are rectangular textures enabled?

RESOLUTION: Since rectangular textures add a new texture target, you enable rectangular textures by enabling this target. Example:

```
glEnable(GL_TEXTURE_RECTANGLE_NV);
```

What is the priority of the rectangular texture target enable relative to existing texture enables?

RESOLUTION: The texture rectangle target is like a 2D texture in many ways so its enable priority is just above GL_TEXTURE_2D. From lowest priority to highest priority: GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_ARB.

What is the default wrap state for a texture rectangle?

RESOLUTION: GL_CLAMP_TO_EDGE. The normal default wrap state is GL_REPEAT, but that mode is not allowed for rectangular textures?

What is the default minification filter for a texture rectangle?

RESOLUTION: GL_LINEAR. The normal default minification filter state is GL_NEAREST_MIPMAP_LINEAR, but that mode is not allowed for rectangular textures because mipmapping is not supported.

Do paletted textures work with rectangular textures?

RESOLUTION: No. Similar (but not identical) functionality can be accomplished using dependent texture shader operations (see NV_texture_shader).

The difference between paletted texture accesses and dependent texture accesses is that paletted texture lookups are "pre-filtering" while dependent texture shader operations are "post-filtering".

Can compressed texture images be specified for a rectangular texture?

RESOLUTION: The generic texture compression internal formats introduced by ARB_texture_compression are supported for rectangular textures because the image is not presented as compressed data and the ARB_texture_compression extension always permits generic texture compression internal formats to be stored in uncompressed form. Implementations are free to support generic compression internal formats for rectangular textures if supported but such support is not required.

This extensions makes a blanket statement that specific compressed internal formats for use with CompressedTexImage<n>DARB are NOT supported for rectangular textures. This is because several

existing hardware implementations of texture compression formats such as S3TC are not designed for compressing rectangular textures. This does not preclude future texture compression extensions from supporting compressed internal formats that do work with rectangular extensions (by relaxing the current blanket error condition).

Does this extension work with SGIX_shadow-style shadow mapping?

RESOLUTION: Yes. The one non-obvious allowance to support SGIX_shadow-style shadow mapping is that the R texture coordinate wrap mode remains UNCHANGED for rectangular textures. Clamping of the R texture coordinate for rectangular textures uses the standard [0,1] interval rather than the [0,ws] or [0,hs] intervals as in the case of S and T. This is because R represents a depth value in the [0,1] range whether using a 2D or rectangular texture.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of Enable, Disable, IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindTexture, GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameteri, TexParameterfv, and TexParameteriv:

TEXTURE_RECTANGLE_NV 0x84F5

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

TEXTURE_BINDING_RECTANGLE_NV 0x84F6

Accepted by the <target> parameter of GetTexImage, GetTexLevelParameteriv, GetTexLevelParameterfv, TexImage2D, CopyTexImage2D, TexSubImage2D, and CopySubTexImage2D:

TEXTURE_RECTANGLE_NV

Accepted by the <target> parameter of GetTexLevelParameteriv, GetTexLevelParameterfv, GetTexParameteriv, and TexImage2D:

PROXY_TEXTURE_RECTANGLE_NV 0x84F7

Accepted by the <pname> parameter of GetBooleanv, GetDoublev, GetIntegerv, and GetFloatv:

MAX_RECTANGLE_TEXTURE_SIZE_NV 0x84F8

Additions to Chapter 2 of the GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the GL Specification (Rasterization)

-- **Section 3.6.3 "Pixel Transfer Modes" under "Color Table Specification" or the ColorTableEXT description in the EXT_paletted_texture specification (rev 1.2)**

Add the following statement after introducing ColorTableEXT:

"The error INVALID_ENUM is generated if the target to ColorTable (or ColorTableEXT or the various ColorTable and ColorTableEXT alternative commands) is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV."

-- **Section 3.6.5 "Pixel Transfer Operations" under "Convolution"**

Change this paragraph (page 103) to add TEXTURE_RECTANGLE_NV to the list of targets so it reads say:

... "If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to DrawPixels, CopyPixels, ReadPixels, TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, and CopyTexSubImage3D, and returned by GetTexImage with one of the targets TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB."

-- **Section 3.8.1 "Texture Image Specification"**

Change the second sentence through the rest of the paragraph describing TexImage2D on page 116 to:

"<target> must be one of TEXTURE_2D for a 2D texture, or one of TEXTURE_RECTANGLE_NV for a rectangle texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB for a cube map texture. Additionally, <target> can be either PROXY_TEXTURE_2D for a 2D proxy texture or PROXY_TEXTURE_RECTANGLE_NV for a rectangle proxy texture or PROXY_TEXTURE_CUBE_MAP_ARB for a cube map proxy texture as discussed in section 3.8.7. The other parameters match the corresponding parameters of TexImage3D."

Add a following paragraph reading:

"Rectangular textures do not support paletted formats. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV and the format is COLOR_INDEX or the internalformat is COLOR_INDEX or one of the COLOR_INDEX<n>_EXT internal formats."

Change the 14th paragraph (page 116) to read:

"In a similar fashion, the maximum allowable width of a rectangular texture image, and the maximum allowable height of a rectangular texture image, must be at least the implementation-dependent value of MAX_RECTANGLE_TEXTURE_SIZE_NV."

Add the following paragraph after the paragraph introducing TexImage2D (page 116):

"When the target is TEXTURE_RECTANGLE_NV, the INVALID_VALUE error is generated if border is any value other than zero or the level is any value other than zero. Also when the target is TEXTURE_RECTANGLE_NV, the texture dimension restrictions specified by equations 3.11, 3.12, and 3.13 are ignored; however, if the width is less than zero or the height is less than zero, the error INVALID_VALUE is generated. In the case of a rectangular texture, ws and hs equal the specified width and height respectively of the rectangular texture image while ds is 1."

Amend the following paragraph that was added by the ARB_texture_cube_map specification after the first paragraph on page 117:

"A 2D texture consists of a single 2D texture image. A rectangle texture consists of a single 2D texture image. A cube map texture is a set of six 2D texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_*_ARB targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map 2D image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X_ARB are used when specifying, updating, or querying one of a cube map's six 2D image, but when enabling cube map texturing or binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular 2D image), the TEXTURE_CUBE_MAP_ARB target is specified."

Append to the end of the third to the last paragraph in the section (page 118):

"A rectangular texture array has depth dt=1, with height ht and width wt defined by the specified image height and width parameters."

-- **Section 3.8.2 "Alternate Texture Image Specification Commands"**

Add TEXTURE_RECTANGLE_NV to the second paragraph (page 120) to say:

... "Currently, <target> must be TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB." ...

Add TEXTURE_RECTANGLE_NV to the fourth paragraph (page 121) to say:

... "Currently the target arguments of TexSubImage1D and CopyTexSubImage1D must be TEXTURE_1D, the <target> arguments of TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, and the <target> arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D." ...

Also add to the end of the fourth paragraph (121):

"If target is TEXTURE_RECTANGLE_NV and level is not zero, the error INVALID_VALUE is generated."

-- **Section "Compressed Texture Images" in the ARB_texture_compression specification**

Add the following paragraph after introducing the CompressedTexImage<n>DARB commands:

"The error INVALID_ENUM is generated if the target parameter to one of the CompressedTexImage<n>DARB commands is TEXTURE_RECTANGLE_NV."

Add the following paragraph after introducing the CompressedTexSubImage<n>DARB commands:

"The error INVALID_ENUM is generated if the target parameter to one of the CompressedTexSubImage<n>DARB commands is TEXTURE_RECTANGLE_NV."

-- **Section 3.8.3 "Texture Parameters"**

Add TEXTURE_RECTANGLE_NV to paragraph one (page 124) to say:

... "<target> is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB." ...

Add the following paragraph to the end of the section (page 134):

"Certain texture parameter values may not be specified for textures with a target of TEXTURE_RECTANGLE_NV. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT or MIRRORED_REPEAT_IBM. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted). The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and TEXTURE_BASE_LEVEL is set to any value other than zero."

-- **Section 3.8.4 "Texture Wrap Modes"**

Add this final additional paragraph:

"Texture coordinates are clamped differently for rectangular textures. The r texture coordinate is wrapped as described above (as required for shadow mapping to operate correctly). When the texture target is TEXTURE_RECTANGLE_NV, the s and t coordinates are wrapped as follows: CLAMP causes the s coordinate to be clamped to the range [0,ws]. CLAMP causes the t coordinate to be clamped to the range [0,hs]. CLAMP_TO_EDGE causes the s coordinate to be clamped to the range [0.5,ws-0.5]. CLAMP_TO_EDGE causes the t coordinate to be clamped to the range [0.5,hs-0.5]. CLAMP_TO_BORDER_ARB causes the s coordinate to be clamped to the range [-0.5,ws+0.5]. CLAMP_TO_BORDER_ARB causes the t coordinate to be clamped to the

range [-0.5,hs+0.5]."

-- **Section 3.8.5 "Texture Minification" under "Mipmapping"**

Change the second full paragraph on page 126 to read:

"For non-rectangular textures, let u(x,y) = 2^n*s(x,y), v(x,y) = 2^m*t(x,y), and w(x,y) = 2^l*r(x,y), where n, m, and l are defined by equations 3.11, 3.12, and 3.13 with ws, hs, and ds equal to the width, height, and depth of the image array whose level is TEXTURE_BASE_LEVEL. However, for rectangular textures let u(x,y) = s(x,y), v(x,y) = t(x,y), and w(x,y) = r(x,y)."

Update the last sentence in the first full paragraph on page 127 to read:

"Depending on whether the texture's target is rectangular or non-rectangular, this means the texel at location (i,j,k) becomes the texture value, with i given by

$$i = \begin{cases} \text{floor}(u), & s < 1 \\ 2^n-1, & s == 1, \text{ non-rectangular texture} \\ ws-1, & s == 1, \text{ rectangular texture} \end{cases} \quad (3.17)$$

(Recall that if TEXTURE_WRAP_S is REPEAT, then 0 <= s < 1.) Similarly, j is found as

$$j = \begin{cases} \text{floor}(v), & t < 1 \\ 2^m-1, & t == 1, \text{ non-rectangular texture} \\ hs-1, & t == 1, \text{ rectangular texture} \end{cases} \quad (3.18)$$

and k is found as

$$k = \begin{cases} \text{floor}(w), & r < 1 \\ 2^l-1, & r == 1, \text{ non-rectangular texture} \\ 0, & r == 1, \text{ rectangular texture} \end{cases} \quad (3.19)$$

Change the last sentence in the partial paragraph after equation 3.19 to read:

"For a two-dimensional or rectangular texture, k is irrelevant; the texel at location (i,j) becomes the texture value."

Change the sentence preceding equation 3.20 (page 128) specifying how to compute the value tau for a two-dimensional texture to:

"For a two-dimensional or rectangular texture,"

Follow the first full paragraph on page 130 with:

"Rectangular textures do not support mipmapping (it is an error to

specify a minification filter that requires mipmapping)."

-- **Section 3.8.7 "Texture State and Proxy State"**

Change the first sentence of the first paragraph (page 131) to say:

"The state necessary for texture can be divided into two categories. First, there are the ten sets of mipmap arrays (one each for the one-, two-, and three-dimensional texture targets, one for the rectangular texture target (though the rectangular texture target has only one mipmap level), and six for the cube map texture targets) and their number." ...

Change the fourth and third to last sentences of the first paragraph to say:

"In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR, except for rectangular textures where the initial value is LINEAR, and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r warp modes are all set to REPEAT, except for rectangular textures where the initial value is CLAMP_TO_EDGE."

Change the second paragraph (page 132) to say:

"In addition to the one-, two-, three-dimensional, rectangular, and the six cube map sets of image arrays, the partially instantiated one-, two-, and three-dimensional, rectangular, and one cube map sets of proxy image arrays are maintained." ...

Change the third paragraph (page 132) to:

"One- and two-dimensional and rectangular proxy arrays are operated on in the same way when TexImage1D is executed with target specified as PROXY_TEXTURE_1D, or TexImage2D is executed with target specified as PROXY_TEXTURE_2D or PROXY_TEXTURE_RECTANGLE_NV."

Change the second sentence of the fourth paragraph (page 132) to:

"Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_RECTANGLE_NV, PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP_ARB cannot be used as textures, and their images must never be queried using GetTexImage." ...

-- **Section 3.8.8 "Texture Objects"**

Change the first sentence of the first paragraph (page 132) to say:

"In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB, named one-dimensional, two-dimensional, rectangular, and three-dimensional texture objects and cube map texture objects can be created and operated on." ...

Change the second paragraph (page 132) to say:

"A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or

TEXTURE_CUBE_MAP_ARB." ... "If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, it remains a one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texture until it is deleted."

Change the third paragraph (page 133) to say:

"BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB."

Change paragraph five (page 133) to say:

"In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, and TEXTURE_CUBE_MAP have one-dimensional, two-dimensional, rectangular, three-dimensional, and cube map state vectors associated with them respectively." ... "The initial, one-dimensional, two-dimensional, rectangular, three-dimensional, and cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, and TEXTURE_CUBE_MAP_ARB respectively while 0 is bound to the corresponding targets."

Change paragraph six (page 133) to say:

... "If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB is deleted, it is as though BindTexture has been executed with the same <target> and <texture> zero." ...

-- **Section 3.8.10 "Texture Application"**

Replace the beginning sentences of the first paragraph (page 138) with:

"Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB to enable the one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texturing respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the rectangular and either of the two- or one-dimensional textures is enabled, the rectangular texture is used. If the three-dimensional and any of the rectangular, two-dimensional, or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-dimensional, rectangular, two-dimensional, or one-dimensional textures is enabled, then cube map texturing is used.

Additions to Chapter 4 of the GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the GL Specification (Special Functions)

-- **Section 5.4 "Display Lists"**

In the first paragraph (page 179), add PROXY_TEXTURE_RECTANGLE_NV to the list of PROXY_* tokens.

Additions to Chapter 6 of the GL Specification (State and State Requests)

-- **Section 6.1.3 "Enumerated Queries"**

Change the fourth paragraph (page 183) to say:

"The GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, or TEXTURE_CUBE_MAP_ARB, indicating the currently bound one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map texture object. For GetTexLevelParameter, <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_RECTANGLE_NV, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_RECTANGLE_NV, PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP_ARB, indicating the one-dimensional texture object, two-dimensional texture object, rectangular texture object, three-dimensional texture object, or one of the six distinct 2D images making up the cube map texture object or one-dimensional, two-dimensional, rectangular, three-dimensional, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP_ARB is not a valid <target> parameter for GetTexLevelParameter because it does not specify a particular cube map face."

-- **Section 6.1.4 "Texture Queries"**

Change the first paragraph (page 184) to read:

... "It is somewhat different from the other get commands; <tex> is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D indicates a one-dimensional texture, TEXTURE_2D indicates a two-dimensional texture, TEXTURE_RECTANGLE_NV indicates a rectangular texture, TEXTURE_3D indicates a three-dimensional texture, and TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, and TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB indicate the respective face of a cube map texture."

Add a final sentence to the fourth paragraph:

"Calling GetTexImage with a lod not zero when the tex is TEXTURE_RECTANGLE_NV causes the error INVALID_VALUE."

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on ARB_texture_border_clamp

If ARB_texture_border_clamp is not supported, references to the CLAMP_TO_BORDER_ARB wrap mode in this document should be ignored.

Dependencies on IBM_mirrored_repeat

If IBM_mirrored_repeat is not supported, references to the MIRRORED_REPEAT_IBM wrap mode in this document should be ignored.

Dependencies on EXT_paletted_texture

If EXT_paletted_texture is not supported, references to the COLOR_INDEX, COLOR_INDEX<n>_EXT, ColorTable, and ColorTableEXT should be ignored.

Dependencies on EXT_texture_compression_s3tc

If EXT_texture_compression_s3tc is not supported, references to CompressedTexImage2DARB and CompressedTexSubImageARB and the COMPRESSED_*_S3TC_DXT*_EXT enumerants should be ignored.

Errors

INVALID_ENUM is generated when ColorTable (or ColorTableEXT or the various ColorTable and ColorTableEXT alternative commands) is called and the target is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV.

INVALID_ENUM is generated when TexImage2D is called and the target is TEXTURE_RECTANGLE_NV or PROXY_TEXTURE_RECTANGLE_NV and the format is COLOR_INDEX or the internalformat is COLOR_INDEX or one of the COLOR_INDEX<n>_EXT internal formats.

INVALID_VALUE is generated when TexImage2D is called when the target is TEXTURE_RECTANGLE_NV if border is any value other than zero or the level is any value other than zero.

INVALID_VALUE is generated when TexImage2D is called when the target is TEXTURE_RECTANGLE_NV if the width is less than zero or the height is less than zero.

INVALID_VALUE is generated when TexSubImage2D or CopyTexSubImage2D is called when the target is TEXTURE_RECTANGLE_NV if the level is any value other than zero.

INVALID_ENUM is generated when one of the CompressedTexImage<n>DARB commands is called when the target parameter is TEXTURE_RECTANGLE_NV.

INVALID_ENUM is generated when one of the CompressedTexSubImage<n>DARB commands is called when the target parameter is TEXTURE_RECTANGLE_NV.

INVALID_ENUM is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T,

or TEXTURE_WRAP_R parameter is set to REPEAT or MIRRORED_REPEAT_IBM.

INVALID_ENUM is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR.

INVALID_VALUE is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_BASE_LEVEL is set to any value other than zero.

INVALID_VALUE is generated when GetTexImage is called with a lod not zero when the tex is TEXTURE_RECTANGLE_NV.

New State

(table 6.12, p202) amend/add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_RECTANGULAR_NV	B	IsEnabled	False	True if rectangular texturing is enabled	3.8.10	texture/enable
TEXTURE_BINDING_RECTANGLE_NV	Z+	GetIntegerv	0	Texture object for TEXTURE_CUBE_MAP	3.8.8	texture
TEXTURE_RECTANGLE_NV	I	GetTexImage	see 3.8	rectangular texture image for lod 0	3.8	-

(table 6.13, p203) amend/add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_WRAP_S	5+xZ5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode S	3.8	texture
TEXTURE_WRAP_T	5+xZ5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode T	3.8	texture
TEXTURE_WRAP_R	5+xZ5	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode R	3.8	texture

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_RECTANGLE_TEXTURE_SIZE_NV	Z+	GetIntegerv	64	Maximum rectangular texture image dimension	3.8.1	-

Revision History

None

Name

NV_texture_shader

Name Strings

GL_NV_texture_shader

Notice

Copyright NVIDIA Corporation, 1999, 2000, 2001.

IP Status

NVIDIA Proprietary.

Version

NVIDIA Date: April 27, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_shader.txt#8 $

Number

230

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification.

Requires support for the ARB_multitexture extension.

Requires support for the ARB_texture_cube_map extension.

NV_register_combiners affects the definition of this extension.

EXT_texture_lod_bias trivially affects the definition of this extension.

ARB_texture_env_combine and/or EXT_texture_env_combine affect the definition of this extension.

NV_texture_env_combine4 affects the definition of this extension.

ARB_texture_env_add and/or EXT_texture_env_add affect the definition of this extension.

NV_texture_rectangle affects the definition of this extension.

NV_texture_shader2 depends on the definition of this extension.

Overview

Standard OpenGL and the ARB_multitexture extension define a straightforward direct mechanism for mapping sets of texture coordinates to filtered colors. This extension provides a more functional mechanism.

OpenGL's standard texturing mechanism defines a set of texture targets. Each texture target defines how the texture image is specified and accessed via a set of texture coordinates. OpenGL 1.0 defines the 1D and 2D texture targets. OpenGL 1.2 (and/or the EXT_texture3D extension) defines the 3D texture target. The ARB_texture_cube_map extension defines the cube map texture target. Each texture unit's texture coordinate set is mapped to a color using the unit's highest priority enabled texture target.

This extension introduces texture shader stages. A sequence of texture shader stages provides a more flexible mechanism for mapping sets of texture coordinates to texture unit RGBA results than standard OpenGL.

When the texture shader enable is on, the extension replaces the conventional OpenGL mechanism for mapping sets of texture coordinates to filtered colors with this extension's sequence of texture shader stages.

Each texture shader stage runs one of 21 canned texture shader programs. These programs support conventional OpenGL texture mapping but also support dependent texture accesses, dot product texture programs, and special modes. (3D texture mapping texture shader operations are NOT provided by this extension; 3D texture mapping texture shader operations are added by the NV_texture_shader2 extension that is layered on this extension. See the NV_texture_shader2 specification.)

To facilitate the new texture shader programs, this extension introduces several new texture formats and variations on existing formats. Existing color texture formats are extended by introducing new signed variants. Two new types of texture formats (beyond colors) are also introduced. Texture offset groups encode two signed offsets, and optionally a magnitude or a magnitude and an intensity. The new HILO (pronounced high-low) formats provide possibly signed, high precision (16-bit) two-component textures.

Each program takes as input the stage's interpolated texture coordinate set (s,t,r,q). Each program generates two results: a shader stage result that may be used as an input to subsequent shader stage programs, and a texture unit RGBA result that becomes the texture color used by the texture unit's texture environment function or becomes the initial value for the corresponding texture register for register combiners. The texture unit RGBA result is always an RGBA color, but the shader stage result may be one of an RGBA color, a HILO value, a texture offset group, a floating-point value, or an invalid result. When both results are RGBA colors, the shader stage result and the texture unit RGBA result are usually identical (though not in all cases).

Additionally, certain programs have a side-effect such as culling the fragment or replacing the fragment's depth value.

The twenty-one programs are briefly described:

<none>

1. NONE - Always generates a (0,0,0,0) texture unit RGBA result. Equivalent to disabling all texture targets in conventional OpenGL.

<conventional textures>

2. TEXTURE_1D - Accesses a 1D texture via (s/q).

3. TEXTURE_2D - Accesses a 2D texture via (s/q,t/q).

4. TEXTURE_RECTANGLE_NV - Accesses a rectangular texture via (s/q,t/q).

5. TEXTURE_CUBE_MAP_ARB - Accesses a cube map texture via (s,t,r).

<special modes>

6. PASS_THROUGH_NV - Converts a texture coordinate (s,t,r,q) directly to a [0,1] clamped (r,g,b,a) texture unit RGBA result.

7. CULL_FRAGMENT_NV - Culls the fragment based on the whether each (s,t,r,q) is "greater than or equal to zero" or "less than zero".

<offset textures>

8. OFFSET_TEXTURE_2D_NV - Transforms the signed (ds,dt) components of a previous texture unit by a 2x2 floating-point matrix and then uses the result to offset the stage's texture coordinates for a 2D non-projective texture.

9. OFFSET_TEXTURE_2D_SCALE_NV - Same as above except the magnitude component of the previous texture unit result scales the red, green, and blue components of the unsigned RGBA texture 2D access.

10. OFFSET_TEXTURE_RECTANGLE_NV - Similar to OFFSET_TEXTURE_2D_NV except that the texture access is into a rectangular non-projective texture.

11. OFFSET_TEXTURE_RECTANGLE_SCALE_NV - Similar to OFFSET_TEXTURE_2D_SCALE_NV except that the texture access is into a rectangular non-projective texture.

<dependent textures>

12. DEPENDENT_AR_TEXTURE_2D_NV - Converts the alpha and red components of a previous shader result into an (s,t) texture coordinate set to access a 2D non-projective texture.

13. DEPENDENT_GB_TEXTURE_2D_NV - Converts the green and blue components of a previous shader result into an (s,t) texture coordinate set to access a 2D non-projective texture.

<dot product textures>

14. DOT_PRODUCT_NV - Computes the dot product of the texture shader's texture coordinate set (s,t,r) with some mapping of the components of a previous texture shader result. The component mapping depends on the type (RGBA or HILO) and signedness of the stage's previous texture input. Other dot product texture programs use the result of this program to compose a texture coordinate set for a dependent texture access. The color result is undefined.

15. DOT_PRODUCT_TEXTURE_2D_NV - When preceded by a DOT_PRODUCT_NV program in the previous texture shader stage, computes a second similar dot product and composes the two dot products into (s,t) texture coordinate set to access a 2D non-projective texture.

16. DOT_PRODUCT_TEXTURE_RECTANGLE_NV - Similar to DOT_PRODUCT_TEXTURE_2D_NV except that the texture acces is into a rectangular non-projective texture.

17. DOT_PRODUCT_TEXTURE_CUBE_MAP_NV - When preceded by two DOT_PRODUCT_NV programs in the previous two texture shader stages, computes a third similar dot product and composes the three dot products into (s,t,r) texture coordinate set to access a cube map texture.

18. DOT_PRODUCT_REFLECT_CUBE_MAP_NV - When preceded by two DOT_PRODUCT_NV programs in the previous two texture shader stages, computes a third similar dot product and composes the three dot products into a normal vector (Nx,Ny,Nz). An eye vector (Ex,Ey,Ez) is composed from the q texture coordinates of the three stages. A reflection vector (Rx,Ry,Rz) is computed based on the normal and eye vectors. The reflection vector forms an (s,t,r) texture coordinate set to access a cube map texture.

19. DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV - Operates like DOT_PRODUCT_REFLECT_CUBE_MAP_NV except that the eye vector (Ex,Ey,Ez) is a user-defined constant rather than composed from the q coordinates of the three stages.

20. DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV - When used instead of the second DOT_PRODUCT_NV program preceding a DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV stage, the normal vector forms an (s,t,r) texture coordinate set to access a cube map texture.

<dot product depth replace>

21. DOT_PRODUCT_DEPTH_REPLACE_NV - When preceded by a DOT_PRODUCT_NV program in the previous texture shader stage, computes a second similar dot product and replaces the fragment's window-space depth value with the first dot product results divided by the second. The texture unit RGBA result is (0,0,0,0).

Issues

What should this extension be called? How does the functionality compare with DirectX 8's pixel shaders?

RESOLUTION: This extension is called NV_texture_shader.

DirectX 8 refers to its similar functionality as "pixel shaders". However, DirectX 8 lumps both the functionality described in this extension and additional functionality similar to the functionality in the NV_register_combiners extension together into what DirectX 8 calls pixel shaders. This is confusing in two ways.

1) Pixels are not being shaded. In fact, the DirectX 8 pixel shaders functionality is, taken as a whole, shading only fragments (though Direct3D tends not to make the same clear distinction between fragments and pixels that OpenGL consistently makes).

2) There are two very distinct tasks being performed.

   First, there is the task of interpolated texture coordinate mapping. This per-fragment task maps from interpolated floating-point texture coordinate sets to (typically fixed-point) texture unit RGBA results. In conventional OpenGL, this mapping is performed by accessing the highest priority enabled texture target using the fragment's corresponding interpolated texture coordinate set. This NV_texture_shader extension provides a significantly more powerful mechanism for performing this mapping.

   Second, there is the task of fragment coloring. Fragment coloring is process of combining (typically fixed-point) RGBA colors to generate a final fragment color that, assuming the fragment is not discarded by subsequent per-fragment tests, is used to update the fragment's corresponding pixel in the frame buffer. In conventional OpenGL, fragment coloring is performed by the enabled texture environment functions, fog, and color sum operations. NVIDIA's register combiners functionality (see the NV_register_combiners and NV_register_combiners2 extensions) provides a substantially more powerful alternative to conventional OpenGL fragment coloring.

DirectX 8 has two types of opcodes for pixel shaders. Texture address opcodes correspond to the first task listed above. Texture register opcodes correspond to the second task listed above.

NVIDIA OpenGL extensions maintain a clear distinction between these two tasks. The texture shaders functionality described in this specification corresponds to the first task listed above.

Here is the conceptual framework that NVIDIA OpenGL extensions use to describe shading: Shading is the process of assigning colors to pixels, fragments, or texels. The texture shaders functionality assigns colors to texture unit results (essentially texture shading). These texture unit RGBA results can be used by fragment coloring (fragment shading). The resulting fragments are used to

update pixels (pixel shading) possibly via blending and/or multiple rendering passes.

The goal of these individual shading operations is per-pixel shading. Per-pixel shading is accomplished by combining the texture shading, fragment shading, and pixel shading operations, possibly with multiple rendering passes.

Programmable shading is a style of per-pixel shading where the shading operations are expressed in a higher level of abstraction than "raw" OpenGL texture, fragment, and pixel shading operations. In our view, programmable shading does not necessarily require a "pixel program" to be downloaded and executed per-pixel by graphics hardware. Indeed, there are many disadvantages to such an approach in practice. An alternative view of programmable shading (the one that we are promoting) treats the OpenGL primitive shading operations as a SIMD machine and decomposes per-pixel shading programs into one or more OpenGL rendering passes that map to "raw" OpenGL shading operations. We believe that conventional OpenGL combined with NV_register_combiners and NV_texture_shader (and further augmented by programmable geometry via NV_vertex_program and higher-order surfaces via NV_evaluators) can become the hardware basis for a powerful programmable shading system.

The roughly equivalent functionality to DirectX 8's pixel shaders in OpenGL is the combination of NV_texture_shader with NV_register_combiners.

Is anyone working on programmable shading using the NV_texture_shader functionality?

Yes. The Stanford Shading Group is actively working on support for programmable shading using NV_texture_shader, NV_register_combiners, and other extensions as the hardware basis for such a system.

What terms are important to this specification?

texture shaders - A series of texture shader stages that map texture coordinate sets to texture unit RGBA results. An alternative to conventional OpenGL texturing.

texture coordinate set - The interpolated (s,t,r,q) value for a particular texture unit of a particular fragment.

conventional OpenGL texturing - The conventional mechanism used by OpenGL to map texture coordinate sets to texture unit RGBA results whereby a given texture unit's texture coordinate set is used to access the highest priority enabled texture target to generate the texture unit's RGBA result. Conventional OpenGL texturing supports 1D, 2D, 3D, and cube map texture targets. In conventional OpenGL texturing each texture unit operates independently.

texture target type - One of the four texture target types: 1D, 2D, 3D, and cube map. (Note that NV_texture_shader does NOT provide support for 3D textures; the NV_texture_shader2 extension adds texture shader operations for 3D texture targets.)

texture internal format - The internal format of a particular
texture object. For example, GL_RGBA8, GL_SIGNED_RGBA8, or
GL_SIGNED_HILO16_NV.

texture format type - One of the three texture format types: RGBA,
HILO, or texture offset group.

texture component signedness - Whether or not a given component
of a texture's texture internal format is signed or not.
Signed components are clamped to the range [-1,1] while unsigned
components are clamped to the range [0,1].

texture shader enable - The OpenGL enable that determines whether
the texture shader functionality (if enabled) or conventional
OpenGL texturing functionality (if disabled) is used to map texture
coordinate sets to texture unit RGBA results. The enable's initial
state is disabled.

texture shader stage - Each texture unit has a corresponding texture
shader stage that can be loaded with one of 21 texture shader
operations. Depending on the stage's texture shader operation,
a texture shader stage uses the texture unit's corresponding
texture coordinate set and other state including the texture shader
results of previous texture shader stages to generate the stage's
particular texture shader result and texture unit RGBA result.

texture unit RGBA result - A (typically fixed-point) color result
generated by either a texture shader or conventional OpenGL
texturing. This is the color that becomes the texture unit's
texture environment function texture input or the initial value
of the texture unit's corresponding texture register in the case
of register combiners.

texture shader result - The result of a texture shader stage that
may be used as an input to a subsequent texture shader stage.
This result is distinct from the texture unit RGBA result.
The texture shader result may be one of four types: an RGBA
color value, a HILO value, a texture offset group value, or a
floating-point value. A few texture shader operations are defined
to always generate an invalid texture shader result.

texture shader result type - One of the four texture shader result
types: RGBA color, HILO, texture offset group, or floating-point.

texture shader operation - One of 21 fixed programs that maps a
texture unit's texture coordinate set to a texture shader result
and a texture unit RGBA result.

texture consistency - Whether or not the texture object for a
given texture target is consistent. The rules for determining
consistency depend on the texture target and the texture object's
filtering state. For example, a mipmapped texture is inconsistent
if its texture levels do not form a consistent mipmap pyramid.
Also, a cube map texture is inconsistent if its (filterable)
matching cube map faces do not have matching dimensions.

texture shader stage consistency - Whether or not a texture shader stage is consistent or not. The rules for determining texture shader stage consistency depend on the texture shader stage operation and the inputs upon which the texture shader operation depends. For example, texture shader operations that depend on accessing a given texture target are not consistent if the given texture target is not consistent. Also, a texture shader operation that depends on a particular texture shader result type for a previous texture shader result is not consistent if the previous texture shader result type is not appropriate or the previous texture shader stage itself is not consistent. If a texture shader stage is not consistent, it operates as if the operation is the GL_NONE operation.

previous texture input - Some texture shader operations depend on a texture shader result from a specific previous texture input designated by the GL_PREVIOUS_TEXTURE_INPUT_NV state.

What should the default state be?

RESOLUTION: Texture shaders disabled with all stages set to GL_NONE.

How is the mipmap lambda parameter computed for dependent texture fetches?

RESOLUTION: Very carefully. NVIDIA's implementation details are NVIDIA proprietary, but mipmapping of dependent texture fetches is supported.

Does this extension support so-called "bump environment mapping"?

Something similar to DirectX 6 so-called bump environment mapping can be emulated with the GL_OFFSET_TEXTURE_2D_NV texture shader.

A more correct form of bump environment mapping can be implemented by using the following texture shaders:

```
texture unit 0: GL_TEXTURE_2D
texture unit 1: GL_DOT_PRODUCT_NV
texture unit 2: GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV
texture unit 3: GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV
```

Texture unit 0 should use a normal map for its 2D texture. A GL_SIGNED_RGB texture can encode signed tangent-space normal perturbations. Or for more precision, a GL_SIGNED_HILO_NV texture can encode the normal perturbations in hemisphere fashion.

The tangent (Tx,Ty,Tz), binormal (Bx,By,Bz), and normal (Nx,Ny,Nz) that together map tangent-space normals to cube map-space normals should be sent as texture coordinates s1, t1, r1, s2, t2, r2, s3, t3, and r3 respectively. Typically, cube map space is aligned to match world space.

The (unnormalized) cube map-space eye vector (Ex,Ey,Ez) should be sent as texture coordinates q1, q2, and q3 respectively.

A vertex programs (using the NV_vertex_program extension) can compute and assign the required tangent, binormal, normal, and

eye vectors to the appropriate texture coordinates. Conventional OpenGL evaluators (or the NV_evaluators extension) can be used to evaluate the tangent and normal automatically for Bezier patches. The binormal is the cross product of the normal and tangent.

Texture units 1, 2, and 3, should also all specify GL_TEXTURE0_ARB (the texture unit accessing the normal map) for their GL_PREVIOUS_TEXTURE_INPUT_NV parameter.

The three dot product texture shader operations performed by the texture shaders for texture units 1, 2, and 3 form a 3x3 matrix that transforms the tangent-space normal (the result of the texture shader for texture unit 0). This rotates the tangent-space normal into a cube map-space.

Texture unit 2's cube map texture should encode a pre-computed diffuse lighting solution. Texture unit 3's cube map texture should encode a pre-computed specular lighting solution. The specular lighting solution can be an environment map.

Texture unit 2 is accessed using the cube map-space normal vector resulting from the three dot product results of the texture shaders for texture units 1, 2, and 3. (While normally texture shader operations are executed in order, preceding GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV by GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV is a special case where a dot product result from texture unit 3 influences the cube map access of texture unit 2.)

Texture unit 3 is accessed using the cube map-space reflection vector computed using the cube map-space normal vector from the three dot product results of the texture shaders for texture units 1, 2, and 3 and the cube-map space eye-vector (q1,q2,q3).

Note that using cube maps to access the diffuse and specular illumination obviates the need for an explicit normalization of the typically unnormalized cube map-space normal and reflection vectors.

The register combiners (using the NV_register_combiners extension) can combine the diffuse and specular contribution available in the GL_TEXTURE2_ARB and GL_TEXTURE3_ARB registers respectively. A constant ambient contribution can be stored in a register combiner constant. The ambient contribution could also be folded into the diffuse cube map.

If desired, the diffuse and ambient contribution can be modulated by a diffuse material parameter encoded in the RGB components of the primary color.

If desired, the specular contribution can be modulated by a specular material parameter encoded in the RGB components of the secondary color.

Yes, this is all quite complicated, but the result is a true bump environment mapping technique with excellent accounting for normalization and per-vertex interpolated diffuse and specular

materials. An environment and/or an arbitrary number of distant or infinite lights can be encoded into the diffuse and specular cube maps.

Why must GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV be used only in conjunction with GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV? Why does the GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV stage rely on a result computed in the following stage?

Think of the GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV and GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV operations as forming a compound operation. The idea is to generate two cube map accesses based on a perturbed normal and reflection vector where the reflection vector is a function of the perturbed normal vector. To minimize the number of stages (three stages only) and reuse the internal computations involved, this is treated as a compound operation.

Note that the GL_DOT_PRODUCT_REFLECT_CUBE_MAP_NV vector can be preceded by two GL_DOT_PRODUCT_NV operations instead of a GL_DOT_PRODUCT_NV operation then a GL_DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV operation. This may be more efficient when only the cube map access using the reflection vector is required (a shiny object without any diffuse reflectance).

Also note that if only the diffuse reflectance cube map access is required, this can be accomplished by simply using the GL_DOT_PRODUCT_CUBE_MAP_NV operation preceded by two GL_DOT_PRODUCT_NV operations.

How do texture shader stages map to register combiner texture registers?

RESOLUTION: If GL_TEXTURE_SHADER_NV is enabled, the texture unit RGBA result for a each texture stage is used to initialize the respective texture register in the register combiners.

So if a texture shader generates a texture unit RGBA result for texture unit 2, use GL_TEXTURE2_ARB for the name of the register value in register combiners.

Should the number of shader stages be settable?

RESOLUTION: No, unused stages can be set to GL_NONE.

How do signed RGBA texture components show up in the register combiners texture registers?

RESOLUTION: As signed values. You can use GL_SIGNED_IDENTITY_NV and get to the signed value directly.

How does the texture unit RGBA result of a GL_NONE, GL_CULL_FRAGMENT_NV, DOT_PRODUCT_NV, or GL_DOT_PRODUCT_DEPTH_REPLACE_NV texture shader operation show up in the register combiners texture registers?

RESOLUTION: Always as the value (0,0,0,0).

How the texture RGBA result of the GL_NONE, GL_CULL_FRAGMENT_NV,

GL_DOT_PRODUCT_NV, and GL_DOT_PRODUCT_DEPTH_REPLACE_NV texture shader operations shows up in the texture environment is not an issue, because the texture environment operation is always assumed to be GL_NONE when the corresponding texture shader is one of GL_NONE, GL_CULL_FRAGMENT_NV, GL_DOT_PRODUCT_NV, or GL_DOT_PRODUCT_DEPTH_REPLACE_NV when GL_TEXTURE_SHADER_NV is enabled.

Why introduce new pixel groups (the HILO and texture offset groups)?

RESOLUTION: In core OpenGL, texture image data is transferred and stored as sets of color components. Such color data can always be promoted to RGBA data.

In addition to color components, there are other types of image data in OpenGL including depth components, stencil components, and color indices. Depth and stencil components can be used by glReadPixels, glDrawPixels, and glCopyPixels, but are not useful for storing texture data in core OpenGL. The EXT_paletted_texture and EXT_index_texture extensions extend the contents of textures to include indices (even though in the case of EXT_paletted_texture, texel fetches are always eventually expanded into color components by the texture palette).

However this these existing pixel groups are not sufficient for all the texture shader operations introduced by this extension. Certain texture shader operations require texture data that is not merely a set of color components. The dot product (GL_DOT_PRODUCT_NV, etc) operations both can utilize high-precision hi and lo components. The offset texture operations (GL_OFFSET_TEXTURE_2D_NV, GL_OFFSET_TEXTURE_2D_SCALE_NV, GL_OFFSET_TEXTURE_RECTANGLE_NV, and GL_OFFSET_TEXTURE_RECTANGLE_SCALE_NV) require textures containing signed offsets used to displace texture coordinates. The GL_OFFSET_TEXTURE_2D_SCALE_NV and GL_OFFSET_TEXTURE_RECTANGLE_SCALE_NV also require an unsigned magnitude for the scaling operation.

To facilitate these new texture representations, this extension introduces several new (external) formats, pixel groups, and internal texture formats. An (external) format is the external representation used by an application to specify pixel data for use by OpenGL. A pixel group is a grouping of components that are transformed by OpenGL's pixel transfer mechanism is a particular manner. For example, RGBA components for colors are transformed differently than stencil components when passed through OpenGL's pixel transfer mechanism. An internal texture format is the representation of texture data within OpenGL. Note that the (external) format used to specify the data by the application may be different than the internal texture format used to store the texture data internally to OpenGL. For example, core OpenGL permits an application to specify data for a texture as GL_LUMINANCE_ALPHA data stored in GLfloats even though the data is to be store in a GL_RGBA8 texture. OpenGL's pixel unpacking and pixel transfer operations perform an appropriate transformation of the data when such a texture download is performed. Also note that data from one pixel group (say stencil components) cannot be supplied as

data for a different pixel group (say RGBA components).

This extension introduces four new (external) formats for texture data: GL_HILO_NV, GL_DSDT_NV, GL_DSDT_MAG_NV, and GL_DSDT_MAG_VIB_NV.

GL_HILO_NV is for specifying high-precision hi and lo components. The other three formats are used to specify texture offset groups. These new formats can only be used for specifying textures (not copying, reading, or writing pixels).

Each of these four pixel formats belong to one of two pixel groups. Pixels specified with the GL_HILO_NV format are transformed as HILO components. Pixels specified with the DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV formats are transformed as texture offset groups.

The HILO component and texture offset group pixel groups have independent scale and bias operations for each component type. Various pixel transfer operations that are performed on the RGBA components pixel group are NOT performed on these two new pixel groups. OpenGL's pixel map, color table, convolution, color matrix, histogram, and min/max are NOT performed on the HILO components or texture offset group pixel groups.

There are four internal texture formats for texture data specified as HILO components: GL_HILO_NV, GL_HILO16_NV, GL_SIGNED_HILO_NV, and GL_SIGNED_HILO16_NV. The HILO data can be stored as either unsigned [0,1] value or [-1,1] signed values. There are also enumerants for both explicitly sized component precision (16-bit components) and unsized component precision. OpenGL implementations are expected to keep HILO components are high precision even if an unsized internal texture format is used.

The expectation with HILO textures is that applications will specify HILO data using a type of GL_UNSIGNED_SHORT or GL_SHORT or larger data types. Specifying HILO data with GL_UNSIGNED_BYTE or GL_BYTE works but does not exploit the full available precision of the HILO internal texture formats.

There are six internal texture formats for texture data specified as texture offset groups: GL_DSDT_NV, GL_DSDT8_NV, GL_DSDT_MAG_NV, GL_DSDT8_MAG8_NV, GL_DSDT_MAG_INTENSITY_NV and GL_DSDT8_MAG8_INTENSITY8_NV. The GL_DSDT_NV formats specify two signed [-1,1] components, ds and dt, used to offset s and t texture coordinates. The GL_DSDT_MAG_NV formats specify an additional third unsigned [0,1] component that is a magnitude to scale an unsigned RGBA texture fetch by. The GL_DSDT_MAG_INTENSITY_NV formats specify an additional fourth [0,1] unsigned component, intensity, that becomes the intensity of the fetched texture for use in the texture environment or register combiners. There are also enumerants for both explicitly sized (8-bit components) and unsized component precision.

Note that the vibrance (VIB) component of the GL_DSDT_MAG_VIB_NV format becomes the intensity component of the GL_DSDT_MAG_INTENSITY_NV internal texture format. Vibrance becomes intensity in the GL_DSDT_MAG_INTENSITY_NV texture format.

The introduction of vibrance is because core OpenGL has no notion of an intensity component in the pixel transfer mechanism or as an external format (instead the red component of an RGBA value becomes the intensity component of intensity textures).

How does the texture unit RGBA result of a texture shader that fetches a texture with a base internal format of GL_HILO_NV, GL_DSDT_NV, or GL_DSDT_MAG_NV show up in the register combiners texture registers?

RESOLUTION: Always as the value (0,0,0,0).

How the texture RGBA result of a texture shader that fetches a texture with a base internal format of GL_HILO_NV, GL_DSDT_NV, or GL_DSDT_MAG_NV the GL_DOT_PRODUCT_NV texture shader shows up in the texture environment is not an issue, because the texture environment operation is always assumed to be GL_NONE in this case when GL_TEXTURE_SHADER_NV is enabled.

Does the GL_DOT_PRODUCT_DEPTH_REPLACE_NV program replace the eye-distance Z or window-space depth?

RESOLUTION: Window-space depth. And if the window-space depth value is outside of the near and far depth range values, the fragment is rejected.

The GL_CULL_FRAGMENT_NV operation always compares against all four texture coordinates. What if I want only one, two, or three comparisons?

RESOLUTION: To compare against a single value, replicate that value in all the coordinates and set the comparison for all components to be identical. Or you can set uninteresting coordinates to zero and use the GL_GEQUAL comparison which will never cull for the value zero.

What is GL_CULL_FRAGMENT_NV good for?

The GL_CULL_FRAGMENT_NV operation provides a mechanism to implement per-fragment clip planes. If a texture coordinate is assigned a signed distance to a plane, the cull fragment test can discard fragments on the wrong side of the plane. Each texture shader stage provides up to four such clip planes. An eye-space clip plane can be established using the GL_EYE_LINEAR texture coordinate generation mode where the clip plane equation is specified via the GL_EYE_PLANE state.

Clip planes are one application for GL_CULL_FRAGMENT_NV, but other clipping approaches are possible too. For example, by computing and assigning appropriate texture coordinates (perhaps with NV_vertex_program), fragments beyond a certain distance from a point can be culled (assuming that it is acceptable to linearly interpolate a distance between vertices).

The texture border color is supposed to be an RGBA value clamped to the range [0,1]. How does the texture border color work in conjunction with signed RGBA color components, HILO components, and texture offset component groups?

RESOLUTION: The per-texture object GL_TEXTURE_BORDER_COLOR is superceded by a GL_TEXTURE_BORDER_VALUES symbolic token. The texture border values are four floats (not clamped to [0,1] when specified). When a texture border is required for a texture, the components for the border texel are determined by the GL_TEXTURE_BORDER_VALUES state. For color components, the GL_TEXTURE_BORDER_VALUES state is treated as a set of RGBA color components. For HILO components, the first value is treated as hi and the second value is treated as lo. For texture offset components, the ds, dt, mag, and vib values correspond to the first, second, third, and fourth texture border values respectively. The particular texture border components are clamped to the range of the component determined by the texture's internal format. So a signed component is clamped to the [-1,1] range and an unsigned component is clamped to the [0,1] range.

For backward compatibility, the GL_TEXTURE_BORDER_COLOR can still be specified and queried. When specified, the values are clamped to [0,1] and used to update the texture border values. When GL_TEXTURE_BORDER_COLOR is queried, there is no clamping of the returned values.

With signed texture components, does the texture environment function discussion need to be amended?

RESOLUTION: Yes. We do not want texture environment results to exceed the range [-1,1].

The GL_DECAL and GL_BLEND operations perform linear interpolations of various components of the form

```
A * B + (1-A) * C
```

The value of A should not be allowed to be negative otherwise, the value of (1-A) may exceed 1.0. These linear interpolations should be written in the form

```
max(0,A) * B + (1-max(0,A)) * C
```

The GL_ADD operation clamps its result to 1.0, but if negative components are permitted, the result should be clamped to the range [-1,1].

The GL_COMBINE_ARB (and GL_COMBINE_EXT) and GL_COMBINE4_NV operations do explicit clamping of all result to [0,1]. In addition, NV_texture_shader adds requirements to clamp inputs to [0,1] too. This is because the GL_ONE_MINUS_SRC_COLOR and GL_ONE_MINUS_SRC_ALPHA operands should really be computing 1-max(0,C). For completeness, GL_SRC_COLOR and GL_SRC_ALPHA should be computing max(0,C).

With signed texture components, does the color sum discussion need to be amended?

RESOLUTION: Yes. The primary and secondary color should both be clamped to the range [0,1] before they are summed.

The unextended OpenGL 1.2 description of color sum does not require a clamp of the primary and secondary colors to the [0,1] range before they are summed. Before signed texture components, the standard texture environment modes either could not generate results outside the [0,1] range or explicitly clamped their results to this range (as in the case of GL_ADD, GL_COMBINE_EXT, and GL_COMBINE4_NV). Now with signed texture components, negative values can be generated by texture environment functions.

We do not want to clamp the intermediate results of texture environment stages since negative results may be useful in subsequent stages, but clamping should be applied to the primary color immediately before the color sum. For symmetry, clamping of the secondary color is specified as well (though there is currently no way to generate a negative secondary color).

Why vibrance?

Vibrance is the fourth component of the external representation of a texture offset group. During pixel transfer, vibrance is scaled and biased based on the GL_VIBRANCE_SCALE and GL_VIBRANCE_BIAS state. Once transformed, the vibrance component becomes the intensity component for textures with a DSDT_MAG_INTENSITY base internal format. Vibrance is meaningful only when specifying texture images with the DS_DT_MAG_VIB_NV external format (and is not supported when reading, drawing, or copying pixels).

There are lots of reasons that a texture shader stage is inconsistent, and in which case, the stage operates as if the operation is NONE. For debugging sanity, is there a way to determine whether a particular texture shader stage is consistent?

RESOLUTION: Yes. Query the shader consistency of a particular texture unit with:

```
GLint consistent;

glActiveTextureARB(stage_to_check);
glGetTexEnviv(GL_TEXTURE_SHADER_NV, GL_SHADER_CONSISTENT_NV,
  &consistent);
```

consistent is one or zero depending on whether the shader stage is consistent or not.

Should there be signed components with sub 8-bit precision?

RESOLUTION: No.

Should packed pixel formats for texture offset groups be supported?

RESOLUTION: Yes, but they are limited to UNSIGNED_INT_S8_S8_8_8_NV and UNSIGNED_INT_8_8_S8_S8_REV_NV for use with the DSDT_MAG_VIB_NV format.

Note that these two new packed pixel formats are only for the DSDT_MAG_VIB_NV and cannot be used with RGBA or BGRA formats. Likewise, the RGBA and BGRA formats cannot be used with the new

UNSIGNED_INT_S8_S8_8_8_NV and UNSIGNED_INT_8_8_S8_S8_REV_NV types.

What should be said about signed fixed-point precision and range of actual implementations?

RESOLUTION: The core OpenGL specification typically specifies fixed-point numerical computations without regard to the specific precision of the computations. This practice is intentional because it permits implementations to vary in the degree of precision used for internal OpenGL computations. When mapping unsigned fixed-point values to a [0,1] range, the mapping is straightforward.

However, this extension supports signed texture components in the range [-1,1]. This presents some awkward choices for how to map [-1,1] to a fixed-point representation. Assuming a binary fixed-point representation with an even distribution of precision, there is no way to exactly represent -1, 0, and 1 and avoid representing values outside the [-1,1] range.

This is not a unique issue for this extension. In core OpenGL, table 2.6 describes mappings from unsigned integer types (GLbyte, GLshort, and GLint) that preclude the exact specification of 0.0. NV_register_combiners supports signed fixed-point values that have similar representation issues.

NVIDIA's solution to this representation problem is to use 8-, 9-, and 16-bit fixed-point representations for signed values in the [-1,1] range such that

floating-point	8-bit fixed-point	9-bit fixed-point	16 bit fixed-point
1.0	n/a	255	n/a
0.99996...	n/a	n/a	32767
0.99218...	127	n/a	n/a
0.0	0	0	0
-1.0	-128	-255	-32768
-1.00392...	n/a	-256	n/a

The 8-bit and 16-bit signed fixed-point types are used for signed internal texture formats, while the 9-bit signed fixed-point type is used for register combiners computations.

The 9-bit signed fixed-point type has the disadvantage that a number slightly more negative than -1 can be represented and this particular value is different dependent on the number of bits of fixed-point precision. The advantage of this approach is that 1, 0, and -1 can all be represented exactly.

The 8-bit and 16-bit signed fixed-point types have the disadvantage that 1.0 cannot be exactly represented (though -1.0 and zero can be exactly represented).

The specification however is written using the conventional OpenGL practice (table 2.6) of mapping signed values evenly over the range [-1,1] so that zero cannot be precisely represented. This is done to keep this specification consistent with OpenGL's existing conventions and to avoid the ugliness of specifying

a precision-dependent range. We expect leeway in how signed fixed-point values are represented.

The spirit of this extension is that an implicit allowance is made for signed fixed-point representations that cannot exactly represent 1.0.

How should NV_texture_rectangle interact with NV_texture_shader?

NV_texture_rectangle introduces a new texture target similar to GL_TEXTURE_2D but that supports non-power-of-two texture dimensions and several usage restrictions (no mipmapping, etc). Also the imaged texture coordinate range for rectangular textures is [0,width]x[0,height] rather than [0,1]x[0,1].

Four texture shader operations will operate like their 2D texture counter-parts, but will access the rectangular texture target rather than the 2D texture target. These are:

GL_TEXTURE_RECTANGLE_NV
GL_OFFSET_TEXTURE_RECTANGLE_NV
GL_OFFSET_TEXTURE_RECTANGLE_SCALE_NV
GL_DOT_PRODUCT_TEXTURE_RECTANGLE_NV

A few 2D texture shader operations, namely GL_DEPENDENT_AR_TEXTURE_2D_NV and GL_DEPENDENT_GB_TEXTURE_2D_NV, do not support rectangular textures because turning colors in the [0,1] range into texture coordinates would only access a single corner texel in a rectangular texture. The offset and dot product rectangular texture shader operations support scaling of the dependent texture coordinates so these operations can access the entire image of a rectangular texture. Note however that it is the responsibility of the application to perform the proper scaling.

Note that the 2D and rectangular "offset texture" shaders both use the same matrix, scale, and bias state.

New Procedures and Functions

None.

New Tokens

Accepted by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of TexEnvf, TexEnvfv, TexEnvi, TexEnviv, GetTexEnvfv, and GetTexEnviv:

TEXTURE_SHADER_NV	0x86DE

When the <target> parameter of TexEnvf, TexEnvfv, TexEnvi, TexEnviv, GetTexEnvfv, and GetTexEnviv is TEXTURE_SHADER_NV, then the value of <pname> may be:

RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV	0x86D9
SHADER_OPERATION_NV	0x86DF
CULL_MODES_NV	0x86E0
OFFSET_TEXTURE_MATRIX_NV	0x86E1
OFFSET_TEXTURE_SCALE_NV	0x86E2
OFFSET_TEXTURE_BIAS_NV	0x86E3
OFFSET_TEXTURE_2D_MATRIX_NV	deprecated alias for OFFSET_TEXTURE_MATRIX_NV
OFFSET_TEXTURE_2D_SCALE_NV	alias for OFFSET_TEXTURE_SCALE_NV
OFFSET_TEXTURE_2D_BIAS_NV	deprecated alias for OFFSET_TEXTURE_BIAS_NV
PREVIOUS_TEXTURE_INPUT_NV	0x86E4
CONST_EYE_NV	0x86E5

When the <target> parameter GetTexEnvfv and GetTexEnviv is TEXTURE_SHADER_NV, then the value of <pname> may be:

SHADER_CONSISTENT_NV	0x86DD

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and SHADER_OPERATION_NV respectively, then the value of <param> or the value pointed to by <params> may be:

NONE	
TEXTURE_1D	
TEXTURE_2D	
TEXTURE_RECTANGLE_NV	(see NV_texture_rectangle)
TEXTURE_CUBE_MAP_ARB	(see ARB_texture_cube_map)
PASS_THROUGH_NV	0x86E6
CULL_FRAGMENT_NV	0x86E7
OFFSET_TEXTURE_2D_NV	0x86E8
OFFSET_TEXTURE_2D_SCALE_NV	see above, note aliasing
OFFSET_TEXTURE_RECTANGLE_NV	0x864C
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	0x864D
DEPENDENT_AR_TEXTURE_2D_NV	0x86E9
DEPENDENT_GB_TEXTURE_2D_NV	0x86EA
DOT_PRODUCT_NV	0x86EC
DOT_PRODUCT_DEPTH_REPLACE_NV	0x86ED
DOT_PRODUCT_TEXTURE_2D_NV	0x86EE
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	0x864E
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	0x86F0
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	0x86F1
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	0x86F2
DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV	0x86F3

When the <target> and <pname> parameters of TexEnvfv and TexEnviv are TEXTURE_SHADER_NV and CULL_MODES_NV respectively, then the value of <param> or the value pointed to by <params> may be:

LESS
GEQUAL

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV respectively, then the value of <param> or the value pointed to by <params> may be:

UNSIGNED_IDENTITY_NV	(see NV_register_combiners)
EXPAND_NORMAL_NV	(see NV_register_combiners)

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and PREVIOUS_TEXTURE_INPUT_NV respectively, then the value of <param> or the value pointed to by <params> may be:

TEXTURE0_ARB
TEXTURE1_ARB
TEXTURE2_ARB
TEXTURE3_ARB
TEXTURE4_ARB
TEXTURE5_ARB
TEXTURE6_ARB
TEXTURE7_ARB

Accepted by the <format> parameter of GetTexImage, TexImage1D, TexImage2D, TexSubImage1D, and TexSubImage2D:

HILO_NV	0x86F4
DSDT_NV	0x86F5
DSDT_MAG_NV	0x86F6
DSDT_MAG_VIB_NV	0x86F7

Accepted by the <type> parameter of GetTexImage, TexImage1D, TexImage2D, TexSubImage1D, and TexSubImage2D:

UNSIGNED_INT_S8_S8_8_8_NV	0x86DA
UNSIGNED_INT_8_8_S8_S8_REV_NV	0x86DB

Accepted by the <internalformat> parameter of CopyTexImage1D, CopyTexImage2D, TexImage1D, and TexImage2D:

SIGNED_RGBA_NV	0x86FB
SIGNED_RGBA8_NV	0x86FC
SIGNED_RGB_NV	0x86FE
SIGNED_RGB8_NV	0x86FF
SIGNED_LUMINANCE_NV	0x8701
SIGNED_LUMINANCE8_NV	0x8702
SIGNED_LUMINANCE_ALPHA_NV	0x8703
SIGNED_LUMINANCE8_ALPHA8_NV	0x8704
SIGNED_ALPHA_NV	0x8705
SIGNED_ALPHA8_NV	0x8706
SIGNED_INTENSITY_NV	0x8707
SIGNED_INTENSITY8_NV	0x8708
SIGNED_RGB_UNSIGNED_ALPHA_NV	0x870C
SIGNED_RGB8_UNSIGNED_ALPHA8_NV	0x870D

Accepted by the <internalformat> parameter of TexImage1D and TexImage2D:

HILO_NV	
HILO16_NV	0x86F8
SIGNED_HILO_NV	0x86F9
SIGNED_HILO16_NV	0x86FA
DSDT_NV	
DSDT8_NV	0x8709
DSDT_MAG_NV	
DSDT8_MAG8_NV	0x870A
DSDT_MAG_INTENSITY_NV	0x86DC
DSDT8_MAG8_INTENSITY8_NV	0x870B

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, GetDoublev, PixelTransferf, and PixelTransferi:

HI_SCALE_NV	0x870E
LO_SCALE_NV	0x870F
DS_SCALE_NV	0x8710
DT_SCALE_NV	0x8711
MAGNITUDE_SCALE_NV	0x8712
VIBRANCE_SCALE_NV	0x8713
HI_BIAS_NV	0x8714
LO_BIAS_NV	0x8715
DS_BIAS_NV	0x8716
DT_BIAS_NV	0x8717
MAGNITUDE_BIAS_NV	0x8718
VIBRANCE_BIAS_NV	0x8719

Accepted by the <pname> parameter of TexParameteriv, TexParameterfv, GetTexParameterfv and GetTexParameteriv:

TEXTURE_BORDER_VALUES_NV	0x871A

Accepted by the <pname> parameter of GetTexParameterfv and GetTexParameteriv:

TEXTURE_HI_SIZE_NV	0x871B
TEXTURE_LO_SIZE_NV	0x871C
TEXTURE_DS_SIZE_NV	0x871D
TEXTURE_DT_SIZE_NV	0x871E
TEXTURE_MAG_SIZE_NV	0x871F

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- **Section 3.6 "Pixel Rectangles"**

Add four new rows to table 3.2:

Parameter Name	Type	Initial Value	Valid Range
HI_SCALE_NV	float	1.0	(-Inf,+Inf)
LO_SCALE_NV	float	1.0	(-Inf,+Inf)
DS_SCALE_NV	float	1.0	(-Inf,+Inf)
DT_SCALE_NV	float	1.0	(-Inf,+Inf)
MAGNITUDE_SCALE_NV	float	1.0	(-Inf,+Inf)
VIBRANCE_SCALE_NV	float	1.0	(-Inf,+Inf)
HI_BIAS_NV	float	0.0	(-Inf,+Inf)
LO_BIAS_NV	float	0.0	(-Inf,+Inf)
DS_BIAS_NV	float	0.0	(-Inf,+Inf)
DT_BIAS_NV	float	0.0	(-Inf,+Inf)
MAGNITUDE_BIAS_NV	float	0.0	(-Inf,+Inf)
VIBRANCE_BIAS_NV	float	0.0	(-Inf,+Inf)

-- **Section 3.6.4 "Rasterization of Pixel Rectangles"**

Add before the subsection titled "Unpacking":

"The HILO_NV, DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV formats are described in this section and section 3.6.5 even though these formats are supported only for texture images. Textures with the HILO_NV format are intended for use with certain dot product texture and dependent texture shader operations (see section 3.8.13). Textures with the DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV format are intended for use with certain offset texture 2D texture shader operations (see section 3.8.13).

The error INVALID_ENUM occurs if HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_VIB_NV is used as the format for DrawPixels, ReadPixels, or other commands that specify or query an image with a format and type parameter though the image is not a texture image. The HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_VIB_NV formats are intended for use with the TexImage and TexSubImage commands.

The HILO_NV format consists of two components, hi and lo, in the hi then lo order. The hi and lo components maintain at least 16 bits of storage per component (at least 16 bits of magnitude for unsigned components and at least 15 bits of magnitude for signed components).

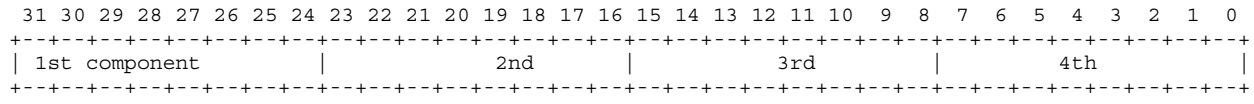
The DSDT_NV format consists of two signed components ds and dt, in the ds then dt order. The DSDT_MAG_NV format consists of three components: the signed ds and dt components and an unsigned magnitude component (mag for short), in the ds, then dt, then mag order. The DSDT_MAG_VIB_NV format consists of four components: the signed ds and dt components, an unsigned magnitude component (mag for short), and an unsigned vibrance component (vib for short), in the ds, then dt, then mag, then vib order."

Add a new row to table 3.8:

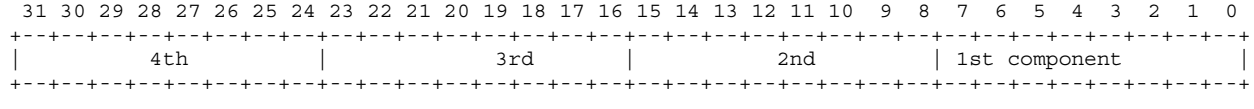
type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_INT_S8_S8_8_8_NV	uint	4	DSDT_MAG_VIB_NV
UNSIGNED_INT_8_8_S8_S8_REV_NV	uint	4	DSDT_MAG_VIB_NV

Add to table 3.11:

UNSIGNED_INT_S8_S8_8_8_NV:



UNSIGNED_INT_8_8_S8_S8_REV_NV:



Replace the fifth paragraph in the subsection titled "Unpacking" with the following:

"Calling DrawPixels with a type of UNSIGNED_BYTE_3_3_2, UNSIGNED_BYTE_2_3_3_REV, UNSIGNED_SHORT_5_6_5, UNSIGNED_SHORT_5_6_5_REV, UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_4_4_4_4_REV, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_SHORT_1_5_5_5_REV, UNSIGNED_INT_8_8_8_8, UNSIGNED_INT_8_8_8_8_REV, UNSIGNED_INT_10_10_10_2, or UNSIGNED_INT_2_10_10_10_REV is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. When packing or unpacking texture images (for example, using TexImage2D or GetTexImage), the type parameter may also be either UNSIGNED_INT_S8_S8_8_8_NV or UNSIGNED_INT_8_8_S8_S8_REV though neither symbolic token is permitted for DrawPixels, ReadPixels, or other commands that specify or query an image with a format and type parameter though the image is not a texture image. The error INVALID_ENUM occurs when UNSIGNED_INT_S8_S8_8_8_NV is used when it is not permitted. When UNSIGNED_INT_S8_S8_8_8_NV or UNSIGNED_INT_8_8_S8_S8_REV_NV is used, the first and second components are treated as signed components. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the format parameter, as listed in table 3.8. The format must also be one of the formats listed in the Matching Pixel Formats column of table 3.8 for the specified packed type. The error INVALID_OPERATION is generated if a mismatch occurs. This constraint also holds for all other functions that accept or return pixel data using type and format parameters to define the type and format of the data."

Amend the second sentence in the sixth paragraph in the subsection titled "Unpacking" to read:

"Each bitfield is interpreted as an unsigned integer value unless

it has been explicitly been stated that the bitfield contains a signed component. Signed bitfields are treated as two's complement numbers."

Add a new row to table 3.12:

Format	First Component	Second Component	Third Component	Fourth Component
DSDT_MAG_VIB_NV	ds	dt	magnitude	vibrance

Change the last sentence in the first paragraph in the subsection titled "Conversion to floating-point" to read:

"For packed pixel types, each unsigned element in the group is converted by computing c / (2^N-1), where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield. In the case of signed elements of a packed pixel type, the signed element is converted by computing 2*c+1 / (2^N-1), where c is the signed integer value of the bitfield containing the element and N is the number of bits in the bitfield."

Change the first sentence in the subsection "Final Expansion to RGBA" to read:

"This step is performed only for groups other than HILO component, depth component, and texture offset groups."

Add the following additional enumeration to the kind of pixel groups in section 3.6.5:

"5. HILO component: Each group comprises two components: hi and lo.

6. Texture offset group: Each group comprises four components: a ds and dt pair, a magnitude, and a vibrance."

Change the subsection "Arithmetic on Components" in section 3.6.5 to read:

"This step applies only to RGBA component, depth component, and HILO component, and texture offset groups. Each component is multiplied by an appropriate signed scale factor: RED_SCALE for an R component, GREEN_SCALE for a G component, BLUE_SCALE for a B component, ALPHA_SCALE, for an A component, HI_SCALE_NV for a HI component, LO_SCALE_NV for a LO component, DS_SCALE_NV for a DS component, DT_SCALE_NV for a DT component, MAGNITUDE_SCALE_NV for a MAG component, VIBRANCE_SCALE_NV for a VIB component, or DEPTH_SCALE for a depth component.

Then the result is added to the appropriate signed bias: RED_BIAS, GREEN_BIAS, BLUE_BIAS, ALPHA_BIAS, HI_BIAS_NV, LO_BIAS_NV, DS_BIAS_NV, DT_BIAS_NV, MAGNITUDE_BIAS_NV, VIBRANCE_BIAS_NV, or DEPTH_BIAS."

-- Section 3.8 "Texturing"

Replace the first paragraph with the following:

"The GL provides two mechanisms for mapping sets of (s,t,r,q) texture coordinates to RGBA colors: conventional texturing and texture shaders.

Conventional texturing maps a portion of a specified image onto each primitive for each enabled texture unit. Conventional texture mapping is accomplished by using the color of an image at the location indicated by a fragment's non-homogeneous (s,t,r) coordinates for a given texture unit.

The alternative to conventional texturing is the texture shaders mechanism. When texture shaders are enabled, each texture unit uses one of twenty-one texture shader operations. Eighteen of the twenty-one shader operations map an (s,t,r,q) texture coordinate set to an RGBA color. Of these, three texture shader operations directly correspond to the 1D, 2D, and cube map conventional texturing operations. Depending on the texture shader operation, the mapping from the (s,t,r,q) texture coordinate set to an RGBA color may depend on the given texture unit's currently bound texture object state and/or the results of previous texture shader operations. The three remaining texture shader operations respectively provide a fragment culling mechanism based on texture coordinates, a means to replace the fragment depth value, and a dot product operation that computes a floating-point value for use by subsequent texture shaders. The specifics of each texture shader operation are described in section 3.8.12.

Texture shading is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant TEXTURE_SHADER_NV. When texture shading is disabled, conventional texturing generates an RGBA color for each enabled textures unit as described in Sections 3.8.10.

After RGBA colors are assigned to each texture unit, either by conventional texturing or texture shaders, the GL proceeds with fragment coloring, either using the texture environment, fog, and color sum operations, or using register combiners extension if supported.

Neither conventional texturing nor texture shaders affects the secondary color."

-- Section 3.8.1 "Texture Image Specification"

Add the following sentence to the first paragraph:

"The formats HILO_NV, DSDT_NV, DSDT_MAG_NV, and DSDT_MAG_VIB_NV are allowed for specifying texture images."

Replace the fourth paragraph with:

"The selected groups are processed exactly as for DrawPixels, stopping just before conversion. Each R, G, B, A, HI, LO, DS, DT,

and MAG value so generated is clamped to [0,1] if the corresponding component is unsigned, or if the corresponding component is signed, is clamped to [-1,1]. The signedness of components depends on the internal format (see table 3.16). The signedness of components for unsized internal formats matches the signedness of components for any respective sized version of the internal format."

Replace table 3.15 with the following table:

Base Internal Format	Component Values	Internal Components	Format Type
ALPHA	A	A	RGBA
LUMINANCE	R	L	RGBA
LUMINANCE_ALPHA	R,A	L,A	RGBA
INTENSITY	R	I	RGBA
RGB	R,G,B	R,G,B	RGBA
RGBA	R,G,B,A	R,G,B,A	RGBA
HILO_NV	HI,LO	HI,LO	HILO
DSDT_NV	DS,DT	DS,DT	texture offset group
DSDT_MAG_NV	DS,DT,MAG	DS,DT,MAG	texture offset group
DSDT_MAG_INTENSITY_NV	DS,DT,MAG,VIB	DS,DT,MAG,I	RGBA/texture offset group

Re-caption table 3.15 as:

"Conversion from RGBA, HILO, and texture offset pixel components to internal texture table, or filter components. See section 3.8.9 for a description of the texture components R, G, B, A, L, and I. See section 3.8.13 for an explanation of the handling of the texture components HI, LO, DS, DT, MAG, and VIB."

Add five more columns to table 3.16 labeled "HI bits", "LO bits", "DS bits", "DT bits", and "MAG bits". Existing table rows should have these column entries blank. Add the following rows to the table:

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	HI bits	LO bits	DS bits	DT bits	MAG bits
HILO16_NV	HILO							16	16			
SIGNED_HILO16_NV	HILO							16*	16*			
SIGNED_RGBA8_NV	RGBA	8*	8*	8*	8*							
SIGNED_RGB8_UNSIGNED_ALPHA8_NV	RGBA	8*	8*	8*	8							
SIGNED_RGB8_NV	RGB	8*	8*	8*								
SIGNED_LUMINANCE8_NV	LUMINANCE					8*						
SIGNED_LUMINANCE8_ALPHA8_NV	LUMINANCE_ALPHA				8*	8*						
SIGNED_ALPHA8_NV	ALPHA				8*							
SIGNED_INTENSITY8_NV	INTENSITY						8*					
DSDT8_NV	DSDT_NV									8*	8*	
DSDT8_MAG8_NV	DSDT_MAG_NV									8*	8*	8
DSDT8_MAG8_INTENSITY8_NV	DSDT_MAG_INTENSITY_NV						8			8*	8*	8

Add to the caption for table 3.16:

"An asterisk (*) following a component size indicates that the corresponding component is signed (the sign bit is included in specified component resolution size)."

Change the first sentences of the fifth paragraph to read:

"Components are then selected from the resulting R, G, B, A, HI, LO, DS, DT, and MAG values to obtain a texture with the base internal format specified by (or derived from) internalformat. Table 3.15 summarizes the mapping of R, G, B, A, HI, LO, DS, DT, and MAG values

to texture components, as a function of the base internal format of the texture image. internalformat may be specified as one of the ten base internal format symbolic constants listed in table 3.15, or as one of the sized internal format symbolic constants listed in table 3.16."

Add these sentences before the last sentence in the fifth paragraph:

"The error INVALID_OPERATION is generated if the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV; or if the internalformat is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, or SIGNED_HILO16_NV and the format is not HILO_NV.

The error INVALID_OPERATION is generated if the format is DSDT_NV and the internalformat is not either DSDT_NV or DSDT8_NV; or if the internal format is either DSDT_NV or DSDT8_NV and the format is not DSDT_NV.

The error INVALID_OPERATION is generated if the format is DSDT_MAG_NV and the internalformat is not either DSDT_MAG_NV or DSDT8_MAG8_NV; or if the internal format is either DSDT_MAG_NV or DSDT8_MAG8_NV and the format is not DSDT_MAG_NV.

The error INVALID_OPERATION is generated if the format is DSDT_MAG_VIB_NV and the internalformat is not either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV; or if the internal format is either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV and the format is not DSDT_MAG_VIB_NV."

Change the first sentence of the sixth paragraph to read:

"The internal component resolution is the number of bits allocated to each value in a texture image (and includes the sign bit if the component is signed)."

Change the third sentence of the sixth paragraph to read:

"If a sized internal format is specified, the mapping of the R, G, B, A, HI, LO, DS, DT, and MAG values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15, and the memory allocations per texture component is assigned by the GL to match the allocations listed in table 3.16 as closely as possible."

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

In the second paragraph (describing CopyTexImage2D), change the third to the last sentence to:

"Parameters level, internalformat, and border are specified using the same values, with the same meanings, as the equivalent arguments of TexImage2D, except that internalformat may not be specified as 1, 2, 3, 4, HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV."

In the third paragraph (describing CopyTexImage1D), change the second to the last sentence to:

"level, internalformat, and border are specified using the same values, with the same meanings, as the equivalent arguments of TexImage1D, except that internalformat may not be specified as 1, 2, 3, 4, HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV."

Insert the following text after the six paragraph reading:

"CopyTexSubImage2D and CopyTexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the pixels are to be copied is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

TexSubImage2D and TexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the texels are to be copied has a different format type (according to table 3.15) than the format type of the texels being specified. Specifically, if the base internal format is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV, then the format parameter must be one of COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA; if the base internal format is HILO_NV, then the format parameter must be HILO_NV; if the base internal format is DSDT_NV, then the format parameter must be DSDT_NV; if the base internal format is DSDT_MAG_NV, then the format parameter must be DSDT_MAG_NV; if the base internal format is DSDT_MAG_INTENSITY_NV, the format parameter must be DSDT_MAG_VIB_NV."

-- **Section 3.8.3 "Texture Parameters"**

Change the TEXTURE_BORDER_COLOR line in table 3.17 to read:

Name	Type	Legal Values
TEXTURE_BORDER_VALUES	4 floats	any value

Add the last two sentences to read:

"The TEXTURE_BORDER_VALUES state can also be specified with the TEXTURE_BORDER_COLOR symbolic constant. When the state is specified via TEXTURE_BORDER_COLOR, each of the four values specified are first clamped to lie in [0,1]. However, if the texture border values state is specified using TEXTURE_BORDER_VALUES, no clamping occurs. In either case, if the values are specified as integers, the conversion for signed integers from table 2.6 is applied to convert the values to floating-point."

-- **Section 3.8.5 "Texture Minification"**

Change the last paragraph to read:

"If any of the selected tauijk, tauij, or taui in the above equations refer to a border texel with i < -bs, j < bs, k < -bs, i >= ws-bs, j

>= hs-bs, or k >= ds-bs, then the border values given by the current setting of TEXTURE_BORDER_VALUES is used instead of the unspecified value or values. If the texture contains color components, the components of the TEXTURE_BORDER_VALUES vector are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. If the texture contains HILO components, the first and second components of the TEXTURE_BORDER_VALUES vector are interpreted as the hi and lo components respectively. If the texture contains texture offset group components, the first, second, third, and fourth components of the TEXTURE_BORDER_VALUES vector are interpreted as ds, dt, mag, and vib components respectively. Additionally, the texture border values are clamped appropriately depending on the signedness of each particular component. Unsigned components are clamped to [0,1]; signed components are clamped to [-1,1]."

-- **Section 3.8.9 "Texture Environment and Texture Functions"**

Augment the list of supported texture functions in the first paragraph to read:

"TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND, ADD, COMBINE_ARB (or COMBINE_EXT), COMBINE4_NV, or NONE;"

Insert this paragraph between the first and second paragraphs:

"When texture shaders are enabled (see section 3.8.13), a given texture unit's texture shader result may be intended for use as an input to another texture shader stage rather than generating a texture unit RGBA result for use in the given texture unit's texture environment function. Additionally, several texture shader operations and texture format types are intended only to generate texture shader results for subsequent texture shaders or perform a side effect (such as culling the fragment or replacing the fragment's depth value) rather than supplying a useful texture unit RGBA result for use in the texture environment function. For this reason, the NONE texture environment ignores the texture unit RGBA result and passes through its input fragment color unchanged."

Change the third sentence of the second paragraph to read:

"If the TEXTURE_SHADER_NV mode is disabled, the precise form of the texture environment function depends on the base internal format of the texture object bound to the given texture unit's highest-precedence enabled texture target. Otherwise if the TEXTURE_SHADER_NV mode is enabled, then the form of the function depends on the texture unit's texture shader operation.

If a texture shader operation requires fetching a filtered texture color value (though not a HILO or texture offset value; see the subsequent HILO and texture offset discussion), the texture environment function depends on the base internal format of the texture shader operation's respective texture target used for fetching by the texture shader operation.

The PASS_THROUGH_NV texture shader operation does not fetch from any texture target, but it generates an RGBA color and therefore always

operates as if the base internal format is RGBA for determining what texture environment function to apply.

If the TEXTURE_SHADER_NV mode is enabled and the texture shader operation for a given texture unit is one of NONE, CULL_FRAGMENT_NV, DOT_PRODUCT_NV, or DOT_PRODUCT_DEPTH_REPLACE_NV, then the given texture unit's texture function always operates as if the texture function is NONE.

If the base internal format of the texture is HILO_NV, DSDT_NV, or DSDT_MAG_NV (independent of whether or not the TEXTURE_SHADER_NV mode is enabled or disabled), then corresponding the texture function always operates as if the texture function is NONE.

If the base internal format of the texture is DSDT_MAG_INTENSITY_NV (independent of whether or not the TEXTURE_SHADER_NV mode is enabled or disabled), then the corresponding texture function operates as if the base internal format is INTENSITY for the purposes of determining the appropriate function using the vibrance component as the intensity value."

Change the phrase in the fourth sentence of the second paragraph describing how Rt, Gt, Bt, At, Lt, and It are assigned to:

"when TEXTURE_SHADER_NV is disabled, Rt, Gt, Bt, At, Lt, and It are the filtered texture values; when TEXTURE_SHADER_NV is enabled, Rt, Gt, Bt, and At are the respective components of the texture unit RGBA result of the texture unit's texture shader stage, and Lt and It are any red, green, or blue component of the texture unit RGBA result (the three components should be the same);"

Change the second to last sentence of the second paragraph to read:

"The initial primary color and texture environment color component values are in the range [0,1]. The filtered texture color and texture function result color component values are in the range [-1,1]. Negative filtered texture color component values are generated by texture internal formats with signed components such as SIGNED_RGBA."

Also amend tables 3.18 and 3.19 based on the following updated columns:

Base Internal Format	DECAL Texture Function	BLEND Texture Function	ADD Texture Function
ALPHA	Rv = Rf (no longer undefined) Gv = Gf Bv = Bf Av = Af	Rv = Rf Gv = Gf Bv = Bf Av = Af*At	Rv = Rf Gv = Gf Bv = Rf Av = Af*Av = At
LUMINANCE (or 1)	Rv = Rf (no longer undefined) Gv = Gf Bv = Bf Av = Af	Rv = Rf*(1-max(0,Lt)) + Rc*max(0,Lt) Gv = Gf*(1-max(0,Lt)) + Gc*max(0,Lt) Bv = Bf*(1-max(0,Lt)) + Bc*max(0,Lt) Av = Af	Rv = max(-1,min(1,Rf+Lt)) Gv = max(-1,min(1,Gf+Lt)) Bv = max(-1,min(1,Bf+Lt)) Av = Af
LUMINANCE_ALPHA (or 2)	Rv = Rf (no longer undefined) Gv = Gf Bv = Bf Av = Af	Rv = Rf*(1-max(0,Lt)) + Rc*max(0,Lt) Gv = Gf*(1-max(0,Lt)) + Gc*max(0,Lt) Bv = Bf*(1-max(0,Lt)) + Bc*max(0,Lt) Av = Af*At	Rv = max(-1,min(1,Rf+Lt)) Gv = max(-1,min(1,Gf+Lt)) Bv = max(-1,min(1,Bf+Lt)) Av = Af*At
INTENSITY	Rv = Rf (no longer undefined) Gv = Gf Bv = Bf Av = Af	Rv = Rf*(1-max(0,It)) + Rc*max(0,It) Gv = Gf*(1-max(0,It)) + Gc*max(0,It) Bv = Bf*(1-max(0,It)) + Bc*max(0,It) Av = Af*(1-max(0,It)) + Ac*max(0,It)	Rv = max(-1,min(1,Rf+It)) Gv = max(-1,min(1,Gf+It)) Bv = max(-1,min(1,Bf+It)) Av = max(-1,min(1,Af+It))
RGB (or 3)	Rv = Rt Gv = Gt Bv = Bt Av = Af	Rv = Rf*(1-max(0,Rt)) + Rc*max(0,Rt) Gv = Gf*(1-max(0,Gt)) + Gc*max(0,Gt) Bv = Bf*(1-max(0,Bt)) + Bc*max(0,Bt) Av = Af	Rv = max(-1,min(1,Rf+Rt)) Gv = max(-1,min(1,Gf+Gt)) Bv = max(-1,min(1,Bf+Bt)) Av = Af
RGBA (or 4)	Rv = Rf*(1-max(0,At)) + Rt*max(0,At) Gv = Gf*(1-max(0,At)) + Gt*max(0,At) Bv = Bf*(1-max(0,At)) + Bt*max(0,At) Av = Af	Rv = Rf*(1-max(0,Rt)) + Rc*max(0,Rt) Gv = Gf*(1-max(0,Gt)) + Gc*max(0,Gt) Bv = Bf*(1-max(0,Bt)) + Bc*max(0,Bt) Av = Af*At	Rv = max(-1,min(1,Rf+Rt)) Gv = max(-1,min(1,Gf+Gt)) Bv = max(-1,min(1,Bf+Bt)) Av = Af*At

Also augment table 3.18 or 3.19 with the following column:

Base Internal Format	NONE Texture Function
ALPHA	Rv = Rf Gv = Gf Bv = Bf Av = Af
LUMINANCE (or 1)	Rv = Rf Gv = Gf Bv = Bf Av = Af
LUMINANCE_ALPHA (or 2)	Rv = Rf Gv = Gf Bv = Bf Av = Af
INTENSITY	Rv = Rf Gv = Gf Bv = Bf Av = Af
RGB (or 3)	Rv = Rf Gv = Gf Bv = Bf Av = Af
RGBA (or 4)	Rv = Rf Gv = Gf Bv = Bf Av = Af

Amend tables 3.21 and 3.22 in the ARB_texture_env_combine specification (or EXT_texture_env_combine specification) to require inputs to be clamped positive (the TEXTURE<n>_ARB entries apply only if NV_texture_env_combine4 is supported):

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
TEXTURE	SRC_COLOR	max(0,Ct)
	ONE_MINUS_SRC_COLOR	(1-max(0,Ct))
	SRC_ALPHA	max(0,At)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At))
CONSTANT_EXT	SRC_COLOR	max(0,Cc
	ONE_MINUS_SRC_COLOR	(1-max(0,Cc)
	SRC_ALPHA	max(0,Ac
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	max(0,Cf
	ONE_MINUS_SRC_COLOR	(1-max(0,Cf)
	SRC_ALPHA	max(0,Af
	ONE_MINUS_SRC_ALPHA	(1-max(0,Af)
PREVIOUS_EXT	SRC_COLOR	max(0,Cp
	ONE_MINUS_SRC_COLOR	(1-max(0,Cp)
	SRC_ALPHA	max(0,Ap
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ap)
TEXTURE<n>_ARB	SRC_COLOR	max(0,Ct<n>)
	ONE_MINUS_SRC_COLOR	(1-max(0,Ct<n>))
	SRC_ALPHA	max(0,At<n>)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At<n>))

Table 3.21: Arguments for COMBINE_RGB_ARB (or COMBINE_RGB_EXT) functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
TEXTURE	SRC_ALPHA	max(0,At)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At))
CONSTANT_EXT	SRC_ALPHA	max(0,Ac)
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ac))
PRIMARY_COLOR_EXT	SRC_ALPHA	max(0,Af)
	ONE_MINUS_SRC_ALPHA	(1-max(0,Af))
PREVIOUS_EXT	SRC_ALPHA	max(0,Ap)
	ONE_MINUS_SRC_ALPHA	(1-max(0,Ap))
TEXTURE<n>_ARB	SRC_ALPHA	max(0,At<n>)
	ONE_MINUS_SRC_ALPHA	(1-max(0,At<n>))

Table 3.22: Arguments for COMBINE_ALPHA_ARB (or COMBINE_ALPHA_EXT) functions

-- **Section 3.9 "Color Sum"**

Update the first paragraph to read:

"At the beginning of color sum, a fragment has two RGBA colors: a primary color cpri (which texturing, if enabled, may have modified) and a secondary color csec. The components of these two colors are clamped to [0,1] and then summed to produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]."

-- **NEW Section 3.8.13 "Texture Shaders"**

"Each texture unit is configured with one of twenty-one texture shader operations. Several texture shader operations require additional state. All per-texture shader stage state is specified using the TexEnv commands with the target specified as TEXTURE_SHADER_NV. The per-texture shader state is replicated per texture unit so the texture unit selected by ActiveTextureARB determines which texture unit's environment is modified by TexEnv calls.

When calling TexEnv with a target of TEXTURE_SHADER_NV, pname must be one of SHADER_OPERATION_NV, CULL_MODES_NV, OFFSET_TEXTURE_MATRIX_NV, OFFSET_TEXTURE_SCALE_NV, OFFSET_TEXTURE_BIAS_NV, PREVIOUS_TEXTURE_INPUT_NV, or CONST_EYE_NV.

When TexEnv is called with the target of TEXTURE_SHADER_NV, SHADER_OPERATION_NV may be set to one of NONE, TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP_ARB, PASS_THROUGH_NV, CULL_FRAGMENT_NV, OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. The semantics of each of these shader operations is described in section 3.8.13.1. Not every operation is supported in every texture unit. The restrictions for how these shader operations can be configured in various texture units are described in section 3.8.13.2.

When TexEnv is called with the target of TEXTURE_SHADER_NV, CULL_MODES_NV is set to a vector of four cull comparisons by providing four symbolic tokens, each being either LESS or GEQUAL. These cull modes are used by the CULL_FRAGMENT_NV operation (see section 3.8.13.1.7).

When TexEnv is called with the target of TEXTURE_SHADER_NV, RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV may be set to either UNSIGNED_IDENTITY_NV or EXPAND_NORMAL_NV. This RGBA unsigned dot product mapping mode is used by the DOT_PRODUCT_NV operation (see section 3.8.13.1.14) and other operations that compute dot products.

When TexEnv is called with the target of TEXTURE_SHADER_NV, PREVIOUS_TEXTURE_INPUT_NV may be set to TEXTUREi_ARB where i is between 0 and n-1 where n is the implementation-dependent number of texture units supported. The INVALID_OPERATION error is generated if i is greater than or equal to the current active texture unit.

When TexEnv is called with the target of TEXTURE_SHADER_NV, OFFSET_TEXTURE_MATRIX_NV may be set to a 2x2 matrix of floating-point values stored in column-major order as 4 consecutive floating-point values, i.e. as:

```
[ a1 a3 ]
[ a2 a4 ]
```

This matrix is used by the OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, and OFFSET_TEXTURE_RECTANGLE_SCALE_NV operations (see sections 3.8.13.1.8 through 3.8.13.1.11).

When TexEnv is called with the target of TEXTURE_SHADER_NV, OFFSET_TEXTURE_SCALE_NV may be set to a floating-point value. When TexEnv is called with the target of TEXTURE_SHADER_NV, OFFSET_TEXTURE_BIAS_NV may be set to a floating-point value. These scale and bias values are used by the OFFSET_TEXTURE_2D_SCALE_NV and OFFSET_TEXTURE_RECTANGLE_SCALE_NV operations (see section 3.8.13.1.9 and 3.8.13.1.11).

When TexEnv is called with the target of TEXTURE_SHADER_NV, CONST_EYE_NV is set to a vector of three floating-point values used as the constant eye vector in the DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV texture shader (see section 3.8.13.1.19).

3.8.13.1 Texture Shader Operations

The texture enables described in section 3.8.10 only affect conventional texturing mode; these enables are ignored when TEXTURE_SHADER_NV is enabled. Instead, the texture shader operation determines how texture coordinates are mapped to filtered texture values.

Tables 3.A, 3.B, 3.C, and 3.D specify inter-stage dependencies, texture target dependencies, relevant inputs, and result types and values respectively for each texture shader operation. Table 3.E specifies how the components of an accessed texture are mapped to the components of the texture unit RGBA result based on the base internal format of the accessed texture. The following discussion describes each possible texture shader operation in detail.

texture shader operation i	previous texture input	texture shader operation i-1	operation i-2	texture shader operation i+1
NONE	-	-	-	-
TEXTURE_1D	-	-	-	-
TEXTURE_2D	-	-	-	-
TEXTURE_RECTANGLE_NV	-	-	-	-
TEXTURE_CUBE_MAP_ARB	-	-	-	-
PASS_THROUGH_NV	-	-	-	-
CULL_FRAGMENT_NV	-	-	-	-
OFFSET_TEXTURE_2D_NV	base internal texture format must be one of DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_TEXTURE_2D_SCALE_NV	base internal texture format must be either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_TEXTURE_RECTANGLE_NV	base internal texture format must be one of DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_INTENSITY_NV	-	-	-
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	base internal texture format must be either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV	-	-	-
DEPENDENT_AR_TEXTURE_2D_NV	shader result type must all be unsigned RGBA	-	-	-
DEPENDENT_GB_TEXTURE_2D_NV	shader result type must all be unsigned RGBA	-	-	-
DOT_PRODUCT_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	-	-	-
DOT_PRODUCT_TEXTURE_2D_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	-
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	-
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	shader operation must be DOT_PRODUCT_NV	-
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA; previous texture input must not be unit i-1	shader operation must be DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	shader operation must be DOT_PRODUCT_NV	-
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	shader operation must be DOT_PRODUCT_NV	-
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	shader operation must be DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV
DOT_PRODUCT_DEPTH_REPLACE_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, or all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	-	-

Table 3.A: Texture shader inter-stage dependencies for each operation. If any one of the dependencies listed above is not met, the texture shader stage is considered inconsistent. Further texture shader target dependencies are listed in table X.Y. Additionally, if any one of the texture shader stages that a particular texture shader stage depends on is inconsistent, then the dependent texture shader stage is also considered inconsistent. When a texture shader stage is considered inconsistent, the inconsistent stage operates as if the stage's operation is NONE.

texture shader operation i	texture unit i
NONE	-
TEXTURE_1D	1D target must be consistent
TEXTURE_2D	2D target must be consistent
TEXTURE_RECTANGLE_NV	rectangle target must be consistent
TEXTURE_CUBE_MAP_ARB	cube map target must be consistent
PASS_THROUGH_NV	-
CULL_FRAGMENT_NV	-
OFFSET_TEXTURE_2D_NV	2D target must be consistent
OFFSET_TEXTURE_2D_SCALE_NV	2D target must be consistent and 2D texture target type must be unsigned RGBA
OFFSET_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	rectangle target must be consistent and rectangle texture target type must be unsigned RGBA
DEPENDENT_AR_TEXTURE_2D_NV	2D target must be consistent
DEPENDENT_GB_TEXTURE_2D_NV	2D target must be consistent
DOT_PRODUCT_NV	-
DOT_PRODUCT_TEXTURE_2D_NV	2D target must be consistent
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	rectangle target must be consistent
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	cube map target must be consistent
DOT_PRODUCT_DEPTH_REPLACE_NV	-

Table 3.B: Texture shader target dependencies for each operation. If the dependency listed above is not met, the texture shader stage is considered inconsistent.

texture shader operation i	texture coordinate set usage	texture target	uses stage result i-1	uses stage result i-2	uses stage result i+1	uses previous texture input	uses cull modes	uses offset texture 2D matrix	offset texture 2D scale and bias	uses const eye vector
NONE	-	-	-	-	-	-	-	-	-	-
TEXTURE_1D	s,q	1D	-	-	-	-	-	-	-	-
TEXTURE_2D	s,t,q	2D	-	-	-	-	-	-	-	-
TEXTURE_RECTANGLE_NV	s,t,q	rectangle	-	-	-	-	-	-	-	-
TEXTURE_CUBE_MAP_ARB	s,t,r	cube map	-	-	-	-	-	-	-	-
PASS_THROUGH_NV	s,t,r,q	-	-	-	-	-	-	-	-	-
CULL_FRAGMENT_NV	s,t,r,q	-	-	-	-	-	y	-	-	-
OFFSET_TEXTURE_2D_NV	s,t	2D	-	-	-	-	-	y	-	-
OFFSET_TEXTURE_2D_SCALE_NV	s,t	2D	-	-	-	-	-	y	y	-
OFFSET_TEXTURE_RECTANGLE_NV	s,t	rectangle	-	-	-	-	-	y	-	-
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	s,t	rectangle	-	-	-	-	-	y	y	-
DEPENDENT_AR_TEXTURE_2D_NV	-	2D	-	-	-	y	-	-	-	-
DEPENDENT_GB_TEXTURE_2D_NV	-	2D	-	-	-	y	-	-	-	-
DOT_PRODUCT_NV	s,t,r (q*)	-	-	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_2D_NV	s,t,r	2D	y	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	s,t,r	rectangle	y	-	-	y	-	-	-	-
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	s,t,r	cube map	y	y	-	y	-	-	-	-
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	s,t,r,q	cube map	y	y	-	y	-	-	-	-
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	s,t,r	cube map	y	y	-	y	-	-	-	y
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	s,t,r (q*)	cube map	y	y	y	y	-	-	-	-
DOT_PRODUCT_DEPTH_REPLACE_NV	s,t,r	-	y	-	-	y	-	-	-	-

Table 3.C: Relevant texture shader computation inputs for each operation. The (q*) for the texture coordinate set usage indicates that the q texture coordinate is used only when the DOT_PRODUCT_NV and DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV operations are used in conjunction with DOT_PRODUCT_REFLECT_CUBE_MAP_NV.

texture shader operation i	shader stage result type	shader stage result	texture unit RGBA color result
NONE	RGBA	invalid	(0,0,0,0)
TEXTURE_1D	matches 1D target type	filtered 1D target texel	if 1D target texture type is RGBA, filtered 1D target texel, else (0,0,0,0)
TEXTURE_2D	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
TEXTURE_CUBE_MAP_ARB	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
PASS_THROUGH_NV	RGBA	(max(0,min(1,s)), max(0,min(1,t)), max(0,min(1,r)), max(0,min(1,q)))	(max(0,min(1,s)), max(0,min(1,t)), max(0,min(1,r)), max(0,min(1,q)))
CULL_FRAGMENT_NV	RGBA	invalid	(0,0,0,0)
OFFSET_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
OFFSET_TEXTURE_2D_SCALE_NV	RGBA	filtered 2D target texel	scaled filtered 2D target texel
OFFSET_TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
OFFSET_TEXTURE_RECTANGLE_SCALE_NV	RGBA	filtered rectangle target texel	scaled filtered rectangle target texel
DEPENDENT_AR_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DEPENDENT_GB_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DOT_PRODUCT_NV	float	dot product	(0,0,0,0)
DOT_PRODUCT_TEXTURE_2D_NV	matches 2D target type	filtered 2D target texel	if 2D target texture type is RGBA, filtered 2D target texel, else (0,0,0,0)
DOT_PRODUCT_TEXTURE_RECTANGLE_NV	matches rectangle target type	filtered rectangle target texel	if rectangle target texture type is RGBA, filtered rectangle target texel, else (0,0,0,0)
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_REFLECT_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_CONST_EYE_-REFLECT_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV	matches cube map target type	filtered cube map target texel	if cube map target texture type is RGBA, filtered cube map target texel, else (0,0,0,0)
DOT_PRODUCT_DEPTH_REPLACE_NV	RGBA	invalid	(0,0,0,0)

Table 3.D: Texture shader stage results for each operation.

Base internal format	Red	Green	Blue	Alpha
ALPHA	1	1	1	At
LUMINANCE	Lt	Lt	Lt	1
INTENSITY	It	It	It	It
LUMINANCE_ALPHA	Lt	Lt	Lt	At
RGB	Rt	Gt	Bt	1
RGBA	Rt	Gt	Bt	At

Table 3.E: How base internal formats components are mapped to RGBA values for texture shaders (note that the mapping for ALPHA is different from the mapping in Table 3.23 in the EXT_texture_env_combine extension).

3.8.13.1.1 None

The NONE texture shader operation ignores the texture unit's texture coordinate set and always generates the texture unit RGBA result (0,0,0,0) for its filtered texel value. The texture shader result is invalid. This texture shader stage is always consistent.

When a texture unit is not needed while texture shaders are enabled, it is most efficient to set the texture unit's texture shader operation to NONE.

3.8.13.1.2 1D Projective Texturing

The TEXTURE_1D texture shader operation accesses the texture unit's 1D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q) for the 1D texture coordinate where s and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's 1D texture target.

If the texture unit's 1D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.3 2D Projective Texturing

The TEXTURE_2D texture shader operation accesses the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q,t/q) for the 2D texture coordinates where s, t, and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's 2D texture target.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.4 Rectangle Projective Texturing

The TEXTURE_RECTANGLE_NV texture shader operation accesses the texture unit's rectangle texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q,t/q) for the 2D texture coordinates where s, t, and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to NV_texture_rectangle's rectangle texture target.

If the texture unit's rectangle texture object is not consistent,
then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if
it is the NONE operation.

3.8.13.1.5 Cube Map Texturing

The TEXTURE_CUBE_MAP_ARB texture shader operation accesses
the texture unit's cube map texture object (as described in the
ARB_texture_cube_map specification) using (s,t,r) for the 3D texture
coordinate where s, t, and r are the homogeneous texture coordinates
for the texture unit. The result of the texture access becomes
both the shader result and texture unit RGBA result (see table
3.E). The type of the shader result depends on the format type
of the accessed texture. This mode is equivalent to conventional
texturing's cube map texture target.

If the texture unit's cube map texture object is not consistent, then
this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if
it is the NONE operation.

3.8.13.1.6 Pass Through

The PASS_THROUGH_NV texture shader operation converts an (s,t,r,q)
texture coordinate set in an RGBA color result (r,g,b,a).
Each texture coordinate is first clamped to [0,1] before being
mapped to its corresponding color component. The texture shader
result and texture unit RGBA result of this operation are both
assigned the clamped RGBA color result.

This operation in no way depends on any of the texture unit's
texture objects.

3.8.13.1.7 Cull Fragment

The CULL_FRAGMENT_NV texture shader operation compares each
component of the texture coordinate set (s,t,r,q) to zero based
on the texture shader's corresponding cull mode. For the LESS
cull mode to succeed, the corresponding component must be less
than zero; otherwise the comparison fails. For the GEQUAL cull
mode to succeed, the corresponding component must be greater or
equal to zero; otherwise the comparison fails. If any of the four
comparisons fails, the fragment is discarded.

The texture unit RGBA result generated is always (0,0,0,0).
The texture shader result is invalid. This texture shader stage
is always consistent.

This operation in no way depends on any of the texture unit's
texture objects.

3.8.13.1.8 Offset Texture 2D

The OFFSET_TEXTURE_2D_NV texture shader operation uses the transformed result of a previous texture shader stage to perturb the current texture shader stage's (s,t) texture coordinates (without a projective division by q). The resulting perturbed texture coordinates (s',t') are used to access the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6).

The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

The perturbed texture coordinates s' and t' are computed with floating-point math as follows:

```
s' = s + a1 * DSprev + a3 * DTprev
t' = t + a2 * DSprev + a4 * DTprev
```

where a1, a2, a3, and a4 are the texture shader stage's OFFSET_TEXTURE_MATRIX_NV values, and DSprev and DTprev are the (signed) DS and DT components of a previous texture shader unit's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a base internalformat that is not one of DSDT_NV, DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.9 Offset Texture 2D and Scale

The OFFSET_TEXTURE_2D_SCALE_NV texture shader operation extends the functionality of the OFFSET_TEXTURE_2D_NV texture shader operation. The texture unit's 2D texture object is accessed by the same perturbed s' and t' coordinates used by the OFFSET_TEXTURE_2D_NV operation. The red, green, and blue components (but not alpha) of the RGBA result of the texture access are further scaled by

the value Scale and clamped to the range [0,1]. This RGBA result is this shader's texture unit RGBA result. This shader's texture shader result is the RGBA result of the texture access prior to scaling and clamping.

Scale is computed with floating-point math as follows:

```
  Scale = textureOffsetBias + textureOffsetScale * MAGprev
```

where textureOffsetBias is the texture shader stage's OFFSET_TEXTURE_BIAS_NV value, textureOffsetScale is the texture shader stage's OFFSET_TEXTURE_SCALE_NV value, and MAGprev is the magnitude component of the a previous texture shader unit's result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value.

The texture unit RGBA result (red',green',blue',alpha') is computed as follows:

```
  red'   = max(0.0, min(1.0, Scale * red))
  green' = max(0.0, min(1.0, Scale * green))
  blue'  = max(0.0, min(1.0, Scale * blue))
  alpha' = alpha
```

where red, green, blue, and alpha are the texture access components.

If the unit's 2D texture object has any signed components, then this texture shader stage is not consistent.

If the texture unit's 2D texture object is has a format type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a base internalformat that is not either DSDT_MAG_NV or DSDT_MAG_INTENSITY_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.10 Offset Texture Rectangle

The OFFSET_TEXTURE_RECTANGLE_NV shader operation operates identically to the OFFSET_TEXTURE_2D_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.11 Offset Texture Rectangle Scale

The OFFSET_TEXTURE_RECTANGLE_SCALE_NV shader operation operates identically to the OFFSET_TEXTURE_2D_SCALE_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.12 Dependent Alpha-Red Texturing

The DEPENDENT_AR_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (Aprev, Rprev) for the 2D texture coordinates where Aprev and Rprev are the are the alpha and red components of a previous texture input's RGBA texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type of RGBA but any of the RGBA components are signed, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value

is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.13 Dependent Green-Blue Texturing

The DEPENDENT_GB_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in section 3.8.4, 3.8.5, and 3.8.6) using (Gprev, Bprev) for the 2D texture coordinates where Gprev and Bprev are the are the green and blue components of a previous texture input's RGBA texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input's texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a texture shader result type of RGBA but any of the RGBA components are signed, then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.14 Dot Product

The DOT_PRODUCT_NV texture shader operation computes a floating-point texture shader result. The texture shader result is the floating-point dot product of the texture unit's (s,t,r)

texture coordinates and a remapped version of the RGBA or HILO
texture shader result from a specified previous texture shader stage.
The RGBA color result of this shader is always (0,0,0,0).

The re-mapping depends on the specified previous texture shader
stage's texture shader result type. Specifically, the re-mapping
depends on whether this texture shader result type has all signed
components or all unsigned components, and whether it has RGBA
components or HILO components, and, in the case of unsigned RGBA
texture shader results, the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV
state.

If the specified previous texture unit's texture shader result
type is HILO and all the type components are unsigned, then the
floating-point result is computed by

```
result = s * HI + t * LO + r
```

where HI and LO are the (unsigned) hi and lo components respectively
of the previous texture unit's HILO texture shader result.

If the specified previous texture unit's texture shader result
type is HILO and all the type components are signed, then the
floating-point result is computed by

```
result = s * HI + t * LO + r * sqrt(max(0, 1.0 - HI*HI - LO*LO))
```

where HI and LO are the (signed) hi and lo components respectively
of the previous texture unit's texture shader result.

If the specified previous texture unit's texture shader result
contains only signed RGBA components, then the floating-point result
is computed by

```
result = s * Rprev + t * Gprev + r * Bprev
```

where Rprev, Gprev, and Bprev are the (signed) red, green, and blue
components respectively of the previous texture unit's RGBA texture
shader result.

If the specified previous texture unit's texture shader result
contains only unsigned RGBA components, then the dot product
computation depends on the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV
state. When the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV is
UNSIGNED_IDENTITY_NV, then the floating-point result for unsigned
RGBA components is computed by

```
result = s * Rprev + t * Gprev + r * Bprev
```

where Rprev, Gprev, and Bprev are the (unsigned) red, green, and
blue components respectively of the previous texture unit's RGBA
texture shader result.

When the RGBA_UNSIGNED_DOT_PRODUCT_MAPPING_NV is EXPAND_NORMAL_NV, then the floating-point result for unsigned RGBA components is computed by

```
result = s * (2.0*Rprev-1.0) + t * (2.0*Gprev-1.0) + r * (2.0*Bprev-1.0)
```

where Rprev, Gprev, and Bprev are the (unsigned) red, green, and blue components respectively of the previous texture unit's RGBA texture shader result.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the components of the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value have mixed signedness, then this texture shader stage is not consistent. For example, the SIGNED_RGB_UNSIGNED_ALPHA_NV base internal format has mixed signedness.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

This operation in no way depends on any of the texture unit's texture objects.

3.8.13.1.15 Dot Product Texture 2D

The DOT_PRODUCT_TEXTURE_2D_NV texture shader operation accesses the texture unit's 2D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (dotP,dotC) for the 2D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotP is the floating-point dot product result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i-1 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's 2D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.16 Dot Product Texture Rectangle Scale

The DOT_PRODUCT_TEXTURE_RECTANGLE_NV shader operation operates identically to the DOT_PRODUCT_TEXTURE_2D_NV shader operation except that the rectangle texture target is accessed rather than the 2D texture target.

If the texture unit's rectangle texture object (rather than the 2D texture object) is not consistent, then this texture shader stage is not consistent.

3.8.13.1.17 Dot Product Texture Cube Map

The DOT_PRODUCT_TEXTURE_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (dotPP,dotP,dotC) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotPP is the floating-point dot product texture shader result from the i-2 texture shader stage, assuming the i-2 texture shader stage's operation is DOT_PRODUCT_NV. dotP is the floating-point dot product texture shader result from the i-1 texture shader stage,

assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.18 Dot Product Reflect Cube Map

The DOT_PRODUCT_REFLECT_CUBE_MAP_NV and DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV (described in the section 3.8.13.1.20) texture shader operations are typically used together.

The DOT_PRODUCT_REFLECT_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (rx,ry,rz) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Let R = (rx,ry,rz), N = (dotPP,dotP,dotC), and E = (qPP,qP,qC), then

```
R = 2 * (N dot E) / (N dot N) * N - E
```

Assuming that i is the current texture shader stage, dotPP is the floating-point dot product texture shader result from the

i-2 texture shader stage, assuming the i-2 texture shader stage's operation is DOT_PRODUCT_NV. dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is either DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

qPP is the q component of the i-2 texture shader stage's texture coordinate set. qP is the q component of the i-1 texture shader stage's texture coordinate set. qC is the q component of the current texture shader stage's texture coordinate set.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If this texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value refers to texture unit i-2 or i-1, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If the i-2 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage is not consistent, then this texture shader stage is not consistent.

If the texture unit's cube map texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.19 Dot Product Constant Eye Reflect Cube Map

The DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV texture shader operation operates the same as the DOT_PRODUCT_REFLECT_CUBE_MAP_NV operation except that the eye vector E is equal to the three

floating-point values assigned to the texture shader's eye constant (rather than the three q components of the given texture unit and the previous two texture units).

The DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV operation has the same texture shader consistency rules as the DOT_PRODUCT_REFLECT_CUBE_MAP_NV operation.

3.8.13.1.20 Dot Product Diffuse Cube Map

The DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV texture shader operation accesses the texture unit's cube map texture object (as described in the ARB_texture_cube_map specification) using (dotP,dotC,dotN) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14. dotN is the floating-point dot product texture shader result from the i+1 texture shader stage, assuming the next texture shader stage's operation is either DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

If the texture unit's cube map texture object is not consistent, then this operation operates as if it is the NONE operation. If the previous texture unit's texture shader operation is not DOT_PRODUCT_NV, then this operation operates as if it is the NONE operation. If the next texture unit's texture shader operation is neither DOT_PRODUCT_REFLECT_CUBE_MAP_NV nor DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, then this operation operates as if it is the NONE operation. If the next texture unit's texture shader operation is either DOT_PRODUCT_REFLECT_CUBE_MAP_NV or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, but the next texture unit operation is operating as if it is the NONE operation, then this operation operates as if it is the NONE operation. If the specified previous input texture unit is inconsistent or uses the DOT_PRODUCT_NV texture shader operation, then this operation operates as if it is the NONE operation.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by
the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value
is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current
texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not
consistent, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not
DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the i+1 texture shader stage operation
is not DOT_PRODUCT_REFLECT_CUBE_MAP_NV or
DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV, then this texture shader
stage is not consistent.

If either the i-1 or i+1 texture shader stage is not consistent,
then this texture shader stage is not consistent.

If the texture unit's cube map texture object is not consistent,
then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if
it is the NONE operation.

3.8.13.1.21 Dot Product Depth Replace

The DOT_PRODUCT_DEPTH_REPLACE_NV texture shader operation replaces
the incoming fragments depth (in window coordinates, before
conversion to fixed-point, i.e. in the [0,1] range) with a new
depth value. The new depth is computed as follows:

```
depth = dotP / dotC
```

Assuming that i is the current texture shader stage, dotP is the
floating-point dot product texture shader result from the i-1 texture
shader stage, assuming the i-1 texture shader stage's operation
is DOT_PRODUCT_NV. dotC is the floating-point dot product result
from the current texture shader stage. dotC is computed in the
identical manner used to compute the floating-point result of the
DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the new depth value is outside of the range of the near and far
depth range values, the fragment is rejected.

The texture unit RGBA result generated is always (0,0,0,0).
The texture shader result is invalid.

If the previous texture input texture object specified by the
current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has
a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV
base internal format does not count as an RGBA format type in this
context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by
the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value
is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by
the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value
is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current
texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not
consistent, then this texture shader stage is not consistent.

If the i-1 texture shader stage operation is not DOT_PRODUCT_NV,
then this texture shader stage is not consistent.

If the i-1 texture shader stage is not consistent, then
this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if
it is the NONE operation.

This operation in no way depends on any of the texture unit's
texture objects.

3.8.13.2 Texture Shader Restrictions

There are various restrictions on possible texture shader
configurations. These restrictions are described in this section.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV
parameter for texture unit 0 is assigned one of OFFSET_TEXTURE_2D_NV,
OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV,
OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV,
DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV,
DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV,
DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these
texture shaders requires a previous texture shader result that
is not possible for texture unit 0. Therefore these shaders are
disallowed for texture unit 0.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV
parameter for texture unit 1 is assigned one of
DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV,
DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture
shaders requires either two previous texture shader results or
a dot product result that cannot be generated by texture unit 0.
Therefore these shaders are disallowed for texture unit 1.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV
parameter for texture unit 2 is assigned one of
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture
shaders requires three previous texture shader results. Therefore
these shaders are disallowed for texture unit 2.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV parameter for texture unit n-1 (where n is the number of supported texture units) is assigned either DOT_PRODUCT_NV or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV. DOT_PRODUCT_NV is invalid for the final texture shader stage because it is only useful as an input to a successive texture shader stage. DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV is invalid for the final texture shader stage because it must be followed by the DOT_PRODUCT_REFLECT_CUBE_MAP_NV operation in the immediately successive stage. Therefore these shaders are disallowed for texture unit n-1.

3.8.13.3 Required State

The state required for texture shaders consists of a single bit to indicate whether or not texture shaders are enabled, a vector of three floating-point values for the constant eye vector, and n sets of per-texture unit state where n is the implementation-dependent number of supported texture units. The set of per-texture unit texture shader state consists of the twenty-one-valued integer indicating the texture shader operation, four two-valued integers indicating the cull modes, an integer indicating the previous texture unit input, a two-valued integer indicating the RGBA unsigned dot product mapping mode, a 2x2 floating-point matrix indicating the texture offset transform, a floating-point value indicating the texture offset scale, a floating-point value indicating the texture offset bias, and a bit to indicate whether or not the texture shader stage is consistent.

In the initial state, the texture shaders state is set as follows: the texture shaders enable is disabled; the constant eye vector is (0,0,-1); all the texture shader operations are NONE; the RGBA unsigned dot product mapping mode is UNSIGNED_IDENTITY_NV; all the cull mode values are GEQUAL; all the previous texture units are TEXTURE0_ARB; each texture offset matrix is an identity matrix; all texture offset scales are 1.0; and all texture offset biases are 0.0."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Texture Environments and Texture Functions"

Change the third sentence of the third paragraph to read:

"The env argument to GetTexEnv must be one of TEXTURE_ENV, TEXTURE_FILTER_CONTROL_EXT, or TEXTURE_SHADER_NV."

Add to the end of the third paragraph:

"For GetTexEnv, when the target is TEXTURE_SHADER_NV, the texture shader stage consistency can be queried with SHADER_CONSISTENT_NV."

Add the following to the end of the fourth paragraph:

"Queries of TEXTURE_BORDER_COLOR return the same values as the TEXTURE_BORDER_VALUES query."

-- **Section 6.1.4 "Texture Queries"**

Add the following to the end of the fourth paragraph:

"Calling GetTexImage with a color format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA) when the texture image is of a format type (see table 3.15) other than RGBA (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context) causes the error INVALID_OPERATION. Calling GetTexImage with a format of HILO when the texture image is of a format type (see table 3.15) other than HILO causes the error INVALID_OPERATION. Calling GetTexImage with a format of DSDT_NV when the texture image is of a base internal format other than DSDT_NV causes the error INVALID_OPERATION. Calling GetTexImage with a format of DSDT_MAG_NV when the texture image is of a base internal format other than DSDT_MAG_NV causes the error INVALID_OPERATION. Calling GetTexImage with a format of DSDT_MAG_VIB_NV when the texture image is of a base internal format other than DSDT_MAG_INTENSITY_NV causes the error INVALID_OPERATION."

Additions to the GLX Specification

None

Dependencies on ARB_texture_env_add or EXT_texture_env_add

If neither ARB_texture_env_add nor EXT_texture_env_add are implemented, then the references to ADD are invalid and should be ignored.

Dependencies on ARB_texture_env_combine or EXT_texture_env_combine

If neither ARB_texture_env_combine nor EXT_texture_env_combine are implemented, then the references to COMBINE_ARB and COMBINE_EXT are invalid and should be ignored.

Dependencies on EXT_texture_lod_bias

If EXT_texture_lod_bias is not implemented, then the references to TEXTURE_FILTER_CONTROL_EXT are invalid and should be ignored.

Dependencies on NV_texture_env_combine4

If NV_texture_env_combine4 is not implemented, then the references to COMBINE4_NV are invalid and should be ignored.

Dependencies on NV_texture_rectangle

If NV_texture_rectangle is not implemented, then the references to TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, and DOT_PRODUCT_TEXTURE_RECTANGLE_NV are invalid and should be ignored.

Errors

INVALID_ENUM is generated if one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_MAG_VIBRANCE_NV is used as the format for DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, or GetMinMax.

INVALID_ENUM is generated if either UNSIGNED_INT_S8_S8_8_8_NV or UNSIGNED_INT_8_8_S8_S8_REV is used as the type for DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, or GetMinMax.

INVALID_OPERATION is generated if a packed pixel format type listed in table 3.8 is used with DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, GetMinMax, TexImage1D, TexImage2D, TexSubImage1D, TexSubImage2D, TexSubImage3d, or GetTexImage but the format parameter does not match on of the allowed Matching Pixel Formats listed in table 3.8 for the specified packed type parameter.

INVALID_OPERATION is generated when TexImage1D or TexImage2D are called and the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV; or if the internalformat is HILO_NV and the format is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, or SIGNED_HILO16_NV.

INVALID_OPERATION is generated when TexImage2D, or TexImage1D is called and if the format is DSDT_NV and the internalformat is not either DSDT_NV or DSDT8_NV; or if the internal format is either DSDT_NV or DSDT8_NV and the format is not DSDT_NV.

INVALID_OPERATION is generated when TexImage2D, or TexImage1D is called and if the format is DSDT_MAG_NV and the internalformat is not either DSDT_MAG_NV or DSDT8_MAG8_NV; or if the internal format is either DSDT_MAG_NV or DSDT8_MAG8_NV and the format is not DSDT_MAG_NV.

INVALID_OPERATION is generated when TexImage2D or TexImage1D is called and if the format is DSDT_MAG_VIB_NV and the internalformat is not either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV; or if the internal format is either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV and the format is not DSDT_MAG_VIB_NV.

INVALID_OPERATION is generated when CopyTexImage2D,
CopyTexImage1D, CopyTexSubImage2D, or
CopyTexSubImage1D is called and the internal format of the texture
array to which the pixels are to be copied is one of HILO_NV,
HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV,
DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or
DSDT8_MAG8_INTENSITY8_NV.

INVALID_OPERATION is generated when TexSubImage2D or
TexSubImage1D is called and the texture array's base internal format
is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV,
and the format parameter is not one of COLOR_INDEX, RED,
GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or
LUMINANCE_ALPHA

INVALID_OPERATION is generated when TexSubImage2D or
TexSubImage1D is called and the texture array's base internal format
is HILO_NV and the format parameter is not HILO_NV.

INVALID_OPERATION is generated when TexSubImage2D or
TexSubImage1D is called and the texture array's base internal format
is DSDT_NV and the format parameter is not DSDT_NV.

INVALID_OPERATION is generated when TexSubImage2D or
TexSubImage1D is called and the texture array's base internal format
is DSDT_MAG_NV and the format parameter is not DSDT_MAG_NV.

INVALID_OPERATION is generated when TexSubImage2D
or TexSubImage1D is called and the texture array's base internal
format is DSDT_MAG_INTENSITY_NV and the format parameter is not
DSDT_MAG_VIRBANCE_NV.

INVALID_OPERATION is generated when TexEnv is called and the
PREVIOUS_TEXTURE_INPUT_NV parameter for texture unit i is assigned
the value TEXTUREi_ARB where f i is greater than or equal to the
current active texture unit.

INVALID_OPERATION is generated when TexEnv is called and the
SHADER_OPERATION_NV parameter for texture unit 0 is assigned
one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV,
OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV,
DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV,
DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV.
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

INVALID_OPERATION is generated when TexEnv is called
and the SHADER_OPERATION_NV parameter for texture
unit 1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

INVALID_OPERATION is generated when TexEnv is called
and the SHADER_OPERATION_NV parameter for texture
unit 2 is assigned one of
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

INVALID_OPERATION is generated when TexEnv is called and the
SHADER_OPERATION_NV parameter for texture unit n-1 (where n is the
number of supported texture units) is assigned either DOT_PRODUCT_NV
or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV.

INVALID_OPERATION is generated when GetTexImage is called with a
color format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE,
or LUMINANCE_ALPHA) when the texture image is of a format type (see
table 3.15) other than RGBA (the DSDT_MAG_INTENSITY_NV base internal
format does not count as an RGBA format type in this context).

INVALID_OPERATION is generated when GetTexImage is called with
a format of HILO when the texture image is of a format type (see
table 3.15) other than HILO.

INVALID_OPERATION is generated when GetTexImage is called with a
format of DSDT_NV when the texture image is of a base internal
format other than DSDT_NV.

INVALID_OPERATION is generated when GetTexImage is called with a
format of DSDT_MAG_NV when the texture image is of a base internal
format other than DSDT_MAG_NV.

INVALID_OPERATION is generated when GetTexImage is called with a
format of DSDT_MAG_VIBRANCE_NV when the texture image is of a base
internal format other than DSDT_MAG_INTENSITY_NV causes the error
INVALID_OPERATION."

New State

Change the TEXTURE_BORDER_COLOR line in table 6.13 to read:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_BORDER_COLOR	4xR	GetTexParameter	(0,0,0,0)	Texture border values	3.8	texture

Table 6.TextureShaders. Texture Shaders.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
HI_BIAS_NV	R	GetFloatv	0.0	Hi bias for HILO	3.6.3	pixel
LO_BIAS_NV	R	GetFloatv	0.0	Lo bias for HILO	3.6.3	pixel
DS_BIAS_NV	R	GetFloatv	0.0	Ds bias	3.6.3	pixel
DT_BIAS_NV	R	GetFloatv	0.0	Dt bias	3.6.3	pixel
MAGNITUDE_BIAS_NV	R	GetFloatv	0.0	Magnitude bias	3.6.3	pixel
VIBRANCE_BIAS_NV	R	GetFloatv	0.0	Vibrance bias	3.6.3	pixel
HI_SCALE_NV	R	GetFloatv	1.0	Hi scale	3.6.3	pixel
LO_SCALE_NV	R	GetFloatv	1.0	Lo scale	3.6.3	pixel
DS_SCALE_NV	R	GetFloatv	1.0	Ds scale	3.6.3	pixel
DT_SCALE_NV	R	GetFloatv	1.0	Dt scale	3.6.3	pixel
MAGNITUDE_SCALE_NV	R	GetFloatv	1.0	Magnitude scale	3.6.3	pixel
VIBRANCE_SCALE_NV	R	GetFloatv	1.0	Vibrance scale	3.6.3	pixel
TEXTURE_SHADER_NV	B	IsEnabled	False	Texture shaders enable	3.8	texture/enable
SHADER_OPERATION_NV	TxZ21	GetTexEnviv	NONE	Texture shader operation	3.8.13	texture
CULL_MODES_NV	Tx4xZ2	GetTexEnviv	GEQUAL,GEQUAL, GEQUAL,GEQUAL	Texture shader cull fragment modes	3.8.13	texture
RGBA_UNSIGNED_- DOT_PRODUCT_MAPPING_NV	TxZ2	GetTexEnviv	UNSIGNED_IDENTITY_NV	Texture shader RGBA dot product mapping	3.8.13	texture
PREVIOUS_TEXTURE_INPUT_NV	TxZn	GetTexEnviv	TEXTURE0_ARB	Texture shader previous tex input	3.8.13	texture
CONST_EYE_NV	TxRx3	GetTexEnvfv	(0,0,-1)	Shader constant eye vector	3.8.13	texture
OFFSET_TEXTURE_MATRIX_NV	TxM2	GetTexEnvfv	(1,0,0,1)	2x2 texture offset matrix	3.8.13	texture
OFFSET_TEXTURE_SCALE_NV	TxR	GetTexEnvfv	1	Texture offset scale	3.8.13	texture
OFFSET_TEXTURE_BIAS_NV	TxR	GetTexEnvfv	0	Texture offset bias	3.8.13	texture
SHADER_CONSISTENT_NV	TxB	GetTexEnviv	True	Texture shader stage consistency	3.8.13	texture

[The "Tx" type prefix means that the state is per-texture unit.]

[The "Zn" type is an n-valued integer where n is the implementation-dependent number of texture units supported.]

New Implementation State

None

Revision History

March 29, 2001 - document that using signed HILO with a dot product shader forces the square root to zero if the 1.0-HI*HI-LO*LO value is negative.

Name

NV_texture_shader2

Name Strings

GL_NV_texture_shader2

Notice

Copyright NVIDIA Corporation, 1999, 2000, 2001.

IP Status

NVIDIA Proprietary.

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_texture_shader2.txt#6 $

Number

231

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification, augmented by the NV_texture_shader extension specification.

Requires support for the NV_texture_shader extension.

Overview

This extension extends the NV_texture_shader functionality to support texture shader operations for 3D textures.

See the NV_texture_shader extension for information about the texture shader operational model.

The two new texture shader operations are:

<conventional textures>

22. TEXTURE_3D - Accesses a 3D texture via (s/q,t/q,r/q).

<dot product textures>

23. DOT_PRODUCT_TEXTURE_3D_NV - When preceded by two DOT_PRODUCT_NV programs in the previous two texture shader stages, computes a third similar dot product and composes the three dot products into (s,t,r) texture coordinate set to access a 3D non-projective texture.

Issues

Why a separate extension?

Not all implementations of NV_texture_shader will support 3D textures in hardware.

Breaking this extension out into a distinct extension allows OpenGL programs that only would use 3D textures if they are supported in hardware to determine whether hardware support is available by explicitly looking for the NV_texture_shader2 extension.

What if an implementation wanted to support NV_texture_shader2 operations within a software rasterizer?

Implementations should be free to implement the 3D texture texture shader operations in software. In this case, the implementation should NOT advertise the NV_texture_shader2 extension, but should still accept the GL_TEXTURE_3D and GL_DOT_PRODUCT_TEXTURE_3D_NV texture shader operations without an error. Likewise, the glTexImage3D and glCopyTexImage3D commands should accept the new internal texture formats, formats, and types allowed by this extension should be accepted without an error.

When NV_texture_shader2 is not advertised in the GL_EXTENSIONS string, but the extension functionality works without GL errors, programs should expect that these two texture shader operations are slow.

New Procedures and Functions

None.

New Tokens

When the <target> and <pname> parameters of TexEnvf, TexEnvfv, TexEnvi, and TexEnviv are TEXTURE_SHADER_NV and SHADER_OPERATION_NV respectively, then the value of <param> or the value pointed to by <params> may be:

TEXTURE_3D	
DOT_PRODUCT_TEXTURE_3D_NV	0x86EF

Accepted by the <format> parameter of TexImage3D and TexSubImage3D:

HILO_NV	0x86F4
DSDT_NV	0x86F5
DSDT_MAG_NV	0x86F6
DSDT_MAG_VIB_NV	0x86F7

Accepted by the <type> parameter of TexImage3D and TexSubImage3D:

UNSIGNED_INT_S8_S8_8_8_NV	0x86DA
UNSIGNED_INT_8_8_S8_S8_REV_NV	0x86DB

Accepted by the <internalformat> parameter of TexImage3D:

SIGNED_RGBA_NV	0x86FB
SIGNED_RGBA8_NV	0x86FC
SIGNED_RGB_NV	0x86FE
SIGNED_RGB8_NV	0x86FF
SIGNED_LUMINANCE_NV	0x8701
SIGNED_LUMINANCE8_NV	0x8702
SIGNED_LUMINANCE_ALPHA_NV	0x8703
SIGNED_LUMINANCE8_ALPHA8_NV	0x8704
SIGNED_ALPHA_NV	0x8705
SIGNED_ALPHA8_NV	0x8706
SIGNED_INTENSITY_NV	0x8707
SIGNED_INTENSITY8_NV	0x8708
SIGNED_RGB_UNSIGNED_ALPHA_NV	0x870C
SIGNED_RGB8_UNSIGNED_ALPHA8_NV	0x870D

Accepted by the <internalformat> parameter of TexImage3D:

HILO_NV	
HILO16_NV	0x86F8
SIGNED_HILO_NV	0x86F9
SIGNED_HILO16_NV	0x86FA
DSDT_NV	
DSDT8_NV	0x8709
DSDT_MAG_NV	
DSDT8_MAG8_NV	0x870A
DSDT_MAG_INTENSITY_NV	0x86DC
DSDT8_MAG8_INTENSITY8_NV	0x870B

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8 "Texturing"

Replace the third paragraph (amended by the NV_texture_shader extension) with the following that includes 3D texture references:

"The alternative to conventional texturing is the texture shaders mechanism. When texture shaders are enabled, each texture unit uses one of twenty-three texture shader operations. Twenty of the twenty-three shader operations map an (s,t,r,q) texture coordinate set to an RGBA color. Of these, four texture shader operations directly correspond to the 1D, 2D, 3D, and cube map conventional texturing operations. Depending on the texture shader operation, the mapping from the (s,t,r,q) texture coordinate set to an RGBA color may depend on the given texture unit's currently bound texture object state and/or the results of previous texture shader operations. The three remaining texture shader operations respectively provide a fragment culling mechanism based on texture coordinates, a means to replace the fragment depth value, and a dot product operation that computes a floating-point value for use by

subsequent texture shaders. The specifics of each texture shader operation are described in section 3.8.12."

-- **Section 3.8.2 "Alternate Texture Image Specification Commands"**

Amend the following text inserted by NV_texture_shader after the six paragraph to include 3D texture references:

"CopyTexSubImage3D, CopyTexSubImage2D, and CopyTexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the pixels are to be copied is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

TexSubImage3D, TexSubImage2D, and TexSubImage1D generate the error INVALID_OPERATION if the internal format of the texture array to which the texels are to be copied has a different format type (according to table 3.15) than the format type of the texels being specified. Specifically, if the base internal format is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV, then the format parameter must be one of COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA; if the base internal format is HILO_NV, then the format parameter must be HILO_NV; if the base internal format is DSDT_NV, then the format parameter must be DSDT_NV; if the base internal format is DSDT_MAG_NV, then the format parameter must be DSDT_MAG_NV; if the base internal format is DSDT_MAG_INTENSITY_NV, the format parameter must be DSDT_MAG_VIB_NV."

-- **Section 3.8.13 "Texture Shaders"**

Amend the designated paragraphs of the NV_texture_shader specification to include discussion of 3D textures.

1st paragraph:

"Each texture unit is configured with one of twenty-three texture shader operations. Several texture shader operations require additional state. All per-texture shader stage state is specified using the TexEnv commands with the target specified as TEXTURE_SHADER_NV. The per-texture shader state is replicated per texture unit so the texture unit selected by ActiveTextureARB determines which texture unit's environment is modified by TexEnv calls."

3rd paragraph:

"When TexEnv is called with the target of TEXTURE_SHADER_NV, SHADER_OPERATION_NV may be set to one of NONE, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, PASS_THROUGH_NV, CULL_FRAGMENT_NV, OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,

DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV, or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. The semantics of each of these shader operations is described in section 3.8.13.1. Not every operation is supported in every texture unit. The restrictions for how these shader operations can be configured in various texture units are described in section 3.8.13.2."

3.8.13.1 Texture Shader Operations

Amend tables 3.A, 3.B, 3.C, and 3.D in the NV_texture_shader specification to include entries for 3D texture operations:

Table 3.A:

texture shader operation i	previous texture input	texture shader operation i-1	operation i-2	texture shader operation i+1
TEXTURE_3D	-	-	-	-
DOT_PRODUCT_TEXTURE_3D_NV	shader result type must be one of signed HILO, unsigned HILO, all signed RGBA, all unsigned RGBA	shader operation must be DOT_PRODUCT_NV	shader operation must be DOT_PRODUCT_NV	-

Table 3.B:

texture shader operation i	texture unit i
TEXTURE_3D	3D target must be consistent
DOT_PRODUCT_TEXTURE_3D_NV	3D target must be consistent

Table 3.C:

texture shader operation i	texture coordinate set usage	texture target	uses stage result i-1	uses stage result i-2	uses stage result i+1	uses previous texture input	uses cull modes	uses offset texture 2D matrix	offset texture 2D scale and bias	uses const eye vector
TEXTURE_3D	s,t,r,q	3D	-	-	-	-	-	-	-	-
DOT_PRODUCT_TEXTURE_3D_NV	s,t,r	3D	y	y	-	y	-	-	-	-

Table 3.D:

texture shader operation i	shader stage result type	shader stage result	texture unit RGBA color result
TEXTURE_3D	matches 3D target type	filtered 3D target texel	if 3D target texture type is RGBA, filtered 3D target texel, else (0,0,0,0)
DOT_PRODUCT_TEXTURE_3D_NV	matches 3D target type	filtered 3D target texel	if 3D target texture type is RGBA, filtered 3D target texel, else (0,0,0,0)

Add the following new sections specifying new 3D texture operations:

3.8.13.1.22 3D Projective Texturing

The TEXTURE_3D texture shader operation accesses the texture unit's

3D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (s/q,t/q,r/q) for the 3D texture coordinates where s, t, r, and q are the homogeneous texture coordinates for the texture unit. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture. This mode is equivalent to conventional texturing's 3D texture target.

If the texture unit's 3D texture object is not consistent, then this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if it is the NONE operation.

3.8.13.1.23 Dot Product Texture 3D

The DOT_PRODUCT_TEXTURE_3D_NV texture shader operation accesses the texture unit's 3D texture object (as described in sections 3.8.4, 3.8.5, and 3.8.6) using (dotPP,dotP,dotC) for the 3D texture coordinates. The result of the texture access becomes both the shader result and texture unit RGBA result (see table 3.E). The type of the shader result depends on the format type of the accessed texture.

Assuming that i is the current texture shader stage, dotPP is the floating-point dot product texture shader result from the i-2 texture shader stage, assuming the i-2 texture shader stage's operation is DOT_PRODUCT_NV. dotP is the floating-point dot product texture shader result from the i-1 texture shader stage, assuming the i-1 texture shader stage's operation is DOT_PRODUCT_NV. dotC is the floating-point dot product result from the current texture shader stage. dotC is computed in the identical manner used to compute the floating-point result of the DOT_PRODUCT_NV texture shader described in section 3.8.13.1.14.

If the previous texture input texture object specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value has a format type other than RGBA or HILO (the DSDT_MAG_INTENSITY_NV base internal format does not count as an RGBA format type in this context), then this texture shader stage is not consistent.

If the previous texture input texture shader operation specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If the previous texture input texture shader result specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is invalid, then this texture shader stage is not consistent.

If the previous texture input shader stage specified by the current texture shader stage's PREVIOUS_TEXTURE_INPUT_NV value is not consistent, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage operation is not DOT_PRODUCT_NV, then this texture shader stage is not consistent.

If either the i-1 or i-2 texture shader stage is not consistent, then

this texture shader stage is not consistent.

If the texture unit's 3D texture object is not consistent, then
this texture shader stage is not consistent.

If this texture shader stage is not consistent, it operates as if
it is the NONE operation.

3.8.13.2 Texture Shader Restrictions

Amend the first four paragraphs in this section to include 3D
texture operations:

"There are various restrictions on possible texture shader
configurations. These restrictions are described in this section.

The error INVALID_OPERATION occurs if the SHADER_OPERATION_NV
parameter for texture unit 0 is assigned one of
OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV,
OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV,
DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV,
DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these
texture shaders requires a previous texture shader result that
is not possible for texture unit 0. Therefore these shaders are
disallowed for texture unit 0.

The error INVALID_OPERATION occurs if the
SHADER_OPERATION_NV parameter for texture unit
1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture
shaders requires either two previous texture shader results or
a dot product result that cannot be generated by texture unit 0.
Therefore these shaders are disallowed for texture unit 1.

The error INVALID_OPERATION occurs if the
SHADER_OPERATION_NV parameter for texture unit
2 is assigned one of DOT_PRODUCT_TEXTURE_3D_NV,
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV. Each of these texture
shaders requires three previous texture shader results. Therefore
these shaders are disallowed for texture unit 2."

3.8.13.3 Required State

Amend the first paragraph in this section to account for the 2 new
3D texture shader operations:

"The state required for texture shaders consists of a single bit to
indicate whether or not texture shaders are enabled, a vector of
three floating-point values for the constant eye vector, and n sets

of per-texture unit state where n is the implementation-dependent number of supported texture units. The set of per-texture unit texture shader state consists of the twenty-three-valued integer indicating the texture shader operation, four two-valued integers indicating the cull modes, an integer indicating the previous texture unit input, a two-valued integer indicating the RGBA unsigned dot product mapping mode, a 2x2 floating-point matrix indicating the texture offset transform, a floating-point value indicating the texture offset scale, a floating-point value indicating the texture offset bias, and a bit to indicate whether or not the texture shader stage is consistent."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on other specifications

Same as the NV_texture_shader extension.

Errors

The following errors are updated to reflect 3D texture operations:

INVALID_OPERATION is generated if a packed pixel format type listed in table 3.8 is used with DrawPixels, ReadPixels, ColorTable, ColorSubTable, ConvolutionFilter1D, ConvolutionFilter2D, SeparableFilter2D, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, GetMinMax, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3d, or GetTexImage but the format parameter does not match on of the allowed Matching Pixel Formats listed in table 3.8 for the specified packed type parameter.

INVALID_OPERATION is generated when TexImage1D, TexImage2D, or TexImage3D are called and the format is HILO_NV and the internalformat is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV; or if the internalformat is HILO_NV and the format is not one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, or SIGNED_HILO16_NV.

INVALID_OPERATION is generated when TexImage3D, TexImage2D, or TexImage1D is called and if the format is DSDT_NV and the internalformat is not either DSDT_NV or DSDT8_NV; or if the internal

format is either DSDT_NV or DSDT8_NV and the format is not DSDT_NV.

INVALID_OPERATION is generated when TexImage3D, TexImage2D, or TexImage1D is called and if the format is DSDT_MAG_NV and the internalformat is not either DSDT_MAG_NV or DSDT8_MAG8_NV; or if the internal format is either DSDT_MAG_NV or DSDT8_MAG8_NV and the format is not DSDT_MAG_NV.

INVALID_OPERATION is generated when TexImage3D, TexImage2D, or TexImage1D is called and if the format is DSDT_MAG_VIB_NV and the internalformat is not either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV; or if the internal format is either DSDT_MAG_INTENSITY_NV or DSDT8_MAG8_INTENSITY8_NV and the format is not DSDT_MAG_VIB_NV.

INVALID_OPERATION is generated when CopyTexImage3D, CopyTexImage2D, CopyTexImage1D, CopyTexSubImage3D, CopyTexSubImage2D, or CopyTexSubImage1D is called and the internal format of the texture array to which the pixels are to be copied is one of HILO_NV, HILO16_NV, SIGNED_HILO_NV, SIGNED_HILO16_NV, DSDT_NV, DSDT8_NV, DSDT_MAG_NV, DSDT8_MAG8_NV, DSDT_MAG_INTENSITY_NV, or DSDT8_MAG8_INTENSITY8_NV.

INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or TexSubImage1D is called and the texture array's base internal format is not one of HILO_NV, DSDT_NV, DSDT_MAG_NV, or DSDT_INTENSITY_NV, and the format parameter is not one of COLOR_INDEX, RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, or LUMINANCE_ALPHA

INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or TexSubImage1D is called and the texture array's base internal format is HILO_NV and the format parameter is not HILO_NV.

INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or TexSubImage1D is called and the texture array's base internal format is DSDT_NV and the format parameter is not DSDT_NV.

INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or TexSubImage1D is called and the texture array's base internal format is DSDT_MAG_NV and the format parameter is not DSDT_MAG_NV.

INVALID_OPERATION is generated when TexSubImage3D, TexSubImage2D, or TexSubImage1D is called and the texture array's base internal format is DSDT_MAG_INTENSITY_NV and the format parameter is not DSDT_MAG_VIRBANCE_NV.

INVALID_OPERATION is generated when TexEnv is called and the SHADER_OPERATION_NV parameter for texture unit 0 is assigned one of OFFSET_TEXTURE_2D_NV, OFFSET_TEXTURE_2D_SCALE_NV, OFFSET_TEXTURE_RECTANGLE_NV, OFFSET_TEXTURE_RECTANGLE_SCALE_NV, DEPENDENT_AR_TEXTURE_2D_NV, DEPENDENT_GB_TEXTURE_2D_NV, DOT_PRODUCT_NV, DOT_PRODUCT_DEPTH_REPLACE_NV, DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV, DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV. or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

INVALID_OPERATION is generated when TexEnv is called
and the SHADER_OPERATION_NV parameter for texture
unit 1 is assigned one of DOT_PRODUCT_DEPTH_REPLACE_NV,
DOT_PRODUCT_TEXTURE_2D_NV, DOT_PRODUCT_TEXTURE_RECTANGLE_NV,
DOT_PRODUCT_TEXTURE_3D_NV, DOT_PRODUCT_TEXTURE_CUBE_MAP_NV,
DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

INVALID_OPERATION is generated when TexEnv is called
and the SHADER_OPERATION_NV parameter for texture
unit 2 is assigned one of DOT_PRODUCT_TEXTURE_3D_NV,
DOT_PRODUCT_TEXTURE_CUBE_MAP_NV, DOT_PRODUCT_REFLECT_CUBE_MAP_NV,
or DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV.

INVALID_OPERATION is generated when TexEnv is called and the
SHADER_OPERATION_NV parameter for texture unit n-1 (where n is the
number of supported texture units) is assigned either DOT_PRODUCT_NV
or DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV.

INVALID_OPERATION is generated when GetTexImage is called with a
color format (one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE,
or LUMINANCE_ALPHA) when the texture image is of a format type (see
table 3.15) other than RGBA (the DSDT_MAG_INTENSITY_NV base internal
format does not count as an RGBA format type in this context).

INVALID_OPERATION is generated when GetTexImage is called with
a format of HILO when the texture image is of a format type (see
table 3.15) other than HILO.

INVALID_OPERATION is generated when GetTexImage is called with a
format of DSDT_NV when the texture image is of a base internal
format other than DSDT_NV.

INVALID_OPERATION is generated when GetTexImage is called with a
format of DSDT_MAG_NV when the texture image is of a base internal
format other than DSDT_MAG_NV.

INVALID_OPERATION is generated when GetTexImage is called with a
format of DSDT_MAG_VIBRANCE_NV when the texture image is of a base
internal format other than DSDT_MAG_INTENSITY_NV causes the error
INVALID_OPERATION."

New State

Table 6.TextureShaders. Texture Shaders.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SHADER_OPERATION_NV	TxZ23	GetTexEnviv	NONE	Texture shader operation	3.8.13	texture

* Z21 in NV_texture_shader is now Z23 with NV_texture_shader2.

[The "Tx" type prefix means that the state is per-texture unit.]

[The "Zn" type is an n-valued integer where n is the implementation-dependent number of texture units supported.]

New Implementation State

None

Revision History

None

Name

NV_vertex_array_range

Name Strings

GL_NV_vertex_array_range

Notice

Copyright NVIDIA Corporation, 1999, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Existing functionality is augmented by NV_vertex_array_range2.

Version

NVIDIA Date: April 4, 2001 (version 1.1)
$Date$ $Revision$
$Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_array_range.txt#24 $

Number

190

Dependencies

None

Overview

The goal of this extension is to permit extremely high vertex processing rates via OpenGL vertex arrays even when the CPU lacks the necessary data movement bandwidth to keep up with the rate at which the vertex engine can consume vertices. CPUs can keep up if they can just pass vertex indices to the hardware and let the hardware "pull" the actual vertex data via Direct Memory Access (DMA). Unfortunately, the current OpenGL 1.1 vertex array functionality has semantic constraints that make such an approach hard. Hence, the vertex array range extension.

This extension provides a mechanism for deferring the pulling of vertex array elements to facilitate DMAed pulling of vertices for fast, efficient vertex array transfers. The OpenGL client need only pass vertex indices to the hardware which can DMA the actual index's vertex data directly out of the client address space.

The OpenGL 1.1 vertex array functionality specifies a fairly strict coherency model for when OpenGL extracts vertex data from a vertex array and when the application can update the in memory vertex array data. The OpenGL 1.1 specification says "Changes made to array data between the execution of Begin and the

corresponding execution of End may affect calls to ArrayElement that are made within the same Begin/End period in non-sequential ways. That is, a call to ArrayElement that precedes a change to array data may access the changed data, and a call that follows a change to array data may access the original data."

This means that by the time End returns (and DrawArrays and DrawElements return since they have implicit Ends), the actual vertex array data must be transferred to OpenGL. This strict coherency model prevents us from simply passing vertex element indices to the hardware and having the hardware "pull" the vertex data out (which is often long after the End for the primitive has returned to the application).

Relaxing this coherency model and bounding the range from which vertex array data can be pulled is key to making OpenGL vertex array transfers faster and more efficient.

The first task of the vertex array range extension is to relax the coherency model so that hardware can indeed "pull" vertex data from the OpenGL client's address space long after the application has completed sending the geometry primitives requiring the vertex data.

The second problem with the OpenGL 1.1 vertex array functionality is the lack of any guidance from the API about what region of memory vertices can be pulled from. There is no size limit for OpenGL 1.1 vertex arrays. Any vertex index that points to valid data in all enabled arrays is fair game. This makes it hard for a vertex DMA engine to pull vertices since they can be potentially pulled from anywhere in the OpenGL client address space.

The vertex array range extension specifies a range of the OpenGL client's address space where vertices can be pulled. Vertex indices that access any array elements outside the vertex array range are specified to be undefined. This permits hardware to DMA from finite regions of OpenGL client address space, making DMA engine implementation tractable.

The extension is specified such that an (error free) OpenGL client using the vertex array range functionality could no-op its vertex array range commands and operate equivalently to using (if slower than) the vertex array range functionality.

Because different memory types (local graphics memory, AGP memory) have different DMA bandwidths and caching behavior, this extension includes a window system dependent memory allocator to allocate cleanly the most appropriate memory for constructing a vertex array range. The memory allocator provided allows the application to tradeoff the desired CPU read frequency, CPU write frequency, and memory priority while still leaving it up to OpenGL implementation the exact memory type to be allocated.

Issues

How does this extension interact with the compiled_vertex_array extension?

I think they should be independent and not interfere with each other. In practice, if you use NV_vertex_array_range, you can surpass the performance of compiled_vertex_array

Should some explanation be added about what happens when an OpenGL application updates its address space in regions overlapping with the currently configured vertex array range?

RESOLUTION: I think the right thing is to say that you get non-sequential results. In practice, you'll be using an old context DMA pointing to the old pages.

If the application change's its address space within the vertex array range, the application should call glVertexArrayRangeNV again. That will re-make a new vertex array range context DMA for the application's current address space.

If we are falling back to software transformation, do we still need to abide by leaving "undefined" vertices outside the vertex array range? For example, pointers that are not 32-bit aligned would likely cause a fall back.

RESOLUTION: No. The fact that vertex is "undefined" means we can do anything we want (as long as we send a vertex and do not crash) so it is perfectly fine for the software puller to grab vertex information not available to the hardware puller.

Should we give a programmer a sense of how big a vertex array range they can specify?

RESOLUTION: No. Just document it if there are limitations. Probably very hardware and operating system dependent.

Is it clear enough that language about ArrayElement also applies to DrawArrays and DrawElements?

Maybe not, but OpenGL 1.1 spec is clear that DrawArrays and DrawElements are defined in terms of ArrayElement.

Should glFlush be the same as glVertexArrayRangeFlush?

RESOLUTION: No. A glFlush is cheaper than a glVertexArrayRangeFlush though a glVertexArrayRangeFlushNV should do a flush.

If any the data for any enabled array for a given array element index falls outside of the vertex array range, what happens?

RESOLUTION: An undefined vertex is generated.

What error is generated in this case?

I don't know yet. We should make sure the hardware really does let us know when vertices are undefined.

Note that this is a little weird for OpenGL since most errors in OpenGL result in the command being ignored. Not in this

case though.

Should this extension support an interface for allocating video and AGP memory?

RESOLUTION: YES. It seems like we should be able to leave the task of memory allocation to DirectDraw, but DirectDraw's asynchronous unmapping behavior and having to hold locks to update DirectDraw surfaces makes that mechanism to cumbersome.

Plus the API is a lot easier if we do it ourselves.

How do we decide what type of memory to allocate for the application?

RESOLUTION: Usage hints. The application rates the read frequency (how often will they read the memory), the write frequency (how often will they write the memory), and the priority (how important is this memory relative to other uses for the memory such as texturing) on a scale of 1.0 to 0.0. Using these hints and the size of the memory requsted, the OpenGL implementation decides where to allocate the memory.

We try to not directly expose particular types of memory (AGP, local memory, cached/uncached, etc) so future memory types can be supported by merely updating the OpenGL implementation.

Should the memory allocator functionality be available be a part of the GL or window system dependent (GLX or WGL) APIs?

RESOLUTION: The window system dependent API.

The memory allocator should be considered a window system/ operating system dependent operation. This also permits memory to be allocated when no OpenGL rendering contexts exist yet.

New Procedures and Functions

```
void VertexArrayRangeNV(sizei length, void *pointer)
void FlushVertexArrayRangeNV(void)
```

New Tokens

Accepted by the <cap> parameter of EnableClientState, DisableClientState, and IsEnabled:

```
VERTEX_ARRAY_RANGE_NV              0x851D
```

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
VERTEX_ARRAY_RANGE_LENGTH_NV       0x851E
VERTEX_ARRAY_RANGE_VALID_NV        0x851F
MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV  0x8520
```

Accepted by the <pname> parameter of GetPointerv:

VERTEX_ARRAY_RANGE_POINTER_NV 0x8521

Additions to Chapter 2 of the OpenGL 1.1 Specification (OpenGL Operation)

After the discussion of vertex arrays (Section 2.8) add a description of the vertex array range:

"The command

void VertexArrayRangeNV(sizei length, void *pointer)

specifies the current vertex array range. When the vertex array range is enabled and valid, vertex array vertex transfers from within the vertex array range are potentially faster. The vertex array range is a contiguous region of (virtual) address space for placing vertex arrays. The "pointer" parameter is a pointer to the base of the vertex array range. The "length" pointer is the length of the vertex array range in basic machine units (typically unsigned bytes).

The vertex array range address space region extends from "pointer" to "pointer + length - 1" inclusive. When specified and enabled, vertex array vertex transfers from within the vertex array range are potentially faster.

There is some system burden associated with establishing a vertex array range (typically, the memory range must be locked down). If either the vertex array range pointer or size is set to zero, the previously established vertex array range is released (typically, unlocking the memory).

The vertex array range may not be established for operating system dependent reasons, and therefore, not valid. Reasons that a vertex array range cannot be established include spanning different memory types, the memory could not be locked down, alignment restrictions are not met, etc.

The vertex array range is enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_NV.

The vertex array range is either valid or invalid and this state can be determined by querying VERTEX_ARRAY_RANGE_VALID_NV. The vertex array range is valid when the following conditions are met:

- VERTEX_ARRAY_RANGE_NV is enabled.
- VERTEX_ARRAY is enabled.
- VertexArrayRangeNV has been called with a non-null pointer and non-zero size.
- The vertex array range has been established.

- An implementation-dependent validity check based on the pointer alignment, size, and underlying memory type of the vertex array range region of memory.

- An implementation-dependent validity check based on the current vertex array state including the strides, sizes, types, and pointer alignments (but not pointer value) for currently enabled vertex arrays.

- Other implementation-dependent validaity checks based on other OpenGL rendering state.

Otherwise, the vertex array range is not valid. If the vertex array range is not valid, vertex array transfers will not be faster.

When the vertex array range is valid, ArrayElement commands may generate undefined vertices if and only if any indexed elements of the enabled arrays are not within the vertex array range or if the index is negative or greater or equal to the implementation-dependent value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV. If an undefined vertex is generated, an INVALID_OPERATION error may or may not be generated.

The vertex array cohenecy model specifies when vertex data must be be extracted from the vertex array memory. When the vertex array range is not valid, (quoting the specification) `Changes made to array data between the execution of Begin and the corresponding execution of End may effect calls to ArrayElement that are made within the same Begin/End period in non-sequential ways. That is, a call to ArrayElement that precedes a change to array data may access the changed data, and a call that follows a change to array data may access the original data.'

When the vertex array range is valid, the vertex array coherency model is relaxed so that changes made to array data until the next "vertex array range flush" may affects calls to ArrayElement in non-sequential ways. That is a call to ArrayElement that precedes a change to array data (without an intervening "vertex array range flush") may access the changed data, and a call that follows a change (without an intervening "vertex array range flush") to array data may access original data.

A 'vertex array range flush' occurs when one of the following operations occur:

- Finish returns.

- FlushVertexArrayRangeNV returns.

- VertexArrayRangeNV returns.

- DisableClientState of VERTEX_ARRAY_RANGE_NV returns.

- EnableClientState of VERTEX_ARRAY_RANGE_NV returns.

- Another OpenGL context is made current.

The client state required to implement the vertex array range consists of an enable bit, a memory pointer, an integer size, and a valid bit.

If the memory mapping of pages within the vertex array range changes, using the vertex array range may or may not result in undefined data being fetched from the vertex arrays when the vertex array range is enabled and valid. To ensure that the vertex array range reflects the address space's current state, the application is responsible for calling VertexArrayRange again after any memory mapping changes within the vertex array range."llo

Additions to Chapter 5 of the OpenGL 1.1 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists"

"VertexArrayRangeNV and FlushVertexArrayRangeNV are not complied into display lists but are executed immediately.

If a display list is compiled while VERTEX_ARRAY_RANGE_NV is enabled, the commands ArrayElement, DrawArrays, DrawElements, and DrawRangeElements are accumulated into a display list as if VERTEX_ARRAY_RANGE_NV is disabled."

Additions to the WGL interface:

"When establishing a vertex array range, certain types of memory may be more efficient than other types of memory. The commands

```
void *wglAllocateMemoryNV(sizei size,
                          float readFrequency,
                          float writeFrequency,
                          float priority)
void wglFreeMemoryNV(void *pointer)
```

allocate and free memory that may be more suitable for establishing an efficient vertex array range than memory allocated by other means. The wglAllocateMemoryNV command allocates <size> bytes of contiguous memory.

The <readFrequency>, <writeFrequency>, and <priority> parameters are usage hints that the OpenGL implementation can use to determine the best type of memory to allocate. These parameters range from 0.0 to 1.0. A <readFrequency> of 1.0 indicates that the application intends to frequently read the allocated memory; a <readFrequency> of 0.0 indicates that the application will rarely or never read the memory. A <writeFrequency> of 1.0 indicates that the application intends to frequently write the allocated memory; a <writeFrequency> of 0.0 indicates that the application will rarely write the memory. A <priority> parameter of 1.0 indicates that memory type should be the most efficient available memory, even at the expense of (for example) available texture memory; a <priority> of 0.0 indicates that the vertex array range does not require an efficient memory type (for example, so that more efficient memory is available for other purposes such as texture memory).

The OpenGL implementation is free to use the <size>, <readFrequency>, <writeFrequency>, and <priority> parameters to determine what memory type should be allocated. The memory types available and how the memory type is determined is implementation dependent (and the implementation is free to ignore any or all of the above parameters).

Possible memory types that could be allocated are uncached memory, write-combined memory, graphics hardware memory, etc. The intent of the wglAllocateMemoryNV command is to permit the allocation of memory for efficient vertex array range usage. However, there is no requirement that memory allocated by wglAllocateMemoryNV must be used to allocate memory for vertex array ranges.

If the memory cannot be allocated, a NULL pointer is returned (and no OpenGL error is generated). An implementation that does not support this extension's memory allocation interface is free to never allocate memory (always return NULL).

The wglFreeMemoryNV command frees memory allocated with wglAllocateMemoryNV. The <pointer> should be a pointer returned by wglAllocateMemoryNV and not previously freed. If a pointer is passed to wglFreeMemoryNV that was not allocated via wglAllocateMemoryNV or was previously freed (without being reallocated), the free is ignored with no error reported.

The memory allocated by wglAllocateMemoryNV should be available to all other threads in the address space where the memory is allocated (the memory is not private to a single thread). Any thread in the address space (not simply the thread that allocated the memory) may use wglFreeMemoryNV to free memory allocated by itself or any other thread.

Because wglAllocateMemoryNV and wglFreeMemoryNV are not OpenGL rendering commands, these commands do not require a current context. They operate normally even if called within a Begin/End or while compiling a display list."

Additions to the GLX Specification

Same language as the "Additions to the WGL Specification" section except all references to wglAllocateMemoryNV and wglFreeMemoryNV should be replaced with glXAllocateMemoryNV and glXFreeMemoryNV respectively.

Additional language:

"OpenGL implementations using GLX indirect rendering should fail to set up the vertex array range (failing to set the vertex array valid bit so the vertex array range functionality is not usable). Additionally, glXAllocateMemoryNV always fails to allocate memory (returns NULL) when used with an indirect rendering context."

GLX Protocol

None

Errors

INVALID_OPERATION is generated if VertexArrayRange or FlushVertexArrayRange is called between the execution of Begin and the corresponding execution of End.

INVALID_OPERATION may be generated if an undefined vertex is generated.

New State

Get Value	Get Command	Type	Initial Value	Attrib
VERTEX_ARRAY_RANGE_NV	IsEnabled	B	False	vertex-array
VERTEX_ARRAY_RANGE_POINTER_NV	GetPointerv	Z+	0	vertex-array
VERTEX_ARRAY_RANGE_LENGTH_NV	GetIntegerv	Z+	0	vertex-array
VERTEX_ARRAY_RANGE_VALID_NV	GetBooleanv	B	False	vertex-array

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV	GetIntegerv	Z+	65535

NV10 Implementation Details

This section describes implementation-defined limits for NV10:

The value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV is 65535.

This section describes bugs in the NV10 vertex array range. These bugs will be fixed in a future hardware release:

If VERTEX_ARRAY is enabled with a format of GL_SHORT and the vertex array range is valid, a vertex array vertex with an X, Y, Z, or W coordinate of -32768 is wrongly interpreted as zero. Example: the X,Y coordinate (-32768,-32768) is incorrectly read as (0,0) from the vertex array.

If TEXTURE_COORD_ARRAY is enabled with a format of GL_SHORT and the vertex array range is valid, a vertex array texture coord with an S, T, R, or Q coordinate of -32768 is wrongly interpreted as zero. Example: the S,T coordinate (-32768,-32768) is incorrectly read as (0,0) from the texture coord array.

This section describes the implementation-dependent validity checks for NV10.

- For the NV10 implementation-dependent validity check for the vertex array range region of memory to be true, all of the following must be true:

  1. The <pointer> must be 32-byte aligned.

```
    2.  The underlying memory types must all be the same (all
        standard system memory -OR- all AGP memory -OR- all video
        memory).

o  For the NV10 implementation-dependent validity check for the
   vertex array state to be true, all of the following must be
   true:

  1.  ( VERTEX_ARRAY must be enabled -AND-
        The vertex array stride must be less than 256 -AND-
        ( ( The vertex array type must be FLOAT -AND-
            The vertex array stride must be a multiple of 4 bytes -AND-
            The vertex array pointer must be 4-byte aligned -AND-
            The vertex array size must be 2, 3, or 4 ) -OR-
          ( The vertex array type must be SHORT -AND-
            The vertex array stride must be a multiple of 4 bytes -AND-
            The vertex array pointer must be 4-byte aligned. -AND-
            The vertex array size must be 2 ) -OR-
          ( The vertex array type must be SHORT -AND-
            The vertex array stride must be a multiple of 8 bytes -AND-
            The vertex array pointer must be 8-byte aligned. -AND-
            The vertex array size must be 4 ) -OR-
          ( The vertex array type must be SHORT -AND-
            The vertex array stride must be a multiple of 8 bytes -AND-
            The vertex array pointer must be 8-byte aligned. )
            The vertex array stride must non-zero -AND-
            The vertex array size must be 3 ) ) )

  2.  ( NORMAL_ARRAY must be disabled. ) -OR -
      ( NORMAL_ARRAY must be enabled -AND-
        The normal array size must be 3 -AND-
        The normal array stride must be less than 256 -AND-
        ( ( The normal array type must be FLOAT -AND-
            The normal array stride must be a multiple of 4 bytes -AND-
            The normal array pointer must be 4-byte aligned. ) -OR-
          ( The normal array type must be SHORT -AND-
            The normal array stride must be a multiple of 8 bytes -AND-
            The normal array stride must non-zero -AND-
            The normal array pointer must be 8-byte aligned. ) ) )

  3.  ( COLOR_ARRAY must be disabled. ) -OR -
      ( COLOR_ARRAY must be enabled -AND-
        The color array type must be FLOAT or UNSIGNED_BYTE -AND-
        The color array stride must be a multiple of 4 bytes -AND-
        The color array stride must be less than 256 -AND-
        The color array pointer must be 4-byte aligned -AND-
        ( ( The color array size must be 3 -AND-
            The color array stride must non-zero ) -OR-
          ( The color array size must be 4 ) )

  4.  ( SECONDARY_COLOR_ARRAY must be disabled. ) -OR -
      ( SECONDARY_COLOR_ARRAY must be enabled -AND-
        The secondary color array type must be FLOAT or UNSIGNED_BYTE -AND-
        The secondary color array stride must be a multiple of 4 bytes -AND-
        The secondary color array stride must be less than 256 -AND-
        The secondary color array pointer must be 4-byte aligned -AND-
        ( ( The secondary color array size must be 3 -AND-
            The secondary color array stride must non-zero ) -OR-
          ( The secondary color array size must be 4 ) )

  5.  For texture units zero and one:
```

```
      ( TEXTURE_COORD_ARRAY must be disabled. ) -OR -
      ( TEXTURE_COORD_ARRAY must be enabled -AND-
        The texture coord array stride must be less than 256 -AND-
        ( ( The texture coord array type must be FLOAT -AND-
            The texture coord array pointer must be 4-byte aligned. )
            The texture coord array stride must be a multiple of 4 bytes -AND-
            The texture coord array size must be 1, 2, 3, or 4 ) -OR-
          ( The texture coord array type must be SHORT -AND-
            The texture coord array pointer must be 4-byte aligned. )
            The texture coord array stride must be a multiple of 4 bytes -AND-
            The texture coord array stride must non-zero -AND-
            The texture coord array size must be 1 ) -OR-
          ( The texture coord array type must be SHORT -AND-
            The texture coord array pointer must be 4-byte aligned. )
            The texture coord array stride must be a multiple of 4 bytes -AND-
            The texture coord array size must be 2 ) -OR-
          ( The texture coord array type must be SHORT -AND-
            The texture coord array pointer must be 8-byte aligned. )
            The texture coord array stride must be a multiple of 8 bytes -AND-
            The texture coord array stride must non-zero -AND-
            The texture coord array size must be 3 ) -OR-
          ( The texture coord array type must be SHORT -AND-
            The texture coord array pointer must be 8-byte aligned. )
            The texture coord array stride must be a multiple of 8 bytes -AND-
            The texture coord array size must be 4 ) ) )

    6.  ( EDGE_FLAG_ARRAY must be disabled. )

    7.  ( VERTEX_WEIGHT_ARRAY_NV must be disabled. ) -OR -
        ( VERTEX_WEIGHT_ARRAY_NV must be enabled. -AND -
          The vertex weight array type must be FLOAT -AND-
          The vertex weight array size must be 1 -AND-
          The vertex weight array stride must be a multiple of 4 bytes -AND-
          The vertex weight array stride must be less than 256 -AND-
          The vertex weight array pointer must be 4-byte aligned )

    8.  ( FOG_COORDINATE_ARRAY must be disabled. ) -OR -
        ( FOG_COORDINATE_ARRAY must be enabled -AND-
          The chip in use must be an NV11 or NV15, not NV10 -AND-
          The fog coordinate array type must be FLOAT -AND-
          The fog coordinate array size must be 1 -AND-
          The fog coordinate array stride must be a multiple of 4 bytes -AND-
          The fog coordinate array stride must be less than 256 -AND-
          The fog coordinate array pointer must be 4-byte aligned )

o  For the NV10 the implementation-dependent validity check based on
   other OpenGL rendering state is FALSE if any of the following are true:

    1.  ( COLOR_LOGIC_OP is enabled -AND-
          The logic op is not COPY ), except in the case of Quadro2
        (Quadro2 Pro, Quadro2 MXR) products.

    2.  ( LIGHT_MODEL_TWO_SIDE is true. )

    3.  Either texture unit is enabled and active with a texture
        with a non-zero border.

    4.  VERTEX_PROGRAM_NV is enabled.

    5.  Several other obscure unspecified reasons.
```

NV20 Implementation Details

This section describes implementation-defined limits for NV20:

The value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV is 1048575.

This section describes the implementation-dependent validity checks for NV20.

- For the NV20 implementation-dependent validity check for the vertex array range region of memory to be true, all of the following must be true:

  1. The <pointer> must be 32-byte aligned.

  2. The underlying memory types must all be the same (all standard system memory -OR- all AGP memory -OR- all video memory).

- To determine whether the NV20 implementation-dependent validity check for the vertex array state is true, the following algorithm is used:

  The currently enabled arrays and their pointers, strides, and types are first determined using the value of VERTEX_PROGRAM_NV. If VERTEX_PROGRAM_NV is disabled, the standard GL vertex arrays are used. If VERTEX_PROGRAM_NV is enabled, the vertex attribute arrays take precedence over the standard vertex arrays. The following table, taken from the NV_vertex_program specification, shows the aliasing between the standard and attribute arrays:

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

For the validity check to be TRUE, the following must all be true:

1. Vertex attribute 0's array must be enabled.

2. EDGE_FLAG_ARRAY must be disabled.
3. For all enabled arrays, all of the following must be true:
   - the pointer must be 4-byte aligned
   - the stride must be less than 256
   - the stride must be a multiple of 4
   - the type must be FLOAT, SHORT, or UNSIGNED_BYTE

- For the NV20 the implementation-dependent validity check based on other OpenGL rendering state is FALSE only for a few obscure and unspecified reasons.

Revision History

January 10, 2001 - Added NV20 implementation details. Made several corrections to the NV10 implementation details. Specifically, noted that on the NV11 and NV15 architectures, the fog coordinate array may be used, and updated the section on other state that may cause the vertex array range to be invalid. Only drivers built after this date will support fog coordinate arrays on NV11 and NV15. Also fixed a few typos in the spec.

Name

NV_vertex_array_range2

Name Strings

GL_NV_vertex_array_range2

Notice

Copyright NVIDIA Corporation, 2001.

IP Status

NVIDIA Proprietary.

Status

Complete

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_array_range2.txt#2 $

Number

232

Dependencies

Assumes support for the NV_vertex_array_range extension (version 1.1).

Support for NV_fence is recommended but not required.

Overview

Enabling and disabling the vertex array range is specified by the original NV_vertex_array_range extension specification to flush the vertex array range implicitly. In retrospect, this semantic is extremely misconceived and creates terrible performance problems for any application that wishes to mix conventional vertex arrays with vertex arrange range-enabled vertex arrays.

This extension provides a new token for enabling/disabling the vertex array range that does NOT perform an implicit vertex array range flush when the enable/disable is performed.

Issues

Should this extension expose a new enable that enables/disables the vertex array range enable/disable semantic of performing an implicit 'vertex array range flush' when GL_VERTEX_ARRAY_RANGE_NV is enabled or disabled, OR should it add a new enable token that acts identically to GL_VERTEX_ARRAY_RANGE_NV without the implicit flush?

RESOLUTION: The second option. Enabling/disabling GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV acts identically to enabling/disabling GL_VERTEX_ARRAY_RANGE_NV, just without the implicit flush.

Should GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV work with glIsEnabled?

RESOLUTION: NO. There is still just a single state boolean to query.

New Procedures and Functions

None

New Tokens

Accepted by the <cap> parameter of EnableClientState, DisableClientState:

VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV 0x8533

Additions to Chapter 2 of the OpenGL 1.1 Specification (OpenGL Operation)

Within the discussion of vertex arrays (Section 2.8) amended by the NV_vertex_array_range extension specification, change the discussion of enabling the vertex array range to:

The vertex array range is enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_NV.

The vertex array range is also enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV. This second means to enable and disable the vertex array range does not perform an implicit vertex array range flush as described subsequently."

Within the discussion of vertex arrays (Section 2.8) amended by the NV_vertex_array_range extension specification, change the discussion of implicit vertex array range flushes to:

"A 'vertex array range flush' occurs when one of the following operations occur:

- Finish returns.
- FlushVertexArrayRangeNV returns.
- VertexArrayRangeNV returns.

- DisableClientState of VERTEX_ARRAY_RANGE_NV returns.
- EnableClientState of VERTEX_ARRAY_RANGE_NV returns.
- Another OpenGL context is made current.

However, use of VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV with DisableClientState or EnableClientState does NOT perform an implicit vertex array range flush."

Additions to Chapter 5 of the OpenGL 1.1 Specification (Special Functions)

None

Additions to the WGL interface:

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

No new errors.

New State

None

New Implementation Dependent State

None

Revision History

4/13/2001 - token value for GL_VERTEX_ARRAY_RANGE_WITHOUT_FLUSH_NV should be 0x8533 (was incorrectly typed as 0x8503)

Name

NV_vertex_program

Name Strings

GL_NV_vertex_program

Notice

Copyright NVIDIA Corporation, 2000, 2001.

IP Status

NVIDIA Proprietary.

Status

Version 1.3

Version

NVIDIA Date: April 13, 2001
$Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_program.txt#5 $

Number

233

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification and requires OpenGL 1.2.1.

Requires support for the ARB_multitexture extension with at least two texture units.

EXT_point_parameters affects the definition of this extension.

EXT_secondary_color affects the definition of this extension.

EXT_fog_coord affects the definition of this extension.

EXT_vertex_weighting affects the definition of this extension.

ARB_imaging affects the definition of this extension.

Overview

Unextended OpenGL mandates a certain set of configurable per-vertex computations defining vertex transformation, texture coordinate generation and transformation, and lighting. Several extensions have added further per-vertex computations to OpenGL. For example, extensions have defined new texture coordinate generation modes (ARB_texture_cube_map, NV_texgen_reflection, NV_texgen_emboss), new vertex transformation modes (EXT_vertex_weighting), new lighting modes (OpenGL 1.2's separate specular and rescale normal functionality),

several modes for fog distance generation (NV_fog_distance), and
eye-distance point size attenuation (EXT_point_parameters).

Each such extension adds a small set of relatively inflexible
per-vertex computations.

This inflexibility is in contrast to the typical flexibility provided
by the underlying programmable floating point engines (whether
micro-coded vertex engines, DSPs, or CPUs) that are traditionally used
to implement OpenGL's per-vertex computations. The purpose of this
extension is to expose to the OpenGL application writer a significant
degree of per-vertex programmability for computing vertex parameters.

For the purposes of discussing this extension, a vertex program is
a sequence of floating-point 4-component vector operations that
determines how a set of program parameters (defined outside of
OpenGL's begin/end pair) and an input set of per-vertex parameters
are transformed to a set of per-vertex output parameters.

The per-vertex computations for standard OpenGL given a particular
set of lighting and texture coordinate generation modes (along with
any state for extensions defining per-vertex computations) is, in
essence, a vertex program. However, the sequence of operations is
defined implicitly by the current OpenGL state settings rather than
defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining vertex
program instruction sequences for application-defined vertex programs.
In order to define such vertex programs, this extension defines
a vertex programming model including a floating-point 4-component
vector instruction set and a relatively large set of floating-point
4-component registers.

The extension's vertex programming model is designed for efficient
hardware implementation and to support a wide variety of vertex
programs. By design, the entire set of existing vertex programs
defined by existing OpenGL per-vertex computation extensions can be
implemented using the extension's vertex programming model.

Issues

What should this extension be called?

RESOLUTION: NV_vertex_program. DirectX 8 refers to its similar
functionality as "vertex shaders". This is a confusing term
because shaders are usually assumed to operate at the fragment or
pixel level, not the vertex level.

Conceptually, what the extension defines is an application-defined
program (admittedly limited by its sequential execution model) for
processing vertices so the "vertex program" term is more accurate.

Additionally, some of the API machinery in this extension for
describing programs could be useful for extending other OpenGL
operations with programs (though other types of programs would
likely look very different from vertex programs).

What terms are important to this specification?

vertex program mode - when vertex program mode is enabled, vertices are transformed by an application-defined vertex program.

conventional GL vertex transform mode - when vertex program mode is disabled (or the extension is not supported), vertices are transformed by GL's conventional texgen, lighting, and transform state.

provoke - the verb that denotes the beginning of vertex transformation by either vertex program mode or conventional GL vertex transform mode. Vertices are provoked when either glVertex or glVertexAttribNV(0, ...) is called.

program target - a type or class of program. This extension supports two program targets: the vertex program and the vertex state program. Future extensions could add other program targets.

vertex program - an application-defined vertex program used to transform vertices when vertex program mode is enabled.

vertex state program - a program similar to a vertex program. Unlike a vertex program, a vertex state program runs outside of a glBegin/glEnd pair. Vertex state programs do not transform a vertex. They just update program parameters.

vertex attribute - one of 16 4-component per-vertex parameters defined by this extension. These attributes alias with the conventional per-vertex parameters.

per-vertex parameter - a vertex attribute or a conventional per-vertex parameter such as set by glNormal3f or glColor3f.

program parameter - one of 96 4-component registers available to vertex programs. The state of these registers is shared among all vertex programs.

What part of OpenGL do vertex programs specifically bypass?

Vertex programs bypass the following OpenGL functionality:

- Normal transformation and normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation
- The texture matrix
- The normalization of AUTO_NORMAL evaluated normals
- The modelview and projection matrix transforms
- The per-vertex processing in EXT_point_parameters
- The per-vertex processing in NV_fog_distance
- Raster position transformation
- Client-defined clip planes

Operations not subsumed by vertex programs

- The view frustum clip
- Perspective divide (division by w)
- The viewport transformation
- The depth range transformation
- Clamping the primary and secondary color to [0,1]
- Primitive assembly and subsequent operations
- Evaluator (except the AUTO_NORMAL normalization)

How specific should this specification be about precision?

RESOLUTION: Reasonable precision requirements are incorporated into the specification beyond the often vague requirements of the core OpenGL specification.

This extension essentially defines an instruction set and its corresponding execution environment. The instruction set specified may find applications beyond the traditional purposes of 3D vertex transformation, lighting, and texture coordinate generation that have fairly lax precision requirements. To facilitate such possibly unexpected applications of this functionality, minimum precision requirements are specified.

The minimum precision requirements in the specification are meant to serve as a baseline so that application developers can write vertex programs with minimal worries about precision issues.

What about when the "execution environment" involves support for other extensions?

This extension assumes support for functionality that includes a fog distance, secondary color, point parameters, and multiple texture coordinates.

There is a trade-off between requiring support for these extensions to guarantee a particular extended execution environment and requiring lots of functionality that everyone might not support.

Application developers will desire a high baseline of functionality so that OpenGL applications using vertex programs can work in the full context of OpenGL. But if too much is required, the implementation burden mandated by the extension may limit the number of available implementations.

Clearly we do not want to require support for 8 texture units even if the machinery is there for it. Still multitexture is a common and important feature for using vertex programs effectively. Requiring at least two texture units seems reasonable.

What do we say about the alpha component of the secondary color?

RESOLUTION: When vertex program mode is enabled, the alpha component of csec used for the color sum state is assumed always zero. Another downstream extension may actually make the alpha component written into the COL1 (or BFC1) vertex result register available.

Should client-defined clip planes operate when vertex program mode is enabled?

RESOLUTION. No.

OpenGL's client-defined clip planes are specified in eye-space. Vertex programs generate homogeneous clip space positions. Unlike the conventional OpenGL vertex transformation mode, vertex program mode requires no semantic equivalent to eye-space.

Applications that require client-defined clip planes can simulate OpenGL-style client-defined clip planes by generating texture coordinates and using alpha testing or other per-fragment tests such as NV_texture_shader's CULL_FRAGMENT_NV program to discard fragments. In many ways, these schemes provide a more flexible mechanism for clipping than client-defined clip planes.

Unfortunately, vertex programs used in conjunction with selection or feedback will not have a means to support client-defined clip planes because the per-fragment culling mechanisms described in the previous paragraph are not available in the selection or feedback render modes. Oh well.

Finally, as a practical concern, client-defined clip planes greatly complicate clipping for various hardware rasterization architectures.

How are edge flags handled?

RESOLUTION: Passed through without the ability to be modified by a vertex program. Applications are free to send edge flags when vertex program mode is enabled.

Should vertex attributes alias with conventional per-vertex parameters?

RESOLUTION. YES.

This aliasing should make it easy to use vertex programs with existing OpenGL code that transfers per-vertex parameters using conventional OpenGL per-vertex calls.

It also minimizes the number of per-vertex parameters that the hardware must maintain.

See Table X.2 for the aliasing of vertex attributes and conventional per-vertex parameters.

How should vertex attribute arrays interact with conventional vertex arrays?

RESOLUTION: When vertex program mode is enabled, a particular vertex attribute array will be used if enabled, but if disabled, and the corresponding aliased conventional vertex array is enabled (assuming that there is a corresponding aliased conventional vertex array for the particular vertex array), the conventional vertex array will be used.

This matches the way immediate mode per-vertex parameter aliasing works.

This does slightly complicate vertex array validation in program mode, but programmers using vertex arrays can simply enable vertex program mode without reconfiguring their conventional vertex arrays and get what they expect.

Note that this does create an asymmetry between immediate mode and vertex arrays depending on whether vertex program mode is enabled or not. The immediate mode vertex attribute commands operate unchanged whether vertex program mode is enabled or not. However the vertex attribute vertex arrays are used only when vertex program mode is enabled.

Supporting vertex attribute vertex arrays when vertex program mode is disabled would create a large implementation burden for existing OpenGL implementations that have heavily optimized conventional vertex arrays. For example, the normal array can be assumed to always contain 3 and only 3 components in conventional OpenGL vertex transform mode, but may contain 1, 2, 3, or 4 components in vertex program mode.

There is not any additional functionality gained by supporting vertex attribute arrays when vertex program mode is disabled, but there is lots of implementation overhead. In any case, it does not seem something worth encouraging so it is simply not supported. So vertex attribute arrays are IGNORED when vertex program mode is not enabled.

Ignoring VertexAttribute commands or treating VertexAttribute commands as an error when vertex program mode is enabled would likely add overhead for such a conditional check. The implementation overhead for supporting VertexAttribute commands when vertex program mode is disabled is not that significant. Additionally, it is likely that setting persistent vertex attribute state while vertex program mode is disabled may be useful to applications. So vertex attribute immediate mode commands are PERMITTED when vertex program mode is not enabled.

Colors and normals specified as ints, uints, shorts, ushorts, bytes, and ubytes are converted to floating-point ranges when supplied to core OpenGL as described in Table 2.6. Other per-vertex attributes such as texture coordinates and positions are not converted. How does this mix with vertex programs where all vertex attributes are supposedly treated identically?

RESOLUTION: Vertex attributes specified as bytes and ubytes are always converted as described in Table 2.6. All other formats are not converted according to Table 2.6 but simply converted directly to floating-point.

The ubyte type is converted because those types seem more useful for passing colors in the [0,1] range.

If an application desires a conversion, the conversion can be incorporated into the vertex program itself.

This also applies to vertex attribute arrays. However, by enabling a color or normal vertex array and not enabling the corresponding aliased vertex attribute array, programmers can get the conventional conversions for color and normal arrays (but only for the vertex attribute arrays that alias to the conventional color and normal arrays and only with the sizes/types supported by these color and normal arrays).

Should programs be C-style null-terminated strings?

RESOLUTION: No. Programs should be specified as an array of GLubyte with an explicit length parameter. OpenGL has no precedent for passing null-terminated strings into the API (though glGetString returns null-terminated strings). Null-terminated strings are problematic for some languages.

Should all existing OpenGL transform functionality and extensions be implementable as vertex programs?

RESOLUTION: Yes. Vertex programs should be a complete superset of what you can do with OpenGL 1.2 and existing vertex transform

extensions.

To implement EXT_point_parameters, the GL_VERTEX_PROGRAM_POINT_SIZE_NV enable is introduced.

To implement two-sided lighting, the GL_VERTEX_PROGRAM_TWO_SIDE_NV enable is introduced.

How does glPointSize work with vertex programs?

RESOLUTION: If GL_VERTEX_PROGRAM_POINT_SIZE_NV is disabled, the size of points is determine by the glPointSize state. If enabled, the point size is determined per-vertex by the clamped value of the vertex result PSIZ register.

Can the currently bound vertex program id be deleted or reloaded?

RESOLUTION. Yes. When a vertex program id is deleted or reloaded when it is the currently bound vertex program, it is as if a rebind occurs after the deletion or reload.

In the case of a reload, the new vertex program will be used from then on. In the case of a deletion, the current vertex program will be treated as if it is nonexistent.

Should program objects have a mechanism for managing program residency?

RESOLUTION: Yes. Vertex program instruction memory is a limited hardware resource. glBindProgramNV will be faster if binding to a resident program. Applications are likely to want to quickly switch between a small collection of programs.

glAreProgramsResidentNV allows the residency status of a group of programs to be queried. This mimics glAreTexturesResident.

Instead of adopting the glPrioritizeTextures mechanism, a new glRequestResidentProgramsNV command is specified instead. Assigning priorities to textures has always been a problematic endeavor and few OpenGL implementations implemented it effectively. For the priority mechanism to work well, it requires the client to routinely update the priorities of textures.

The glRequestResidentProgramsNV indicates to the GL that a set of programs are intended for use together. Because all the programs are requesting residency as a group, drivers should be able to attempt to load all the requested programs at once (and remove from residency programs not in the group if necessary). Clients can use glAreProgramsResidentNV to query the relative success of the request.

glRequestResidentProgramsNV should be superior to loading programs on-demand because fragmentation can be avoided.

What happens when you execute a nonexistent or invalid program?

RESOLUTION: glBegin will fail with a GL_INVALID_OPERATION if the currently bound vertex program is nonexistent or invalid. The same applies to glRasterPos and any command that implies a glBegin.

Because the glVertex and glVertexAttribNV(0, ...) are ignored outside of a glBegin/glEnd pair (without generating an error) it is impossible to provoke a vertex program if the current vertex program is nonexistent or invalid. Other per-vertex parameters (for examples those set by glColor, glNormal, and glVertexAttribNV when the attribute number is not zero) are recorded since they are legal outside of a glBegin/glEnd.

For vertex state programs, the problem is simpler because glExecuteProgramNV can immediately fail with a GL_INVALID_OPERATION when the named vertex state program is nonexistent or invalid.

What happens when a matrix has been tracked into a set of program parameters, but then glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, addr, GL_NONE, GL_IDENTITY_NV) is performed?

RESOLUTION: The specified program parameters stop tracking a matrix, but they retain the values of the matrix they were last tracking.

Can rows of tracked matrices be queried by querying the program parameters that track them?

RESOLUTION: Yes.

Discussing matrices is confusing because of row-major versus column-major issues. Can you give an example of how a matrix is tracked?

```
GLfloat matrix[16] = { 1, 5, 9,  13,
                       2, 6, 10, 14,
                       3, 7, 11, 15,
                       4, 8, 12, 16 };
GLfloat row1[4], row2[4];

glMatrixMode(GL_MATRIX0_NV);
glLoadMatrixf(matrix);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MATRIX0_NV, GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_MATRIX0_NV, GL_TRANSPOSE_NV);
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 5,
  GL_PROGRAM_PARAMETER_NV, row1);
/* row1 is now [ 2 6 10 14 ] */
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 9,
  GL_PROGRAM_PARAMETER_NV, row2);
/* row2 is now [ 5 6 7 8 ] because the tracked matrix is transposed */
```

Should evaluators be extended to evaluate arbitrary vertex attributes?

RESOLUTION: Yes. We'll support 32 new maps (16 for MAP1 and 16 for MAP2) that take priority over the conventional maps that they

might alias to (only when vertex program mode is enabled).

These new maps always evaluate all four components. The rationale for this is that if we supported 1, 2, 3, or 4 components, that would add 128 (16*4*2) enumerants which is too many. In addition, if you wanted to evaluate two 2-component vertex attributes, you could instead generate one 4-component vertex attribute and use the vertex program with swizzling to treat this as two-components.

Moreover, we are assuming 4-component vector instructions so less than 4-component evaluations might not be any more efficient than 4-component evaluations. Implementations that use vector instructions such as Intel's SSE instructions will be easier to implement since they can focus on optimizing just the 4-component case.

How should GL_AUTO_NORMAL work with vertex programs?

RESOLUTION: GL_AUTO_NORMAL should NOT guarantee that the generated analytical normal be normalized. In vertex program mode, the current vertex program can easily normalize the normal if required.

This can lead to greater efficiency if the vertex program transforms the normal to another coordinate system such as eye-space with a transform that preserves vector length. Then a single normalize after transform is more efficient than normalizing after evaluation and also normalizing after transform.

Conceptually, the normalize mandated for AUTO_NORMAL in section 5.1 is just one of the many transformation operations subsumed by vertex programs.

Should the new vertex program related enables push/pop with GL_ENABLE_BIT?

RESOLUTION: Yes. Pushing and popping enable bits is easy. This includes the 32 new evaluator map enable bits. These evaluator enable bits are also pushed and popped using GL_EVAL_BIT.

Should all the vertex attribute state push/pop with GL_CURRENT_BIT?

RESOLUTION: Yes. The state is aliased with the conventional per-vertex parameter state so it really should push/pop.

Should all the vertex attrib vertex array state push/pop with GL_CLIENT_VERTEX_ARRAY_BIT?

RESOLUTION: Yes.

Should all the other vertex program-related state push/pop somehow?

RESOLUTION: No.

The other vertex program doesn't fit well with the existing bits. To be clear, GL_ALL_ATTRIB_BITS does not push/pop vertex program state other than enables.

Should we generate a GL_INVALID_OPERATION operation if updating a vertex attribute greater than 15?

RESOLUTION: Yes.

The other option would be to mask or modulo the vertex attribute index with 16. This is cheap, but it would make it difficult to increase the number of vertex attributes in the future.

If we check for the error, it should be a well predicted branch for immediate mode calls. For vertex arrays, the check is only required at vertex array specification time.

Hopefully this will encourage people to use vertex arrays over immediate mode.

Should writes to program parameter registers during a vertex program be supported?

RESOLUTION. No.

Writes to program parameter registers from within a vertex program would require the execution of vertex programs to be serialized with respect to each other. This would create an unwarranted implementation penalty for parallel vertex program execution implementations.

However vertex state programs may write to program parameter registers (that is the whole point of vertex state programs).

Should we support variously sized immediate mode byte and ubyte commands? How about for vertex arrays?

RESOLUTION. Only support the 4ub mode.

There are simply too many glVertexAttribNV routines. Passing less than 4 bytes at a time is inefficient. We expect the main use for bytes to be for colors where these will be unsigned bytes. So let's just support 4ub mode for bytes. This applies to vertex arrays too.

Should we support integer, unsigned integer, and unsigned short formats for vertex attributes?

RESOLUTION: No. It's just too many immediate mode entry points, most of which are not that useful. Signed shorts are supported however. We expect signed shorts to be useful for passing compact texture coordinates.

Should we support doubles for vertex attributes?

RESOLUTION: Yes. Some implementation of the extension might support double precision. Lots of math routines output double precision.

Should there be a way to determine where in a loaded program string the first parse error occurs?

RESOLUTION: Yes. You can query PROGRAM_ERROR_POSITION_NV.

Should program objects be shared among rendering contexts in the same manner as display lists and texture objects?

RESOLUTION: Yes.

How should this extension interact with color material?

RESOLUTION: It should not. Color material is a conventional OpenGL vertex transform mode. It does not have a place for vertex programs. If you want to emulate color material with vertex programs, you would simply write a program where the material parameters feed from the color vertex attribute.

Should there be a glMatrixMode or glActiveTextureARB style selector for vertex attributes?

RESOLUTION: No. While this would let us reduce a lot of enumerants down, it would make programming a hassle in lots of cases. Consider having to change the vertex attribute mode to enable a set of vertex arrays.

How should gets for vertex attribute array pointers?

RESOLUTION: Add new get commands. Using the existing calls would require adding 4 sets of 16 enumerants stride, type, size, and pointer. That's too many gets.

Instead add glGetVertexAttribNV and glGetVertexAttribPointervNV. glGetVertexAttribNV is also useful for querying the current vertex attribute.

glGet and glGetPointerv will not return vertex attribute array pointers.

Why is the address register numbered and why is it a vector register?

In the future, A0.y and A0.z and A0.w may exist. For this extension, only A0.x is useful. Also in the future, there may be more than one address register.

There's a nice consistency in thinking about all the registers as 4-component vectors even if the address register has only one usable component.

Should vertex programs and vertex state programs be required to have a header token and an end token?

RESOLUTION: Yes.

The "!!VP1.0" and "!!VSP1.0" tokens start vertex programs and vertex state programs respectively. Both types of programs must

end with the "END" token.

The initial header token reminds the programmer what type of program they are writing. If vertex programs and vertex state programs are ever read from disk files, the header token can serve as a magic number for identifying vertex programs and vertex state programs.

The target type for vertex programs and vertex state programs can be distinguished based on their respective grammars independent of the initial header tokens, but the initial header tokens will make it easier for programmers to distinguish the two program target types.

We expect programs to often be generated by concatenation of program fragments. The "END" token will hopefully reduce bugs due to specifying an incorrectly concatenated program.

It's tempting to make these additional header and end tokens optional, but if there is a sanity check value in header and end tokens, that value is undermined if the tokens are optional.

What should be said about rendering invariances?

RESOLUTION: See the Appendix A additions below.

The justification for the two rules cited is to support multi-pass rendering when using vertex programs. Different rendering passes will likely use different programs so there must be some means of guaranteeing that two different programs can generate particular identical vertex results between different passes.

In practice, this does limit the type of vertex program implementations that are possible.

For example, consider a limited hardware implementation of vertex programs that uses a different floating-point implementation than the CPU's floating-point implementation. If the limited hardware implementation can only run small vertex programs (say the hardware provides on 4 temporary registers instead of the required 12), the implementation is incorrect and non-conformant if programs that only require 4 temporary registers use the vertex program hardware, but programs that require more than 4 temporary registers are implemented by the CPU.

This is a very important practical requirement. Consider a multi-pass rendering algorithm where one pass uses a vertex program that uses only 4 temporary registers, but a different pass uses a vertex program that uses 5 temporary registers. If two programs have instruction sequences that given the same input state compute identical resulting vertex positions, the multi-pass algorithm should generate identically positioned primitives for each pass. But given the non-conformant vertex program implementation described above, this could not be guaranteed.

This does not mean that schemes for splitting vertex program implementations between dedicated hardware and CPUs are impossible. If the CPU and dedicated vertex program hardware used IDENTICAL floating-point implementations and therefore generated exactly

identical results, the above described could work.

While these invariance rules are vital for vertex programs operating correctly for multi-pass algorithms, there is no requirement that conventional OpenGL vertex transform mode will be invariant with vertex program mode. A multi-pass algorithm should not assume that one pass using vertex program mode and another pass using conventional GL vertex transform mode will generate identically positioned primitives.

Consider that while the conventional OpenGL vertex program mode is repeatable with itself, the exact procedure used to transform vertices is not specified nor is the procedure's precision specified. The GL specification indicates that vertex coordinates are transformed by the modelview matrix and then transformed by the projection matrix. Some implementations may perform this sequence of transformations exactly, but other implementations may transform vertex coordinates by the composite of the modelview and projection matrices (one matrix transform instead of two matrix transforms in sequence). Given this implementation flexibility, there is no way for a vertex program author to exactly duplicate the precise computations used by the conventional OpenGL vertex transform mode.

The guidance to OpenGL application programs is clear. If you are going to implement multi-pass rendering algorithms that require certain invariances between the multiple passes, choose either vertex program mode or the conventional OpenGL vertex transform mode for your rendering passes, but do not mix the two modes.

What range of relative addressing offsets should be allowed?

RESOLUTION: -64 to 63.

Negative offsets are useful for accessing a table centered at zero without extra bias instructions. Having the offsets support much larger magnitudes just seems to increase the required instruction widths. The -64 to 63 range seems like a reasonable compromise.

When EXT_secondary_color is supported, how does the GL_COLOR_SUM_EXT enable affect vertex program mode?

RESOLUTION: The GL_COLOR_SUM_EXT enable has no affect when vertex program mode is enabled.

When vertex program mode is enabled, the color sum operation is always in operation. A program can "avoid" the color sum operation by not writing the COL1 (or BFC1 when GL_VERTEX_PROGRAM_TWO_SIDE_NV) vertex result registers because the default values of all vertex result registers is (0,0,0,1). For the color sum operation, the alpha value is always assumed zero. So by not writing the secondary color vertex result registers, the program assures that zero is added as part of the color sum operation.

If there is a cost to the color sum operation, OpenGL implementations may be smart enough to determine at program bind time whether a secondary color vertex result is generated and implicitly disable the color sum operation.

Why must RCP of 1.0 always be 1.0?

This is important for 3D graphics so that non-projective textures and orthogonal projections work as expected. Basically when q or w is 1.0, things should work as expected.

Stronger requirements such as "RCP of -1.0 must always be -1.0" are encouraged, but there is no compelling reason to state such requirements explicitly as is the case for "RCP of 1.0 must always be 1.0".

What happens when the source scalar value for the ARL instruction is an extremely positive or extremely negative floating-point value? Is there a problem mapping the value to a constrained integer range?

RESOLUTION: It is not a problem. Relative addressing can by offset by a limited range of offsets (-64 to 63). Relative addressing that falls outside of the 0 to 95 range of program parameter registers is automatically mapped to (0,0,0,0).

Clamping the source scalar value for ARL to the range -64 to 160 inclusive is sufficient to ensure that relative addressing is out of range.

How do you perform a 3-component normalize in three instructions?

```
#
# R1 = (nx,ny,nz)
#
# R0.xyz = normalize(R1)
# R0.w   = 1/sqrt(nx*nx + ny*ny + nz*nz)
#
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, R1, R0.w;
```

How do you perform a 3-component cross product in two instructions?

```
#
# Cross product |  i     j     k   | into R2.
#               | R0.x  R0.y  R0.z |
#               | R1.x  R1.y  R1.z |
#
MUL R2, R0.zxyw, R1.yzxw;
MAD R2, R0.yzxw, R1.zxyw, -R2;
```

How do you perform a 4-component vector absolute value in one instruction?

```
#
# Absolute value is the maximum of the negative and positive
# components of a vector.
#
# R1 = abs(R0)
#
MAX R1, R0, -R0;
```

How do you compute the determinant of a 3x3 matrix in three instructions?

```
  #
  # Determinant of | R0.x  R0.y  R0.z | into R3
  #                | R1.x  R1.y  R1.z |
  #                | R2.x  R2.y  R2.z |
  #
  MUL R3, R1.zxyw, R2.yzxw;
  MAD R3, R1.yzxw, R2.zxyw, -R3;
  DP3 R3, R0, R3;
```

How do you transform a vertex position by a 4x4 matrix and then perform a homogeneous divide?

```
  #
  # c[20] = modelview row 0
  # c[21] = modelview row 1
  # c[22] = modelview row 2
  # c[23] = modelview row 3
  #
  # result = R5
  #
  DP4 R5.w, v[OPOS], C[23];
  DP4 R5.x, v[OPOS], C[20];
  DP4 R5.y, v[OPOS], C[21];
  DP4 R5.z, v[OPOS], C[22];
  RCP R11, R5.w;
  MUL R5,R5,R11;
```

How do you perform a vector weighting of two vectors using a single weight?

```
  #
  # c[45] = (1.0, 1.0, 1.0, 1.0)
  #
  # R2        = vector 0
  # R3        = vector 1
  # v[WGHT].x = scalar weight to blend vectors 0 and 1
  # result    = R4 * v[WGHT].x + R4 * (1-v[WGHT])
  #
  ADD R11, -v[WGHT].x, c[45];   # compute (1-v[WGHT])
  MUL R4, R3, R11;
  MAD R4, v[WGHT].x, R3, R4
```

*How do you reduce a value to some fundamental period such as 2*PI?*

```
  #
  # C[36] = (1.0/(2*PI), 2*PI, 0.0, 0.0)
  #
  # R1.x = input value
  # R2   = result
  #
  MUL R0, R1, c[36].x;
  EXP R4, R0.x;
  MUL R2, R4.y, c[36].y;
```

How do you implement a simple specular and diffuse lighting computation with an eye-space normal?

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[4-7]  = modelview inverse transpose
# c[32]   = normalized eye-space light direction (infinite light)
# c[33]   = normalized constant eye-space half-angle vector (infinite viewer)
# c[35].x = pre-multiplied monochromatic diffuse light color & diffuse material
# c[35].y = pre-multiplied monochromatic ambient light color & diffuse material
# c[36]   = specular color
# c[38].x = specular power
#
# outputs homogenous position and color
#
DP4   o[HPOS].x, c[0], v[OPOS];
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];          # R0 = n' = transformed normal
DP3   R1.x, c[32], R0;              # R1.x = Lpos DOT n'
DP3   R1.y, c[33], R0;              # R1.y = hHat DOT n'
MOV   R1.w, c[38].x;                # R1.w = specular power
LIT   R2, R1;                       # Compute lighting values
MAD   R3, c[35].x, R2.y, c[35].y;   # diffuse + emissive
MAD   o[COL0].xyz, c[36], R2.z, R3; # + specular
END
```

Can you perturb transformed vertex positions with a vertex program?

Yes. Here is an example that performs an object-space diffuse lighting computations and perturbs the vertex position based on this lighting result. Do not take this example too seriously.

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[32]   = normalized light direction in object-space
# c[35]   = yellow diffuse material, (1.0, 1.0, 0.0, 1.0)
# c[64].x = 0.0
# c[64].z = 0.125, a scaling factor
#
# outputs diffuse illumination for color and perturbed position
#
DP3   R0, c[32], v[NRML];     # light direction DOT normal
MUL   o[COL0].xyz, R0, c[35];
MAX   R0, c[64].x, R0;
MUL   R0, R0, v[NRML];
MUL   R0, R0, c[64].z;
ADD   R1, v[OPOS], -R0;       # perturb object space position
DP4   o[HPOS].x, c[0], R1;
DP4   o[HPOS].y, c[1], R1;
DP4   o[HPOS].z, c[2], R1;
DP4   o[HPOS].w, c[3], R1;
END
```

What if more exponential precision is needed than provided by the builtin EXP instruction?

A sequence of vertex program instructions can be used refine the initial EXP approximation. The pseudo-macro below shows an example of how to refine the EXP approximation.

The psuedo-macro requires 10 instructions, 1 temp register, and 2 constant locations.

```
CE0 = { 9.61597636e-03, -1.32823968e-03, 1.47491097e-04, -1.08635004e-05 };
CE1 = { 1.00000000e+00, -6.93147182e-01, 2.40226462e-01, -5.55036440e-02 };

/* Rt != Ro && Rt != Ri */
EXP_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   EXP Rt, Ri.x;                  /* Use appropriate component of Ri */
   MAD Rt.w, c[CE0].w, Rt.y, c[CE0].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].x;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].w;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].x;
   RCP Rt.w, Rt.w;
   MUL Ro, Rt.w, Rt.x;            /* Apply user write mask to Ro */
}
```

Simulation gives |max abs error| < 3.77e-07 over the range (0.0 <= x < 1.0). Actual vertex program precision may be slightly less accurate than this.

What if more exponential precision is needed than provided by the builtin LOG instruction?

The pseudo-macro requires 10 instructions, 1 temp register, and 3 constant locations.

```
CL0 = { 2.41873696e-01, -1.37531206e-01, 5.20646796e-02, -9.31049418e-03 };
CL1 = { 1.44268966e+00, -7.21165776e-01, 4.78684813e-01, -3.47305417e-01 };
CL2 = { 1.0, NA, NA, NA };

/* Rt != Ro && Rt != Ri */
LOG_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   LOG Rt, Ri.x;                  /* Use appropriate component of Ri */
   ADD Rt.y, Rt.y, -c[CL2].x;
   MAD Rt.w, c[CL0].w, Rt.y, c[CL0].z;
   MAD Rt.w, Rt.w, Rt.y,c[CL0].y;
   MAD Rt.w, Rt.w, Rt.y,c[CL0].x;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].w;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].z;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].y;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].x;
   MAD Ro, Rt.w, Rt.y, Rt.x;      /* Apply user write mask to Ro */
}
```

Simulation gives |max abs error| < 1.79e-07 over the range (1.0 <= x < 2.0). Actual vertex program precision may be slightly less accurate than this.

New Procedures and Functions

```
    void BindProgramNV(enum target, uint id);

    void DeleteProgramsNV(sizei n, const uint *ids);

    void ExecuteProgramNV(enum target, uint id, const float *params);

    void GenProgramsNV(sizei n, uint *ids);

    boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                  boolean *residences);

    void RequestResidentProgramsNV(sizei n, uint *ids);

    void GetProgramParameterfvNV(enum target, uint index,
                                 enum pname, float *params);
    void GetProgramParameterdvNV(enum target, uint index,
                                 enum pname, double *params);

    void GetProgramivNV(uint id, enum pname, int *params);

    void GetProgramStringNV(uint id, enum pname, ubyte *program);

    void GetTrackMatrixivNV(enum target, uint address,
                            enum pname, int *params);

    void GetVertexAttribdvNV(uint index, enum pname, double *params);
    void GetVertexAttribfvNV(uint index, enum pname, float *params);
    void GetVertexAttribivNV(uint index, enum pname, int *params);

    void GetVertexAttribPointervNV(uint index, enum pname, void **pointer);

    boolean IsProgramNV(uint id);

    void LoadProgramNV(enum target, uint id, sizei len,
                       const ubyte *program);

    void ProgramParameter4fNV(enum target, uint index,
                              float x, float y, float z, float w)
    void ProgramParameter4dNV(enum target, uint index,
                              double x, double y, double z, double w)

    void ProgramParameter4dvNV(enum target, uint index,
                               const double *params);
    void ProgramParameter4fvNV(enum target, uint index,
                               const float *params);

    void ProgramParameters4dvNV(enum target, uint index,
                                uint num, const double *params);
    void ProgramParameters4fvNV(enum target, uint index,
                                uint num, const float *params);

    void TrackMatrixNV(enum target, uint address,
                       enum matrix, enum transform);
```

```
    void VertexAttribPointerNV(uint index, int size, enum type, sizei stride,
                               const void *pointer);

    void VertexAttrib1sNV(uint index, short x);
    void VertexAttrib1fNV(uint index, float x);
    void VertexAttrib1dNV(uint index, double x);
    void VertexAttrib2sNV(uint index, short x, short y);
    void VertexAttrib2fNV(uint index, float x, float y);
    void VertexAttrib2dNV(uint index, double x, double y);
    void VertexAttrib3sNV(uint index, short x, short y, short z);
    void VertexAttrib3fNV(uint index, float x, float y, float z);
    void VertexAttrib3dNV(uint index, double x, double y, double z);
    void VertexAttrib4sNV(uint index, short x, short y, short z, short w);
    void VertexAttrib4fNV(uint index, float x, float y, float z, float w);
    void VertexAttrib4dNV(uint index, double x, double y, double z, double w);
    void VertexAttrib4ubNV(uint index, ubyte x, ubyte y, ubyte z, ubyte w);

    void VertexAttrib1svNV(uint index, const short *v);
    void VertexAttrib1fvNV(uint index, const float *v);
    void VertexAttrib1dvNV(uint index, const double *v);
    void VertexAttrib2svNV(uint index, const short *v);
    void VertexAttrib2fvNV(uint index, const float *v);
    void VertexAttrib2dvNV(uint index, const double *v);
    void VertexAttrib3svNV(uint index, const short *v);
    void VertexAttrib3fvNV(uint index, const float *v);
    void VertexAttrib3dvNV(uint index, const double *v);
    void VertexAttrib4svNV(uint index, const short *v);
    void VertexAttrib4fvNV(uint index, const float *v);
    void VertexAttrib4dvNV(uint index, const double *v);
    void VertexAttrib4ubvNV(uint index, const ubyte *v);

    void VertexAttribs1svNV(uint index, sizei n, const short *v);
    void VertexAttribs1fvNV(uint index, sizei n, const float *v);
    void VertexAttribs1dvNV(uint index, sizei n, const double *v);
    void VertexAttribs2svNV(uint index, sizei n, const short *v);
    void VertexAttribs2fvNV(uint index, sizei n, const float *v);
    void VertexAttribs2dvNV(uint index, sizei n, const double *v);
    void VertexAttribs3svNV(uint index, sizei n, const short *v);
    void VertexAttribs3fvNV(uint index, sizei n, const float *v);
    void VertexAttribs3dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4svNV(uint index, sizei n, const short *v);
    void VertexAttribs4fvNV(uint index, sizei n, const float *v);
    void VertexAttribs4dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4ubvNV(uint index, sizei n, const ubyte *v);
```

New Tokens

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindProgramNV, ExecuteProgramNV, GetProgramParameter[df]vNV, GetTrackMatrixivNV, LoadProgramNV, ProgramParameter[s]4[df][v]NV, and TrackMatrixNV:

VERTEX_PROGRAM_NV	0x8620

```
Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
and GetDoublev:

    VERTEX_PROGRAM_POINT_SIZE_NV                    0x8642
    VERTEX_PROGRAM_TWO_SIDE_NV                      0x8643

Accepted by the <target> parameter of ExecuteProgramNV and
LoadProgramNV:

    VERTEX_STATE_PROGRAM_NV                         0x8621

Accepted by the <pname> parameter of GetVertexAttrib[dfi]vNV:

    ATTRIB_ARRAY_SIZE_NV                            0x8623
    ATTRIB_ARRAY_STRIDE_NV                          0x8624
    ATTRIB_ARRAY_TYPE_NV                            0x8625
    CURRENT_ATTRIB_NV                               0x8626

Accepted by the <pname> parameter of GetProgramParameterfvNV
and GetProgramParameterdvNV:

    PROGRAM_PARAMETER_NV                            0x8644

Accepted by the <pname> parameter of GetVertexAttribPointervNV:

    ATTRIB_ARRAY_POINTER_NV                         0x8645

Accepted by the <pname> parameter of GetProgramivNV:

    PROGRAM_TARGET_NV                               0x8646
    PROGRAM_LENGTH_NV                               0x8627
    PROGRAM_RESIDENT_NV                             0x8647

Accepted by the <pname> parameter of GetProgramStringNV:

    PROGRAM_STRING_NV                               0x8628

Accepted by the <pname> parameter of GetTrackMatrixivNV:

    TRACK_MATRIX_NV                                 0x8648
    TRACK_MATRIX_TRANSFORM_NV                       0x8649

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:

    MAX_TRACK_MATRIX_STACK_DEPTH_NV                 0x862E
    MAX_TRACK_MATRICES_NV                           0x862F
    CURRENT_MATRIX_STACK_DEPTH_NV                   0x8640
    CURRENT_MATRIX_NV                               0x8641
    VERTEX_PROGRAM_BINDING_NV                       0x864A
    PROGRAM_ERROR_POSITION_NV                       0x864B
```

Accepted by the <matrix> parameter of TrackMatrixNV:

NONE	
MODELVIEW	
PROJECTION	
TEXTURE	
COLOR (if ARB_imaging is supported)	
MODELVIEW_PROJECTION_NV	0x8629

Accepted by the <matrix> parameter of TrackMatrixNV and by the <mode> parameter of MatrixMode:

MATRIX0_NV	0x8630
MATRIX1_NV	0x8631
MATRIX2_NV	0x8632
MATRIX3_NV	0x8633
MATRIX4_NV	0x8634
MATRIX5_NV	0x8635
MATRIX6_NV	0x8636
MATRIX7_NV	0x8637

(Enumerants 0x8638 through 0x863F are reserved for further matrix enumerants 8 through 15.)

Accepted by the <transform> parameter of TrackMatrixNV:

IDENTITY_NV	0x862A
INVERSE_NV	0x862B
TRANSPOSE_NV	0x862C
INVERSE_TRANSPOSE_NV	0x862D

Accepted by the <array> parameter of EnableClientState and DisableClientState, by the <cap> parameter of IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

VERTEX_ATTRIB_ARRAY0_NV	0x8650
VERTEX_ATTRIB_ARRAY1_NV	0x8651
VERTEX_ATTRIB_ARRAY2_NV	0x8652
VERTEX_ATTRIB_ARRAY3_NV	0x8653
VERTEX_ATTRIB_ARRAY4_NV	0x8654
VERTEX_ATTRIB_ARRAY5_NV	0x8655
VERTEX_ATTRIB_ARRAY6_NV	0x8656
VERTEX_ATTRIB_ARRAY7_NV	0x8657
VERTEX_ATTRIB_ARRAY8_NV	0x8658
VERTEX_ATTRIB_ARRAY9_NV	0x8659
VERTEX_ATTRIB_ARRAY10_NV	0x865A
VERTEX_ATTRIB_ARRAY11_NV	0x865B
VERTEX_ATTRIB_ARRAY12_NV	0x865C
VERTEX_ATTRIB_ARRAY13_NV	0x865D
VERTEX_ATTRIB_ARRAY14_NV	0x865E
VERTEX_ATTRIB_ARRAY15_NV	0x865F

Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv, Map1d and Map1f and by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MAP1_VERTEX_ATTRIB0_4_NV	0x8660
MAP1_VERTEX_ATTRIB1_4_NV	0x8661
MAP1_VERTEX_ATTRIB2_4_NV	0x8662
MAP1_VERTEX_ATTRIB3_4_NV	0x8663
MAP1_VERTEX_ATTRIB4_4_NV	0x8664
MAP1_VERTEX_ATTRIB5_4_NV	0x8665
MAP1_VERTEX_ATTRIB6_4_NV	0x8666
MAP1_VERTEX_ATTRIB7_4_NV	0x8667
MAP1_VERTEX_ATTRIB8_4_NV	0x8668
MAP1_VERTEX_ATTRIB9_4_NV	0x8669
MAP1_VERTEX_ATTRIB10_4_NV	0x866A
MAP1_VERTEX_ATTRIB11_4_NV	0x866B
MAP1_VERTEX_ATTRIB12_4_NV	0x866C
MAP1_VERTEX_ATTRIB13_4_NV	0x866D
MAP1_VERTEX_ATTRIB14_4_NV	0x866E
MAP1_VERTEX_ATTRIB15_4_NV	0x866F

Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv, Map2d and Map2f and by the <cap> parameter of Enable, Disable, and IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

MAP2_VERTEX_ATTRIB0_4_NV	0x8670
MAP2_VERTEX_ATTRIB1_4_NV	0x8671
MAP2_VERTEX_ATTRIB2_4_NV	0x8672
MAP2_VERTEX_ATTRIB3_4_NV	0x8673
MAP2_VERTEX_ATTRIB4_4_NV	0x8674
MAP2_VERTEX_ATTRIB5_4_NV	0x8675
MAP2_VERTEX_ATTRIB6_4_NV	0x8676
MAP2_VERTEX_ATTRIB7_4_NV	0x8677
MAP2_VERTEX_ATTRIB8_4_NV	0x8678
MAP2_VERTEX_ATTRIB9_4_NV	0x8679
MAP2_VERTEX_ATTRIB10_4_NV	0x867A
MAP2_VERTEX_ATTRIB11_4_NV	0x867B
MAP2_VERTEX_ATTRIB12_4_NV	0x867C
MAP2_VERTEX_ATTRIB13_4_NV	0x867D
MAP2_VERTEX_ATTRIB14_4_NV	0x867E
MAP2_VERTEX_ATTRIB15_4_NV	0x867F

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

-- Section 2.10 "Coordinate Transformations"

Add this initial discussion:

"Per-vertex parameters are transformed before the transformation results are used to generate primitives for rasterization, establish a raster position, or generate vertices for selection or feedback.

Each vertex's per-vertex parameters are transformed by one of two vertex transformation modes. The first vertex transformation mode is GL's conventional vertex transformation model. The second mode,

known as 'vertex program' mode, transforms the vertex's per-vertex parameters by an application-supplied vertex program.

Vertex program mode is enabled and disabled, respectively, by

```
  void Enable(enum target);
```

and

```
  void Disable(enum target);
```

with target equal to VERTEX_PROGRAM_NV. When vertex program mode is enabled, vertices are transformed by the currently bound vertex program as discussed in section 2.14."

Update the original initial paragraph in the section to read:

"When vertex program mode is disabled, vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how the transformation is controlled in the case when vertex program mode is disabled. The discussion that continues through section 2.13 applies when vertex program mode is disabled."

-- Section 2.10.2 "Matrices"

Change the first paragraph to read:

"The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIXi_NV as the argument. In the case of MATRIXi_NV, i is an integer between 0 and n-1 indicating one of n tracking matrices where n is the value of the implementation defined constant MAX_TRACK_MATRICES_NV. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. The tracking matrices of the form MATRIXi_NV are described in section 2.14.5. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix."

Change the last paragraph to read:

"The state required to implement transformations consists of a n-value integer indicating the current matrix mode (where n is 4 + the number of tracking matrices supported), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, n stacks (where n is at least 8) of at least one 4x4 matrix for each MATRIXi_NV with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW."

-- NEW Section 2.14 "Vertex Programs"

"The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable but essentially hard-wired sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters.

The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

2.14.1 The Vertex Program Execution Model

A vertex program is a sequence of floating-point 4-component vector operations that operate on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from the processing of other vertices. Vertex programs execute a finite fixed sequence of instructions with no branching or looping. Vertex programs execute without data hazards so results computed in one operation can be used immediately afterwards. The result of a vertex program is a set of vertex result vectors that becomes the transformed vertex parameters used by primitive assembly.

Vertex programs use a specific well-defined instruction set, register set, and operational model defined in the following sections.

The vertex program register set consists of five types of registers described in the following five sections.

2.14.1.1 The Vertex Attribute Registers

The Vertex Attribute Registers are sixteen 4-component vector floating-point registers containing the current vertex's per-vertex attributes. These registers are numbered 0 through 15. These registers are private to each vertex program invocation and are initialized at each vertex program invocation by the current vertex attribute state specified with VertexAttribNV commands. These registers are read-only during vertex program execution. The VertexAttribNV commands used to update the vertex attribute registers can be issued both outside and inside of Begin/End pairs. Vertex program execution is provoked by updating vertex attribute zero. Updating vertex attribute zero outside of a Begin/End pair is ignored without generating any error (identical to the Vertex command operation).

The commands

```
void VertexAttrib{1234}{sfd}NV(uint index, T coords);
void VertexAttrib{1234}{sfd}vNV(uint index, T coords);
```

```
  void VertexAttrib4ubNV(uint index, T coords);
  void VertexAttrib4ubvNV(uint index, T coords);
```

specify the particular current vertex attribute indicated by index. The coordinates for each vertex attribute are named x, y, z, and w. The VertexAttrib1NV family of commands sets the x coordinate to the provided single argument while setting y and z to 0 and w to 1. Similarly, VertexAttrib2NV sets x and y to the specified values, z to 0 and w to 1; VertexAttrib3NV sets x, y, and z, with w set to 1, and VertexAttrib4NV sets all four coordinates. The error INVALID_VALUE is generated if index is greater than 15.

No conversions are applied to the vertex attributes specified as type short, int, float, or double. However, vertex attributes specified as type ubyte are converted as described by Table 2.6.

The commands

```
  void VertexAttribs{1234}{sfd}vNV(uint index, sizei n, T coords[]);
  void VertexAttribs4ubvNV(uint index, sizei n, GLubyte coords[]);
```

specify a contiguous set of n vertex attributes. The effect of

```
  VertexAttribs{1234}{sfd}vNV(index, n, coords)
```

is the same as the command sequence

```
  #define NUM k  /* where k is 1, 2, 3, or 4 components */
  int i;
  for (i=n-1; i>=0; i--) {
    VertexAttrib{NUM}{sfd}vNV(i+index, &coords[i*NUM]);
  }
```

VertexAttribs4ubvNV behaves similarly.

The VertexAttribNV calls equivalent to VertexAttribsNV are issued in reverse order so that vertex program execution is provoked when index is zero only after all the other vertex attributes have first been specified.

2.14.1.2 The Program Parameter Registers

The Program Parameter Registers are ninety-six 4-component floating-point vector registers containing the vertex program parameters. These registers are numbered 0 through 95. This relatively large set of registers is intended to hold parameters such as matrices, lighting parameters, and constants required by vertex programs. Vertex program parameter registers can be updated in one of two ways: by the ProgramParameterNV commands outside of a Begin/End pair or by a vertex state program executed outside of a Begin/End pair (vertex state programs are discussed in section 2.14.3).

```
The commands

  void ProgramParameter4fNV(enum target, uint index,
                            float x, float y, float z, float w)
  void ProgramParameter4dNV(enum target, uint index,
                            double x, double y, double z, double w)

specify the particular program parameter indicated by index.
The coordinates values x, y, and z are assigned to the respective
components of the particular program parameter.  target must
be VERTEX_PROGRAM_NV.

The commands

  void ProgramParameter4dvNV(enum target, uint index, double *params);
  void ProgramParameter4fvNV(enum target, uint index, float *params);

operate identically to ProgramParameter4fNV and ProgramParameter4dNV
respectively except that the program parameters are passed as an
array of four components.

The commands

  void ProgramParameters4dvNV(enum target, uint index,
                              uint num, double *params);
  void ProgramParameters4fvNV(enum target, uint index,
                              uint num, float *params);

specify a contiguous set of num program parameters.  The effect is
the same as

  for (i=index; i<index+num; i++) {
    ProgramParameter4{fd}vNV(i, params + i*4);
  }

The program parameter registers are shared to all vertex program
invocations within a rendering context.  ProgramParameterNV command
updates and vertex state program executions are serialized with
respect to vertex program invocations and other vertex state program
executions.

Writes to the program parameter registers during vertex state program
execution can be maskable on a per-component basis.

The error INVALID_VALUE is generated if any ProgramParameterNV has
an index is greater than 95.

The initial value of all ninety-six program parameter registers is
(0,0,0,0).
```

2.14.1.3 The Address Register

```
The Address Register is a single 4-component vector signed 32-bit
integer register though only the x component of the vector is
accessible.  The register is private to each vertex program invocation
and is initialized to (0,0,0,0) at every vertex program invocation.
This register can be written during vertex program execution (but
```

not read) and its value can be used for as a relative offset for reading vertex program parameter registers. Only the vertex program parameter registers can be read using relative addressing (writes using relative addressing are not supported).

See the discussion of relative addressing of program parameters in section 2.14.1.9 and the discussion of the ARL instruction in section 2.14.1.10.1.

2.14.1.4 The Temporary Registers

The Temporary Registers are twelve 4-component floating-point vector registers used to hold temporary results during vertex program execution. These registers are numbered 0 through 11. These registers are private to each vertex program invocation and initialized to (0,0,0,0) at every vertex program invocation. These registers can be read and written during vertex program execution. Writes to these registers can be maskable on a per-component basis.

2.14.1.5 The Vertex Result Register Set

The Vertex Result Registers are fifteen 4-component floating-point vector registers used to write the results of a vertex program. Each register value is initialized to (0,0,0,1) at the invocation of each vertex program. Writes to the vertex result registers can be maskable on a per-component basis. These registers are named in Table X.1 and further discussed below.

Vertex Result Register Name	Description	Component Interpretation
HPOS	Homogeneous clip space position	(x,y,z,w)
COL0	Primary color (front-facing)	(r,g,b,a)
COL1	Secondary color (front-facing)	(r,g,b,a)
BFC0	Back-facing primary color	(r,g,b,a)
BFC1	Back-facing secondary color	(r,g,b,a)
FOGC	Fog coordinate	(f,*,*,*)
PSIZ	Point size	(p,*,*,*)
TEX0	Texture coordinate set 0	(s,t,r,q)
TEX1	Texture coordinate set 1	(s,t,r,q)
TEX2	Texture coordinate set 2	(s,t,r,q)
TEX3	Texture coordinate set 3	(s,t,r,q)
TEX4	Texture coordinate set 4	(s,t,r,q)
TEX5	Texture coordinate set 5	(s,t,r,q)
TEX6	Texture coordinate set 6	(s,t,r,q)
TEX7	Texture coordinate set 7	(s,t,r,q)

Table X.1: Vertex Result Registers.

HPOS is the transformed vertex's homogeneous clip space position. The vertex's homogeneous clip space position is converted to normalized device coordinates and transformed to window coordinates as described at the end of section 2.10 and in section 2.11. Further processing (subsequent to vertex program termination) is responsible for clipping primitives assembled from vertex program-generated vertices as described in section 2.10 but all

client-defined clip planes are treated as if they are disabled when vertex program mode is enabled.

Four distinct color results can be generated for each vertex. COL0 is the transformed vertex's front-facing primary color. COL1 is the transformed vertex's front-facing secondary color. BFC0 is the transformed vertex's back-facing primary color. BFC1 is the transformed vertex's back-facing secondary color.

Primitive coloring may operate in two-sided color mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_NV. The selection between the back-facing colors and the front-facing colors depends on the primitive of which the vertex is a part. If the primitive is a point or a line segment, the front-facing colors are always selected. If the primitive is a polygon and two-sided color mode is disabled, the front-facing colors are selected. If it is a polygon and two-sided color mode is enabled, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. This facingness determination is identical to the two-sided lighting facingness determination described in section 2.13.1.

The selected primary and secondary colors for each primitive are clamped to the range [0,1] and then interpolated across the assembled primitive during rasterization with at least 8-bit accuracy for each color component.

FOGC is the transformed vertex's fog coordinate. The register's first floating-point component is interpolated across the assembled primitive during rasterization and used as the fog distance to compute per-fragment the fog factor when fog is enabled. However, if both fog and vertex program mode are enabled, but the FOG vertex result register is not written, the fog factor is overridden to 1.0. The register's other three components are ignored.

Point size determination may operate in program-specified point size mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_NV. If the vertex is for a point primitive and the mode is enabled and the PSIZ vertex result is written, the point primitive's size is determined by the clamped x component of the PSIZ register. Otherwise (because vertex program mode is disabled, program-specified point size mode is disabled, or because the vertex program did not write PSIZ), the point primitive's size is determined by the point size state (the state specified using the PointSize command).

The PSIZ register's x component is clamped to the range zero through either the hi value of ALIASED_POINT_SIZE_RANGE if point smoothing is disabled or the hi value of the SMOOTH_POINT_SIZE_RANGE if point smoothing is enabled. The register's other three components are ignored.

If the vertex is not for a point primitive, the value of the PSIZ vertex result register is ignored.

TEX0 through TEX7 are the transformed vertex's texture coordinate

sets for texture units 0 through 7. These floating-point coordinates are interpolated across the assembled primitive during rasterization and used for accessing textures. If the number of texture units supported is less than eight, the values of vertex result registers that do not correspond to existent texture units are ignored.

2.14.1.6 Semantic Meaning for Vertex Attributes and Program Parameters

One important distinction between the conventional GL vertex transformation mode and the vertex program mode is that per-vertex parameters and other state parameters in vertex program mode do not have dedicated semantic interpretations the way that they do with the conventional GL vertex transformation mode.

For example, in the conventional GL vertex transformation mode, the Normal command specifies a per-vertex normal. The semantic that the Normal command supplies a normal for lighting is established because that is how the per-vertex attribute supplied by the Normal command is used by the conventional GL vertex transformation mode. Similarly, other state parameters such as a light source position have semantic interpretations based on how the conventional GL vertex transformation model uses each particular parameter.

In contrast, vertex attributes and program parameters for vertex programs have no pre-defined semantic meanings. The meaning of a vertex attribute or program parameter in vertex program mode is defined by how the vertex attribute or program parameter is used by the current vertex program to compute and write values to the Vertex Result Registers. This is the reason that per-vertex attributes and program parameters for vertex programs are numbered instead of named.

For convenience however, the existing per-vertex parameters for the conventional GL vertex transformation mode (vertices, normals, colors, fog coordinates, vertex weights, and texture coordinates) are aliased to numbered vertex attributes. This aliasing is specified in Table X.2. The table includes how the various conventional components map to the 4-component vertex attribute components.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.2: Aliasing of vertex attributes with conventional per-vertex parameters.

Only vertex attribute zero is treated specially because it is the attribute that provokes the execution of the vertex program; this is the attribute that aliases to the Vertex command's vertex coordinates.

The result of a vertex program is the set of post-transformation vertex parameters written to the Vertex Result Registers. All vertex programs must write a homogeneous clip space position, but the other Vertex Result Registers can be optionally written.

Clipping and culling are not the responsibility of vertex programs because these operations assume the assembly of multiple vertices into a primitive. View frustum clipping is performed subsequent to vertex program execution. Clip planes are not supported in vertex program mode.

2.14.1.7 Vertex Program Specification

Vertex programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program.

The command

```
LoadProgramNV(enum target, uint id, sizei len,
              const ubyte *program);
```

loads a vertex program when the target parameter is VERTEX_PROGRAM_NV. Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the positive integers (zero is reserved). The error INVALID_VALUE occurs if a program is loaded with an id of zero. The error INVALID_OPERATION is generated if a program is loaded for an id that is currently loaded with a

program of a different program target. Managing the program name space and binding to vertex programs is discussed later in section 2.14.1.8.

program is a pointer to an array of ubytes that represents the program being loaded. The length of the array is indicated by len.

A second program target type known as vertex state programs is discussed in 2.14.4.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for vertex programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

```
<program>              ::= "!!VP1.0" <instructionSequence> "END"

<instructionSequence>  ::= <instructionSequence> <instructionLine>
                         | <instructionLine>

<instructionLine>      ::= <instruction> ";"

<instruction>          ::= <ARL-instruction>
                         | <VECTORop-instruction>
                         | <SCALARop-instruction>
                         | <BINop-instruction>
                         | <TRIop-instruction>

<ARL-instruction>      ::= "ARL" <addrReg> "," <scalarSrcReg>

<VECTORop-instruction> ::= <VECTORop> <maskedDstReg> "," <swizzleSrcReg>

<SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrcReg>

<BINop-instruction>    ::= <BINop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg>

<TRIop-instruction>    ::= <TRIop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg> ","
                           <swizzleSrcReg>

<VECTORop>             ::= "MOV"
                         | "LIT"

<SCALARop>             ::= "RCP"
                         | "RSQ"
                         | "EXP"
                         | "LOG"
```

```
    <BINop>                ::= "MUL"
                             | "ADD"
                             | "DP3"
                             | "DP4"
                             | "DST"
                             | "MIN"
                             | "MAX"
                             | "SLT"
                             | "SGE"

    <TRIop>                ::= "MAD"

    <scalarSrcReg>         ::= <optionalSign> <srcReg> <scalarSuffix>

    <swizzleSrcReg>        ::= <optionalSign> <srcReg> <swizzleSuffix>

    <maskedDstReg>         ::= <dstReg> <optionalMask>

    <optionalMask>         ::= ""
                             | "." "x"
                             | "."     "y"
                             | "." "x" "y"
                             | "."         "z"
                             | "." "x"     "z"
                             | "."     "y" "z"
                             | "." "x" "y" "z"
                             | "."             "w"
                             | "." "x"         "w"
                             | "."     "y"     "w"
                             | "." "x" "y"     "w"
                             | "."         "z" "w"
                             | "." "x"     "z" "w"
                             | "."     "y" "z" "w"
                             | "." "x" "y" "z" "w"

    <optionalSign>         ::= "-"
                             | ""

    <srcReg>               ::= <vertexAttribReg>
                             | <progParamReg>
                             | <temporaryReg>

    <dstReg>               ::= <temporaryReg>
                             | <vertexResultReg>

    <vertexAttribReg>      ::= "v" "[" vertexAttribRegNum "]"
```

```
<vertexAttribRegNum>   ::= decimal integer from 0 to 15 inclusive
                         | "OPOS"
                         | "WGHT"
                         | "NRML"
                         | "COL0"
                         | "COL1"
                         | "FOGC"
                         | "TEX0"
                         | "TEX1"
                         | "TEX2"
                         | "TEX3"
                         | "TEX4"
                         | "TEX5"
                         | "TEX6"
                         | "TEX7"

<progParamReg>         ::= <absProgParamReg>
                         | <relProgParamReg>

<absProgParamReg>      ::= "c" "[" <progParamRegNum> "]"

<progParamRegNum>      ::= decimal integer from 0 to 95 inclusive

<relProgParamReg>      ::= "c" "[" <addrReg> "]"
                         | "c" "[" <addrReg> "+" <progParamPosOffset> "]"
                         | "c" "[" <addrReg> "-" <progParamNegOffset> "]"

<progParamPosOffset>   ::= decimal integer from 0 to 63 inclusive

<progParamNegOffset>   ::= decimal integer from 0 to 64 inclusive

<addrReg>              ::= "A0" "." "x"

<temporaryReg>         ::= "R0"
                         | "R1"
                         | "R2"
                         | "R3"
                         | "R4"
                         | "R5"
                         | "R6"
                         | "R7"
                         | "R8"
                         | "R9"
                         | "R10"
                         | "R11"

<vertexResultReg>      ::= "o" "[" vertexResultRegName "]"
```

```
    <vertexResultRegName>  ::= "HPOS"
                             | "COL0"
                             | "COL1"
                             | "BFC0"
                             | "BFC1"
                             | "FOGC"
                             | "PSIZ"
                             | "TEX0"
                             | "TEX1"
                             | "TEX2"
                             | "TEX3"
                             | "TEX4"
                             | "TEX5"
                             | "TEX6"
                             | "TEX7"

    <scalarSuffix>         ::= "." <component>

    <swizzleSuffix>        ::= ""
                             | "." <component>
                             | "." <component> <component>
                                 <component> <component>

    <component>            ::= "x"
                             | "y"
                             | "z"
                             | "w"
```

The <vertexAttribRegNum> rule matches both register numbers 0 through 15 and a set of mnemonics that abbreviate the aliasing of conventional the per-vertex parameters to vertex attribute register numbers. Table X.3 shows the mapping from mnemonic to vertex attribute register number and what the mnemonic abbreviates.

Mnemonic	Vertex Attribute Register Number	Meaning
"OPOS"	0	object position
"WGHT"	1	vertex weight
"NRML"	2	normal
"COL0"	3	primary color
"COL1"	4	secondary color
"FOGC"	5	fog coordinate
"TEX0"	8	texture coordinate 0
"TEX1"	9	texture coordinate 1
"TEX2"	10	texture coordinate 2
"TEX3"	11	texture coordinate 3
"TEX4"	12	texture coordinate 4
"TEX5"	13	texture coordinate 5
"TEX6"	14	texture coordinate 6
"TEX7"	15	texture coordinate 7

Table X.3: The mapping between vertex attribute register numbers, mnemonics, and meanings.

A vertex programs fails to load if it does not write at least one component of the HPOS register.

A vertex program fails to load if it contains more than 128 instructions.

A vertex program fails to load if any instruction sources more than one unique program parameter register.

A vertex program fails to load if any instruction sources more than one unique vertex attribute register.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the last loaded program string indicating where the first error in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

2.14.1.8 Vertex Program Binding and Program Management

The current vertex program is invoked whenever vertex attribute zero is updated (whether by a VertexAttributeNV or Vertex command). The current vertex program is updated by

```
BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV. This binds the vertex program named by id as the current vertex program. The error INVALID_OPERATION is generated if id names a program that is not a vertex program (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error. However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program

mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

Programs are deleted by calling

```
void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
void GenProgramsNV(sizei n, uint *ids);
```

returns n previously unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV. The error INVALID_VALUE is generated if n is negative.

An implementation may choose to establish a working set of programs on which binding and ExecuteProgramNV operations (execute programs are explained in section 2.14.4) are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
boolean AreProgramsResidentNV(sizei n, const uint *ids,
                              boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
  void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

2.14.1.9 Vertex Program Register Accesses

There are 17 vertex program instructions. The instructions and their respective input and output parameters are summarized in Table X.4.

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum
MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set on greater equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients

Table X.4: Summary of vertex program instructions. "v" indicates a vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Instructions use either scalar source values or swizzled source values, indicated in the grammar (see section 2.14.1.7) by the rules <scalarSrcReg> and <swizzleSrcReg> respectively. Either type of source value is negated when the <optionalSign> rule matches "-".

Scalar source register values select one of the source register's four components based on the <component> of the <scalarSuffix> rule. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. The indicated component is used as a scalar for the particular source value.

Swizzled source register values may arbitrarily swizzle the source register's components based on the <swizzleSuffix> rule. In the case where the <swizzleSuffix> matches (ignoring whitespace) the pattern ".????" where each question mark is one of "x", "y", "z", or "w", this indicates the ith component of the source register value should come from the component named by the ith component in the sequence. For example, if the swizzle suffix is ".yzzx" and the source register contains [2.0, 8.0, 9.0, 0.0] the swizzled source register value used by the instruction is [8.0, 9.0, 9.0, 2.0].

If the <swizzleSuffix> rule matches "", this is treated the same as ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

The register sourced for either a scalar source register value or a swizzled source register value is indicated in the grammar by the rule <srcReg>. The <vertexAttribReg>, <progParamReg>, and <temporaryReg> sub-rules correspond to one of the vertex attribute registers, program parameter registers, or temporary register respectively.

The vertex attribute and temporary registers are accessed absolutely based on the numbered register. In the case of vertex attribute registers, if the <vertexAttribRegNum> corresponds to a mnemonic, the corresponding register number from Table X.3 is used.

Either absolute or relative addressing can be used to access the program parameter registers. Absolute addressing is indicated by the grammar by the <absProgParamReg> rule. Absolute addressing accesses the numbered program parameter register indicated by the <progParamRegNum> rule. Relative addressing accesses the numbered program parameter register plus an offset. The offset is the positive value of <progParamPosOffset> if the <progParamPosOffset> rule is matched, or the offset is the negative value of <progParamNegOffset> if the <progParamNegOffset> rule is matched, or otherwise the offset is zero. Relative addressing is available only for program parameter registers and only for reads (not writes). Relative addressing reads outside of the 0 to 95 inclusive range always read the value (0,0,0,0).

The result of all instructions except ARL is written back to a masked destination register, indicated in the grammar by the rule <maskedDstReg>.

Writes to each component of the destination register can be masked, indicated in the grammar by the <optionalMask> rule. If the optional mask is "", all components are written. Otherwise, the optional mask names particular components to write. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be written but not the y component. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The actual destination register is indicated in the grammar by the rule <dstReg>. The <temporaryReg> and <vertexResultReg>

sub-rules correspond to either the temporary registers or vertex result registers. The temporary registers are determined and accessed as described earlier.

The vertex result registers are accessed absolutely based on the named register. The <vertexResultRegName> rule corresponds to registers named in Table X.1.

2.14.1.10 Vertex Program Instruction Set Operations

The operation of the 17 vertex program instructions are described in this section. After the textual description of each instruction's operation, a register transfer level description is also presented.

The following conventions are used in each instruction's register transfer level description. The 4-component vector variables "t", "u", and "v" are assigned intermediate results. The destination register is called "destination". The three possible source registers are called "source0", "source1", and "source2" respectively.

The x, y, z, and w vector components are referred to with the suffixes ".x", ".y", ".z", and ".w" respectively. The suffix ".c" is used for scalar source register values and c represents the particular source register's selected scalar component. Swizzling of components is indicated with the suffixes ".c***", ".*c**", ".**c*", and ".***c" where c is meant to indicate the x, y, z, or w component selected for the particular source operand swizzle configuration. For example:

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
```

This example indicates that t should be assigned the swizzled version of the source0 operand based on the source0 operand's swizzle configuration.

The variables "negate0", "negate1", and "negate2" are booleans that are true when the respective source value should be negated. The variables "xmask", "ymask", "zmask", and "wmask" are booleans that are true when the destination write mask for the respective component is enabled for writing.

Otherwise, the register transfer level descriptions mimic ANSI C syntax.

The idiom "IEEE(expression)" represents the s23e8 single-precision result of the expression if evaluated using IEEE single-precision floating point operations. The IEEE idiom is used to specify the maximum allowed deviation from IEEE single-precision floating-point arithmetic results.

The following abbreviations are also used:

```
  +Inf    floating-point representation of positive infinity
  -Inf    floating-point representation of negative infinity
  +NaN    floating-point representation of positive not a number
  -NaN    floating-point representation of negative not a number
  NA      not applicable or not used
```

2.14.1.10.1 ARL: Address Register Load

The ARL instruction moves value of the source scalar into the address register. Conceptually, the address register load instruction is a 4-component vector signed integer register, but the only valid address register component for writing and indexing is the x component. The only use for A0.x is as a base address for program parameter reads. The source value is a float that is truncated towards negative infinity into a signed integer.

```
t.x = source0.c;
if (negate0) t.x = -t.x;
A0.x = floor(t.x);
```

2.14.1.10.2 MOV: Move

The MOV instruction moves the value of the source vector into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
if (xmask) destination.x = t.x;
if (ymask) destination.y = t.y;
if (zmask) destination.z = t.z;
if (wmask) destination.w = t.w;
```

2.14.1.10.3 MUL: Multiply

The MUL instruction multiplies the values of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = t.x * u.x;
if (ymask) destination.y = t.y * u.y;
if (zmask) destination.z = t.z * u.z;
if (wmask) destination.w = t.w * u.w;
```

2.14.1.10.4 ADD: Add

The ADD instruction adds the values of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = t.x + u.x;
if (ymask) destination.y = t.y + u.y;
if (zmask) destination.z = t.z + u.z;
if (wmask) destination.w = t.w + u.w;
```

2.14.1.10.5 MAD: Multiply and Add

The MAD instruction adds the value of the third source vector to the product of the values of the first and second two source vectors, writing the result to the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = source2.c***;
v.y = source2.*c**;
v.z = source2.**c*;
v.w = source2.***c;
if (negate2) {
  v.x = -v.x;
  v.y = -v.y;
  v.z = -v.z;
  v.w = -v.w;
}
if (xmask) destination.x = t.x * u.x + v.x;
if (ymask) destination.y = t.y * u.y + v.y;
if (zmask) destination.z = t.z * u.z + v.z;
if (wmask) destination.w = t.w * u.w + v.w;
```

2.14.1.10.6 RCP: Reciprocal

The RCP instruction inverts the value of the source scalar into the destination register. The reciprocal of exactly 1.0 must be exactly 1.0.

Additionally the reciprocal of negative infinity gives [-0.0, -0.0, -0.0, -0.0]; the reciprocal of negative zero gives [-Inf, -Inf, -Inf, -Inf]; the reciprocal of positive zero gives [+Inf, +Inf, +Inf, +Inf]; and the reciprocal of positive infinity gives [0.0, 0.0, 0.0, 0.0].

```
t.x = source0.c;
if (negate0) {
  t.x = -t.x;
}
if (t.x == 1.0f) {
  u.x = 1.0f;
} else {
  u.x = 1.0f / t.x;
}
if (xmask) destination.x = u.x;
if (ymask) destination.y = u.x;
if (zmask) destination.z = u.x;
if (wmask) destination.w = u.x;
```

where

```
| u.x - IEEE(1.0f/t.x) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 2.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.7 RSQ: Reciprocal Square Root

The RSQ instruction assigns the inverse square root of the absolute value of the source scalar into the destination register.

Additionally, RSQ(0.0) gives [+Inf, +Inf, +Inf, +Inf]; and both RSQ(+Inf) and RSQ(-Inf) give [0.0, 0.0, 0.0, 0.0];

```
t.x = source0.c;
if (negate0) {
  t.x = -t.x;
}
u.x = 1.0f / sqrt(fabs(t.x));
if (xmask) destination.x = u.x;
if (ymask) destination.y = u.x;
if (zmask) destination.z = u.x;
if (wmask) destination.w = u.x;
```

where

```
| u.x - IEEE(1.0f/sqrt(fabs(t.x))) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 4.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.8 DP3: Three-Component Dot Product

The DP3 instruction assigns the three-component dot product of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.9 DP4: Four-Component Dot Product

The DP4 instruction assigns the four-component dot product of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z + t.w * u.w;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.10 DST: Distance Vector

The DST instructions calculates a distance vector for the values of two source vectors. The first vector is assumed to be [NA, d*d, d*d, NA] and the second source vector is assumed to be [NA, 1.0/d, NA, 1.0/d], where the value of a component labeled NA is undefined. The destination vector is then assigned [1,d,d*d,1.0/d].

```
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0) {
  t.y = -t.y;
  t.z = -t.z;
}
u.y = source1.*c**;
u.w = source1.***c;
if (negate1) {
  u.y = -u.y;
  u.w = -u.w;
}
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.y*u.y;
if (zmask) destination.z = t.z;
if (wmask) destination.w = u.w;
```

2.14.1.10.11 MIN: Minimum

The MIN instruction assigns the component-wise minimum of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x < u.x) ? t.x : u.x;
if (ymask) destination.y = (t.y < u.y) ? t.y : u.y;
if (zmask) destination.z = (t.z < u.z) ? t.z : u.z;
if (wmask) destination.w = (t.w < u.w) ? t.w : u.w;
```

2.14.1.10.12 MAX: Maximum

The MAX instruction assigns the component-wise maximum of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x >= u.x) ? t.x : u.x;
if (ymask) destination.y = (t.y >= u.y) ? t.y : u.y;
if (zmask) destination.z = (t.z >= u.z) ? t.z : u.z;
if (wmask) destination.w = (t.w >= u.w) ? t.w : u.w;
```

2.14.1.10.13 SLT: Set On Less Than

The SLT instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is less than the value of the second source vector; otherwise, 0.0 is assigned.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x < u.x) ? 1.0 : 0.0;
if (ymask) destination.y = (t.y < u.y) ? 1.0 : 0.0;
if (zmask) destination.z = (t.z < u.z) ? 1.0 : 0.0;
if (wmask) destination.w = (t.w < u.w) ? 1.0 : 0.0;
```

2.14.1.10.14 SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is greater than or equal the value of the second source vector; otherwise, 0.0 is assigned.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x >= u.x) ? 1.0 : 0.0;
if (ymask) destination.y = (t.y >= u.y) ? 1.0 : 0.0;
if (zmask) destination.z = (t.z >= u.z) ? 1.0 : 0.0;
if (wmask) destination.w = (t.w >= u.w) ? 1.0 : 0.0;
```

2.14.1.10.15 EXP: Exponential Base 2

The EXP instruction generates an approximation of the exponential base 2 for the value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The exponential base 2 of the source scalar can be better approximated by destination.x*FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) to 2^destination.y where 0.0 <= destination.y < 1.0.

Additionally, EXP(-Inf) or if the exponential result underflows gives [0.0, 0.0, 0.0, 0.0]; and EXP(+Inf) or if the exponential result overflows gives [+Inf, 0.0, +Inf, 1.0].

```
t.x = source0.c;
if (negate0) {
  t.x = -t.x;
}
q.x = 2^floor(t.x);
q.y = t.x - floor(t.x);
q.z = q.x * APPX(q.y);
if (xmask) destination.x = q.x;
if (ymask) destination.y = q.y;
if (zmask) destination.z = q.z;
if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of exponential base 2 such that

```
| exp(q.y*log(2.0))-APPX(q.y) | < 1/(2^11)
```

for all 0 <= q.y < 1.0.

The expression "2^floor(t.x)" should overflow to +Inf and underflow to zero.

2.14.1.10.16 LOG: Logarithm Base 2

The LOG instruction generates an approximation of the logarithm base 2 for the absolute value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The logarithm base 2 of the absolute value of the source scalar can be better approximated by destination.x+FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) of log2(destination.y) where 1.0 <= destination.y < 2.0.

Additionally, LOG(0.0) gives [-Inf, 1.0, -Inf, 1.0]; and both LOG(+Inf) and LOG(-Inf) give [+Inf, 1.0, +Inf, 1.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (fabs(t.x) != 0.0f) {
      if (fabs(t.x) == +Inf) {
        q.x = +Inf;
        q.y = 1.0;
        q.z = +Inf;
      } else {
        q.x = Exponent(t.x);
        q.y = Mantissa(t.x);
        q.z = q.x + APPX(q.y);
      }
    } else {
      q.x = -Inf;
      q.y = 1.0;
      q.z = -Inf;
    }
    if (xmask) destination.x = q.x;
    if (ymask) destination.y = q.y;
    if (zmask) destination.z = q.z;
    if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of logarithm base 2 such that

```
    | log(q.y)/log(2.0) - APPX(q.y) | < 1/(2^11)
```

for all 1.0 <= q.y < 2.0.

The "Exponent(t.x)" function returns the unbiased exponent between -126 and 127. For example, "Exponent(1.0)" equals 0.0. (Note that the IEEE floating-point representation maintains the exponent as a biased value.) Larger or smaller exponents should generate +Inf or -Inf respectively. The "Mantissa(t.x)" function returns a value in the range [1.0f, 2.0). The intent of these functions is that fabs(t.x) is approximately "Mantissa(t.x)*2^Exponent(t.x)".

2.14.1.10.17 LIT: Light Coefficients

The LIT instruction is intended to compute ambient, diffuse, and specular lighting coefficients from a diffuse dot product, a specular dot product, and a specular power that is clamped to (-128,128) exclusive. The x component of the source vector is assumed to contain a diffuse dot product (unit normal vector dotted with a unit light vector). The y component of the source vector is assumed to contain a Blinn specular dot product (unit normal vector dotted with a unit half-angle vector). The w component is assumed to contain a specular power.

An implementation must support at least 8 fraction bits in the specular power. Note that because 0.0 times anything must be 0.0, taking any base to the power of 0.0 will yield 1.0.

```
t.x = source0.c***;
t.y = source0.*c**;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.w = -t.w;
}
if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
else if (t.w > 128-epsilon) t.w = 128-epsilon;
if (t.x < 0.0) t.x = 0.0;
if (t.y < 0.0) t.y = 0.0;
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.x;
if (zmask) destination.z = (t.x > 0.0) ? EXP(t.w*LOG(t.y)) : 0.0;
if (wmask) destination.w = 1.0;
```

where EXP and LOG are functions that approximate the exponential base 2 and logarithm base 2 with the identical accuracy and special case requirements of the EXP and LOG instructions. epsilon is 1.0/256.0 or approximately 0.0039 which would correspond to representing the specular power with a s8.8 representation.

2.14.1.11 Vertex Program Floating Point Requirements

All vertex program calculations are assumed to use IEEE single precision floating-point math with a format of s1e8m23 (one signed bit, 8 bits of exponent, 23 bits of magnitude) or better and the round-to-zero rounding mode. The only exceptions to this are the RCP, RSQ, LOG, EXP, and LIT instructions.

Note that (positive or negative) 0.0 times anything is (positive) 0.0.

The RCP and RSQ instructions deliver results accurate to 1.0/(2^22) and the approximate output (the z component) of the EXP and LOG instructions only has to be accurate to 1.0/(2^11). The LIT instruction specular output (the z component) is allowed an error equivalent to the combination of the EXP and LOG combination to implement a power function.

The floor operations used by the ARL and EXP instructions must operate identically. Specifically, the EXP instruction's floor(t.x) intermediate result must exactly match the integer stored in the address register by the ARL instruction.

Since distance is calculated as (d^2)*(1/sqrt(d^2)), 0.0 multiplied by anything must be 0.0. This affects the MUL, MAD, DP3, DP4, DST, and LIT instructions.

Because if/then/else conditional evaluation is done by multiplying by 1.0 or 0.0 and adding, the floating point computations require:

```
0.0 * x = 0.0    for all x (including +Inf, -Inf, +NaN, and -NaN)
1.0 * x = x      for all x (including +Inf and -Inf)
0.0 + x = x      for all x (including +Inf and -Inf)
```

Including +Inf, -Inf, +NaN, and -NaN when applying the above three rules is recommended but not required. (The recommended inclusion of +Inf, -Inf, +NaN, and -NaN when applying the first rule is inconsistent with IEEE floating-point requirements.)

For the purpose of comparisons performed by the SGE and SLT instructions, -0.0 is less than +0.0. (This is inconsistent with IEEE floating-point requirements).

No floating-point exceptions or interrupts are generated. Denorms are not supported; if a denorm is input, it is treated as 0.0 (ie, denorms are flushed to zero).

Computations involving +NaN or -NaN generate +NaN, except for the requirement that zero times +NaN or -NaN must always be zero. (This exception is inconsistent with IEEE floating-point requirements).

2.14.2 Vertex Program Update for the Current Raster Position

When vertex programs are enabled, the raster position is determined by the current vertex program. The raster position specified by RasterPos is treated as if they were specified in a Vertex command. The contents of vertex result register set is used to update respective raster position state.

Assuming an existent program, the homogeneous clip-space coordinates are passed to clipping as if they represented a point and assuming no client-defined clip planes are enabled. If the point is not culled, then the projection to window coordinates is computed (section 2.10) and saved as the current raster position and the valid bit is set. If the current vertex program is nonexistent or the "point" is culled, the current raster position and its associated data become indeterminate and the raster position valid bit is cleared.

2.14.3 Vertex Arrays for Vertex Attributes

Data for vertex attributes in vertex program mode may be specified using vertex array commands. The client may specify and enable any of sixteen vertex attribute arrays.

The vertex attribute arrays are ignored when vertex program mode is disabled. When vertex program mode is enabled, vertex attribute arrays are used.

The command

```
  void VertexAttribPointerNV(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

describes the locations and organizations of the sixteen vertex attribute arrays. index specifies the particular vertex attribute to be described. size indicates the number of values per vertex that are stored in the array; size must be one of 1, 2, 3, or 4. type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, double, and ubyte respectively. The INVALID_OPERATION error is generated if type is UNSIGNED_BYTE and size is not 4. The INVALID_VALUE error is generated if index is greater than 15. The INVALID_VALUE error is generated if stride is negative.

The one, two, three, or four values in an array that correspond to a single vertex attribute comprise an array element. The values within each array element at stored sequentially in memory. If the stride is specified as zero, then array elements are stored sequentially as well. Otherwise points to the ith and (i+1)st elements of an array differ by stride basic machine units (typically unsigned bytes), the pointer to the (i+1)st element being greater. pointer specifies the location in memory of the first value of the first element of the array being specified.

Vertex attribute arrays are enabled with the EnableClientState command and disabled with the DisableClientState command. The value of the argument to either command is VERTEX_ATTRIB_ARRAYi_NV where i is an integer between 0 and 15; specifying a value of i enables or disables the vertex attribute array with index i. The constants obey VERTEX_ATTRIB_ARRAYi_NV = VERTEX_ATTRIB_ARRAY0_NV + i.

When vertex program mode is enabled, the ArrayElement command operates as described in this section in contrast to the behavior described in section 2.8. Likewise, any vertex array transfer commands that are defined in terms of ArrayElement (DrawArrays, DrawElements, and DrawRangeElements) assume the operation of ArrayElement described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the ArrayElement command transfers the ith element of particular enabled vertex arrays as described below. For each enabled vertex attribute array, it is as though the corresponding command from section 2.14.1.1 were called with a pointer to element i. For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,d,ub], corresponding to the array types short, int, float, double, and ubyte respectively.

However, if a given vertex attribute array is disabled, but its corresponding aliased conventional per-vertex parameter's vertex array (as described in section 2.14.1.6) is enabled, then it is

as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. In this case, the corresponding command is determined as described in section 2.8's description of ArrayElement.

If the vertex attribute array 0 is enabled, it is as though VertexAttrib[size][type]v(0, ...) is executed last, after the executions of other corresponding commands. If the vertex attribute array 0 is disabled but the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of other corresponding commands.

2.14.4 Vertex State Programs

Vertex state programs share the same instruction set as and a similar execution model to vertex programs. While vertex program are executed implicitly when a vertex transformation is provoked, vertex state programs are executed explicitly, independently of any vertices. Vertex state programs can write program parameter registers, but may not write vertex result registers.

The purpose of a vertex state program is to update program parameter registers by means of an application-defined program. Typically, an application will load a set of program parameters and then execute a vertex state program that reads and updates the program parameter registers. For example, a vertex state program might normalize a set of unnormalized vectors previously loaded as program parameters. The expectation is that subsequently executed vertex programs would use the normalized program parameters.

Vertex state programs are loaded with the same LoadProgramNV command (see section 2.14.1.7) used to load vertex programs except that the target must be VERTEX_STATE_PROGRAM_NV when loading a vertex state program.

Vertex state programs must conform to a more limited grammar than the grammar for vertex programs. The vertex state program grammar for syntactically valid sequences is the same as the grammar defined in section 2.14.1.7 with the following modified rules:

```
<program>              ::= "!!VSP1.0" <instructionSequence> "END"

<dstReg>               ::= <absProgParamReg>
                         | <temporaryReg>

<vertexAttribReg>      ::= "v" "[" "0" "]"
```

A vertex state program fails to load if it does not write at least one program parameter register.

A vertex state program fails to load if it contains more than 128 instructions.

A vertex state program fails to load if any instruction sources more than one unique program parameter register.

A vertex state program fails to load if any instruction sources more than one unique vertex attribute register (this is necessarily true because only vertex attribute 0 is available in vertex state programs).

The error INVALID_OPERATION is generated if a vertex state program fails to load because it is not syntactically correct or for one of the other reasons listed above.

A successfully loaded vertex state program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

Executing vertex state programs is legal only outside a Begin/End pair. A vertex state program may not read any vertex attribute register other than register zero. A vertex state program may not write any vertex result register.

The command

```
ExecuteProgramNV(enum target, uint id, const float *params);
```

executes the vertex state program named by id. The target must be VERTEX_STATE_PROGRAM_NV and the id must be the name of program loaded with a target type of VERTEX_STATE_PROGRAM_NV. params points to an array of four floating-point values that are loaded into vertex attribute register zero (the only vertex attribute readable from a vertex state program).

The INVALID_OPERATION error is generated if the named program is nonexistent, is invalid, or the program is not a vertex state program. A vertex state program may not be valid for reasons explained in section 2.14.5.

2.14.5 Tracking Matrices

As a convenience to applications, standard GL matrix state can be tracked into program parameter vectors. This permits vertex programs to access matrices specified through GL matrix commands.

In addition to GL's conventional matrices, several additional matrices are available for tracking. These matrices have names of the form MATRIXi_NV where i is between zero and n-1 where n is the value of the MAX_TRACK_MATRICES_NV implementation dependent constant. The MATRIXi_NV constants obey MATRIXi_NV = MATRIX0_NV + i. The value of MAX_TRACK_MATRICES_NV must be at least eight. The maximum stack depth for tracking matrices is defined by the MAX_TRACK_MATRIX_STACK_DEPTH_NV and must be at least 1.

The command

```
TrackMatrixNV(enum target, uint address, enum matrix, enum transform);
```

tracks a given transformed version of a particular matrix into a contiguous sequence of four vertex program parameter registers beginning at address. target must be VERTEX_PROGRAM_NV (though

tracked matrices apply to vertex state programs as well because
both vertex state programs and vertex programs shared the same
program parameter registers). matrix must be one of NONE, MODELVIEW,
PROJECTION, TEXTURE, COLOR (if the ARB_imaging subset is supported),
MODELVIEW_PROJECTION_NV, or MATRIXi_NV. transform must be one of
IDENTITY_NV, INVERSE_NV, TRANSPOSE_NV, or INVERSE_TRANSPOSE_NV.
The INVALID_VALUE error is generated if address is not a multiple
of four.

The MODELVIEW_PROJECTION_NV matrix represents the concatenation of
the current modelview and projection matrices. If M is the current
modelview matrix and P is the current projection matrix, then the
MODELVIEW_PROJECTION_NV matrix is C and computed as

    C = P M

Matrix tracking for the specified program parameter register and the
next consecutive three registers is disabled when NONE is supplied
for matrix. When tracking is disabled the previously tracked program
parameter registers retain the state of their last tracked values.
Otherwise, the specified transformed version of matrix is tracked into
the specified program parameter register and the next three registers.
Whenever the matrix changes, the transformed version of the matrix
is updated in the specified range of program parameter registers.
If TEXTURE is specified for matrix, the texture matrix for the
current active texture unit is tracked.

Matrices are tracked row-wise meaning that the top row of the
transformed matrix is loaded into the program parameter address,
the second from the top row of the transformed matrix is loaded into
the program parameter address+1, the third from the top row of the
transformed matrix is loaded into the program parameter address+2,
and the bottom row of the transformed matrix is loaded into the
program parameter address+3. The transformed matrix may be identical
to the specified matrix, the inverse of the specified matrix, the
transpose of the specified matrix, or the inverse transpose of the
specified matrix, depending on the value of transform.

When matrix tracking is enabled for a particular program parameter
register sequence, updates to the program parameter using
ProgramParameterNV commands, a vertex program, or a vertex state
program are not possible. The INVALID_OPERATION error is generated
if a ProgramParameterNV command is used to update a program parameter
register currently tracking a matrix.

When a vertex program that writes a program parameter register with
tracking enabled is bound using BindProgramNV, the vertex program
is considered invalid. As described in section 2.14.1.8, the
INVALID_OPERATION error is generated by Begin, RasterPos, or a command
that does an implicit Begin operation when the current vertex program
is invalid.

The INVALID_OPERATION error is generated by ExecuteProgramNV when
the vertex state program requested for execution writes to a program
parameter register that is currently tracking a matrix because the
program is considered invalid.

2.14.6 Required Vertex Program State

The state required for vertex programs consists of:

a bit indicating whether or not program mode is enabled;

a bit indicating whether or not two-sided color mode is enabled;

a bit indicating whether or not program-specified point size mode is enabled;

96 4-component floating-point program parameter registers;

16 4-component vertex attribute registers (though this state is aliased with the current normal, primary color, secondary color, fog coordinate, weights, and texture coordinate sets);

24 sets of matrix tracking state for each set of four sequential program parameter registers, consisting of a n-valued integer indicated the tracked matrix or GL_NONE (where n is 5 + the number of texture units supported + the number of tracking matrices supported) and a four-valued integer indicating the transformation of the tracked matrix;

an unsigned integer naming the currently bound vertex program

and the state must be maintained to indicate which integers are currently in use as program names.

Each existent program object consists of a target, a boolean indicating whether the program is resident, an array of type ubyte containing the program string, and the length of the program string array. Initially, no program objects exist.

Program mode, two-sided color mode, and program-specified point size mode are all initially disabled.

The initial state of all 96 program parameter registers is (0,0,0,0).

The initial state of the 16 vertex attribute registers is (0,0,0,1) except in cases where a vertex attribute register aliases to a conventional GL transform mode vertex parameter in which case the initial state is the initial state of the respective aliased conventional vertex parameter.

The initial state of the 24 sets of matrix tracking state is NONE for the tracked matrix and IDENTITY_NV for the transformation of the tracked matrix.

The initial currently bound program is zero.

The client state required to implement the 16 vertex attribute arrays consists of 16 boolean values, 16 memory pointers, 16 integer stride values, 16 symbolic constants representing array types, and 16 integers representing values per element. Initially, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the

integers representing values per element are each four."

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

-- Section 3.3 "Points"

Change the first paragraph to read:

"When program vertex mode is disabled, the point size for rasterizing points is controlled with

```
void PointSize(float size);
```

size specifies the width or diameter of a point. The initial point size value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE. When vertex program mode is enabled, the point size for rasterizing points is determined as described in section 2.14.1.5."

-- Section 3.9 "Color Sum"

Change the first paragraph to read:

"At the beginning of color sum, a fragment has two RGBA colors: a primary color cpri (which texturing, if enabled, may have modified) and a secondary color csec. If vertex program mode is disabled, csec is defined by the lighting equations in section 2.13.1. If vertex program mode is enabled, csec is the fragment's secondary color, obtained by interpolating the COL1 (or BFC1 if the primitive is a polygon, the vertex program two-sided color mode is enabled, and the polygon is back-facing) vertex result register RGB components for the vertices making up the primitive; the alpha component of csec when program mode is enabled is always zero. The components of these two colors are summed to produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]."

-- Section 3.10 "Fog"

Change the initial sentences in the second paragraph to read:

"This factor f may be computed according to one of three equations:

```
f = exp(-d*c)                                   (3.24)
f = exp(-(d*c)^2)                               (3.25)
f = (e-c)/(e-s)                                 (3.26)
```

If vertex program mode is enabled, then c is the fragment's fog coordinate, obtained by interpolating the FOGC vertex result register values for the vertices making up the primitive. When vertex program mode is disabled, the c is the eye-coordinate distance from the eye, (0,0,0,1) in eye-coordinates, to the fragment center." ...

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

-- Section 5.1 "Evaluators"

Add the following lines to the end of table 5.1 (page 165):

target	k	values
MAP1_VERTEX_ATTRIB0_4_NV	4	x, y, z, w vertex attribute 0
MAP1_VERTEX_ATTRIB1_4_NV	4	x, y, z, w vertex attribute 1
MAP1_VERTEX_ATTRIB2_4_NV	4	x, y, z, w vertex attribute 2
MAP1_VERTEX_ATTRIB3_4_NV	4	x, y, z, w vertex attribute 3
MAP1_VERTEX_ATTRIB4_4_NV	4	x, y, z, w vertex attribute 4
MAP1_VERTEX_ATTRIB5_4_NV	4	x, y, z, w vertex attribute 5
MAP1_VERTEX_ATTRIB6_4_NV	4	x, y, z, w vertex attribute 6
MAP1_VERTEX_ATTRIB7_4_NV	4	x, y, z, w vertex attribute 7
MAP1_VERTEX_ATTRIB8_4_NV	4	x, y, z, w vertex attribute 8
MAP1_VERTEX_ATTRIB9_4_NV	4	x, y, z, w vertex attribute 9
MAP1_VERTEX_ATTRIB10_4_NV	4	x, y, z, w vertex attribute 10
MAP1_VERTEX_ATTRIB11_4_NV	4	x, y, z, w vertex attribute 11
MAP1_VERTEX_ATTRIB12_4_NV	4	x, y, z, w vertex attribute 12
MAP1_VERTEX_ATTRIB13_4_NV	4	x, y, z, w vertex attribute 13
MAP1_VERTEX_ATTRIB14_4_NV	4	x, y, z, w vertex attribute 14
MAP1_VERTEX_ATTRIB15_4_NV	4	x, y, z, w vertex attribute 15

Replace the four paragraphs on pages 167-168 that explain the operation of EvalCoord:

"EvalCoord operates differently depending on whether vertex program mode is enabled or not. We first discuss how EvalCoord operates when vertex program mode is disabled.

When one of the EvalCoord commands is issued and vertex program mode is disabled, all currently enabled maps (excluding the maps that correspond to vertex attributes, i.e. maps of the form MAPx_VERTEX_ATTRIBn_4_NV). ..."

Add a paragraph before the initial paragraph discussing AUTO_NORMAL:

"When one of the EvalCoord commands is issued and vertex program mode is enabled, the evaluation and the issuing of per-vertex parameter commands matches the discussion above, except that if any vertex attribute maps are enabled, the corresponding VertexAttribNV call for each enabled vertex attribute map is issued with the map's evaluated coordinates and the corresponding aliased per-vertex parameter map is ignored if it is also enabled, with one important difference. As is the case when vertex program mode is disabled, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise the current values are used). The order of the effective commands is immaterial, except that Vertex or VertexAttribNV(0, ...) (the commands that issue provoke vertex program execution) must be issued last. Use of evaluators has no effect on the current vertex attributes or conventional per-vertex parameters. If a vertex attribute map is disabled, but its corresponding conventional per-vertex parameter map is enabled, the conventional per-vertex parameter map is evaluated and issued as when vertex program mode is not enabled."

Replace the two paragraphs discussing AUTO_NORMAL with:

"Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled or if both MAP2_VERTEX_ATTRIB0_4_NV and vertex program mode are enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex (when program mode is disabled) or as vertex attribute 2 (when vertex program mode is enabled). Automatic normal generation is controlled with Enable and Disable with the symbolic constant AUTO_NORMAL. If automatic normal generation is disabled and vertex program mode is enabled, then vertex attribute 2 is evaluated as usual. If automatic normal generation and vertex program mode are disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a map corresponding to the normal per-vertex parameter (or vertex attribute 2 in program mode) are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used). For MAP_VERTEX3, let q=p. For MAP_VERTEX_4 or MAP2_VERTEX_ATTRBI0_4_NV, let q = (x/w, y/w, z/w) where (x,y,z,w)=p. Then let

m = (partial q / partial u) cross (partial q / partial v)

Then when vertex program mode is disabled, the generated analytic normal, n, is given by n=m/||m||. However, when vertex program mode is enabled, the generated analytic normal used for vertex attribute 2 is simply (mx,my,mz,1). In vertex program mode, the normalization of the generated analytic normal can be performed by the current vertex program."

Change the respective sentences of the last paragraph discussing required evaluator state to read:

"The state required for evaluators potentially consists of 9 conventional one-dimensional map specifications, 16 vertex attribute one-dimensional map specifications, 9 conventional two-dimensional map specifications, and 16 vertex attribute two-dimensional map specifications indicating which are enabled. ... All vertex coordinate maps produce the coordinates (0,0,0,1) (or the appropriate subset); all normal coordinate maps produce (0,0,1); RGBA maps produce (1,1,1,1); color index maps produce 1.0; texture coordinate maps produce (0,0,0,1); and vertex attribute maps produce (0,0,0,1). ... If any evaluation command is issued when none of MAPn_VERTEX_3, MAPn_VERTEX_4, or MAPn_VERTEX_ATTRIB0_NV (where n is the map dimension being evaluated) are enabled, nothing happens."

-- Section 5.4 "Display Lists"

Add to the list of commands not compiled into display lists in the third to the last paragraph:

"AreProgramsResidentNV, IsProgramNV, GenProgramsNV, DeleteProgramsNV, VertexAttribPointerNV"

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

-- Section 6.1.12 "Saving and Restoring State"

Only the enables and vertex array state introduced by this extension can be pushed and popped.

See the attribute column in table X.5 for determining what vertex program state can be pushed and popped with PushAttrib, PopAttrib, PushClientAttrib, and PopClientAttrib.

The new evaluator enables in table 6.22 can also be pushed and popped.

-- NEW Section 6.1.13 "Vertex Program Queries"

"The commands

```
  void GetProgramParameterfvNV(enum target, uint index,
                               enum pname, float *params);
  void GetProgramParameterdvNV(enum target, uint index,
                               enum pname, double *params);
```

obtain the current program parameters for the given program target and parameter index into the array params. target must be VERTEX_PROGRAM_NV. pname must be PROGRAM_PARAMETER_NV. The INVALID_VALUE error is generated if index is greater than 95. Each program parameter is an array of four values.

The command

```
  void GetProgramivNV(uint id, enum pname, int *params);
```

obtains program state named by pname for the program named id in the array params. pname must be one of PROGRAM_TARGET_NV, PROGRAM_LENGTH_NV, or PROGRAM_RESIDENT_NV. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetProgramStringNV(uint id, enum pname,
                          ubyte *program);
```

obtains the program string for program id. pname must be PROGRAM_STRING_NV. n ubytes are returned into the array program where n is the length of the program in ubytes. GetProgramivNV with PROGRAM_LENGTH_NV can be used to query the length of a program's string. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetTrackMatrixivNV(enum target, uint address,
                          enum pname, int *params);
```

obtains the matrix tracking state named by pname for the specified

address in the array params. target must be VERTEX_PROGRAM_NV. pname must be either TRACK_MATRIX_NV or TRACK_MATRIX_TRANSFORM_NV. The INVALID_VALUE error is generated if address is not divisible by four and is not less than 96.

The commands

```
void GetVertexAttribdvNV(uint index, enum pname, double *params);
void GetVertexAttribfvNV(uint index, enum pname, float *params);
void GetVertexAttribivNV(uint index, enum pname, int *params);
```

obtain the vertex attribute state named by pname for the vertex attribute numbered index. pname must be one of ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, ATTRIB_ARRAY_TYPE_NV, or CURRENT_ATTRIB_NV. Note that all the queries except CURRENT_ATTRIB_NV return client state. The INVALID_VALUE error is generated if index greater than 15 or equal to zero.

The command

```
void GetVertexAttribPointervNV(uint index,
                               enum pname, void **pointer);
```

obtains the pointer named pname in the array params for vertex attribute numbered index. pname must be ATTRIB_ARRAY_POINTER_NV. The INVALID_VALUE error is generated if index greater than 15.

The command

```
boolean IsProgramNV(uint id);
```

returns TRUE if program is the name of a program object. If program is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramNV returns FALSE. A name returned by GenProgramsNV but not yet loaded with a program is not the name of a program object."

-- NEW Section 6.1.14 "Querying Current Matrix State"

"Instead of providing distinct symbolic tokens for querying each matrix and matrix stack depth, the symbolic tokens CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV in conjunction with the GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev return the respective state of the current matrix given the current matrix mode.

Querying CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV is the only means for querying the matrix and matrix stack depth of the tracking matrices described in section 2.14.5."

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

Add the following rule:

"Rule X Vertex program and vertex state program instructions not relevant to the calculation of any result must have no effect on that result.

Rules X+1 Vertex program and vertex state program instructions relevant to the calculation of any result must always produce the identical result. In particular, the same instruction with the same source inputs must produce the identical result whether executed by a vertex program or a vertex state program.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between vertices transformed by conventional GL vertex transform mode and vertices transformed by vertex program mode. Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional GL vertex transform mode with vertex program mode for different rendering passes. However such algorithms will operate correctly if the algorithms limit themselves to a single mode of vertex transformation."

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Dependencies on EXT_vertex_weighting

If the EXT_vertex_weighting extension is not supported, there is no aliasing between vertex attribute 1 and the current vertex weight. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Dependencies on EXT_point_parameters

When EXT_point_parameters is supported, the amended discussion of point size determination should be further amended with the language from the EXT_point_parameters specification though the point parameters functionality only applies when vertex program mode is disabled.

Even if the EXT_point_parameters extension is not supported, the PSIZ vertex result register must operate as specified.

Dependencies on ARB_multitexture

ARB_multitexture is required to support NV_vertex_program and the value of MAX_TEXTURE_UNITS_ARB must be at least 2. If more than 8 texture units are supported, only the first 8 texture units can be assigned texture coordinates when vertex program mode is enabled. Texture units beyond 8 are implicitly disabled when vertex program mode is enabled.

Dependencies on EXT_fog_coord

If the EXT_fog_coord extension is not supported, there is no aliasing between vertex attribute 5 and the current fog coordinate. Replace the contents of the last three columns in row 5 of table

X.2 with dashes.

Even if the EXT_fog_coord extension is not supported, the FOGC vertex result register must operate as specified. Note that the FOGC vertex result register behaves identically to the EXT_fog_coord extension's FOG_COORDINATE_SOURCE_EXT being FOG_COORDINATE_EXT. This means that the functionality of EXT_fog_coord is required to implement NV_vertex_program even if the EXT_fog_coord extension is not supported.

If the EXT_fog_coord extension is supported, the state of FOG_COORDINATE_SOURCE_EXT only applies when vertex program mode is disabled and the discussion in section 3.10 is further amended by the discussion of FOG_COORDINATE_SOURCE_EXT in the EXT_fog_coord specification.

Dependencies on EXT_secondary_color

If the EXT_secondary_color extension is not supported, there is no aliasing between vertex attribute 4 and the current secondary color. Replace the contents of the last three columns in row 4 of table X.2 with dashes.

Even if the EXT_secondary_color extension is not supported, the COL1 and BFC1 vertex result registers must operate as specified. These vertex result registers are required to implement OpenGL 1.2's separate specular mode within a vertex program.

GLX Protocol

Forty-five new GL commands are added.

The following thirty-five rendering commands are sent to the sever as part of a glXRender request:

```
BindProgramNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          id

ExecuteProgramNV
    2           12+4*n          rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
                0x8621  n=4     GL_VERTEX_STATE_PROGRAM_NV
                else    n=0     command is erroneous
    4           CARD32          id
    4*n         LISTofFLOAT32   params

RequestResidentProgramsNV
    2           8+4*n           rendering command length
    2           ????            rendering command opcode
    4           INT32           n
    n*4         CARD32          programs
```

```
LoadProgramNV
    2           16+n+p          rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          id
    4           INT32           len
    n           LISTofCARD8     n
    p                           unused, p=pad(n)

ProgramParameter4fvNV
    2           32              rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           FLOAT32         params[0]
    4           FLOAT32         params[1]
    4           FLOAT32         params[2]
    4           FLOAT32         params[3]

ProgramParameter4dvNV
    2           44              rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    8           FLOAT64         params[0]
    8           FLOAT64         params[1]
    8           FLOAT64         params[2]
    8           FLOAT64         params[3]

ProgramParameters4fvNV
    2           16+16*n         rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           CARD32          n
    16*n        FLOAT32         params

ProgramParameters4dvNV
    2           16+32*n         rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           CARD32          n
    32*n        FLOAT64         params

TrackMatrixNV
    2           20              rendering command length
    2           ????            rendering command opcode
    4           ENUM            target
    4           CARD32          address
    4           ENUM            matrix
    4           ENUM            transform

VertexAttribPointerNV is an entirely client-side command

VertexAttrib1svNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2                           unused

VertexAttrib2svNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
```

```
VertexAttrib3svNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2                           unused

VertexAttrib4svNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2           INT16           v[3]

VertexAttrib1fvNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]

VertexAttrib2fvNV
    2           16              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]

VertexAttrib3fvNV
    2           20              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]

VertexAttrib4fvNV
    2           24              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]
    4           FLOAT32         v[3]

VertexAttrib1dvNV
    2           16              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]

VertexAttrib2dvNV
    2           24              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]

VertexAttrib3dvNV
    2           32              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]
```

```
VertexAttrib4dvNV
    2           40              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]
    8           FLOAT64         v[3]

VertexAttrib4ubvNV
    2           12              rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    1           CARD8           v[0]
    1           CARD8           v[1]
    1           CARD8           v[2]
    1           CARD8           v[3]

VertexAttribs1svNV
    2           12+2*n+p        rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    2*n         INT16           v
    p                           unused, p=pad(2*n)

VertexAttribs2svNV
    2           12+4*n          rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    4*n         INT16           v

VertexAttribs3svNV
    2           12+6*n+p        rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    6*n         INT16           v
    p                           unused, p=pad(6*n)

VertexAttribs4svNV
    2           12+8*n          rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    8*n         INT16           v

VertexAttribs1fvNV
    2           12+4*n          rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    4*n         FLOAT32         v

VertexAttribs2fvNV
    2           12+8*n          rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    8*n         FLOAT32         v

VertexAttribs3fvNV
    2           12+12*n         rendering command length
    2           ????            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    12*n        FLOAT32         v
```

```
    VertexAttribs4fvNV
        2           12+16*n         rendering command length
        2           ????            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        16*n        FLOAT32         v

    VertexAttribs1dvNV
        2           12+8*n          rendering command length
        2           ????            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        8*n         FLOAT64         v

    VertexAttribs2dvNV
        2           12+16*n         rendering command length
        2           ????            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        16*n        FLOAT64         v

    VertexAttribs3dvNV
        2           12+24*n         rendering command length
        2           ????            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        24*n        FLOAT64         v

    VertexAttribs4dvNV
        2           12+32*n         rendering command length
        2           ????            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        32*n        FLOAT64         v

    VertexAttribs4ubvNV
        2           12+4*n          rendering command length
        2           ????            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        4*n         CARD8           v
```

The remaining twelve commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
    AreProgramsResidentNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           ????            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           (n+p)/4         reply length
        4           BOOL32          return value
        20                          unused
        n           LISTofBOOL      programs
        p                           unused, p=pad(n)
```

```
DeleteProgramsNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4+n             request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
    n*4         LISTofCARD32    programs

GenProgramsNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           n               reply length
    24                          unused
    n*4         LISTofCARD322   programs

GetProgramParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

```
GetProgramParameterdvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params

GetProgramivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetProgramStringNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           (n+p)/4         reply length
    4                           unused
    4           CARD32          n
    16                          unused
    n           STRING          program
    p                           unused, p=pad(n)
```

```
GetTrackMatrixivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          address
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

Note that ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, and
ATTRIB_ARRAY_TYPE_NV may be queried by GetVertexAttribNV but
return client-side state.

GetVertexAttribdvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params
```

```
GetVertexAttribfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params

GetVertexAttribivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetVertexAttribPointervNV is an entirely client-side command

IsProgramNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           ????            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           0               reply length
    4           BOOL32          return value
    20                          unused
```

Errors

The error INVALID_VALUE is generated if VertexAttribNV is called where index is greater than 15.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 95.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where index is greater than 15.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where size is not one of 1, 2, 3, or 4.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where stride is negative.

The error INVALID_OPERATION is generated if VertexAttribPointerNV is called where type is UNSIGNED_BYTE and size is not 4.

The error INVALID_VALUE is generated if LoadProgramNV is used to load a program with an id of zero.

The error INVALID_OPERATION is generated if LoadProgramNV is used to load an id that is currently loaded with a program of a different program target.

The error INVALID_OPERATION is generated if the program passed to LoadProgramNV fails to load because it is not syntactically correct based on the specified target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target of VERTEX_PROGRAM_NV and the specified program fails to load because it does not write the HPOS register at least once. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target of VERTEX_STATE_PROGRAM_NV and the specified program fails to load because it does not write at least one program parameter register. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if the vertex program or vertex state program passed to LoadProgramNV fails to load because it contains more than 128 instructions. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

The error INVALID_OPERATION is generated if a program is loaded with LoadProgramNV for id when id is currently loaded with a program of a different target.

The error INVALID_OPERATION is generated if BindProgramNV attempts to bind to a program name that is not a vertex program (for example, if the program is a vertex state program).

The error INVALID_VALUE is generated if GenProgramsNV is called where n is negative.

The error INVALID_VALUE is generated if AreProgramsResidentNV is called and any of the queried programs are zero or do not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes a program that does not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes a program that is not a vertex state program.

The error INVALID_OPERATION is generated if Begin, RasterPos, or a command that performs an explicit Begin is called when vertex program mode is enabled and the currently bound vertex program writes program parameters that are currently being tracked.

The error INVALID_OPERATION is generated if ExecuteProgramNV is called and the vertex state program to execute writes program parameters that are currently being tracked.

The error INVALID_VALUE is generated if TrackMatrixNV has a target of VERTEX_PROGRAM_NV and attempts to track an address is not a multiple of four.

The error INVALID_VALUE is generated if GetProgramParameterNV is called to query an index greater than 95.

The error INVALID_VALUE is generated if GetVertexAttribNV is called to query an index greater than 15 or equal to zero.

The error INVALID_VALUE is generated if GetVertexAttribPointervNV is called to query an index greater than 15.

The error INVALID_OPERATION is generated if GetProgramivNV is called and the program named id does not exist.

The error INVALID_OPERATION is generated if GetProgramStringNV is called and the program named id does not exist.

The error INVALID_VALUE is generated if GetTrackMatrixivNV is called with an address that is not divisible by four and not less than 96.

The error INVALID_VALUE is generated if AreProgramsResidentNV, DeleteProgramsNV, GenProgramsNV, or RequestResidentProgramsNV are called where n is negative.

The error INVALID_VALUE is generated if LoadProgramNV is called where len is negative.

The error INVALID_VALUE is generated if ProgramParameters4dvNV or ProgramParameters4fvNV are called where count is negative.

The error INVALID_VALUE is generated if VertexAttribs{1,2,3,4}{d,f,s}vNV is called where count is negative.

New State

update table 6.22 (page 212) so that all the "9"s are "25"s because there are 9 conventional map targets and 16 vertex attribute map targets making a total of 25.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_PROGRAM_NV	B	IsEnabled	False	vertex program enable	2.10	enable
VERTEX_PROGRAM_POINT_SIZE_NV	B	IsEnabled	False	program-specified point size mode	2.14.1.5	enable
VERTEX_PROGRAM_TWO_SIDE_NV	B	IsEnabled	False	two-sided color mode	2.14.1.5	enable
PROGRAM_ERROR_POSITION_NV	Z	GetIntegerv	-1	last program error position	2.14.1.7	-
PROGRAM_PARAMETER_NV	96xR4	GetProgramParameterNV	(0,0,0,0)	program parameters	2.14.1.2	-
CURRENT_ATTRIB_NV	16xR4	GetVertexAttribNV but zero cannot be queried, aliased with per-vertex parameters	see 2.14.6	vertex attributes	2.14.1.1	current
TRACK_MATRIX_NV	24xZ8+	GetTrackMatrixivNV	NONE	track matrix	2.14.5	-
TRACK_MATRIX_TRANSFORM_NV	24xZ8+	GetTrackMatrixivNV	IDENTITY_NV	track matrix transform	2.14.5	-
VERTEX_PROGRAM_BINDING_NV	Z+	GetIntegerv	0	bound vertex program	2.14.1.8	-
VERTEX_ATTRIB_ARRAYn_NV	16xB	IsEnabled	False	vertex attrib array enable	2.14.3	vertex-array
ATTRIB_ARRAY_SIZE_NV	16xZ	GetVertexAttribNV	4	vertex attrib array size	2.14.3	vertex-array
ATTRIB_ARRAY_STRIDE_NV	16xZ+	GetVertexAttribNV	0	vertex attrib array stride	2.14.3	vertex-array
ATTRIB_ARRAY_TYPE_NV	16xZ6	GetVertexAttribNV	FLOAT	vertex attrib array type	2.14.3	vertex-array

Table X.5. New State Introduced by NV_vertex_program.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PROGRAM_TARGET_NV	Z2	GetProgramivNV	0	program target	6.1.13	-
PROGRAM_LENGTH_NV	Z+	GetProgramivNV	0	program length	6.1.13	-
PROGRAM_RESIDENT_NV	Z2	GetProgramivNV	False	program residency	6.1.13	-
PROGRAM_STRING_NV	ubxn	GetProgramStringNV	""	program string	6.1.13	-

Table X.6. Program Object State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	12xR4	-	(0,0,0,0)	temporary registers	2.14.1.4	-
-	15xR4	-	(0,0,0,1)	vertex result registers	2.14.1.4	-
	Z4	-	(0,0,0,0)	vertex program address register	2.14.1.3	-

Table X.7. Vertex Program Per-vertex Execution State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_MATRIX_STACK_DEPTH_NV	m*Z+	GetIntegerv	1	current stack depth	6.1.14	-
CURRENT_MATRIX_NV	m*n*xM^4	GetFloatv	Identity	current matrix	6.1.14	-

Table X.8. Current matrix state where m is the total number of matrices including texture matrices and tracking matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TRACK_MATRIX_STACK_DEPTH_NV	Z+	GetIntegerv	1	maximum tracking matrix stack depth	2.14.5	-
MAX_TRACK_MATRICES_NV	Z+	GetIntegerv	8 (not to exceed 32)	maximum number of tracking matrices	2.14.5	-

Table X.9. New Implementation-Dependent Values Introduced by NV_vertex_program.

Revision History

Version 1.1:

Added normalization example to Issues.

Fix explanation of EXP and ARL floor equivalence.

Clarify that vertex state programs fail if they load more than one vertex attribute (though only one is possible).

Version 1.2

Add GLX protocol for VertexAttrib4ubvNV and VertexAttribs4ubvNV

Add issue about TrackMatrixNV transform behavior with example

Fix the C code specifying VertexAttribsvNV

Version 1.3

Dropped support for INT typed vertex attrib arrays.

Clarify that when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is enabled, the vertex attrib 0 array command is executed last. However when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is disabled and the vertex array is enabled, the vertex array command is executed last.

Name

SGIS_generate_mipmap

Name Strings

GL_SGIS_generate_mipmap

Version

SGI Date: 1997/02/26 03:36:30 SGI Revision: 1.6
$Id: //sw/main/docs/OpenGL/specs/GL_SGIS_generate_mipmap.txt#2 $

Number

32

Dependencies

EXT_texture is required
EXT_texture3D affects the definition of this extension
EXT_texture_object affects the definition of this extension
SGIS_texture_lod affects the definition of this extension

Overview

This extension defines a mechanism by which OpenGL can derive the entire set of mipmap arrays when provided with only the base level array. Automatic mipmap generation is particularly useful when texture images are being provided as a video stream.

Issues

* How are edges handled?

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameter of TexParameteri, TexParameterf, TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

GENERATE_MIPMAP_SGIS 0x8191

Accepted by the <target> parameter of Hint, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

GENERATE_MIPMAP_HINT_SGIS 0x8192

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE

Table 3.7: Texture parameters and their values.

This extension introduces a side effect to the modification of the base level mipmap array. The side effect is enabled on a per-texture basis by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to GENERATE_MIPMAP_SGIS, and <param> set to TRUE (or <params> pointing to TRUE). It is disabled using the same call, with <param> set to FALSE, or <params> pointing to FALSE. If SGIS_texture_lod is supported, the base level array is the array number TEXTURE_BASE_LEVEL_SGIS. Otherwise the base level array is array zero.

If GENERATE_MIPMAP_SGIS is enabled, the side effect occurs whenever any change is made to the interior or edge image values of the base level texture array. The side effect is computation of a complete set of mipmap arrays, all derived from the modified base level array. Array levels BASE+1 through BASE+p are replaced with derived arrays, regardless of their previous contents. All other texture arrays, including the base array, are left unchanged by this mipmap computation.

The internal formats and border widths of the derived mipmap arrays all match those of the base array, and the dimensions of the derived arrays follow the requirements described in the Mipmapping section of the GL Specification. The result is that the set of mipmap arrays is

complete as defined by the GL Specification. The contents of the derived image arrays are computed by repeated, filtered reduction of the base level image array. This specification does not require any particular filter algorithm, though a simple 2x2 box filter is recommended as the default filter. Hint variable GENERATE_MIPMAP_HINT_SGIS can be changed from its default value of DONT_CARE to either FASTEST or NICEST, indicating to the implementation that either the fastest or highest quality filter operation is desired. These operations are not defined by this specification, however. The single hint value controls the filtering of all the textures, and is evaluated when the filtering operation takes place.

Automatic mipmap generation is available for texture targets TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT only.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, references to 3D texture mapping and to TEXTURE_3D_EXT in this document are invalid and should be ignored.

Dependencies on EXT_texture_object

If EXT_texture_object is implemented, the state value named

GENERATE_MIPMAP_SGIS

is added to the state vector of each texture object. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound textures have their GENERATE_MIPMAP_SGIS parameters restored to their pushed values.

Dependencies on SGIS_texture_lod

If SGIS_texture_lod is not supported, the base array level is always level zero. References in this document to TEXTURE_BASE_LEVEL_SGIS

should be ignored.

Errors

None

New State

Get Value	Get Command	Type	Value	Initial Attrib
GENERATE_MIPMAP_SGIS	GetTexParameteriv	B	FALSE	texture
GENERATE_MIPMAP_HINT_SGIS	GetIntegerv	Z3	DONT_CARE	hint

New Implementation Dependent State

None

Name

SGIS_texture_lod

Name Strings

GL_SGIS_texture_lod

Version

$Date: 1997/05/30 01:34:44 $ $Revision: 1.8 $

Number

24

Dependencies

EXT_texture is required
EXT_texture3D affects the definition of this extension
EXT_texture_object affects the definition of this extension
SGI_detail_texture affects the definition of this extension
SGI_sharpen_texture affects the definition of this extension

Overview

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.

Issues

* Should detail and sharpen texture operate when the level 0 image is not being used?

A: Sharpen yes, detail no.

* Should the shape of the mipmap array be determined by the dimensions of the level 0 array, regardless of the base level?

A: Yes, this is the better solution. Driving everything from the base level breaks the proxy query process, and allows mipmap arrays to be placed arbitrarily. The issues of requiring a level 0 array are partially overcome by the use of null-point loads, which avoid data transfer and, potentially, data storage allocation.

* With the arithmetic as it is, a linear filter might access an array past the limit specified by MAX_LEVEL or p. But the results of this access are not significant, because the blend will weight them as zero.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameter of TexParameteri, TexParameterf, TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

TEXTURE_MIN_LOD_SGIS	0x813A
TEXTURE_MAX_LOD_SGIS	0x813B
TEXTURE_BASE_LEVEL_SGIS	0x813C
TEXTURE_MAX_LEVEL_SGIS	0x813D

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer

Table 3.7: Texture parameters and their values.

Base Array

Although it is not explicitly stated, it is the clear intention of the OpenGL specification that texture minification filters NEAREST and LINEAR, and all texture magnification filters, be applied to image array zero. This extension introduces a parameter, BASE_LEVEL, that explicitly specifies which array level is used for these filter operations. Base level is specified for a specific texture by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_BASE_LEVEL_SGIS, and <param> set to (or <params> pointing to) the desired value. The error INVALID_VALUE is generated if the specified BASE_LEVEL is negative.

Level of Detail Clamping

The level of detail parameter LOD is defined in the first paragraph of Section 3.8.1 (Texture Minification) of the GL Specification, where it is represented by the Greek character lambda. This extension redefines the definition of LOD as follows:

```
LOD'(x,y) = log_base_2 (Q(x,y))


       /  MAX_LOD LOD' > MAX_LOD
LOD = (   LOD'          LOD' >= MIN_LOD and LOD' <= MAX_LOD
       \  MIN_LOD LOD' < MIN_LOD
        \ undefined     MIN_LOD > MAX_LOD
```

The variable Q in this definition represents the Greek character rho, as it is used in the OpenGL Specification. (Recall that Q is computed based on the dimensions of the BASE_LEVEL image array.) MIN_LOD is the value of the per-texture variable TEXTURE_MIN_LOD_SGIS, and MAX_LOD is the value of the per-texture variable TEXTURE_MAX_LOD_SGIS.

Initially TEXTURE_MIN_LOD_SGIS and TEXTURE_MAX_LOD_SGIS are -1000 and 1000 respectively, so they do not interfere with the normal operation of texture mapping. These values are respecified for a specific texture by calling TexParameteri, TexParemeterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_MIN_LOD_SGIS or TEXTURE_MAX_LOD_SGIS, and <param> set to (or <params> pointing to) the new value. It is not an error to specify a maximum LOD value that is less than the minimum LOD value, but the resulting LOD values are not defined.

LOD is clamped to the specified range prior to any use. Specifically, the mipmap image array selection described in the Mipmapping Subsection of the GL Specification is based on the clamped LOD value. Also, the determination of whether the minification or magnification filter is used is based on the clamped LOD.

Mipmap Completeness

The GL Specification describes a "complete" set of mipmap image arrays as array levels 0 through p, where p is a well defined function of the dimensions of the level 0 image. This extension modifies the notion of completeness: instead of requiring that all arrays 0 through p meet the requirements, only arrays 0 and arrays BASE_LEVEL through MAX_LEVEL (or p, whichever is smaller) must meet these requirements. The specification of BASE_LEVEL was described above. MAX_LEVEL is specified by calling TexParameteri, TexParemeterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_MAX_LEVEL_SGIS, and <param> set to (or <params> pointing to) the desired value. The error INVALID_VALUE is generated if the specified MAX_LEVEL is negative. If MAX_LEVEL is smaller than BASE_LEVEL, or if BASE_LEVEL is greater than p, the set of arrays is incomplete.

Array Selection

Magnification and non-mipmapped minification are always performed using only the BASE_LEVEL image array. If the minification filter is one that requires mipmapping, one or two array levels are selected using the equations in the table below, and the LOD value is clamped to a maximum value that insures that no array beyond

the limits specified by MAX_LEVEL and p is accessed.

Minification Filter	Maximum LOD	Array level(s)
NEAREST_MIPMAP_NEAREST	M + 0.4999	floor(B + 0.5)
LINEAR_MIPMAP_NEAREST	M + 0.4999	floor(B + 0.5)
NEAREST_MIPMAP_LINEAR	M	floor(B), floor(B)+1
LINEAR_MIPMAP_LINEAR	M	floor(B), floor(B)+1

where:

M = min(MAX_LEVEL,p) - BASE_LEVEL
B = BASE_LEVEL + LOD

For NEAREST_MIPMAP_NEAREST and LINEAR_MIPMAP_NEAREST the specified image array is filtered according to the rules for NEAREST or LINEAR respectively. For NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR both selected arrays are filtered according to the rules for NEAREST or LINEAR, respectively. The resulting values are then blended as described in the Mipmapping section of the OpenGL specification.

Additional Filters

Sharpen filters (described in SGIS_sharpen_texture) operate on array levels BASE_LEVEL and BASE_LEVEL+1. If the minimum of MAX_LEVEL and p is not greater than BASE_LEVEL, then sharpen texture reverts to a LINEAR magnification filter. Detail filters (described in SGIS_detail_texture) operate only when BASE_LEVEL is zero.

Texture Capacity

In Section 3.8 the OpenGL specification states:

"In order to allow the client to meaningfully query the maximum image array sizes that are supported, an implementation must not allow an image array of level one or greater to be created if a `complete' set of image arrays consistent with the requested array could not be supported."

Given this extension's redefinition of completeness, the above paragraph should be rewritten to indicate that all levels of the `complete' set of arrays must be supportable. E.g.

"In order to allow the client to meaningfully query the maximum image array sizes that are supported, an implementation must not allow an image array of level one or greater to be created if a `complete' set of image arrays (all levels 0 through p) consistent with the requested array could not be supported."

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, references to 3D texture mapping and to TEXTURE_3D_EXT in this document are invalid and should be ignored.

Dependencies on EXT_texture_object

If EXT_texture_object is implemented, the state values named

TEXTURE_MIN_LOD_SGIS
TEXTURE_MAX_LOD_SGIS
TEXTURE_BASE_LEVEL_SGIS
TEXTURE_MAX_LEVEL_SGIS

are added to the state vector of each texture object. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound textures have their LOD and LEVEL parameters restored to their pushed values.

Dependencies on SGIS_detail_texture

If SGIS_detail_texture is not supported, references to detail texture mapping in this document are invalid and should be ignored.

Dependencies on SGIS_sharpen_texture

If SGIS_sharpen_texture is not supported, references to sharpen texture mapping in this document are invalid and should be ignored.

Errors

INVALID_VALUE is generated if an attempt is made to set TEXTURE_BASE_LEVEL_SGIS or TEXTURE_MAX_LEVEL_SGIS to a negative value.

New State

Get Value	Get Command	Initial Type	Value	Attrib
TEXTURE_MIN_LOD_SGIS	GetTexParameterfv	n x R	-1000	texture
TEXTURE_MAX_LOD_SGIS	GetTexParameterfv	n x R	1000	texture
TEXTURE_BASE_LEVEL_SGIS	GetTexParameteriv	n x R	0	texture
TEXTURE_MAX_LEVEL_SGIS	GetTexParameteriv	n x R	1000	texture

New Implementation Dependent State

None

Name

SGIX_depth_texture

Name Strings

GL_SGIX_depth_texture

Version

$Date: 1997/02/26 03:36:29 $ $Revision: 1.5 $
$Id: //sw/main/docs/OpenGL/specs/GL_SGIX_depth_texture.txt#3 $

Number

63

Dependencies

EXT_texture is required
EXT_subtexture affects the definition of this extension
EXT_copy_texture affects the definition of this extension

Overview

This extension defines a new depth texture format. An important application of depth texture images is shadow casting, but separating this from the shadow extension allows for the potential use of depth textures in other applications such as image-based rendering or displacement mapping. This extension does not define new depth-texture environment functions, such as filtering or applying the depth values computed from a texture, but leaves this to other extensions, such as the shadow extension.

New Procedures and Functions

None

New Tokens

Accepted by the <components> parameters of TexImage1D and TexImage2D, and by the <internalformat> parameters of TexImage3DEXT, CopyTexImage1DEXT, and CopyTexImage2DEXT:

DEPTH_COMPONENT16_SGIX	0x81A5
DEPTH_COMPONENT24_SGIX	0x81A6
DEPTH_COMPONENT32_SGIX	0x81A7

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

XXX - lots

Notes:

* Defines DEPTH_COMPONENT as a new base internal format for textures. Defines 16, 24, and 32 bit specific internal formats for texture. Just as for the specific color internal formats, an implementation can choose whether to implement them or not.

* Texture commands that accept images from memory now allow the internal format to be DEPTH_COMPONENT or DEPTH_COMPONENT*_SGIX when the format of the image data is DEPTH_COMPONENT. Depth, not color pixel transfer operations are applied to depth images.

* Texture commands that accept images from the framebuffer now take their data from the depth buffer when the internal format is DEPTH_COMPONENT or DEPTH_COMPONENT*_SGIX, or when no internal format is specified, and the internal format of the target texture is DEPTH_COMPONENT or DEPTH_COMPONENT*_SGIX.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

EXT_texture3D is not required, but if it is not supported, the implementation must compute the R texture coordinate as if it were. If EXT_texture3D is not supported, references to TexImage3DEXT and TexSubImage3DEXT in this document are invalid and should be ignored.

Dependencies on EXT_subtexture

If EXT_subtexture is not supported, references to TexSubImage1DEXT, TexSubImage2DEXT, and TexSubImage3DEXT in this document are invalid and should be ignored. If EXT_subtexture is supported, the operations of these three commands are affected by this extension.

Dependencies on EXT_copy_texture

If EXT_copy_texture is not supported, references to CopyTexImage1DEXT and CopyTexImage2DEXT in this document are invalid and should be ignored. If EXT_copy_texture is supported, the operations of these

two commands, and of CopyTexSubImage1DEXT, CopyTexSubImage2DEXT, and CopyTexSubImage3DEXT are affected by this extension.

Errors

INVALID_OPERATION is generated if TexImage1D or TexImage2D parameter <format> is DEPTH_COMPONENT and parameter <components> is not DEPTH_COMPONENT, DEPTH_COMPONENT16_SGI, DEPTH_COMPONENT24_SGI, or DEPTH_COMPONENT32_SGI.

INVALID_OPERATION is generated if TexImage3DEXT parameter <format> is DEPTH_COMPONENT and parameter <internalformat> is not DEPTH_COMPONENT, DEPTH_COMPONENT16_SGI, DEPTH_COMPONENT24_SGI, or DEPTH_COMPONENT32_SGI.

INVALID_OPERATION is generated if CopyTexImage1DEXT or CopyTexImage2DEXT parameter <internalformat> is DEPTH_COMPONENT, DEPTH_COMPONENT16_SGI, DEPTH_COMPONENT24_SGI, or DEPTH_COMPONENT32_SGI, and there is no depth buffer.

New State

None

New Implementation Dependent State

None

Name

SGIX_shadow

Name Strings

GL_SGIX_shadow

Version

$Date: 1997/08/27 19:54:45 $ $Revision: 1.15 $
$Id: //sw/main/docs/OpenGL/specs/GL_SGIX_shadow.txt#4 $

Number

34

Dependencies

None.

Overview

This extension defines two new operations to be performed
on texture values before they are passed on to the filtering
subsystem. These operations perform either a <= or >= test
on the value from texture memory and the iterated R value,
and return 1.0 or 0.0 if the test passes or fails, respectively.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameter of TexParameterf, TexParameteri,
TexParameterfv, TexParameteriv, GetTexParameterfv, and
GetTexParameteriv, with the <pname> parameter of TRUE or FALSE:

TEXTURE_COMPARE_SGIX

Accepted by the <pname> parameter of TexParameterf, TexParameteri,
TexParameterfv, TexParameteriv, GetTexParameterfv, and
GetTexParameteriv:

TEXTURE_COMPARE_OPERATOR_SGIX

Accepted by the <param> parameter of TexParameterf and
TexParameteri, and by the <params> parameter of TexParameterfv
and TexParameteriv, when their <pname> parameter is
TEXTURE_COMPARE_OPERATOR_SGIX:

TEXTURE_LEQUAL_R_SGIX
TEXTURE_GEQUAL_R_SGIX

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

XXX - lots

GL Specification Table 3.8 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS, LINEAR_CLIPMAP_LINEAR_SGIX
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS,
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE
TEXTURE_CLIPMAP_OFFSET_SGIX	2 floats	any 2 values
TEXTURE_COMPARE_SGIX	boolean	TRUE or FALSE
TEXTURE_COMPARE_OPERATOR_SGIX	integer	TEXTURE_LEQUAL_R_SGIX, TEXTURE_GEQUAL_R_SGIX

Table 3.8: Texture parameters and their values.

Notes:

* Two new texture operators are defined which alter the sampled texture values before they are filtered. These operators are defined only for textures with internal format DEPTH_COMPONENT or DEPTH_COMPONENTS*_SGI.

* The new operators compare the sample texel value to the value of the third texture coordinate, R. The texture components are treated as though they range from 0.0 through 1.0. The value of the test is zero if the test fails, and one if it passes.

* The test for operator TEXTURE_LEQUAL_R_SGIX passes if the texel value is less than or equal to R. The test for operator

TEXTURE_GEQUAL_R_SGIX passes if the texel value is greater than or equal to R.

* The modified texels (with value 0.0 or 1.0 depending on the test result) are treated as if the texture internal format were LUMINANCE.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

The texture compare operator is queried by calling GetTexParameteriv and GetTexParameterfv with <pname> set to TEXTURE_COMPARE_OPERATOR_SGIX. Texture compare enable/disable state is queried by calling GetTexParameteriv or GetTexParameterif with <pname> TEXTURE_COMPARE_SGIX.

Additions to the GLX Specification

None

Errors

INVALID_OPERATION is generated if TexParameter[if] parameter <pname> is TEXTURE_COMPARE_OPERATOR_SGIX and parameter <param> is not TEXTURE_LEQUAL_R_SGIX,or TEXTURE_GEQUAL_R_SGIX.

New State

Get Value	Get Command	Type	Initial Value	Attribute
TEXTURE_COMPARE_SGIX	GetTexParameter[if]v	B	False	texture
TEXTURE_COMPARE_OPERATOR_SGIX	GetTexParameter[if]v	Z_2	TEXTURE_LEQUAL_R_SGIX	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

The specification is unclear if the R texture coordinate is clamped to the range [0,1]. NVIDIA hardware supporting this extension does clamp the R texture coordinate to the range [0,1] on a per-fragment basis.

The behavior of the NV_register_combiners SIGNED_NEGATE_NV mapping mode is undefined when used to map the initial value of a texture register corresponding to an enabled texture with a base internal format of GL_DEPTH_COMPONENT and a true TEXTURE_COMPARE_SGIX mode when multiple enabled textures have different values for TEXTURE_COMPARE_OPERATOR_SGIX. . Values subsequently assigned

```
to such registers and then mapped with SIGNED_NEGATIE_NV operate
as expected.
```

Name

WGL_ARB_buffer_region

Name Strings

WGL_ARB_buffer_region

Status

Complete. Approved by ARB on 12/8/1999

Version

Last Modified Date: December 10, 2000
Intergraph Revision 1.0

Number

ARB Extension #4

Dependencies

The extension is written against the OpenGL 1.2.1 Specification although it should work on any previous OpenGL specification.

The WGL_EXT_extensions_string extension is required.

Overview

The buffer region extension is a mechanism that allows an area of an OpenGL window to be saved in off-screen memory for quick restores. The off-screen memory can either be frame buffer memory or system memory, although frame buffer memory might offer optimal performance.

A buffer region can be created for the front color, back color, depth, and/or stencil buffer. Multiple buffer regions for the same buffer type can exist.

IP Status

None

Issues

1. Do we need the glBufferRegionEnabled call that is in the Kinetix extensions?

   The reason behind this function was so that a single driver could be used on adapters with various amounts of memory -- the extension would always be present but its use would depend on a separate call. The same functionality could be achieved by not advertising this extension or always returning a value of NULL from wglCreateBufferRegionARB.

2. Should the width/height be specified on the create.

   Because applications create regions that are not used, it would be better to leave the size as a parameter on the save.

3. Should information be added to the create to allow for layer support?

   Layer support has been added.

4. Which DC gets used for buffer region operations?

   The DC that was allocated on the CreateBufferRegionARB call is saved and used for subsequent save and restore operations. It must remain valid during the life of the buffer region. This is analogous to the RC method for handling the DC.

5. Does the driver do a flush before the save and restore?

   In keeping with the same paradigm as SwapBuffers, a flush will be made by the driver for the RC bound to the calling thread before the save and restore operations.

6. Which coordinate system is used?

   The KTX_buffer_region and WIN_swap_hint extensions specify the (x,y) origin as the lower left corner of the rectangle. This extension adopts the same philosophy.

New Procedures and Functions

```
HANDLE wglCreateBufferRegionARB(HDC hDC,
                                int iLayerPlane,
                                UINT uType)

VOID wglDeleteBufferRegionARB(HANDLE hRegion)

BOOL wglSaveBufferRegionARB(HANDLE hRegion,
                            int x,
                            int y,
                            int width,
                            int height)

BOOL wglRestoreBufferRegionARB(HANDLE hRegion,
                               int x,
                               int y,
                               int width,
                               int height,
                               int xSrc,
                               int ySrc)
```

New Tokens

Accepted by the <uType> parameter of wglCreateBufferRegionARB is the bitwise OR of any of the following values:

```
WGL_FRONT_COLOR_BUFFER_BIT_ARB          0x00000001
WGL_BACK_COLOR_BUFFER_BIT_ARB           0x00000002
WGL_DEPTH_BUFFER_BIT_ARB                0x00000004
WGL_STENCIL_BUFFER_BIT_ARB              0x00000008
```

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

None

Additions to the GLX Specification

None

GLX Protocol

None

Additions to the WGL Specification

A buffer region can be created with wglCreateBufferRegionARB which returns a handle associated with the buffer region.

```
HANDLE wglCreateBufferRegionARB(HDC hDC,
                                INT iLayerPlane,
                                UINT uType)
```

<hDC> specifies a device context for the device on which the buffer region is created. <iLayerPlane> specifies the layer. Positive values identify overlay planes, negative values identify underlay planes. A value of 0 identifies the main plane.

<uType> is a bitwise OR of any of the following values indicating which buffers can be saved or restored. Multiple bits can be set and may result in better performance if multiple buffers are saved or restored.

```
    WGL_FRONT_COLOR_BUFFER_BIT_ARB
    WGL_BACK_COLOR_BUFFER_BIT_ARB
    WGL_DEPTH_BUFFER_BIT_ARB
    WGL_STENCIL_BUFFER_BIT_ARB
```

For stereo windows, WGL_FRONT_COLOR_BUFFER_BIT_ARB implies both the left and right front buffers, and WGL_BACK_COLOR_BUFFER_BIT_ARB implies both the left and right back buffers.

When wglCreateBufferRegionARB fails to create a buffer region, a value of NULL is returned. To get extended error information, call GetLastError.

Image, depth, and stencil data can be saved into the buffer region by calling wglSaveBufferRegionARB.

```
  BOOL wglSaveBufferRegionARB(HANDLE hRegion,
                              int x,
                              int y,
                              int width,
                              int height)
```

<hRegion> is a handle to a buffer region previously created with wglCreateBufferRegionARB. The DC specified when the region was created is used as the device context specifying the window.

<x> and <y> specify the window position for the source rectangle. <width> and <height> specify the width and height of the source rectangle. Data outside the window for the specified rectangle is undefined. The OpenGL coordinate system is used for specifying the rectangle (<x> and <y> specify the lower-left corner of the rectangle).

If an RC is current to the calling thread, a flush will occur before the save operation.

The saved buffer region area can be freed by calling wglSaveBufferRegionARB with <width> or <height> set to a value of 0.

If the call to wglSaveBufferRegionARB is successful, a value of TRUE is returned. Otherwise, a value of FALSE is returned. To get extended error information, call GetLastError.

A previously saved region can be restored (multiple times) with the wglRestoreBufferRegionARB function.

```
BOOL wglRestoreBufferRegionARB(HANDLE hRegion,
                               int x,
                               int y,
                               int width,
                               int height,
                               int xSrc,
                               int ySrc)
```

<hRegion> is a handle to a buffer region previously created with wglCreateBufferRegionARB. The DC specified when the region was created is used as the device context specifying the window.

<x> and <y> specify the window position for the destination rectangle. <width> and <height> specify the width and height of the destination rectangle. The OpenGL coordinate system is used for specifying the rectangle (<x> and <y> specify the lower-left corner of the rectangle).

<xSrc> and <ySrc> specify the position in the buffer region for the source of the data. Any portion of the rectangle outside of the saved region is ignored.

If an RC is current to the calling thread, a flush will occur before the restore operation.

If the call to wglRestoreBufferRegionARB is successful, a value of TRUE is returned. Otherwise, a value of FALSE is returned. To get extended error information, call GetLastError.

The buffer region can be deleted with wglDeleteBufferRegionARB.

```
VOID wglDeleteBufferRegionARB(HANDLE hRegion)
```

<hRegion> is a handle to a buffer region previously created with wglCreateBufferRegionARB. Any saved data associated with <hRegion> is discarded. The DC used to create the region must still be valid for the delete to work.

Dependencies on WGL_EXT_extensions_string

Because there is no way to extend wgl, these calls are defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this extension is a WGL extension, it is not included in the GL_EXTENSIONS string. Its existence can be determined with the WGL_EXT_extensions_string extension.

Errors

ERROR_NO_SYSTEM_RESOURCES is generated if memory cannot be allocated for storing the saved data.

ERROR_INVALID_HANDLE is generated if <hRegion> is not a valid handle that was previously returned by wglCreateBufferRegionARB.

ERROR_INVALID_DATA is generated if <uType> is zero or includes an undefined bit.

ERROR_INVALID_DATA is generated if <width> or <height> is negative.

New State

None

New Implementation Dependent State

None

Conformance Test

1. Clear the window to blue.
2. Save an area of the window using wglSaveBufferRegionARB.
3. Clear the window to red.
4. Restore the area of the window using wglRestoreBufferRegionARB.
5. Verify that the area was restored.
6. Repeat for the depth buffer.
7. Repeat for the stencil buffer.
8. Repeat for image and depth buffer.

Revision History

12/10/99 1.0 ARB extension - based on the wgl_buffer_region extension.

Name

WGL_ARB_extensions_string

Name Strings

WGL_ARB_extensions_string

Status

Complete. Approved by ARB on March 15, 2000

Version

Last Modified Date: March 22, 2000
Author Revision: 1.0

Number

ARB Extension #8

Dependencies

None

Overview

This extension provides a way for applications to determine which WGL extensions are supported by a device. This is the foundation upon which other WGL extensions are built.

IP Status

No issues.

Issues

1. Note that extensions that were previously advertised via glGetString (e.g., the swap interval extension) should continue to be advertised there so existing applications don't break. They should also be advertised via wglGetExtensionsStringARB so new applications can make one call to find out which WGL extensions are supported.

2. Should this function take an hdc? It seems like a good idea. At some point MS may want to incorporate this into OpenGL32. If they do this and and they want to support more than one ICD, then an HDC would be needed.

New Procedures and Functions

```
const char *wglGetExtensionsStringARB(HDC hdc);
```

New Tokens

None

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the WGL Specification

Advertising WGL Extensions

Applications should call wglGetProcAddress to see whether or not wglGetExtensionsStringARB is supported. If it is supported then it can be used to determine which WGL extensions are supported by the device.

```
const char *wglGetExtensionsStringARB(HDC hdc);
```

<hdc> device context to query extensions for

If the function succeeds, it returns a list of supported extensions to WGL. Although the contents of the string is implementation specific, the string will be NULL terminated and will contain a space-separated list of extension names. (The extension names themselves do not contain spaces.) If there are no extensions then the empty string is returned.

If <hdc> does not indicate a valid device context then the function fails and the error ERROR_DC_NOT_FOUND is generated. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

New State

None

New Implementation Dependent State

None

Revision History

Changes from EXT_extension_string:

Added hdc parameter to facilitate moving this function into OPENGL32
Added WGL to name to avoid name collisions with GL and GLX

Name

WGL_ARB_pbuffer

Name Strings

WGL_ARB_pbuffer

Status

Complete. Approved by ARB on March 15, 2000

Version

Last Modified Date: 03/22/2000
Author Revision: 1.0

Based on: WGL_EXT_pbuffer specification
Date: 4/21/1999 Version 1.8

Number

ARB Extension #11

Dependencies

WGL_ARB_extensions_string is required.
WGL_ARB_pixel_format is required.
WGL_ARB_make_current_read affects the definition of this extension.

Overview

This extension defines pixel buffers (pbuffer for short). Pbuffers are additional non-visible rendering buffers for an OpenGL renderer. Pbuffers are equivalent to a window that has the same pixel format descriptor with the following exceptions:

1. There is no rendering to a pbuffer by GDI.

2. The pixel format descriptors used for a pbuffer can only be those that are supported by the ICD. Generic formats are not valid.

3. The allocation of a pbuffer can fail if there are insufficient resources (i.e., all the pbuffer memory has been allocated).

4. The pixel buffer might be lost if a display mode change occurs. A query is provided that can be called after a display mode change to determine the state of the pixel buffer.

The intent of the pbuffer semantics is to enable implementations to allocate pbuffers in non-visible frame buffer memory. These pbuffers are intended to be "static" resources in that a program will typically allocate them only once rather than as a part of its rendering loop. (Pbuffers should be deallocated when the program is no longer using them -- for example, if the program is iconified.)

The frame buffer resources that are associated with a pbuffer are also static and are deallocated when the pbuffer is destroyed or possibly when a display mode change occurs.

IP Status

TBD

Issues

1. Should the OPTIMUM width and heights and PBUFFER_LARGEST_ARB be taken out of the spec since they may be misleading or hard for some implementations to support?

   PBUFFER_LARGEST_ARB has been left in the extension. It was originally requested by an application. The OPTIMUM queries have been removed to match the GLX pixel buffer specification.

New Procedures and Functions

```
DECLARE_HANDLE(HPBUFFERARB);

HPBUFFERARB wglCreatePbufferARB(HDC hDC,
                                int iPixelFormat,
                                int iWidth,
                                int iHeight,
                                const int *piAttribList);

HDC wglGetPbufferDCARB(HPBUFFERARB hPbuffer);

int wglReleasePbufferDCARB(HPBUFFERARB hPbuffer,
                           HDC hDC);

BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);

BOOL wglQueryPbufferARB(HPBUFFERARB hPbuffer,
                        int iAttribute,
                        int *piValue);
```

New Tokens

Accepted by the <attribute> parameter of wglChoosePixelFormatEXT:

WGL_DRAW_TO_PBUFFER_ARB	0x202D

Accepted by the <attribute> parameter of wglGetPixelFormatAttribivEXT, and wglGetPixelFormatAttribfvEXT:

WGL_DRAW_TO_PBUFFER_ARB	0x202D
WGL_MAX_PBUFFER_PIXELS_ARB	0x202E
WGL_MAX_PBUFFER_WIDTH_ARB	0x202F
WGL_MAX_PBUFFER_HEIGHT_ARB	0x2030

Accepted by the <piAttribList> parameter of wglCreatePbufferARB:

WGL_PBUFFER_LARGEST_ARB	0x2033

Accepted by the <iAttribute> parameter of wglQueryPbufferARB:

WGL_PBUFFER_WIDTH_ARB	0x2034
WGL_PBUFFER_HEIGHT_ARB	0x2035
WGL_PBUFFER_LOST_ARB	0x2036

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

This specification is written for WGL.

GLX Protocol

This specification is written for WGL.

Additions to the WGL Specification

A pixel buffer (pbuffer) can be created with wglCreatePbufferARB which returns a handle associated with the pbuffer.

```
HPBUFFERARB wglCreatePbufferARB(HDC hDC,
                                int iPixelFormat,
                                int iWidth,
                                int iHeight,
                                const int *piAttribList);
```

<hDC> specifies a device context for the device on which the pbuffer is created. <iPixelFormat> specifies a non-generic pixel format descriptor index. Support for pbuffers may be restricted to specific pixel formats. Use wglGetPixelFormatAttribivEXT or wglGetPixelFormatAttribfvEXT to query the WGL_DRAW_TO_PBUFFER_ARB attribute to determine which pixel formats support the creation of pbuffers.

<iWidth> and <iHeight> specify the pixel width and height of the rectangular pbuffer.

<piAttribList> is a list of attributes {type, value} pairs containing integer attribute values. All of the attributes in the <piAttribList> are followed by the corresponding required value. The list is terminated with a value of 0.

The following attributes are supported by wglCreatePbufferARB:

WGL_PBUFFER_LARGEST_ARB	If this attribute is set to a non-zero value, the largest available pbuffer is allocated when the allocation of the pbuffer would otherwise fail due to insufficient resources. The width or height of the allocated pbuffer never exceeds <iWidth> and <iHeight>, respectively. Use wglQueryPbufferARB to retrieve the dimensions of the allocated pbuffer.

The resulting pbuffer will contain color buffers and ancillary buffers as specified by <iPixelFormat>. Note that pbuffers use framebuffer resources so applications should consider deallocating them when they are not in use.

It is possible to create a pbuffer with back buffers and to swap the front and back buffers by calling wglSwapLayerBuffers. The contents of the back buffers after the swap depends on the <iPixelFormat>. (Pbuffers are the same as windows in this respect.)

When wglCreatePbufferARB fails to create a pbuffer, NULL is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_PIXEL_FORMAT	Pixel format is not valid.
ERROR_NO_SYSTEM_RESOURCES	Insufficient resources exist.
ERROR_INVALID_DATA	<iWidth> or <iHeight> is negative or zero.
ERROR_INVALID_DATA	<piAttribList> is not a valid attribute.

To create a device context for the pbuffer, call

```
HDC wglGetPbufferDCARB(HPBUFFERARB hPbuffer);
```

where <hPbuffer> is a handle returned from a previous call to wglCreatePbufferARB. A device context is returned by wglGetPbufferDCARB which can be used to associate a rendering context with the pbuffer. Any rendering context created with a wglCreateContext that is "compatible" with the <iPixelFormat> may be used to render into the pbuffer. (See the description of

wglCreateContext, wglMakeCurrent, and wglMakeCurrentReadEXT for a definition of "compatible".)

When wglGetPbufferDCARB fails, NULL is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.

To release a device context obtained from a previous call to wglGetPbufferDCARB, call

```
int wglReleasePbufferDCARB(HPBUFFERARB hPbuffer,
                           HDC hDC);
```

If the return value is a value of 1, the device context was released. If the device context was not released, the return value is 0. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.
ERROR_DC_NOT_FOUND	<hDC> is not a valid DC.

A pbuffer is destroyed by calling

```
BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);
```

The pbuffer is destroyed once it is no longer current to any rendering context. When a pbuffer is destroyed, any memory resources that are attached to it are freed and its handle is no longer valid.

If wglDestroyPbufferARB fails, FALSE is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.

To query the maximum width, height, or number of pixels in any given pbuffer for a specific pixel format, use wglGetPixelFormatAttribivEXT or wglGetPixelFormatAttribfvEXT with <attribute> set to one of WGL_MAX_PBUFFER_WIDTH_ARB, WGL_MAX_PBUFFER_HEIGHT_ARB, or WGL_MAX_PBUFFER_PIXELS_ARB.

WGL_MAX_PBUFFER_WIDTH_ARB and WGL_MAX_PBUFFER_HEIGHT_ARB indicate the maximum width and height that can be passed into wglCreatePbufferARB and WGL_MAX_PBUFFER_PIXELS_ARB indicates the maximum number of pixels (width x height) for a pbuffer. Note that an implementation may return a value for WGL_MAX_PBUFFER_PIXELS_ARB that is less than the maximum width times the maximum height. Also, the value for WGL_MAX_PBUFFER_PIXELS_ARB is static and assumes that no other pbuffers are contending for the framebuffer memory. Thus it may not be possible to allocate a pbuffer of the size given by WGL_MAX_PBUFFER_PIXELS_ARB.

To query an attribute associated with a specific pbuffer, call

```
BOOL wglQueryPbufferARB(HPBUFFERARB hPbuffer,
                        int iAttribute,
                        int *piValue);
```

with <hPbuffer> set to a previously returned pbuffer handle. <iAttribute> must be set to one of WGL_PBUFFER_WIDTH_ARB, WGL_PBUFFER_HEIGHT_ARB, or WGL_PBUFFER_LOST_ARB.

The WGL_PBUFFER_LOST_ARB query can be used to determine if the pixel buffer memory was lost due to a display mode change. A value of TRUE is returned in <iAttribute> if the display mode change lost the memory for the pixel buffer. It is not an error to render to a pixel buffer in this state, but the effect of rendering to it is the same as if the pixel buffer was destroyed: the context state will be updated, but the values of the returned pixels are undefined. The pixel buffer must be destroyed and recreated if the pixel buffer memory has been lost. A value of FALSE is returned to indicate that the contents of the pixel buffer are unaffected by the display mode change.

If wglQueryPbufferARB fails, FALSE is returned. To get extended error information, call GetLastError. Possible errors are as follows:

ERROR_INVALID_HANDLE	<hPbuffer> is not a valid handle.
ERROR_INVALID_DATA	<iAttribute> is not a valid attribute.

Dependencies on WGL_ARB_pixel_format

The WGL_ARB_pixel_format extension must be used to determine a pixel format that can be used to create the pixel buffer.

Dependencies on WGL_ARB_extensions_string

Because there is no way to extend wgl, these calls are defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this extension is a WGL extension, it is not included in the GL_EXTENSIONS string. Its existence can be determined with the WGL_ARB_extensions_string extension.

New State

None

New Implementation Dependent State

None

Conformance Testing

All of the current conformance tests can be run on a pixel buffer to validate its conformance. The only change to the conformance tests would be to create a context for the pixel buffer.

Revision History

```
    12/16/1999  0.1
        - First ARB draft based on the EXT specification.

    02/28/2000  0.2
        - Added a query for a damaged pixel buffer due to a display
          mode change.

    03/15/2000  0.3
        - Changed the lost definition of a pixel buffer.
        - Removed the OPTIMAL size queries.
        - Added a dependency on WGL_ARB_pixel_format.

    03/22/2000  1.0
        - Changed "mode change" to "display mode change".
        - Added the condition that the resources associated with a
          pbuffer may be lost due to a display mode change.
        - Fixed issue 1 to address the OPTIMUM values.
        - Added the declaration of HPBUFFERARB in the Procedures and
          Functions section.
        - Changed the wording of "undamaged" to "unaffected"
        - Approved by ARB: 10-0-0.
```

Name

WGL_ARB_pixel_format

Name Strings

WGL_ARB_pixel_format

Status

Complete. Approved by ARB on 3/15/2000.

Version

Last Modified Date: March 22, 2000
Author Revision: 1.0

Number

ARB Extension #9

Dependencies

WGL_ARB_extensions_string is required.

Overview

This extension adds functions to query pixel format attributes and to choose from the list of supported pixel formats.

These functions treat pixel formats as opaque types: attributes are specified by name rather than by accessing them directly as fields in a structure. Thus the list of attributes can be easily extended.

Attribute names are defined which correspond to all of the values in the PIXELFORMATDESCRIPTOR and LAYERPLANEDESCRIPTOR data structures. Additionally this interface allows pixel formats to be supported which have attributes that cannot be represented using the standard pixel format functions, i.e. DescribePixelFormat, DescribeLayerPlane, ChoosePixelFormat, SetPixelFormat, and GetPixelFormat.

IP Status

No issues.

Issues and Notes

1. No provision is made to support extended pixel format attributes in metafiles.
2. Should the transparent value pixel format attribute have separate red, green and blue values? Yes.
3. What data type should the transparent value be? This is no longer an issue since the transparent value is no longer a packed pixel value (it has separate r,g,b,a and index values).
4. Should we add DONT_CARE values for some of the pixel format attributes? No we should just ignore attributes that aren't specified in the list

passed to wglChoosePixelFormatARB.

5. Should wglGetPixelFormatAttrib*vARB ignore the <iLayerPlane> parameter when the attribute specified only applies to the main planes (e.g., when the attribute is set to WGL_NUMBER_OVERLAYS) or should it require <iLayerPlane> to be set to zero? It will just ignore the parameter. This allows these attributes to be queried at the same time as attributes of the overlay planes.
6. Should wglGetPixelFormatAttribivARB convert floating point values to fixed point? No, wglChoosePixelFormatARB needs a way to accept floating point values. pfAttribFList accomplishes this.
7. Should wglChoosePixelFormatARB take an <iLayerPlane> parameter? Typically <iLayerPlane> would be set to zero and a pixel format would be selected based on the attributes of the main plane, so there is no <iLayerPlane> parameter. This should be OK; applications won't typically select a pixel format on the basis of overlay attributes. They can always call wglGetPixelFormatAttrib*vARB to get a pixel format that has the desired overlay values.
8. Application programmers must check to see if a particular extension is supported before using any pixel format attributes associated with the extension. For example, if WGL_ARB_pbuffer is not supported then it is an error to specify WGL_DRAW_TO_PBUFFER_ARB in the attribute list to wglGetPixelFormatAttrib*vARB or wglChoosePixelFormatARB.
9. Should WGLChoosePixelFormatARB consider pixel formats at other display depths? It would be useful to have an argument to WGLChoosePixelFormatARB indicating what display depth should be used. However, there is no good way to implement this in the ICD since pixel format handles are sequential indices and the pixel format for index n differs depending on the display mode.
10. Should we allow non-displayable pixel formats for pbuffers? Yes, although many (most?) implementations will use displayable pixel formats for pbuffers, this is a useful feature and the spec should allow for it.
11. Should we create all new calls for pixel formats, specifically should we introduce SetPixelFormatARB? No, this doesn't offer any value over the existing SetPixelFormat call.
12. Should we add support for triple buffering? No, triple buffering needs to be covered by a separate extension.

New Procedures and Functions

```
BOOL wglGetPixelFormatAttribivARB(HDC hdc,
                                  int iPixelFormat,
                                  int iLayerPlane,
                                  UINT nAttributes,
                                  const int *piAttributes,
                                  int *piValues);

BOOL wglGetPixelFormatAttribfvARB(HDC hdc,
                                  int iPixelFormat,
                                  int iLayerPlane,
                                  UINT nAttributes,
                                  const int *piAttributes,
                                  FLOAT *pfValues);

BOOL wglChoosePixelFormatARB(HDC hdc,
                             const int *piAttribIList,
                             const FLOAT *pfAttribFList,
```

```
                                UINT nMaxFormats,
                                int *piFormats,
                                UINT *nNumFormats);
```

New Tokens

Accepted in the <piAttributes> parameter array of wglGetPixelFormatAttribivARB, and wglGetPixelFormatAttribfvARB, and as a type in the <piAttribIList> and <pfAttribFList> parameter arrays of wglChoosePixelFormatARB:

WGL_NUMBER_PIXEL_FORMATS_ARB	0x2000
WGL_DRAW_TO_WINDOW_ARB	0x2001
WGL_DRAW_TO_BITMAP_ARB	0x2002
WGL_ACCELERATION_ARB	0x2003
WGL_NEED_PALETTE_ARB	0x2004
WGL_NEED_SYSTEM_PALETTE_ARB	0x2005
WGL_SWAP_LAYER_BUFFERS_ARB	0x2006
WGL_SWAP_METHOD_ARB	0x2007
WGL_NUMBER_OVERLAYS_ARB	0x2008
WGL_NUMBER_UNDERLAYS_ARB	0x2009
WGL_TRANSPARENT_ARB	0x200A
WGL_TRANSPARENT_RED_VALUE_ARB	0x2037
WGL_TRANSPARENT_GREEN_VALUE_ARB	0x2038
WGL_TRANSPARENT_BLUE_VALUE_ARB	0x2039
WGL_TRANSPARENT_ALPHA_VALUE_ARB	0x203A
WGL_TRANSPARENT_INDEX_VALUE_ARB	0x203B
WGL_SHARE_DEPTH_ARB	0x200C
WGL_SHARE_STENCIL_ARB	0x200D
WGL_SHARE_ACCUM_ARB	0x200E
WGL_SUPPORT_GDI_ARB	0x200F
WGL_SUPPORT_OPENGL_ARB	0x2010
WGL_DOUBLE_BUFFER_ARB	0x2011
WGL_STEREO_ARB	0x2012
WGL_PIXEL_TYPE_ARB	0x2013
WGL_COLOR_BITS_ARB	0x2014
WGL_RED_BITS_ARB	0x2015
WGL_RED_SHIFT_ARB	0x2016
WGL_GREEN_BITS_ARB	0x2017
WGL_GREEN_SHIFT_ARB	0x2018
WGL_BLUE_BITS_ARB	0x2019
WGL_BLUE_SHIFT_ARB	0x201A
WGL_ALPHA_BITS_ARB	0x201B
WGL_ALPHA_SHIFT_ARB	0x201C
WGL_ACCUM_BITS_ARB	0x201D
WGL_ACCUM_RED_BITS_ARB	0x201E
WGL_ACCUM_GREEN_BITS_ARB	0x201F
WGL_ACCUM_BLUE_BITS_ARB	0x2020
WGL_ACCUM_ALPHA_BITS_ARB	0x2021
WGL_DEPTH_BITS_ARB	0x2022
WGL_STENCIL_BITS_ARB	0x2023
WGL_AUX_BUFFERS_ARB	0x2024

Accepted as a value in the <piAttribIList> and <pfAttribFList> parameter arrays of wglChoosePixelFormatARB, and returned in the <piValues> parameter array of wglGetPixelFormatAttribivARB, and the <pfValues> parameter array of wglGetPixelFormatAttribfvARB:

```
    WGL_NO_ACCELERATION_ARB                 0x2025
    WGL_GENERIC_ACCELERATION_ARB            0x2026
    WGL_FULL_ACCELERATION_ARB               0x2027

    WGL_SWAP_EXCHANGE_ARB                   0x2028
    WGL_SWAP_COPY_ARB                       0x2029
    WGL_SWAP_UNDEFINED_ARB                  0x202A

    WGL_TYPE_RGBA_ARB                       0x202B
    WGL_TYPE_COLORINDEX_ARB                 0x202C
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the WGL Specification

Pixel Formats

WGL uses pixel format indices to refer to the pixel formats supported by a device. The standard pixel format functions DescribePixelFormat, DescribeLayerPlane, ChoosePixelFormat, SetPixelFormat, and GetPixelFormat specify pixel format attributes using the PIXELFORMATDESCRIPTOR and LAYERPLANEDESCRIPTOR data structures.

An additional set of functions may be used to query and specify pixel format attributes by name.

Querying Pixel Format Attributes

The following two functions can be used to query pixel format attributes by specifying a list of attributes to be queried and providing a buffer in which to receive the results from the query. These functions can be used to query the attributes of both the main plane and layer planes of a given pixel format.

```
BOOL wglGetPixelFormatAttribivARB(HDC hdc,
                                  int iPixelFormat,
```

```
                                    int iLayerPlane,
                                    UINT nAttributes,
                                    const int *piAttributes,
                                    int *piValues);
```

<hdc> specifies the device context on which the pixel format is supported.

<iPixelFormat> is an index that specifies the pixel format. The pixel formats that a device context supports are identified by positive one-based integer indexes.

<iLayerPlane> specifies which plane is being queried. Positive values of <iLayerPlane> identify overlay planes, where 1 is the first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane and so on. Use zero for the main plane.

<nAttributes> number of attributes being queried.

<piAttributes> list containing an array of pixel format attribute identifiers which specify the attributes to be queried. The following values are accepted:

WGL_NUMBER_PIXEL_FORMATS_ARB
The number of pixel formats for the device context. The <iLayerPlane> and <iPixelFormat> parameters are ignored if this attribute is specified.

WGL_DRAW_TO_WINDOW_ARB
True if the pixel format can be used with a window. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_DRAW_TO_BITMAP_ARB
True if the pixel format can be used with a memory bitmap. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_ACCELERATION_ARB
Indicates whether the pixel format is supported by the driver. If this is set to WGL_NO_ACCELERATION_ARB then only the software renderer supports this pixel format; if this is set to WGL_GENERIC_ACCELERATION_ARB then the pixel format is supported by an MCD driver; if this is set to WGL_FULL_ACCELERATION_ARB then the pixel format is supported by an ICD driver.

WGL_NEED_PALETTE_ARB
A logical palette is required to achieve the best results for this pixel format. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_NEED_SYSTEM_PALETTE_ARB
The hardware supports one hardware palette in 256-color mode only. The <iLayerPlane> parameter is ignored if this attribute

is specified.

WGL_SWAP_LAYER_BUFFERS_ARB
True if the pixel format supports swapping layer planes independently of the main planes. If the pixel format does not support a back buffer then this is set to FALSE. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_SWAP_METHOD_ARB
If the pixel format supports a back buffer, then this indicates how they are swapped. If this attribute is set to WGL_SWAP_EXCHANGE_ARB then swapping exchanges the front and back buffer contents; if it is set to WGL_SWAP_COPY_ARB then swapping copies the back buffer contents to the front buffer; if it is set to WGL_SWAP_UNDEFINED_ARB then the back buffer contents are copied to the front buffer but the back buffer contents are undefined after the operation. If the pixel format does not support a back buffer then this parameter is set to WGL_SWAP_UNDEFINED_ARB. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_NUMBER_OVERLAYS_ARB
The number of overlay planes. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_NUMBER_UNDERLAYS_ARB
The number of underlay planes. The <iLayerPlane> parameter is ignored if this attribute is specified.

WGL_TRANSPARENT_ARB
True if transparency is supported.

WGL_TRANSPARENT_RED_VALUE_ARB
Specifies the transparent red color value. Typically this value is the same for all layer planes. This value is undefined if transparency is not supported.

WGL_TRANSPARENT_GREEN_VALUE_ARB
Specifies the transparent green value. Typically this value is the same for all layer planes. This value is undefined if transparency is not supported.

WGL_TRANSPARENT_BLUE_VALUE_ARB
Specifies the transparent blue color value. Typically this value is the same for all layer planes. This value is undefined if transparency is not supported.

WGL_TRANSPARENT_ALPHA_VALUE_ARB
Specifies the transparent alpha value. This is reserved for future use.

WGL_TRANSPARENT_INDEX_VALUE_ARB
Specifies the transparent color index value. Typically this value is the same for all layer planes. This value is undefined if transparency is not supported.

WGL_SHARE_DEPTH_ARB
True if the layer plane shares the depth buffer with the main planes. If <iLayerPlane> is zero, this is always true.

WGL_SHARE_STENCIL_ARB
True if the layer plane shares the stencil buffer with the main planes. If <iLayerPlane> is zero, this is always true.

WGL_SHARE_ACCUM_ARB
True if the layer plane shares the accumulation buffer with the main planes. If <iLayerPlane> is zero, this is always true.

WGL_SUPPORT_GDI_ARB
True if GDI rendering is supported.

WGL_SUPPORT_OPENGL_ARB
True if OpenGL is supported.

WGL_DOUBLE_BUFFER_ARB
True if the color buffer has back/front pairs.

WGL_STEREO_ARB
True if the color buffer has left/right pairs.

WGL_PIXEL_TYPE_ARB
The type of pixel data. This can be set to WGL_TYPE_RGBA_ARB or WGL_TYPE_COLORINDEX_ARB.

WGL_COLOR_BITS_ARB
The number of color bitplanes in each color buffer. For RGBA pixel types, it is the size of the color buffer, excluding the alpha bitplanes. For color-index pixels, it is the size of the color index buffer.

WGL_RED_BITS_ARB
The number of red bitplanes in each RGBA color buffer.

WGL_RED_SHIFT_ARB
The shift count for red bitplanes in each RGBA color buffer.

WGL_GREEN_BITS_ARB
The number of green bitplanes in each RGBA color buffer.

WGL_GREEN_SHIFT_ARB
The shift count for green bitplanes in each RGBA color buffer.

WGL_BLUE_BITS_ARB
The number of blue bitplanes in each RGBA color buffer.

WGL_BLUE_SHIFT_ARB
The shift count for blue bitplanes in each RGBA color buffer.

WGL_ALPHA_BITS_ARB
The number of alpha bitplanes in each RGBA color buffer.

WGL_ALPHA_SHIFT_ARB
The shift count for alpha bitplanes in each RGBA color buffer.

```
    WGL_ACCUM_BITS_ARB
    The total number of bitplanes in the accumulation buffer.

    WGL_ACCUM_RED_BITS_ARB
    The number of red bitplanes in the accumulation buffer.

    WGL_ACCUM_GREEN_BITS_ARB
    The number of green bitplanes in the accumulation buffer.

    WGL_ACCUM_BLUE_BITS_ARB
    The number of blue bitplanes in the accumulation buffer.

    WGL_ACCUM_ALPHA_BITS_ARB
    The number of alpha bitplanes in the accumulation buffer.

    WGL_DEPTH_BITS_ARB
    The depth of the depth (z-axis) buffer.

    WGL_STENCIL_BITS_ARB
    The depth of the stencil buffer.

    WGL_AUX_BUFFERS_ARB
    The number of auxiliary buffers.

<piValues> points to a buffer into which the results of the query
will be placed. Floating point attribute values are rounded to the
nearest integer value. The caller must allocate this array and it
must have at least <nattributes> entries.

If the function succeeds, the return value is TRUE. If the function
fails, the return value is FALSE. To get extended error information,
call GetLastError.

An error is generated if <piAttributes> contains an invalid
attribute, if <iPixelFormat> is not a positive integer or is larger
than the number of pixel formats, if <iLayerPlane> doesn't refer to
an existing layer plane, or if <hdc> is invalid.

If FALSE is returned, the contents of <piValues> are undefined.

BOOL wglGetPixelFormatAttribfvARB(HDC hdc,
                                  int iPixelFormat,
                                  int iLayerPlane,
                                  UINT nAttributes,
                                  const int *piAttributes,
                                  FLOAT *pfValues);

<hdc> specifies the device context on which the pixel format is
supported.

<iPixelFormat> is an index that specifies the pixel format. The
pixel formats that a device context supports are identified by
positive one-based integer indexes.

<iLayerPlane> specifies which plane is being queried. Positive
values of <iLayerPlane> identify overlay planes, where 1 is the
```

first overlay plane over the main plane, 2 is the second overlay plane over the first overlay plane, and so on. Negative values identify underlay planes, where -1 is the first underlay plane under the main plane, -2 is the second underlay plane under the first underlay plane and so on. Use zero for the main plane.

<nAttributes> number of attributes being queried.

<piAttributes> list containing an array of pixel format attribute identifiers which specify the attributes to be queried. The values accepted are the same as for wglGetPixelFormatAttribivARB.

<pfValues> is a pointer to a buffer into which the results of the query will be placed. Integer attribute values are converted floating point The caller must allocate this array and it must have at least at least <nAttributes> entries.

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

An error is generated if <piAttributes> contains an invalid attribute, if <iPixelFormat> is not a positive integer or is larger than the number of pixel formats, if <iLayerPlane> doesn't refer to an existing layer plane, or if <hdc> is invalid.

If FALSE is returned, the contents of <pfValues> are undefined.

Supported Pixel Formats

The maximum index of the pixel formats which can be referenced by the standard pixel format functions is returned by a successful call to DescribePixelFormat. This may be less than the maximum index of the pixel formats which can be referenced by wglGetPixelFormatAttribivARB and wglGetPixelFormatAttribfvARB. (determined by querying WGL_NUMBER_PIXEL_FORMATS_ARB).

The pixel format of a "displayable" object (e.g. window, bitmap) is specified by passing its index to SetPixelFormat. Therefore, pixel formats which cannot be referenced by the standard pixel format functions are "non displayable".

Indices are assigned to pixel formats in the following order:

1. Accelerated pixel formats that are displayable

2. Accelerated pixel formats that are displayable and which have extended attributes

3. Generic pixel formats

4. Accelerated pixel formats that are non displayable

ChoosePixelFormat will never select pixel formats from either group 2 or group 4. Each pixel format in group 2 is required to appear identical to some pixel format in group 1 when queried by DescribePixelFormat. Consequently, ChoosePixelFormat will always

select a format from group 1 when it might otherwise have selected a format from group 2. Pixel formats in group 4 cannot be accessed by ChoosePixelFormat at all.

SetPixelFormat and DescribePixelFormat will only accept pixel formats from groups 1-3. If a non-displayable pixel format is specified to SetPixelFormat or DescribePixelFormat an error will result. These pixel formats are only for use with WGL extensions, such as WGLCreatePbufferARB.

The following function may be used to select from among all of the available pixel formats (including both accelerated and generic formats and non-displayable formats). This function accepts attributes for the main planes. A list of pixel formats that match the specified attributes is returned with the "best" pixel formats at the start of the list (order is device dependent).

```
BOOL wglChoosePixelFormatARB(HDC hdc,
                             const int *piAttribIList,
                             const FLOAT *pfAttribFList,
                             UINT nMaxFormats,
                             int *piFormats,
                             UINT *nNumFormats);
```

<hdc> specifies the device context.

<piAttribIList> specifies a list of attribute {type, value} pairs containing integer attribute values. All the attributes in <piAttribIList> are followed by the corresponding desired value. The list is terminated with 0. If <piAttribList> is NULL then the result is the same as if <piAttribList> was empty.

<pfAttribFList> specifies a list of attribute {type, value} pairs containing floating point attribute values. All the attributes in <pfAttribFList> are followed by the corresponding desired value. The list is terminated with 0. If <pfAttribList> is NULL then the result is the same as if <pfAttribList> was empty.

<nMaxFormats> specifies the maximum number of pixel formats to be returned.

<piFormats> points to an array of returned indices of the matching pixel formats. The best pixel formats (i.e., closest match and best format for the hardware) are at the head of the list. The caller must allocate this array and it must have at least <nMaxFormats> entries.

<nNumFormats> returns the number of matching formats. This value may be larger than <nMaxFormats>.

If the function succeeds, the return value is TRUE. If the function fails the return value is FALSE. To get extended error information, call GetLastError. If no matching formats are found then nNumFormats is set to zero and the function returns TRUE.

If FALSE is returned, the contents of <piFormats> are undefined.

wglChoosePixelFormatARB selects pixel formats to return based on the attribute values specified in <piAttribIList> and <pfAttribFList>. Some attribute values must match the pixel format value exactly when the attribute is specified while others specify a minimum criteria, meaning that the pixel format value must meet or exceed the specified value. See the table below for details.

Attribute	Type	Match Criteria
WGL_DRAW_TO_WINDOW_ARB	boolean	exact
WGL_DRAW_TO_BITMAP_ARB	boolean	exact
WGL_ACCELERATION_ARB	enum	exact
WGL_NEED_PALETTE_ARB	boolean	exact
WGL_NEED_SYSTEM_PALETTE_ARB	boolean	exact
WGL_SWAP_LAYER_BUFFERS_ARB	boolean	exact
WGL_SWAP_METHOD_ARB	enum	exact
WGL_NUMBER_OVERLAYS_ARB	integer	minimum
WGL_NUMBER_UNDERLAYS_ARB	integer	minimum
WGL_SHARE_DEPTH_ARB	boolean	exact
WGL_SHARE_STENCIL_ARB	boolean	exact
WGL_SHARE_ACCUM_ARB	boolean	exact
WGL_SUPPORT_GDI_ARB	boolean	exact
WGL_SUPPORT_OPENGL_ARB	boolean	exact
WGL_DOUBLE_BUFFER_ARB	boolean	exact
WGL_STEREO_ARB	boolean	exact
WGL_PIXEL_TYPE_ARB	enum	exact
WGL_COLOR_BITS_ARB	integer	minimum
WGL_RED_BITS_ARB	integer	minimum
WGL_GREEN_BITS_ARB	integer	minimum
WGL_BLUE_BITS_ARB	integer	minimum
WGL_ALPHA_BITS_ARB	integer	minimum
WGL_ACCUM_BITS_ARB	integer	minimum
WGL_ACCUM_RED_BITS_ARB	integer	minimum
WGL_ACCUM_GREEN_BITS_ARB	integer	minimum
WGL_ACCUM_BLUE_BITS_ARB	integer	minimum
WGL_ACCUM_ALPHA_BITS_ARB	integer	minimum
WGL_DEPTH_BITS_ARB	integer	minimum
WGL_STENCIL_BITS_ARB	integer	minimum
WGL_AUX_BUFFERS_ARB	integer	minimum

All attributes except WGL_NUMBER_OVERLAYS_ARB, WGL_NUMBER_UNDERLAYS_ARB, WGL_SHARE_DEPTH_ARB, WGL_SHARE_STENCIL_ARB, and WGL_SHARE_ACCUM_ARB apply to the main planes and not to any layer planes. If WGL_SHARE_DEPTH_ARB, WGL_SHARE_STENCIL_ARB, and WGL_SHARE_ACCUM_ARB are specified in either <piAttribList> or <pfAttribList>, then a pixel format will only be selected if it has no overlays or underlays or if all of its overlays and underlays match the specified value. Applications that need to find a pixel format that supports a layer plane with other buffer attributes (such as WGL_SUPPORT_OPENGL_ARB set to TRUE), must go through the list that is returned and call wglGetPixelFormatAttrib*vARB to find one with the appropriate attributes.

Attributes that are specified in neither <piAttribIList> nor <pfAttribFList> are ignored (i.e., they are not looked at during the selection process). In addition the following attributes are always

ignored, even if specified: WGL_NUMBER_PIXEL_FORMATS_ARB, WGL_RED_SHIFT_ARB, WGL_GREEN_SHIFT_ARB, WGL_BLUE_SHIFT_ARB, WGL_ALPHA_SHIFT_ARB, WGL_TRANSPARENT_ARB, WGL_TRANSPARENT_RED_VALUE_ARB,WGL_TRANSPARENT_GREEN_VALUE_ARB, WGL_TRANSPARENT_BLUE_VALUE_ARB, WGL_TRANSPARENT_ALPHA_VALUE_ARB, and WGL_TRANSPARENT_INDEX_ARB.

If both <piAttribIList> and <pfAttribFList> are NULL or empty then all pixel formats for this device are returned.

An error is generated if <piAttribIList> or <pfAttribFList> contain an invalid attribute or if <hdc> is invalid.

Although it is not an error, wglChoosePixelFormat and wglChoosePixelFormatARB should not be used together. It is not necessary to change existing OpenGL programs but application writers should use wglChoosePixelFormatARB whenever possible. New pixel format attributes introduced by extensions (such as the number of multisample buffers) will only be known to the new calls, wglChoosePixelFormatARB and wglGetPixelFormatAttrib*vARB..

New State

None

New Implementation Dependent State

None

Dependencies on WGL_ARB_extensions_string

Because there is no way to extend WGL, these calls are defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this extension is a WGL extension, it is not included in the extension string returned by glGetString. Its existence can be determined with the WGL_ARB_extensions_string extension.

Revision History

Changes from EXT_pixel_format:

* Added WGL prefix to name to avoid possible name collisions
* EXT suffix changed to ARB
* Updated to new template, adding contact, status and revision sections
* Version is no longer an RCS version
* Attribute list passed to wglGetPixelFormatAttrib*v is type const
* Separate red,green,blue,alpha and index transparent values
* WGL_SWAP_LAYER_BUFFERS and WGL_SWAP_METHOD values defined for single buffered pixel formats
* Array of return values for wglGetPixelFormatAttrib*v and wglChoosePixelFormatARB is undefined if function fails
* Error returned if iPixelFormat is zero or negative in wglGetPixelFormat*v
* Under "Supported Pixel Formats", indicate that SetPixelFormat and DescribePixelFormat do not accept non displayable pixel formats. Passing one in results in an error
* If either piAttribIList of pfAttribFList are NULL when

```
    wglChoosePixelFormatARB is called then it is as if they were empty
  * Clarify that wglChoosePixelFormatARB returns TRUE even if no matching
    formats found
  * wglChoosePixelFormatARB will only match an overlay attribute (eg,
    WGL_SHARE_DEPTH_ARB) if there are no overlay planes or if all
    overlay/underlay plane attributes match the specified criteria
  * Be careful about using term hardware (change to pixel format where
    appropriate)
  * wglChoosePixelFormatARB now ignores the following attributes (in
    addition to WGL_NUMBER_PIXEL_FORMATS_ARB): WGL_*_SHIFT_ARB,
    WGL_TRANSPARENT_ARB, WGL_TRANSPARENT_*_VALUE_ARB.
  * Clarify that new pixel format attributes (eg, attributes introduced by
    extensions such as multisampling) are only known to the new pixel
    format calls, wglChoosePixelFormatARB and wglGetPixelFormat*vARB.
  * Add dependency on WGL_ARB_extensions_string
```

Name

EXT_swap_control

Name Strings

WGL_EXT_swap_control

Version

Date: 1/27/1999 Revision: 1.3

Number

172

Dependencies

WGL_EXT_extensions_string is required.

Overview

This extension allows an application to specify a minimum periodicity of color buffer swaps, measured in video frame periods.

New Procedures and Functions

BOOL wglSwapIntervalEXT(int interval)

int wglGetSwapIntervalEXT(void)

New Tokens

None

Additions to Chapter 2 of the 1.2 GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 GL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 GL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.2 GL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 GL Specification (State and State Requests)

None

Additions to the WGL Specification

wglSwapIntervalEXT specifies the minimum number of video frame periods per buffer swap for the window associated with the current context. The interval takes effect when SwapBuffers or wglSwapLayerBuffer is first called subsequent to the wglSwapIntervalEXT call.

The parameter 'interval' specifies the minimum number of video frames that are displayed before a buffer swap will occur.

A video frame period is the time required by the monitor to display a full frame of video data. In the case of an interlaced monitor, this is typically the time required to display both the even and odd fields of a frame of video data. An interval set to a value of 2 means that the color buffers will be swapped at most every other video frame.

If 'interval' is set to a value of 0, buffer swaps are not synchronized to a video frame. The 'interval' value is silently clamped to the maximum implementation-dependent value supported before being stored.

The swap interval is not part of the render context state. It cannot be pushed or popped. The current swap interval for the window associated with the current context can be obtained by calling wglGetSwapIntervalEXT. The default swap interval is 1.

Because there is no way to extend wgl, this call is defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this is not a GL extension, it is not included in the GL_EXTENSIONS string.

Errors

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

ERROR_INVALID_DATA The 'interval' parameter is negative.

New State

None

New Implementation Dependent State

None