NVIDIA OpenGL
Extension Specifications for the
GeForce 8 Series Architecture (G8x)

February 14, 2008

This document is an abridged collection of OpenGL extension specifications limited to those extensions for *new* OpenGL functionality introduced by the GeForce 8 Series (G8*x*) architecture. See the unabridged document "NVIDIA OpenGL Extension Specifications" for a complete collection.

NVIDIA-specific OpenGL extension specifications, possibly more up-to-date, can be found at:

http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs

Other OpenGL extension specifications can be found at:

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

Corrections? Email opengl-specs@nvidia.com

Table of Contents

Table of NVIDIA OpenGL Extension Support

Extension	NV1*x*	NV2*x*	*NV3x*	*NV4x*	*G8x*	Notes
ARB_color_buffer_float				R75	X	
ARB_depth_texture		R25+	X	X	X	1.4 functionality
ARB_draw_buffers				R75	X	2.0 functionality
ARB_fragment_program			X	X	X	
ARB_fragment_program_shadow			R55	X	X	
ARB_fragment_shader			R60	X	X	2.0 functionality, GLSL
ARB_half_float_pixel			R75	R75	X	
ARB_imaging	R10	X	X	X	X	1.2 imaging subset
ARB_multisample		X	X	X	X	1.3 functionality
ARB_multitexture	X	X	X	X	X	1.3 functionality
ARB_occlusion_query		R50	R50	R50	X	1.5 functionality
ARB_pixel_buffer_object	R80	R80	R80	R80	X	2.1 functionality
ARB_point_parameters	R35	R35	X	X	X	1.4 functionality
ARB_point_sprite	R50	R50	R50	X	X	
ARB_shader_objects	R60	R60	R60	X	X	2.0 functionality, GLSL
ARB_shading_language_100	R60	R60	R60	X	X	2.0 functionality, GLSL
ARB_shadow		R25+	X	X	X	1.4 functionality
ARB_texture_border_clamp		X	X	X	X	1.3 functionality
ARB_texture_compression	X	X	X	X	X	1.3 functionality
ARB_texture_cube_map	X	X	X	X	X	1.3 functionality
ARB_texture_env_add	X	X	X	X	X	1.3 functionality
ARB_texture_env_combine	X	X	X	X	X	1.3 functionality
ARB_texture_env_crossbar						see explanation
ARB_texture_env_dot3	X	X	X	X	X	1.3 functionality
ARB_texture_mirrored_repeat	R40	R40	X	X	X	1.4, same as IBM
ARB_texture_non_power_of_two				X	X	2.0 functionality
ARB_texture_rectangle	R62	R60+	R62	R62	X	
ARB_transpose_matrix	X	X	X	X	X	1.3 functionality
ARB_vertex_buffer_object	R65	R65	R65	R65	X	1.5 functionality
ARB_vertex_program	R40+	R40+	X	X	X	
ARB_vertex_shader	R60	R60	R60	R60	X	2.0 functionality, GLSL
ARB_window_pos	R40	R40	X	X	X	1.4 functionality
ATI_draw_buffers				X	X	
ATI_texture_float				X	X	
ATI_texture_mirror_once				X	X	use EXT_texture_mirror_clamp
EXT_abgr	X	X	X	X	X	
EXT_bgra	X	X	X	X	X	1.2 functionality
EXT_bindable_uniform					X	GLSL extension
EXT_blend_color	X	X	X	X	X	1.4 functionality
EXT_blend_equation_separate				R60	X	2.0 functionality
EXT_blend_func_separate			X	X	X	1.4 functionality
EXT_blend_minmax	X	X	X	X	X	1.4 functionality
EXT_blend_subtract	X	X	X	X	X	1.4 functionality
EXT_Cg_shader	R60	R60	R60	R60	X	Cg through GLSL API
EXT_clip_volume_hint	R20+					
EXT_compiled_vertex_array	X	X	X	X	X	
EXT_depth_bounds_test			R50	X	X	NV35, NV36, NV4x in hw only
EXT_draw_buffers2					X	ARB_draw_buffers extension
EXT_draw_instanced					X	
EXT_draw_range_elements	R20	R20	X	X	X	1.2 functionality
EXT_fog_coord	X	X	X	X	X	1.4 functionality
EXT_framebuffer_blit			R95	R95	X	
EXT_framebuffer_multisample			R95	R95	X	
EXT_framebuffer_object			R75	R75	X	
EXT_framebuffer_sRGB					X	
EXT_geometry_shader4					X	GLSL extension
EXT_gpu_program_parameters	R95	R95	R95	R95	X	
EXT_gpu_shader4					X	GLSL extension
EXT_multi_draw_arrays	R25	R25	X	X	X	1.4 functionality
EXT_packed_depth_stencil			R80	X	X	
EXT_packed_float					X	
EXT_packed_pixels	X	X	X	X	X	1.2 functionality

Extension	NV1*x*	NV2*x*	*NV3x*	*NV4x*	*G8x*	Notes
EXT_paletted_texture	X	X	X			no NV4x hw support
EXT_pixel_buffer_object	R55	R55	R55	X	X	2.1 functionality
EXT_point_parameters	X	X	X	X	X	1.4 functionality
EXT_rescale_normal	X	X	X	X	X	1.2 functionality
EXT_secondary_color	X	X	X	X	X	1.4 functionality
EXT_separate_specular_color	X	X	X	X	X	1.2 functionality
EXT_shadow_funcs		R25+	X	X	X	1.5 functionality
EXT_shared_texture_palette	X	X	X			no NV4x hw support
EXT_stencil_clear_tag				R70		NV44 only
EXT_stencil_two_side			X	X	X	2.0 functionality
EXT_stencil_wrap	X	X	X	X	X	1.4 functionality
EXT_texture3D	sw	X	X	X	X	1.2 functionality
EXT_texture_array					X	
EXT_texture_buffer_object					X	
EXT_texture_compression_latc					X	
EXT_texture_compression_rgtc					X	
EXT_texture_compression_s3tc	X	X	X	X	X	
EXT_texture_cube_map	X	X	X	X	X	1.2 functionality
EXT_texture_edge_clamp	X	X	X	X	X	1.2 functionality
EXT_texture_env_add	X	X	X	X	X	1.3 functionality
EXT_texture_env_combine	X	X	X	X	X	1.3 functionality
EXT_texture_env_dot3	X	X	X	X	X	1.3 functionality
EXT_texture_filter_anisotropic	X	X	X	X	X	
EXT_texture_integer					X	
EXT_texture_lod	X	X	X	X	X	1.2 functionality; no spec
EXT_texture_lod_bias	X	X	X	X	X	1.4 functionality
EXT_texture_mirror_clamp				X	X	
EXT_texture_object	X	X	X	X	X	1.1 functionality
EXT_texture_shared_exponent					X	
EXT_texture_sRGB				X	X	2.1 functionality
EXT_timer_query		R80	R80	R80	X	
EXT_vertex_array	X	X	X	X	X	1.1 functionality
EXT_vertex_weighting	X	X				Discontinued
KTX_buffer_region	X	X	X	X	X	
HP_occlusion_test		R25	X	X	X	
IBM_rasterpos_clip	R40+	R40+	R40+	X	X	
IBM_texture_mirrored_repeat	X	X	X	X	X	1.4 functionality
KTX_buffer_region	X	X	X	X	X	use ARB_buffer_region
NV_blend_square	X	X	X	X	X	1.4 functionality
NV_conditional_render					X	
NV_copy_depth_to_color		R20	X	X	X	
NV_depth_buffer_float					X	
NV_depth_clamp		R25+	X	X	X	
NV_evaluators	R10	X				Discontinued
NV_fence	X	X	X	X	X	
NV_float_buffer			X	X	X	
NV_fog_distance	X	X	X	X	X	
NV_fragment_program			X	X	X	
NV_fragment_program_option			R55	X	X	NV_fp features for ARB_fp
NV_fragment_program2				X	X	
NV_fragment_program4					X	See NV_gpu_program4
NV_framebuffer_multisample_coverage			Nf	Nf	X	FBO extension
NV_geometry_program4					X	See NV_gpu_program4
NV_geometry_shader4					X	
NV_gpu_program4					X	
NV_half_float			X	X	X	
NV_light_max_exponent	X	X	X	X	X	
NV_multisample_filter_hint		X	X	X	X	
NV_occlusion_query		R25	X	X	X	
NV_packed_depth_stencil	R10+	R10+	X	X	X	
NV_parameter_buffer_object					X	See NV_gpu_program4
NV_pixel_data_range	R40	R40	X	X	X	
NV_point_sprite	R35+	R25	X	X	X	
NV_primitive_restart			X	X	X	

Extension	NV1*x*	NV2*x*	*NV3x*	*NV4x*	*G8x*	Notes
NV_register_combiners	X	X	X	X	X	
NV_register_combiners2		X	X	X	X	
NV_texgen_emboss	X					Discontinued
NV_texgen_reflection	X	X	X	X	X	use 1.3 functionality
NV_texture_compression_vtc		X	X	X	X	
NV_texture_env_combine4	X	X	X	X	X	
NV_texture_expand_normal			X	X	X	
NV_texture_rectangle	X	X	X	X	X	
NV_texture_shader		X	X	X	X	
NV_texture_shader2		X	X	X	X	
NV_texture_shader3		R25	X	X	X	only NV25 and up in HW
NV_transform_feedback					X	
NV_vertex_array_range	X	X	X	X	X	
NV_vertex_array_range2	R10	R10	X	X	X	
NV_vertex_program	R10	X	X	X	X	
NV_vertex_program1_1	R25	R25	X	X	X	
NV_vertex_program2			X	X	X	
NV_vertex_program2_option			R55	X	X	
NV_vertex_program3				X	X	
NV_vertex_program4					X	See NV_gpu_program4
S3_s3tc	X	X	X	X	X	no spec; use EXT_t_c_s3tc
SGIS_generate_mipmap	R10	X	X	X	X	1.4 functionality
SGIS_multitexture	X	X				use 1.3 version
SGIS_texture_lod	X	X	X	X	X	1.2 functionality
SGIX_depth_texture		X	X	X	X	use 1.4 version
SGIX_shadow		X	X	X	X	use 1.4 version
SUN_slice_accum	R50	R50	R50	X	X	accelerated on NV3x/NV4x
WGL_ARB_buffer_region	X	X	X	X	X	Win32
WGL_ARB_extensions_string	X	X	X	X	X	Win32
WGL_ARB_make_current_read	R55	R55	R55	X	X	
WGL_ARB_multisample		X	X	X	X	see ARB_multisample
WGL_ARB_pixel_format	R10	X	X	X	X	Win32
WGL_ARB_pbuffer	R10	X	X	X	X	Win32
WGL_ARB_render_texture	R25	R25	X	X	X	Win32
WGL_ATI_pixel_format_float				X	X	Win32
WGL_EXT_extensions_string	X	X	X	X	X	Win32
WGL_EXT_swap_control	X	X	X	X	X	Win32
WGL_NV_float_buffer			X	X	X	Win32, see NV_float_buffer
WGL_NV_render_depth_texture		R25	X	X	X	Win32
WGL_NV_render_texture_rectangle	R25	R25	X	X	X	Win32
WIN_swap_hint	X	X	X	X	X	Win32, no spec

Key for table entries:

X = *supported*

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

Nf = *Extension advertised but rendering functionality not available*

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

R20 = *introduced in the Detanator XP (also known as Release 20) OpenGL driver (not supported by earlier drivers)*

R20+ = *introduced after the Detanator XP (also known as Release 20) OpenGL driver (not supported by earlier drivers)*

R25 = *introduced in the GeForce4 launch (also known as Release 25) OpenGL driver (not supported by earlier drivers)*

R25+ = *introduced after the GeForce4 launch (also known as Release 25) OpenGL driver (not supported by earlier drivers)*

R35 = *post-GeForce4 launch OpenGL driver release (not supported by earlier drivers)*

R40 = Detonator 40 release, August 2002.

R40+ = *introduced after the Detanator 40 (also known as Release 40) OpenGL driver (not supported by earlier drivers)*

R50 = Detonator 50 release

R55 = Detonator 55 release

R60 = Detonator 60 release, May 2004

R65 = Release 65

R70 = Release 70

R80 = Release 80

R95 = Release 95

no spec = *no suitable specification available*

Discontinued = earlier drivers (noted by 25% gray entries) supported this extension but support for the extension is discontinued in current and future drivers

Notices:

Emulation: While disabled by default, older GPUs can support extensions supported in hardware by newer GPUs through a process called emulation though any functionality unsupported by the older GPU must be emulated via software. For more details see: http://developer.nvidia.com/object/nvemulate.html

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.

Discontinuation of support: NVIDIA drivers from release 95 no longer support NV1x- and NV2x-based GPUs.

Name

EXT_bindable_uniform

Name String

GL_EXT_bindable_uniform

Contact

Pat Brown, NVIDIA (pbrown 'at' nvidia.com)
Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 12/13/2007
Author revision: 13

Number

342

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 2.0 specification and version 1.10.59 of the OpenGL Shading Language specification.

This extension interacts with GL_EXT_geometry_shader4.

Overview

This extension introduces the concept of bindable uniforms to the OpenGL Shading Language. A uniform variable can be declared bindable, which means that the storage for the uniform is not allocated by the compiler/linker anymore, but is backed by a buffer object. This buffer object is bound to the bindable uniform through the new command UniformBufferEXT(). Binding needs to happen after linking a program object.

Binding different buffer objects to a bindable uniform allows an application to easily use different "uniform data sets", without having to re-specify the data every time.

A buffer object can be bound to bindable uniforms in different program objects. If those bindable uniforms are all of the same type, accessing a bindable uniform in program object A will result in the same data if the same access is made in program object B. This provides a mechanism for 'environment uniforms', uniform values that can be shared among multiple program objects.

New Procedures and Functions

```
void UniformBufferEXT(uint program, int location, uint buffer);
int GetUniformBufferSizeEXT(uint program, int location);
intptr GetUniformOffsetEXT(uint program, int location);
```

New Tokens

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

MAX_VERTEX_BINDABLE_UNIFORMS_EXT	0x8DE2
MAX_FRAGMENT_BINDABLE_UNIFORMS_EXT	0x8DE3
MAX_GEOMETRY_BINDABLE_UNIFORMS_EXT	0x8DE4
MAX_BINDABLE_UNIFORM_SIZE_EXT	0x8DED
UNIFORM_BUFFER_BINDING_EXT	0x8DEF

Accepted by the <target> parameters of BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, GetBufferSubData, and GetBufferPointerv:

UNIFORM_BUFFER_EXT	0x8DEE

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify section 2.15.3 "Shader Variables", page 75.

Add the following paragraph between the second and third paragraph on page 79, "Uniform Variables"

Uniform variables can be further characterized into bindable uniforms. Storage for bindable uniforms does not come out of the, potentially limited, uniform variable storage discussed in the previous paragraph. Instead, storage for a bindable uniform is provided by a buffer object that is bound to the uniform variable. Binding different buffer objects to a bindable uniform allows an application to easily use different "uniform data sets", without having to re-specify the data every time. A buffer object can be bound to bindable uniforms in different program objects. If those bindable uniforms are all of the same type, accessing a bindable uniform in program object A will result in the same data if the same access is made in program object B. This provides a mechanism for 'environment', uniform values that can be shared among multiple program objects.

Change the first sentence of the third paragraph, p. 79, as follows:

When a program object is successfully linked, all non-bindable active uniforms belonging to the program object are initialized to zero (FALSE for Booleans). All active bindable uniforms have their buffer object bindings reset to an invalid state. A successful link will also generate a location for each active uniform, including active bindable uniforms. The values of active uniforms can be changed using this location and the appropriate Uniform* command (see below). For bindable uniforms, a buffer object has to be first bound to the uniform before changing its value. These locations are invalidated.

Change the second to last paragraph, p. 79, as follows:

A valid name for a non-bindable uniform cannot be a structure, an array of structures, or any portion of a single vector or a matrix. A valid name for a bindable uniform cannot be any portion of a single vector or matrix. In order to identify a valid name, ...

Change the fifth paragraph, p. 81, as follows:

The given values are loaded into the uniform variable location identified by <location>. The parameter <location> cannot identify a bindable uniform structure or a bindable uniform array of structures. When loading data for a bindable uniform, the data will be stored in the appropriate location of the buffer object bound to the bindable uniform (see UniformBufferEXT below).

Add the following bullets to the list of errors on p. 82:

- If <location> refers to a bindable uniform structure or a bindable uniform array of structures.

- If <location> refers to a bindable uniform that has no buffer object bound to the uniform.

- If <location> refers to a bindable uniform and the bound buffer object is not of sufficient size. This means that the buffer object is smaller than the size that would be returned by GetUniformBufferSizeEXT for the bindable uniform.

- If <location> refers to a bindable uniform and the buffer object is bound to multiple bindable uniforms in the currently active program object.

Add a sub-section called "Bindable Uniforms" above the section "Samplers", p. 82:

The number of active bindable uniform variables that can be supported by a vertex shader is limited and specified by the implementation dependent constant MAX_VERTEX_BINDABLE_UNIFORMS_EXT. The minimum supported number of bindable uniforms is eight. A link error will be generated if the program object contains more active bindable uniform variables.

To query the minimum size needed for a buffer object to back a given bindable uniform, use the command:

```
int GetUniformBufferSizeEXT(uint program, int location);
```

This command returns the size in basic machine units of the smallest buffer object that can be used for the bindable uniform given by <location>. The size returned is intended to be passed as the <size> parameter to the BufferData() command. The error INVALID_OPERATION will be generated if <location> does not correspond to an active bindable uniform in <program>. The parameter <location> has to be location corresponding to the name of the bindable uniform itself, otherwise the error INVALID_OPERATION is generated. If the bindable uniform is a structure, <location> can not refer to a structure member. If it is an array, <location> can not refer to any array member other than the first one. If

<program> has not been successfully linked, the error INVALID_OPERATION is generated.

There is an implementation-dependent limit on the size of bindable uniform variables. LinkProgram will fail if the storage required for the uniform (in basic machine units) exceeds MAX_BINDABLE_UNIFORM_SIZE_EXT.

To bind a buffer object to a bindable uniform, use the command:

```
void UniformBufferEXT(uint program, int location, uint buffer)
```

This command binds the buffer object <buffer> to the bindable uniform <location> in the program object <program>. Any previous binding to the bindable uniform <location> is broken. Before calling UniformBufferEXT the buffer object has to be created, but it does not have to be initialized with data nor its size set. Passing the value zero in <buffer> will unbind the currently bound buffer object. The error INVALID_OPERATION is generated if <location> does not correspond to an active bindable uniform in <program>. The parameter <location> has to correspond to the name of the uniform variable itself, as described for GetUniformBufferSizeEXT, otherwise the error INVALID_OPERATION is generated. If <program> has not been successfully linked, or if <buffer> is not the name of an existing buffer object, the error INVALID_OPERATION is generated.

A buffer object cannot be bound to more than one uniform variable in any single program object. However, a buffer object can be bound to bindable uniform variables in multiple program objects. Furthermore, if those bindable uniforms are all of the same type, accessing a scalar, vector, a member of a structure, or an element of an array in program object A will result in the same data if the same scalar, vector, structure member, or array element is accessed in program object B. Additionally the structures in both program objects have to have the same members, specified in the same order, declared with the same data types and have the same name. If the buffer object bound to the uniform variable is smaller than the minimum size required to store the uniform variable, as reported by GetUniformbufferSizeEXT, the results of reading the variable (or any portion thereof) are undefined.

If LinkProgram is called on a program object that has already been linked, any buffer objects bound to the bindable uniforms in the program are unbound prior to linking, as though UniformBufferEXT were called for each bindable uniform with a <buffer> value of zero.

Buffer objects used to store uniform variables may be created and manipulated by buffer object functions (e.g., BufferData, BufferSubData, MapBuffer) by calling BindBuffer with a <target> of UNIFORM_BUFFER_EXT. It is not necessary to bind a buffer object to UNIFORM_BUFFER_EXT in order to use it with an active program object.

The size and layout of a bindable uniform variable in buffer object storage is not defined. However, the values of signed integer, unsigned integer, or floating-point uniforms may be updated by modifying the underying buffer object storage using either MapBuffer or BufferSubData. The command

```
intptr GetUniformOffsetEXT(uint program, int location);
```

returns the offset (in bytes) of the uniform in <program> whose location returned by GetUniformLocation is <location>. The error INVALID_VALUE is generated if the object named by <program> does not exist. The error INVALID_OPERATION is generated if <program> is not a program object, if <program> was not linked successfully, or if <location> refers to a uniform that was not declared as bindable. The memory layout of matrix, boolean, or boolean vector uniforms is not defined, and the error INVALID_OPERATION will be generated if <location> refers to a boolean, boolean vector, or matrix uniform. The value -1 is returned by GetUniformOffsetEXT if an error is generated.

The values of such uniforms may be changing by writing signed integer, unsigned integer, or floating-point values into the buffer object at the byte offset returned by GetUniformOffsetEXT. For vectors, two to four integers or floating-point values should be written to consecutive locations in the buffer object storage. For arrays of scalar or vector variables, the number of bytes between individual array members is guaranteed to be constant, but array members are not guaranteed to be stored in adjacent locations. For example, some implementations may pad scalars, or two- or three-component vectors out to a four-component vector.

Change the first paragraph below the sub-heading 'Samplers', p. 82, as follows:

Samplers are special uniforms used in the OpenGL Shading Language to identify the texture object used for each texture lookup. Samplers cannot be declared as bindable in a shader. The value of a sampler indicates the texture image unit being accessed. Setting a sampler's value.

Add the following bullets to the list of error conditions for Begin on p. 87:

- There is one, or more, bindable uniform(s) in the currently active program object that does not have a buffer object bound to it.

- There is one, or more, bindable uniform(s) in the currently active program object that have a buffer object bound to it of insufficient size. This means that the buffer object is smaller than the size that would be returned by GetUniformBufferSizeEXT for the bindable uniform.

- A buffer object is bound to multiple bindable uniforms in the currently active program object.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.11.1 "Shader Variables", p. 193

Add a paragraph between the first and second paragraph, p. 194

The number of active bindable uniform variables that can be supported by a fragment shader is limited and specified by the implementation dependent constant MAX_FRAGMENT_BINDABLE_UNIFORMS_EXT. The minimum supported number of bindable uniforms is eight. A link error will be generated if the program object contains more active bindable uniform variables.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the command UniformBufferEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with GL_EXT_geometry_shader4

If GL_EXT_geometry_shader4 is supported, a geometry shader will also support bindable uniforms. The following paragraph needs to be added to the section that discusses geometry shaders:

"The number of active bindable uniform variables that can be supported by a geometry shader is limited and specified by the implementation dependent constant MAX_GEOMETRY_BINDABLE_UNIFORMS_EXT. The minimum supported number of bindable uniforms is eight. A link error will be generated if the program object contains more active bindable uniform variables."

The implementation dependent value MAX_GEOMETRY_BINDABLE_UNIFORMS_EXT will need to be added to the state tables and assigned an enum value.

Errors

The error INVALID_VALUE is generated by UniformBufferEXT, GetUniformBufferSize, or GetUniformOffsetEXT if <program> is not the name of a program or shader object.

The error INVALID_OPERATION is generated by UniformBufferEXT, GetUniformBufferSize, or GetUniformOffsetEXT if <program> is the name of a shader object.

The error INVALID_OPERATION is generated by the Uniform* commands if <location> refers to a bindable uniform structure or an array of such structures.

The error INVALID_OPERATION is generated by the Uniform* commands if

<location> refers to a bindable uniform that has no buffer object bound.

The error INVALID_OPERATION is generated by the Uniform* commands if <location> refers to a bindable uniform and the bound buffer object is not of sufficient size to store data into <location>.

The error INVALID_OPERATION is generated by the GetUniformBufferSizeEXT and UniformBufferEXT commands if <program> has not been successfully linked.

The error INVALID_OPERATION is generated by the GetUniformBufferSizeEXT and UniformBufferEXT commands if <location> is not the location corresponding to the name of the bindable uniform itself or if <location> does not correspond to an active bindable uniform in <program>.

The error INVALID_OPERATION is generated by GetUniformOffsetEXT if <program> was not linked successfully, if <location> refers to a uniform that was not declared as bindable, or if <location> refers to a boolean, boolean vector, or matrix uniform.

The error INVALID_OPERATION is generated by the UniformBufferEXT command if <buffer> is not the name of a buffer object.

The error INVALID_OPERATION is generated by Begin, Rasterpos or any command that performs an implicit Begin if:

- A buffer object is bound to multiple bindable uniforms in the currently active program object.

- There is one, or more, bindable uniform(s) in the currently active program object that does not have a buffer object bound to it.

- There is one, or more, bindable uniform(s) in the currently active program object that have a buffer object bound to it of insufficient size. This means that the buffer object is smaller than the size that would be returned by GetUniformBufferSizeEXT for the bindable uniform.

New State

Get Value	Type	Get Command	Minimum Value	Description	Section	Attrib
MAX_BINDABLE_VERTEX_UNIFORMS_EXT	Z+	GetIntegerv	8	Number of bindable uniforms per vertex shader	2.15	-
MAX_BINDABLE_FRAGMENT_UNIFORMS_EXT	Z+	GetIntegerv	8	Number of bindable uniforms per fragment shader	3.11.1	-
MAX_BINDABLE_GEOMETRY_UNIFORMS_EXT	Z+	GetIntegerv	8	Number of bindable uniforms per geometry shader	X.X.X	-
MAX_BINDABLE_UNIFORM_SIZE_EXT	Z+	GetIntegerv	16384	Maximum size (in bytes) for bindable uniform storage.	2.15	-

New Implementation Dependent State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
UNIFORM_BUFFER_BINDING_EXT	Z+	GetIntegerv	0	Uniform buffer bound to the context for buffer object manipulation.	2.15	-

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

Including the following line in a shader can be used to control the language features described in this extension:

```
#extension GL_EXT_bindable_uniform: <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
#define GL_EXT_bindable_uniform 1
```

Add to section 3.6 "Keywords"

Add the following keyword:

```
bindable
```

Change section 4.3 "Type Qualifiers"

In the qualifier table, add the following sub-qualifiers under the uniform qualifier:

```
bindable uniform
```

Change section 4.3.5 "Uniform"

Add the following paragraphs between the last and the second to last paragraphs:

Uniform variables, except for samplers, can optionally be further qualified with "bindable". If "bindable" is present, the storage for the uniform comes from a buffer object, which is bound to the uniform through the GL API, as described in section 2.15.3 of the OpenGL 2.0 specification. In this case, the memory used does not count against the storage limit described in the previous paragraph. When using the "bindable" keyword, it must immediately precede the "uniform" keyword.

An example bindable uniform declaration is:

```
bindable uniform float foo;
```

Only a limited number of uniforms can be bindable for each type of shader. If this limit is exceeded, it will cause a compile-time or link-time error. Bindable uniforms that are declared but not used do not count against this limit.

Add to section 9 "Shading Language Grammar"

```
type_qualifer:
   CONST
   ATTRIBUTE  // Vertex only
   uniform-modifieropt UNIFORM

uniform-modifier:
   BINDABLE
```

Issues

1. *Is binding a buffer object to a uniform done before or after linking a program object?*

   DISCUSSION: There is no need to re-link when changing the buffer object that backs a uniform. Re-binding can therefore be relatively quickly. Binding is be done using the location of the uniform retrieved by GetUniformLocation, to make it even faster (instead of binding by name of the uniform).

   Reasons to do this before linking: The linker might want to know what buffer object backs the uniform. Binding of a buffer object to a bindable uniform, in this case, will have to be done using the name of the uniform (no location is available until after linking). Changing the binding of a buffer object to a bindable uniform means the program object will have to be re-linked, which would substantially increase the overhead of using multiple different "constant sets" in a single program.

   RESOLUTION: Binding a buffer object to a bindable uniform needs to be done after the program object is linked. One of the purposes of this extension is to be able to switch among multiple sets of uniform values efficiently.

2. *Is the memory layout of a bindable uniform available to an application?*

   DISCUSSION: Buffer objects are arrays of bytes. The application can map a buffer object and retrieve a pointer to it, and read or write into it directly. Or, the application can use the BufferSubData() command to store data in a buffer object. They can also be filled using ReadPixels (with ARB_pixel_buffer_object), or filled using extensions such as the new transform feedback extension.

   If the layout of a uniform in buffer object memory is known, these different ways of filling a buffer object could be leveraged. On the other hand, different compiler implementations may want a different packing schemes that may or may not match an end-user's expectations (e.g., all individual uniforms might be stored as vec4's). If only the Uniform*() API were allowed to modify buffer objects, we could completely hide the layout of bindable uniforms. Unfortuantely, that would limit how the buffer object can be linked to other sources of data.

   RESOLUTION: RESOLVED. The memory layout of a bindable uniform variable will not be specified. However, a query function will be added that

allows applications to determine the layout and load their buffer object via API's other than Uniform*() accordingly if they choose. Unfortunately, the layout may not be consistent across implementations of this extension.

Providing a better standard set of packing rules is highly desirable, and we hope to design and add such functionality in an extension in the near future.

3. *How is synchronization handled between a program object using a buffer object and updates to the buffer object?*

   DISCUSSION: For example, what happens when a ReadPixels into a buffer object is outstanding, that is bound to a bindable uniform while the program object, containing the bindable uniform, is in use?

   RESOLUTION: UNRESOLVED. It is probably the GL implementation's responsibility to properly synchronize such usages. This issue needs solving for GL_EXT_texture_buffer_object also, and should be consistent.

4. *A limited number of bindable uniforms can exist in one program object. Should this limit be queriable?*

   DISCUSSION: The link operation will fail if too many bindable uniforms are declared and active. Should the limit on the number of active bindable uniforms be queriable by the application?

   RESOLUTION: Yes, this limit is queriable.

5. *Is the limit discussed in the previous issue per shader type?*

   DISCUSSION: Is there a different limit for vertex shader and fragment shaders? Hardware might support different limits. The storage for uniform variables is a limit queriable per shader type, thus it would be nice to be consistent with the existing model.

   RESOLUTION: YES.

6. *Can an application find out programmatically that a uniform is declared as a bindable uniform?*

   DISCUSSION: Using GetActiveUniform() the application can programmatically find out which uniforms are active, what their type and size etc it. Do we need to add a mechanism for an application to find out if an active uniform is a bindable uniform?

   RESOLUTION: UNRESOLVED. To be consistent, the answer should be yes. However, extending GetActiveUniform() is not possible, which means we need a new API command. If we define a new API command, it probably is better to define something like: GetNewActiveUniform(int program, uint index, enum property, void *data); Or alternatively, define new API to query the properties of a uniform per uniform location: GetActiveUniformProperty(int program, int location, enum property, void *data)

7. *What to do when the buffer object bound to a bindable uniform is not big enough to back the uniform or if no buffer object is bound at all?*

DISCUSSION: The size of a buffer object can be changed, after it is bound, by calling BufferData. It is possible that the buffer object isn't sufficiently big enough to back the bindable uniform. This is an issue when loading values for uniforms and when actually rendering. In the case of loading uniforms, should the Uniform* API generate an error? In the case of rendering, should this be a Begin error?

RESOLUTION: RESOLVED. It is a Begin error if a buffer object is too small or no buffer object is bound at all. The Uniform* commands will generate an error in these cases as well.

8. *What restrictions are there on binding a buffer object to more than one bindable uniform?*

   DISCUSSION: Can a buffer object be bound to more than one uniform within a program object? No, this does not seem to be a good idea. Can a buffer object be bound to more than one uniform in different program objects? Yes, this is useful functionality to have. If each uniform is also of the same type, then data access in program object A then the same access in program object B results in the same data. In the latter case, if the uniform variables are arrays, must the arrays have the same length declared? No, that is too big of a restriction. The application is responsible for making sure the buffer object is sufficiently sized to provide storage for the largest bindable uniform array.

   RESOLUTION: RESOLVED.

9. *It is not allowed to bind a buffer object to more than one bindable uniform in a program object. There are several operations that could be affected by this rule: UseProgram(), the uniform loading commands Uniform*, Begin, RasterPos and any related rendering command. Should each operation generate an error if the rule is violated?*

   DISCUSSION: See also issue 7. The UseProgram command could generate an error if the rule is violated. However, it is possible to change the binding of a buffer object to a bindable uniform even after UseProgram has been issued. Thus should the Uniform* commands also check for this? If so, is that going to be a performance burden on uniform loading? Or should it be undefined? Finally, at rendering time violation of this rule will have to be checked. If violated, it seems to make sense to generate an error.

   RESOLUTION: RESOLVED. Make violation of the rule a Begin error and a Uniform* error.

10. *How to provide the ability to use bindable uniform arrays (or bindable uniform arrays of structures) where the amount of data can differ based on the buffer object bound to it?*

    DISCUSSION: In other words, the size of the bindable uniform is no longer declared in the shader, but determined by the buffer object backing it. This can be achieved through a variety of ways:

    bindable uniform vec3 foo[1];

    Where we would allow indexing 'off the end' of the array 'foo', because

it is backed by a buffer object. The actual size of the array will be implicitly inferred from the buffer object bound to it. It'll be the shader's responsibility to not index outside the size of the buffer object. That in turn means that the layout in buffer object memory of a bindable uniform needs to be exposed to the application.

Or we could support something like:

```
bindable uniform vec3 foo[100000]; // Some really big number
```

and make all accesses inside the buffer object bound to "foo" legal.

Or we could support something like:

```
bindable uniform float foo[];

foo[3] = 1.0;
foo[i]  = .
```

Where 'i' could be a run-time index.

RESOLUTION: For now, we will not support this functionality.

11. *Do we want to have bindable namespaces instead of the uniform qualifier "bindable"?*

DISCUSSION: Something like this:

```
bindable {
  vec3 blarg;
  int booyah;
};
```

where "blarg" and "booyah" can be referred to directly, but are both bindable to the same buffer. You can achieve this with bindable uniforms stored in structures:

```
bindable uniform struct {
  vec3 blarg;
  int booyah;
} foo;
```

but then have to use "foo.blarg" and "foo.booyah".

RESOLUTION: Not in this extension. This might be nice programming sugar, but not essential. Such a feature may be added in a future extension building on this one.

12. *How can an application load data into a bindable uniform?*

RESOLUTION: See also issue 2. Uniform variables declared as bindable can be loaded using the existing Uniform* commands, or data can be loaded in the buffer object bound to the uniform using any of the existing mechanisms.

13. *Should it be allowed to load data, using the Uniform* commands, into a buffer object that is bound to more than one bindable uniform variable in a program object?*

    DISCUSSION: It is a Begin error to attempt to render in this situation.

    RESOLUTION: Yes, to be consistent with the Begin error, it is also an error to load a value in this case.

14. *Should a buffer object binding point be provided for bindable uniforms?*

    DISCUSSION: All current OpenGL buffer object manipulation functions take a <target> to which a buffer object must be bound. In this extension, buffer objects are bound to uniforms stored in a program, and are not bound directly to the context. So these bindings may not be used to manipulate the

    RESOLUTION: Yes, a new <target> called UNIFORM_BUFFER_EXT is provided.

    The following is a simple example of creating, binding, and populating a buffer object for a bindable uniform named "stuff", which is an array of vec4 values:

```
GLuint program, buffer;
GLint location, size;
GLfloat values;

// ... compile shaders and link <program>
location = glGetUniformLocation(program, "stuff");
size = GetUniformBufferSize(program, location);
glGenBuffers(1, &buffer);
glBindBuffer(GL_UNIFORM_BUFFER_EXT, buffer);
glBufferData(GL_UNIFORM_BUFFER_EXT, size, NULL, STATIC_READ);
glUniformBufferEXT(program, location, buffer);
...
glUseProgram(program);
glUniform4fv(location, count, values);
```

Revision History

Rev.	Date	Author	Changes
13	12/13/07	pbrown	Minor clarification on what values can be passed to GetUniformBufferSizeEXT and UniformBufferEXT.
12	12/15/06	pbrown	Documented that the '#extension' token for this extension should begin with "GL_", as apparently called for per convention.
11	--		Pre-release revisions.

Name

EXT_draw_buffers2

Name Strings

GL_EXT_draw_buffers2

Contact

Mike Strauss, NVIDIA Corporation (mstrauss 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 11/06/2006
NVIDIA Revision: 9

Number

340

Dependencies

The extension is written against the OpenGL 2.0 Specification.

OpenGL 2.0 is required.

Overview

This extension builds upon the ARB_draw_buffers extension and provides separate blend enables and color write masks for each color output. In ARB_draw_buffers (part of OpenGL 2.0), separate values can be written to each color buffer, but the blend enable and color write mask are global and apply to all color outputs.

While this extension does provide separate blend enables, it does not provide separate blend functions or blend equations per color output.

New Procedures and Functions

```
void ColorMaskIndexedEXT(uint buf, boolean r, boolean g,
                         boolean b, boolean a);

void GetBooleanIndexedvEXT(enum value, uint index, boolean *data);

void GetIntegerIndexedvEXT(enum value, uint index, int *data);

void EnableIndexedEXT(enum target, uint index);

void DisableIndexedEXT(enum target, uint index);

boolean IsEnabledIndexedEXT(enum target, uint index);
```

New Tokens

None.

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modify the thrid paragraph of section 4.1.8 (Blending) , p206, to read as follows:

Blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; in color index mode it is bypassed. Blending is enabled or disabled for an individual draw buffer using

```
void EnableIndexedEXT(GLenum target, GLuint index);
void DisableIndexedEXT(GLenum target, GLuint index);
```

<target> is the symbolic constant BLEND and <index> is an integer i specifying the draw buffer associated with the symbolic constant DRAW_BUFFERi. If the color buffer associated with DRAW_BUFFERi is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using Enable or Disable with the symbolic constant BLEND. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 4.1.10), proceed to the next operation.

Modify the first paragraph of section 4.1.8 (Blending - Blending State), p209, to read as follows:

The state required for blending is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, four floating-point values to store the RGBA constant blend color, and n bits indicating whether blending is enabled or disabled for each of the n draw buffers. The initial blend equations for RGB and alpha are both FUNC_ADD. The initial blending functions are ONE for the source RGB and alpha functions, and ZERO for the destination RGB and alpha functions. The initial constant blend color is (R, G, B, A) = (0, 0, 0, 0). Initially, blending is disabled for all draw buffers.

Modify the first paragraph of section 4.2.2 (Fine Control of Buffer Updates) to read as followS:

Three commands are used to mask the writing of bits to each of the logical draw buffers after all per-fragment operations have been performed.

The commands

```
void IndexMask(uint mask);
void ColorMask(boolean r, boolean g, boolean b, boolean a);
void ColorMaskIndexedEXT(uint buf, boolean r, boolean g,
                         boolean b, boolean a);
```

control writes to the active draw buffers.

The least significant n bits of <mask>, where n is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode.

In RGBA mode, ColorMask and ColorMaskIndexedEXT are used to mask the writing of R, G, B and A values to the draw buffer or buffers. ColorMaskIndexedEXT sets the mask for a particular draw buffer. The mask for DRAW_BUFFERi is modified by passing i as the parameter <buf>. <r>, <g>, <b>, and <a> indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). The mask specified by <r>, <g>, <b>, and <a> is applied to the color buffer associated with DRAW_BUFFERi. If DRAW_BUFFERi is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers) then the mask is applied to all of the buffers. ColorMask sets the mask for all draw buffers to the same values as specified by <r>, <g>, <b>, and <a>.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify the second paragraph of section 6.1.1 (Simple Queries) p244 to read as follows:

...<data> is a pointer to a scalar or array of the indicated type in which to place the returned data.

```
void GetBooleanIndexedvEXT(enum target, uint index, boolean *data);
void GetIntegerIndexedvEXT(enum target, uint index, int *data);
```

are used to query indexed state. <target> is the name of the indexed state and <index> is the index of the particular element being queried. <data> is a pointer to a scalar or array

of the indicated type in which to place the returned data. In addition

boolean IsEnabled(enum value);

can be used to determine if <value> is currently enabled (as with Enable) or disabled.

boolean IsEnabledIndexedEXT(enum target, uint index);

can be used to determine if the index state corresponding to <target> and <index> is enabled or disabled.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Errors

The error INVALID_ENUM is generated by EnableIndexedEXT and DisableIndexedEXT if the <target> parameter is not BLEND.

The error INVALID_OPERATION is generated by EnableIndexedEXT and DisableIndexeEXT if the <target> parameter is BLEND and the <index> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

The error INVALID_ENUM is generated by IsEnabledIndexedEXT if the <target> parameter is not BLEND.

The error INVALID_OPERATION is generated by IsEnabledIndexedEXT if the <target> parameter is BLEND and the <index> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

The error INVALID_OPERATION is generated by DrawBufferColorMaskEXT if the <buf> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

The error INVALID_ENUM is generated by GetBooleanIndexedvEXT if the <target> parameter is not BLEND.

The error INVALID_OPERATION is generated by GetBooleanIndexedvEXT if the <target> parameter is BLEND and the <index> parameter is outside the range [0, MAX_DRAW_BUFFERS-1].

New State

Modify (table 6.20, p281), modifying the entry for BLEND and adding a new one.

Get Target	Type	Get Command	Value	Description	Section	Attribute
BLEND	B	IsEnabled	False	Blending enabled for draw buffer 0	4.1.8	color-buffer/enable
BLEND	B	IsEnabledIndexedEXT	False	Blending enabled for draw buffer i where i is specified as <index>	4.1.8	color-buffer/enable

Modify (table 6.21, p282), modifying the entry for COLOR_WRITEMASK and adding a new one.

Get Value	Type	Get Command	Value	Description	Section	Attribute
COLOR_WRITEMASK	4xB	GetBooleanv	True	Color write mask for draw buffer 0	4.2.2	color-buffer
COLOR_WRITEMASK	4xB	GetBooleanIndexedvEXT	True	Color write mask for draw buffer i where i is specified as <index>	4.2.2	color-buffer

Issues

1. Should the extension provide support for per draw buffer index masks as well as per draw buffer color masks?

RESOLVED: No. Color index rendering is not interesting enough to warrant extending the API in this direction.

2. Should the API for specifying separate color write masks be based on DrawBuffers() (specifying an array of write masks at once)?

RESOLVED: No. There are two ways to mimic the DrawBuffers() API. A function, ColorMasks(), could take an an element count and an array of four element boolean arrays as parameters. Each four element boolean array contains a set of red, green, blue, and alpha write masks for a specific color buffer. An alternative is a ColorMasks() function that takes an element count and four parallel boolean arrays with one array per color channel. Neither approach is particularly clean. A cleaner approach, taken by ColorMaskIndexedEXT(), is to specify a color mask for a single draw buffer where the draw buffer is specified as a parameter to the function.

3. How should ColorMask() affect the per color buffer write masks?

RESOLVED: ColorMask() should set all color buffer write masks to the same values. This is backwards compatible with the way ColorMask() behaves in the absence of this extension.

4. What should GetBooleanv return when COLOR_WRITEMASK is queried?

RESOLVED: COLOR_WRITEMASK should return DRAW_BUFFER0_COLOR_WRITEMASK_EXT. This is backwards compatible with the way the query works without this extension. To query the writemask associated with a particular draw buffer, an application can use GetBooleanIndexedvEXT.

5. How are separate blend enables controlled? Should a new function be introduced, or do Enable() and Disable() provide sufficient functionality?

RESOLVED: This extension introduces new functions EnableIndexedEXT and DisableIndexedEXT that can be used to enable/disable individual states of a state array. These functions are introduced because there is a trend towards introducing arrays of state. Rather than creating enums for each index in the array, it is better to give applications a mechanism for accessing a particular element of the state array given the name of the state and an index into the array.

6. What effect does enabling or disabling blending using BLEND have on per draw buffer blend enables?

RESOLVED: BLEND, used with Enable() and Disable(), should enable or disable all per draw buffer blend enables. This is similar to the way that ColorMask() affects the per draw buffer write masks.

Revision History

None

Name

EXT_draw_instanced

Name Strings

GL_EXT_draw_instanced

Contact

Michael Gold, NVIDIA Corporation (gold 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: November 6, 2006
Author Revision: 1.4

Number

327

Dependencies

OpenGL 2.0 is required.

EXT_gpu_shader4 or NV_vertex_shader4 is required.

Overview

This extension provides the means to render multiple instances of an object with a single draw call, and an "instance ID" variable which can be used by the vertex program to compute per-instance values, typically an object's transform.

New Tokens

None

New Procedures and Functions

```
void DrawArraysInstancedEXT(enum mode, int first, sizei count,
        sizei primcount);
void DrawElementsInstancedEXT(enum mode, sizei count, enum type,
        const void *indices, sizei primcount);
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify section 2.8 (Vertex Arrays), p. 23

(insert before the final paragraph, p. 30)

The internal counter <instanceID> is a 32-bit integer value which may be read by a vertex program as <vertex.instance>, as described in section 2.X.3.2, or vertex shader as <gl_InstanceID>, as described in section 2.15.4.2. The value of this counter is always zero, except as noted below.

The command

```
void DrawArraysInstancedEXT(enum mode, int first, sizei count,
        sizei primcount);
```

behaves identically to DrawArrays except that <primcount> instances of the range of elements are executed and the value of <instanceID> advances for each iteration. It has the same effect as:

```
if (mode or count is invalid)
    generate appropriate error
else {
    for (i = 0; i < primcount; i++) {
        instanceID = i;
        DrawArrays(mode, first, count, i);
    }
    instanceID = 0;
}
```

The command

```
void DrawElementsInstancedEXT(enum mode, sizei count, enum type,
        const void *indices, sizei primcount);
```

behaves identically to DrawElements except that <primcount> instances of the set of elements are executed, and the value of <instanceID> advances for each iteration. It has the same effect as:

```
if (mode, count, or type is invalid )
    generate appropriate error
else {
    for (int i = 0; i < primcount; i++) {
        instanceID = i;
        DrawElements(mode, count, type, indices, i);
    }
    instanceID = 0;
}
```

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

The error INVALID_OPERATION is generated if DrawArraysInstancedEXT or DrawElementsInstancedEXT is called during display list compilation.

Dependencies on NV_vertex_program4

If NV_vertex_program4 is not supported, all references to vertex.instance are deleted.

Dependencies on EXT_gpu_shader4

If EXT_gpu_shader4 is not supported, all references to gl_InstanceID are deleted.

Errors

INVALID_ENUM is generated by DrawElementsInstancedEXT if <type> is not one of UNSIGNED_BYTE, UNSIGNED_SHORT or UNSIGNED_INT.

INVALID_VALUE is generated by DrawArraysInstancedEXT if <first> is less than zero.

Issues

(1) Should instanceID be provided by this extension, or should it be provided by EXT_gpu_shader4, thus creating a dependence on that spec?

Resolved: While this extension could stand alone, its utility would be limited without the additional functionality provided by EXT_gpu_shader4; also, the spec language is cleaner if EXT_gpu_shader4 assumes instanceID is always available, even if its value is always zero without this extension.

(2) Should MultiDrawArrays and MultiDrawElements affect the value of instanceID?

Resolved: No, this may cause implementation difficulties and is considered unlikely to provide any real benefit.

(3) Should DrawArraysInstanced and DrawElementsInstanced be compiled into display lists?

Resolved: No, calling these during display list compilation generate INVALID_OPERATION.

Revision History

None

Name

EXT_framebuffer_sRGB

Name Strings

GL_EXT_framebuffer_sRGB
GLX_EXT_framebuffer_sRGB
WGL_EXT_framebuffer_sRGB

Contributors

Herb (Charles) Kuta, Quantum3D

From the EXT_texture_sRGB specification...

Alain Bouchard, Matrox
Brian Paul, Tungsten Graphics
Daniel Vogel, Epic Games
Eric Werness, NVIDIA
Kiril Vidimce, Pixar
Mark J. Kilgard, NVIDIA
Pat Brown, NVIDIA
Yanjun Zhang, S3 Graphics
Jeremy Sandmel, Apple

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Date: November 6, 2006
Revision: 2

Number

337

Dependencies

OpenGL 1.1 required

This extension is written against the OpenGL 2.0 (September 7, 2004) specification.

WGL_EXT_extensions_string is required for WGL support.

WGL_EXT_pixel_format is required for WGL support.

ARB_color_buffer_float interacts with this extension.

EXT_framebuffer_object interacts with this extension.

EXT_texture_sRGB interacts with this extension.

ARB_draw_buffers interacts with this extension.

Overview

Conventionally, OpenGL assumes framebuffer color components are stored in a linear color space. In particular, framebuffer blending is a linear operation.

The sRGB color space is based on typical (non-linear) monitor characteristics expected in a dimly lit office. It has been standardized by the International Electrotechnical Commission (IEC) as IEC 61966-2-1. The sRGB color space roughly corresponds to 2.2 gamma correction.

This extension adds a framebuffer capability for sRGB framebuffer update and blending. When blending is disabled but the new sRGB updated mode is enabled (assume the framebuffer supports the capability), high-precision linear color component values for red, green, and blue generated by fragment coloring are encoded for sRGB prior to being written into the framebuffer. When blending is enabled along with the new sRGB update mode, red, green, and blue framebuffer color components are treated as sRGB values that are converted to linear color values, blended with the high-precision color values generated by fragment coloring, and then the blend result is encoded for sRGB just prior to being written into the framebuffer.

The primary motivation for this extension is that it allows OpenGL applications to render into a framebuffer that is scanned to a monitor configured to assume framebuffer color values are sRGB encoded. This assumption is roughly true of most PC monitors with default gamma correction. This allows applications to achieve faithful color reproduction for OpenGL rendering without adjusting the monitor's gamma correction.

New Procedures and Functions

None

New Tokens

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

GLX_FRAMEBUFFER_SRGB_CAPABLE_EXT	0x20B2

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

WGL_FRAMEBUFFER_SRGB_CAPABLE_EXT	0x20A9

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

FRAMEBUFFER_SRGB_EXT	0x8DB9

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

FRAMEBUFFER_SRGB_CAPABLE_EXT	0x8DBA

Additions to Chapter 2 of the 2.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 2.0 Specification (Rasterization)

None

Additions to Chapter 4 of the 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

DELETE the following sentence from section 4.1.8 (Blending) because it is moved to the new "sRGB Conversion" section:

"Each of these floating-point values is clamped to [0,1] and converted back to a fixed-point value in the manner described in section 2.14.9."

If ARB_color_buffer_float is supported, the following paragraph is modified to eliminate the fixed-point clamping and conversion because this behavior is moved to the new "sRGB Conversion" section.

"If the color buffer is fixed-point, the components of the source and destination values and blend factors are clamped to [0, 1] prior to evaluating the blend equation, the components of the blending result are clamped to [0,1] and converted to fixed-point values in the manner described in section 2.14.9. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation."

The modified ARB_color_buffer_float paragraph should read:

"If the color buffer is fixed-point, the components of the source and destination values and blend factors are clamped to [0, 1] prior to evaluating the blend equation. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation."

Replace the following sentence:

"Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme in section 2.14.9 (Final Color Processing), as are source (fragment) components."

with the following sentences:

"Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme in section 2.14.9 (Final Color Processing). If FRAMEBUFFER_SRGB_EXT is enabled and the boolean FRAMEBUFFER_SRGB_CAPABLE_EXT state for the drawable is true, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence need to be linearized prior to their use in blending. Each R, G, and B component is linearized by some approximation of the following:

```
        {  cs / 12.92,                cs <= 0.04045
   cl = {
        {  ((cs + 0.055)/1.055)^2.4,  cs >  0.04045
```

where cs is the component value prior to linearization and cl is the result. Otherwise if FRAMEBUFFER_SRGB_EXT is disabled, or the drawable is not sRGB capable, or the value corresponds to the A component, then cs = cl for such components. The corresponding cs values for R, G, B, and A are recombined as the destination color used subsequently by blending."

ADD new section 4.1.X "sRGB Conversion" after section 4.1.8 (Blending) and before section 4.1.9 (Dithering). With this new section added, understand the "next operation" referred to in the section 4.1.8 (Blending) to now be "sRGB Conversion" (instead of "Dithering").

"If FRAMEBUFFER_SRGB_EXT is enabled and the boolean FRAMEBUFFER_SRGB_CAPABLE_EXT state for the drawable is true, the R, G, and B values after blending are converted into the non-linear sRGB color space by some approximation of the following:

```
         {  0.0,                          0         <= cl
         {  12.92 * c,                    0         <  cl < 0.0031308
    cs = {  1.055 * cl^0.41666 - 0.055,   0.0031308 <= cl < 1
         {  1.0,                                        cl >= 1
```

where cl is the R, G, or B element and cs is the result (effectively converted into an sRGB color space). Otherwise if FRAMEBUFFER_SRGB_EXT is disabled, or the drawable is not sRGB capable, or the value corresponds to the A element, then cs = cl for such elements.

The resulting cs values form a new RGBA color value. If the color buffer is fixed-point, the components of this RGBA color value are clamped to [0,1] and then converted to a fixed-point value in the manner described in section 2.14.9. The resulting four values are sent to the subsequent dithering operation."

Additions to Chapter 5 of the 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 2.0 Specification (State and State Requests)

None

Additions to the OpenGL Shading Language specification

None

Additions to the GLX Specification

None

Dependencies on ARB_color_buffer_float

If ARB_color_buffer_float is not supported, ignore the edits to ARB_color_buffer_float language.

Dependencies on EXT_texture_sRGB and EXT_framebuffer_object

If EXT_texture_sRGB and EXT_framebuffer_object are both supported, the implementation should set FRAMEBUFFER_SRGB_CAPABLE_EXT to false when rendering to a color texture that is not one of the EXT_texture_sRGB introduced internal formats. An implementation can determine whether or not it will set FRAMEBUFFER_SRGB_CAPABLE_EXT to true for the EXT_texture_sRGB introduced internal formats. Implementations are encouraged to allow sRGB update and blending when rendering to sRGB textures using EXT_framebuffer_object but this is not required. In any case, FRAMEBUFFER_SRGB_CAPABLE_EXT should indicate whether or not sRGB update and blending is supported.

Dependencies on ARB_draw_buffers, EXT_texture_sRGB, and EXT_framebuffer_object

If ARB_draw_buffers, EXT_texture_sRGB, and EXT_framebuffer_object are supported and an application attempts to render to a set of color buffers where some but not all of the color buffers are FRAMEBUFFER_SRGB_CAPABLE_EXT individually, the query of FRAMEBUFFER_SRGB_CAPABLE_EXT should return true.

However sRGB update and blending only apply to the color buffers that are actually sRGB-capable.

GLX Protocol

None.

Errors

Relaxation of INVALID_ENUM errors

Enable, Disable, IsEnabled, GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev now accept the new token as allowed in the "New Tokens" section.

New State

Add to table 6.20 (Pixel Operations)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_SRGB_EXT	B	IsEnabled	False	sRGB update and blending enable	4.1.X	color-buffer/enable

Add to table 6.33 (Implementation Dependent Values)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_SRGB_CAPABLE_EXT	B	IsEnabled	-	true if drawable supports sRGB update and blending	4.1.X	-

New Implementation Dependent State

None

Issues

1) What should this extension be called?

RESOLVED: EXT_framebuffer_sRGB.

The "EXT_framebuffer" part indicates the extension is in the framebuffer domain and "sRGB" indicates the extension is adding a set of sRGB formats. This mimics the naming of the EXT_texture_sRGB extension that adds sRGB texture formats.

The mixed-case spelling of sRGB is the established usage so "_sRGB" is preferred to "_srgb". The "s" stands for standard (color space).

For token names, we use "SRGB" since token names are uniformly capitalized.

2) Should alpha be sRGB encoded?

RESOLVED: No. Alpha remains linear.

A rationale for this resolution is found in Alvy Ray's "Should Alpha Be Nonlinear If RGB Is?" Tech Memo 17 (December 14, 1998). See: ftp://ftp.alvyray.com/Acrobat/17_Nonln.pdf

3) Should the ability to support sRGB framebuffer update and blending be an attribute of the framebuffer?

RESOLVED: Yes. It should be a capability of some pixel formats (mostly likely just RGB8 and RGBA8) that says sRGB blending can be enabled.

This allows an implementation to simply mark the existing RGB8 and RGBA8 pixel formats as supporting sRGB blending and then

just provide the functionality for sRGB update and blending for such formats.

sRGB support for floating-point formats makes little sense (because floating-point already provide a non-linear distribution of precision and typically have considerably more precision than 8-bit fixed-point framebuffer components allow) and would be expensive to support.

Requiring sRGB support for all fixed-point buffers means that support for 16-bit components or very small 5-bit or 6-bit components would require special sRGB conversion hardware. Typically sRGB is well-suited for 8-bit fixed-point components so we do not want this extension to require expensive tables for other component sizes that are unlikely to ever be used. Implementations could support sRGB conversion for any color framebuffer format but implementations are not required to (honestly nor are implementations like to support sRGB on anything but 8-bit fixed-point color formats).

4) Should there be an enable for sRGB update and blending?

RESOLVED: Yes, and it is disabled by default. The enable only applies if the framebuffer's underlying pixel format is capable of sRGB update and blending. Otherwise, the enable is silently ignored (similar to how the multisample enables are ignored when the pixel format lacks multisample supports).

5) How is sRGB blending done?

RESOLVED: Blending is a linear operation so should be performed on values in linear spaces. sRGB-encoded values are in a non-linear space so sRGB blending should convert sRGB-encoded values from the framebuffer to linear values, blend, and then sRGB-encode the result to store it in the framebuffer.

The destination color RGB components are each converted from sRGB to a linear value. Blending is then performed. The source color and constant color are simply assumed to be treated as linear color components. Then the result of blending is converted to an sRGB encoding and stored in the framebuffer.

6) What happens if GL_FRAMEBUFFER_SRGB_EXT is enabled (and GL_FRAMEBUFFER_SRGB_CAPABLE_EXT is true for the drawable) but GL_BLEND is not enabled?

RESOLVED: The color result from fragment coloring (the source color) is converted to an sRGB encoding and stored in the framebuffer.

7) How are multiple render targets handled?

RESOLVED: Render targets that are not GL_FRAMEBUFFER_SRGB_CAPABLE_EXT ignore the state of the GL_FRAMEBUFFER_SRGB_EXT enable for sRGB update and blending. So only the render targets that are sRGB-capable perform sRGB blending and update when GL_FRAMEBUFFER_SRGB_EXT is enabled.

8) Should sRGB framebuffer support affect the pixel path?

RESOLVED: No.

sRGB conversion only applies to color reads for blending and color writes. Color reads for glReadPixels, glCopyPixels, or glAccum have no sRGB conversion applied.

For pixel path operations, an application could use pixel maps or color tables to perform an sRGB-to-linear conversion with these lookup tables.

9) Can luminance (single color component) framebuffer formats support sRGB blending?

RESOLVED: Yes, if an implementation chooses to advertise such a format and set the sRGB attribute for the format too.

Implementations are not obliged to provide such formats.

10) Should all component sizes be supported for sRGB components or just 8-bit?

RESOLVED: This is at the implementation's discretion since the implementation decides what pixel formats such support sRGB update and blending.

It likely implementations will only provide sRGB-capable framebuffer configurations for configurations with 8-bit components.

11) What must be specified as far as how do you convert to and from sRGB and linear RGB color spaces?

RESOLVED: The specification language needs to only supply the linear RGB to sRGB conversion (see section 4.9.X above).

The sRGB to linear RGB conversion is documented in the EXT_texture_sRGB specification.

For completeness, the accepted linear RGB to sRGB conversion (the inverse of the function specified in section 3.8.x) is as follows:

Given a linear RGB component, cl, convert it to an sRGB component, cs, in the range [0,1], with this pseudo-code:

```
if (isnan(cl)) {
    /* Map IEEE-754 Not-a-number to zero. */
    cs = 0.0;
} else if (cl > 1.0) {
    cs = 1.0;
} else if (cl < 0.0) {
    cs = 0.0;
} else if (cl < 0.0031308) {
    cs = 12.92 * cl;
} else {
    cs = 1.055 * pow(cl, 0.41666) - 0.055;
}
```

The NaN behavior in the pseudo-code is recommended but not specified in the actual specification language.

sRGB components are typically stored as unsigned 8-bit fixed-point values. If cs is computed with the above pseudo-code, cs can be converted to a [0,255] integer with this formula:

```
csi = floor(255.0 * cs + 0.5)
```

12) Does this extension guarantee images rendered with sRGB textures will "look good" when output to a device supporting an sRGB color space?

RESOLVED: No.

Whether the displayed framebuffer is displayed to a monitor that faithfully reproduces the sRGB color space is beyond the scope of this extension. This involves the gamma correction and color calibration of the physical display device.

13) How does this extension interact with EXT_framebuffer_object?

RESOLVED: When rendering to a color texture, an application can query GL_FRAMEBUFFER_SRGB_CAPABLE_EXT to determine if the color texture image is capable of sRGB rendering.

This boolean should be false for all texture internal formats except may be true (but are not required to be true) for the sRGB internal formats introduced by EXT_texture_sRGB. The expectation is that implementations of this extension will be able to support sRGB update and blending of sRGB textures.

14) How is the constant blend color handled for sRGB framebuffers?

RESOLVED: The constant blend color is specified as four floating-point values. Given that the texture border color can be specified at such high precision, it is always treated as a linear RGBA value.

15) How does glCopyTex[Sub]Image work with sRGB? Suppose we're rendering to a floating point pbuffer or framebuffer object and do CopyTexImage. Are the linear framebuffer values converted to sRGB during the copy?

RESOLVED: No, linear framebuffer values will NOT be automatically converted to the sRGB encoding during the copy. If such a conversion is desired, as explained in issue 12, the red, green, and blue pixel map functionality can be used to implement a linear-to-sRGB encoding translation.

16) Should this extension explicitly specify the particular sRGB-to-linear and linear-to-sRGB conversions it uses?

RESOLVED: The conversions are explicitly specified but allowance for approximations is provided. The expectation is that the implementation is likely to use a table to implement the conversions the conversion is necessarily then an approximation.

17) How does this extension interact with multisampling?

RESOLVED: There are no explicit interactions. However, arguably if the color samples for multisampling are sRGB encoded, the samples should be linearized before being "resolved" for display and then recoverted to sRGB if the output device expects sRGB encoded color components.

This is really a video scan-out issue and beyond the scope of this extension which is focused on the rendering issues. However some implementation advice is provided:

The implementation sufficiently aware of the gamma correction configured for the display device could decide to perform an sRGB-correct multisample resolve. Whether this occurs or not could be determined by a control panel setting or inferred by the application's use of this extension.

18) Why is the sRGB framebuffer GL_FRAMEBUFFER_SRGB_EXT enable disabled by default?

RESOLVED: This extension could have a boolean sRGB-versus-non-sRGB pixel format configuration mode that determined whether or not sRGB framebuffer update and blending occurs. The problem with this approach is 1) it creates may more pixel formation configurations because sRGB and non-sRGB versions of lots of existing configurations must be advertised, and 2) applicaitons unaware of sRGB might unknowingly select an sRGB configuration and then generate over-bright rendering.

It seems more appropriate to have a capability for sRGB framebuffer update and blending that is disabled by default. This allows existing RGB8 and RGBA8 framebuffer configurations to be marked as sRGB capable (so no additional configurations need be enumerated). Applications that desire sRGB rendering should identify an sRGB-capable framebuffer configuration and then enable sRGB rendering.

This is different from how EXT_texture_sRGB handles sRGB support for texture formats. In the EXT_texture_sRGB extension, textures are either sRGB or non-sRGB and there is no texture parameter to switch textures between the two modes. This makes sense for EXT_texture_sRGB because it allows implementations to fake sRGB textures with higher-precision linear textures that simply convert sRGB-encoded texels to sufficiently precise linear RGB values.

Texture formats also don't have the problem enumerated pixel format descriptions have where a naive application could stumble upon an sRGB-capable pixel format. sRGB textures require explicit use of one of the new EXT_texture_sRGB-introduced internal formats.

19) How does sRGB and this extension interact with digital video output standards, in particular DVI?

RESOLVED: The DVI 1.0 specification recommends "as a default position that digital moniotrs of all types support a color transfer function similar to analog CRT monitors (gamma=2.2) which makes up the majority of the compute display market." This means DVI output devices should benefit from blending in the sRGB color space just like analog monitors.

Revision History

None

Name

EXT_geometry_shader4

Name String

GL_EXT_geometry_shader4

Contact

Pat Brown, NVIDIA (pbrown 'at' nvidia.com)
Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Status

Multi-vendor extension

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 05/22/2007
NVIDIA Revision: 17

Number

324

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 2.0 specification.

EXT_framebuffer_object interacts with this extension.

EXT_framebuffer_blit interacts with this extension.

EXT_texture_array interacts with this extension.

ARB_texture_rectangle trivially affects the definition of this extension.

EXT_texture_buffer_object trivially affects the definition of this extension.

NV_primitive_restart trivially affects the definition of this extension.

This extension interacts with EXT_tranform_feedback.

Overview

EXT_geometry_shader4 defines a new shader type available to be run on the GPU, called a geometry shader. Geometry shaders are run after vertices are transformed, but prior to color clamping, flat shading and clipping.

A geometry shader begins with a single primitive (point, line, triangle). It can read the attributes of any of the vertices in the primitive and use them to generate new primitives. A geometry shader has a fixed output primitive type (point, line strip, or triangle strip) and emits vertices to define a new primitive. A geometry shader can emit multiple disconnected primitives. The primitives emitted by the geometry shader are clipped and then processed like an equivalent OpenGL primitive specified by the application.

Furthermore, EXT_geometry_shader4 provides four additional primitive types: lines with adjacency, line strips with adjacency, separate triangles with adjacency, and triangle strips with adjacency. Some of the vertices specified in these new primitive types are not part of the ordinary primitives, instead they represent neighboring vertices that are adjacent to the two line segment end points (lines/strips) or the three triangle edges (triangles/tstrips). These vertices can be accessed by geometry shaders and used to match up the vertices emitted by the geometry shader with those of neighboring primitives.

Since geometry shaders expect a specific input primitive type, an error will occur if the application presents primitives of a different type. For example, if a geometry shader expects points, an error will occur at Begin() time, if a primitive mode of TRIANGLES is specified.

New Procedures and Functions

```
void ProgramParameteriEXT(uint program, enum pname, int value);
void FramebufferTextureEXT(enum target, enum attachment,
                           uint texture, int level);
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
void FramebufferTextureFaceEXT(enum target, enum attachment,
                               uint texture, int level, enum face);
```

New Tokens

Accepted by the <type> parameter of CreateShader and returned by the <params> parameter of GetShaderiv:

GEOMETRY_SHADER_EXT	0x8DD9

Accepted by the <pname> parameter of ProgramParameteriEXT and GetProgramiv:

GEOMETRY_VERTICES_OUT_EXT	0x8DDA
GEOMETRY_INPUT_TYPE_EXT	0x8DDB
GEOMETRY_OUTPUT_TYPE_EXT	0x8DDC

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

MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT	0x8C29
MAX_GEOMETRY_VARYING_COMPONENTS_EXT	0x8DDD
MAX_VERTEX_VARYING_COMPONENTS_EXT	0x8DDE
MAX_VARYING_COMPONENTS_EXT	0x8B4B
MAX_GEOMETRY_UNIFORM_COMPONENTS_EXT	0x8DDF
MAX_GEOMETRY_OUTPUT_VERTICES_EXT	0x8DE0
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT	0x8DE1

Accepted by the <mode> parameter of Begin, DrawArrays, MultiDrawArrays, DrawElements, MultiDrawElements, and DrawRangeElements:

LINES_ADJACENCY_EXT	0xA
LINE_STRIP_ADJACENCY_EXT	0xB
TRIANGLES_ADJACENCY_EXT	0xC
TRIANGLE_STRIP_ADJACENCY_EXT	0xD

Returned by CheckFramebufferStatusEXT:

FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT	0x8DA8
FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT	0x8DA9

Accepted by the <pname> parameter of GetFramebufferAttachment-ParameterivEXT:

FRAMEBUFFER_ATTACHMENT_LAYERED_EXT	0x8DA7
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT	0x8CD4

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

PROGRAM_POINT_SIZE_EXT	0x8642

(Note: FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is simply an alias for the FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT token provided in EXT_framebuffer_object. This extension generalizes the notion of "<zoffset>" to include layers of an array texture.)

(Note: PROGRAM_POINT_SIZE_EXT is simply an alias for the VERTEX_PROGRAM_POINT_SIZE token provided in OpenGL 2.0, which is itself an alias for VERTEX_PROGRAM_POINT_SIZE_ARB provided by ARB_vertex_program. Program-computed point sizes can be enabled if geometry shaders are enabled.)

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.6.1 (Begin and End Objects), p. 13

(Add to end of section, p. 18)

(add figure)

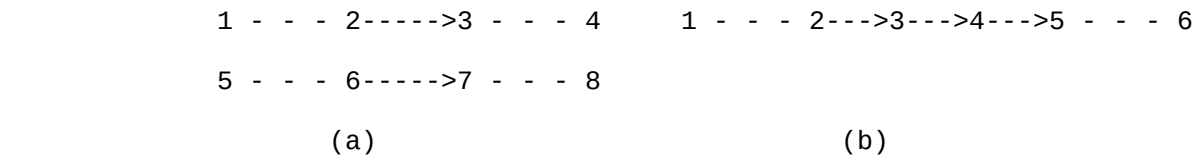


Figure 2.X1 (a) Lines with adjacency, (b) Line strip with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Lines with Adjacency

Lines with adjacency are independent line segments where each endpoint has a corresponding "adjacent" vertex that can be accessed by a geometry shader (Section 2.16). If a geometry shader is not active, the "adjacent" vertices are ignored.

A line segment is drawn from the 4i + 2nd vertex to the 4i + 3rd vertex for each i = 0, 1, ... , n-1, where there are 4n+k vertices between the Begin and End. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the 4i + 1st and 4i + 4th vertices are considered adjacent to the 4i + 2nd and 4i + 3rd vertices, respectively. See Figure 2.X1.

Lines with adjacency are generated by calling Begin with the argument value LINES_ADJACENCY_EXT.

Line Strips with Adjacency

Line strips with adjacency are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by a geometry shader (Section 2.15). If a geometry shader is not active, the "adjacent" vertices are ignored.

A line segment is drawn from the i + 2nd vertex to the i + 3rd vertex for each i = 0, 1, ..., n-1, where there are n+3 vertices between the Begin and End. If there are fewer than four vertices between a Begin and End, all vertices are ignored. For line segment i, the i + 1st and i + 4th vertex are considered adjacent to the i + 2nd and i + 3rd vertices, respectively. See Figure 2.X1.

Line strips with adjacency are generated by calling Begin with the argument value LINE_STRIP_ADJACENCY_EXT.

(add figure)

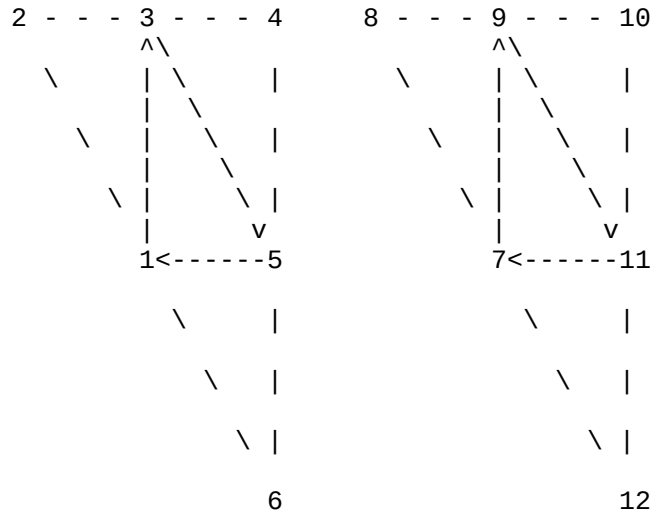


Figure 2.X2 Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Triangles with Adjacency

Triangles with adjacency are similar to separate triangles, except that each triangle edge has an adjacent vertex that can be accessed by a geometry shader (Section 2.15). If a geometry shader is not active, the "adjacent" vertices are ignored.

The 6i + 1st, 6i + 3rd, and 6i + 5th vertices (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 6n+k vertices between the Begin and End. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the i + 2nd, i + 4th, and i + 6th vertices are considered adjacent to edges from the i + 1st to the i + 3rd, from the i + 3rd to the i + 5th, and from the i + 5th to the i + 1st vertices, respectively. See Figure 2.X2.

Triangles with adjacency are generated by calling Begin with the argument value TRIANGLES_ADJACENCY_EXT.

(add figure)

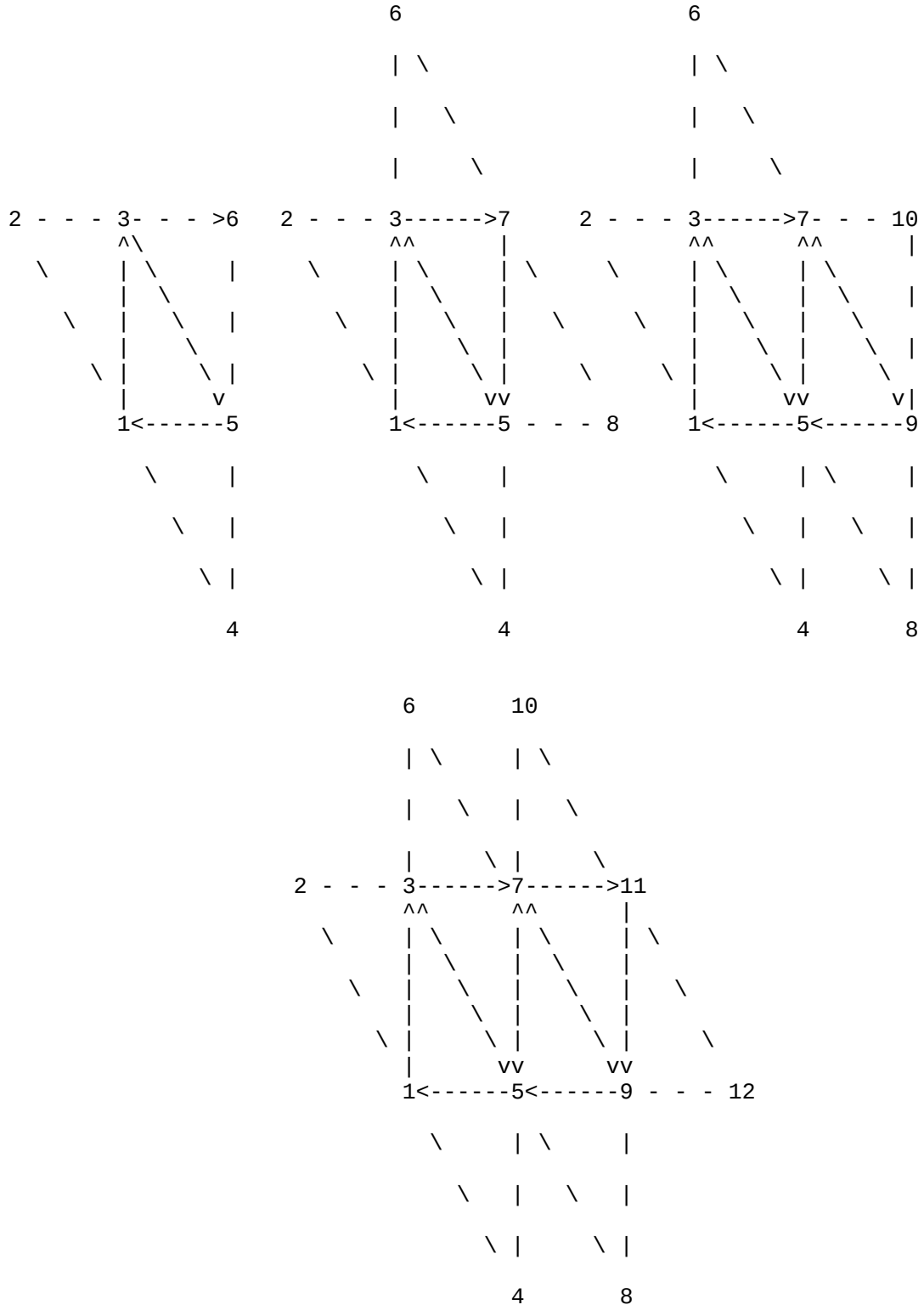


Figure 2.X3 Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Triangle Strips with Adjacency

Triangle strips with adjacency are similar to triangle strips, except that each line triangle edge has an adjacent vertex that can be accessed by a geometry shader (Section 2.15). If a geometry shader is not active, the "adjacent" vertices are ignored.

In triangle strips with adjacency, n triangles are drawn using 2 * (n+2) + k vertices between the Begin and End. k is either 0 or 1; if k is 1, the final vertex is ignored. If fewer than 6 vertices are specified between the Begin and End, the entire primitive is ignored. Table 2.X1 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle. See Figure 2.X3.

(add table)

	primitive vertices			adjacent vertices		
primitive	1st	2nd	3rd	1/2	2/3	3/1
only (i==0, n==1)	1	3	5	2	6	4
first (i==0)	1	3	5	2	7	4
middle (i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+7
middle (i even)	2i+1	2i+3	2i+5	2i-1	2i+7	2i+4
last (i==n-1, i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+6
last (i==n-1, i even)	2i+1	2i+3	2i+5	2i-1	2i+6	2i+4

Table 2.X1: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices in the "1st", "2nd", and "3rd" columns under "primitive vertices", in that order. The vertices in the "1/2", "2/3", and "3/1" columns under "adjacent vertices" are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to the six cases: the first and only triangle (i=0, n=1), the first triangle of several (i=0, n>0), "odd" middle triangles (i=1,3,5...), "even" middle triangles (i=2,4,6,...), and special cases for the last triangle inside the Begin/End, when i is either even or odd. For the purposes of this table, the first vertex specified after Begin is numbered "1" and the first triangle is numbered "0".

Triangle strips with adjacency are generated by calling Begin with the argument value TRIANGLE_STRIP_ADJACENCY_EXT.

Modify Section 2.14.1, Lighting (p. 59)

(modify fourth paragraph, p. 63) Additionally, vertex and geometry shaders can operate in two-sided color mode, which is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE. When a vertex or geometry shader is active, the shaders can write front and back color values to the gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor outputs. When a vertex or geometry shader is active and two-sided color mode is enabled, the GL chooses between front and back colors, as described below. If two-sided color mode is disabled, the front color output is always selected.

Modify Section 2.15.2 Program Objects, p. 73

Change the first paragraph on p. 74 as follows:

Program objects are empty when they are created. Default values for program object parameters are discussed in section 2.15.5, Required State. A non-zero name that can be used to reference the program object is returned.

Change the language below the LinkProgram command on p. 74 as follows:

... Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to <program> are not compiled successfully, or if more active uniform or active sampler variables are used in <program> than allowed (see sections 2.15.3 and 2.16.3). Linking will also fail if the program object contains objects to form a geometry shader (see section 2.16), but no objects to form a vertex shader or if the program object contains objects to form a geometry shader, and the value of GEOMETRY_VERTICES_OUT_EXT is zero. If LinkProgram failed, ..

Add the following paragraphs above the description of DeleteProgram, p. 75:

To set a program object parameter, call

```
void ProgramParameteriEXT(uint program, enum pname, int value)
```

<param> identifies which parameter to set for <program>. <value> holds the value being set. Legal values for <param> and <value> are discussed in section 2.16.

Modify Section 2.15.3, Shader Variables, p. 75

Modify the first paragraph of section 'Varying Variables' p. 83 as follows:

A vertex shader may define one or more varying variables (see the OpenGL Shading Language specification). Varying variables are outputs of a vertex shader. They are either used as the mechanism to communicate values to a geometry shader, if one is active, or to communicate values to the fragment shader. The OpenGL Shading Language specification also defines a set of built-in varying variables that vertex shaders can write to (see section 7.6 of the OpenGL Shading Language Specification). These variables can also be used to communicate values to a geometry shader, if one is active, or to communicate values to the fragment shader and to the fixed-function processing that occurs after vertex shading.

If a geometry shader is not active, the values of all varying variables, including built-in variables, are expected to be interpolated across the primitive being rendered, unless flat shaded. The number of interpolators available for processing varying variables is given by the implementation-dependent constant MAX_VARYING_COMPONENTS_EXT. This value represents the number of individual components that can be interpolated; varying variables declared as vectors, matrices, and arrays will all consume multiple interpolators. When a program is linked, all components of any varying variable written by a vertex shader, or read by a fragment

shader, will count against this limit. The transformed vertex position (gl_Position) does not count against this limit. A program whose vertex and/or fragment shaders access more than MAX_VARYING_COMPONENTS_EXT components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Note that the two values MAX_VARYING_FLOATS and MAX_VARYING_COMPONENTS_EXT are aliases of each other. The use of MAX_VARYING_FLOATS however is discouraged; varying variables can be declared as integers as well.

If a geometry shader is active, the values of varying variables are collected by the primitive assembly stage and passed on to the geometry shader once enough data for one primitive has been collected (see also section 2.16). The OpenGL Shading Language specification also defines a set of built-in varying and built-in special variables that vertex shaders can write to (see sections 7.1 and 7.6 of the OpenGL Shading Language Specification). These variables are also collected and passed on to the geometry shader once enough data has been collected. The number of components of varying and special variables that can be collected per vertex by the primitive assembly stage is given by the implementation dependent constant MAX_VERTEX_VARYING_COMPONENTS_EXT. This value represents the number of individual components that can be collected; varying variables declared as vectors, matrices, and arrays will all consume multiple components. When a program is linked, all components of any varying variable written by a vertex shader, or read by a geometry shader, will count against this limit. A program whose vertex and/or geometry shaders access more than MAX_VERTEX_VARYING_COMPONENTS_EXT components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Modify Section 2.15.4 Shader Execution, p. 84

Change the following sentence:

"The following operations are applied to vertex values that are the result of executing the vertex shader:"

As follows:

If no geometry shader (see section 2.16) is present in the program object, the following operations are applied to vertex values that are the result of executing the vertex shader:

[bulleted list of operations]

On page 85, below the list of bullets, add the following:

If a geometry shader is present in the program object, geometry shading (section 2.16) is applied to vertex values that are the result of executing the vertex shader.

Modify the first paragraph of the section 'Texture Access', p. 85, as follows:

Vertex shaders have the ability to do a lookup into a texture map, if supported by the GL implementation. The maximum number of texture image units available to a vertex shader is MAX_VERTEX_TEXTURE_IMAGE_UNITS; a maximum number of zero indicates that the GL implementation does not support texture accesses in vertex shaders. The vertex shader, geometry shader, if exists, and fragment processing combined cannot use more than MAX_COMBINED_TEXTURE_IMAGE_UNITS texture image units. If the vertex shader, geometry shader and the fragment processing stage access the same texture image unit, then that counts as using three texture image units against the MAX_COMBINED_TEXTURE_IMAGE_UNITS limit.

Modify Section 2.15.5, Required State, p. 88

Add the following bullets to the state required per program object:

* One integer to store the value of GEOMETRY_VERTICES_OUT_EXT, initially zero.

* One integer to store the value of GEOMETRY_INPUT_TYPE_EXT, initially set to TRIANGLES.

* One integer to store the value of GEOMETRY_OUTPUT_TYPE_EXT, initially set to TRIANGLE_STRIP.

Insert New Section 2.16, Geometry Shaders after p. 89

After vertices are processed, they are arranged into primitives, as described in section 2.6.1 (Begin/End Objects). This section described a new pipeline stage that processes those primitives. A geometry shader defines the operations that are performed in this new pipeline stage. A geometry shader is an array of strings containing source code. The source code language used is described in the OpenGL Shading Language specification. A geometry shader operates on a single primitive at a time and emits one or more output primitives, all of the same type, which are then processed like an equivalent OpenGL primitive specified by the application. The original primitive is discarded after the geometry shader completes. The inputs available to a geometry shader are the transformed attributes of all the vertices that belong to the primitive. Additional "adjacency" primitives are available which also make the transformed attributes of neighboring vertices available to the shader. The results of the shader are a new set of transformed vertices, arranged into primitives by the shader.

This new geometry shader pipeline stage is inserted after primitive assembly, right before color clamping (section 2.14.6), flat shading (section 2.14.7) and clipping (sections 2.12 and 2.14.8).

A geometry shader only applies when the GL is in RGB mode. Its operation in color index mode is undefined.

Geometry shaders are created as described in section 2.15.1 using a type parameter of GEOMETRY_SHADER_EXT. They are attached to and used in program objects as described in section 2.15.2. When the program object currently in use includes a geometry shader, its geometry shader is considered active, and is used to process primitives. If the program object has no geometry shader, or no program object is in use, this new primitive processing pipeline stage is bypassed.

A program object that includes a geometry shader must also include a vertex shader; otherwise a link error will occur.

Section 2.16.1, Geometry shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided by the GL. If a geometry shader is active, Begin, or any function that implicitly calls Begin, will produce an INVALID_OPERATION error if the <mode> parameter is incompatible with the input primitive type of the currently active program object, as discussed below.

The input primitive type is a parameter of the program object, and must be set before linking by calling ProgramParameteriEXT with <pname> set to GEOMETRY_INPUT_TYPE_EXT and <value> set to one of POINTS, LINES, LINES_ADJACENCY_EXT, TRIANGLES or TRIANGLES_ADJACENCY_EXT. This setting will not be in effect until the next time LinkProgram has been called successfully. Note that queries of GEOMETRY_INPUT_TYPE_EXT will return the last value set. This is not necessarily the value used to generate the executable code in the program object. After a program object has been created it will have a default value for GEOMETRY_INPUT_TYPE_EXT, as discussed in section 2.15.5, Required State.

Note that a geometry shader that accesses more input vertices than are available for a given input primitive type can be successfully compiled, because the input primitive type is not part of the shader object. However, a program object, containing a shader object that access more input vertices than are available for the input primitive type of the program object, will not link.

The supported input primitive types are:

Points (POINTS)

Geometry shaders that operate on points are valid only for the POINTS primitive type. There is only a single vertex available for each geometry shader invocation.

Lines (LINES)

Geometry shaders that operate on line segments are valid only for the LINES, LINE_STRIP, and LINE_LOOP primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 2.16.4.

Lines with Adjacency (LINES_ADJACENCY_EXT)

Geometry shaders that operate on line segments with adjacent vertices are valid only for the LINES_ADJACENCY_EXT and LINE_STRIP_ADJACENCY_EXT primitive types. There are four vertices available for each program invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at

the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (TRIANGLES)

Geometry shaders that operate on triangles are valid for the TRIANGLES, TRIANGLE_STRIP and TRIANGLE_FAN primitive types.

There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

Triangles with Adjacency (TRIANGLES_ADJACENCY_EXT)

Geometry shaders that operate on triangles with adjacent vertices are valid for the TRIANGLES_ADJACENCY_EXT and TRIANGLE_STRIP_ADJACENCY_EXT primitive types. There are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

Section 2.16.2, Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (POINTS), line strips (LINE_STRIP), and triangle strips (TRIANGLE_STRIP). The vertices output by the geometry shader are decomposed into points, lines, or triangles based on the output primitive type in the manner described in section 2.6.1. The resulting primitives are then further processed as shown in figure 2.16.xxx. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn.

The output primitive type is a parameter of the program object, and can be set by calling ProgramParameteriEXT with <pname> set to GEOMETRY_OUTPUT_TYPE_EXT and <value> set to one of POINTS, LINE_STRIP or TRIANGLE_STRIP. This setting will not be in effect until the next time LinkProgram has been called successfully. Note that queries of GEOMETRY_OUTPUT_TYPE_EXT will return the last value set; which is not necessarily the value used to generate the executable code in the program object. After a program object has been created it will have a default value for GEOMETRY_OUTPUT_TYPE_EXT, as discussed in section 2.15.5, Required State. .

Section 2.16.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. The amount of storage available for geometry shader uniform variables is specified by the implementation dependent constant MAX_GEOMETRY_UNIFORM_COMPONENTS_EXT. This value represents the number of individual floating-point, integer, or Boolean values that can be held in uniform variable storage for a geometry shader. A link error will be generated if an attempt is made to utilize more than the space available for geometry shader uniform variables. Uniforms are manipulated as described in section 2.15.3. Geometry shaders also have access to

samplers, to perform texturing operations, as described in sections 2.15.3 and 3.8.

Geometry shaders can access the transformed attributes of all vertices for its input primitive type through input varying variables. A vertex shader, writing to output varying variables, generates the values of these input varying variables. This includes values for built-in as well as user-defined varying variables. Values for any varying variables that are not written by a vertex shader are undefined. Additionally, a geometry shader has access to a built-in variable that holds the ID of the current primitive. This ID is generated by the primitive assembly stage that sits in between the vertex and geometry shader.

Additionally, geometry shaders can write to one, or more, varying variables for each primitive it outputs. These values are optionally flat shaded (using the OpenGL Shading Language varying qualifier "flat") and clipped, then the clipped values interpolated across the primitive (if not flat shaded). The results of these interpolations are available to a fragment shader, if one is active. Furthermore, geometry shaders can write to a set of built- in varying variables, defined in the OpenGL Shading Language, that correspond to the values required for the fixed-function processing that occurs after geometry processing.

Section 2.16.4, Geometry Shader Execution Environment

If a successfully linked program object that contains a geometry shader is made current by calling UseProgram, the executable version of the geometry shader is used to process primitives resulting from the primitive assembly stage.

The following operations are applied to the primitives that are the result of executing a geometry shader:

* color clamping or masking (section 2.14.6),

* flat shading (section 2.14.7),

* clipping, including client-defined clip planes (section 2.12),

* front face determination (section 2.14.1),

* color and associated data clipping (section 2.14.8),

* perspective division on clip coordinates (section 2.11),

* final color processing (section 2.14.9), and

* viewport transformation, including depth-range scaling (section 2.11.1).

There are several special considerations for geometry shader execution described in the following sections.

Texture Access

Geometry shaders have the ability to do a lookup into a texture map, if supported by the GL implementation. The maximum number of texture image

units available to a geometry shader is MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT; a maximum number of zero indicates that the GL implementation does not support texture accesses in geometry shaders.

The vertex shader, geometry shader and fragment processing combined cannot use more than MAX_COMBINED_TEXTURE_IMAGE_UNITS texture image units. If the vertex shader, geometry shader and the fragment processing stage access the same texture image unit, then that counts as using three texture image units against the MAX_COMBINED_TEXTURE_IMAGE_UNITS limit.

When a texture lookup is performed in a geometry shader, the filtered texture value tau is computed in the manner described in sections 3.8.8 and 3.8.9, and converted to a texture source color Cs according to table 3.21 (section 3.8.13). A four component vector (Rs,Gs,Bs,As) is returned to the geometry shader. In a geometry shader it is not possible to perform automatic level-of- detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.8.8. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled by a level-of-detail value optionally passed as an argument in the texture lookup functions. If the texture lookup function supplies an explicit level-of-detail value lambda, then the pre-bias level-of-detail value LAMBDAbase(x, y) = lambda (replacing equation 3.18). If the texture lookup function does not supply an explicit level-of-detail value, then LAMBDAbase(x, y) = 0. The scale factor Rho(x, y) and its approximation function f(x, y) (see equation 3.21) are ignored.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* the sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE;

* the sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE; or

* the sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

If a geometry shader uses a sampler where the associated texture object is not complete as defined in section 3.8.10, the texture image unit will return (R,G,B,A) = (0, 0, 0, 1).

Geometry Shader Inputs

The OpenGL Shading Language specification describes the set of built-in variables that are available as inputs to the geometry shader. This set receives the values from the equivalent built-in output variables written

by the vertex shader. These built-in variables are arrays; each element in the array holds the value for a specific vertex of the input primitive. The length of each array depends on the value of the input primitive type, as determined by the program object value GEOMETRY_INPUT_TYPE_EXT, and is set by the GL during link. Each built-in variable is a one-dimensional array, except for the built-in texture coordinate variable, which is a two- dimensional array. The vertex shader built-in output gl_TexCoord[] is a one-dimensional array. Therefore, the geometry shader equivalent input variable gl_TexCoordIn[][] becomes a two-dimensional array. See the OpenGL Shading Language Specification, sections 4.3.6 and 7.6 for more information.

The built-in varying variables gl_FrontColorIn[], gl_BackColorIn[], gl_FrontSecondaryColorIn[] and gl_BackSecondaryColorIn[] hold the per-vertex front and back colors of the primary and secondary colors, as written by the vertex shader to its equivalent built-in output variables.

The built-in varying variable gl_TexCoordIn[][] holds the per- vertex values of the array of texture coordinates, as written by the vertex shader to its built-in output array gl_TexCoord[].

The built-in varying variable gl_FogFragCoordIn[] holds the per- vertex fog coordinate, as written by the vertex shader to its built- in output variable gl_FogFragCoord.

The built-in varying variable gl_PositionIn[] holds the per-vertex position, as written by the vertex shader to its output variable gl_Position. Note that writing to gl_Position from either the vertex or fragment shader is optional. See also section 7.1 "Vertex and Geometry Shader Special Variables" of the OpenGL Shading Language specification.

The built-in varying variable gl_ClipVertexIn[] holds the per-vertex position in clip coordinates, as written by the vertex shader to its output variable gl_ClipVertex.

The built-in varying variable gl_PointSizeIn[] holds the per-vertex point size written by the vertex shader to its built-in output varying variable gl_PointSize. If the vertex shader does not write gl_PointSize, the value of gl_PointSizeIn[] is undefined, regardless of the value of the enable VERTEX_PROGRAM_POINT_SIZE.

The built-in special variable gl_PrimitiveIDIn is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in varying variables, user-defined input varying variables need to be declared as arrays. Declaring a size is optional. If no size is specified, it will be inferred by the linker from the input primitive type. If a size is specified, it has to be of the size matching the number of vertices of the input primitive type, otherwise a link error

will occur. The built-in variable gl_VerticesIn, if so desired, can be used to size the array correctly for each input primitive type. User-defined varying variables can be declared as arrays in the vertex shader. This means that those, on input to the geometry shader, must be declared as two-dimensional arrays. See sections 4.3.6 and 7.6 of the OpenGL Shading Language Specification for more information.

Using any of the built-in or user-defined input varying variables can count against the limit MAX_VERTEX_VARYING_COMPONENTS_EXT as discussed in section 2.15.3.

Geometry Shader outputs

A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit needs to be set as a parameter of the program object that contains the geometry shader. To do so, call ProgramParameteriEXT with <pname> set to GEOMETRY_VERTICES_OUT_EXT and <value> set to the maximum number of vertices the geometry shader will emit in one invocation. This setting will not be guaranteed to be in effect until the next time LinkProgram has been called successfully. If a geometry shader, in one invocation, emits more vertices than the value GEOMETRY_VERTICES_OUT_EXT, these emits may have no effect.

There are two implementation-dependent limits on the value of GEOMETRY_VERTICES_OUT_EXT. First, the error INVALID_VALUE will be generated by ProgramParameteriEXT if the number of vertices specified exceeds the value of MAX_GEOMETRY_OUTPUT_VERTICES_EXT. Second, the product of the total number of vertices and the sum of all components of all active varying variables may not exceed the value of MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT. LinkProgram will fail if it determines that the total component limit would be violated.

A geometry shader can write to built-in as well as user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. In order to seamlessly be able to insert or remove a geometry shader from a program object, the rules, names and types of the output built-in varying variables and user-defined varying variables are the same as for the vertex shader. Refer to section 2.15.3 and the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

The built-in output variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output variable gl_TexCoord[] is an array and holds the set of texture coordinates for the current vertex.

The built-in output variable gl_FogFragCoord is used as the "c" value, as described in section 3.10 "Fog" of the OpenGL 2.0 specification.

The built-in special variable gl_Position is intended to hold the homogeneous vertex position. Writing gl_Position is optional.

The built-in special variable gl_ClipVertex holds the vertex coordinate used in the clipping stage, as described in section 2.12 "Clipping" of the OpenGL 2.0 specification.

The built-in special variable gl_PointSize, if written, holds the size of the point to be rasterized, measured in pixels.

Additionally, a geometry shader can write to the built-in special variables gl_PrimitiveID and gl_Layer, whereas a vertex shader cannot. The built-in gl_PrimitiveID provides a single integer that serves as a primitive identifier. This written primitive ID is available to fragment shaders. If a fragment shader using primitive IDs is active and a geometry shader is also active, the geometry shader must write to gl_PrimitiveID or the primitive ID number is undefined. The built-in variable gl_Layer is used in layered rendering, and discussed in the next section.

The number of components available for varying variables is given by the implementation-dependent constant MAX_GEOMETRY_VARYING_COMPONENTS_EXT. This value represents the number of individual components of a varying variable; varying variables declared as vectors, matrices, and arrays will all consume multiple components. When a program is linked, all components of any varying variable written by a geometry shader, or read by a fragment shader, will count against this limit. The transformed vertex position (gl_Position) does not count against this limit. A program whose geometry and/or fragment shaders access more than MAX_GEOMETRY_VARYING_COMPONENTS_EXT worth of varying variable components may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Layered rendering

Geometry shaders can be used to render to one of several different layers of cube map textures, three-dimensional textures, plus one- dimensional and two-dimensional texture arrays. This functionality allows an application to bind an entire "complex" texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, this mechanism can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable gl_layer. Layered rendering requires the use of framebuffer objects. Refer to the section 'Dependencies on EXT_framebuffer_object' for details.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.3, Points (p. 95)

(replace all Section 3.3 text on p. 95)

A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If no vertex or geometry shader is active, the size of the point is controlled by

```
void PointSize(float size);
```

<size> specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

```
derived size = clamp(size * sqrt(1/(a+b*d+c*d^2)))
```

where d is the eye-coordinate distance from the eye, (0,0,0,1) in eye coordinates, to the vertex, and a, b, and c are distance attenuation function coefficients.

If a vertex or geometry shader is active, the derived size depends on the per-vertex point size mode enable. Per-vertex point size mode is enabled or disabled by calling Enable or Disable with the symbolic value PROGRAM_POINT_SIZE_EXT. If per-vertex point size is enabled and a geometry shader is active, the derived point size is taken from the (potentially clipped) point size variable gl_PointSize written by the geometry shader. If per-vertex point size is enabled and no geometry shader is active, the derived point size is taken from the (potentially clipped) point size variable gl_PointSize written by the vertex shader. If per-vertex point size is disabled and a geometry and/or vertex shader is active, the derived point size is taken from the <size> value provided to PointSize, with no distance attenuation applied. In all cases, the derived point size is clamped to the implementation-dependent point size range.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width. ...

Modify section 3.10 "Fog", p. 191

Modify the third paragraph of this section as follows.

If a vertex or geometry shader is active, or if the fog source, as defined below, is FOG_COORD, then c is the interpolated value of the fog coordinate for this fragment. Otherwise, ...

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the command ProgramParameteriEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify section 6.1.14, Shader and Program Objects, p. 256

Add to the second paragraph on p. 257:

... if <shader> is a fragment shader object, and GEOMETRY_SHADER_EXT is returned if <shader> is a geometry shader object.

Add to the end of the description of GetProgramiv, p. 257:

If <pname> is GEOMETRY_VERTICES_OUT_EXT, the current value of the maximum number of vertices the geometry shader will output is returned. If <pname> is GEOMETRY_INPUT_TYPE_EXT, the current geometry shader input type is returned and can be one of POINTS, LINES, LINES_ADJACENCY_EXT, TRIANGLES or TRIANGLES_ADJACENCY_EXT. If <pname> is GEOMETRY_OUTPUT_TYPE_EXT, the current geometry shader output type is returned and can be one of POINTS, LINE_STRIP or TRIANGLE_STRIP.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter gl_PrimitiveIDIn. If NV_primitive_restart is not supported, references to that extension in the discussion of the primitive ID should be removed.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object (or similar functionality) is not supported, the gl_Layer output has no effect. "FramebufferTextureEXT" and "FramebufferTextureLayerEXT" should be removed from "New Procedures and Functions", and FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT, and FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT should be removed from "New Tokens".

Otherwise, this extension modifies EXT_framebuffer_object to add the notion of layered framebuffer attachments and framebuffers that can be used in conjunction with geometry shaders to allow programs to direct

primitives to a face of a cube map or layer of a three-dimensional texture or one- or two-dimensional array texture. The layer used for rendering can be selected by the geometry shader at run time.

(insert before the end of Section 4.4.2, Attaching Images to Framebuffer Objects)

There are several types of framebuffer-attachable images:

* the image of a renderbuffer object, which is always two-dimensional,

* a single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one,

* a single level of a two-dimensional or rectangle texture,

* a single face of a cube map texture level, which is treated as a two-dimensional image, or

* a single layer of a one- or two-dimensional array texture or three-dimensional texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be layered.

(replace section 4.4.2.3, "Attaching Texture Images to a Framebuffer")

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines CopyTexImage{1D|2D}, and CopyTexSubImage{1D|2D|3D}. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

```
void FramebufferTextureEXT(enum target, enum attachment,
                           uint texture, int level);
```

<target> must be FRAMEBUFFER_EXT. <attachment> must be one of the attachment points of the framebuffer listed in table 1.nnn.

If <texture> is zero, any image or array of images attached to the attachment point named by <attachment> is detached, and the state of the attachment point is reset to its initial values. <level> is ignored if <texture> is zero.

If <texture> is non-zero, FramebufferTextureEXT attaches level <level> of the texture object named <texture> to the framebuffer attachment point named by <attachment>. The error INVALID_VALUE is generated if <texture> is not the name of a texture object, or if <level> is not a supported texture level number for textures of the type corresponding to <target>.

The error INVALID_OPERATION is generated if <texture> is the name of a buffer texture.

If <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

The command

```
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
```

operates like FramebufferTextureEXT, except that only a single layer of the texture level, numbered <layer>, is attached to the attachment point. If <texture> is non-zero, the error INVALID_VALUE is generated if <layer> is negative, or if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional or one- or two-dimensional array texture.

The command

```
void FramebufferTextureFaceEXT(enum target, enum attachment,
                               uint texture, int level, enum face);
```

operates like FramebufferTextureEXT, except that only a single face of a cube map texture, given by <face>, is attached to the attachment point. <face> is one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z. If <texture> is non-zero, the error INVALID_VALUE is generated if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a cube map texture.

The command

```
void FramebufferTexture1DEXT(enum target, enum attachment,
                             enum textarget, uint texture, int level);
```

operates identically to FramebufferTextureEXT, except for two additional restrictions. If <texture> is non-zero, the error INVALID_ENUM is generated if <textarget> is not TEXTURE_1D and the error INVALID_OPERATION is generated unless <texture> is the name of a one-dimensional texture.

The command

```
void FramebufferTexture2DEXT(enum target, enum attachment,
                             enum textarget, uint texture, int level);
```

operates similarly to FramebufferTextureEXT. If <textarget> is TEXTURE_2D or TEXTURE_RECTANGLE_ARB, <texture> must be zero or the name of a two-dimensional or rectangle texture. If <textarget> is TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, <texture> must be zero or the name of a cube map texture. For cube map textures, only the single face of the cube map texture level given by <textarget> is

attached. The error INVALID_ENUM is generated if <texture> is not zero and <textarget> is not one of the values enumerated above. The error INVALID_OPERATION is generated if <texture> is the name of a texture whose type does not match the texture type required by <textarget>.

The command

```
void FramebufferTexture3DEXT(enum target, enum attachment,
                             enum textarget, uint texture,
                             int level, int zoffset);
```

behaves identically to FramebufferTextureLayerEXT, with the <layer> parameter set to the value of <zoffset>. The error INVALID_ENUM is generated if <textarget> is not TEXTURE_3D. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional texture.

For all FramebufferTexture commands, if <texture> is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to <attachment> is updated based on the new attachment. FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is set to TEXTURE, FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is set to <texture>, and FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to <level>. FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_FACE is set to <textarget> if FramebufferTexture2DEXT is called and <texture> is the name of a cubemap texture; otherwise, it is set to TEXTURE_CUBE_MAP_POSITIVE_X. FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is set to <layer> or <zoffset> if FramebufferTextureLayerEXT or FramebufferTexture3DEXT is called; otherwise, it is set to zero. FRAMEBUFFER_ATTACHMENT_LAYERED_EXT is set to TRUE if FramebufferTextureEXT is called and <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture; otherwise it is set to FALSE.

(modify Section 4.4.4.1, Framebuffer Attachment Completeness -- add to the conditions necessary for attachment completeness)

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if ...:

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a three-dimensional texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the depth of the texture.

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a one- or two-dimensional array texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the number of layers in the texture.

(modify section 4.4.4.2, Framebuffer Completeness -- add to the list of conditions necessary for completeness)

* If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target (i.e., three-dimensional, cube map, or one- or two-dimensional array textures).
  { FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT }

* If any framebuffer attachment is layered, all attachments must have the same layer count. For three-dimensional textures, the layer count is the depth of the attached volume. For cube map textures, the layer count is always six. For one- and two-dimensional array textures, the layer count is simply the number of layers in the array texture. { FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT }

The enum in { brackets } after each clause of the framebuffer completeness rules specifies the return value of CheckFramebufferStatusEXT (see below) that is generated when that clause is violated. ...

(add section 4.4.7, Layered Framebuffers)

A framebuffer is considered to be layered if it is complete and all of its populated attachments are layered. When rendering to a layered framebuffer, each fragment generated by the GL is assigned a layer number. The layer number for a fragment is zero if

* the fragment is generated by DrawPixels, CopyPixels, or Bitmap,

* geometry shaders are disabled, or

* the current geometry shader does not contain an instruction that statically assigns a value to the built-in output variable gl_Layer.

Otherwise, the layer for each point, line, or triangle emitted by the geometry shader is taken from the layer output of one of the vertices of the primitive. The vertex used is implementation-dependent. To get defined results, all vertices of each primitive emitted should set the same value for gl_Layer. Since the EndPrimitive() built-in function starts a new output primitive, defined results can be achieved if EndPrimitive() is called between two vertices emitted with different layer numbers. A layer number written by a geometry shader has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment's layer number selects an image from the array of images at each attachment point from which to obtain the destination R, G, B, A values for blending (Section 4.1.8) and to which to write the final color values for that attachment. If the fragment's layer number is negative or greater than the number of layers attached, the effects of the fragment on the framebuffer contents are undefined.

When the Clear command is used to clear a layered framebuffer attachment, all layers of the attachment are cleared.

When commands such as ReadPixels or CopyPixels read from a layered framebuffer, the image at layer zero of the selected attachment is always used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there are six layers attached, numbered zero through five. Each layer number is mapped to a cube map face, as indicated in Table X.4.

layer number	cube map face
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

Table X.4, Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

(modify Section 6.1.3, Enumerated Queries -- Modify/add to list of <pname> values for GetFramebufferAttachmentParameterivEXT if FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE)

If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT and the attached image is a layer of a three-dimensional texture or one- or two-dimensional array texture, then <params> will contain the specified layer number. Otherwise, <params> will contain the value zero.

If <pname> is FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, then <params> will contain TRUE if an entire level of a three-dimesional texture, cube map texture, or one- or two-dimensional array texture is attached to the <attachment>. Otherwise, <params> will contain FALSE.

(Modify the Additions to Chapter 5, section 5.4)

Add the commands FramebufferTextureEXT, FramebufferTextureLayerEXT, and FramebufferTextureFaceEXT to the list of commands that are not compiled into a display list, but executed immediately.

Dependencies on EXT_framebuffer_blit

If EXT_framebuffer_blit is supported, the EXT_framebuffer_object language should be further amended so that <target> values passed to FramebufferTextureEXT and FramebufferTextureLayerEXT can be DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT, and that those functions set/query state for the draw framebuffer if <target> is FRAMEBUFFER_EXT.

Dependencies on EXT_texture_array

If EXT_texture_array is not supported, the discussion array textures the layered rendering edits to EXT_framebuffer_object should be removed. Layered rendering to cube map and 3D textures would still be supported.

If EXT_texture_array is supported, the edits to EXT_framebuffer_object supersede those made in EXT_texture_array, except for language pertaining to mipmap generation of array textures.

There are no functional incompatibilities between the FBO support in these two specifications. The only differences are that this extension supports layered rendering and also rewrites certain sections of the core FBO specification more aggressively.

Dependencies on ARB_texture_rectangle

If ARB_texture_rectangle is not supported, all references to rectangle textures in the EXT_framebuffer_object spec language should be removed.

Dependencies on EXT_texture_buffer_object

If EXT_buffer_object is not supported, the reference to an INVALID_OPERATION error if a buffer texture is passed to FramebufferTextureEXT should be removed.

GLX protocol

TBD

Errors

The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is GEOMETRY_INPUT_TYPE_EXT and <value> is not one of POINTS, LINES, LINES_ADJACENCY_EXT, TRIANGLES or TRIANGLES_ADJACENCY_EXT.

The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is GEOMETRY_OUTPUT_TYPE_EXT and <value> is not one of POINTS, LINE_STRIP or TRIANGLE_STRIP.

The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is GEOMETRY_VERTICES_OUT_EXT and <value> is negative.

The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is GEOMETRY_VERTICES_OUT_EXT and <value> exceeds MAX_GEOMETRY_OUTPUT_VERTICES_EXT.

The error INVALID_VALUE is generated by ProgramParameteriEXT if <pname> is set to GEOMETRY_VERTICES_OUT_EXT and the product of <value> and the sum of all components of all active varying variables exceeds MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT.

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when a geometry shader is active and:

* the input primitive type of the current geometry shader is POINTS and <mode> is not POINTS,

* the input primitive type of the current geometry shader is LINES and <mode> is not LINES, LINE_STRIP, or LINE_LOOP,

* the input primitive type of the current geometry shader is TRIANGLES and <mode> is not TRIANGLES, TRIANGLE_STRIP or TRIANGLE_FAN,

* the input primitive type of the current geometry shader is LINES_ADJACENCY_EXT and <mode> is not LINES_ADJACENCY_EXT or LINE_STRIP_ADJACENCY_EXT, or

* the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY_EXT and <mode> is not TRIANGLES_ADJACENCY_EXT or TRIANGLE_STRIP_ADJACENCY_EXT.

New State

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_ATTACHMENT_LAYERED_EXT	nxB	GetFramebuffer-Attachment-ParameterivEXT	FALSE	Framebuffer attachment is layered	4.4.2.3	-

Modify the following state value in Table 6.28, Shader Object State, p. 289.

Get Value	Type	Get Command	Value	Description	Sec.	Attribute
SHADER_TYPE	Z2	GetShaderiv	-	Type of shader (vertex, Fragment, geometry)	2.15.1	-

Add the following state to Table 6.29, Program Object State, p. 290

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
GEOMETRY_VERTICES_OUT_EXT	Z+	GetProgramiv	0	max # of output vertices	2.16.4	-
GEOMETRY_INPUT_TYPE_EXT	Z5	GetProgramiv	TRIANGLES	Primitive input type	2.16.1	-
GEOMETRY_OUTPUT_TYPE_EXT	Z3	GetProgramiv	TRIANGLE_STRIP	Primitive output type	2.16.2	-

New Implementation Dependent State

Get Value	Type	Get Command	Min. Value	Description	Sec.	Attrib
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT	Z+	GetIntegerv	16	maximum number of texture image units accessible in a geometry shader	2.16.4	-
MAX_GEOMETRY_OUTPUT_VERTICES_EXT	Z+	GetIntegerv	256	maximum number of vertices that any geometry shader can can emit	2.16.4	-
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT	Z+	GetIntegerv	1024	maximum number of total components (all vertices) of active varyings that a geometry shader can emit	2.16.4	-
MAX_GEOMETRY_UNIFORM_COMPONENTS_EXT	Z+	GetIntegerv	512	Number of words for geometry shader uniform variables	2.16.3	-
MAX_GEOMETRY_VARYING_COMPONENTS_EXT	Z+	GetIntegerv	32	Number of components for varying variables between geometry and fragment shaders	2.16.4	-
MAX_VERTEX_VARYING_COMPONENTS_EXT	Z+	GetIntegerv	32	Number of components for varying variables between Vertex and geometry shaders	2.15.3	-
MAX_VARYING_COMPONENTS_EXT	Z+	GetIntegerv	32	Alias for MAX_VARYING_FLOATS	2.15.3	-

Modifications to the OpenGL Shading Language Specification version 1.10.59

Including the following line in a shader can be used to control the language features described in this extension:

```
#extension GL_EXT_geometry_shader4 : <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
#define GL_EXT_geometry_shader4 1
```

Change the introduction to Chapter 2 "Overview of OpenGL Shading" as follows:

The OpenGL Shading Language is actually three closely related languages. These languages are used to create shaders for the programmable processors contained in the OpenGL processing pipeline. The precise definition of these programmable units is left to separate specifications. In this document, we define them only well enough to provide a context for defining these languages. Unless otherwise noted in this paper, a language feature applies to all languages, and common usage

will refer to these languages as a single language. The specific languages will be referred to by the name of the processor they target: vertex, geometry or fragment.

Change the last sentence of the first paragraph of section 3.2 "Source Strings" to:

Multiple shaders of the same language (vertex, geometry or fragment) can be linked together to form a single program.

Change the first paragraph of section 4.1.3, "Integers" as follows:

... integers are limited to 16 bits of precision, plus a sign representation in the vertex, geometry and fragment languages..

Change the first paragraph of section 4.1.9, "Arrays", as follows:

Variables of the same type can be aggregated into one- and two-dimensional arrays by declaring a name followed by brackets ([ ] for one-dimensional arrays and [][] for two-dimensional arrays) enclosing an optional size. When an array size is specified in a declaration, it must be an integral constant expression (see Section 4.3.3 "Integral Constant Expressions") greater than zero. If an array is indexed with an expression that is not an integral constant expression or passed as an argument to a function, then its size must be declared before any such use. It is legal to declare an array without a size and then later re-declare the same name as an array of the same type and specify a size. It is illegal to declare an array with a size, and then later (in the same shader) index the same array with an integral constant expression greater than or equal to the declared size. It is also illegal to index an array with a negative constant expression. Arrays declared as formal parameters in a function declaration must specify a size. Undefined behavior results from indexing an array with a non-constant expression that's greater than or equal to the array's size or less than 0. All basic types and structures can be formed into arrays.

Two-dimensional arrays can only be declared as "varying in" variables in a geometry shader. See section 4.3.6 for details. All other declarations of two-dimensional arrays are illegal.

Change the fourth paragraph of section 4.2 "Scoping", as follows:

Shared globals are global variables declared with the same name in independently compiled units (shaders) of the same language (vertex, geometry or fragment) that are linked together .

Change section 4.3 "Type Qualifiers"

Change the "varying", "in" and "out" qualifiers as follows:

varying - linkage between a vertex shader and geometry shader, or between a geometry shader and a fragment shader, or between a vertex shader and a fragment shader.

in - for function parameters passed into a function or for input varying variables (geometry only)

out - for function parameters passed back out of a function, but not initialized for use when passed in. Also for output varying variables (geometry only).

Change section 4.3.6 "Varying" as follows:

Varying variables provide the interface between the vertex shader and geometry shader and also between the geometry shader and fragment shader and the fixed functionality between them. If no geometry shader is present, varying variables also provide the interface between the vertex shader and fragment shader.

The vertex, or geometry shader will compute values per vertex (such as color, texture coordinates, etc) and write them to output variables declared with the "varying" qualifier (vertex or geometry) or "varying out" qualifiers (geometry only). A vertex or geometry shader may also read these output varying variables, getting back the same values it has written. Reading an output varying variable in a vertex or geometry shader returns undefined results if it is read before being written.

A geometry shader may also read from an input varying variable declared with the "varying in" qualifiers. The value read will be the same value as written by the vertex shader for that varying variable. Since a geometry shader operates on primitives, each input varying variable needs to be declared as an array. Each element of such an array corresponds to a vertex of the primitive being processed. If the varying variable is declared as a scalar or matrix in the vertex shader, it will be a one-dimensional array in the geometry shader. Each array can optionally have a size declared. If a size is not specified, it inferred by the linker and depends on the value of the input primitive type. See table 4.3.xxx to determine the exact size. The read-only built-in constant gl_VerticesIn will be set to this value by the linker. If a size is specified, it has to be the size as given by table 4.3.xxx, otherwise a link error will occur. The built-in constant gl_VerticesIn, if so desired, can be used to size the array correctly for each input primitive type. Varying variables can also be declared as arrays in the vertex shader. This means that those, on input to the geometry shader, must be declared as two- dimensional arrays. The first index to the two-dimensional array holds the vertex number. Declaring a size for the first range of the array is optional, just as it is for one-dimensional arrays. The second index holds the per-vertex array data. Declaring a size for the second range of the array is not optional, and has to match the declaration in the vertex shader.

Input primitive type	Value of built-in gl_VerticesIn
POINTS	1
LINES	2
LINES_ADJACENCY_EXT	4
TRIANGLES	3
TRIANGLES_ADJACENCY_EXT	6

Table 4.3.xxxx The value of the built-in variable gl_VerticesIn is determined at link time, based on the input primitive type.

It is illegal to index these varying arrays, or in the case of two-dimensional arrays, the first range of the array, with a negative integral constant expression or an integral constant expression greater than or equal to gl_VerticesIn. A link error will occur in these cases.

Varying variables that are part of the interface to the fragment shader are set per vertex and interpolated in a perspective correct manner, unless flat shaded, over the primitive being rendered. If single-sampling, the interpolated value is for the fragment center. If multi-sampling, the interpolated value can be anywhere within the pixel, including the fragment center or one of the fragment samples.

A fragment shader may read from varying variables and the value read will be the interpolated value, as a function of the fragment's position within the primitive, unless the varying variable is flat shaded. A fragment shader cannot write to a varying variable.

If a geometry shader is present, the type of the varying variables with the same name declared in the vertex shader and the input varying variables in the geometry shader must match, otherwise the link command will fail. Likewise, the type of the output varying variables with the same name declared in the geometry shader and the varying variables in the fragment shader must match.

If a geometry shader is not present, the type of the varying variables with the same name declared in both the vertex and fragment shaders must match, otherwise the link command will fail.

Only those varying variables used (i.e. read) in the geometry or fragment shader must be written to by the vertex or geometry shader; declaring superfluous varying variables in the vertex shader or declaring superfluous output varying variables in the geometry shader is permissible.

Varying variables are declared as in the following example:

```
  varying in float foo[];    // geometry shader input. Size of the
                             // array set as a result of link, based
                             // on the input primitive type.

  varying in float foo[gl_VerticesIn]; // geometry shader input

  varying in float foo[3];   // geometry shader input. Only legal for
                             // the TRIANGLES input primitive type

  varying in float foo[][5]; // Size of the first range set as a
                             // result of link. Each vertex holds an
                             // array of 5 floats.

  varying out vec4 bar;      // geometry output
  varying vec3 normal;       // vertex shader output or fragment
                             // shader input
```

The varying qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3 and mat4 or arrays of these. Structures cannot be varying. Additionally, the "varying in" and "varying out" qualifiers can only be used in a geometry shader.

If no vertex shader is active, the fixed functionality pipeline of OpenGL will compute values for the built-in varying variables that will be consumed by the fragment shader. Similarly, if no fragment shader is active, the vertex shader or geometry shader is responsible for computing and writing to the built-in varying variables that are needed for OpenGL's fixed functionality fragment pipeline.

Varying variables are required to have global scope, and must be declared outside of function bodies, before their first use.

Change section 7.1 "Vertex Shader Special Variables"

Rename this section to "Vertex and Geometry Shader Special Variables"

Anywhere in this section where it reads "vertex language" replace it with "vertex and geometry language".

Anywhere in this section where it reads "vertex shader" replace it with "vertex shader or geometry shader".

Change the second paragraph to:

The variable gl_Position is available only in the vertex and geometry language and is intended for writing the homogeneous vertex position. It can be written at any time during shader execution. It may also be read back by the shader after being written. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations that operate on primitives after vertex or geometry processing has occurred. Compilers may generate a diagnostic message if they detect gl_Position is read before being written, but not all such cases are detectable. Writing to gl_Position is optional. If gl_Position is not written but subsequent stages of the OpenGL pipeline consume gl_Position, then results are undefined.

Change the last sentence of this section into the following:

The read-only built-in gl_PrimitiveIDIn is available only in the geometry language and is filled with the number of primitives processed by the geometry shader since the last time Begin was called (directly or indirectly via vertex array functions). See section 2.16.4 for more information.

This variable is intrinsically declared as:

```
int gl_PrimitiveIDIn; // read only
```

The built-in output variable gl_PrimitiveID is available only in the geometry language and provides a single integer that serves as a primitive identifier. This written primitive ID is available to fragment shaders. If a fragment shader using primitive IDs is active and a geometry shader is also active, the geometry shader must write to gl_PrimitiveID or the primitive ID in the fragment shader number is undefined.

The built-in output variable gl_Layer is available only in the geometry language, and provides the number of the layer of textures attached to a FBO to direct rendering to. If a shader statically assigns a value to gl_Layer, layered rendering mode is enabled. See section 2.16.4 for a detailed explanation. If a shader statically assigns a value to gl_Layer, and there is an execution path through the shader that does not set gl_Layer, then the value of gl_Layer may be undefined for executions of the shader that take that path.

These variables area intrinsically declared as:

```
int gl_PrimitiveID;
int gl_Layer;
```

These variables can be read back by the shader after writing to them, to retrieve what was written. Reading the variable before writing it results in undefined behavior. If it is written more than once, the last value written is consumed by the subsequent operations.

All built-in variables discussed in this section have global scope.

Change section 7.2 "Fragment Shader Special Variables"

Change the first paragraph on p. 44 as follows:

The fragment shader has access to the read-only built-in variable gl_FrontFacing whose value is true if the fragment belongs to a front-facing primitive. One use of this is to emulate two-sided lighting by selecting one of two colors calculated by the vertex shader or geometry shader.

Change the first sentence of section 7.4 "Built-in Constants"

The following built-in constant is provided to geometry shaders.

```
const int gl_VerticesIn; // Value set at link time
```

The following built-in constants are provided to the vertex, geometry and fragment shaders:

Change section 7.6 "Varing Variables"

Unlike user-defined varying variables, the built-in varying variables don't have a strict one-to-one correspondence between the vertex language, geometry language and the fragment language. Four sets are provided, one set for the vertex language output, one set for the geometry language output, one set for the fragment language input and another set for the geometry language input. Their relationship is described below.

The following built-in varying variables are available to write to in a vertex shader or geometry shader. A particular one should be written to if any functionality in a corresponding geometry shader or fragment shader or fixed pipeline uses it or state derived from it. Otherwise, behavior is undefined.

Vertex language built-in outputs:

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[]; // at most will be gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

Geometry language built-in outputs:

```
varying out vec4 gl_FrontColor;
varying out vec4 gl_BackColor;
varying out vec4 gl_FrontSecondaryColor;
varying out vec4 gl_BackSecondaryColor;
varying out vec4 gl_TexCoord[]; // at most gl_MaxTextureCoords
varying out float gl_FogFragCoord;
```

For gl_FogFragCoord, the value written will be used as the "c" value on page 160 of the OpenGL 1.4 Specification by the fixed functionality pipeline. For example, if the z-coordinate of the fragment in eye space is desired as "c", then that's what the vertex or geometry shader should write into gl_FogFragCoord.

Indices used to subscript gl_TexCoord must either be an integral constant expressions, or this array must be re-declared by the shader with a size. The size can be at most gl_MaxTextureCoords. Using indexes close to 0 may aid the implementation in preserving varying resources.

The following input varying variables are available to read from in a geometry shader.

```
  varying in vec4 gl_FrontColorIn[gl_VerticesIn];
  varying in vec4 gl_BackColorIn[gl_VerticesIn];
  varying in vec4 gl_FrontSecondaryColorIn[gl_VerticesIn];
  varying in vec4 gl_BackSecondaryColorIn[gl_VerticesIn];
  varying in vec4 gl_TexCoordIn[gl_VerticesIn][]; // at most will be
                                                  // gl_MaxTextureCoords
  varying in float gl_FogFragCoordIn[gl_VerticesIn];
  varying in vec4 gl_PositionIn[gl_VerticesIn];
  varying in float gl_PointSizeIn[gl_VerticesIn];
  varying in vec4 gl_ClipVertexIn[gl_VerticesIn];
```

All built-in variables are one-dimensional arrays, except for gl_TexCoordIn, which is a two-dimensional array. Each element of a one-dimensional array, or the first index of a two-dimensional array, corresponds to a vertex of the primitive being processed and receives their value from the equivalent vertex output varying variables. See also section 4.3.6.

The following varying variables are available to read from in a fragment shader. The gl_Color and gl_SecondaryColor names are the same names as attributes passed to the vertex shader. However, there is no name conflict, because attributes are visible only in vertex shaders and the following are only visible in a fragment shader.

```
  varying vec4 gl_Color;
  varying vec4 gl_SecondaryColor;
  varying vec4 gl_TexCoord[]; // at most will be gl_MaxTextureCoords
  varying float gl_FogFragCoord;
```

The values in gl_Color and gl_SecondaryColor will be derived automatically by the system from gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor. This selection process is described in section 2.14.1 of the OpenGL 2.0 Specification. If fixed functionality is used for vertex processing, then gl_FogFragCoord will either be the z-coordinate of the fragment in eye space, or the interpolation of the fog coordinate, as described in section 3.10 of the OpenGL 1.4 Specification. The gl_TexCoord[] values are the interpolated gl_TexCoord[] values from a vertex or geometry shader or the texture coordinates of any fixed pipeline based vertex functionality.

Indices to the fragment shader gl_TexCoord array are as described above in the vertex and geometry shader text.

Change section 8.7 "Texture Lookup Functions"

Change the first paragraph to:

Texture lookup functions are available to vertex, geometry and fragment shaders. However, level of detail is not computed by fixed functionality for vertex or geometry shaders, so there are some differences in operation between texture lookups. The functions.

Change the third and fourth paragraphs to:

In all functions below, the bias parameter is optional for fragment shaders. The bias parameter is not accepted in a vertex or geometry shader. For a fragment shader, if bias is present, it is added to the calculated level of detail prior to performing the texture access operation. If the bias parameter is not provided, then the implementation automatically selects level of detail: For a texture that is not mip-mapped, the texture is used directly. If it is mip- mapped and running in a fragment shader, the LOD computed by the implementation is used to do the texture lookup. If it is mip- mapped and running on the vertex or geometry shader, then the base LOD of the texture is used.

The built-ins suffixed with "Lod" are allowed only in a vertex or geometry shader. For the "Lod" functions, lod is directly used as the level of detail.

Change section 8.9 Noise Functions

Change the first paragraph to:

Noise functions are available to the vertex, geometry and fragment shaders. They are...

Add a section 8.10 Geometry Shader Functions

This section contains functions that are geometry language specific.

Syntax:

```
void EmitVertex();   // Geometry only
void EndPrimitive(); // Geometry only
```

Description:

The function EmitVertex() specifies that a vertex is completed. A vertex is added to the current output primitive using the current values of the varying output variables and the current values of the special built-in output variables gl_PointSize, gl_ClipVertex, gl_Layer, gl_Position and gl_PrimitiveID. The values of any unwritten output variables are undefined. The values of all varying output variables and the special built-in output variables are undefined after a call to EmitVertex(). If a geometry shader, in one invocation, emits more vertices than the value GEOMETRY_VERTICES_OUT_EXT, these emits may have no effect.

The function EndPrimitive() specifies that the current output primitive is completed and a new output primitive (of the same type) should be started. This function does not emit a vertex. The effect of EndPrimitive() is roughly equivalent to calling End followed by a new Begin, where the primitive mode is taken from the program object parameter GEOMETRY_OUTPUT_TYPE_EXT. If the output primitive type is POINTS, calling EndPrimitive() is optional.

A geometry shader starts with an output primitive containing no vertices. When a geometry shader terminates, the current output primitive is automatically completed. It is not necessary to call EndPrimitive() if the geometry shader writes only a single primitive.

Add/Change section 9 (Shading language grammar):

```
init_declarator_list:
  single_declaration
  init_declarator_list COMMA IDENTIFIER
  init_declarator_list COMMA IDENTIFIER array_declarator_suffix
  init_declarator_list COMMA IDENTIFIER EQUAL initializer

single_declaration:
  fully_specified_type
  fully_specified_type IDENTIFIER
  fully_specified_type IDENTIFIER array_declarator_suffix
  fully_specified_type IDENTIFIER EQUAL initializer

array_declarator_suffix:
  LEFT_BRACKET RIGHT_BRACKET
  LEFT_BRACKET constant_expression RIGHT_BRACKET
  LEFT_BRACKET RIGHT_BRACKET array_declarator_suffix
  LEFT_BRACKET constant_expression RIGHT_BRACKET
array_declarator_suffix

type_qualifier:
  CONST
  ATTRIBUTE     // Vertex only
  VARYING
  VARYING IN    // Geometry only
  VARYING OUT   // Geometry only
  UNIFORM
```

NVIDIA Implementation Details

Because of a hardware limitation, some GeForce 8 series chips use the odd vertex of an incomplete TRIANGLE_STRIP_ADJACENCY_EXT primitive as a replacement adjacency vertex rather than ignoring it.

Issues

1. *How do geometry shaders fit into the existing GL pipeline?*

   RESOLVED: The following diagram illustrates how geometry shaders fit into the "vertex processing" portion of the GL (Chapter 2 of the OpenGL 2.0 Specification).

   First, vertex attributes are specified via immediate-mode commands or through vertex arrays. They can be conventional attributes (e.g., glVertex, glColor, glTexCoord) or generic (numbered) attributes.

   Vertices are then transformed, either using a vertex shader or fixed-function vertex processing. Fixed-function vertex processing includes position transformation (modelview and projection matrices), lighting, texture coordinate generation, and other calculations. The results of either method are a "transformed vertex", which has a position (in clip coordinates), front and back colors, texture coordinates, generic attributes (vertex shader only), and so on. Note that on many current GL implementations, vertex processing is performed by executing a "fixed function vertex shader" generated by the driver.

After vertex transformation, vertices are assembled into primitives, according to the topology (e.g., TRIANGLES, QUAD_STRIP) provided by the call to glBegin(). Primitives are points, lines, triangles, quads, or polygons. Many GL implementations do not directly support quads or polygons, but instead decompose them into triangles as permitted by the spec.

After initial primitive assembly, a geometry shader is executed on each individual point, line, or triangle primitive, if one is active. It can read the attributes of each transformed vertex, perform arbitrary computations, and emit new transformed vertices. These emitted vertices are themselves assembled into primitives according to the output primitive type of the geometry shader.

Then, the colors of the vertices of each primitive are clamped to [0,1] (if color clamping is enabled), and flat shading may be performed by taking the color from the provoking vertex of the primitive.

Each primitive is clipped to the view volume, and to any enabled user-defined clip planes. Color, texture coordinate, and other attribute values are computed for each new vertex introduced by clipping.

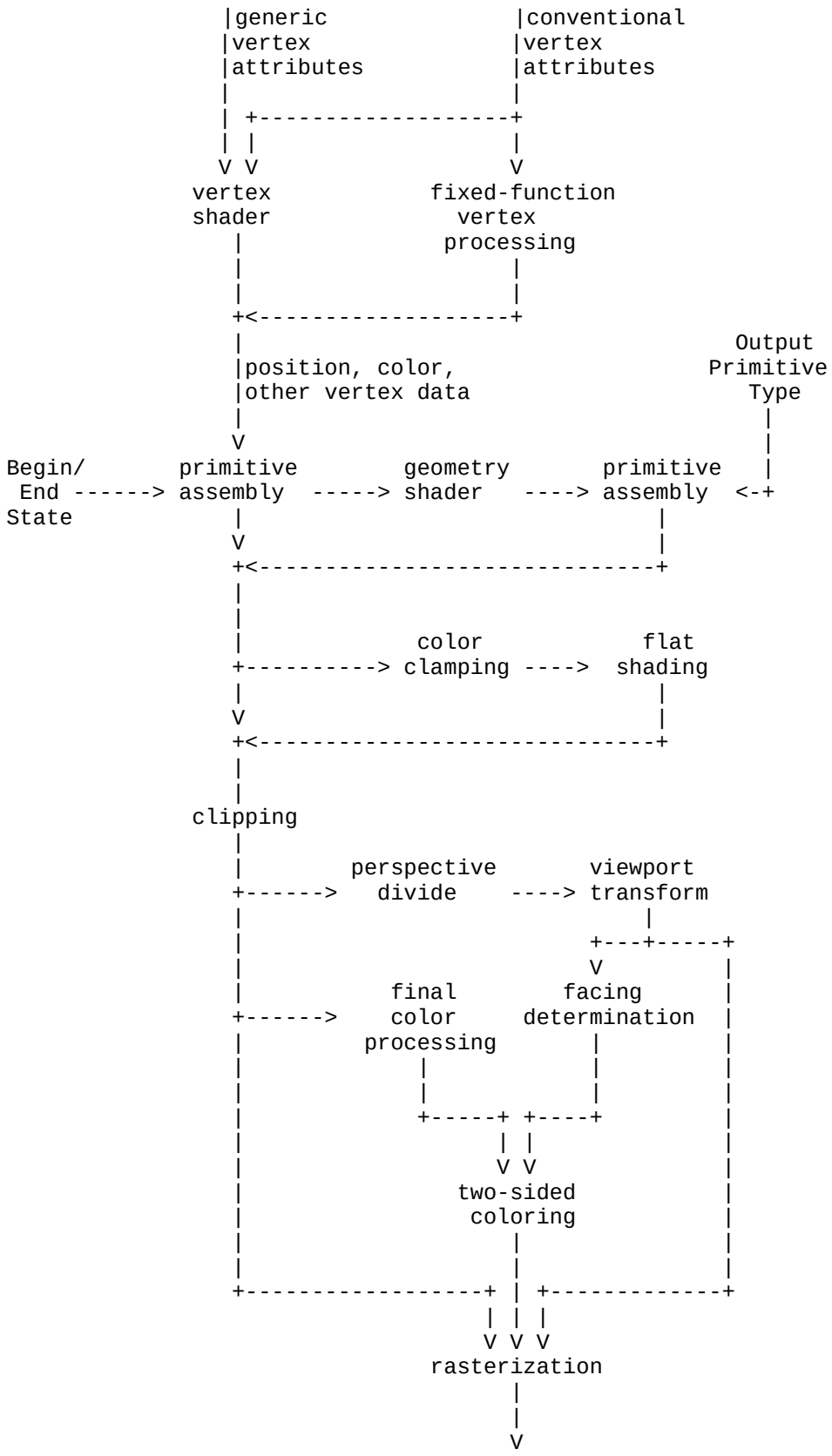
After clipping, the position of each vertex (in clip coordinates) is converted to normalized device coordinates in the perspective division (divide by w) step, and to window coordinates in the viewport transformation step.

At the same time, color values may be converted to normalized fixed-point values according to the "Final Color Processing" portion of the specification.

After the vertices of the primitive are transformed to window coordinate, the GL determines if the primitive is front- or back-facing. That information is used for two-sided color selection, where a single set of colors is selected from either the front or back colors associated with each transformed vertex.

When all this is done, the final transformed position, colors (primary and secondary), and other attributes are used for rasterization (Chapter 3 in the OpenGL 2.0 Specification).

When the raster position is specified (via glRasterPos), it goes through the entire vertex processing pipeline as though it were a point. However, geometry shaders are never run on the raster position.


generic vertex attributes
conventional vertex attributes
vertex shader
fixed-function vertex processing
position, color, other vertex data
Output Primitive Type
Begin/ End
State
primitive assembly
geometry shader
primitive assembly
color clamping
flat shading
clipping
perspective divide
viewport transform
final color processing
facing determination
two-sided coloring
rasterization

2. *Why is this called GL_EXT_geometry_shader4? There aren't any previous versions of this extension, let alone three?*

   RESOLVED: To match its sibling, EXT_gpu_shader4 and the assembly version NV_gpu_program4. This is the fourth generation of shading functionality, hence the "4" in the name.

3. *Should the GL produce errors at Begin time if an application specifies a primitive mode that is "incompatible" with the geometry shader? For example, if the geometry shader operates on triangles and the application sends a POINTS primitive?*

   RESOLVED: Yes. Mismatches of app-specified primitive types and geometry shader input primitive types appear to be errors and would produce weird and wonderful effects.

4. *Can the input primitive type of a geometry shader be determined at run time?*

   RESOLVED: No. Each geometry shader has a single input primitive type, and vertices are presented to the shader in a specific order based on that type.

5. *Can the input primitive type of a geometry shader be changed?*

   DISCUSSION: The input primitive type is a property of the program object. A change of the input primitive type means the program object will need to be re-linked. It would be nice if the input primitive type was known at compile time, so that the compiler can do error checking of the type and the number of vertices being accessed by the shader. Since we allow multiple compilation units to form one geometry shader, it is not clear how to achieve that. Therefore, the input primitive type is a property of the program object, and not of a shader object.

   RESOLVED: Yes, but each change means the program object will have to be re-linked.

6. *Can the output primitive type of a geometry shader be determined at run time?*

   RESOLVED: Not in this extension.

7. *Can the output primitive type of a program object be changed?*

   RESOLVED: Yes, but the program object will have to be re-linked in order for the change to have effect on program execution.

8. *Must the output primitive type of a geometry shader match the input primitive type in any way?*

   RESOLVED: No, you can have a geometry shader generate points out of triangles or triangles out of points. Some combinations are analogous to existing OpenGL operations: reading triangles and writing points or line strips can be used to emulate a subset of PolygonMode functionality. Reading points and writing triangle strips can be used to emulate point sprites.

9. *Are primitives emitted by a geometry shader processed like any other OpenGL primitive?*

   RESOLVED: Yes. Antialiasing, stippling, polygon offset, polygon mode, culling, two-sided lighting and color selection, point sprite operations, and fragment processing all work as expected.

   One limitation is that the only output primitive types supported are points, line strips, and triangle strips, none of which meaningfully support edge flags that are sometimes used in conjunction with the POINT and LINE polygon modes. Edge flags are always ignored for line-mode triangle strips.

10. *Should geometry shaders support additional input primitive types?*

    RESOLVED: Possibly in a future extension. It should be straightforward to build a future extension to support geometry shaders that operate on quads. Other primitive types might be more demanding on hardware. Quads with adjacency would require 12 vertices per shader execution. General polygons may require even more, since there is no fixed bound on the number of vertices in a polygon.

11. *Should geometry shaders support additional output primitive types?*

    RESOLVED: Possibly in a future extension. Additional output types (e.g., independent lines, line loops, triangle fans, polygons) may be useful in the future; triangle fans/polygons seem particularly useful.

12. *How are adjacency primitives processed by the GL?*

    RESOLVED: The primitive type of an adjacent primitive is set as a Begin mode parameter. Any vertex of an adjacency primitive will be treated as a regular vertex, and processed by a vertex shader as well as the geometry shader. The geometry shader cannot output adjacency primitives, thus processing stops with the geometry shader. If a geometry shader is not active, the GL ignores the "adjacent" vertices in the adjacency primitive.

13. *Should we provide additional adjacency primitive types that can be used inside a Begin/End?*

    RESOLVED: Not in this extension. It may be desirable to add new primitive types (e.g., TRIANGLE_FAN_ADJACENCY) in a future extension.

14. *How do geometry shaders interact with RasterPos?*

    RESOLVED: Geometry shaders are ignored when specifying the raster position.

15. *How do geometry shaders interact with pixel primitives (DrawPixels, Bitmap)?*

    RESOLVED: They do not.

16. *Is there a limit on the number of vertices that can be emitted by a geometry shader?*

    RESOLVED: Unfortunately, yes. Besides practical hardware limits, there may also be practical performance advantages when applications guarantee a tight upper bound on the number of vertices a geometry shader will emit. GPUs frequently excecute programs in parallel, and there are substantial implementation challenges to parallel execution of geometry threads that can write an unbounded number of results, particular given that all the primitives generated by the first geometry shader invocation must be consumed before any of the primitives generated by the second program invocation. Limiting the amount of data a geometry shader can write substantially eases the implementation burden.

    A program object, holding a geometry shader, must declare a maximum number of vertices that can be emitted. There is an implementation-dependent limit on the total number of vertices a program object can emit (256 minimum) and the product of the number of vertices emitted and the number of components of all active varying variables (1024 minimum).

    It would be ideal if the limit could be inferred from the instructions in the shader itself, and that would be possible for many shaders, particularly ones with straight-line flow control. For shaders with more complicated flow control (subroutines, data- dependent looping, and so on), it would be impossible to make such an inference and a "safe" limit would have to be used with adverse and possibly unexpected performance consequences.

    The limit on the number of EmitVertex() calls that can be issued can not always be enforced at compile time, or even at Begin time. We specify that if a shader tries to emit more vertices than allowed, emits that exceed the limit may or may not have any effect.

17. *Should it be possible to change the limit GEOMETRY_VERTICES_OUT_EXT, the number of vertices emitted by a geometry shader, after the program object, containing the shader, is linked?*

    RESOLVED: NO. See also issue 31. Changing this limit might require a re-compile and/or re-link of the shaders and program object on certain implementations. Pretending that this limit can be changed without re-linking does not reflect reality.

18. *How do user clipping and geometry shaders interact?*

    RESOLVED: Just like vertex shaders and user clipping interact. The geometry shader needs to provide the (eye) position gl_ClipVertex. Primitives are clipped after geometry shader execution, not before.

19. *How do edge flags interact with adjacency primitives?*

    RESOLVED: If geometry programs are disabled, adjacency primitives are still supported. For TRIANGLES_ADJACENCY_EXT, edge flags will apply as they do for TRIANGLES. Such primitives are rendered as independent triangles as though the adjacency vertices were not provided. Edge flags for the "real" vertices are supported. For all other adjacency primitive types, edge flags are irrelevant.

20. Now that a third shader object type is added, what combinations of GLSL, assembly (ARB or NV) low level and fixed-function do we want to support?

DISCUSSION: With the addition of the geometry shader, the number of combinations the GL pipeline could support doubled (there is no fixed-function geometry shading). Possible combinations now are:

vertex	geometry	fragment
ff/ASM/GLSL	none/ASM/GLSL	ff/ASM/GLSL

for a total of 3 x 3 x 3 is 27 combinations. Before the geometry shader was added, the number of combinations was 9, and those we need to support. We have a choice on the other 18.

RESOLUTION: It makes sense to draw a line at raster in the GL pipeline. The 'north' side of this line covers vertex and geometry shaders, the 'south' side fragment shaders. We now add a simple rule that states that if a program object contains anything north of this line, the north side will be 100% GLSL. This means that:

a) GLSL program objects with a vertex shader can only use a geometry shader and not an assembly geometry program. If an assembly geometry program is enabled, it is bypassed. This also avoids a tricky case -- a GLSL program object with a vertex and a fragment program linked together. Injecting an assembly geometry shader in the middle at run time won't work well.

b) GLSL program objects with a geometry shader must have a vertex shader (cannot be ARB/NV or fixed-function vertex shading).

The 'south' side in this program object still can be any of ff/ARB/NV/GLSL.

21. How do geometry shaders interact with color clamping?

RESOLVED: Geometry shader execution occurs prior to color clamping in the pipeline. This means the colors written by vertex shaders are not clamped to [0,1] before they are read by geometry shaders. If color clamping is enabled, any vertex colors written by the geometry shader will have their components clamped to [0,1].

22. What is a primitive ID and a vertex ID? I am confused.

DISCUSSION: A vertex shader can read a built-in attribute that holds the ID of the current vertex it is processing. See the EXT_gpu_shader4 spec for more information on vertex ID. If the geometry shader needs access to a vertex ID as well, it can be passed as a user-defined varying variable. A geometry shader can read a built-in varying variable that holds the ID of the current primitive it is processing. It also has the ability to write to a built-in output primitive ID variable, to communicate the primitive ID to a fragment shader. A fragment shader can read a built-in attribute that holds the ID of the current primitive it is processing. A primitive ID will be generated even if no geometry shader is active.

23. After a call to EmitVertex(), should the values of the output varying variables be retained or be undefined?

DISCUSSION: There is not a clear answer to this question .The underlying HW mechanism is as follows. An array of output registers is set aside to store vertices that make up primitives. After each EmitVertex() a pointer into that array is incremented. The shader no longer has access to the previous set of values. This argues that the values of output varying variables should be undefined after an EmitVertex() call. The shader is responsible for writing values to all varying variables it wants to emit, for each emit. The counter argument to this is that this is not a nice model for GLSL to program in. The compiler can store varying outputs in a temp register and preserve their values across EmitVertex() calls, at the cost of increased register pressure.

RESOLUTION: For now, without being a clear winner, we've decided to go with the undefined option. The shader is responsible for writng values to all varying variabvles it wants to emit, for each emit.

24. How to distinguish between input and output "varying" variables?

DISCUSSION: Geometry shader outputs are varying variables consistent with the existing definition of varying (used to communicate to the fragment processing stage). Geometry inputs are received from a vertex shader writing to its varying variable outputs. The inputs could be called "varying", to match with the vertex shader, or could be called "attributes" to match the vertex shader inputs (which are called attributes).

RESOLUTION: We'll call input variables "varying", and not "attributes". To distinguish between input and output, they will be further qualified with the words "in" and "out" resulting in, for example:

```
varying in float foo;
varying out vec4 bar[];
```

25. What is the syntax for declaring varying input variables?

DISCUSSION: We need a way to distinguish between the vertices of the input primitive. Suggestions:

1. Declare each input varying variable as an unsized array. Its size is inferred by the linker based on the output primitive type.

2. Declare each input varying variable as a sized array. If the size does not match the output primitive type, a link error occurs.

3. Have an array of structures, where the structure contains the attributes for each vertex.

RESOLUTION: Option 1 seems simple and solves the problem, but it is not a clear winner over the other two. To aid the shader writer in figuring out the size of each array, a new built-in constant, gl_VerticesIn, is defined that holds the number of vertices for the current input primitive type.

26. Does gl_PointSize, gl_Layer, gl_ClipVertex count agains the MAX_GEOMETRY_VARYING_COMPONENTS limit?

RESOLUTION: Core OpenGL 2.0 makes a distinction between varying variables, output from a vertex shader and interpolated over a primitive, and 'special built-in variables' that are outputs, but not interpolated across a primitive. Only varying variables do count against the MAX_VERTEX_VARYING_COMPONENTS limit. gl_PointSize, gl_Layer, gl_ClipVertex and gl_Position are 'special built-in' variables, and therefore should not count against the limit. If HW does need to take components away to support those, that is ok. The actual spec language does mention possible implementation dependencies.

27. Should writing to gl_Position be optional?

DISCUSSION: Before this extensions, the OpenGL Shading Language required that gl_Position be written to in a vertex shader. With the addition of geometry shaders, it is not necessary anymore for a vertex shader to output gl_Position. The geometry shader can do so. With the addition of transform-feedback (see the transform feedback specification) it is not necessary useful for the geometry shader to write out gl_Position either.

RESOLUTION: Yes, this should be optional.

28. Should geometry shaders be able to select a layer of a 3D texture, cube map texture, or array texture at run time? If so, how?

RESOLVED: See also issue 32. This extension provides a per-vertex output called "gl_Layer", which is an integer specifying the layer to render to. In order to get defined results, the value of gl_Layer needs to be constant for each primitive (point, line or triangle) being emitted by a geometry shader. This layer value is used for all fragments generated by that primitive.

The EXT_framebuffer_object (FBO) extension is used for rendering to textures, but for cube maps and 3D textures, it only provides the ability to attach a single face or layer of such textures.

This extension generalizes FBO by creates new entry points to bind an entire texture level (FramebufferTextureEXT) or a single layer of a texture level (FramebufferTextureLayerEXT) or a single face of a level of a cube map texture (FramebufferTextureFaceEXT) to an attachment point. The existing FBO binding functions, FramebufferTexture[123]DEXT are retained, and are defined in terms of the more general new functions.

The new functions do not have a dimension in the function name or a <textarget> parameter, which can be inferred from the provided texture.

When an entire texel level of a cube map, 3D, or array texture is attached, that attachment is considered layered. The framebuffer is considered layered if any attachment is layered. When the framebuffer is layered, there are three additional completeness requirements:

```
  * all attachments must be layered
  * all color attachments must be from textures of identical type
  * all attachments must have the same number of layers
```

We expect subsequent versions of the FBO spec to relax the requirement that all attachments must have the same width and height, and plan to relax the similar requirement for layer count at that time.

When rendering to a layered framebuffer, layer zero is used unless a geometry shader that writes (statically assings, to be precise) to gl_Layer. When rendering to a non-layered framebuffer, the value of gl_Layer is ignored and the set of single-image attachments are used. When reading from a layered framebuffer (e.g., ReadPixels), layer zero is always used. When clearing a layered framebuffer, all layers are cleared to the corresponding clear values.

Several other approaches were considered, including leveraging existing FBO attachment functions and requiring the use of FramebufferTexture3D with a <zoffset> of zero to make a framebuffer attachment "layerable" (attaching layer zero means that the attachment could be used for either layered- or non- layered rendering). Whether rendering was layered or not could either be inferred from the active geometry shader, or set as a new property of the framebuffer object. There is presently no FramebufferParameter API to set a property of a framebuffer, so it would have been necessary to create new set/query APIs if this approach were chosen.

29. *How should per-vertex point size work with geometry shaders?*

RESOLVED: The value of the existing VERTEX_PROGRAM_POINT_SIZE enable, to control the point size behavior of a vertex shader, does not affect geometry shaders. Specifically, If a geometry shader is active, the point size is taken from the point size output gl_PointSize of the vertex shader, regardless of the value of VERTEX_PROGRAM_POINT_SIZE.

30. *Geometry shaders don't provide a QUADS or generic POLYGON input primitive type. In this extension, what happens if an application provides QUADS, QUAD_STRIP, or POLYGON primitives?*

RESOLVED: Not all vendors supporting this extension were able to accept quads and polygon primitives as input, so such functionality was not provided in this extension. This extension requires that primitives provided to the GL must match the input primitive type of the active geometry shader (if any). QUADS, QUAD_STRIP, and POLYGON primitives are considered not to match any input primitive type, so an INVALID_OPERATION error will result.

The NV_geometry_shader4 extension (built on top of this one) allows applications to provide quads or general polygon primitives to a geometry shader with an input primitive type of TRIANGLES. Such primitives are decomposed into triangles, and a geometry shader is run on each triangle independently.

31. *Geometry shaders provide a limit on the number of vertices that can be emitted. Can this limit be changed at dynamically?*

    RESOLVED: See also issue 17. Not in this extension. This functionality was not provided because it would be an expensive operation on some implementations of this extension. The NV_geometry_shader4 extension (layered on top of this one) does allow applications to change this limit dynamically.

    An application can change the vertex output limit at any time. To allow for the possibility of dynamic changes (as in NV_geometry_shader4) but not require it, a limit change is not guaranteed to take effect unless the program object is re-linked. However, there is no guarantee that such limit changes will not take effect immediately.

32. *See also issue 28. Each vertex emitted by a geometry shader can specify a layer to render to using the output variable "gl_Layer". For LINE_STRIP and TRIANGLE_STRIP output primitive types, which vertex's layer is used?*

    RESOLVED: The vertex from which the layer is extracted is unfortunately undefined. In practice, some implementations of this extension will extract the layer number from the first vertex of the output primitive; others will extract it from the last (provoking) vertex. A future geometry shader extension may choose to define this behavior one way or the other.

    To get portable results, the layer number should be the same for all vertices in any single primitive emitted by the geometry shader. The EndPrimitive() built-in function available in a geometry shader starts a new primitive, and the layer number emitted can be safely changed after EndPrimitive() is called.

33. The grammar allows "varying", "varying out", and "varying in" as type-qualifiers for geometry shaders. What does "varying" without "in" or "out" mean for a geometry shader?

    RESOLVED: The "varying" type qualifier in a geometry shader not followed by "in" or "out" means the same as "varying out".

    This is consistent with the specification saying: "In order to seamlessly be able to insert or remove a geometry shader from a program object, the rules, names and types of the output built-in varying variables and user-defined varying variables are the same as for the vertex shader."

Revision History

Rev.	Date	Author	Changes
17	05/22/07	mjk	Clarify that "varying" means the same as "varying out" in a geometry shader.
16	01/10/07	pbrown	Specify that the total component limit is enforced at LinkProgram time.

```
    15   12/15/06  pbrown    Documented that the '#extension' token
                              for this extension should begin with "GL_",
                              as apparently called for per convention.

    14      --               Pre-release revisions.
```

Name

EXT_gpu_shader4

Name Strings

GL_EXT_gpu_shader4

Contact

Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)
Pat Brown, NVIDIA (pbrown 'at' nvidia.com)

Status

Multi vendor extension

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 02/04/2008
Author revision: 12

Number

326

Dependencies

OpenGL 2.0 is required.

This extension is written against the OpenGL 2.0 specification and version 1.10.59 of the OpenGL Shading Language specification.

This extension trivially interacts with ARB_texture_rectangle.

This extension trivially interacts with GL_EXT_texture_array.

This extension trivially interacts with GL_EXT_texture_integer.

This extension trivially interacts with GL_EXT_geometry_shader4

This extension trivially interacts with GL_EXT_texture_buffer_object.

NV_primitive_restart trivially affects the definition of this extension.

ARB_color_buffer_float affects the definition of this extension.
EXT_draw_instanced affects the definition of this extension.

Overview

This extension provides a set of new features to the OpenGL Shading Language and related APIs to support capabilities of new hardware. In particular, this extension provides the following functionality:

* New texture lookup functions are provided that allow shaders to

access individual texels using integer coordinates referring to the texel location and level of detail. No filtering is performed. These functions allow applications to use textures as one-, two-, and three-dimensional arrays.

* New texture lookup functions are provided that allow shaders to query the dimensions of a specific level-of-detail image of a texture object.

* New texture lookup functions variants are provided that allow shaders to pass a constant integer vector used to offset the texel locations used during the lookup to assist in custom texture filtering operations.

* New texture lookup functions are provided that allow shaders to access one- and two-dimensional array textures. The second, or third, coordinate is used to select the layer of the array to access.

* New "Grad" texture lookup functions are provided that allow shaders to explicitely pass in derivative values which are used by the GL to compute the level-of-detail when performing a texture lookup.

* A new texture lookup function is provided to access a buffer texture.

* The existing absolute LOD texture lookup functions are no longer restricted to the vertex shader only.

* The ability to specify and use cubemap textures with a DEPTH_COMPONENT internal format. This also enables shadow mapping on cubemaps. The 'q' coordinate is used as the reference value for comparisons. A set of new texture lookup functions is provided to lookup into shadow cubemaps.

* The ability to specify if varying variables are interpolated in a non-perspective correct manner, if they are flat shaded or, if multi-sampling, if centroid sampling should be performed.

* Full signed integer and unsigned integer support in the OpenGL Shading Language:

    - Integers are defined as 32 bit values using two's complement.

    - Unsigned integers and vectors thereof are added.

    - New texture lookup functions are provided that return integer values. These functions are to be used in conjunction with new texture formats whose components are actual integers, rather than integers that encode a floating-point value. To support these lookup functions, new integer and unsigned-integer sampler types are introduced.

    - Integer bitwise operators are now enabled.

    - Several built-in functions and operators now operate on integers or vectors of integers.

```
            - New vertex attribute functions are added that load integer
              attribute data and can be referenced in a vertex shader as
              integer data.

            - New uniform loading commands are added to load unsigned integer
              data.

            - Varying variables can now be (unsigned) integers. If declared
              as such, they have to be flat shaded.

            - Fragment shaders can define their own output variables, and
              declare them to be of type floating-point, integer or unsigned
              integer. These variables are bound to a fragment color index
              with the new API command BindFragDataLocationEXT(), and directed
              to buffers using the existing DrawBuffer or DrawBuffers API
              commands.

        * Added new built-in functions truncate() and round() to the shading
          language.

        * A new built-in variable accessible from within vertex shaders that
          holds the index <i> implicitly passed to ArrayElement to specify the
          vertex. This is called the vertex ID.

        * A new built-in variable accessible from within fragment and geometry
          shaders that hold the index of the currently processed
          primitive. This is called the primitive ID.

    This extension also briefly mentions a new shader type, called a geometry
    shader. A geometry shader is run after vertices are transformed, but
    before clipping. A geometry shader begins with a single primitive (point,
    line, triangle. It can read the attributes of any of the vertices in the
    primitive and use them to generate new primitives. A geometry shader has a
    fixed output primitive type (point, line strip, or triangle strip) and
    emits vertices to define a new primitive. Geometry shaders are discussed
    in detail in the GL_EXT_geometry_shader4 specification.
```

New Procedures and Functions

```
    void VertexAttribI1iEXT(uint index, int x);
    void VertexAttribI2iEXT(uint index, int x, int y);
    void VertexAttribI3iEXT(uint index, int x, int y, int z);
    void VertexAttribI4iEXT(uint index, int x, int y, int z, int w);

    void VertexAttribI1uiEXT(uint index, uint x);
    void VertexAttribI2uiEXT(uint index, uint x, uint y);
    void VertexAttribI3uiEXT(uint index, uint x, uint y, uint z);
    void VertexAttribI4uiEXT(uint index, uint x, uint y, uint z,
                             uint w);

    void VertexAttribI1ivEXT(uint index, const int *v);
    void VertexAttribI2ivEXT(uint index, const int *v);
    void VertexAttribI3ivEXT(uint index, const int *v);
    void VertexAttribI4ivEXT(uint index, const int *v);
```

```
    void VertexAttribI1uivEXT(uint index, const uint *v);
    void VertexAttribI2uivEXT(uint index, const uint *v);
    void VertexAttribI3uivEXT(uint index, const uint *v);
    void VertexAttribI4uivEXT(uint index, const uint *v);

    void VertexAttribI4bvEXT(uint index, const byte *v);
    void VertexAttribI4svEXT(uint index, const short *v);
    void VertexAttribI4ubvEXT(uint index, const ubyte *v);
    void VertexAttribI4usvEXT(uint index, const ushort *v);

    void VertexAttribIPointerEXT(uint index, int size, enum type,
                                 sizei stride, const void *pointer);

    void GetVertexAttribIivEXT(uint index, enum pname, int *params);
    void GetVertexAttribIuivEXT(uint index, enum pname,
                                uint *params);

    void Uniform1uiEXT(int location, uint v0);
    void Uniform2uiEXT(int location, uint v0, uint v1);
    void Uniform3uiEXT(int location, uint v0, uint v1, uint v2);
    void Uniform4uiEXT(int location, uint v0, uint v1, uint v2,
                       uint v3);

    void Uniform1uivEXT(int location, sizei count, const uint *value);
    void Uniform2uivEXT(int location, sizei count, const uint *value);
    void Uniform3uivEXT(int location, sizei count, const uint *value);
    void Uniform4uivEXT(int location, sizei count, const uint *value);

    void GetUniformuivEXT(uint program, int location, uint *params);

    void BindFragDataLocationEXT(uint program, uint colorNumber,
                                 const char *name);
    int GetFragDataLocationEXT(uint program, const char *name);
```

New Tokens

Accepted by the <pname> parameters of GetVertexAttribdv, GetVertexAttribfv, GetVertexAttribiv, GetVertexAttribIuivEXT and GetVertexAttribIivEXT:

```
  VERTEX_ATTRIB_ARRAY_INTEGER_EXT                    0x88FD
```

Returned by the <type> parameter of GetActiveUniform:

SAMPLER_1D_ARRAY_EXT	0x8DC0
SAMPLER_2D_ARRAY_EXT	0x8DC1
SAMPLER_BUFFER_EXT	0x8DC2
SAMPLER_1D_ARRAY_SHADOW_EXT	0x8DC3
SAMPLER_2D_ARRAY_SHADOW_EXT	0x8DC4
SAMPLER_CUBE_SHADOW_EXT	0x8DC5
UNSIGNED_INT	0x1405
UNSIGNED_INT_VEC2_EXT	0x8DC6
UNSIGNED_INT_VEC3_EXT	0x8DC7
UNSIGNED_INT_VEC4_EXT	0x8DC8
INT_SAMPLER_1D_EXT	0x8DC9
INT_SAMPLER_2D_EXT	0x8DCA
INT_SAMPLER_3D_EXT	0x8DCB
INT_SAMPLER_CUBE_EXT	0x8DCC
INT_SAMPLER_2D_RECT_EXT	0x8DCD
INT_SAMPLER_1D_ARRAY_EXT	0x8DCE
INT_SAMPLER_2D_ARRAY_EXT	0x8DCF
INT_SAMPLER_BUFFER_EXT	0x8DD0
UNSIGNED_INT_SAMPLER_1D_EXT	0x8DD1
UNSIGNED_INT_SAMPLER_2D_EXT	0x8DD2
UNSIGNED_INT_SAMPLER_3D_EXT	0x8DD3
UNSIGNED_INT_SAMPLER_CUBE_EXT	0x8DD4
UNSIGNED_INT_SAMPLER_2D_RECT_EXT	0x8DD5
UNSIGNED_INT_SAMPLER_1D_ARRAY_EXT	0x8DD6
UNSIGNED_INT_SAMPLER_2D_ARRAY_EXT	0x8DD7
UNSIGNED_INT_SAMPLER_BUFFER_EXT	0x8DD8

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

MIN_PROGRAM_TEXEL_OFFSET_EXT	0x8904
MAX_PROGRAM_TEXEL_OFFSET_EXT	0x8905

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.7 "Vertex Specification", p.20

Insert before last paragraph, p.22:

The VertexAttrib* commands described so far should not be used to load data for vertex attributes declared as signed or unsigned integers or vectors thereof in a vertex shader. If they are used to load signed or unsigned integer vertex attributes, the value in those attributes are undefined. Instead use the commands

```
void VertexAttribI[1234]{i,ui}EXT(uint index, T values);
void VertexAttribI[1234]{i,ui}vEXT(uint index, T values);
void VertexAttribI4{b,s,ub,us}vEXT(uint index, T values);
```

to specify fixed-point attributes that are not converted to floating-point. These attributes can be accessed in vertex shaders that declare attributes as signed or unsigned integers or vectors. The VertexAttribI4* commands extend the data passed in to a full signed or unsigned integer. If a VertexAttribI* command is used that does not match

the type of the attribute declared in a vertex shader, the values in the attributes are undefined. This means that the unsigned versions of the VertexAttribI* commands need to be used to load data for unsigned integer vertex attributes or vectors, and the signed versions of the VertexAttribI* commands for signed integer vertex attributes or vectors. Note that this also means that the VertexAttribI* commands should not be used to load data for a vertex attribute declared as a float, float vector or matrix, otherwise their values are undefined.

Insert at end of function list, p.24:

```
void VertexAttribIPointerEXT(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

(modify last paragraph, p.24) The <index> parameter in the VertexAttribPointer and VertexAttribIPointerEXT commands identify the generic vertex attribute array being described. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS. Generic attribute arrays with integer <type> arguments can be handled in one of three ways: converted to float by normalizing to [0,1] or [-1,1] as specified in table 2.9, converted directly to float, or left as integers. Data for an array specified by VertexAttribPointer will be converted to floating-point by normalizing if the <normalized> parameter is TRUE, and converted directly to floating-point otherwise. Data for an array specified by VertexAttribIPointerEXT will always be left as integer values.

(modify Table 2.4, p. 25)

Command	Sizes	Integer Handling	Types
VertexPointer	2,3,4	cast	...
NormalPointer	3	normalize	...
ColorPointer	3,4	normalize	...
SecondaryColorPointer	3	normalize	...
IndexPointer	1	cast	...
FogCoordPointer	1	n/a	...
TexCoordPointer	1,2,3,4	cast	...
EdgeFlagPointer	1	integer	...
VertexAttribPointer	1,2,3,4	flag	...
VertexAttribIPointerEXT	1,2,3,4	integer	byte, ubyte, short, ushort, int, uint

Table 2.4: Vertex array sizes (values per vertex) and data types. The "integer handling" column indicates how fixed-point data types are handled: "cast" means that they converted to floating-point directly, "normalize" means that they are converted to floating-point by normalizing to [0,1] (for unsigned types) or [-1,1] (for signed types), "integer" means that they remain as integer values, and "flag" means that either "cast" or "normalized" applies, depending on the setting of the <normalized> flag in VertexAttribPointer.

```
(modify end of pseudo-code, pp. 27-28)

  for (j = 1; j < genericAttributes; j++) {
    if (generic vertex attribute j array enabled) {
      if (generic vertex attribute j array is a pure integer array)
      {
        VertexAttribI[size][type]vEXT(j, generic vertex attribute j
                                        array element i);
      } else if (generic vertex attribute j array normalization
                 flag is set and <type> is not FLOAT or DOUBLE) {
        VertexAttrib[size]N[type]v(j, generic verex attribute j
                                      array element i);
      } else {
        VertexAttrib[size][type]v(j, generic verex attribute j
                                     array element i);
      }
    }
  }

  if (generic vertex attribute 0 array enabled) {
    if (generic vertex attribute 0 array is a pure integer array) {
      VertexAttribI[size][type]vEXT(0, generic verex attribute 0
                                      array element i);
    } else if (generic vertex attribute 0 array normalization flag
               is set and <type> is not FLOAT or DOUBLE) {
     VertexAttrib[size]N[type]v(0, generic verex attribute 0
                                   array element i);
    } else {
      VertexAttrib[size][type]v(0, generic verex attribute 0
                                   array element i);
    }
  }
```

Modify section 2.14.7, "Flatshading", p. 69

Add a new paragraph at the end of the section on p. 70 as follows:

If a vertex or geometry shader is active, the flat shading control described so far applies to the built-in varying variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor. Through the OpenGL Shading Language varying qualifier flat any vertex attribute can be flagged to be flat-shaded. See the OpenGL Shading Language Specification section 4.3.6 for more information.

Modify section 2.14.8, "Color and Associated Data Clipping", p. 71

Add to the end of this section:

For vertex shader varying variables specified to be interpolated without perspective correction (using the noperspective keyword), the value of t used to obtain the varying value associated with P will be adjusted to produce results that vary linearly in screen space.

Modify section 2.15.3, "Shader Variables", page 75

Add the following new return types to the description of GetActiveUniform on p. 81.

```
  SAMPLER_1D_ARRAY_EXT,
  SAMPLER_2D_ARRAY_EXT,
  SAMPLER_1D_ARRAY_SHADOW_EXT,
  SAMPLER_2D_ARRAY_SHADOW_EXT,
  SAMPLER_CUBE_SHADOW_EXT,
  SAMPLER_BUFFER_EXT,

  INT_SAMPLER_1D_EXT,
  INT_SAMPLER_2D_EXT,
  INT_SAMPLER_3D_EXT,
  INT_SAMPLER_CUBE_EXT,
  INT_SAMPLER_2D_RECT_EXT,
  INT_SAMPLER_1D_ARRAY_EXT,
  INT_SAMPLER_2D_ARRAY_EXT,
  INT_SAMPLER_BUFFER_EXT,

  UNSIGNED_INT,
  UNSIGNED_INT_VEC2_EXT,
  UNSIGNED_INT_VEC3_EXT,
  UNSIGNED_INT_VEC4_EXT,
  UNSIGNED_INT_SAMPLER_1D_EXT,
  UNSIGNED_INT_SAMPLER_2D_EXT,
  UNSIGNED_INT_SAMPLER_3D_EXT,
  UNSIGNED_INT_SAMPLER_CUBE_EXT,
  UNSIGNED_INT_SAMPLER_2D_RECT_EXT,
  UNSIGNED_INT_SAMPLER_1D_ARRAY_EXT,
  UNSIGNED_INT_SAMPLER_2D_ARRAY_EXT,
  UNSIGNED_INT_SAMPLER_BUFFER_EXT.
```

Add the following uniform loading command prototypes on p. 81 as follows:

```
  void Uniform{1234}uiEXT(int location, T value);
  void Uniform{1234}uivEXT(int location, sizei count, T value);
```

(add the following paragraph to the description of the above commands)

The Uniform*ui{v} commands will load count sets of one to four unsigned integer values into a uniform location defined as a unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

(change the first sentence of the last paragraph as follows)

When loading values for a uniform declared as a Boolean, the Uniform*i{v}, Uniform*ui{v} and Uniform*f{v} set of commands can be used to load boolean values.

Modify section 2.15.4 Shader execution, p. 84.

Add a new section "2.15.4.1 Shader Only Texturing" before the sub-section "Texture Access" on p. 85

This section describes texture functionality that is only accessible through vertex, geometry or fragment shaders. Also refer to the OpenGL Shading Language Specification, section 8.7 and Section 3.8 of the OpenGL 2.0 specification.

Note: For unextended OpenGL 2.0 and the OpenGL Shading Language version 1.20, all supported texture internal formats store unsigned integer values but return floating-point results in the range [0, 1] and are considered unsigned "normalized" integers. The ARB_texture_float extension introduces floating-point internal format where components are both stored and returned as floating-point values, and are not clamped. The EXT_texture_integer extension introduces formats that store either signed or unsigned integer values.

This extension defines additional OpenGL Shading Language texture lookup functions, see section 8.7 of the OpenGL Shading Language, that return either signed or unsigned integer values if the internal format of the texture is signed or unsigned, respectively.

Texel Fetches

The OpenGL Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer coordinates passed to the texel fetch functions are used directly as the texel coordinates (i, j, k) into the texture image. This in turn means the texture image is point-sampled (no filtering is performed).

The level of detail accessed is computed by adding the specified level-of-detail parameter <lod> to the base level of the texture, level_base.

The texel fetch functions can not perform depth comparisons or access cube maps. Unlike filtered texel accesses, texel fetches do not support LOD clamping or any texture wrap mode, and require a mipmapped minification filter to access any level of detail other than the base level.

The results of the texel fetch are undefined:

* if the computed LOD is less than the texture's base level (level_base) or greater than the maximum level (level_max),

* if the computed LOD is not the texture's base level and the texture's minification filter is NEAREST or LINEAR,

* if the layer specified for array textures is negative or greater than the number of layers in the array texture,

* if the texel at (i,j,k) coordinates refer to a border texel outside the defined extents of the specified LOD, where

  ```
  i < -b_s, j < -b_s, k < -b_s,
  i >= w_s - b_s, j >= h_s - b_s, or k >= d_s - b_s,
  ```

  where the size parameters (w_s, h_s, d_s, and b_s) refer to the width, height, depth, and border size of the image, as in equations 3.15, 3.16, and 3.17, or

. if the texture being accessed is not complete (or cube complete for cubemaps).

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value <lod> passed in as an argument to the texture size functions is added to the level_base of the texture to determine a texture image level. The dimensions of that image level, excluding a possible border, are then returned. If the computed texture image level is outside the range [level_base, level_max], the results are undefined. When querying the size of an array texture, both the dimensions and the layer index are returned. Note that buffer textures do not support mipmapping, therefore the previous lod discussion does not apply to buffer textures

Make the section "Texture Access" a subsection of 2.15.4.1

Modify the first paragraph on p. 86 as follows:

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

Add a new section "2.15.4.2 Shader Inputs" before "Position Invariance" on p. 86

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables gl_VertexID and gl_InstanceID. The gl_VertexID variable holds the integer index <i> implicitly passed to ArrayElement() to specify

the vertex. The variable gl_InstanceID holds the integer index of the current primitive in an instanced draw call. See also section 7.1 of the OpenGL Shading Language Specification.

Add a new section "2.15.4.3 Shader Outputs"

A vertex shader can write to built-in as well as user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. Refer to section 2.15.3 and the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

The built-in output variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output variable gl_TexCoord[] is an array and holds the set of texture coordinates for the current vertex.

The built-in output variable gl_FogFragCoord is used as the "c" value, as described in section 3.10 "Fog" of the OpenGL 2.0 specification.

The built-in special variable gl_Position is intended to hold the homogeneous vertex position. Writing gl_Position is optional.

The built-in special variable gl_ClipVertex holds the vertex coordinate used in the clipping stage, as described in section 2.12 "Clipping" of the OpenGL 2.0 specification.

The built in special variable gl_PointSize, if written, holds the size of the point to be rasterized, measured in pixels.

Number section "Position Invariance", "Validation" and "Undefined Behavior" as sections 2.15.4.4, 2.15.4.5, and 2.15.4.6 respectively.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.8.1, Texture Image Specification, p. 150

(modify 4th paragraph, p. 151 -- add cubemaps to the list of texture targets that can be used with DEPTH_COMPONENT textures)

Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if <target> is TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, or PROXY_TEXTURE_RECTANGLE_ARB. Using this format in conjunction with any other target will result in an INVALID_OPERATION error.

Modify Section 3.8.8, Texture Minification:

(replace the last paragraph, p. 171): Let s(x,y) be the function that associates an s texture coordinate with each set of window coordinates (x,y) that lie within a primitive; define t(x,y) and r(x,y) analogously.

Let

```
u(x,y) = w_t * s(x,y)
v(x,y) = h_t * t(x,y)    (3.20a)
w(x,y) = d_t * r(x,y)
```

where w_t, h_t, and d_t are as defined by equations 3.15, 3.16, and 3.17 with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is level_base. For a one-dimensional texture, define v(x,y) == 0 and w(x,y) == 0; for two-dimensional textures, define w(x,y) == 0.

(start a new paragraph with "For a polygon, rho is given at a fragment with window coordinates...", and then continue with the original spec text.)

(replace text starting with the last paragraph on p. 172, continuing to the end of p. 174)

The (u,v,w) coordinates are then modified, as follows:

```
u'(x,y) = u(x,y) + offsetu_shader,
v'(x,y) = v(x,y) + offsetv_shader,
w'(x,y) = w(x,y) + offsetw_shader
```

where (offsetu_shader, offsetv_shader, offsetw_shader) is the texel offset specified in the OpenGL Shading Language texture lookup functions that support offsets. If the texture function used does not support offsets, or for fixed-function texture accesses, all three shader offsets are taken to be zero.

The (u',v',w') coordinates are then further modified according the texture wrap modes, as specified in Table X.19, to generate a new set of coordinates (u'',v'',w'').

TEXTURE_WRAP_S	Coordinate Transformation
CLAMP	u'' = clamp(u', 0, w_t-0.5), if NEAREST filtering, clamp(u', 0, w_t), otherwise
CLAMP_TO_EDGE	u'' = clamp(u', 0.5, w_t-0.5)
CLAMP_TO_BORDER	u'' = clamp(u', -0.5, w_t+0.5)
REPEAT	u'' = clamp(fmod(u', w_t), 0.5, w_t-0.5)
MIRROR_CLAMP_EXT	u'' = clamp(fabs(u'), 0.5, w_t-0.5), if NEAREST filtering, or = clamp(fabs(u'), 0.5, w_t), otherwise
MIRROR_CLAMP_TO_EDGE_EXT	u'' = clamp(fabs(u'), 0.5, w_t-0.5)
MIRROR_CLAMP_TO_BORDER_EXT	u'' = clamp(fabs(u'), 0.5, w_t+0.5)
MIRRORED_REPEAT	u'' = w_t - clamp(fabs(w_t - fmod(u', 2*w_t)), 0.5, w_t-0.5)

Table X.19: Texel coordinate wrap mode application. clamp(a,b,c) returns b if a<b, c if a>c, and a otherwise. fmod(a,b) returns a-b*floor(a/b), and fabs(a) returns the absolute value of a. For the v and w coordinates, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t, respectively, are used.

When lambda indicates minification, the value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When TEXTURE_MIN_FILTER is NEAREST the texel in the image array of level level_base that is nearest (in Manhattan distance) to (u'',v'',w'') is obtained. The coordinate (i,j,k) is then computed as (floor(u''), floor(v''), floor(w'')).

For a three-dimensional texture, the texel at location (i,j,k) becomes the texture value. For a two-dimensional texture, k is irrelevant, and the texel at location (i,j) becomes the texture value. For a one-dimensional texture, j and k are irrelevant, and the texel at location i becomes the texture value.

If the selected (i,j,k), (i,j), or i location refers to a border texel that satisfies any of the following conditions:

```
i < -b_s,
j < -b_s,
k < -b_s,
i >= w_l + b_s,
j >= h_l + b_s, or
j >= d_l + b_s,
```

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of TEXTURE_BORDER_COLOR 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 depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When TEXTURE_MIN_FILTER is LINEAR, a 2x2x2 cube of texels in the image array of level level_base is selected. Let:

```
i_0   = floor(u'' - 0.5),
j_0   = floor(v'' - 0.5),
k_0   = floor(w'' - 0.5),
i_1   = i_0 + 1,
j_1   = j_0 + 1,
k_1   = k_0 + 1,
alpha = frac(u'' - 0.5),
beta  = frac(v'' - 0.5), and
gamma = frac(w'' - 0.5),
```

For a three-dimensional texture, the texture value tau is found as...

(replace last paragraph, p.174) For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

Rename section 3.8.9 "Texture Magnification" to section 3.8.8

modify the first paragraph of section 3.8.8 "Texture Magnification" as follows:

When lambda indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER and LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER, as described in the previous section, including the wrapping calculations. The level-of-detail level_base texture array is always used for magnification.

modify the last paragraph of section 3.8.8, p. 175, as follows:

The rules for NEAREST or LINEAR filtering are then applied to the selected array. Specifically, the coordinate (u,v,w) is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd'.

Modify the second paragraph on p. 176

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture valutes Tau1 and Tau2. Specifically, for level d1, the coordinate (u,v,w) is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd1'. For level d2 the coordinate (u', v', w') is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd2'.

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify 2nd paragraph, p. 188, indicating that the Q texture coordinate is used for depth comparisons on cubemap textures)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture

coordinate, clamped to the range [0, 1]. For texture lookups generated by an OpenGL Shading Language lookup function, let R be the reference value for depth comparisons provided in the lookup function, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows:

Modify section 3.11, Fragment Shaders, p. 193

Modify the third paragraph on p. 194 as follows:

Additionally, when a vertex shader is active, it may define one or more varying variables (see section 2.15.3 and the OpenGL Shading Language Specification). These values are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

Add the following paragraph to the end of section 3.11.1, p. 194

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations. Varying out variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. The subsection 'Shader Outputs' of the next section describes API how to direct these values to buffers.

Add a new paragraph at the beginning of the section "Texture Access", p. 194

Section 2.15.4.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

Modify the second paragraph on p. 195 as follows:

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

Add the following paragraph to the section Shader Inputs, p. 196

If a geometry shader is active, the built-in variable gl_PrimitiveID contains the ID value emitted by the geometry shader for the provoking vertex. If no geometry shader is active, gl_PrimitiveID is filled with the number of primitives processed by the rasterizer since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is undefined. The primitive ID is undefined for fragments generated by DrawPixels or Bitmap. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Modify the first paragraph of the section Shader Outputs, p. 196 as follows

The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These outputs are split into two categories. User-defined varying out variables and built-in variables. The built-in variables are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled, the final fragment color values or the final fragment data values or the final varying out variable values written by a fragment shader are clamped to the range [0,1] and then may be converted to fixed-point as described in section 2.14.9. Only user-defined varying out variables declared as a floating-point type are clamped and may be converted. If fragment clamping is disabled, the final fragment color values or the final fragment data values or the final varying output variable values are not modified. The final fragment depth written...

Modify the second paragraph of the section Shader Outputs, p. 196 as follows

...A fragment shader may not statically assign values to more than one of gl_FragColor, gl_FragData or any user-defined varying output variable. In this case, a compile or link error will result. A shader statically...

Add the following to the end of the section Shader Outputs, p. 197

The values of user-defined varying out variables are directed to a color buffer in a two step process. First the varying out variable is bound to a fragment color by using its number. The GL will assign a number to each varying out variable, unless overridden by the command BindFragDataLocationEXT(). The number of the fragment color assigned for each user-defined varying out variable can be queried with GetFragDataLocationEXT(). Next, the DrawBuffer or DrawBuffers commands (see section 4.2.1) direct each fragment color to a particular buffer.

The binding of a user-defined varying out variable to a fragment color number can be specified explicitly. The command

```
void BindFragDataLocationEXT(uint program, uint colorNumber,
                             const char *name);
```

specifies that the varying out variable name in program should be bound to fragment color colorNumber when the program is next linked. If name was bound previously, its assigned binding is replaced with colorNumber. name must be a null terminated string. The error INVALID_VALUE is generated if colorNumber is equal or greater than MAX_DRAW_BUFFERS. BindFragDataLocationEXT has no effect until the program is linked. In particular, it doesn't modify the bindings of varying out variables in a program that has already been linked. The error INVALID OPERATION is generated if name starts with the reserved "gl_" prefix.

When a program is linked, any varying out variables without a binding specified through BindFragDataLocationEXT will automatically be bound to fragment colors by the GL. Such bindings can be queried using the command GetFragDataLocationEXT. LinkProgram will fail if the assigned binding of a varying out variable would cause the GL to reference a non-existant fragment color number (one greater than or equal to MAX DRAW_BUFFERS). LinkProgram will also fail if more than one varying out variable is bound to the same number. This type of aliasing is not allowed.

BindFragDataLocationEXT may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with "gl_") to a color number, including a name that is never used as a varying out variable in any fragment shader object. Assigned bindings for variables that do not exist are ignored.

After a program object has been linked successfully, the bindings of varying out variable names to color numbers can be queried. The command

```
int GetFragDataLocationEXT(uint program, const char *name);
```

returns the number of the fragment color that the varying out variable name was bound to when the program object program was last linked. name must be a null terminated string. If program has not been successfully linked, the error INVALID OPERATION is generated. If name is not a varying out variable, or if an error occurs, -1 will be returned.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Section 4.2.1, Selecting a Buffer for Writing (p. 212)

(modify next-to-last paragraph, p. 213) If a fragment shader writes to gl_FragColor, DrawBuffers specifies a set of draw buffers into which the single fragment color defined by gl_FragColor is written. If a fragment shader writes to gl_FragData or a user-defined varying out variable, DrawBuffers specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to neither gl_FragColor, nor gl FragData, nor any user-defined varying out variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the commands VertexAttribIPointerEXT and BindFragDataLocationEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify section 6.1.14 "Shader and Program Queries", p. 256

Modify 2nd paragraph, p.259:

Add the following to the list of GetVertexAttrib* commands:

```
  void GetVertexAttribIivEXT(uint index, enum pname, int *params);
  void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);
```

obtain the... <pname> must be one of VERTEX_ATTRIB_ARRAY_ENABLED ,., VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER_EXT, or CURRENT_VERTEX_ATTRIB. ...

Split 3rd paragraph, p.259

... The size, stride, type, normalized flag, and unconverted integer flag are set by the commands VertexAttribPointer and VertexAttribIPointerEXT. The normalized flag is always set to FALSE by by VertexAttribIPointerEXT. The unconverted integer flag is always set to FALSE by VertexAttribPointer and TRUE by VertexAttribIPointerEXT.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute <index>. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to integer values; GetVertexAttribIivEXT reads and returns them as integers; GetVertexAttribIuivEXT reads and returns them as unsigned integers. The results of the query are undefined if the current attribute values are read using one data type but were specified using a different one. The error INVALID_OPERATION is generated if <index> is zero.

Change the prototypes in the first paragraph on page 260 as follows:

```
  void GetUniformfv(uint program, int location, float *params);
  void GetUniformiv(uint program, int location, int *params);
  void GetUniformuivEXT(uint program, int location, uint *params);
```

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with GL_ARB_color_buffer_float

If the GL_ARB_color_buffer_float extension is not supported then any reference to fragment clamping in section 3.11.2 "Shader Execution" needs to be deleted.

Interactions with GL_ARB_texture_rectangle

If the GL_ARB_texture_rectangle extension is not supported then all references to texture lookup functions with 'Rect' in the name need to be deleted.

Interactions with GL_EXT_texture_array

If the GL_EXT_texture_array extension is not supported, all references to one- and two-dimensional array texture sampler types (e.g., sampler1DArray, sampler2DArray) and the texture lookup functions that use them need to be deleted.

Interactions with GL_EXT_geometry_shader4

If the GL_EXT_geometry_shader4 extension is not supported, all references to a geometry shader need to be deleted.

Interactions with GL_NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count). If NV_primitive_restart is not supported, references to that extension in the discussion of the primitive ID counter should be removed.

If NV_primitive_restart is supported, index values causing a primitive restart are not considered as specifying an End command, followed by another Begin. Primitive restart is therefore not guaranteed to immediately update material properties when a vertex shader is active. The spec language on p.64 of the OpenGL 2.0 specification says "changes are not guaranteed to update material parameters, defined in table 2.11, until the following End command."

Interactions with EXT_texture_integer

If the EXT_texture_integer spec is not supported, the discussion about this spec in section 2.15.4.1 needs to be removed. All texture lookup functions that return integers or unsigned integers, as discussed in section 8.7 of the OpenGL Shading Language specification, also need to be removed.

Interactions with EXT_texture_buffer_object

If EXT_texture_buffer_object is not supported, references to buffer textures, as well as the texelFetchBuffer and texelSizeBuffer lookup functions and samplerBuffer types, need to be removed.

Interactions with EXT_draw_instanced

If EXT_draw_instanced is not supported, the value of gl_InstanceID is always zero.

Errors

The error INVALID_VALUE is generated by BindFragDataLocationEXT() if colorNumber is equal or greater than MAX_DRAW_BUFFERS.

The error INVALID OPERATION is generated by BindFragDataLocationEXT() if name starts with the reserved "gl_" prefix.

The error INVALID_OPERATOIN is generated by BindFragDataLocationEXT() or GetFragDataLocationEXT if program is not the name of a program object.

The error INVALID_OPERATION is generated by GetFragDataLocationEXT() if program has not been successfully linked.

New State

(add to table 6.7, p. 268)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY_INTEGER_EXT	16+xB	GetVertexAttrib	FALSE	vertex attrib array has unconverted ints	2.8	vertex-array

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MIN_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	-8	minimum texel offset allowed in lookup	2.x.4.4	-
MAX_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	+7	maximum texel offset allowed in lookup	2.x.4.4	-

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

Including the following line in a shader can be used to control the language features described in this extension:

```
#extension GL_EXT_gpu_shader4 : <behavior>
```

where <behavior> is as specified in section 3.3.

A new preprocessor #define is added to the OpenGL Shading Language:

```
#define GL_EXT_gpu_shader4 1
```

Add to section 3.6 "Keywords"

Add the following keywords:

```
noperspective, flat, centroid
```

Remove the unsigned keyword from the list of keywords reserved for future

```
use, and add it to the list of keywords.

The following new vector types are added:

  uvec2, uvec3, uvec4

The following new sampler types are added:

  sampler1DArray, sampler2DArray, sampler1DArrayShadow,
  sampler2DArrayShadow, samplerCubeShadow

  isampler1D, isampler2D, isampler3D, isamplerCube, isampler2DRect,
  isampler1DArray, isampler2DArray

  usampler1D, usampler2D, usampler3D, usamplerCube, usampler2DRect,
  usampler1DArray, usampler2DArray

  samplerBuffer, isamplerBuffer, usamplerBuffer
```

Add to section 4.1 "Basic Types"

```
Break the table in this section up in several tables. The first table
4.1.1 is named "scalar, vector and matrix data types". It includes the
first row through the 'mat4" row.

Add the following to the first section of this table:

  unsigned int           An unsigned integer
  uvec2                  A two-component unsigned integer vector
  uvec3                  A three-component unsigned integer vector
  uvec4                  A four-component unsigned integer vector

Break out the sampler types in a separate table, and name that table 4.1.2
"default sampler types". Add the following sampler types to this new
table:

  sampler1DArray         handle for accessing a 1D array texture
  sampler2DArray         handle for accessing a 2D array texture
  sampler1DArrayShadow   handle for accessing a 1D array depth texture
                         with comparison
  sampler2DArrayShadow   handle for accessing a 2D array depth texture
                         with comparison
  samplerBuffer          handle for accessing a buffer texture

Add a table 4.1.3 called "integer sampler types":

  isampler1D             handle for accessing an integer 1D texture
  isampler2D             handle for accessing an integer 2D texture
  isampler3D             handle for accessing an integer 3D texture
  isamplerCube           handle for accessing an integer cube map texture
  isampler2DRect         handle for accessing an integer rectangle texture
  isampler1DArray        handle for accessing an integer 1D array texture
  isampler2DArray        handle for accessing an integer 2D array texture
  isamplerBuffer         handle for accessing an integer buffer texture
```

Add a table 4.1.4 called "unsigned integer sampler types":

usampler1D	handle for accessing an unsigned integer 1D texture
usampler2D	handle for accessing an unsigned integer 2D texture
usampler3D	handle for accessing an unsigned integer 3D texture
usamplerCube	handle for accessing an unsigned integer cube map texture
usampler2DRect	handle for accessing an unsigned integer rectangle texture
usampler1DArray	handle for accessing an unsigned integer 1D array texture
usampler2DArray	handle for accessing an unsigned integer 2D array texture
usamplerBuffer	handle for accessing an unsigned integer buffer texture

Change section 4.1.3 "Integers"

Remove the first two paragraphs and replace with the following:

Signed, as well as unsigned integers, are fully supported. Integers hold whole numbers. Integers have at least 32 bits of precision, including a sign bit. Signed integers are stored using a two's complement representation.

Integers are declared and optionally initialized with integer expressions as in the following example:

```
int i, j = 42;
unsigned int k = 3u;
```

Literal integer constants can be expressed in decimal (base 10), octal (base 8), or hexadecimal (base 16) as follows.

```
integer-constant:
      decimal-constant integer-suffix_opt
      octal-constant integer-suffix_opt
      hexadecimal-constant integer-suffix_opt

integer-suffix:  one of
      u U
```

Change section 4.3 "Type Qualifiers"

Change the "varying" and "out" qualifier as follows:

varying - linkage between a vertex shader and fragment shader, or between a fragment shader and the back end of the OpenGL pipeline.

out - for function parameters passed back out of a function, but not initialized for use when passed in. Also for output varying variables (fragment only).

In the qualifier table, add the following sub-qualifiers under the varying qualifier:

```
  flat varying
  noperspective varying
  centroid varying
```

Change section 4.3.4 "Attribute"

Change the sentence:

The attribute qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4.

To:

The attribute qualifier can be used only with the data types int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3, uvec4, float, vec2, vec3, vec4, mat2, mat3, and mat4.

Change the fourth paragraph to:

It is expected that graphics hardware will have a small number of fixed locations for passing vertex attributes. Therefore, the OpenGL Shading language defines each non-matrix attribute variable as having space for up to four integer or floating-point values (i.e., a vec4, ivec4 or uvec4). There is an implementation dependent limit on the number of attribute variables that can be used and if this is exceeded it will cause a link error. (Declared attribute variables that are not used do not count against this limit.) A scalar attribute counts the same amount against this limit as a vector of size four, so applications may want to consider packing groups of four unrelated scalar attributes together into a vector to better utilize the capabilities of the underlying hardware. A mat4 attribute will...

Change section 4.3.6 "Varying"

Change the first paragraph to:

Varying variables provide the interface between the vertex shader, the fragment shader, and the fixed functionality between the vertex and fragment shader, as well as the interface from the fragment shader to the back-end of the OpenGL pipeline.

The vertex shader will compute values per vertex (such as color, texture coordinates, etc.) and write them to variables declared with the varying qualifier. A vertex shader may also read varying variables, getting back the same values it has written. Reading a varying variable in a vertex shader returns undefined values if it is read before being written.

The fragment shader will compute values per fragment and write them to variables declared with the varying out qualifier. A fragment shader may also read varying variables, getting back the same result it has written. Reading a varying variable in a fragment shader returns undefined values if it is read before being written.

Varying variables may be written more than once. If so, the last value assigned is the one used.

Change the second paragraph to:

Varying variables that are set per vertex are interpolated by default in a perspective-correct manner over the primitive being rendered, unless the varying is further qualified with noperspective. Interpolation in a perspective correct manner is specified in equations 3.6 and 3.8 in the OpenGL 2.0 specification. When noperspective is specified, interpolation must be linear in screen space, as described in equation 3.7 and the approximation that follows equation 3.8.

If single-sampling, the value is interpolated to the pixel's center, and the centroid qualifier, if present, is ignored. If multi-sampling, and the varying is not qualified with centroid, then the value must be interpolated to the pixel's center, or anywhere within the pixel, or to one of the pixel's samples. If multi-sampling and the varying is qualified with centroid, then the value must be interpolated to a point that lies in both the pixel and in the primitive being rendered, or to one of the pixel's samples that falls within the primitive.

[NOTE: Language for centroid sampling taken from the GLSL 1.20.4 specification]

Varying variables, set per vertex, can be computed on a per-primitive basis (flat shading), or interpolated over a line or polygon primitive (smooth shading). By default, a varying variable is smooth shaded, unless the varying is further qualified with flat. When smooth shading, the varying is interpolated over the primitive. When flat shading, the varying is constant over the primitive, and is taken from the single provoking vertex of the primitive, as described in Section 2.14.7 of the OpenGL 2.0 specification.

Change the fourth paragraph to:

The type and any qualifications (flat, noperspective, centroid) of varying variables with the same name declared in both the vertex and fragment shaders must match, otherwise the link command will fail. Note that built-in varying variables, which have names starting with "gl_", can not be further qualified with flat, noperspective or centroid. The flat keyword cannot be used together with either the noperspective or centroid keywords to further qualify a single varying variable, otherwise a compile error will occur. When using the keywords centroid, flat or noperspective, it must immediately precede the varying keyword. When using both centroid and noperspective keywords, either one can be specified first. Only those varying variables used (i.e. read) in the fragment shader must be written to by the vertex shader; declaring superfluous varying variables in the vertex shader is permissible. Varying out variables, set per fragment, can not be further qualified with flat, noperspective or centroid.

Fragment shaders output values to the back-end of the OpenGL pipeline using either user-defined varying out variables or built-in variables, as described in section 7.2, unless the discard keyword is executed. If the back-end of the OpenGL pipeline consumes a user-defined varying out variable and an execution of a fragment shader does not write a value to that variable, then the value consumed is undefined. If the back-end of

the OpenGL pipeline consumes a varying out variable and a fragment shader either writes values into less components of the variable, or if the variable is declared to have less components, than needed, the values of the missing component(s) are undefined. The OpenGL specification, section 3.x.x, describes API to route varying output variables to color buffers.

Add the following examples:

```
noperspective varying float temperature;
flat varying vec3 myColor;
centroid varying vec2 myTexCoord;
centroid noperspective varying vec2 myTexCoord;
varying out ivec3 foo;
```

Change the third paragraph on p. 25 as follows:

The "varying" qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4, int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3, uvec4 or arrays of these. Structures cannot be varying. If the varying is declared as one of the integer or unsigned integer data type variants, then it has to also be qualified as being flat shaded, otherwise a compile error will occur.

The "varying out" qualifier can be used only with the data types float, vec2, vec3, vec4, int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3 or uvec4. Structures or arrays cannot be declared as varying out.

Change section 5.1 "Operators"

Remove the "reserved" qualifications from the following operator precedence table entries:

Precedence	Operator class
3	(tilde is reserved)
4	(modulus reserved)
6	bit-wise shift (reserved)
9	bit-wise and (reserved)
10	bit-wise exclusive or (reserved)
11	bit-wise inclusive or (reserved)
16	(modulus, shift, and bit-wise are reserved)

Change section 5.8 "Assignments"

Change the first bullet from:

* The arithmetic assignments add into (+=)..

To:

* The arithmetic assignments add into (+=), subtract from (-=), multiply into (*=), and divide into (/=) as well as the assignments modulus into (%=), left shift by (<<=), right shift by (>>=), and into (&=), inclusive or into (|=), exclusive or into (^=). The expression

Delete the last bullet in this paragraph.

Remove the second bullet in the section starting with: The assignments modulus into..

Change section 5.9 "Expressions"

Change the bullet: The operator modulus (%) is reserved for future use to:

* The arithmetic operator % that operates on signed or unsigned integer typed expressions (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If the second operand is zero, results are undefined. If one operand is scalar and the other is a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. If both operands are non-negative, then the remainder is non-negative. Results are undefined if one, or both, operands are negative.

Change the last bullet: "Operators and (&), or (|), exclusive or (^), not (~), right-shift (>>), left shift (<<). These operators are reserved for future use." To the following bullets:

* The one's complement operator ~. The operand must be of type signed or unsigned integer (including vectors), and the result is the one's complement of its operand. If the operand is a vector, the operator is applied component-wise to the vector. If the operand is unsigned, the result is computed by subtracting the value from the largest unsigned integer value. If the operand is signed, the result is computed by converting the operand to an unsigned integer, applying ~, and converting back to a signed integer.

* The shift operators << and >>. For both operators, the operands must be of type signed or unsigned integer (including vectors). If the first operand is a scalar, the second operand has to be a scalar as well. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type. The value of E1 << E2 is E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 >> E2 is E1 right-shifted by E2 bit positions. If E1 is a signed integer, the right-shift will extend the sign bit. If E1 is an unsigned integer, the right-shift will zero-extend.

* The bitwise AND operator &. The operands must be of type signed or unsigned integer (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The result is the bitwise AND function of the operands.

* The bitwise exclusive OR operator ^. The operands must be of type signed or unsigned integer (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The result is the bitwise exclusive OR function of the operands.

* The bitwise inclusive OR operator |. The operands must be of type signed or unsigned integer (including vectors). The two operands must be of the same type, or one can be a signed or unsigned integer scalar and the other a signed or unsigned integer vector. If one operand is a scalar and the other a vector, the scalar is applied component-wise to the vector, resulting in the same type as the vector. The result is the bitwise inclusive OR function of the operands.

Change Section 7.1 "Vertex Shader Special Variables"

Add the following definition to the list of built-in variable definitions:

```
int gl_VertexID   // read-only
int gl_InstanceID // read-only
```

Add the following paragraph at the end of the section:

The variable gl_VertexID is available as a read-only variable from within vertex shaders and holds the integer index <i> implicitly passed to ArrayElement() to specify the vertex. The value of gl_VertexID is defined if and only if:

* the vertex comes from a vertex array command that specifies a complete primitive (e.g. DrawArrays, DrawElements),

* all enabled vertex arrays have non-zero buffer object bindings, and

* the vertex does not come from a display list, even if the display list was compiled using DrawArrays / DrawElements with data sourced from buffer objects.

The variable gl_InstanceID is availale as a read-only variable from within vertex shaders and holds holds the integer index of the current primitive in an instanced draw call (DrawArraysInstancedEXT, DrawElementsInstancedEXT). If the current primitive does not come from an instanced draw call, the value of gl_InstanceID is zero.

Change Section 7.2 "Fragment Shader Special Variables"

Change the 8th and 9th paragraphs on p. 43 as follows:

If a shader statically assigns a value to gl_FragColor, it may not assign a value to any element of gl_FragData nor to any user-defined varying output variable (section 4.3.6). If a shader statically writes a value to any element of gl_FragData, it may not assign a value to gl_FragColor nor to any user-defined varying output variable. That is, a shader may assign values to either gl_FragColor, gl_FragData, or any user-defined varying output variable, but not to a combination of the three options.

If a shader executes the discard keyword, the fragment is discarded, and the values of gl_FragDepth, gl_FragColor, gl_FragData and any user-defined varying output variables become irrelevant.

Add the following paragraph to the top of p. 44:

The variable gl_PrimitiveID is available as a read-only variable from within fragment shaders and holds the id of the currently processed primitive. Section 3.11, subsection "Shader Inputs" of the OpenGL 2.0 specification describes what value it holds based on the primitive type.

Add the following prototype to the list of built-in variables accessible from a fragment shader:

```
int gl_PrimitiveID;
```

Change Chapter 8, sixth paragraph on page 50:

Change the sentence:

When the built-in functions are specified below, where the input arguments (and corresponding output)can be float, vec2, vec3, or vec4, genType is used as the argument.

To:

When the built-in functions are specified below, where the input arguments (and corresponding output) can be float, vec2, vec3, or vec4, genType is used as the argument. Where the input arguments (and corresponding output) can be int, ivec2, ivec3 or ivec4, genIType is used as the argument. Where the input arguments (and corresponding output) can be unsigned int, uvec2, uvec3, or uvec4, genUType is used as the argument.

Add to section 8.3 "Common functions"

Add integer versions of the abs, sign, min, max and clamp functions, as follows:

Syntax:

```
genIType abs(genIType x)

genIType sign(genIType x)

genIType min(genIType x, genIType y)
genIType min(genIType x, int y)
genUType min(genUType x, genUType y)
genUType min(genUType x, unsigned int y)

genIType max(genIType x, genIType y)
genIType max(genIType x, int y)
genUType max(genUType x, genUType y)
genUType max(genUType x, unsigned int y)

genIType clamp(genIType x, genIType minval, genIType maxval)
genIType clamp(genIType x, int minval, int maxval)
genUType clamp(genUType x, genUType minval, genUType maxval)
genUType clamp(genUType x, unsigned int minval,
               unsigned int maxval)
```

Add the following new functions:

Syntax:

```
genType truncate(genType x)
```

Description:

Returns a value equal to the integer closest to x whose absolute value is not larger than the absolute value of x.

Syntax:

```
genType round(genType x)
```

Description:

Returns a value equal to the closest integer to x. If the fractional portion of the operand is 0.5, the nearest even integer is returned. For example, round (1.0) returns 1.0. round(-1.5) returns -2.0. round(3.5) and round (4.5) both return 4.0.

Add to section 8.6 "Vector Relational Functions"

Change the sentence:

Below, "bvec" is a placeholder for one of bvec2, bvec3, or bvec4, "ivec" is a placeholder for one of ivec2, ivec3, or ivec4, and "vec" is a placeholder for vec2, vec3, or vec4.

To:

Below, "bvec" is a placeholder for one of bvec2, bvec3, or bvec4, "ivec" is a placeholder for one of ivec2, ivec3, or ivec4, "uvec" is a placeholder for one of uvec2, uvec3 or uvec4 and "vec" is a placeholder for vec2, vec3, or vec4.

Add uvec versions of all but the any, all and not functions to the table in this section, as follows:

```
bvec lessThan(uvec x, uvec y)
bvec lessThanEqual(uvec x, uvec y)

bvec greaterThan(uvec x, uvec y)
bvec greaterThanEqual(uvec x, uvec y)

bvec equal(uvec x, uvec y)
bvec notEqual(uvec x, uvec y)
```

Add to section 8.7 "Texture Lookup Functions"

Remove the first sentence in the last paragraph:

"The built-ins suffixed with "Lod" are allowed only in a vertex shader.".

Add to this section:

Texture data can be stored by the GL as floating point, unsigned normalized integer, unsigned integer or signed integer data. This is determined by the type of the internal format of the texture. Texture lookups on unsigned normalized integer and floating point data return floating point values in the range [0, 1]. See also section 2.15.4.1 of the OpenGL specification.

Texture lookup functions are provided that can return their result as floating point, unsigned integer or signed integer, depending on the sampler type passed to the lookup function. Care must be taken to use the right sampler type for texture access. Table 8.xxx lists the supported combinations of sampler types and texture internal formats.

texture internal format	default (float) sampler	integer sampler	unsigned integer sampler
float	vec4	n/a	n/a
normalized	vec4	n/a	n/a
signed int	n/a	ivec4	n/a
unsigned int	n/a	n/a	uvec4

Table 8.xxx Valid combinations of the type of the internal format of a texture and the type of the sampler used to access the texture. Each cell in the table indicates the type of the return value of a texture lookup. N/a means this combination is not supported. A texture lookup using a n/a combination will return undefined values. The exceptions to this table are the "textureSize" lookup functions, which will return an integer or integer vector, regardless of the sampler type.

If a texture with a signed integer internal format is accessed, one of the signed integer sampler types must be used. If a texture with an unsigned integer internal format is accessed, one of the unsigned integer sampler types must be used. Otherwise, one of the default (float) sampler types must be used. If the types of a sampler and the corresponding texture internal format do not match, the result of a texture lookup is undefined.

If an integer sampler type is used, the result of a texture lookup is an ivec4. If an unsigned integer sampler type is used, the result of a texture lookup is a uvec4. If a default sampler type is used, the result of a texture lookup is a vec4, where each component is in the range [0, 1].

Integer and unsigned integer functions of all the texture lookup functions described in this section are also provided, except for the "shadow" versions, using function overloading. Their prototypes, however, are not listed separately. These overloaded functions use the integer or unsigned-integer versions of the sampler types and will return an ivec4 or an uvec4 respectively, except for the "textureSize" functions, which will always return an integer, or integer vector. Refer also to table 8.xxxx for valid combinations of texture internal formats and sampler types. For example, for the texture1D function, the complete set of prototypes is:

```
   vec4 texture1D(sampler1D sampler, float coord
                  [, float bias])
   ivec4 texture1D(isampler1D sampler, float coord
                   [, float bias])
   uvec4 texture1D(usampler1D sampler, float coord
                   [, float bias])

Add the following new texture lookup functions:

Syntax:

  vec4 texelFetch1D(sampler1D sampler, int coord, int lod)
  vec4 texelFetch2D(sampler2D sampler, ivec2 coord, int lod)
  vec4 texelFetch3D(sampler3D sampler, ivec3 coord, int lod)
  vec4 texelFetch2DRect(sampler2DRect sampler, ivec2 coord)
  vec4 texelFetch1DArray(sampler1DArray sampler, ivec2 coord, int lod)
  vec4 texelFetch2DArray(sampler2DArray sampler, ivec3 coord, int lod)

Description:

Use integer texture coordinate <coord> to lookup a single texel from the
level-of-detail <lod> on the texture bound to <sampler> as described in
section 2.15.4.1 of the OpenGL specification "Texel Fetches". For the
"array" versions, the layer of the texture array to access is either
coord.t or coord.p, depending on the use of the 1D or 2D texel fetch
lookup, respectively. Note that texelFetch2DRect does not take a
level-of-detail input.

Syntax:

  vec4 texelFetchBuffer(samplerBuffer sampler, int coord)

Description:

Use integer texture coordinate <coord> to lookup into the buffer texture
bound to <sampler>.

Syntax:

  int textureSizeBuffer(samplerBuffer sampler)
  int textureSize1D(sampler1D sampler, int lod)
  ivec2 textureSize2D(sampler2D sampler, int lod)
  ivec3 textureSize3D(sampler3D sampler, int lod)
  ivec2 textureSizeCube(samplerCube sampler, int lod)
  ivec2 textureSize2DRect(sampler2DRect sampler, int lod)
  ivec2 textureSize1DArray(sampler1DArray sampler, int lod)
  ivec3 textureSize2DArray(sampler2DArray sampler, int lod)

Description:

Returns the dimensions, width, height, depth, and number of layers, of
level <lod> for the texture bound to <sampler>, as described in section
2.15.4.1 of the OpenGL specification section "Texture Size Query". For the
textureSize1DArray function, the first (".x") component of the returned
vector is filled with the width of the texture image and the second
component with the number of layers in the texture array. For the
textureSize2DArray function, the first two components (".x" and ".y") of
```

```
the returned vector are filled with the width and height of the texture
image respectively. The third component (".z") is filled with the number
of layers in the texture array.

Syntax:

  vec4 texture1DArray(sampler1DArray sampler, vec2 coord
                      [, float bias])
  vec4 texture1DArrayLod(sampler1DArray sampler, vec2 coord,
                         float lod)

Description:

Use the first element (coord.s) of texture coordinate coord to do a
texture lookup in the layer indicated by the second coordinate coord.t of
the 1D texture array currently bound to sampler. The layer to access is
computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is
the depth of the texture array.

Syntax:

  vec4 texture2DArray(sampler2DArray sampler, vec3 coord
                      [, float bias])
  vec4 texture2DArrayLod(sampler2DArray sampler, vec3 coord,
                         float lod)
Description:

Use the first two elements (coord.s, coord.t) of texture coordinate coord
to do a texture lookup in the layer indicated by the third coordinate
coord.p of the 2D texture array currently bound to sampler. The layer to
access is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5))
where 'd' is the depth of the texture array.

Syntax:

  vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                     [float bias])
  vec4 shadow1DArrayLod(sampler1DArrayShadow sampler,
                        vec3 coord, float lod)
Description:

Use texture coordinate coord.s to do a depth comparison lookup on an array
layer of the depth texture bound to sampler, as described in section
3.8.14 of version 2.0 of the OpenGL specification. The layer to access is
indicated by the second coordinate coord.t and is computed by layer = max
(0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the
texture array. The third component of coord (coord.p) is used as the R
value. The texture bound to sampler must be a depth texture, or results
are undefined.

Syntax:

  vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
```

```
Description:

Use texture coordinate (coord.s, coord.t) to do a depth comparison lookup
on an array layer of the depth texture bound to sampler, as described in
section 3.8.14 of version 2.0 of the OpenGL specification. The layer to
access is indicated by the third coordinate coord.p and is computed by
layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth
of the texture array. The fourth component of coord (coord.q) is used as
the R value. The texture bound to sampler must be a depth texture, or
results are undefined.

Syntax:

    vec4 shadowCube(samplerCubeShadow sampler, vec4 coord)

Description:

Use texture coordinate (coord.s, coord.t, coord.p) to do a depth
comparison lookup on the depth cubemap bound to sampler, as described in
section 3.8.14. The direction of the vector (coord.s, coord.t, coord.p) is
used to select which face to do a two-dimensional texture lookup in, as
described in section 3.8.6 of the OpenGL 2.0 specification. The fourth
component of coord (coord.q) is used as the R value. The texture bound to
sampler must be a depth cubemap, otherwise results are undefined.

Syntax:

  vec4 texture1DGrad(sampler1D sampler, float coord,
                     float ddx, float ddy);
  vec4 texture1DProjGrad(sampler1D sampler, vec2 coord,
                         float ddx, float ddy);
  vec4 texture1DProjGrad(sampler1D sampler, vec4 coord,
                         float ddx, float ddy);
  vec4 texture1DArrayGrad(sampler1DArray sampler, vec2 coord,
                          float ddx, float ddy);

  vec4 texture2DGrad(sampler2D sampler, vec2 coord,
                     vec2 ddx, vec2 ddy);
  vec4 texture2DProjGrad(sampler2D sampler, vec3 coord,
                         vec2 ddx, vec2 ddy);
  vec4 texture2DProjGrad(sampler2D sampler, vec4 coord,
                         vec2 ddx, vec2 ddy);
  vec4 texture2DArrayGrad(sampler2DArray sampler, vec3 coord,
                          vec2 ddx, vec2 ddy);

  vec4 texture3DGrad(sampler3D sampler, vec3 coord,
                     vec3 ddx, vec3 ddy);
  vec4 texture3DProjGrad(sampler3D sampler, vec4 coord,
                         vec3 ddx, vec3 ddy);

  vec4 textureCubeGrad(samplerCube sampler, vec3 coord,
                       vec3 ddx, vec3 ddy);
```

```
  vec4 shadow1DGrad(sampler1DShadow sampler, vec3 coord,
                    float ddx, float ddy);
  vec4 shadow1DProjGrad(sampler1DShadow sampler, vec4 coord,
                        float ddx, float ddy);
  vec4 shadow1DArrayGrad(sampler1DArrayShadow sampler, vec3 coord,
                         float ddx, float ddy);

  vec4 shadow2DGrad(sampler2DShadow sampler, vec3 coord,
                    vec2 ddx, vec2 ddy);
  vec4 shadow2DProjGrad(sampler2DShadow sampler, vec4 coord,
                        vec2 ddx, vec2 ddy);
  vec4 shadow2DArrayGrad(sampler2DArrayShadow sampler, vec4 coord,
                         vec2 ddx, vec2 ddy);

  vec4 texture2DRectGrad(sampler2DRect sampler, vec2 coord,
                         vec2 ddx, vec2 ddy);
  vec4 texture2DRectProjGrad(sampler2DRect sampler, vec3 coord,
                             vec2 ddx, vec2 ddy);
  vec4 texture2DRectProjGrad(sampler2DRect sampler, vec4 coord,
                             vec2 ddx, vec2 ddy);

  vec4 shadow2DRectGrad(sampler2DRectShadow sampler, vec3 coord,
                        vec2 ddx, vec2 ddy);
  vec4 shadow2DRectProjGrad(sampler2DRectShadow sampler, vec4 coord,
                            vec2 ddx, vec2 ddy);

  vec4 shadowCubeGrad(samplerCubeShadow sampler, vec4 coord,
                      vec3 ddx, vec3 ddy);
```

Description:

The "Grad" functions map the partial derivatives ddx and ddy to ds/dx, dt/dx, dr/dx, and ds/dy, dt/dy, dr/dy respectively and use texture coordinate "coord" to do a texture lookup as described for their non "Grad" counterparts. The derivatives ddx and ddy are used as the explicit derivate of "coord" with respect to window x and window y respectively and are used to compute lambda_base(x,y) as in equation 3.18 in the OpenGL 2.0 specification. For the "Proj" versions, it is assumed that the partial derivatives ddx and ddy are already projected. I.e. the GL assumes that ddx and ddy represent d(s/q)/dx, d(t/q)/dx, d(r/q)/dx and d(s/q)/dy, d(t/q)/dy, d(r/q)/dy respectively. For the "Cube" versions, the partial derivatives ddx and ddy are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face. The partial derivatives of the post-projection texture coordinates, which are used for level-of-detail and anisotropic filtering calculations, are derived from coord, ddx and ddy in an implementation-dependent manner.

NOTE: Except for the "array" and shadowCubeGrad() functions, these functions are taken from the ARB_shader_texture_lod spec and are functionally equivalent.

```
Syntax:

  vec4 texture1DOffset(sampler1D sampler, float coord,
                       int offset [,float bias])
  vec4 texture1DProjOffset(sampler1D sampler, vec2 coord,
                           int offset [,float bias])
  vec4 texture1DProjOffset(sampler1D sampler, vec4 coord,
                           int offset [,float bias])
  vec4 texture1DLodOffset(sampler1D sampler, float coord,
                          float lod, int offset)
  vec4 texture1DProjLodOffset(sampler1D sampler, vec2 coord,
                              float lod, int offset)
  vec4 texture1DProjLodOffset(sampler1D sampler, vec4 coord,
                              float lod, int offset)

  vec4 texture2DOffset(sampler2D sampler, vec2 coord,
                       ivec2 offset [,float bias])
  vec4 texture2DProjOffset(sampler2D sampler, vec3 coord,
                           ivec2 offset [,float bias])
  vec4 texture2DProjOffset(sampler2D sampler, vec4 coord,
                           ivec2 offset [,float bias])
  vec4 texture2DLodOffset(sampler2D sampler, vec2 coord,
                          float lod, ivec2 offset)
  vec4 texture2DProjLodOffset(sampler2D sampler, vec3 coord,
                              float lod, ivec2 offset)
  vec4 texture2DProjLodOffset(sampler2D sampler, vec4 coord,
                              float lod, ivec2 offset)

  vec4 texture3DOffset(sampler3D sampler, vec3 coord,
                       ivec3 offset [,float bias])
  vec4 texture3DProjOffset(sampler3D sampler, vec4 coord,
                           ivec3 offset [,float bias])
  vec4 texture3DLodOffset(sampler3D sampler, vec3 coord,
                          float lod, ivec3 offset)
  vec4 texture3DProjLodOffset(sampler3D sampler, vec4 coord,
                              float lod, ivec3 offset)

  vec4 shadow1DOffset(sampler1DShadow sampler, vec3 coord,
                      int offset [,float bias])
  vec4 shadow2DOffset(sampler2DShadow sampler, vec3 coord,
                      ivec2 offset [,float bias])
  vec4 shadow1DProjOffset(sampler1DShadow sampler, vec4 coord,
                          int offset [,float bias])
  vec4 shadow2DProjOffset(sampler2DShadow sampler, vec4 coord,
                          ivec2 offset [,float bias])
  vec4 shadow1DLodOffset(sampler1DShadow sampler, vec3 coord,
                         float lod, int offset)
  vec4 shadow2DLodOffset(sampler2DShadow sampler, vec3 coord,
                         float lod, ivec2 offset)
  vec4 shadow1DProjLodOffset(sampler1DShadow sampler, vec4 coord,
                             float lod, int offset)
  vec4 shadow2DProjLodOffset(sampler2DShadow sampler, vec4 coord,
                             float lod, ivec2 offset)
```

```
vec4 texture2DRectOffset(sampler2DRect sampler, vec2 coord,
                         ivec2 offset)
vec4 texture2DRectProjOffset(sampler2DRect sampler, vec3 coord,
                          ivec2 offset)
vec4 texture2DRectProjOffset(sampler2DRect sampler, vec4 coord,
                          ivec2 offset)
vec4 shadow2DRectOffset(sampler2DRectShadow sampler, vec3 coord,
                        ivec2 offset)
vec4 shadow2DRectProjOffset(sampler2DRectShadow sampler, vec4 coord,
                           ivec2 offset)

vec4 texelFetch1DOffset(sampler1D sampler, int coord, int lod,
                        int offset)
vec4 texelFetch2DOffset(sampler2D sampler, ivec2 coord, int lod,
                        ivec2 offset)
vec4 texelFetch3DOffset(sampler3D sampler, ivec3 coord, int lod,
                        ivec3 offset)
vec4 texelFetch2DRectOffset(sampler2DRect sampler, ivec2 coord,
                           ivec2 offset)
vec4 texelFetch1DArrayOffset(sampler1DArray sampler, ivec2 coord,
                          int lod, int offset)
vec4 texelFetch2DArrayOffset(sampler2DArray sampler, ivec3 coord,
                          int lod, ivec2 offset)

vec4 texture1DArrayOffset(sampler1DArray sampler, vec2 coord,
                          int offset [, float bias])
vec4 texture1DArrayLodOffset(sampler1DArray sampler, vec2 coord,
                          float lod, int offset)

vec4 texture2DArrayOffset(sampler2DArray sampler, vec3 coord,
                          ivec2 offset [, float bias])
vec4 texture2DArrayLodOffset(sampler2DArray sampler, vec3 coord,
                          float lod, ivec2 offset)

vec4 shadow1DArrayOffset(sampler1DArrayShadow sampler, vec3 coord,
                         int offset, [float bias])
vec4 shadow1DArrayLodOffset(sampler1DArrayShadow sampler, vec3 coord,
                           float lod, int offset)

vec4 shadow2DArrayOffset(sampler2DArrayShadow sampler,
                         vec4 coord, ivec2 offset)

vec4 texture1DGradOffset(sampler1D sampler, float coord,
                         float ddx, float ddy, int offset);
vec4 texture1DProjGradOffset(sampler1D sampler, vec2 coord,
                          float ddx, float ddy, int offset);
vec4 texture1DProjGradOffset(sampler1D sampler, vec4 coord,
                          float ddx, float ddy, int offset);
vec4 texture1DArrayGradOffset(sampler1DArray sampler, vec2 coord,
                           float ddx, float ddy, int offset);
```

```
  vec4 texture2DGradOffset(sampler2D sampler, vec2 coord,
                           vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DProjGradOffset(sampler2D sampler, vec3 coord,
                               vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DProjGradOffset(sampler2D sampler, vec4 coord,
                               vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DArrayGradOffset(sampler2DArray sampler, vec3 coord,
                                vec2 ddx, vec2 ddy, ivec2 offset);

  vec4 texture3DGradOffset(sampler3D sampler, vec3 coord,
                           vec3 ddx, vec3 ddy, ivec3 offset);
  vec4 texture3DProjGradOffset(sampler3D sampler, vec4 coord,
                               vec3 ddx, vec3 ddy, ivec3 offset);

  vec4 shadow1DGradOffset(sampler1DShadow sampler, vec3 coord,
                          float ddx, float ddy, int offset);
  vec4 shadow1DProjGradOffset(sampler1DShadow sampler,
                              vec4 coord, float ddx, float ddy,
                              int offset);
  vec4 shadow1DArrayGradOffset(sampler1DArrayShadow sampler,
                               vec3 coord, float ddx, float ddy,
                               int offset);

  vec4 shadow2DGradOffset(sampler2DShadow sampler, vec3 coord,
                          vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 shadow2DProjGradOffset(sampler2DShadow sampler, vec4 coord,
                              vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 shadow2DArrayGradOffset(sampler2DArrayShadow sampler,
                               vec4 coord, vec2 ddx, vec2 ddy,
                               ivec2 offset);

  vec4 texture2DRectGradOffset(sampler2DRect sampler, vec2 coord,
                               vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DRectProjGradOffset(sampler2DRect sampler, vec3 coord,
                                   vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DRectProjGradOffset(sampler2DRect sampler, vec4 coord,
                                   vec2 ddx, vec2 ddy, ivec2 offset);

  vec4 shadow2DRectGradOffset(sampler2DRectShadow sampler,
                              vec3 coord, vec2 ddx, vec2 ddy,
                              ivec2 offset);
  vec4 shadow2DRectProjGradOffset(sampler2DRectShadow sampler,
                                  vec4 coord, vec2 ddx, vec2 ddy,
                                  ivec2 offset);
```

Description:

The "offset" version of each function provides an extra parameter <offset> which is added to the (u,v,w) texel coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by MIN_PROGRAM_TEXEL_OFFSET_EXT and MAX_PROGRAM_TEXEL_OFFSET_EXT, respectively. Note that <offset> does not apply to the layer coordinate for texture arrays. This is explained in detail in section 3.8.7 of the OpenGL Specification. Note that texel offsets are also not supported for cubemaps or buffer textures.

Add to section 9 "Grammar"

```
type_qualifer:
    CONST
    ATTRIBUTE  // Vertex only
    varying-modifier_opt VARYING
    UNIFORM

varying-modifier:
    FLAT
    CENTROID
    NOPERSPECTIVE

type_specifier:
    VOID
    FLOAT
    INT
    UNSIGNED_INT
    BOOL
```

Issues

1. Should we support shorts in GLSL?

DISCUSSION:

RESOLUTION: UNRESOLVED

2. Do bitwise shifts, AND, exclusive OR and inclusive OR support all combinations of scalars and vectors for each operand?

DISCUSSION: It seems sense to support scalar OP scalar, vector OP scalar and vector OP vector. But what about scalar OP vector? Should the scalar be promoted to a vector first?

RESOLUTION: RESOLVED. Yes, this should work essentially as the '+' operator. The scalar is applied to each component of the vector.

3. Which built-in functions should also operate on integers?

DISCUSSION: There are several that don't make sense to define to operate on integers at all, but the following can be debated: pow, sqrt, dot (and the functions that use dot), cross.

RESOLUTION: RESOLVED. Integer versions of the abs, sign, min, max and clamp functions are defined. Note that the modulus operator % has been defined for integer operands.

4. Do we need to support integer matrices?

DISCUSSION:

RESOLUTION: RESOLVED No, not at the moment.

5. Which texture array lookup functions do we need to support?

DISCUSSION: We don't want to support lookup functions that need more than four components passed as parameters. Components can be used for texture coordinates, layer selection, 'R' depth compare and the 'q' coordinate for projection. However, texture projection might be relatively easy to support through code-generation, thus we might be able to support functions that need five components, as long as one of them is 'q' for projective texturing. Specifically, should we support:

```
vec4 texture2DArrayProjLod(sampler2DArray sampler, vec4 coord,
                           float lod)
vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                   [float bias])
vec4 shadow1DArrayProj(sampler1DArrayShadow sampler, vec4 coord,
                       [float bias])
vec4 shadow1DArrayLod(sampler1DArrayShadow sampler, vec3 coord,
                      float lod)
vec4 shadow1DArrayProjLod(sampler1DArrayShadow sampler,
                          vec4 coord, float lod)
vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
vec4 shadow2DArrayProj(sampler2DArrayShadow sampler, vec4 coord,
                       float refValue)
```

RESOLUTION: RESOLVED, We'll support all but the "Proj" versions. The assembly spec (NV_gpu_program4) doesn't support the equivalent functionality, either.

6. How do we handle conversions between integer and unsigned integers?

DISCUSSION: Do we allow automatic type conversions between signed and unsigned integers?

RESOLUTION: RESOLVED. We will not add this until GLSL version 1.20 has been defined, and the implicit conversion rules have been established there. If we do this, we would likely only support implicit conversion from int to unsigned int, just like C does.

7. Should varying modifiers (flat, noperspective) apply to built-in varying variables also?

DISCUSSION: There is API to control flat vs smooth shading for colors through glShadeModel(). There is also API to hint if colors should be interpolated perspective correct, or not, through glHint(). These API commands apply to the built-in color varying variables (gl_FrontColor etc). If the varying modifiers in a shader also apply to the color built-ins, which has precedence?

RESOLUTION: RESOLVED. It is simplest and cleanest to only allow the varying modifiers to apply to user-defined varying variables. The behavior of the built-in color varying variables can still be controlled through the API.

8. How should perspective-incorrect interpolation (linear in screen space) and clipping interact?

RESOLVED: Primitives with attributes specified to be perspective-incorrect should be clipped so that the vertices introduced by clipping should have attribute values consistent with the interpolation mode. We do not want to have large color shifts introduced by clipping a perspective-incorrect attribute. For example, a primitive that approaches, but doesn't cross, a frustum clip plane should look pretty much identical to a similar primitive that just barely crosses the clip plane.

Clipping perspective-incorrect interpolants that cross the W==0 plane is very challenging. The attribute clipping equation provided in the spec effectively projects all the original vertices to screen space while ignoring the X and Y frustum clip plane. As W approaches zero, the projected X/Y window coordinates become extremely large. When clipping an edge with one vertex inside the frustum and the other out near infinity (after projection, due to W approaching zero), the interpolated attribute for the entire visible portion of the edge should almost exactly match the attribute value of the visible vertex.

If an outlying vertex approaches and then goes past W==0, it can be said to go "to infinity and beyond" in screen space. The correct answer for screen-linear interpolation is no longer obvious, at least to the author of this specification. Rather than trying to figure out what the "right" answer is or if one even exists, the results of clipping such edges is specified as undefined.

9. Do we need to support a non-MRT fragment shader writing to (unsigned) integer outputs?

DISCUSSION: Fragment shaders with only one fragment output are considered non-MRT shaders. This means that the output of the shader gets smeared across all color buffers attached to the framebuffer. Fragment shaders with multiple fragment outputs are MRT shaders. Each output is directed to a color buffer using the DrawBuffers API (for gl_FragData) and a combination of the BindFragDataLocationEXT and DrawBuffers API (for varying out variables). Before this extension, a non-MRT shader would write to gl_Color only. A shader writing to gl_FragData[] is a MRT shader. With the addition of varying out variables in this extension, any shader writing to a variable out variable is a MRT shader. It is not possible to construct a non-MRT shader writing to varying out variables. Varying out variables can be declared to be of type integer or unsigned integer. In order to support a non-MRT shader that can write to (unsigned) integer outputs, we could define two new built-in variables:

```
ivec4 gl_FragColorInt;
uvec4 gl_FragColorUInt;
```

Or we could add a special rule stating that if the program object writes to exactly one varying out variable, it is considered to be non-MRT.

RESOLUTION: NO. We don't care enough to support this.

10. Is section 2.14.8, "Color and Associated Data Clipping" in the core specification still correct?

DISCUSSION: This section is in need of some updating, now that varying variables can be interpolated without perspective correction. Some (not so precise) language has been added in the spec body, suggesting that the interpolation needs to be performed in such a way as to produce results that vary linearly in screen space. However, we could define the exact interpolation method required to achieve this. A suggested updated paragraph follows, but we'll leave updating section 2.14.8 to a future edit of the core specification, not this extension.

Replace Section 2.14.8, and rename it to "Vertex Attribute Clipping"

After lighting, clamping or masking and possible flatshading, vertex attributes, including colors, texture and fog coordinates, shader varying variables, and point sizes computed on a per vertex basis, are clipped. Those attributes associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the attributes assigned to vertices produced by clipping are produced by interpolating attributes along the clipped edge.

Let the attributes assigned to the two vertices P_1 and P_2 of an unclipped edge be a_1 and a_2. The value of t (section 2.12) for a clipped point P is used to obtain the attribute associated with P as

```
a = t * a_1 + (1-t) * a_2
```

unless the attribute is specified to be interpolated without perspective correction in a shader (using the noperspective keyword). In that case, the attribute associated with P is

```
a = t' * a_1 + (1-t') * a_2
```

where

```
t' = (t * w_1) / (t * w_1 + (1-t) * w_2)
```

and w_1 and w_2 are the w clip coordinates of P_1 and P_2, respectively. If w_1 or w_2 is either zero or negative, the value of the associated attribute is undefined.

For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For a vector attribute, it means multiplying each vector component by the scalar. Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Attribute clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

11. When and where in the texture filtering process are texel offsets applied?

DISCUSSION: Texel offsets are applied to the (u,v,w) coordinates of the base level of the texture if the texture filter mode does not indicate

mipmapping. Otherwise, texel offsets are applied to the (u,v,w) coordinates of the mipmap level 'd', as found by equation 3.27 or to mipmap levels 'd1' and 'd2' as found by equation 3.28 in the OpenGL 2.0 specification. In other words, texel offsets are applied to the (u,v,w) coordinate of whatever mipmap level is accessed.

12. *Why is writing to the built-in output variable "gl_Position" in a vertex shader now optional?*

    DISCUSSION: Before this specification, writing to gl_Position in a vertex shader was mandatory. The GL pipeline required a vertex position to be written in order to produce well-defined output. This is still the case if the GL pipeline indeed needs a vertex position. However, with fourth-generation programmable hardware there are now cases where the GL pipeline no longer requires a vertex position in order to produce well-defined results. If a geometry shader is present, the vertex shader does not need to write to gl_Position anymore. Instead, the geometry shader can compute a vertex position and write to its gl_Position output. In case of transform-feedback, where the output of a vertex or geometry shader is streamed to one or more buffer objects, perfectly valid results can be obtained without either the vertex shader nor geometry shader writing to gl_Position. The transform-feedback specification adds a new enable to discard primitives right before rasterization, making it potentially unnecessary to write to gl_Position.

Revision History

Rev.	Date	Author	Changes
12	02/04/08	pbrown	Fix errors in texture wrap mode handling. Added a missing clamp to avoid sampling border in REPEAT mode. Fixed incorrectly specified weights for LINEAR filtering.
11	05/08/07	pbrown	Add VertexAttribIPointerEXT to the list of commands that can't go in display lists.
10	01/23/07	pbrown	Fix prototypes for a variety of functions that were specified with an incorrect sampler type.
9	12/15/06	pbrown	Documented that the '#extension' token for this extension should begin with "GL_", as apparently called for per convention.
8	--		Pre-release revisions.

Name

EXT_packed_float

Name Strings

GL_EXT_packed_float
WGL_EXT_pixel_format_packed_float
GLX_EXT_fbconfig_packed_float

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Date: November 6, 2006
Revision: 0.4

Number

328

Dependencies

OpenGL 1.1 required

ARB_color_buffer_float affects this extension.

EXT_texture_shared_exponent trivially affects this extension.

EXT_framebuffer_object affects this extension.

WGL_ARB_pixel_format is required for use with WGL.

WGL_ARB_pbuffer affects WGL pbuffer support for this extension.

GLX 1.3 is required for use with GLX.

This extension is written against the OpenGL 2.0 (September 7, 2004) specification.

Overview

This extension adds a new 3-component floating-point texture format that fits within a single 32-bit word. This format stores 5 bits of biased exponent per component in the same manner as 16-bit floating-point formats, but rather than 10 mantissa bits, the red, green, and blue components have 6, 6, and 5 bits respectively. Each mantissa is assumed to have an implied leading one except in the denorm exponent case. There is no sign bit so only non-negative values can be represented. Positive infinity, positive denorms,

and positive NaN values are representable. The value of the fourth component returned by a texture fetch is always 1.0.

This extension also provides support for rendering into an unsigned floating-point rendering format with the assumption that the texture format described above could also be advertised as an unsigned floating-point format for rendering.

The extension also provides a pixel external format for specifying packed float values directly.

New Procedures and Functions

None

New Tokens

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

R11F_G11F_B10F_EXT	0x8C3A

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable:

UNSIGNED_INT_10F_11F_11F_REV_EXT	0x8C3B

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

RGBA_SIGNED_COMPONENTS_EXT	0x8C3C

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_TYPE_RGBA_UNSIGNED_FLOAT_EXT	0x20A8

Accepted as values of the <render_type> arguments in the glXCreateNewContext and glXCreateContext functions

GLX_RGBA_UNSIGNED_FLOAT_TYPE_EXT	0x20B1

Returned by glXGetFBConfigAttrib (when <attribute> is set to GLX_RENDER_TYPE) and accepted by the <attrib_list> parameter of glXChooseFBConfig (following the GLX_RENDER_TYPE token):

GLX_RGBA_UNSIGNED_FLOAT_BIT_EXT	0x00000008

Additions to Chapter 2 of the 2.0 Specification (OpenGL Operation)

-- Add two new sections after Section 2.1.2, (page 6):

2.1.A Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value of an unsigned 11-bit floating-point number (represented as an 11-bit unsigned integer N) is determined by the following:

```
    0.0,                      if E == 0 and M == 0,
    2^-14 * (M / 64),         if E == 0 and M != 0,
    2^(E-15) * (1 + M/64),    if 0 < E < 31,
    INF,                      if E == 31 and M == 0, or
    NaN,                      if E == 31 and M != 0,
```

where

```
    E = floor(N / 64), and
    M = N mod 64.
```

Implementations are also allowed to use any of the following alternative encodings:

```
    0.0,                      if E == 0 and M != 0
    2^(E-15) * (1 + M/64)     if E == 31 and M == 0
    2^(E-15) * (1 + M/64)     if E == 31 and M != 0
```

When a floating-point value is converted to an unsigned 11-bit floating-point representation, finite values are rounded to the closet representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 65024 (the maximum finite representable unsigned 11-bit floating-point value) are converted to 65024. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 11-bit floating-point value is legal as input to a GL command that accepts 11-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

2.1.B Unsigned 10-Bit Floating-Point Numbers

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 5-bit mantissa (M). The value of an unsigned 10-bit floating-point number (represented as an 10-bit unsigned integer N) is determined by the following:

```
    0.0,                    if E == 0 and M == 0,
    2^-14 * (M / 32),       if E == 0 and M != 0,
    2^(E-15) * (1 + M/32),  if 0 < E < 31,
    INF,                    if E == 31 and M == 0, or
    NaN,                    if E == 31 and M != 0,
```

where

```
    E = floor(N / 32), and
    M = N mod 32.
```

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closet representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 10-bit floating-point value is legal as input to a GL command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

Additions to Chapter 3 of the 2.0 Specification (Rasterization)

-- Section 3.6.4, Rasterization of Pixel Rectangles

Add a new row to Table 3.5 (page 128):

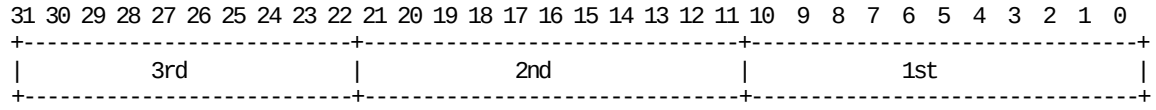
type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_INT_10F_11F_11F_REV_EXT	uint	yes

Add a new row to table 3.8: Packed pixel formats (page 132):

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_INT_10F_11F_11F_REV_EXT	uint	3	RGB

Add a new entry to table 3.11: UNSIGNED_INT formats (page 134):

UNSIGNED_INT_10F_11F_11F_REV_EXT:



Add to the end of the 2nd paragraph starting "Pixels are draw using":

"If type is UNSIGNED_INT_10F_11F_11F_REV_EXT and format is not RGB then the error INVALID_ENUM occurs."

Add UNSIGNED_INT_10F_11F_11F_REV_EXT to the list of packed formats in the 10th paragraph after the "Packing" subsection (page 130).

Add before the 3rd paragraph (page 135, starting "Calling DrawPixels with a type of BITMAP...") from the end of the "Packing" subsection:

"Calling DrawPixels with a type of UNSIGNED_INT_10F_11F_11F_REV_EXT and format of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 3 packed components as shown in table 3.11. The 1st, 2nd, and 3rd components are called f_red (11 bits), f_green (11 bits), and f_blue (10 bits) respectively.

f_red and f_green are treated as unsigned 11-bit floating-point values and converted to floating-point red and green components respectively as described in section 2.1.A. f_blue is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.1.B."

-- Section 3.8.1, Texture Image Specification:

"Alternatively if the internalformat is R11F_G11F_B10F_EXT, the red, green, and blue bits are converted to unsigned 11-bit, unsigned 11-bit, and unsigned 10-bit floating-point values as described in sections 2.1.A and 2.1.B. These encoded values can be later decoded back to floating-point values due to texture image sampling or querying."

Add a new row to Table 3.16 (page 154).

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	D bits
R11F_G11F_B10F_EXT	RGB	11	11	10				

Additions to Chapter 4 of the 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

-- Modify Chapter 4 Introduction, (page 198)

Modify first sentence of third paragraph (page 198):

"Color buffers consist of either signed or unsigned integer color indices, R, G, B and optionally A signed or unsigned integer values, or R, G, B, and optionally A signed or unsigned floating-point values."

-- Section 4.3.2, Reading Pixels

Add a row to table 4.7 (page 224);

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_INT_10F_11F_11F_REV_EXT	uint	special

Replace second paragraph of "Final Conversion" (page 222) to read:

For an RGBA color, if <type> is not one of FLOAT, UNSIGNED_INT_5_9_9_9_REV_EXT, or UNSIGNED_INT_10F_11F_11F_REV_EXT, or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color (or texture) buffer is a fixed-point buffer, each component is first clamped to [0,1]. Then the appropriate conversion formula from table 4.7 is applied the component."

Add a paragraph after the second paragraph of "Final Conversion" (page 222):

"In the special case when calling ReadPixels with a type of UNSIGNED_INT_10F_11F_11F_REV_EXT and format of RGB, the conversion is done as follows: The returned data are packed into a series of GL uint values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating point as described in section 2.1.A and 2.1.B. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the UNSIGNED_INT_10F_11F_11F_REV_EXT format as shown in table 3.11."

Additions to Chapter 5 of the 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 2.0 Specification (State and State Requests)

None

Additions to the OpenGL Shading Language specification

None

Additions to Chapter 3 of the GLX 1.3 Specification (Functions and Errors)

Replace Section 3.3.3 (p.12) Paragraph 4 to:

The attribute GLX_RENDER_TYPE has as its value a mask indicating what type of GLXContext a drawable created with the corresponding GLXFBConfig can be bound to. The following bit settings are supported: GLX_RGBA_BIT, GLX_RGBA_FLOAT_BIT, GLX_RGBA_UNSIGNED_FLOAT_BIT, GLX_COLOR_INDEX_BIT. If combinations of bits are set in the mask then drawables created with the GLXFBConfig can be bound to those corresponding types of rendering contexts.

Add to Section 3.3.3 (p.15) after first paragraph:

Note that unsigned floating point rendering is only supported for GLXPbuffer drawables. The GLX_DRAWABLE_TYPE attribute of the GLXFBConfig must have the GLX_PBUFFER_BIT bit set and the GLX_RENDER_TYPE attribute must have the GLX_RGBA_UNSIGNED_FLOAT_BIT set. Unsigned floating point rendering assumes the framebuffer format has no sign bits so all component values are non-negative. In contrast, conventional floating point rendering assumes signed components.

Modify Section 3.3.7 (p.25 Rendering Contexts) remove period at end of second paragraph and replace with:

; if render_type is set to GLX_RGBA_UNSIGNED_FLOAT_TYPE then a context that supports unsigned floating point RGBA rendering is created.

GLX Protocol

None.

Additions to the WGL Specification

Modify the values accepted by WGL_PIXEL_TYPE_ARB to:

> WGL_PIXEL_TYPE_ARB
> The type of pixel data. This can be set to WGL_TYPE_RGBA_ARB, WGL_TYPE_RGBA_FLOAT_ARB, WGL_TYPE_RGBA_UNSIGNED_FLOAT_EXT, or WGL_TYPE_COLORINDEX_ARB.

Add this explanation of unsigned floating point rendering:

"Unsigned floating point rendering assumes the framebuffer format has no sign bits so all component values are non-negative. In contrast, conventional floating point rendering assumes signed components."

Dependencies on WGL_ARB_pbuffer

Ignore the "Additions to the WGL Specification" section if WGL_ARB_pbuffer is not supported.

Dependencies on WGL_ARB_pixel_format

The WGL_ARB_pixel_format extension must be used to determine a pixel format with unsigned float components.

Dependencies on ARB_color_buffer_float

If ARB_color_buffer_float is not supported, replace this amended sentence from 4.3.2 above

For an RGBA color, if <type> is not one of FLOAT, UNSIGNED_INT_5_9_9_9_REV_EXT, or UNSIGNED_INT_10F_11F_11F_REV_EXT, or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color (or texture) buffer is a fixed-point buffer, each component is first clamped to [0,1]."

with

"For an RGBA color, if <type> is not one of FLOAT, UNSIGNED_INT_5_9_9_9_REV_EXT, or UNSIGNED_INT_10F_11F_11F_REV_EXT and the selected color buffer (or texture image for GetTexImage) is a fixed-point buffer (or texture image for GetTexImage), each component is first clamped to [0,1]."

Dependencies on EXT_texture_shared_exponent

If EXT_texture_shared_exponent is not supported, delete the reference to UNSIGNED_INT_5_9_9_9_REV_EXT in section 4.3.2.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object is not supported, then RenderbufferStorageEXT is not supported and the R11F_G11F_B10F_EXT internalformat is therefore not supported by RenderbufferStorageEXT.

If EXT_framebuffer_object is supported, glRenderbufferStorageEXT accepts GL_RG11F_B10F_EXT for its internalformat parameter because GL_RG11F_B10F_EXT has a base internal format of GL_RGB that is listed as color-renderable by the EXT_framebuffer_object specification.

Errors

Relaxation of INVALID_ENUM errors

TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D, and RenderbufferStorageEXT accept the new R11F_G11F_B10F_EXT token for internalformat.

DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable accept the new UNSIGNED_INT_10F_11F_11F_REV_EXT token for type.

New errors

INVALID_OPERATION is generated by DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable if <type> is UNSIGNED_INT_10F_11F_11F_REV_EXT and <format> is not RGB.

New State

In table 6.17, Textures (page 278), increment the 42 in "n x Z42*" by 1 for the R11F_G11F_B10F_EXT format.

[NOTE: The OpenGL 2.0 specification actually should read "n x Z48*" because of the 6 generic compressed internal formats in table 3.18.]

(modify table 6.33, p. 294)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RGBA_SIGNED_COMPONENTS_EXT	4xB	GetIntegerv	-	True if respective R, G, B, and A components are signed	4	-

New Implementation Dependent State

None

Issues

1) What should this extension be called?

RESOLVED: EXT_packed_float

This extension provides a new 3-component packed float format for use as a texture internal format, pixel external format, and framebuffer color format.

"packed" indicates the extension is packing components at reduced precisions (similar to EXT_packed_pixels or NV_packed_depth_stencil).

EXT_r11f_g11f_b10f_float was considered but there's no precedent for extension names to be so explicit (or cryptic?) about format specifics in the extension name.

2) Should there be an rgb11f_b10f framebuffer format?

RESOLVED: Yes. Unsigned floating-point rendering formats for GLX and WGL are provided. The assumption is that this functionality could be used to advertise a pixel format with 11 bits of unsigned

floating-point red, 11 bits of unsigned floating-point green, and 10 bits of floating-point blue.

In theory, an implementation could advertise other component sizes other than 11/11/10 for an unsigned floating-point framebuffer format but that is not expected.

3) *Should there be GLX and WGL extension strings?*

RESOLVED: Yes, there are WGL and GLX tokens added to support querying unsigned floating-point color buffer formats named WGL_EXT_pixel_format_packed_float and GLX_EXT_fbconfig_packed_float respectively.

4) *Should there be an unequal distribution of red, green, and blue mantissa bits?*

RESOLVED: Yes. A 6-bit mantissa for red and green is unbalanced with the 5-bit mantissa for blue, but this allows all the bits of a 32 bit word (6+6+5+3*5=32) to be used. The blue component is chosen to have fewer bits because 1) it is the third component, and 2) there's a belief that the human eye is less sensitive to blue variations..

Developers should be aware that subtle yellowing or bluing of gray-scale values is possible because of the extra bit of mantissa in the red and green components.

5) *Should there be an external format for r11f_g11f_b10f?*

RESOLVED: Yes. This makes it fast to load GL_R11F_G11F_B10F_EXT textures without any translation by the driver.

6) *What is the exponent bias?*

RESOLVED: 15, just like 16-bit half-precision floating-point values.

7) *Can s10e5 floating-point filtering be used to filter r11f_g11f_b10f values? If so, how?*

RESOLVED: Yes. It is easy to promote r11f_g11f_b10f values to s10e5 components.

8) *Should automatic mipmap generation be supported for r11f_g11f_b10f textures?*

RESOLVED: Yes.

9) *Should non-texture and non-framebuffer commands for loading pixel data accept the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT type?*

RESOLVED: Yes.

Once the pixel path has to support the new type/format combination of GL_UNSIGNED_INT_5_9_9_9_REV_EXT / GL_RGB for specifying and querying texture images, it might as well be supported for all

commands that pack and unpack RGB pixel data.

The specification is written such that the glDrawPixels type/format parameters are accepted by glReadPixels, glTexGetImage, glTexImage2D, and other commands that are specified in terms of glDrawPixels.

10) *Should non-texture internal formats (such as for color tables, convolution kernels, histogram bins, and min/max tables) accept GL_R11F_G11F_B10F_EXT format?*

RESOLVED: No.

That's pointless. No hardware is ever likely to support GL_R11F_G11F_B10F_EXT internal formats for anything other than textures and maybe color buffers in the future. This format is not interesting for color tables, convolution kernels, etc.

11) *Should a format be supported with sign bits for each component?*

RESOLVED: No. A sign bit for each of the three components would steal too many bits from the mantissa. This format is intended for storing radiance and irradiance values that are physically non-negative.

12) *Should we support a non-REV version of the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT token?*

RESOLVED: No. We don't want to promote different arrangements of the bitfields for r11f_g11f_b10f values.

13) *Can you use the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT format with just any format?*

RESOLVED: You can only use the GL_UNSIGNED_INT_10F_11F_11F_REV_EXT format with GL_RGB. Otherwise, the GL generates an GL_INVALID_OPERATION error. Just as the GL_UNSIGNED_BYTE_3_3_2 format just works with GL_RGB (or else the GL generates an GL_INVALID_OPERATION error), so should GL_UNSIGNED_INT_10F_11F_11F_REV_EXT.

14) *Should blending be supported for a packed float framebuffer format?*

RESOLVED: Yes. Blending is required for other floating-point framebuffer formats introduced by ARB_color_buffer_float. The equations for blending should be evaluated with signed floating-point math but the result will have to be clamped to non-negative values to be stored back into the packed float format of the color buffer.

15) *Should unsigned floating-point framebuffers be queried differently from conventional (signed) floating-point framebuffers?*

RESOLVED: Yes. An existing application using ARB_color_buffer_float can rightfully expect a floating-point

color buffer format to provide signed components. The packed float format does not provide a sign bit. Simply treating packed float color buffer formats as floating-point might break some existing applications that depend on a float color buffer to be signed.

For this reason, there are new WGL_TYPE_RGBA_UNSIGNED_FLOAT_EXT (for WGL) and GLX_RGBA_UNSIGNED_FLOAT_BIT_EXT (for GLX) framebuffer format parameters.

16) *What should glGet of GL_RGBA_FLOAT_MODE_ARB return for unsigned float color buffer formats?*

RESOLVED. GL_RGBA_FLOAT_MODE_ARB should return true. The packed float components are unsigned but still floating-point.

17) *Can you query with glGet to determine if the color buffer has unsigned float components?*

RESOLVED: Yes. Call glGetIntegerv on GL_RGBA_SIGNED_COMPONENTS_EXT. The value returned is a 4-element array. Element 0 corresponds to red, element 1 corresponds to green, element 2 corresponds to blue, and element 3 corresponds to alpha. If a color component is signed, its corresponding element is true (GL_TRUE). This is the same way the GL_COLOR_WRITEMASK bits are formatted.

For the packed float format, all the elements are zeroed since the red, green, and blue components are unsigned and the alpha component is non-existent. All elements are also zeroed for conventional fixed-point color buffer formats. Elements are set for signed floating-point formats such as those introduced by ARB_color_buffer_float. If a component (such as alpha) has zero bits, the component should not be considered signed and so the bit for the respective component should be zeroed.

This generality allows a future extension to specify float color buffer formats that had a mixture of signed and unsigned floating-point components. However, this extension only provides a packed float color format with all unsigned components.

18) *How many bits of alpha should GL_ALPHA_BITS return for the packed float color buffer format?*

RESOLVED: Zero.

19) *Can you render to a packed float texture with the EXT_framebuffer_object functionality?*

RESOLVED: Yes.

Potentially an implementation could return GL_FRAMEBUFFER_UNSUPPORTED_EXT when glCheckFramebufferStatusEXT for a framebuffer object including a packed float color buffer, but implementations are likely to support (and strongly encouraged to support) the packed float format for use with a framebuffer object because the packed float format is expected to be a

memory-efficient floating-point color format well-suited for rendering, particularly rendering involving high-dynamic range.

20) This extension is for a particular packed float format. What if new packed float formats come along?

RESOLVED: A new extension could be introduced with a name like EXT_packed_float2, but at this time, no other such extensions are expected except for the EXT_texture_shared_exponent extension. It simply hard to justify packing three or more components into a single 32-bit word in lots of different ways since any approach is going to be a compromise of some sort. For two-component or one-component floating-point formats, the existing ARB_texture_float formats fit nicely into 16 or 32 bits by simply using half precision floating-point. If 64 bits are allowed for a pixel, the GL_RGBA16F_ARB is a good choice.

The packed float format is similar to the format introduced by the EXT_texture_shared_exponent extension, but that extension is not a pure packed float format. Unlike the packed float format, the EXT_texture_shared_exponent format shares a single exponent between the RGB components rather than providing an independent exponent for each component. Because the EXT_texture_shared_exponent uses fewer bits to store exponent values, more mantissa precision is provided.

21) Should this extension provide pbuffer support?

RESOLVED: Yes. Pbuffers are core GLX 1.3 functionality. While using FBO is probably the preferred way to render to r11f_g11f_b10f framebuffers but pbuffer support is natural to provide. WGL should have r11f_g11f_b10f pbuffer support too.

22) Must an implementation support NaN, Infinity, and/or denorms?

RESOLVED: The preferred implementation is to support NaN, Infinity, and denorms. Implementations are allowed to flush denorms to zero, and treat NaN and Infinity values as large finite values.

This allowance flushes denorms to zero:

```
0.0,                    if E == 0 and M != 0
```

This allowance treats Infinity as a finite value:

```
2^16                    if E == 31 and M == 0
```

This allowance treats NaN encodings as finite values:

```
2^16 * (1 + M/64)       if E == 31 and M != 0
```

The expectation is that mainstream GPUs will support NaN, Infinity, and denorms while low-end implementations such as for OpenGL ES 2.0 will likely support denorms but neither NaN nor Infinity.

There is not an indication of how these floating-point special values are treated (though an application could test an implementation if necessary).

23) Should this extension interoperate with framebuffer objects?

RESOLVED: Definitely. No particular specification language is required.

In particular, glRenderbufferStorageEXT should accept GL_R11F_G11F_B10F_EXT for its internalformat parameter (true because this extension adds a new format to Table 3.16).

24) Are negative color components clamped to zero when written into an unsigned floating-point color buffer? If so, do we need to say in the Blending or Dithering language that negative color components are clamped to zero?

RESOLVED: Yes, negative color components are clamped to zero when written to an unsigned floating-point color buffer. No specification language is required for this behavior because the ARB_color_buffer_float extension says

"In RGBA mode dithering selects, for each color component, either the most positive representable color value (for that particular color component) that is less than or equal to the incoming color component value, c, or the most negative representable color value that is greater than or equal to c.

If dithering is disabled, then each incoming color component c is replaced with the most positive representable color value (for that particular component) that is less than or equal to c, or by the most negative representable value, if no representable value is less than or equal to c;"

The most negative representable value for unsigned floating-point values is zero. So the existing language from ARB_color_buffer_float already indicates that negative values are clamped to zero for unsigned floating-point color buffers. No additional specification language is required.

25) Prior texture internal formats have generic formats (example: GL_RGB) and corresponding sized formats (GL_RGB8, GL_RGB10, etc.). Should we add a generic format corresponding to GL_R11F_G11F_B10F_EXT?

RESOLVED: No. It's unlikely there will be any other unsigned floating-point texture formats.

Revision History

None

Name

EXT_texture_array

Name Strings

GL_EXT_texture_array

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006, Release 95)

Version

Last Modified Date: 02/04/2008
Author revision: 6

Number

329

Dependencies

This extension is written against the OpenGL 2.0 specification and version 1.10.59 of the OpenGL Shading Language specification.

This extension is interacts with EXT_framebuffer_object.

This extension interacts with NV_geometry_program4.

This extension interacts with NV_gpu_program4 or the OpenGL Shading Language, which provide the mechanisms necessary to access array textures.

This extension interacts with EXT_texture_compression_s3tc and NV_texture_compression_vtc.

Overview

This extension introduces the notion of one- and two-dimensional array textures. An array texture is a collection of one- and two-dimensional images of identical size and format, arranged in layers. A one-dimensional array texture is specified using TexImage2D; a two-dimensional array texture is specified using TexImage3D. The height (1D array) or depth (2D array) specify the number of layers in the image.

An array texture is accessed as a single unit in a programmable shader, using a single coordinate vector. A single layer is selected, and that layer is then accessed as though it were a one- or two-dimensional texture. The layer used is specified using the "t" or "r" texture coordinate for 1D and 2D array textures, respectively. The layer coordinate is provided as an unnormalized floating-point value in the range [0,<n>-1], where <n> is the number of layers in the array texture. Texture lookups do not filter between layers, though such filtering can be

achieved using programmable shaders. When mipmapping is used, each level of an array texture has the same number of layers as the base level; the number of layers is not reduced as the image size decreases.

Array textures can be rendered to by binding them to a framebuffer object (EXT_framebuffer_object). A single layer of an array texture can be bound using normal framebuffer object mechanisms, or an entire array texture can be bound and rendered to using the layered rendering mechanisms provided by NV_geometry_program4.

This extension does not provide for the use of array textures with fixed-function fragment processing. Such support could be added by providing an additional extension allowing applications to pass the new target enumerants (TEXTURE_1D_ARRAY_EXT and TEXTURE_2D_ARRAY_EXT) to Enable and Disable.

New Procedures and Functions

```
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
```

New Tokens

Accepted by the <target> parameter of TexParameteri, TexParameteriv, TexParameterf, TexParameterfv, and BindTexture:

TEXTURE_1D_ARRAY_EXT	0x8C18
TEXTURE_2D_ARRAY_EXT	0x8C1A

Accepted by the <target> parameter of TexImage3D, TexSubImage3D, CopyTexSubImage3D, CompressedTexImage3D, and CompressedTexSubImage3D:

TEXTURE_2D_ARRAY_EXT	
PROXY_TEXTURE_2D_ARRAY_EXT	0x8C1B

Accepted by the <target> parameter of TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, CompressedTexImage2D, and CompressedTexSubImage2D:

TEXTURE_1D_ARRAY_EXT	
PROXY_TEXTURE_1D_ARRAY_EXT	0x8C19

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

TEXTURE_BINDING_1D_ARRAY_EXT	0x8C1C
TEXTURE_BINDING_2D_ARRAY_EXT	0x8C1D
MAX_ARRAY_TEXTURE_LAYERS_EXT	0x88FF

```
    Accepted by the <param> parameter of TexParameterf, TexParameteri,
    TexParameterfv, and TexParameteriv when the <pname> parameter is
    TEXTURE_COMPARE_MODE_ARB:

        COMPARE_REF_DEPTH_TO_TEXTURE_EXT                0x884E

    (Note:  COMPARE_REF_DEPTH_TO_TEXTURE_EXT is simply an alias for the
    existing COMPARE_R_TO_TEXTURE token in OpenGL 2.0; the alternate name
    reflects the fact that the R coordinate is not always used.)

    Accepted by the <internalformat> parameter of TexImage3D and
    CompressedTexImage3D, and by the <format> parameter of
    CompressedTexSubImage3D:

        COMPRESSED_RGB_S3TC_DXT1_EXT
        COMPRESSED_RGBA_S3TC_DXT1_EXT
        COMPRESSED_RGBA_S3TC_DXT3_EXT
        COMPRESSED_RGBA_S3TC_DXT5_EXT

    Accepted by the <pname> parameter of
    GetFramebufferAttachmentParameterivEXT:

        FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT          0x8CD4

    (Note:  FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is simply an alias for the
    FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT token provided in
    EXT_framebuffer_object.  This extension generalizes the notion of
    "<zoffset>" to include layers of an array texture.)

    Returned by the <type> parameter of GetActiveUniform:

        SAMPLER_1D_ARRAY_EXT                            0x8DC0
        SAMPLER_2D_ARRAY_EXT                            0x8DC1
        SAMPLER_1D_ARRAY_SHADOW_EXT                     0x8DC3
        SAMPLER_2D_ARRAY_SHADOW_EXT                     0x8DC4
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify section 2.15.3, "Shader Variables", page 75

Add the following new return types to the description of GetActiveUniform on p. 81.

```
  SAMPLER_1D_ARRAY_EXT,
  SAMPLER_2D_ARRAY_EXT,
  SAMPLER_1D_ARRAY_SHADOW_EXT,
  SAMPLER_2D_ARRAY_SHADOW_EXT
```

Modify Section 2.15.4, Shader Execution (p. 84)

(modify first paragraph, p. 86 -- two simple edits:

(1) Change reference to the "r" coordinate to simply indicate that the reference value for shadow mapping is provided in the lookup function. It's still usually in the "r" coordinate, except for two-dimensional array textures, where it's in "q".

(2) Add new EXT_gpu_shader4 sampler types used for array textures.)

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function, as described in section 3.8.14. The comparison operation is requested in the shader by using the shadow sampler types (sampler1DShadow, sampler2DShadow, sampler1DArrayShadow, or sampler2DArrayShadow) and in the texture using the TEXTURE_COMPARE_MODE parameter. ...

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.8, Texturing (p. 149).

(add new paragraph at the top of p. 150) Six types of texture are supported; each is a collection of images built from one-, two-, or three-dimensional array of image elements referred to as texels. One-, two-, and three-dimensional textures consist of a one-, two-, or three-dimensional texel arrays. One- and two-dimensional array textures are arrays of one- or two-dimensional images, consisting of one or more layers. Finally, a cube map is a special two-dimensional array texture with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces.

Modify Section 3.8.1, Texture Image Specification (p. 150).

(modify first paragraph of section, p. 150) The command

```
void TexImage3D( enum target, int level, int internalformat,
                 sizei width, sizei height, sizei depth, int border,
                 enum format, enum type, void *data );
```

is used to specify a three-dimensional texture image. target must be one of TEXTURE_3D for a three-dimensional texture or TEXTURE_2D_ARRAY_EXT for an two-dimensional array texture. Additionally, target may be either PROXY_TEXTURE_3D for a three-dimensional proxy texture, or PROXY_TEXTURE_2D_ARRAY_EXT for a two-dimensional proxy array texture. ...

(modify the fourth paragraph on p. 151) Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if target is TEXTURE_1D, TEXTURE_2D, TEXTURE_1D_ARRAY_EXT, TEXTURE_2D_ARRAY_EXT, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY_EXT, or PROXY_TEXTURE_2D_ARRAY_EXT. Using this format in conjunction with any other target will result in an INVALID OPERATION error.

(modify the first paragraph on p. 153 -- In particular, add new terms w_b, h_b, and d_b to represent border width, height, or depth, instead of a single border size term b_s. Subsequent equations referring to b_s should be modified to refer to w_b, h_b, and d_b, as appropriate.)

```
... Counting from zero, each resulting Nth texel is assigned internal
integer coordinates (i, j, k), where

  i = (N mod width) - w_b
  j = (floor(N/width) mod height) - h_b
  k = (floor(N/(width*height)) mod depth) - d_b

and w_b, h_b, and d_b are the specified border width, height, and depth.
w_b and h_b are the specified <border> value; d_b is the specified
<border> value if <target> is TEXTURE_3D or zero if <target> is
TEXTURE_2D_ARRAY_EXT. ...

(modify equations 3.15-3.17 and third paragraph of p. 155)

  w_s = w_t + 2 * w_b                    (3.15)
  h_s = h_t + 2 * h_b                    (3.16)
  d_s = d_t + 2 * d_b                    (3.17)

... If <border> is less than zero, or greater than b_t, then the error
INVALID_VALUE is generated.

(modify the last paragraph on p. 155 on to p. 156)

The maximum allowable width, height, or depth of a texel array for a
three-dimensional texture is an implementation dependent function of the
level-of-detail and internal format of the resulting image array.  It must
be at least 2^(k-lod) + 2 * b_t for image arrays of level-of-detail 0
through k, where k is the log base 2 of MAX_3D_TEXTURE_SIZE, lod is the
level-of-detail of the image array, and b_t is the maximum border width.
It may be zero for image arrays of any level-of-detail greater than k. The
error INVALID VALUE is generated if the specified image is too large to be
stored under any conditions.

In a similar fashion, the maximum allowable width of a texel array for a
one- or two-dimensional, or one- or two-dimensional array texture, and the
maximum allowable height of a two-dimensional or two-dimensional array
texture, must be at least 2^(k-lod) + 2 * b_t for image arrays of level 0
through k, where k is the log base 2 of MAX_TEXTURE_SIZE. The maximum
allowable width and height of a cube map texture must be the same, and
must be at least 2^(k-lod) + 2 * b_t for image arrays level 0 through k,
where k is the log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE.  The maximum
number of layers for one- and two-dimensional array textures (height or
depth, respectively) must be at least MAX_ARRAY_TEXTURE_LAYERS_EXT for all
levels.

(modify the fourth paragraph on p. 156) The command

  void TexImage2D( enum target, int level,
                   int internalformat, sizei width, sizei height,
                   int border, enum format, enum type, void *data );

is used to specify a two-dimensional texture image. target must be one of
TEXTURE_2D for a two-dimensional texture, TEXTURE_1D_ARRAY_EXT for a
one-dimensional array texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X,
TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y,
TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or
TEXTURE_CUBE_MAP_NEGATIVE_Z for a cube map texture. Additionally, target
```

may be either PROXY_TEXTURE_2D for a two-dimensional proxy texture, PROXY_TEXTURE_1D_ARRAY_EXT for a one-dimensional proxy array texture, or PROXY TEXTURE_CUBE_MAP for a cube map proxy texture in the special case discussed in section 3.8.11. The other parameters match the corresponding parameters of TexImage3D.

For the purposes of decoding the texture image, TexImage2D is equivalent to calling TexImage3D with corresponding arguments and depth of 1, except that

* The border depth, d_b, is zero, and the depth of the image is always 1 regardless of the value of border.

* The border height, h_b, is zero if <target> is TEXTURE_1D_ARRAY_EXT, and <border> otherwise.

* Convolution will be performed on the image (possibly changing its width and height) if SEPARABLE 2D or CONVOLUTION 2D is enabled.

* UNPACK SKIP IMAGES is ignored.

(modify the fourth paragraph on p. 157) For the purposes of decoding the texture image, TexImage1D is equivalent to calling TexImage2D with corresponding arguments and height of 1, except that

* The border height and depth (h_b and d_b) are always zero, regardless of the value of <border>.

* Convolution will be performed on the image (possibly changing its width) only if CONVOLUTION 1D is enabled.

(modify the last paragraph on p. 157 and the first paragraph of p. 158 -- changing the phrase "texture array" to "texel array" to avoid confusion with array textures. All subsequent references to "texture array" in the specification should also be changed to "texel array".)

We shall refer to the (possibly border augmented) decoded image as the texel array. A three-dimensional texel array has width, height, and depth ws, hs, and ds as defined respectively in equations 3.15, 3.16, and 3.17. A two-dimensional texel array has depth ds = 1, with height hs and width ws as above, and a one-dimensional texel array has depth ds = 1, height hs = 1, and width ws as above.

An element (i,j,k) of the texel array is called a texel (for a two-dimensional texture or one-dimensional array texture, k is irrelevant; for a one-dimensional texture, j and k are both irrelevant). The texture value used in texturing a fragment is determined by that fragment’s associated (s,t,r) coordinates, but may not correspond to any actual texel. See figure 3.10.

Modify Section 3.8.2, Alternate Texture Image Specification Commands (p. 159)

(modify second paragraph, p. 159 -- allow 1D array textures) The command

```
  void CopyTexImage2D( enum target, int level,
                       enum internalformat, int x, int y, sizei width,
                       sizei height, int border );
```

defines a two-dimensional texture image in exactly the manner of TexImage2D, except that the image data are taken from the framebuffer rather than from client memory. Currently, target must be one of TEXTURE_2D, TEXTURE_1D_ARRAY_EXT, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z.

(modify last paragraph, p. 160) ... 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_1D_ARRAY_EXT, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, and the target arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D or TEXTURE_2D_ARRAY_EXT. ...

(modify last paragraph, p. 161 and subsequent inequalities)

Negative values of xoffset, yoffset, and zoffset correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s, h_s, d_s, w_b, h_b, and d_b to be the specified width, height, depth, and border width, height, and depth of the texture array, and taking x, y, z, w, h, and d to be the xoffset, yoffset, zoffset, width, height, and depth argument values, any of the following relationships generates the error INVALID VALUE:

```
    x < -w_b
    x + w > w_s - w_b
    y < -h_b
    y + h > h_s - h_b
    z < -d_b
    z + d > d_s - d_b
```

Modify Section 3.8.4, Texture Parameters (p. 166)

(modify first paragraph of section, p. 166) Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

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

target is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT.

Modify Section 3.8.8, Texture Minification (p. 170)

(modify first paragraph, p. 172) ... For a one-dimensional or one-dimensional array texture, define v(x, y) == 0 and w(x, y) == 0; for a two-dimensional, two-dimensional array, or cube map texture, define w(x, y) == 0. ...

(modify second paragraph, p. 173) For one-dimensional or one-dimensional array textures, j and k are irrelevant; the texel at location i becomes the texture value. For two-dimensional, two-dimensional array, or cube map textures, k is irrelevant; the texel at location (i, j) becomes the texture value. For one- and two-dimensional array textures, the texel is obtained from image layer l, where

```
  l = clamp(floor(t + 0.5), 0, h_t-1), for one-dimensional array textures,
      clamp(floor(r + 0.5), 0, d_t-1), for two-dimensional array textures.
```

(modify third paragraph, p. 174) For a two-dimensional, two-dimensional array, or cube map texture,

```
  tau = ...
```

where tau_ij is the texel at location (i, j) in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer l, where

```
  l = clamp(floor(r + 0.5), 0, d_t-1).
```

And for a one-dimensional or one-dimensional array texture,

```
  tau = ...
```

where tau_i is the texel at location i in the one-dimensional texture. For one-dimensional array textures, both texels are obtained from layer l, where

```
  l = clamp(floor(t + 0.5), 0, h_t-1).
```

(modify first two paragraphs of "Mipmapping", p. 175) 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 mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one.

If the image array of level level_base, excluding its border, has dimensions, w_t × h_t × d_t, then there are floor(log2(maxsize)) + 1 levels in the mipmap, where

```
  maxsize = w_t,                 for one-dimensional and one-dimensional
                                 array textures,
            max(w_t, h_t),       for two-dimensional, two-dimensional
                                 array, and cube map textures
            max(w_t, h_t, d_t),  for three dimensional textures.
```

Numbering the levels such that level level_base is the 0th level, the ith array has dimensions

max(1, floor(w_t/w_d)) x max(1, floor(h_t/h_d)) x max(1, floor(d_t/d_d))

where

w_d = 2 ^ i;
h_d = 1, for one-dimensional array textures and
2 ^ i, otherwise; and
d_d = 1, for two-dimensional array textures and
2 ^ i, otherwise,

until the last array is reached with dimension 1 × 1 × 1.

Each array in a mipmap is defined using TexImage3D, TexImage2D, CopyTexImage2D, TexImage1D, or CopyTexImage1D; the array being set is indicated with the level-of-detail argument level. Level-of-detail numbers proceed from level_base for the original texture array through p = floor(log2(maxsize)) + level_base with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. All arrays from level_base through q = min{p, level_max} must be defined, as discussed in section 3.8.10.

(modify third paragraph in the "Mipmap Generation" section, p. 176)

The contents of the derived arrays are computed by repeated, filtered reduction of the level_base array. For one- and two-dimensional array textures, each layer is filtered independently. ...

Modify Section 3.8.10, Texture Completeness (p. 177)

(modify second paragaph of section, p. 177) For one-, two-, or three-dimensional textures and one- or two-dimensional array textures, a texture is complete if the following conditions all hold true: ...

Modify Section 3.8.11, Texture State and Proxy State (p. 178)

(modify second and third paragraphs, p. 179, adding array textures and making minor wording changes)

In addition to image arrays for one-, two-, and three-dimensional textures, one- and two-dimensional array textures, and the six image arrays for the cube map texture, partially instantiated image arrays are maintained for one-, two-, and three-dimensional textures and one- and two-dimensional array textures. Additionally, a single proxy image array is maintained for the cube map texture. Each proxy image array includes width, height, depth, border width, and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy image arrays do not include image data, nor do they include texture properties. When TexImage3D is executed with target specified as PROXY_TEXTURE_3D, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by TexImage3D called with target set to TEXTURE 3D, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero. If the image

array would be supported by such a call to TexImage3D, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

Proxy arrays for one- and two-dimensional textures and one- and two-dimensional array textures are operated on in the same way when TexImage1D is executed with target specified as PROXY_TEXTURE_1D, TexImage2D is executed with target specified as PROXY_TEXTURE_2D or PROXY_TEXTURE_1D_ARRAY_EXT, or TexImage3D is executed with target specified as PROXY_TETXURE_2D_ARRAY_EXT.

Modify Section 3.8.12, Texture Objects (p. 180)

(update most of the beginning of the section to allow array textures)

In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_EXT, named one-, two-, and three-dimensional, cube map, and one- and two-dimensional array texture objects can be created and operated upon. 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, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT. The binding is effected by calling

  void BindTexture(enum target, uint texture);

with <target> set to the desired texture target and <texture> set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.11, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT, it is and remains a one-, two-, three-dimensional, cube map, one- or two-dimensional array texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified target. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to target is broken.

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_ARRAY_EXT have one-, two-, three-dimensional, cube map, and one- and two-dimensional array texture state vectors respectively associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-,

three-dimensional, cube map, one- and two-dimensional array textures are therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_ARRAY_EXT respectively while 0 is bound to the corresponding targets.

(modify second paragraph, p. 181) ... If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT is deleted, it is as though BindTexture had been executed with the same target and texture zero. ...

(modify second paragraph, p. 182) The texture object name space, including the initial one-, two-, and three dimensional, cube map, and one- and two-dimensional array texture objects, is shared among all texture units. ...

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify second through fourth paragraphs, p. 188, reflecting that the texture coordinate used for depth comparisons varies, including a new enum name)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture coordinate, clamped to the range [0, 1]. For texture lookups generated by a program instruction, let R be the reference value for depth comparisons provided in the instruction, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows: ...

If the value of TEXTURE_COMPARE_MODE is NONE, then

  r = Dt

If the value of TEXTURE_COMPARE_MODE is COMPARE_REF_DEPTH_TO_TEXTURE_EXT), then r depends on the texture comparison function as shown in table 3.27.

Modify Section 3.11.2, Shader Execution (p. 194)

(modify second paragraph, p. 195 -- two simple edits:

  (1) Change reference to the "r" coordinate to simply indicate that the reference value for shadow mapping is provided in the lookup function. It's still usually in the "r" coordinate, except for two-dimensional array textures, where it's in "q".
  (2) Add new EXT_gpu_shader4 sampler types used for array textures.)

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function. The comparison operation is requested in the shader by using the shadow sampler types (sampler1DShadow, sampler2DShadow, sampler1DArrayShadow, and sampler2DArrayShadow) and in the texture using the TEXTURE COMPARE MODE parameter. ...

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Modify Section 5.4, Display Lists (p. 237)

(modify first paragraph, p. 242) TexImage3D, TexImage2D, TexImage1D, Histogram, and ColorTable are executed immediately when called with the corresponding proxy arguments PROXY_TEXTURE_3D or PROXY_TEXTURE_2D_ARRAY_EXT; PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, or PROXY_TEXTURE_1D_ARRAY_EXT; PROXY_TEXTURE_1D; PROXY_HISTOGRAM; and PROXY_COLOR_TABLE, PROXY_POST_CONVOLUTION_COLOR_TABLE, or PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.3, Enumerated Queries (p. 246)

(modify second paragraph, p. 247)

GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY_EXT, or TEXTURE_2D_ARRAY_EXT, indicating the currently bound one-, two-, three-dimensional, cube map, or one- or two-dimensional array texture. GetTexLevelParameter parameter target may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z, TEXTURE_1D_ARRAY_EXT, TEXTURE_2D_ARRAY_EXT, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_CUBE_MAP, PROXY_TEXTURE_1D_ARRAY, or PROXY_TEXTURE_2D_ARRAY, indicating the one-, two-, or three-dimensional texture, one of the six distinct 2D images making up the cube map texture, the one- or two-dimensional array texture, or the one-, two-, three-dimensional, cube map, or one- or two-dimensional array proxy state vector. ...

Modify Section 6.1.4, Texture Queries (p. 248)

(modify first three paragraphs of section, p. 248) The command

```
  void GetTexImage( enum tex, int lod, enum format,
                    enum type, void *img );
```

is used to obtain texture images. 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, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY_EXT, and TEXTURE_2D_ARRAY_EXT indicate a one-, two-, or three-dimensional texture, or one- or two-dimensional array texture, respectively. TEXTURE_CUBE_MAP_POSITIVE_X, ...

GetTexImage obtains... from the first image to the last for three-dimensional textures. One- and two-dimensional array textures are

treated as two- and three-dimensional images, respectively, where the layers are treated as rows or images. These groups are then...

For three-dimensional and two-dimensional array textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT and PACK_SKIP_IMAGES are applied. ...

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object is supported, a single layer of an array texture can be bound to a framebuffer attachment point, and manual mipmap generation support is extended to include array textures.

Several modifications are made to the EXT_framebuffer_object specification. First, the token identifying the attached layer of a 3D texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT, is renamed to FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT. This is done because this extension generalizes the "z offset" concept to become notion of attaching a layer of a multi-layer texture, which is applicable for both three-dimensional and array textures. All references to this token in EXT_framebuffer_object should be changed to the new token, and references to "z offset" in the specification text should be replaced with "layer" as appropriate. Additional edits follow.

(modify "Manual Mipmap Generation" in edits to Section 3.8.8)

Mipmaps can be generated manually with the command

```
void GenerateMipmapEXT(enum target);
```

where <target> is one of TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_3D, TEXTURE_1D_ARRAY, or TEXTURE_2D_ARRAY. Mipmap generation affects the texture image attached to <target>. ...

(modify Section 4.4.2.3, Attaching Texture Images to a Framebuffer -- add to the end of the section)

The command

```
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
```

operates identically to FramebufferTexture3DEXT, except that it attaches a single layer of a three-dimensional texture or a one- or two-dimensional

array texture. <layer> is an integer indicating the layer number, and is treated identically to the <zoffset> parameter in FramebufferTexture3DEXT. The error INVALID_VALUE is generated if <layer> is negative. The error INVALID_OPERATION is generated if <texture> is non-zero and is not the name of a three dimensional texture or one- or two-dimensional array texture. Unlike FramebufferTexture3D, no <textarget> parameter is accepted.

If <texture> is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to <attachment> is updated as in the other FramebufferTexture commands, except that FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is set to <layer>.

(modify Section 4.4.4.1, Framebuffer Attachment Completeness)

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if ...:

...

* If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a one- or two-dimensional array texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the number of layers in the texture.

(modify Section 6.1.3, Enumerated Queries)

...

If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT and the texture object named FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is a three-dimensional texture or a one- or two-dimensional array texture, then <params> will contain the number of texture layer attached to the attachment point. Otherwise, <params> will contain the value zero.

Dependencies on NV_geometry_program4

NV_geometry_program4 provides additional modifications to EXT_framebuffer_object to support layered rendering, which allows applications to bind entire three-dimensional, cube map, or array textures to a single attachment point, and select a layer to render to according to a layer number written by the geometry program.

The framebuffer object modifications provided in NV_geometry_program4 are more extensive than the more limited support provided for array textures. The edits in this spec are a functional subset of the edits in NV_geometry_program4. All of the modifications that this extension makes to EXT_framebuffer_object are superseded by NV_geometry_program4, except for the minor language changes made to GenerateMipmapsEXT().

Dependencies on NV_gpu_program4 and the OpenGL Shading Language (GLSL)

If NV_gpu_program4, EXT_gpu_shader4, and the OpenGL Shading Language (GLSL) are not supported, and no other mechanism is provided to perform texture lookups into array textures, this extension is pointless, given that it provides no fixed-function mechanism to access texture arrays.

If GLSL is supported, the language below describes the modifications to the shading language to support array textures. The extension EXT_gpu_shader4 provides a broader set of shading language modifications that include array texture lookup functions described here, plus a number of additional functions.

If GLSL is not supported, the shading language below and references to the SAMPLER_{1D,2D}_ARRAY_EXT and SAMPLER_{1D,2D}_ARRAY_SHADOW_EXT tokens should be removed.

Dependencies on EXT_texture_compression_s3tc and NV_texture_compression_vtc

S3TC texture compression is supported for two-dimensional array textures. When <target> is TEXTURE_2D_ARRAY_EXT, each layer is stored independently as a compressed two-dimensional textures. When specifying or querying compressed images using one of the S3TC formats, the images are provided and/or returned as a series of two-dimensional textures stored consecutively in memory, with the layer closest to zero specified first. For array textures, images are not arranged in 4x4x4 or 4x4x2 blocks as in the three-dimensional compression format provided in the EXT_texture_compression_vtc extension. Pixel store parameters, including those specific to three-dimensional images, are ignored when compressed image data are provided or returned, as in the EXT_texture_compression_s3tc extension.

S3TC compression is not supported for one-dimensional texture targets in EXT_texture_compression_s3tc, and is not supported for one-dimensional array textures in this extension. If compressed one-dimensional arrays are needed, use a two-dimensional texture with a height of one.

As with NV_texture_compression_vtc, this extension allows the use of the four S3TC internal format types in TexImage3D, CompressedTexImage3D, and CompressedTexSubImage3D calls. Unlike NV_texture_compression_vtc (for 3D textures), compressed sub-image updates are allowed at arbitrary locations along the Z axis. The language describing CompressedTexSubImage* APIs, edited by EXT_texture_compression_s3tc (allowing updates at 4x4 boundaries for 2D textures) and NV_texture_compression_vtc (allowing updates at 4x4x4 boundaries for 3D textures) is updated as follows:

"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, 4x4x1, or 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.
      * <xoffset> or <yoffset> is not a multiple of four.
      * <depth> is not a multiple of four or equal to TEXTURE_DEPTH, and
        <target> is TEXTURE_3D.
      * <zoffset> is not a multiple of four and <target> is TEXTURE_3D."

   (Note:  The original version of this specification incorrectly failed to
    allow compressed subimage updates of array textures via
    CompressedTexSubImage3D, except at 4x4x4 boundaries/sizes.  This
    undesirable behavior was also implemented by all NVIDIA OpenGL drivers
    published prior to February 2008.)
```

Errors

```
    None.  Some error conditions are removed, due to the ability to use the
    new TEXTURE_1D_ARRAY_EXT and TEXTURE_2D_ARRAY_EXT enums.
```

New State

```
    (add to table 6.15, p. 276)
```

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BINDING_1D_ARRAY_EXT	2*xZ+	GetIntegerv	0	texture object bound to TEXTURE_1D_ARRAY	3.8.12	texture
TEXTURE_BINDING_2D_ARRAY_EXT	2*xZ+	GetIntegerv	0	texture object bound to TEXTURE_2D_ARRAY	3.8.12	texture

New Implementation Dependent State

```
    (add to Table 6.32, p. 293)
```

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TEXTURE_ARRAY_LAYERS_EXT	Z+	GetIntegerv	64	maximum number of layers for texture arrays	3.8.1	-

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

```
    (This section describes additions to GLSL to allow shaders to access array
     textures.  This is a subset of the new shading language provided by the
     EXT_gpu_shader4 extension, limited to array texture support.  It is
     provided here in case implementations choose to support EXT_texture_array
     without supporting EXT_gpu_shader4 or equivalent functionality.

     Note that if the EXT_gpu_shader4 extension is enabled in a shader via an
     "#extension" line, there is no need to separately enable
     EXT_texture_array.)

    Including the following line in a shader can be used to control the
    language features described in this extension:

      #extension GL_EXT_texture_array : <behavior>

    where <behavior> is as specified in section 3.3.
```

A new preprocessor #define is added to the OpenGL Shading Language:

```
  #define GL_EXT_texture_array 1
```

Add to section 3.6 "Keywords"

The following new sampler types are added:

```
  sampler1DArray, sampler2DArray, sampler1DArrayShadow,
  sampler2DArrayShadow
```

Add to section 4.1 "Basic Types"

Add the following entries to the type table:

```
  sampler1DArray         handle for accessing a 1D array texture
  sampler2DArray         handle for accessing a 2D array texture
  sampler1DArrayShadow   handle for accessing a 1D array depth texture
                         with comparison
  sampler2DArrayShadow   handle for accessing a 2D array depth texture
                         with comparison
```

Add to section 8.7 "Texture Lookup Functions"

Add new functions to the set of allowed texture lookup functions:

Syntax:

```
  vec4 texture1DArray(sampler1DArray sampler, vec2 coord
                      [, float bias])
  vec4 texture1DArrayLod(sampler1DArray sampler, vec2 coord,
                         float lod)
```

Description:

Use the first element (coord.s) of texture coordinate coord to do a texture lookup in the layer indicated by the second coordinate coord.t of the 1D texture array currently bound to sampler. The layer to access is computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the texture array.

Syntax:

```
  vec4 texture2DArray(sampler2DArray sampler, vec3 coord
                      [, float bias])
  vec4 texture2DArrayLod(sampler2DArray sampler, vec3 coord,
                         float lod)
```

Description:

Use the first two elements (coord.s, coord.t) of texture coordinate coord to do a texture lookup in the layer indicated by the third coordinate coord.p of the 2D texture array currently bound to sampler. The layer to access is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth of the texture array.

Syntax:

```
vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                   [float bias])
vec4 shadow1DArrayLod(sampler1DArrayShadow sampler,
                      vec3 coord, float lod)
```

Description:

Use texture coordinate coord.s to do a depth comparison lookup on an array layer of the depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The layer to access is indicated by the second coordinate coord.t and is computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the texture array. The third component of coord (coord.p) is used as the R value. The texture bound to sampler must be a depth texture, or results are undefined.

Syntax:

```
vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
```

Description:

Use texture coordinate (coord.s, coord.t) to do a depth comparison lookup on an array layer of the depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The layer to access is indicated by the third coordinate coord.p and is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth of the texture array. The fourth component of coord (coord.q) is used as the R value. The texture bound to sampler must be a depth texture, or results are undefined.

Issues

(1) Should this extension generalize the notion of 1D and 2D textures to be arrays of 1D or 2D images, or simply introduce new targets?

RESOLVED: Introduce new targets.

It would have been possible to simply extend the notion of 1D and 2D textures, and allow applications to pass TEXTURE_1D to TexImage2D (1D arrays) or TEXTURE_2D to TexImage3D (2D arrays). This would have avoided introducing a new set of texture targets (and proxy targets), and a "default texture" (object zero) for each new target.

It is desirable to have a distinction between array and non-array textures in programmable shaders, so compilers can generate code appropriate to the texture type. For "normal" textures, a 2D texture requires two component texture coordinates, while a 2D array texture requires three. Without a distinction between array and non-array textures, implementations must choose between compiling shaders to the most general form (2D arrays) or recompiling shaders based on texture usage. Texture lookups with shadow mapping, LOD bias, or per-pixel LOD have additional complexity, and the interpretation of a coordinate vector may need to depend on whether the texture was an array or non-array texture.

It would be possible to limit the distinction between array and non-array textures to the shaders, but it could then become the responsibility of the application developer to ensure that a texture with multiple layers is used when an "array lookup" is performed, and that a single-layer texture is used when a "non-array lookup" is performed. That begs the question of what the distinction between an "array texture" and a "non-array texture" is. At least two possible distinctions have been identified: one vs. multiple layers, or the API call used to specify the texture (TexImage3D with TEXTURE_2D == array texture, TexImage2D == non-array texture). The former does not allow for the possibility of single-layer array textures; it may be the case that application developers want to use a general shader supporting array textures, but there may be cases where only a single layer might be provided. The latter approach allows for single-layer array textures, but the distinction is now based on the API call.

Adding separate targets eliminates the need for such a distinction. "Array lookups" refer to the TEXTURE_1D_ARRAY_EXT or TEXTURE_2D_ARRAY_EXT targets; "non-array lookups" refer to TEXTURE_1D or TEXTURE_2D. There is never a case where the wrong kind of texture can be used, as TEXTURE_1D_ARRAY_EXT and TEXTURE_2D_ARRAY_EXT textures are always arrays by definition.

This distinction should also be helpful if and when fixed-function fragment processing is supported; the enabled texture target is used to generate an internal fragment shader using the proper "array lookup". There would be no need to recompile shaders depending on whether an enabled texture is an "array texture" or not.

(2) Should texture arrays be supported for fixed-function fragment processing?

RESOLVED: No; it's not believed to be worth the effort. Fixed-function fragment processing could be easily supported by allowing applications to enable or disable TEXTURE_1D_ARRAY_EXT or TEXTURE_2D_ARRAY_EXT.

Note that for fixed-function fragment processing, there would be issues with texture lookups of two-dimensional array textures with shadow mapping. Given that all texture lookups are projective, a total of five coordinate components would be required (s, t, layer, depth, q).

(3) If fixed-function were supported, should the layer number (T or R) be divided by Q in projective texture lookups?

RESOLVED: It doesn't need to be resolved in this extension, but it would be a problem. There are probably cases where an application would want the divide (handle R more-or-less like S/T); there are probably other cases where the divide would not be wanted. Many developers won't care, and may not even know what the Q coordinate is used for! The default of 1.0 allows applications that don't care about projective lookups to simply ignore that fact.

For programmable fragment shading, an application can code it either way and use non-projective lookups. To the extent that the divide by Q for projective lookups is "free" or "cheap" on OpenGL hardware, compilers may be able to recognize a projective pattern in the computed coordinates and generate code appropriately.

(4) Should DEPTH_COMPONENT textures be supported for texture arrays?

RESOLVED: Yes; multi-layer shadow maps are useful.

(5) How should shadow mapping in texture arrays work with programmable shaders, and fixed-function shaders (if ever supported)?

RESOLVED: The layer number is in the "next" coordinate following the normal 1D or 2D coordinate. That's the "t" coordinate for 1D arrays and the "r" coordinate for 2D arrays. For shadow maps, this is a problem, as the "r" coordinate is generally used as the depth reference value. This is resolved by instead taking the depth reference value from the "q" coordinate.

For some programmable texture lookups (explicit LOD, LOD bias, projective), "too many" coordinates are required. Such lookups are not possible with four-component vectors; it would require at least two parameters to perform such operations.

For fixed-function shading, it is recommended that shadow mapping lookups in two-dimensional array textures be treated as non-projective, even though all other lookups would be projective. Additionally, the "q" coordinate should be used for the depth reference value in this case.

(6) How do texture borders interact with array textures?

RESOLVED: Each individual layer of an array texture can have a border, as though it were a normal one- or two-dimensional texture. However, there are no "border layers".

(7) How does mipmapping work with array textures?

RESOLVED: Level <N+1> is half the size of level <N> in width and/or height, but the number of layers is always the same for each level -- layer <M> of level <N+1> is expected to be a filtered version of layer <M> of the higher mipmap levels. This behavior impacts the texture consistency rules for array textures.

(8) Are compressed textures supported for array textures?

RESOLVED: Yes; they may be loaded via normal TexImage APIs, as well as CompressedTexImage2D and CompressedTexImage3D. Compressed array textures are treated as arrays of compressed 1D or 2D images.

(9) Should these things be called "array textures" or "texture arrays"?

RESOLVED: "Array textures", mostly because it was easier spec wording. Calling them "array textures" also seems like better disambiguation; there are several different things that can be thought of as "texture arrays":

* the array of texture levels (mipmapping)
* the array of texture layers (array textures)
* the array of texels in each image

This spec changes the use of "texture array" in the core specification (which means the array of texels) to instead refer to "texel array".

(10) If they're called "array textures", why does the extension name include "texture_array"?

RESOLVED: Because this is primarily a texture extension, and all such extensions start with "texture".

(11) Should new functions be provided for loading or modifying array textures?

RESOLVED: No. Existing TexImage2D (1D arrays) and TexImage3D (2D arrays), plus corresponding TexSubImage, CopyTexImage, and CopyTexSubImage calls are sufficient.

(12) Should ARB_imaging functionality to be extended to support two-dimensional array textures?

RESOLVED: No. Convolution is rarely used when texture images are defined, and is even less likely for array teture images. This could be addressed via a separate extension if the need were identified, and such operations could be defined for 3D textures as well at that time.

Note that with the API chosen, one-dimensional array textures do have convolution applied (if enabled), because image data is treated as a normal two-dimensional image.

(13) What if an application wants to populate an array texture using separate mipmap chains a layer at a time rather than specifying all layers of a given mipmap level at once?

RESOLVED: For 2D array textures, call TexImage3D once with a NULL image pointer for each level to establish the texel array sizes. Then, call TexSubImage3D for each layer/mipmap level to define individual images.

(14) Should we provide a way to query a single layer of an array texture?

RESOLVED: No; we don't expect this to be an issue in practice. GetTexImage() will return a two- or three-dimensional image for one- and two-dimensional arrays, including all levels. If this were identified as an important need, a follow-on extension could be added in the future.

(15) How is the LOD (lambda) computed for array textures?

RESOLVED: LOD is computed in the same manner for 1D and 2D array textures as it is for normal 1D and 2D textures. The layer coordinate has no effect on LOD computations.

(16) What's the deal with this new "COMPARE_REF_DEPTH_TO_TEXTURE_EXT"?

RESOLVED: It's a new name for the existing enumerant "COMPARE_R_TO_TEXTURE". This alternate name is provided to reflect the fact that it's not always the R coordinate that is used for depth comparisons.

(17) How do array textures work with framebuffer objects (EXT_framebuffer_object extension, also known as "FBO")?

RESOLVED: A new function, FramebufferTextureLayerEXT(), is provided to attach a single layer of a one- or two-dimensional array texture to an framebuffer attachment point. That new function can also be used to attach a layer of a three-dimensional texture.

In addition to supporting FBO attachments, the manual mipmap generation support provided by glGenerateMipmapEXT is extended to array textures. Mipmap generation applies to each layer of the array texture independently, as is the case with the GENERATE_MIPMAPS texture parameter.

This support provided here a limited subset of the FBO support added by NV_geometry_program4, which additionally provides the ability to attach an entire level of a three-dimensional, cube map, or array texture. When such attachments are performed, a geometry program can be used to select a layer to render each emitted primitive to.

(18) Should array texture targets be supported for creation of "render buffers"?

RESOLVED: No. These are inherently two-dimensional images.

(19) Should we provide a mipmap generation function to generate mipmaps for only a single layer of an array texture?

RESOLVED: Not in this extension. We considered adding this toward the end of the development of this extension, but decided not to add it because this mipmap generation function would have very different requirements from the GenerateMipmapEXT function provided by EXT_framebuffer_object.

The existing GenerateMipmapEXT function replaces all levels of detail below the base level with generated mipmaps. If those mipmap levels are unpopulated or inconsistent with the base level, they are completely overwritten with a generated image that is consistent with the base level. If we were to provide a function to generate mipmaps for only a single layer, all other layers of non-base levels would need to be preserved. However, since there are not separate formats or sizes per level, this form of mipmap generation would require that all non-base levels be present and consistent with the base level, or mipmap generation wouldn't work.

We expect that future revisions of the GL will change the specification of mipmapped textures in

(20) This extension allows the use of S3TC texture internal formats in TexImage3D and CompressedTexImage3D. Does this mean that they are now supported for 3D textures?

RESOLVED: No. With this extension alone, TexImage3D and CompressedTexImage3D only support S3TC compressed formats with a target of TEXTURE_2D_ARRAY_EXT. The S3TC tokens were added to the list of internal formats supported by TexImage3D and friends because

two-dimensional array textures are specified using the three-dimensional TexImage functions.

The existing extension NV_texture_compression_vtc does provides support for S3TC-style compressed 3D textures.

Revision History

Rev.	Date	Author	Changes
6	02/04/08	pbrown	Added a missing interaction with the VTC texture compression spec allowing updates of compressed 2D array textures along 4x4x1 boundaries (we previously inherited the VTC restriction of 4x4x4).
5	12/15/06	pbrown	Documented that the '#extension' token for this extension should begin with "GL_", as apparently called for per convention.
4	--		Pre-release revisions.

Name

EXT_texture_buffer_object

Name Strings

GL_EXT_texture_buffer_object

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006, Release 95)

Version

Last Modified Date: 10/30/2007
NVIDIA Revision: 4

Number

330

Dependencies

OpenGL 2.0 is required.

NV_gpu_program4 is required.

This extension is written against the OpenGL 2.0 specification.

This extension depends trivially on EXT_texture_array.

This extension depends trivially on NV_texture_shader.

This extension depends trivially on EXT_texture_integer.

This extension depends trivially on ARB_texture_float.

This extension depends trivially on ARB_half_float_pixel.

Overview

This extension provides a new texture type, called a buffer texture. Buffer textures are one-dimensional arrays of texels whose storage comes from an attached buffer object. When a buffer object is bound to a buffer texture, a format is specified, and the data in the buffer object is treated as an array of texels of the specified format.

The use of a buffer object to provide storage allows the texture data to be specified in a number of different ways: via buffer object loads (BufferData), direct CPU writes (MapBuffer), framebuffer readbacks (EXT_pixel_buffer_object extension). A buffer object can also be loaded by transform feedback (NV_transform_feedback extension), which captures selected transformed attributes of vertices processed by the GL. Several

of these mechanisms do not require an extra data copy, which would be required when using conventional TexImage-like entry points.

Buffer textures do not support mipmapping, texture lookups with normalized floating-point texture coordinates, and texture filtering of any sort, and may not be used in fixed-function fragment processing. They can be accessed via single texel fetch operations in programmable shaders. For assembly shaders (NV_gpu_program4), the TXF instruction is used. For GLSL, a new sampler type and texel fetch function are used.

While buffer textures can be substantially larger than equivalent one-dimensional textures; the maximum texture size supported for buffer textures in the initial implementation of this extension is 2^27 texels, versus 2^13 (8192) texels for otherwise equivalent one-dimensional textures. When a buffer object is attached to a buffer texture, a size is not specified; rather, the number of texels in the texture is taken by dividing the size of the buffer object by the size of each texel.

New Procedures and Functions

```
void TexBufferEXT(enum target, enum internalformat, uint buffer);
```

New Tokens

Accepted by the <target> parameter of BindBuffer, BufferData, BufferSubData, MapBuffer, BindTexture, UnmapBuffer, GetBufferSubData, GetBufferParameteriv, GetBufferPointerv, and TexBufferEXT, and the <pname> parameter of GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

TEXTURE_BUFFER_EXT	0x8C2A

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

MAX_TEXTURE_BUFFER_SIZE_EXT	0x8C2B
TEXTURE_BINDING_BUFFER_EXT	0x8C2C
TEXTURE_BUFFER_DATA_STORE_BINDING_EXT	0x8C2D
TEXTURE_BUFFER_FORMAT_EXT	0x8C2E

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

(Insert new Section 3.8.4, Buffer Textures. Renumber subsequent sections.)

In addition to one-, two-, and three-dimensional and cube map textures described in previous sections, one additional type of texture is supported. A buffer texture is similar to a one-dimensional texture. However, unlike other texture types, the texel array is not stored as part of the texture. Instead, a buffer object is attached to a buffer texture and the texel array is taken from the data store of an attached buffer object. When the contents of a buffer object's data store are modified, those changes are reflected in the contents of any buffer texture to which

the buffer object is attached. Also unlike other textures, buffer textures do not have multiple image levels; only a single data store is available.

The command

```
void TexBufferEXT(enum target, enum internalformat, uint buffer);
```

attaches the storage for the buffer object named <buffer> to the active buffer texture, and specifies an internal format for the texel array found in the attached buffer object. If <buffer> is zero, any buffer object attached to the buffer texture is detached, and no new buffer object is attached. If <buffer> is non-zero, but is not the name of an existing buffer object, the error INVALID_OPERATION is generated. <target> must be TEXTURE_BUFFER_EXT. <internalformat> specifies the storage format, and must be one of the sized internal formats found in Table X.1.

When a buffer object is attached to a buffer texture, the buffer object's data store is taken as the texture's texel array. The number of texels in the buffer texture's texel array is given by

```
floor(<buffer_size> / (<components> * sizeof(<base_type>)),
```

where <buffer_size> is the size of the buffer object, in basic machine units and <components> and <base_type> are the element count and base data type for elements, as specified in Table X.1. The number of texels in the texel array is then clamped to the implementation-dependent limit MAX_TEXTURE_BUFFER_SIZE_EXT. When a buffer texture is accessed in a shader, the results of a texel fetch are undefined if the specified texel number is greater than or equal to the clamped number of texels in the texel array.

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel number being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object's data store is interpreted as an array of elements of the GL data type corresponding to <internalformat>. Each texel consists of one to four elements that are mapped to texture components (R, G, B, A, L, and I). Element <m> of the texel numbered <n> is taken from element <n> * <components> + <m> of the attached buffer object's data store. Elements and texels are both numbered starting with zero. For texture formats with normalized components, the extracted values are converted to floating-point values according to Table 2.9. The components of the texture are then converted to an (R,G,B,A) vector according to Table X.21, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture's internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in Table X.1.

Sized Internal Format	Base Type	Components	Norm	Component 0 1 2 3
ALPHA8	ubyte	1	Y	A . . .
ALPHA16	ushort	1	Y	A . . .
ALPHA16F_ARB	half	1	N	A . . .
ALPHA32F_ARB	float	1	N	A . . .
ALPHA8I_EXT	byte	1	N	A . . .
ALPHA16I_EXT	short	1	N	A . . .
ALPHA32I_EXT	int	1	N	A . . .
ALPHA8UI_EXT	ubyte	1	N	A . . .
ALPHA16UI_EXT	ushort	1	N	A . . .
ALPHA32UI_EXT	uint	1	N	A . . .
LUMINANCE8	ubyte	1	Y	L . . .
LUMINANCE16	ushort	1	Y	L . . .
LUMINANCE16F_ARB	half	1	N	L . . .
LUMINANCE32F_ARB	float	1	N	L . . .
LUMINANCE8I_EXT	byte	1	N	L . . .
LUMINANCE16I_EXT	short	1	N	L . . .
LUMINANCE32I_EXT	int	1	N	L . . .
LUMINANCE8UI_EXT	ubyte	1	N	L . . .
LUMINANCE16UI_EXT	ushort	1	N	L . . .
LUMINANCE32UI_EXT	uint	1	N	L . . .
LUMINANCE8_ALPHA8	ubyte	2	Y	L A . .
LUMINANCE16_ALPHA16	ushort	2	Y	L A . .
LUMINANCE_ALPHA16F_ARB	half	2	N	L A . .
LUMINANCE_ALPHA32F_ARB	float	2	N	L A . .
LUMINANCE_ALPHA8I_EXT	byte	2	N	L A . .
LUMINANCE_ALPHA16I_EXT	short	2	N	L A . .
LUMINANCE_ALPHA32I_EXT	int	2	N	L A . .
LUMINANCE_ALPHA8UI_EXT	ubyte	2	N	L A . .
LUMINANCE_ALPHA16UI_EXT	ushort	2	N	L A . .
LUMINANCE_ALPHA32UI_EXT	uint	2	N	L A . .
INTENSITY8	ubyte	1	Y	I . . .
INTENSITY16	ushort	1	Y	I . . .
INTENSITY16F_ARB	half	1	N	I . . .
INTENSITY32F_ARB	float	1	N	I . . .
INTENSITY8I_EXT	byte	1	N	I . . .
INTENSITY16I_EXT	short	1	N	A . . .
INTENSITY32I_EXT	int	1	N	A . . .
INTENSITY8UI_EXT	ubyte	1	N	A . . .
INTENSITY16UI_EXT	ushort	1	N	A . . .
INTENSITY32UI_EXT	uint	1	N	A . . .
RGBA8	ubyte	4	Y	R G B A
RGBA16	ushort	4	Y	R G B A
RGBA16F_ARB	half	4	N	R G B A
RGBA32F_ARB	float	4	N	R G B A
RGBA8I_EXT	byte	4	N	R G B A
RGBA16I_EXT	short	4	N	R G B A
RGBA32I_EXT	int	4	N	R G B A
RGBA8UI_EXT	ubyte	4	N	R G B A
RGBA16UI_EXT	ushort	4	N	R G B A
RGBA32UI_EXT	uint	4	N	R G B A

Table X.1, Internal Formats for Buffer Textures. For each format, the data type of each element is indicated in the "Base Type" column and the element count is in the "Components" column. The "Norm" column indicates whether components should be treated as normalized floating-point values. The "Component 0, 1, 2, and 3" columns indicate the mapping of each element of a texel to texture components.

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named TEXTURE_BUFFER_EXT, in order to specify, modify, or read the buffer object's data store. The buffer object bound to TEXTURE_BUFFER_EXT has no effect on rendering. A buffer object is bound to TEXTURE_BUFFER_EXT by calling BindBuffer with <target> set to TEXTURE_BUFFER_EXT. If no corresponding buffer object exists, one is initialized as defined in section 2.9.

The commands BufferData, BufferSubData, MapBuffer, and UnmapBuffer may all be used with <target> set to TEXTURE_BUFFER_EXT. In this case, these commands operate in the same fashion as described in section 2.9, but on the buffer currently bound to the TEXTURE_BUFFER_EXT target.

Modify Section 3.8.11, Texture State and Proxy State (p. 178)

(insert into the first paragraph of the section, p. 178) ... a zero compressed size, and zero-sized components). The buffer texture target contains an integer identifying the buffer object that buffer that provided the data store for the texture, initially zero, and an integer identifying the internal format of the texture, initially LUMINANCE8. Next, there are the two sets of texture properties; ...

Modify Section 3.8.12, Texture Objects (p. 180)

(modify first paragraphs of section, p. 180, simply adding references to buffer textures, which are treated as texture objects)

In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, and TEXTURE_BUFFER_EXT, named one-, two-, and three-dimensional, cube map, and buffer texture objects can be created and operated upon. 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, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT. The binding is effected by calling

```
  void BindTexture( enum target, uint texture );
```

with target set to the desired texture target and texture set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.11, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT, it is and remains a one-, two-, three-dimensional, cube map, or buffer texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified target. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to target is broken.

...

In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, and TEXTURE_BUFFER_EXT have one-, two-, three-dimensional, cube map, and buffer texture state vectors respectively associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, three-dimensional, cube map, and buffer texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT respectively while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

```
void DeleteTextures( sizei n, uint *textures );
```

textures contains n names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP, or TEXTURE_BUFFER_EXT is deleted, it is as though BindTexture had been executed with the same target and texture zero. Unused names in textures are silently ignored, as is the value zero.

(modify second paragraph, p. 182, adding buffer textures, plus cube map textures, which is an oversight in the core specification)

The texture object name space, including the initial one-, two-, and three-dimensional, cube map, and buffer texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Modify Section 5.4, Display Lists (p. 237)

(modify "Vertex buffer objects" portion of the list of non-listable commands, p. 241)

Buffer objects: GenBuffers, DeleteBuffers, BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, and TexBufferEXT.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.13, Buffer Object Queries (p. 255)

(modify the first paragraph on p. 256) The command

```
void GetBufferSubData( enum target, intptr offset,
                       sizeiptr size, void *data );
```

queries the data contents of a buffer object. target is ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, or TEXTURE_BUFFER_EXT. ...

(modify the last paragraph of the section, p. 256) While the data store of a buffer object is mapped, the pointer to the data store can be queried by calling

```
void GetBufferPointerv( enum target, enum pname, void **params );
```

with target set to ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, or TEXTURE_BUFFER_EXT, and pname set to BUFFER MAP POINTER.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on EXT_texture_array

If EXT_texture_array is supported, the introductory language describing buffer textures should acknowledge the existence of array textures. Other than that, there are no dependencies between the two extensions.

Dependencies on NV_texture_shader

If NV_texture_shader is not supported, references to the signed normalized internal formats provided by that extension should be removed, and such formats may not be passed to TexBufferEXT.

Dependencies on EXT_texture_integer

If EXT_texture_integer is not supported, references to the signed and unsigned integer internal formats provided by that extension should be removed, and such formats may not be passed to TexBufferEXT.

Dependencies on ARB_texture_float

If ARB_texture_float is not supported, references to the floating-point internal formats provided by that extension should be removed, and such formats may not be passed to TexBufferEXT.

Dependencies on ARB_half_float_pixel

If ARB_texture_float is not supported, references to the 16-bit floating-point internal formats provided by ARB_texture_float should be removed, and such formats may not be passed to TexBufferEXT. If an implementation supports ARB_texture_float, but does not support ARB_half_float_pixel, 16-bit floating-point texture formats may be available using normal texture mechanisms, but not with buffer textures.

Errors

INVALID_OPERATION is generated by TexBufferEXT if <buffer> is non-zero and is not the name of an existing buffer object.

New State

(add to table 6.15, Texture State Per Texture Unit/Binding Point p. 276)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BINDING_BUFFER_EXT	2*xZ+	GetIntegerv	0	Texture object bound to TEXTURE_BUFFER_EXT	3.8.12	texture

(add to table 6.16, Texture State Per Texture Object, p. 276)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BUFFER_DATA_STORE_BINDING_EXT	nxZ+	GetIntegerv	0	Buffer object bound as the data store for the active image unit's buffer texture	3.8.12	texture
TEXTURE_BUFFER_FORMAT_EXT	nxZ+	GetIntegerv	LUMINANCE8	Internal format for the active image unit's buffer texture	3.8.12	texture

(add to table 6.37, Miscellaneous State, p. 298)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BUFFER_EXT	Z+	GetIntegerv	0	Buffer object bound to the generic buffer texture binding point	3.8.12	texture

New Implementation Dependent State

(modify Table 6.32, p. 293)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TEXTURE_BUFFER_SIZE_EXT	Z+	GetIntegerv	65536	number of addressable texels for buffer textures	3.8.4	-

Issues

(1) Buffer textures are potentially large one-dimensional arrays that can be accessed with single-texel fetches. How should this functionality be exposed?

RESOLVED: Several options were considered. The final approach creates a new type of texture object, called a buffer texture, whose texel array is taken from the data store from a buffer object. The combined set of extensions using buffer objects provides numerous locations where the GL can read and write data to a buffer object:

EXT_vertex_buffer_object allows vertex attributes to be pulled from a buffer object.

EXT_pixel_buffer_object allows pixel operations (DrawPixels, ReadPixels, TexImage) to read or write data to a buffer object.

EXT_parameter_buffer_object and EXT_bindable_uniform allows assembly vertex, fragment, and geometry programs, and all GLSL shaders to read program parameter / uniform data from a buffer object.

EXT_texture_buffer_object allows programs to read texture data from a buffer object.

NV_transform_feedback allows programs to write transformed vertex attributes to a buffer object.

When combined, interesting feedback paths are possible, where large arrays of data can be generated by the GPU and the consumed by it in multi-pass algorithms, using the buffer object's storage to hold intermediate data. This allows applications to run complicated algorithms on the GPU without necessarily pulling data back to host CPU for additional processing.

Given that buffer object memory is visible to users as raw memory, all uses of the memory must have well-defined data formats. For VBO and PBO, those formats are explicitly given by calls such as VertexPointer, TexImage2D, or ReadPixels. When used as a buffer texture, it is necessary to specify an internal format with which the bytes of the buffer object's data store are interpreted.

Another option considered was to greatly increase the maximum texture size for 1D texture. This has the advantage of not requiring new mechanisms. However, there are a couple limitations of this approach. First, conventional textures have their own storage that is not accessible elsewhere, which limits some of the sharing opportunities described above. Second, buffer textures do have slightly different hardware implementations than 1D textures. In the hardware of interest, "normal" 1D textures can be mipmapped and filtered, but have a maximum size that is considerably smaller than that supported for buffer textures. If both texture types used the same API mechanism, it might be necessary to reprogram texture hardware and/or shaders depending on the size of the textures used. This will incur CPU overhead to determine if such reprogramming is necessary and to perform the reprogramming if so.

(2) *Since buffer textures borrow storage from buffer objects, whose storage is visible to applications, a format must be imposed on the bytes of the buffer object. What texture formats are supported for buffer objects?*

RESOLVED: All sized one-, two-, and four-component internal formats with 8-, 16-, and 32-bit components are supported. Unsized internal formats, and sized formats with other component sizes are also not supported. Three-component (RGB) formats are not supported due to hardware limitations.

All component data types supported for normal textures are also supported for buffer textures. This includes unsigned [0,1] normalized components (e.g., RGBA8), floating-point components from ARB_texture_float (e.g., RGBA32F_ARB), signed and unsigned integer components from EXT_texture_integer (e.g., RGBA8I_EXT, RGBA16UI_EXT),

and signed [-1,+1] normalized components from NV_texture_shader (e.g., SIGNED_RGBA8_NV).

(3) How can arrays of three-component vectors be accessed by applications?

RESOLVED: Several approaches are possible.

First, the vectors can be padded out to four components (RGBA), with an extra unused component for each texel. This has a couple undesirable properties: it adds 33% to the required storage and adding the extra component may require reformatting of original data generated by the application. However, the data in this format can be retrieved with a single 32-, 64-, or 128-bit lookup.

Alternately, the buffer texture can be defined using a single component, and a shader can perform three lookups to separately fetch texels 3*N, 3*N+1, and 3*N+2, combining the result in a three-component vector representing "RGB" texel N. This doesn't require extra storage or reformatting and doesn't require additional bandwidth for texture fetches. But it does require additional shader instructions to obtain each texel.

(4) Does this extension support fixed-function fragment processing, somehow allowing buffer textures to be accessed without programmable shaders?

RESOLVED: No. We expect that it would be difficult to properly access a buffer texture and combine the returned texel with other color or texture data, given the extremely limited programming model provided by fixed-function fragment processing.

Note also that the single-precision floating-point representation commonly used by current graphics hardware is not sufficiently precise to exactly represent all texels in a large buffer texture. For example, it is not possible to represent 2^24+1 using the 32-bit IEEE floating-point representation.

(5) What happens if a buffer object is deleted or respecified when bound to a buffer texture?

RESOLVED: BufferData is allowed to be used to update a buffer object that has already been bound to a texture with TexBuffer. The update to the data is not guaranteed to affect the texture until next time it is bound to a texture image unit. When DeleteBuffers is called, any buffer that is bound to a texture is removed from the names array, but remains as long as it is bound to a texture. The buffer is fully removed when the texture unbinds it or when the texture buffer object is deleted.

(6) Should applications be able to modify the data store of a buffer object while it is bound to a buffer texture?

RESOLVED: An application is allowed to update the data store for a buffer object when the buffer object is bound to a texture.

(7) Do buffer textures support texture parameters (TexParameter) or queries (GetTexParameter, GetTexLevelParameter, GetTexImage)?

RESOLVED: No. None of the existing parameters apply to buffer textures, and this extension doesn't introduce the need for any new ones. Buffer textures have no levels, and the size in texels is implicit (based on the data store). Given that the texels themselves are obtained from a buffer object, it seems more appropriate to retrieve such data with buffer object queries. The only "parameter" of a buffer texture is the internal format, which is specified at the same time the buffer object is bound.

Note that the spec edits above don't add explicit error language for any of these cases. That is because each of the functions enumerate the set of valid <target> parameters. Not editing the spec to allow TEXTURE_BUFFER_EXT in these cases means that target is not legal, and an INVALID_ENUM error should be generated.

(8) What about indirect rendering with a mix of big- and little-endian clients? If components are 16- or 32-bit, how are they interpreted?

RESOLVED: Buffer object data are interpreted according to the native representation of the server. If the server and client have different endianness, applications must perform byte swapping as needed to match the server's representation. No mechanism is provided to perform this byte swapping on buffer object updates or when texels are fetched.

The same problem also exists when buffer objects are used for vertex arrays (VBO). For buffer objects used for pixel packing and unpacking (ARB_pixel_buffer_object), the PixelStore byte swapping parameters (PACK_SWAP_BYTES, UNPACK_SWAP_BYTES) would presumably apply and could be used to perform the necessary byte swapping.

(9) Should the set of formats supported for buffer textures be enumerated, or should the extension instead nominally support all formats, but accept only an implementation-dependent subset?

RESOLVED: Provide a specified set of supported formats. This extension simply enumerates all 8-, 16-, and 32-byte internal formats with 1, 2, or 4 components, and specifies the mapping of unformatted buffer object data to texture components. A follow-on extension could be done to support 3-component texels when better native hardware support is available.

Other than 3-component texels, the set of formats supported seems pretty compehensive. We expect that buffer textures would be used for general computational tasks, where there is little need for formats with smaller components (e.g., RGBA4444). Such formats are generally not supported natively on CPUs today. With the general computational model provided by NV_gpu_program4 and EXT_gpu_shader4, it would be possible to treat such "packed" formats as larger single-component formats and unpack them with a small number of shader instructions.

If and when double-precision floats or 64-bit integers are supported as basic types usable by shaders, we would expect that an extension would add new texture internal formats with 64-bit components and that those

formats would also be supported for general-purpose textures and buffer textures as well.

(10) How are buffer textures supported in GLSL?

RESOLVED: Create a new sampler type (samplerBuffer) for buffer textures and add a new lookup function (texelFetchBuffer) to explicitly access them using texture hardware.

Other possibilities considered included extending the notion of bindable uniforms to support uniforms whose corresponding buffer objects can be bound to texture resources (e.g., "texture bindable uniform" instead of "bindable uniform"). We also considered automatically assigning bindable uniforms to texture or shader resources as appropriate. Note that the restrictions, size limits, and performance characterstics of buffer textures and parameter buffers (NV_parameter_buffer_object) differ. Automatic handling of uniforms adds driver complexity and may tend to hide performance characteristics since it isn't clear what resource would be used for what variable. Additionally, it could require shader recompilation if the size of a uniform array is variable, and the hardware resource used depended on the size.

In the end, the texture approach seemed the simplest, and we chose that. It might be worth doing something more complex in the future.

(11) What is the TEXTURE_BUFFER_EXT buffer object binding point good for?

RESOLVED: It can be used for loading data into buffer objects, and for mapping and unmapping buffers, both without disturbing other binding points. Otherwise, it has no effect on GL operations, since buffer objects are bound to textures using the TexBufferEXT() command that does not affect the buffer object binding point.

Buffer object binding points have mixed usage. In the EXT_vertex_buffer_object extension (OpenGL 1.5), there are two binding points. The ELEMENT_ARRAY_BUFFER has a direct effect on rendering, as it modifies DrawElements() calls. The effect of ARRAY_BUFFER is much more indirect; it is only used to affect subsequent vertex array calls (e.g., VertexPointer) and has no direct effect on rendering. The reason for this is that the API was retrofitted on top of existing vertex array APIs. If a new vertex array API were created that emphasized or even required the use of buffer objects, it seems likely that the buffer object would be included in the calls equivalent to today's VertexPointer() call.

(12) How is the various buffer texture-related state queried?

RESOLVED: There are three pieces of state that can be queried: (a) the texture object bound to buffer texture binding point for the active texture image unit, (b) the buffer object whose data store was used by that texture object, and (c) the buffer object bound to the TEXTURE_BUFFER_EXT binding point.

All three are queried with GetIntegerv, because it didn't seem worth the trouble to add one or more new query functions. Note that for (a) and (b), the texture queried is the one bound to TEXTURE_BUFFER_EXT on the active texture image unit.

(13) Should we provide a new set of names for the signed normalized textures introduced in NV_texture_shader that match the convention used for floating-point and integer textures?

RESOLVED: No.

(14) Can a buffer object be attached to more than one buffer texture at once?

RESOLVED: Multiple buffer textures may attach to the same buffer object simultaneously.

(15) How does this extension interact with display lists?

RESOLVED: Buffer object commands can't be compiled into a display list. The new command in this extension uses buffer objects, so we specify that it also can't be compiled into a display list.

Revision History

Rev.	Date	Author	Changes
4	10/30/07	ewerness	Add resolutions to various issues
3	--		Pre-release revisions.

Name

EXT_texture_compression_latc

Name Strings

GL_EXT_texture_compression_latc
GL_NV_texture_compression_latc (legacy)

Contributors

Mark J. Kilgard, NVIDIA
Pat Brown, NVIDIA
Yanjun Zhang, S3

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 1/21/2008
Revision: 1.2

Number

331

Dependencies

OpenGL 1.3 or ARB_texture_compression required

This extension is written against the OpenGL 2.0 (September 7, 2004) specification.

Overview

This extension introduces four new block-based texture compression formats suited for unsigned and signed luminance and luminance-alpha textures (hence the name "latc" for Luminance-Alpha Texture Compression).

These formats are designed to reduce the storage requirements and memory bandwidth required for luminance and luminance-alpha textures by a factor of 2-to-1 over conventional uncompressed luminance and luminance-alpha textures with 8-bit components (GL_LUMINANCE8 and GL_LUMINANCE8_ALPHA8).

The compressed signed luminance-alpha format is reasonably suited for storing compressed normal maps.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2D and the <format> parameter of CompressedTexSubImage2D:

COMPRESSED_LUMINANCE_LATC1_EXT	0x8C70
COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT	0x8C71
COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT	0x8C72
COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT	0x8C73

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

-- Section 3.8.1, Texture Image Specification

Add to Table 3.17 (page 155): Specific compressed internal formats

Compressed Internal Format	Base Internal Format
COMPRESSED_LUMINANCE_LATC1_EXT	LUMINANCE
COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT	LUMINANCE
COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT	LUMINANCE_ALPHA
COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT	LUMINANCE_ALPHA

-- Section 3.8.2, Alternative Texture Image Specification Commands

Add to the end of the section (page 163):

"If the internal format of the texture image being modified is COMPRESSED_LUMINANCE_LATC1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT, the texture is stored using one of the two LATC compressed texture image encodings (see appendix). 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 LATC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

-- Section 3.8.3, Compressed Texture Images

Add after the 4th paragraph (page 164) at the end of the CompressedTexImage discussion:

"If <internalformat> is COMPRESSED_LUMINANCE_LATC1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT, the compressed texture is stored using one of several LATC compressed texture image formats. The LATC texture compression algorithm supports only 2D images without borders. CompressedTexImage1D and CompressedTexImage3D produce an INVALID_ENUM error if <internalformat> is an LATC format. CompressedTexImage2D will produce an INVALID_OPERATION error if <border> is non-zero.

Add to the end of the section (page 166) at the end of the CompressedTexSubImage discussion:

"If the internal format of the texture image being modified is COMPRESSED_LUMINANCE_LATC1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT, the texture is stored using one of the several LATC compressed texture image formats. Since the LATC texture compression algorithm supports only 2D images, CompressedTexSubImage1D and CompressedTexSubImage3D produce an INVALID_ENUM error if <format> is an LATC format. Since LATC 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 LATC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on ARB_texture_compression

If ARB_texture_compression is supported, all the errors and accepted tokens for CompressedTexImage1D, CompressedTexImage2D, CompressedTexImage3D, CompressedTexSubImage1D, CompressedTexSubImage2D, and CompressedTexSubImage3D also apply respectively to the ARB-suffixed CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, and CompressedTexSubImage3DARB.

Errors

INVALID_ENUM is generated by CompressedTexImage1D or CompressedTexImage3D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT.

INVALID_OPERATION is generated by CompressedTexImage2D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1D or CompressedTexSubImage3D if <format> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT.

INVALID_OPERATION is generated by TexSubImage2D CopyTexSubImage2D, or CompressedTexSubImage2D if TEXTURE_INTERNAL_FORMAT is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT, COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT, or COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_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 LATC 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 CompressedTexSubImage1D,
DELETE: CompressedTexSubImage2D, or CompressedTexSubImage3D 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

4 new state values are added for the per-texture object
GL_TEXTURE_INTERNAL_FORMAT state.

In the "Textures" state table(page 278), increment the
TEXTURE_INTERNAL_FORMAT subscript for Z by 4 in the "Type" row.

[NOTE: The OpenGL 2.0 specification actually should read "n x Z48*"
because of the 6 generic compressed internal formats in table 3.18.]

New Implementation Dependent State

None

Appendix

LATC Compressed Texture Image Formats

Compressed texture images stored using the LATC compressed image
encodings 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 LATC image has a width or height
less than four, the data corresponding to texels outside the image
are irrelevant and undefined.

When an LATC 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 LATC image, the block containing the texel at offset
(<x>, <y>) begins at an offset (in bytes) relative to the base of the
image of:

blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).

The data corresponding to a specific texel (<x>, <y>) are extracted from a 4x4 texel block using a relative (x,y) value of

(<x> modulo 4, <y> modulo 4).

There are four distinct LATC image formats:

COMPRESSED_LUMINANCE_LATC1: Each 4x4 block of texels consists of 64 bits of unsigned luminance image data.

Each luminance image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

lum0, lum1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5

The 6 "bits_*" bytes of the block are decoded into a 48-bit bit vector:

```
bits   = bits_0 +
         256 * (bits_1 +
                256 * (bits_2 +
                       256 * (bits_3 +
                              256 * (bits_4 +
                                     256 * bits_5))))
```

lum0 and lum1 are 8-bit unsigned integers that are unpacked to luminance values LUM0 and LUM1 as though they were pixels with a <format> of LUMINANCE and a type of UNSIGNED_BTYE.

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)+2..3*(4*y+x)+0]

where bit 47 is the most significant and bit 0 is the least significant bit.

```
    The luminance value L for a texel at location (x,y) in the block
    is given by:

        LUM0,              if lum0 > lum1 and code(x,y) == 0
        LUM1,              if lum0 > lum1 and code(x,y) == 1
        (6*LUM0+  LUM1)/7, if lum0 > lum1 and code(x,y) == 2
        (5*LUM0+2*LUM1)/7, if lum0 > lum1 and code(x,y) == 3
        (4*LUM0+3*LUM1)/7, if lum0 > lum1 and code(x,y) == 4
        (3*LUM0+4*LUM1)/7, if lum0 > lum1 and code(x,y) == 5
        (2*LUM0+5*LUM1)/7, if lum0 > lum1 and code(x,y) == 6
        (  LUM0+6*LUM1)/7, if lum0 > lum1 and code(x,y) == 7

        LUM0,              if lum0 <= lum1 and code(x,y) == 0
        LUM1,              if lum0 <= lum1 and code(x,y) == 1
        (4*LUM0+  LUM1)/5, if lum0 <= lum1 and code(x,y) == 2
        (3*LUM0+2*LUM1)/5, if lum0 <= lum1 and code(x,y) == 3
        (2*LUM0+3*LUM1)/5, if lum0 <= lum1 and code(x,y) == 4
        (  LUM0+4*LUM1)/5, if lum0 <= lum1 and code(x,y) == 5
        MINLUM,            if lum0 <= lum1 and code(x,y) == 6
        MAXLUM,            if lum0 <= lum1 and code(x,y) == 7

    MINLUM and MAXLUM are 0.0 and 1.0 respectively.

Since the decoded texel has a luminance format, the resulting RGBA
value for the texel is (L,L,L,1).
```

COMPRESSED_SIGNED_LUMINANCE_LATC1: Each 4x4 block of texels consists of 64 bits of signed luminance image data. The luminance values of a texel are extracted in the same way as COMPRESSED_LUMINANCE_LATC1 except lum0, lum1, LUM0, LUM1, MINLUM, and MAXLUM are signed values defined as follows:

```
    lum0 and lum1 are 8-bit signed (two's complement) integers.

           { lum0 / 127.0, lum0 > -128
    LUM0 = {
           { -1.0,         lum0 == -128

           { lum1 / 127.0, lum1 > -128
    LUM1 = {
           { -1.0,         lum1 == -128

    MINLUM = -1.0

    MAXLUM =  1.0
```

CAVEAT for signed lum0 and lum1 values: the expressions "lum0 > lum1" and "lum0 <= lum1" above are considered undefined (read: may vary by implementation) when lum0 equals -127 and lum1 equals -128, This is because if lum0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed LA formats should avoid encoding blocks where lum0 equals -127 and lum1 equals -128.

COMPRESSED_LUMINANCE_ALPHA_LATC2: Each 4x4 block of texels consists of 64 bits of compressed unsigned luminance image data followed by 64 bits of compressed unsigned alpha image data.

The first 64 bits of compressed luminance are decoded exactly like COMPRESSED_LUMINANCE_LATC1 above.

The second 64 bits of compressed alpha are decoded exactly like COMPRESSED_LUMINANCE_LATC1 above except the decoded value L for this second block is considered the resulting alpha value A.

Since the decoded texel has a luminance-alpha format, the resulting RGBA value for the texel is (L,L,L,A).

COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2: Each 4x4 block of texels consists of 64 bits of compressed signed luminance image data followed by 64 bits of compressed signed alpha image data.

The first 64 bits of compressed luminance are decoded exactly like COMPRESSED_SIGNED_LUMINANCE_LATC1 above.

The second 64 bits of compressed alpha are decoded exactly like COMPRESSED_SIGNED_LUMINANCE_LATC1 above except the decoded value L for this second block is considered the resulting alpha value A.

Since this image has a luminance-alpha format, the resulting RGBA value is (L,L,L,A).

Issues

1) What should these new formats be called?

RESOLVED: "latc" for Luminance-Alpha Texture Compression.

2) How should the uncompressed and filtered texels be returned by texture fetches?

RESOLVED: Luminance values show up as they do conventionally as (L,L,L,1) where the luminance value L is replicated in the red, green, and blue components and alpha is forced to 1. Likewise, luminance-alpha values show up as (L,L,L,A) where A is the alpha value.

Alternatively, prior extensions such as NV_float_buffer and NV_texture_shader have introduced formats such as GL_FLOAT_R_NV and GL_DSDT_NV where one- and two-component texture formats show up as (X,0,0,1) or (X,Y,0,1) RGBA texels. Such formats have not proven popular. In particular, they interact awkwardly with the pixel path and conventional texture environment modes.

The (X,Y,0,1) convention, particularly with signed components, is nice for normal maps because a normalized vector can be formed by a shader program by computing sqrt(abs(1-X*X-Y*Y)) for the Z component. However, this niceness is mostly conceptual however since the same effect can be accomplished with swizzling as shown in this GLSL code:

```
    vec2 texLA  = texture2D(samplerLA, gl_TexCoord[0]).xw;
    vec3 normal = vec3(texLA.x,
                       texLA.y,
                       sqrt(abs(1-texLA.x*texLA.x-texLA.y*texLA.y)));
```

The most important reason to make these new compressed formats show up identically to conventional luminance and luminance-alpha texels is to allow applications to seamlessly substitute the new compressed formats for existing GL_LUMINANCE and GL_LUMINANCE_ALPHA textures. Alternative component arrangements would make it more cumbersome for existing applications to switch over luminance and luminance-alpha textures to these compressed formats.

3) *Should luminance and luminance-alpha compression formats with signed components be introduced when the core specification lacked uncompressed luminance and luminance-alpha texture formats?*

RESOLVED: Yes, signed luminance and luminance-alpha compression formats should be added.

Signed luminance-alpha formats are suited for compressed normal maps. Compressed normal maps may well be the dominant use of this extension.

Unsigned luminance-alpha formats require an extra "expand normal" operation to convert [0,1] to [-1,+1]. Direct support for signed luminance-alpha formats avoids this step in a shader program.

4) *Should there be a mix of signed luminance and unsigned alpha or vice versa?*

RESOLVED: No.

NV_texture_shader provided an internal format (GL_SIGNED_RGB_UNSIGNED_ALPHA_NV) with mixed signed and unsigned components. The format saw little usage. There's no reason to think a GL_SIGNED_LUMINANCE_UNSIGNED_ALPHA format would be any more useful or popular.

5) *How are signed integer values mapped to floating-point values?*

RESOLVED: A signed 8-bit two's complement value X is computed to a floating-point value Xf with the formula:

```
          { X / 127.0, X > -128
    Xf = {
          { -1.0,      X == -128
```

This conversion means -1, 0, and +1 are all exactly representable, however -128 and -127 both map to -1.0. Mapping -128 to -1.0 avoids the numerical awkwardness of have a representable value slightly more negative than -1.0.

This conversion is intentionally NOT the "byte" conversion listed in Table 2.9 for component conversions. That conversion says:

```
Xf = (2*X + 1) / 255.0
```

The Table 2.9 conversion is incapable of exactly representing zero.

6) *How will signed components resulting from GL_COMPRESSED_SIGNED_LUMINANCE_LATC1_EXT and GL_COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT texture fetches interact with fragment coloring?*

RESOLVED: The specification language for this extension is silent about clamping behavior leaving this to the core specification and other extensions. The clamping or lack of clamping is left to the core specification and other extensions.

For assembly program extensions supporting texture fetches (ARB_fragment_program, EXT_fragment_program, EXT_vertex_program3, etc.) or the OpenGL Shading Language, these signed formats will appear as expected with unclamped signed components as a result of a texture fetch instruction.

If ARB_color_buffer_float is supported, its clamping controls will apply.

NV_texture_shader extension, if supported, adds support for fixed-point textures with signed components and relaxed the fixed-function texture environment clamping appropriately. If the NV_texture_shader extension is supported, its specified behavior for the texture environment applies where intermediate values are clamped to [-1,1] unless stated otherwise as in the case of explicitly clamped to [0,1] for GL_COMBINE. or clamping the linear interpolation weight to [0,1] for GL_DECAL and GL_BLEND.

Otherwise, the conventional core texture environment clamps incoming, intermediate, and output color components to [0,1].

This implies that the conventional texture environment functionality of unextended OpenGL 1.5 or OpenGL 2.0 without using GLSL (and with none of the extensions referred to above) is unable to make proper use of the signed texture formats added by this extension because the conventional texture environment requires texture source colors to be clamped to [0,1]. Texture filtering of these signed formats would be still signed, but negative values generated post-filtering would be clamped to zero by the core texture environment functionality. The expectation is clearly that this extension would be co-implemented with one of the previously referred to extensions or used with GLSL for the new signed formats to be useful.

7) *Should a specific normal map compression format be added?*

RESOLVED: No.

It's probably short-sighted to design a format just for normal

maps. Indeed, NV_texture_shader added a GL_SIGNED_HILO_NV format with exactly the kind of "hemisphere remap" useful for normal maps and the format went basically unused. Instead, this extension provides the mechanism for compressed normal maps based on the more conventional luminance-alpha format.

The GL_COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT and GL_COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT formats are sufficient for normal maps with additional shader instructions used to generate the 3rd component.

8) *Should uncompressed signed luminance and luminance-alpha formats be added by this extension?*

RESOLVED: No, this extension is focused on just adding compressed texture formats.

The NV_texture_shader extension adds such uncompressed signed texture formats. A distinct multi-vendor extension for signed fixed-point texture formats could provide all or a subset of the signed fixed-point uncompressed texture formats introduced by NV_texture_shader.

9) *What compression ratios does this extension provide?*

The LATC1 formats are 8 bytes (64 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8 data requires 16 bytes (1 byte per texel). This is a 2-to-1 compression ratio.

The LATC2 formats are 16 bytes (128 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8_ALPHA8 data requires 32 bytes (2 bytes per texel). This is again a 2-to-1 compression ratio.

In contrast, the comparable compression ratio for the S3TC formats is 4-to-1.

Arguably, the lower compression ratio allows better compression quality particularly because the LATC formats compress each component separately.

10) *How do these new formats compare with the existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats?*

RESOLVED: The existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats provide a similar 2-to-1 compression ratio but mandate a uniform quantization for all components. In contrast, this extension provides a compression format with 3-bit quantization over a specifiable min/max range that can vary per 4x4 texel tile.

Additionally, many OpenGL implementations do not natively support the GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats but rather silently promote these formats to store 8 bits per component, thereby eliminating any storage/bandwidth advantage for these formats.

11) Does this extension require EXT_texture_compression_s3tc?

RESOLVED: No.

As written, this specification does not rely on wording of the EXT_texture_compression_s3tc extension. For example, certain discussion added to Sections 3.8.2 and 3.8.3 is quite similar to corresponding EXT_texture_compression_s3tc language.

12) Should anything be said about the precision of texture filtering for these new formats?

RESOLVED: No precision requirements are part of the specification language since OpenGL extensions typically leave precision details to the implementation.

Realistically, at least 8-bit filtering precision can be expected from implementations (and probably more).

13) Should these formats be allowed to specify 3D texture images when NV_texture_compression_vtc is supported?

RESOLVED: The NV_texture_compression_vtc stacks 4x4 blocks into 4x4x4 bricks. It may be more desirable to represent compressed 3D textures as simply slices of 4x4 blocks.

However the NV_texture_compression_vtc extension expects data passed to the glCompressedTexImage commands to be "bricked" rather than blocked slices.

14) Why is GL_NV_texture_compression_latc also listed in the Name Strings section?

The very first GeForce 8800 driver shipped with the extension designated as NV before EXT-ization with S3 was agreed. Subsequent NVIDIA drivers will rename the extension to its EXT name only.

15) Should the the generic formats GL_COMPRESSED_LUMINANCE and GL_COMPRESSED_LUMINANCE_ALPHA correspond to COMPRESSED_LUMINANCE_LATC1_EXT and COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT respecitively when this extension is supported?

RESOLVED: Yes. While no generic compression is strictly required for an implementation and there might exist superior compression schemes for luminance and luminance-alpha textures in the future, an application should reasonably expect that an implementation that supports EXT_texture_compression_latc will also use these formats for the generic compressed luminance and luminance-alpha formats.

The COMPRESSED_LUMINANCE_LATC1_EXT and COMPRESSED_LUMINANCE_ALPHA_LATC2_EXT are generic enough in their respective luminance and luminance-alpha behavior that these compression formats are acceptable generic compressed formats for luminance and luminance-alpha generic compressed formats.

16) Should the GL_NUM_COMPRESSED_TEXTURE_FORMATS and GL_COMPRESSED_TEXTURE_FORMATS queries return the LATC formats?

RESOLVED: No.

The OpenGL 2.1 specification says "The only values returned by this query [GL_COMPRESSED_TEXTURE_FORMATS"] are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use."

Historically, OpenGL implementation have advertised the RGB and RGBA versions of the S3TC extensions compressed format tokens through this mechanism.

The specification is not sufficiently clear about what "suitable for general-purpose usage" means. Historically that seems to mean unsigned RGB or unsigned RGBA. The DXT1 format supporting alpha (GL_COMPRESSED_RGBA_S3TC_DXT1_EXT) is not exposed in the list (at least for NVIDIA drivers) because the alpha is always 1.0 expect when it is 0.0 when RGB is required to be black. NVIDIA's even limits itself to true linear RGB or RGBA formats, specifically not including EXT_texture_sRGB's sRGB S3TC compressed formats.

Adding luminance and luminance-alpha texture formats (and certainly signed versions of luminance and luminance-alpha formats!) invites potential comptaibility problems with old applications using this mechanism since old applications are unlikely to expect non-RGB or non-RGBA formats to be advertised through this mechanism. However no specific misinteractions with old applications is known.

Applications that seek to use the LATC formats should do so by looking for this extension's name in the string returned by glGetString(GL_EXTENSIONS) rather than what GL_NUM_COMPRESSED_TEXTURE_FORMATS and GL_COMPRESSED_TEXTURE_FORMATS return.

Revision History

Revision 1.1, April 24, 2007: mjk
- Add caveat about how signed LA decompression happens when lum0 equals -127 and lum1 equals -128. This caveat matches a decoding allowance in DirectX 10.

Revision 1.2, January 21, 2008: mjk
- Add issues #15 and #16.

Name

EXT_texture_compression_rgtc

Name Strings

GL_EXT_texture_compression_rgtc

Contributors

Mark J. Kilgard, NVIDIA
Pat Brown, NVIDIA
Yanjun Zhang, S3
Attila Barsi, Holografika

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006, Release 95)

Version

Date: January 21, 2008
Revision: 1.2

Number

332

Dependencies

OpenGL 1.3 or ARB_texture_compression required

This extension is written against the OpenGL 2.0 (September 7, 2004) specification.

Overview

This extension introduces four new block-based texture compression formats suited for unsigned and signed red and red-green textures (hence the name "rgtc" for Red-Green Texture Compression).

These formats are designed to reduce the storage requirements and memory bandwidth required for red and red-green textures by a factor of 2-to-1 over conventional uncompressed luminance and luminance-alpha textures with 8-bit components (GL_LUMINANCE8 and GL_LUMINANCE8_ALPHA8).

The compressed signed red-green format is reasonably suited for storing compressed normal maps.

This extension uses the same compression format as the EXT_texture_compression_latc extension except the color data is stored in the red and green components rather than luminance and alpha.

Representing compressed red and green components is consistent with the BC4 and BC5 compressed formats supported by DirectX 10.

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2D and the <format> parameter of CompressedTexSubImage2D:

COMPRESSED_RED_RGTC1_EXT	0x8DBB
COMPRESSED_SIGNED_RED_RGTC1_EXT	0x8DBC
COMPRESSED_RED_GREEN_RGTC2_EXT	0x8DBD
COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT	0x8DBE

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

-- Section 3.8.1, Texture Image Specification

Add to Table 3.17 (page 155): Specific compressed internal formats

Compressed Internal Format	Base Internal Format
COMPRESSED_RED_RGTC1_EXT	RGB
COMPRESSED_SIGNED_RED_RGTC1_EXT	RGB
COMPRESSED_RED_GREEN_RGTC2_EXT	RGB
COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT	RGB

-- Section 3.8.2, Alternative Texture Image Specification Commands

Add to the end of the section (page 163):

"If the internal format of the texture image being modified is COMPRESSED_RED_RGTC1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT, the texture is stored using one of the two RGTC compressed texture image encodings (see appendix). 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 RGTC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

-- Section 3.8.3, Compressed Texture Images

Add after the 4th paragraph (page 164) at the end of the CompressedTexImage discussion:

"If <internalformat> is COMPRESSED_RED_RGTC1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT, the compressed texture is stored using one of several RGTC compressed texture image formats. The RGTC texture compression algorithm supports only 2D images without borders. CompressedTexImage1D and CompressedTexImage3D produce an INVALID_ENUM error if <internalformat> is an RGTC format. CompressedTexImage2D will produce an INVALID_OPERATION error if <border> is non-zero.

Add to the end of the section (page 166) at the end of the CompressedTexSubImage discussion:

"If the internal format of the texture image being modified is COMPRESSED_RED_RGTC1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT, the texture is stored using one of the several RGTC compressed texture image formats. Since the RGTC texture compression algorithm supports only 2D images, CompressedTexSubImage1D and CompressedTexSubImage3D produce an INVALID_ENUM error if <format> is an RGTC format. Since RGTC 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 RGTC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage2D and CopyTexSubImage2D calls."

-- Section 3.8.8, Texture Minification

Add a sentence to the last paragraph (page 174) just prior to the "Mipmapping" subheading:

"If the texture's internal format lacks components that exist in the texture's base internal format, such components are considered zero when the texture border color is sampled. (So despite the RGB base internal format of the COMPRESSED_RED_RGTC1_EXT and COMPRESSED_SIGNED_RED_RGTC1_EXT formats, the green and blue components of the texture border color are always considered zero. Likewise for the COMPRESSED_RED_GREEN_RGTC2_EXT, and COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT formats, the blue component is always considered zero.)"

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on ARB_texture_compression

If ARB_texture_compression is supported, all the errors and accepted tokens for CompressedTexImage1D, CompressedTexImage2D, CompressedTexImage3D, CompressedTexSubImage1D, CompressedTexSubImage2D, and CompressedTexSubImage3D also apply respectively to the ARB-suffixed CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, and CompressedTexSubImage3DARB.

Errors

INVALID_ENUM is generated by CompressedTexImage1D or CompressedTexImage3D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT.

INVALID_OPERATION is generated by CompressedTexImage2D if <internalformat> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1D or CompressedTexSubImage3D if <format> is COMPRESSED_LUMINANCE_LACT1_EXT, COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT, or COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT.

INVALID_OPERATION is generated by TexSubImage2D
CopyTexSubImage2D, or CompressedTexSubImage2D if
TEXTURE_INTERNAL_FORMAT is COMPRESSED_LUMINANCE_LACT1_EXT,
COMPRESSED_SIGNED_RED_RGTC1_EXT, COMPRESSED_RED_GREEN_RGTC2_EXT,
or COMPRESSED_SIGNED_RED_GREEN_RGTC2_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 RGTC 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 CompressedTexSubImage1D,
DELETE: CompressedTexSubImage2D, or CompressedTexSubImage3D 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

4 new state values are added for the per-texture object
GL_TEXTURE_INTERNAL_FORMAT state.

In the "Textures" state table(page 278), increment the
TEXTURE_INTERNAL_FORMAT subscript for Z by 4 in the "Type" row.

[NOTE: The OpenGL 2.0 specification actually should read "n x Z48*"
because of the 6 generic compressed internal formats in table 3.18.]

New Implementation Dependent State

None

Appendix

RGTC Compressed Texture Image Formats

Compressed texture images stored using the RGTC compressed image
encodings 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 RGTC image has a width or height

less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC 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 RGTC image, the block containing the texel at offset (<x>, <y>) begins at an offset (in bytes) relative to the base of the image of:

    blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).

The data corresponding to a specific texel (<x>, <y>) are extracted from a 4x4 texel block using a relative (x,y) value of

    (<x> modulo 4, <y> modulo 4).

There are four distinct RGTC image formats:

COMPRESSED_RED_RGTC1: Each 4x4 block of texels consists of 64 bits of unsigned red image data.

Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

    red0, red1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5

The 6 "bits_*" bytes of the block are decoded into a 48-bit bit vector:

```
    bits   = bits_0 +
             256 * (bits_1 +
                    256 * (bits_2 +
                           256 * (bits_3 +
                                  256 * (bits_4 +
                                         256 * bits_5))))
```

red0 and red1 are 8-bit unsigned integers that are unpacked to red values RED0 and RED1 as though they were pixels with a <format> of LUMINANCE and a type of UNSIGNED_BTYE.

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)+2..3*(4*y+x)+0]

where bit 47 is the most significant and bit 0 is the least significant bit.

The red value R for a texel at location (x,y) in the block is given by:

```
        RED0,              if red0 > red1 and code(x,y) == 0
        RED1,              if red0 > red1 and code(x,y) == 1
        (6*RED0+  RED1)/7, if red0 > red1 and code(x,y) == 2
        (5*RED0+2*RED1)/7, if red0 > red1 and code(x,y) == 3
        (4*RED0+3*RED1)/7, if red0 > red1 and code(x,y) == 4
        (3*RED0+4*RED1)/7, if red0 > red1 and code(x,y) == 5
        (2*RED0+5*RED1)/7, if red0 > red1 and code(x,y) == 6
        (  RED0+6*RED1)/7, if red0 > red1 and code(x,y) == 7

        RED0,              if red0 <= red1 and code(x,y) == 0
        RED1,              if red0 <= red1 and code(x,y) == 1
        (4*RED0+  RED1)/5, if red0 <= red1 and code(x,y) == 2
        (3*RED0+2*RED1)/5, if red0 <= red1 and code(x,y) == 3
        (2*RED0+3*RED1)/5, if red0 <= red1 and code(x,y) == 4
        (  RED0+4*RED1)/5, if red0 <= red1 and code(x,y) == 5
        MINRED,            if red0 <= red1 and code(x,y) == 6
        MAXRED,            if red0 <= red1 and code(x,y) == 7
```

    MINRED and MAXRED are 0.0 and 1.0 respectively.

Since the decoded texel has a red format, the resulting RGBA value for the texel is (R,0,0,1).

COMPRESSED_SIGNED_RED_RGTC1: Each 4x4 block of texels consists of 64 bits of signed red image data. The red values of a texel are extracted in the same way as COMPRESSED_RED_RGTC1 except red0, red1, RED0, RED1, MINRED, and MAXRED are signed values defined as follows:

    red0 and red1 are 8-bit signed (two's complement) integers.

```
           { red0 / 127.0, red0 > -128
    RED0 = {
           { -1.0,         red0 == -128

           { red1 / 127.0, red1 > -128
    RED1 = {
           { -1.0,         red1 == -128

    MINRED = -1.0

    MAXRED =  1.0
```

CAVEAT for signed red0 and red1 values: the expressions "red0 > red1" and "red0 <= red1" above are considered undefined (read: may vary by implementation) when red0 equals -127 and red1 equals -128, This is because if red0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed LA formats should avoid encoding blocks where red0 equals -127 and red1 equals -128.

COMPRESSED_RED_GREEN_RGTC2: Each 4x4 block of texels consists of 64 bits of compressed unsigned red image data followed by 64 bits of compressed unsigned green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_RED_RGTC1 above except the decoded value R for this second block is considered the resulting green value G.

Since the decoded texel has a red-green format, the resulting RGBA value for the texel is (R,G,0,1).

COMPRESSED_SIGNED_RED_GREEN_RGTC2: Each 4x4 block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above except the decoded value R for this second block is considered the resulting green value G.

Since this image has a red-green format, the resulting RGBA value is (R,G,0,1).

Issues

1) What should these new formats be called?

RESOLVED: "rgtc" for Red-Green Texture Compression.

2) How should the uncompressed and filtered texels be returned by texture fetches?

RESOLVED: Red values show up as (R,0,0,1) where R is the red value, green and blue are forced to 0, and alpha is forced to 1. Likewise, red-green values show up as (R,G,0,1) where G is the green value.

Prior extensions such as NV_float_buffer and NV_texture_shader have introduced formats such as GL_FLOAT_R_NV and GL_DSDT_NV where one- and two-component texture formats show up as (X,0,0,1) or (X,Y,0,1) RGBA texels. The RGTC formats mimic these two-component formats.

The (X,Y,0,1) convention, particularly with signed components, is nice for normal maps because a normalized vector can be formed by a shader program by computing sqrt(abs(1-X*X-Y*Y)) for the Z component.

While GL_RED is a valid external format, core OpenGL provides no GL_RED_GREEN external format. Applications can either use GL_RGB or GL_RGBA and pad out the blue and alpha components, or use the two-component GL_LUMINANCE_ALPHA color format and use the color matrix functionality to swizzle the luminance and alpha values into red and green respectively.

3) Should red and red-green compression formats with signed components be introduced when the core specification lacked uncompressed red and red-green texture formats?

RESOLVED: Yes, signed red and red-green compression formats should be added.

Signed red-green formats are suited for compressed normal maps. Compressed normal maps may well be the dominant use of this extension.

Unsigned red-green formats require an extra "expand normal" operation to convert [0,1] to [-1,+1]. Direct support for signed red-green formats avoids this step in a shader program.

4) Should there be a mix of signed red and unsigned green or vice versa?

RESOLVED: No.

NV_texture_shader provided an internal format (GL_SIGNED_RGB_UNSIGNED_ALPHA_NV) with mixed signed and unsigned components. The format saw little usage. There's no reason to think a GL_SIGNED_RED_UNSIGNED_GREEN format would be any more useful or popular.

5) How are signed integer values mapped to floating-point values?

RESOLVED: A signed 8-bit two's complement value X is computed to a floating-point value Xf with the formula:

```
          { X / 127.0, X > -128
    Xf = {
          { -1.0,      X == -128
```

This conversion means -1, 0, and +1 are all exactly representable, however -128 and -127 both map to -1.0. Mapping -128 to -1.0 avoids the numerical awkwardness of have a representable value slightly more negative than -1.0.

This conversion is intentionally NOT the "byte" conversion listed in Table 2.9 for component conversions. That conversion says:

```
Xf = (2*X + 1) / 255.0
```

The Table 2.9 conversion is incapable of exactly representing zero.

6) How will signed components resulting from GL_COMPRESSED_SIGNED_RED_RGTC1_EXT and GL_COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT texture fetches interact with fragment coloring?

RESOLVED: The specification language for this extension is silent about clamping behavior leaving this to the core specification and other extensions. The clamping or lack of clamping is left to the core specification and other extensions.

For assembly program extensions supporting texture fetches (ARB_fragment_program, NV_fragment_program, NV_vertex_program3, etc.) or the OpenGL Shading Language, these signed formats will appear as expected with unclamped signed components as a result of a texture fetch instruction.

If ARB_color_buffer_float is supported, its clamping controls will apply.

NV_texture_shader extension, if supported, adds support for fixed-point textures with signed components and relaxed the fixed-function texture environment clamping appropriately. If the NV_texture_shader extension is supported, its specified behavior for the texture environment applies where intermediate values are clamped to [-1,1] unless stated otherwise as in the case of explicitly clamped to [0,1] for GL_COMBINE. or clamping the linear interpolation weight to [0,1] for GL_DECAL and GL_BLEND.

Otherwise, the conventional core texture environment clamps incoming, intermediate, and output color components to [0,1].

This implies that the conventional texture environment functionality of unextended OpenGL 1.5 or OpenGL 2.0 without using GLSL (and with none of the extensions referred to above) is unable to make proper use of the signed texture formats added by this extension because the conventional texture environment requires texture source colors to be clamped to [0,1]. Texture filtering of these signed formats would be still signed, but negative values generated post-filtering would be clamped to zero by the core texture environment functionality. The expectation is clearly that this extension would be co-implemented with one of the previously referred to extensions or used with GLSL for the new signed formats to be useful.

7) *Should a specific normal map compression format be added?*

RESOLVED: No.

It's probably short-sighted to design a format just for normal maps. Indeed, NV_texture_shader added a GL_SIGNED_HILO_NV format with exactly the kind of "hemisphere remap" useful for normal maps and the format went basically unused. Instead, this extension provides the mechanism for compressed normal maps based on the more conventional red-green format.

The GL_COMPRESSED_RED_GREEN_RGTC2_EXT and GL_COMPRESSED_SIGNED_RED_GREEN_RGTC2_EXT formats are sufficient for normal maps with additional shader instructions used to generate the 3rd component.

8) *Should uncompressed signed red and red-green formats be added by this extension?*

RESOLVED: No, this extension is focused on just adding compressed texture formats.

The NV_texture_shader extension adds such uncompressed signed texture formats. A distinct multi-vendor extension for signed fixed-point texture formats could provide all or a subset of the signed fixed-point uncompressed texture formats introduced by NV_texture_shader.

9) *What compression ratios does this extension provide?*

The RGTC1 formats are 8 bytes (64 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8 data requires 16 bytes (1 byte per texel). This is a 2-to-1 compression ratio.

The RGTC2 formats are 16 bytes (128 bits) per 4x4 pixel block. A 4x4 block of GL_LUMINANCE8_ALPHA8 data requires 32 bytes (2 bytes per texel). This is again a 2-to-1 compression ratio.

In contrast, the comparable compression ratio for the S3TC formats is 4-to-1.

Arguably, the lower compression ratio allows better compression quality particularly because the RGTC formats compress each component separately.

10) *How do these new formats compare with the existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats?*

RESOLVED: The existing GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats provide a similar 2-to-1 compression ratio but mandate a uniform quantization for all components. In contrast, this extension provides a compression format with 3-bit quantization over a specifiable min/max range that can vary per 4x4 texel tile.

Additionally, many OpenGL implementations do not natively support the GL_LUMINANCE4, GL_LUMINANCE4_ALPHA4, and GL_LUMINANCE6_ALPHA2 internal formats but rather silently promote these formats to store 8 bits per component, thereby eliminating any storage/bandwidth advantage for these formats.

11) *Does this extension require EXT_texture_compression_s3tc?*

RESOLVED: No.

As written, this specification does not rely on wording of the EXT_texture_compression_s3tc extension. For example, certain discussion added to Sections 3.8.2 and 3.8.3 is quite similar to corresponding EXT_texture_compression_s3tc language.

12) *Should anything be said about the precision of texture filtering for these new formats?*

RESOLVED: No precision requirements are part of the specification language since OpenGL extensions typically leave precision details to the implementation.

Realistically, at least 8-bit filtering precision can be expected from implementations (and probably more).

13) Should these formats be allowed to specify 3D texture images when NV_texture_compression_vtc is supported?

RESOLVED: The NV_texture_compression_vtc stacks 4x4 blocks into 4x4x4 bricks. It may be more desirable to represent compressed 3D textures as simply slices of 4x4 blocks.

However the NV_texture_compression_vtc extension expects data passed to the glCompressedTexImage commands to be "bricked" rather than blocked slices.

14) How is the texture border color handled for the blue component of an RGTC2 texture and the green and blue components of an RGTC1 texture?

RESOLVED: The base texture format is RGB for the RGTC1 and RGTC2 texture formats. This would mean table 3.15 would be used to determine how the texture border color is interpreted and which components are considered.

However since only red or red/green components exist for the RGTC1 and RGTC2 formats, it makes little sense to require the blue component be supplied by the texture border color and hence be involved (meaningfully only when the border is sampled) in texture filtering.

For this reason, a statement is added to section 3.8.8 says that if a texture's internal format lacks components that exist in the texture's base internal format, such components contain zero (ignoring the texture's corresponding texture border color component value) when the texture border color is sampled.

So the green and blue components of the filtered result of a RGTC1 texture are always zero, even when the border is sampled. Similarly the blue component of the filtered result of a RGTC2 texture is always zero, even when the border is sampled.

15) What should glGetTexLevelParameter return for *GL_TEXTURE_GREEN_SIZE and GL_TEXTURE_BLUE_SIZE for the RGTC1 formats? What should glGetTexLevelParameter return for GL_TEXTURE_BLUE_SIZE for the RGTC2 formats?*

RESOLVED: Zero bits.

These formats always return 0.0 for these respective components and have no bits devoted to these components.

Returning 8 bits for red size of RGTC1 and the red and green sizes of RGTC2 makes sense because that's the maximum potential precision for the uncompressed texels.

16) Should the token names contain R and RG or RED and RED_GREEN?

RESOLVED: RED and RED_GREEN.

Saying RGB and RGBA makes sense for three- and four-component formats rather than spelling out the component names because RGB and RGBA are used so commonly and spelling out the names it too wordy.

But for 1- and 2-component names, we follow the precedent by GL_LUMINANCE and GL_LUMINANCE_ALPHA. This extension spells out the component names of 1- and 2-component names.

Another reason to avoid R and RG is the existing meaning of the GL_R and GL_RED tokens. GL_RED already exists as a token name for a single-component external format. GL_R also already exists as a token name but refers to the R texture coordinate, not the red color component.

17) Can you use the GL_RED external format with glTexImage2D and other such commands to load textures with the GL_COMPRESSED_RED_RGTC1_EXT or GL_COMPRESSED_SIGNED_RED_RGTC1_EXT internal formats?

RESOLVED: Yes.

GL_RED has been a valid external format parameter to glTexImage and similar commands since OpenGL 1.0.

18) Should any of the generic compression GL_COMPRESSED_* tokens in OpenGL 2.1 map to RGTC formats?

RESOLVED: No. The RGTC formats are missing color components so are not adequate implementations for any of the generic compression formats.

19) Should the GL_NUM_COMPRESSED_TEXTURE_FORMATS and GL_COMPRESSED_TEXTURE_FORMATS queries return the RGTC formats?

RESOLVED: No.

The OpenGL 2.1 specification says "The only values returned by this query [GL_COMPRESSED_TEXTURE_FORMATS"] are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use."

Compressed textures with just red or red-green components are not general-purpose so should not be returned by these queries because they have restrictions.

Applications that seek to use the RGTC formats should do so by looking for this extension's name in the string returned by glGetString(GL_EXTENSIONS) rather than what GL_NUM_COMPRESSED_TEXTURE_FORMATS and GL_COMPRESSED_TEXTURE_FORMATS return.

Revision History

```
    Revision 1.1, April 24, 2007: mjk
        -  Add caveat about how signed LA decompression happens when
           lum0 equals -127 and lum1 equals -128.  This caveat matches
           a decoding allowance in DirectX 10.

    Revision 1.2, January 21, 2008: mjk
        -  Add issues #18 and #19.
```

Name

EXT_texture_integer

Name Strings

GL_EXT_texture_integer

Contact

Michael Gold, NVIDIA Corporation (gold 'at' nvidia.com)
Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 07/15/2006
NVIDIA Revision: 5

Number

343

Dependencies

OpenGL 2.0 is required.

NV_gpu_program4 or EXT_gpu_shader4 is required.

ARB_texture_float affects the definition of this extension.

ARB_color_buffer_float affects the definition of this extension.

EXT_framebuffer_object affects the definition of this extension.

This extension is written against the OpenGL 2.0 specification.

Overview

Fixed-point textures in unextended OpenGL have integer components, but those values are taken to represent floating-point values in the range [0,1]. These integer components are considered "normalized" integers. When such a texture is accessed by a shader or by fixed-function fragment processing, floating-point values are returned.

This extension provides a set of new "unnormalized" integer texture formats. Formats with both signed and unsigned integers are provided. In these formats, the components are treated as true integers. When such textures are accessed by a shader, actual integer values are returned.

Pixel operations that read from or write to a texture or color buffer with unnormalized integer components follow a path similar to that used for color index pixel operations, except that more

than one component may be provided at once. Integer values flow through the pixel processing pipe, and no pixel transfer operations are performed. Integer format enumerants used for such operations indicate unnormalized integer data.

Textures or render buffers with unnormalized integer formats may also be attached to framebuffer objects to receive fragment color values written by a fragment shader. Per-fragment operations that require floating-point color components, including multisample alpha operations, alpha test, blending, and dithering, have no effect when the corresponding colors are written to an integer color buffer. The NV_gpu_program4 and EXT_gpu_shader4 extensions add the capability to fragment programs and fragment shaders to write signed and unsigned integer output values.

This extension does not enforce type consistency for texture accesses or between fragment shaders and the corresponding framebuffer attachments. The results of a texture lookup from an integer texture are undefined:

* for fixed-function fragment processing, or

* for shader texture accesses expecting floating-point return values.

The color components used for per-fragment operations and written into a color buffer are undefined:

* for fixed-function fragment processing with an integer color buffer,

* for fragment shaders that write floating-point color components to an integer color buffer, or

* for fragment shaders that write integer color components to a color buffer with floating point or normalized integer components.

New Procedures and Functions

```
void ClearColorIiEXT ( int r, int g, int b, int a );
void ClearColorIuiEXT ( uint r, uint g, uint b, uint a );
void TexParameterIivEXT( enum target, enum pname, int *params );
void TexParameterIuivEXT( enum target, enum pname, uint *params );
void GetTexParameterIivEXT ( enum target, enum pname, int *params);
void GetTexParameterIuivEXT ( enum target, enum pname, uint *params);
```

New Tokens

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

RGBA_INTEGER_MODE_EXT	0x8D9E

```
Accepted by the <internalFormat> parameter of TexImage1D,
TexImage2D, and TexImage3D:

    RGBA32UI_EXT                                0x8D70
    RGB32UI_EXT                                 0x8D71
    ALPHA32UI_EXT                               0x8D72
    INTENSITY32UI_EXT                           0x8D73
    LUMINANCE32UI_EXT                           0x8D74
    LUMINANCE_ALPHA32UI_EXT                     0x8D75

    RGBA16UI_EXT                                0x8D76
    RGB16UI_EXT                                 0x8D77
    ALPHA16UI_EXT                               0x8D78
    INTENSITY16UI_EXT                           0x8D79
    LUMINANCE16UI_EXT                           0x8D7A
    LUMINANCE_ALPHA16UI_EXT                     0x8D7B

    RGBA8UI_EXT                                 0x8D7C
    RGB8UI_EXT                                  0x8D7D
    ALPHA8UI_EXT                                0x8D7E
    INTENSITY8UI_EXT                            0x8D7F
    LUMINANCE8UI_EXT                            0x8D80
    LUMINANCE_ALPHA8UI_EXT                      0x8D81

    RGBA32I_EXT                                 0x8D82
    RGB32I_EXT                                  0x8D83
    ALPHA32I_EXT                                0x8D84
    INTENSITY32I_EXT                            0x8D85
    LUMINANCE32I_EXT                            0x8D86
    LUMINANCE_ALPHA32I_EXT                      0x8D87

    RGBA16I_EXT                                 0x8D88
    RGB16I_EXT                                  0x8D89
    ALPHA16I_EXT                                0x8D8A
    INTENSITY16I_EXT                            0x8D8B
    LUMINANCE16I_EXT                            0x8D8C
    LUMINANCE_ALPHA16I_EXT                      0x8D8D

    RGBA8I_EXT                                  0x8D8E
    RGB8I_EXT                                   0x8D8F
    ALPHA8I_EXT                                 0x8D90
    INTENSITY8I_EXT                             0x8D91
    LUMINANCE8I_EXT                             0x8D92
    LUMINANCE_ALPHA8I_EXT                       0x8D93
```

Accepted by the <format> parameter of TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, DrawPixels and ReadPixels:

RED_INTEGER_EXT	0x8D94
GREEN_INTEGER_EXT	0x8D95
BLUE_INTEGER_EXT	0x8D96
ALPHA_INTEGER_EXT	0x8D97
RGB_INTEGER_EXT	0x8D98
RGBA_INTEGER_EXT	0x8D99
BGR_INTEGER_EXT	0x8D9A
BGRA_INTEGER_EXT	0x8D9B
LUMINANCE_INTEGER_EXT	0x8D9C
LUMINANCE_ALPHA_INTEGER_EXT	0x8D9D

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.6.4 (Rasterization of Pixel Rectangles), p. 126:

(modify the last paragraph, p. 126)
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 twenty type token values and the GL data types they indicate is given in table 3.5. 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. If the GL is in RGBA mode and the color buffer is an integer format and no fragment shader is active, 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 one of the integer component formats as defined in table 3.6, and <type> is FLOAT, then the error INVALID_ENUM occurs. Some additional constraints on the combinations of format and type values that are accepted is discussed below.

(add the following to table 3.6, p. 129)

Format Name	Element Meaning and Order	Target Buffer
RED_INTEGER_EXT	iR	Color
GREEN_INTEGER_EXT	iG	Color
BLUE_INTEGER_EXT	iB	Color
ALPHA_INTEGER_EXT	iA	Color
RGB_INTEGER_EXT	iR, iG, iB	Color
RGBA_INTEGER_EXT	iR, iG, iB, iA	Color
BGR_INTEGER_EXT	iB, iG, iR	Color
BGRA_INTEGER_EXT	iB, iG, iR, iA	Color
LUMINANCE_INTEGER_EXT	iLuminance	Color
LUMINANCE_ALPHA_INTEGER_EXT	iLuminance, iA	Color

Table 3.6: DrawPixels and ReadPixels formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components. Components are floating-point unless prefixed with the letter 'i' which indicates they are integer.

(modify first paragraph, p. 129)
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, or four values, depending on the format, to form a group. Table 3.6 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

(modify the last paragraph, p. 135)
Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components.

(modify the third paragraph, p. 136)
Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1 for integer components or 1.0 for floating-point components. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0 for integer components or 0.0 for floating-point components.

(modify the last paragraph, p. 136)
Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by 2^n - 1, where n is the number of bits in an index buffer. For floating-point RGBA components, each element is clamped to [0, 1]. The resulting values are converted to fixed-point according to the rules given in section 2.14.9 (Final Color Processing). For integer RGBA

components, no conversion is applied. For a depth component, an element is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.11.1, Controlling the Viewport). Stencil indices are masked by 2^n - 1, where n is the number of bits in the stencil buffer.

Modify Section 3.6.5 (Pixel Transfer Operations), p. 137

(modify last paragraph, p. 137)
The GL defines five kinds of pixel groups:

1. Floating-point RGBA component: Each group comprises four color components in floating point format: red, green, blue, and alpha.

2. Integer RGBA component: Each group comprises four color components in integer format: red, green, blue, and alpha.

3. Depth component: Each group comprises a single depth component.

4. Color index: Each group comprises a single color index.

5. Stencil index: Each group comprises a single stencil index.

(modify second paragraph, p. 138)
Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped. None of the operations defined in this section affect integer RGBA component pixel groups.

Modify Section 3.8 (Texturing), p. 149

(insert between the first and second paragraphs, p. 150)
The internal data type of a texture may be fixed-point, floating-point, signed integer or unsigned integer, depending on the internalformat of the texture. The correspondence between internalformat and the internal data type is given in table 3.16. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. When a fragment shader is active, the shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined. Fixed functionality assumes floating-point data, hence the result of using fixed functionality with integer textures is undefined.

Modify Section 3.8.1 (Texture Image Specification), p. 150

(modify second paragraph, p. 151) The selected groups are processed exactly as for DrawPixels, stopping just before final conversion. If the <internalformat> of the texture is integer, the components are clamped to the representable range of the internal format: for signed formats, this is [-2^(n-1), 2^(n-1)-1] where n is the number of bits per component; for unsigned formats, the range is [0, 2^n-1]. For R, G, B, and A, if the

<internalformat> of the texture is fixed-point, the components are clamped to [0, 1]. Otherwise, the components are not modified.

(insert between paragraphs five and six, p. 151)
Textures with integer internal formats (table 3.16) require integer data. The error INVALID_OPERATION is generated if the internal format is integer and <format> is not one of the integer formats listed in table 3.6, or if the internal format is not integer and <format> is an integer format, or if <format> is an integer format and <type> is FLOAT.

(add the following to table 3.16, p. 154)

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
ALPHA8I_EXT	ALPHA				i8		
ALPHA8UI_EXT	ALPHA				ui8		
ALPHA16I_EXT	ALPHA				i16		
ALPHA16UI_EXT	ALPHA				ui16		
ALPHA32I_EXT	ALPHA				i32		
ALPHA32UI_EXT	ALPHA				ui32		
LUMINANCE8I_EXT	LUMINANCE					i8	
LUMINANCE8UI_EXT	LUMINANCE					ui8	
LUMINANCE16I_EXT	LUMINANCE					i16	
LUMINANCE16UI_EXT	LUMINANCE					ui16	
LUMINANCE32I_EXT	LUMINANCE					i32	
LUMINANCE32UI_EXT	LUMINANCE					ui32	
LUMINANCE_ALPHA8I_EXT	LUMINANCE_ALPHA				i8	i8	
LUMINANCE_ALPHA8UI_EXT	LUMINANCE_ALPHA				ui8	ui8	
LUMINANCE_ALPHA16I_EXT	LUMINANCE_ALPHA				i16	i16	
LUMINANCE_ALPHA16UI_EXT	LUMINANCE_ALPHA				ui16	ui16	
LUMINANCE_ALPHA32I_EXT	LUMINANCE_ALPHA				i32	i32	
LUMINANCE_ALPHA32UI_EXT	LUMINANCE_ALPHA				ui32	ui32	
INTENSITY8I_EXT	INTENSITY						i8
INTENSITY8UI_EXT	INTENSITY						ui8
INTENSITY16I_EXT	INTENSITY						i16
INTENSITY16UI_EXT	INTENSITY						ui16
INTENSITY32I_EXT	INTENSITY						i32
INTENSITY32UI_EXT	INTENSITY						ui32
RGB8I_EXT	RGB	i8	i8	i8			
RGB8UI_EXT	RGB	ui8	ui8	ui8			
RGB16I_EXT	RGB	i16	i16	i16			
RGB16UI_EXT	RGB	ui16	ui16	ui16			
RGB32I_EXT	RGB	i32	i32	i32			
RGB32UI_EXT	RGB	ui32	ui32	ui32			
RGBA8I_EXT	RGBA	i8	i8	i8	i8		
RGBA8UI_EXT	RGBA	ui8	ui8	ui8	ui8		
RGBA16I_EXT	RGBA	i16	i16	i16	i16		
RGBA16UI_EXT	RGBA	ui16	ui16	ui16	ui16		
RGBA32I_EXT	RGBA	i32	i32	i32	i32		
RGBA32UI_EXT	RGBA	ui32	ui32	ui32	ui32		

Table 3.16: Correspondence of sized internal formats to base internal formats, internal data type and desired component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: <f> is

floating point, <i> is signed integer, <ui> is unsigned integer, and no prefix is fixed-point.

Modify Section 3.8.2 (Alternate Texture Image Specification Commands), p. 159:

(modify the second paragraph, p. 159)
The error INVALID_OPERATION is generated if depth component data is required and no depth buffer is present, or if integer RGBA data is required and the format of the current color buffer is not integer, or if floating-point or fixed-point RGBA data is required and the format of the current color buffer is integer.

Modify Section 3.8.4 (Texture Parameters), p. 166:

Various parameters control how the texture array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

```
    void TexParameter{if}( enum target, enum pname, T param );
    void TexParameter{if}v( enum target, enum pname, T params );
    void TexParameterIivEXT( enum target, enum pname, int *params );
    void TexParameterIuivEXT( enum target, enum pname, uint *params );
```

<target> is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP. <pname> is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.19. In the first form of the command, <param> is a value to which to set a single-valued parameter; in the second and third forms of the command, <params> is an array of parameters whose type depends on the parameter being set.

If the value for TEXTURE_PRIORITY is specified as an integer, the conversion for signed integers from table 2.9 is applied to convert the value to floating-point. The floating point value of TEXTURE_PRIORITY is clamped to lie in [0, 1].

If the values for TEXTURE_BORDER_COLOR are specified with TexParameterIivEXT or TexParameterIuivEXT, the values are unmodified and stored with an internal data type of integer. If specified with TexParameteriv, the conversion for signed integers from table 2.9 is applied to convert these values to floating-point. Otherwise the values are unmodified and stored as floating-point.

(modify table 3.19, p. 167)

Name	Type	Legal Values
TEXTURE_BORDER_COLOR	4 floats or 4 ints or 4 uints	any 4 values

Table 3.19: Texture parameters and their values.

Modify Section 3.8.8 (Texture Minification), p. 170

(modify last paragraph, p. 174)

... If the texture contains color components, the values of TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. The internal data type of the border values must be consistent with the type returned by the texture as described in section 3.8, or the result is undefined. The border values for texture components stored as fixed-point values are clamped to [0, 1] before they are used. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value

Modify Section 3.8.10 (Texture Completeness), p. 177:

(add to the requirements for one-, two-, or three-dimensional textures)
If the internalformat is integer, TEXTURE_MAG_FILTER must be NEAREST and TEXTURE_MIN_FILTER must be NEAREST or NEAREST_MIPMAP_NEAREST.

Modify Section 3.11.2 (Shader Execution), p. 194

(modify Shader Outputs, first paragraph, p. 196)
... These are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled and the color buffer has a fixed-point or floating-point format, the final fragment color values or the final fragment data values written by a fragment shader are clamped to the range [0, 1]. If fragment clamping is disabled or the color buffer has an integer format, the final fragment color values or the final fragment data values are not modified. The final fragment depth...

(insert between the first paragraph and second paragraphs of "Shader Outputs", p. 196)
Colors values written by the fragment shader may be floating-point, signed integer or unsigned integer. If the color buffer has a fixed-point format, the color values are assumed to be floating-point and are converted to fixed-point as described in section 2.14.9; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Chapter 4 Introduction, (p. 198)

(modify third paragraph, p. 198)
Color buffers consist of unsigned integer color indices, R, G, B and optionally A floating-point components represented as fixed-point unsigned integer or floating-point values, or R, G, B and optionally A integer components represented as signed or unsigned integer values. The number of bitplanes...

Modify Section 4.1.3 (Multisample Fragment Operations), p. 200

(modify the second paragraph in this section)
... If SAMPLE_ALPHA_TO_COVERAGE is enabled and the color buffer has a fixed-point or floating-point format, a temporary coverage value is generated ...

Modify Section 4.1.4 (Alpha Test), p. 201

(modify the first paragraph in this section)
This step applies only in RGBA mode and only if the color buffer has a fixed-point or floating-point format. In color index mode or if the color buffer has an integer format, proceed to the next operation. The alpha test discards ...

Modify Section 4.1.8 (Blending), p. 205

(modify the second paragraph, p. 206)
... Blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode and only if the color buffer has a fixed-point or floating-point format; in color index mode or if the color buffer has an integer format, it is bypassed. ...

Modify Section 4.2.3 (Clearing the Buffers), p. 215

```
void ClearColor(float r, float g, float b, float a);
```

sets the clear value for fixed-point and floating-point color buffers in RGBA mode. The specified components are stored as floating-point values.

```
void ClearColorIiEXT(int r, int g, int b, int a);
void ClearColorIuiEXT(uint r, uint g, uint b, uint a);
```

set the clear value for signed integer and unsigned integer color buffers, respectively, in RGBA mode. The specified components are stored as integer values.

(add to the end of first partial paragraph, p. 217) ... then a Clear directed at that buffer has no effect. When fixed-point RGBA color buffers are cleared, the clear color values are assumed to be floating-point and are clamped to [0,1] before being converted to fixed-point according to the rules of section 2.14.9. The result of clearing fixed-point or floating-point color buffers is undefined if the clear color was specified as integer values. The result of when clearing integer color buffers is undefined if the clear color was specified as floating-point values.

Modify Section 4.3.2 (Reading Pixels), p. 219

(append to the last paragraph, p. 221)
The error INVALID_OPERATION occurs if <format> is an integer format and the color buffer is not an integer format, or if the color buffer is an integer format and <format> is not. The error INVALID_ENUM occurs if <format> is an integer format and <type> is FLOAT.

(modify the first paragraph, p. 222)
... For a fixed-point color buffer, each element is taken to be a fixed-point value in [0, 1] with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.14.9). For an integer or floating-point color buffer, the elements are unmodified.

(modify the section labeled "Conversion to L", p. 222)
This step applies only to RGBA component groups. If the format is either LUMINANCE or LUMINANCE_ALPHA, a value L is computed as

```
L = R + G + B
```

otherwise if the format is either LUMINANCE_INTEGER_EXT or LUMINANCE_ALPHA_INTEGER_EXT, L is computed as

```
L = R
```

where R, G, and B are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

(modify the section labeled "Final Conversion", p. 222)

For a floating-point RGBA color, each component is first clamped to [0, 1]. Then the appropriate conversion formula from table 4.7 is applied to the component. For an integer RGBA color, each component is clamped to the representable range of <type>.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.3 (Enumerated Queries), p. 246

(insert in the list of query functions, p. 246)
```
void GetTexParameterIivEXT( enum target, enum value, int *data );
void GetTexParameterIuivEXT( enum target, enum value, uint *data );
```

(modify the second paragraph, p. 247)
... For GetTexParameter, value must be either TEXTURE_RESIDENT, or one of the symbolic values in table 3.19. Querying <value> TEXTURE_BORDER_COLOR with GetTexParameterIivEXT or GetTexParameterIuivEXT returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined. The <lod> argument ...

(add to end of third paragraph, p. 247) Queries with a <value> of TEXTURE_RED_TYPE_ARB, TEXTURE_GREEN_TYPE_ARB, TEXTURE_BLUE_TYPE_ARB, TEXTURE_ALPHA_TYPE_ARB, TEXTURE_LUMINANCE_TYPE_ARB, TEXTURE_INTENSITY_TYPE_ARB, or TEXTURE_DEPTH_TYPE_ARB, return the data type used to store the component. Values of NONE, UNSIGNED_NORMALIZED_ARB, FLOAT, INT, or UNSIGNED_INT, indicate missing,

unsigned normalized integer, floating-point, signed unnormalized integer, and unsigned unnormalized integer components, respectively.

GLX Protocol

TBD

Dependencies on ARB_texture_float

The following changes should be made if ARB_texture_float is not supported:

The references to floating-point data types in section 3.8, p. 150 should be deleted.

The language in section 3.8.1 should indicate that final conversion always clamps when the internalformat is not integer.

The description of table 3.16 should not mention the <f> floating-point formats.

Section 3.8.4 should indicate that border color values should be clamped to [0,1] before being stored, if not specified with one of the TexParameterI* functions.

Section 3.8.8 should not mention clamping border color values to [0,1] for fixed-point textures, since this occurs in 3.8.4 at TexParameter specification.

Dependencies on ARB_color_buffer_float

The following changes should be made if ARB_color_buffer_float is not supported:

Section 3.11.2, subsection "Shader Outputs: p. 196 should not mention fragment clamping or color buffers with floating-point formats.

Chapter 4, p. 198 should not mention components represented as floating-point values.

Section 4.1.3, p. 200, section 4.1.4 p. 205, section 4.1.8 p. 206, section 4.2.3 p. 215 and section 4.3.2 p. 222 should not mention color buffers with a floating-point format.

Section 4.2.3 p. 217 should not mention clamping the clear color values to [0,1].

Errors

INVALID_OPERATION is generated by Begin, DrawPixels, Bitmap, CopyPixels, or a command that performs an explicit Begin if the color buffer has an integer RGBA format and no fragment shader is active.

INVALID_ENUM is generated by DrawPixels, TexImage* and SubTexImage* if <format> is one of the integer component formats

described in table 3.6 and <type> is FLOAT.

INVALID_OPERATION is generated by TexImage* and SubTexImage* if the texture internalformat is an integer format as described in table 3.16 and <format> is not one of the integer component formats described in table 3.6, or if the internalformat is not an integer format and <format> is an integer format.

INVALID_OPERATION is generated by CopyTexImage* and CopyTexSubImage* if the texture internalformat is an integer format and the read color buffer is not an integer format, or if the internalformat is not an integer format and the read color buffer is an integer format.

INVALID_ENUM is generated by ReadPixels if <format> is an integer format and <type> is FLOAT.

INVALID_OPERATON is generated by ReadPixels if <format> is an integer format and the color buffer is not an integer format, or if <format> is not an integer format and the color buffer is an integer format.

New State

(modify table 6.33, p. 294)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
RGBA_INTEGER_MODE_EXT	B	GetBooleanv	-	True if RGBA components are integers	2.7	-

Issues

How should the integer pixel path be triggered: by the destination type, new source types, or new source formats?

RESOLVED: New source formats, based on the precedence of COLOR_INDEX and STENCIL_INDEX formats which invoke distinct pixel path behavior with identical data types and independent of the destination.

Should pixel transfer operations be defined for the integer pixel path?

RESOLVED: No. Fragment shaders can achieve similar results with more flexibility. There is no need to aggrandize this legacy mechanism.

What happens if a shader reads a float texel from an integer texture or vice-versa?

RESOLVED: The result is undefined. The shader must have knowledge of the texture internal data type.

How do integer textures behave in fixed function fragment processing?

RESOLVED: The fixed function texture pipeline assumes textures return floating-point values, hence the return value from an integer texture will not be in a meaningful format.

How does TEXTURE_BORDER_COLOR work with integer textures?

RESOLVED: The internal storage of border values effectively becomes a union, and the returned values are interpreted as the same type as the texture. New versions of TexParameter allow specification of signed and unsigned integer border values.

How does logic op behave with RGBA mode rendering into integer color buffer?

RESOLVED: The color logic op operates when enabled when rendering into integer color buffers.

Logic op operations make sense for integer color buffers so the COLOR_LOGIC_OP enable is respected when rendering into integer color buffers.

Blending does not apply to RGBA mode rendering when rendering into integer color buffers (as section 4.1.8 is updated to say). The color logic op (described in section 4.1.10) is not a blending operation (though it does take priority over the blending enable).

Revision History

Rev.	Date	Author	Changes
5	07/15/07	pbrown	Fix typo in GetTexParameterIuivEXT function name in "New Procedures and Functions".
4	--		Pre-release revisions.

Name

EXT_texture_shared_exponent

Name Strings

GL_EXT_texture_shared_exponent

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Contributors

Pat Brown
Jon Leech

Status

Shipping for GeForce 8 Series (November 2006)

Version

Date: February 6, 2007
Revision: 0.5

Number

333

Dependencies

OpenGL 1.1 required

ARB_color_buffer_float affects this extension.

EXT_framebuffer_object affects this extension.

This extension is written against the OpenGL 2.0 (September 7, 2004) specification.

Overview

Existing texture formats provide either fixed-point formats with limited range and precision but with compact encodings (allowing 32 or fewer bits per multi-component texel), or floating-point formats with tremendous range and precision but without compact encodings (typically 16 or 32 bits per component).

This extension adds a new packed format and new internal texture format for encoding 3-component vectors (typically RGB colors) with a single 5-bit exponent (biased up by 15) and three 9-bit mantissas for each respective component. There is no sign bit so all three components must be non-negative. The fractional mantissas are stored without an implied 1 to the left of the decimal point. Neither infinity nor not-a-number (NaN) are representable in this shared exponent format.

This 32 bits/texel shared exponent format is particularly well-suited to high dynamic range (HDR) applications where light intensity is typically stored as non-negative red, green, and blue components with considerable range.

New Procedures and Functions

None

New Tokens

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

RGB9_E5_EXT	0x8C3D

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable:

UNSIGNED_INT_5_9_9_9_REV_EXT	0x8C3E

Accepted by the <pname> parameter of GetTexLevelParameterfv and GetTexLevelParameteriv:

TEXTURE_SHARED_SIZE_EXT	0x8C3F

Additions to Chapter 2 of the 2.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 2.0 Specification (Rasterization)

-- Section 3.6.4, Rasterization of Pixel Rectangles

Add a new row to Table 3.5 (page 128):

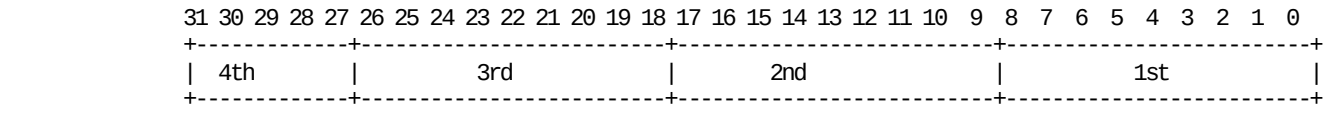
type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_INT_5_9_9_9_REV_EXT	uint	yes

Add a new row to table 3.8: Packed pixel formats (page 132):

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_INT_5_9_9_9_REV_EXT	uint	4	RGB

Add a new entry to table 3.11: UNSIGNED_INT formats (page 134):

UNSIGNED_INT_5_9_9_9_REV_EXT:



Add to the end of the 2nd paragraph starting "Pixels are draw using":

"If type is UNSIGNED_INT_5_9_9_9_REV_EXT and format is not RGB then the error INVALID_ENUM occurs."

Add UNSIGNED_INT_5_9_9_9_REV_EXT to the list of packed formats in the 10th paragraph after the "Packing" subsection (page 130).

Add before the 3rd paragraph (page 135, starting "Calling DrawPixels with a type of BITMAP...") from the end of the "Packing" subsection:

"Calling DrawPixels with a type of UNSIGNED_INT_5_9_9_9_REV_EXT and format of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 4 packed components as shown in table 3.11. The 1st, 2nd, 3rd, and 4th components are called p_red, p_green, p_blue, and p_exp respectively and are treated as unsigned integers. These are then used to compute floating-point RGB components (ignoring the "Conversion to floating-point" section below in this case) as follows:

```
red   = p_red   * 2^(p_exp - B)
green = p_green * 2^(p_exp - B)
blue  = p_blue  * 2^(p_exp - B)
```

where B is 15."

-- Section 3.8.1, Texture Image Specification:

"Alternatively if the internalformat is RGB9_E5_EXT, the red, green, and blue bits are converted to a shared exponent format according to the following procedure:

Components red, green, and blue are first clamped (in the process, mapping NaN to zero) so:

```
red_c   = max(0, min(sharedexp_max, red))
green_c = max(0, min(sharedexp_max, green))
blue_c  = max(0, min(sharedexp_max, blue))
```

where sharedexp_max is (2^N-1)/2^N * 2^(Emax-B), N is the number of mantissa bits per component, Emax is the maximum allowed biased exponent value (careful: not necessarily 2^E-1 when E is the number of exponent bits), bits, and B is the exponent bias. For the RGB9_E5_EXT format, N=9, Emax=30 (careful: not 31!), and B=15.

The largest clamped component, max_c, is determined:

```
max_c = max(red_c, green_c, blue_c)
```

A shared exponent is computed:

```
exp_shared = max(-B-1, floor(log2(max_c))) + 1 + B
```

These integers values in the range 0 to 2^N-1 are then computed:

```
red_s   = floor(red_c   / 2^(exp_shared - B + N) + 0.5)
green_s = floor(green_c / 2^(exp_shared - B + N) + 0.5)
blue_s  = floor(blue_c  / 2^(exp_shared - B + N) + 0.5)
```

Then red_s, green_s, and blue_s are stored along with exp_shared in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of type UNSIGNED_INT_5_9_9_9_REV_EXT with a format of RGB is allowed to store the components "as is" if the implementation can determine the current pixel transfer state act as an identity transform on the components."

Add a new row and the "shared bits" column (blank for all existing rows) to Table 3.16 (page 154).

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits	D bits	shared bits
RGB9_E5_EXT	RGB	9	9	9					5

-- Section 3.8.x, Shared Exponent Texture Color Conversion

Insert this section AFTER section 3.8.14 Texture Comparison Modes and BEFORE section 3.8.15 Texture Application (and after the "sRGB Texture Color Conversion" if EXT_texture_sRGB is supported).

"If the currently bound texture's internal format is RGB9_E5_EXT, the red, green, blue, and shared bits are converted to color components (prior to filtering) using the following shared exponent decoding.

The components red_s, green_s, blue_s, and exp_shared values (see section 3.8.1) are treated as unsigned integers and are converted to red, green, blue as follows:

```
red   = red_s   * 2^(exp_shared - B)
green = green_s * 2^(exp_shared - B)
blue  = blue_s  * 2^(exp_shared - B)"
```

Additions to Chapter 4 of the 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

-- Section 4.3.2, Reading Pixels

Add a row to table 4.7 (page 224);

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_INT_5_9_9_9_REV_EXT	uint	special

Replace second paragraph of "Final Conversion" (page 222) to read:

For an RGBA color, if <type> is not FLOAT or UNSIGNED_INT_5_9_9_9_REV_EXT, or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color (or texture) buffer is a fixed-point buffer, each component is first clamped to [0,1]. Then the appropriate conversion formula from table 4.7 is applied the component.

In the special case when calling ReadPixels with a type of UNSIGNED_INT_5_9_9_9_REV_EXT and format of RGB, the conversion is done as follows: The returned data are packed into a series of GL uint values. The red, green, and blue components are converted to red_s, green_s, blue_s, and exp_shared integers as described in section 3.8.1 when the internalformat is RGB9_E5_EXT. The red_s, green_s, blue_s, and exp_shared are then packed as the 1st, 2nd, 3rd, and 4th components of the UNSIGNED_INT_5_9_9_9_REV_EXT format as shown in table 3.11."

Additions to Chapter 5 of the 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 2.0 Specification (State and State Requests)

-- Section 6.1.3, Enumerated Queries

Add TEXTURE_SHARED_SIZE_EXT to the list of queries in the first sentence of the fifth paragraph (page 247) so it reads:

"For texture images with uncompressed internal formats, queries of value of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_DEPTH_SIZE, TEXTURE_SHARED_SIZE_EXTT, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined."

Additions to the OpenGL Shading Language specification

None

Additions to the GLX Specification

None

GLX Protocol

None.

Dependencies on ARB_color_buffer_float

If ARB_color_buffer_float is not supported, replace this amended sentence from 4.3.2 above

"For an RGBA color, if <type> is not FLOAT or UNSIGNED_INT_5_9_9_9_REV_EXT, or if the CLAMP_READ_COLOR_ARB is TRUE, or CLAMP_READ_COLOR_ARB is FIXED_ONLY_ARB and the selected color buffer

```
    (or texture image for GetTexImage) is a fixed-point buffer (or texture
    image for GetTexImage), each component is first clamped to [0,1]."

    with

    "For an RGBA color, if <type> is not FLOAT or
    UNSIGNED_INT_5_9_9_9_REV_EXT and the selected color buffer (or
    texture image for GetTexImage) is a fixed-point buffer (or texture
    image for GetTexImage), each component is first clamped to [0,1]."
```

Dependencies on EXT_framebuffer_object

```
    If EXT_framebuffer_object is not supported, then
    RenderbufferStorageEXT is not supported and the RGB9_E5_EXT
    internalformat is therefore not supported by RenderbufferStorageEXT.
```

Errors

```
    Relaxation of INVALID_ENUM errors
    ---------------------------------

    TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D,
    and RenderbufferStorageEXT accept the new RGB9_E5_EXT token for
    internalformat.

    DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage,
    TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D,
    GetHistogram, GetMinmax, ConvolutionFilter1D, ConvolutionFilter2D,
    ConvolutionFilter3D, GetConvolutionFilter, SeparableFilter2D,
    GetSeparableFilter, ColorTable, ColorSubTable, and GetColorTable
    accept the new UNSIGNED_INT_5_9_9_9_REV_EXT token for type.

    GetTexLevelParameterfv and GetTexLevelParameteriv accept the new
    TEXTURE_SHARED_SIZE_EXT token for <pname>.

    New errors
    ----------

    INVALID_OPERATION is generated by DrawPixels, ReadPixels, TexImage1D,
    TexImage2D, GetTexImage, TexImage3D, TexSubImage1D, TexSubImage2D,
    TexSubImage3D, GetHistogram, GetMinmax, ConvolutionFilter1D,
    ConvolutionFilter2D, ConvolutionFilter3D, GetConvolutionFilter,
    SeparableFilter2D, GetSeparableFilter, ColorTable, ColorSubTable,
    and GetColorTable if <type> is UNSIGNED_INT_5_9_9_9_REV_EXT
    and <format> is not RGB.
```

New State

```
    In table 6.17, Textures (page 278), increment the 42 in "n x Z42*"
    by 1 for the RGB9_E5_EXT format.

    [NOTE: The OpenGL 2.0 specification actually should read "n x Z48*"
    because of the 6 generic compressed internal formats in table 3.18.]
```

Add the following entry to table 6.17:

Get Value	Type	Get Command	Value	Description	Sec.	Attribute
TEXTURE_SHARED_SIZE_EXT	n x Z+	GetTexLevelParameter	0	xD texture image i's shared exponent field size	3.8	-

New Implementation Dependent State

None

Appendix

This source code provides ANSI C routines. It assumes the C "float" data type is stored with the IEEE 754 32-bit floating-point format. Make sure you define __LITTLE_ENDIAN or __BIG_ENDIAN appropriate for your target system.

XXX: code below not tested on big-endian platform...

```
------------------- start of source code ------------------------

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define __LITTLE_ENDIAN  1
#define __BIG_ENDIAN     2

#ifdef _WIN32
#define __BYTE_ORDER __LITTLE_ENDIAN
#endif

#define RGB9E5_EXPONENT_BITS          5
#define RGB9E5_MANTISSA_BITS          9
#define RGB9E5_EXP_BIAS               15
#define RGB9E5_MAX_VALID_BIASED_EXP   31

#define MAX_RGB9E5_EXP               (RGB9E5_MAX_VALID_BIASED_EXP - RGB9E5_EXP_BIAS)
#define RGB9E5_MANTISSA_VALUES       (1<<RGB9E5_MANTISSA_BITS)
#define MAX_RGB9E5_MANTISSA          (RGB9E5_MANTISSA_VALUES-1)
#define MAX_RGB9E5        ((float)MAX_RGB9E5_MANTISSA)/RGB9E5_MANTISSA_VALUES * (1<<MAX_RGB9E5_EXP))
#define EPSILON_RGB9E5          ((1.0/RGB9E5_MANTISSA_VALUES) / (1<<RGB9E5_EXP_BIAS))

typedef struct {
#ifdef __BYTE_ORDER
#if __BYTE_ORDER == __BIG_ENDIAN
  unsigned int negative:1;
  unsigned int biasedexponent:8;
  unsigned int mantissa:23;
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  unsigned int mantissa:23;
  unsigned int biasedexponent:8;
  unsigned int negative:1;
#endif
#endif
} BitsOfIEEE754;

typedef union {
  unsigned int raw;
  float value;
  BitsOfIEEE754 field;
} float754;
```

```
typedef struct {
#ifdef __BYTE_ORDER
#if __BYTE_ORDER == __BIG_ENDIAN
  unsigned int biasedexponent:RGB9E5_EXPONENT_BITS;
  unsigned int b:RGB9E5_MANTISSA_BITS;
  unsigned int g:RGB9E5_MANTISSA_BITS;
  unsigned int r:RGB9E5_MANTISSA_BITS;
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  unsigned int r:RGB9E5_MANTISSA_BITS;
  unsigned int g:RGB9E5_MANTISSA_BITS;
  unsigned int b:RGB9E5_MANTISSA_BITS;
  unsigned int biasedexponent:RGB9E5_EXPONENT_BITS;
#endif
#endif
} BitsOfRGB9E5;

typedef union {
  unsigned int raw;
  BitsOfRGB9E5 field;
} rgb9e5;

float ClampRange_for_rgb9e5(float x)
{
  if (x > 0.0) {
    if (x >= MAX_RGB9E5) {
      return MAX_RGB9E5;
    } else {
      return x;
    }
  } else {
    /* NaN gets here too since comparisons with NaN always fail! */
    return 0.0;
  }
}

float MaxOf3(float x, float y, float z)
{
  if (x > y) {
    if (x > z) {
      return x;
    } else {
      return z;
    }
  } else {
    if (y > z) {
      return y;
    } else {
      return z;
    }
  }
}

/* Ok, FloorLog2 is not correct for the denorm and zero values, but we
   are going to do a max of this value with the minimum rgb9e5 exponent
   that will hide these problem cases. */
int FloorLog2(float x)
{
  float754 f;

  f.value = x;
  return (f.field.biasedexponent - 127);
}

int Max(int x, int y)
{
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

```
rgb9e5 float3_to_rgb9e5(const float rgb[3])
{
  rgb9e5 retval;
  float maxrgb;
  int rm, gm, bm;
  float rc, gc, bc;
  int exp_shared;
  double denom;

  rc = ClampRange_for_rgb9e5(rgb[0]);
  gc = ClampRange_for_rgb9e5(rgb[1]);
  bc = ClampRange_for_rgb9e5(rgb[2]);

  maxrgb = MaxOf3(rc, gc, bc);
  exp_shared = Max(-RGB9E5_EXP_BIAS-1, FloorLog2(maxrgb)) + 1 + RGB9E5_EXP_BIAS;
  assert(exp_shared <= RGB9E5_MAX_VALID_BIASED_EXP);
  assert(exp_shared >= 0);
  /* This pow function could be replaced by a table. */
  denom = pow(2, exp_shared - RGB9E5_EXP_BIAS - RGB9E5_MANTISSA_BITS);

  rm = (int) floor(rc / denom + 0.5);
  gm = (int) floor(gc / denom + 0.5);
  bm = (int) floor(bc / denom + 0.5);

  assert(rm <= MAX_RGB9E5_MANTISSA);
  assert(gm <= MAX_RGB9E5_MANTISSA);
  assert(bm <= MAX_RGB9E5_MANTISSA);
  assert(rm >= 0);
  assert(gm >= 0);
  assert(bm >= 0);

  retval.field.r = rm;
  retval.field.g = gm;
  retval.field.b = bm;
  retval.field.biasedexponent = exp_shared;

  return retval;
}

void rgb9e5_to_float3(rgb9e5 v, float retval[3])
{
  int exponent = v.field.biasedexponent - RGB9E5_EXP_BIAS - RGB9E5_MANTISSA_BITS;
  float scale = (float) pow(2, exponent);

  retval[0] = v.field.r * scale;
  retval[1] = v.field.g * scale;
  retval[2] = v.field.b * scale;
}

------------------- end of source code ------------------------
```

Issues

1) What should this extension be called?

RESOLVED: EXT_texture_shared_exponent

The "EXT_texture" part indicates the extension is in the texture domain and "shared_exponent" indicates the extension is adding a new shared exponent formats.

EXT_texture_rgb9e5 was considered but there's no precedent for extension names to be so explicit (or cryptic?) about format specifics in the extension name.

2) There are many possible encodings for a shared exponent format. Which encoding does this extension specify?

RESOLVED: A single 5-bit exponent stored as an unsigned value biased by 15 and three 9-bit mantissas for each of 3 components. There are no sign bits so all three components must be non-negative. The fractional mantissas assume an implied 0 left of the decimal point because having an implied leading 1 is inconsistent with sharing the exponent. Neither Infinity nor Not-a-Number (NaN) are representable in this shared exponent format.

We chose this format because it closely matches the range and precision of the s10e5 half-precision floating-point described in the ARB_half_float_pixel and ARB_texture_float specifications.

3) Why not an 8-bit shared exponent?

RESOLVED: Greg Ward's RGBE shared exponent encoding uses an 8-bit exponent (same as a single-precision IEEE value) but we believe the rgb9e5 is more generally useful than rgb8e8.

An 8-bit exponent provides far more range than is typically required for graphics applications. However, an extra bit of precision for each component helps in situations where a high magnitude component dominates a low magnitude component. Having an 8-bit shared exponent and 8-bit mantissas are amenable to CPUs that facilitate 8-bit sized reads and writes over non-byte aligned fields, but GPUs do not suffer from this issue.

Indeed GPUs with s10e5 texture filtering can use that same filtering hardware for rgb9e5 textures.

However, future extensions could add other shared exponent formats so we name the tokens to indicate the

4) Should there be an external format and type for rgb9e5?

RESOLVED: Yes, hence the external format GL_RGB9_E5_EXT and type GL_UNSIGNED_INT_5_9_9_9_REV_EXT. This makes it fast to load GL_RGB9_E5_EXT textures without any translation by the driver.

5) Why is the exponent bias 15?

RESOLVED: The best technical choice of 15. Hopefully, this discussion sheds insight into the numerics of the shared exponent format in general.

With conventional floating-point formats, the number corresponding to a finite, non-denorm, non-zero floating-point value is

```
value = -1^sgn * 2^(exp-bias) * 1.frac
```

where sgn is the sign bit (so 1 for sgn negative because -1^-1 == -1 and 0 means positive because -1^0 == +1), exp is an (unsigned) BIASED exponent and bias is the format's constant bias to subtract to get the unbiased (possibly negative) exponent;

and frac is the fractional portion of the mantissa with the "1." indicating an implied leading 1.

An exp value of zero indicates so-called denormalized values (denorms). With conventional floating-point formats, the number corresponding to a denorm floating-point value is

```
value = -1^sgn * 2^(exp-bias+1) * 0.frac
```

where the only difference between the denorm and non-denorm case is the bias is one greater in the denorm case and the implied leading digit is a zero instead of a one.

Ideally, the rgb9e5 shared exponent format would represent roughly the same range of finite values as the s10e5 format specified by the ARB_texture_float extension. The s10e5 format has an exponent bias of 15.

While conventional floating-point formats cleverly use an implied leading 1 for non-denorm, finite values, a shared exponent format cannot use an implied leading 1 because each component may have a different magnitude for its most-significant binary digit. The implied leading 1 assumes we have the flexibility to adjust the mantissa and exponent together to ensure an implied leading 1. That flexibility is not present when the exponent is shared.

So the rgb9e5 format cannot assume an implied leading one. Instead, an implied leading zero is assumed (much like the conventional denorm case).

The rgb9e5 format eliminate support representing negative, Infinite, not-a-number (NaN), and denorm values.

We've already discussed how the BIASED zero exponent is used to encode denorm values (and zero) with conventional floating-point formats. The largest BIASED exponent (31 for s10e5, 127 for s23e8) indicates Infinity and NaN values. This means these two extrema exponent values are "off limits" for run-of-the-mill values.

The numbers corresponding to a shared exponent format value are:

```
value_r = 2^(exp-bias) * 0.frac_r
value_g = 2^(exp-bias) * 0.frac_g
value_b = 2^(exp-bias) * 0.frac_b
```

where there is no sgn since all values are non-negative, exp is the (unsigned) BIASED exponent and bias is the format's constant bias to subtract to get the unbiased (possibly negative) exponent; and frac_r, frac_g, and frac_b are the fractional portion of the mantissas of the r, g, and b components respectively with "0." indicating an implied leading 0.

There should be no "off limits" exponents for the shared exponent format since there is no requirement for representing Infinity or NaN values and denorm is not a special case. Because of

the implied leading zero, any component with all zeros for its mantissa is zero, no matter the shared exponent's value.

So the run-of-the-mill BIASED range of exponents for s10e5 is 1 to 30. But the rgb9e5 shared exponent format consistently uses the same rule for all exponents from 0 to 31.

What exponent bias best allows us to represent the range of s10e5 with the rgb9e5 format? 15.

Consider the maximum representable finite s10e5 magnitude. The exponent would be 30 (31 would encode an Infinite or NaN value) and the binary mantissa would be 1 followed by ten fractional 1's. Effectively:

```
s10e5_max  =  1.1111111111 * 2^(30-15)
           =  1.1111111111 * 2^15
```

For an rgb9e5 value with a bias of 15, the largest representable value is:

```
rgb9e5_max =  0.111111111  * 2^(31-15)
           =  0.111111111  * 2^16
           =  1.11111111   * 2^15
```

If you ignore two LSBs, these values are nearly identical. The rgb9e5_max value is exactly representable as an s10e5 value.

For an rgb9e5 value with a bias of 15, the smallest non-zero representable value is:

```
rgb9e5_min =  0.000000001  * 2^(0-15)
rgb9e5_min =  0.000000001  * 2^-15
rgb9e5_min =  0.0000000001 * 2^-14
```

So the s10e5_min and rgb9e5_min values exactly match (of course, this assumes the shared exponent bias is 15 which might not be the case if other components demand higher exponents).

8) Should there be an rgb9e5 framebuffer format?

RESOLVED: No. Rendering to rgb9e5 is better left to another extension and would require the hardware to convert from a (floating-point) RGBA value into an rgb9e5 encoding.

Interactions with EXT_framebuffer_object are specified, but the expectation is this is not a renderable format and glCheckFramebufferStatusEXT would return GL_FRAMEBUFFER_UNSUPPORTED_EXT.

An implementation certainly could make this texture internal format renderable when used with a framebuffer object. Note that the shared exponent means masked components may be lossy in their masking. For example, a very small but non-zero value in a masked component could get flushed to zero if a large enough value is written into an unmasked component.

9) Should automatic mipmap generation be supported for rgb9e5 textures?

RESOLVED: Yes.

10) Should non-texture and non-framebuffer commands for loading pixel data accept the GL_UNSIGNED_INT_5_9_9_9_REV_EXT type?

RESOLVED: Yes.

Once the pixel path has to support the new type/format combination of GL_UNSIGNED_INT_5_9_9_9_REV_EXT / GL_RGB for specifying and querying texture images, it might as well be supported for all commands that pack and unpack RGB pixel data.

The specification is written such that the glDrawPixels type/format parameters are accepted by glReadPixels, glTexGetImage, glTexImage2D, and other commands that are specified in terms of glDrawPixels.

11) Should non-texture internal formats (such as for color tables, convolution kernels, histogram bins, and min/max tables) accept GL_RGB9_E5_EXT format?

RESOLVED: No.

That's pointless. No hardware is ever likely to support GL_RGB9_E5_EXT internalformats for anything other than textures and maybe color buffers in the future. This format is not interesting for color tables, convolution kernels, etc.

12) Should a format be supported with sign bits for each component?

RESOLVED: No.

An srgb8e5 format with a sign bit per component could be useful but is better left to another extension.

13) The rgb9e5 allows two 32-bit values encoded as rgb9e5 to correspond to the exact same 3 components when expanded to floating-point. Is this a problem?

RESOLVED: No, there's no problem here.

An encoder is likely to always pack components so at least one mantissa will have an explicit leading one, but there's no requirement for that.

Applications might be able to take advantage of this by quickly dividing all three components by a power-of-two by simply subtracting log2 of the power-of-two from the shared exponent (as long as the exponent is greater than zero prior to the subtract).

Arguably, the shared exponent format could maintain a slight amount of extra precision (one bit per mantissa) if the format said if the most significant bits of all three mantissas are either all one or all zero and the biased shared exponent was not

zero, then an implied leading 1 should be assumed and the shared exponent should be treated as one smaller than it really is. While this would preserve an extra least-significant bit of mantissa precision for components of approximately the same magnitude, it would complicate the encoding and decoding of shared exponent values.

14) *Can you provide some C code for encoding three floating-point values into the rgb9e5 format?*

RESOLVED: Sure. See the Appendix.

15) *Should we support a non-REV version of the GL_UNSIGNED_INT_5_9_9_9_REV_EXT token?*

RESOLVED: No. The shared exponent is always the 5 most significant bits of the 32 bit word. The first (red) mantissa is in the least significant 9 bits, followed by 9 bits for the second (green) mantissa, followed by 9 bits for the third (blue) mantissa. We don't want to promote different arrangements of the bitfields for rgb9e5 values.

16) *Can you use the GL_UNSIGNED_INT_5_9_9_9_REV_EXT format with just any format?*

RESOLVED: You can only use the GL_UNSIGNED_INT_5_9_9_9_REV_EXT format with GL_RGB. Otherwise, the GL generates an GL_INVALID_OPERATION error. Conceptually, GL_UNSIGNED_INT_5_9_9_9_REV_EXT is a 3-component format that just happens to have 5 shared bits too. Just as the GL_UNSIGNED_BYTE_3_3_2 format just works with GL_RGB (or else the GL generates an GL_INVALID_OPERATION error), so should GL_UNSIGNED_INT_5_9_9_9_REV_EXT.

17) *What should GL_TEXTURE_SHARED_SIZE_EXT return when queried with GetTexLevelParameter?*

RESOLVED: Return 5 for the RGB9_E5_EXT internal format and 0 for all other existing formats.

This is a count of the number of bits in the shared exponent.

18) *What should GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, and GL_TEXTURE_BLUE_SIZE return when queried with GetTexLevelParameter for a GL_RGB9_E5_EXT texture?*

RESOLVED: Return 9 for each.

Revision History

None

Name

NV_conditional_render

Name Strings

GL_NV_conditional_render

Status

Shipping.

Version

Last Modified Date: 11/29/2007
NVIDIA Revision: 2

Number

Unassigned.

Dependencies

The extension is written against the OpenGL 2.0 Specification.

ARB_occlusion_query or OpenGL 1.5 is required.

Overview

This extension provides support for conditional rendering based on the results of an occlusion query. This mechanism allows an application to potentially reduce the latency between the completion of an occlusion query and the rendering commands depending on its result. It additionally allows the decision of whether to render to be made without application intervention.

This extension defines two new functions, BeginConditionalRenderNV and EndConditionalRenderNV, between which rendering commands may be discarded based on the results of an occlusion query. If the specified occlusion query returns a non-zero value, rendering commands between these calls are executed. If the occlusion query returns a value of zero, all rendering commands between the calls are discarded.

If the occlusion query results are not available when BeginConditionalRenderNV is executed, the <mode> parameter specifies whether the GL should wait for the query to complete or should simply render the subsequent geometry unconditionally.

Additionally, the extension provides a set of "by region" modes, allowing for implementations that divide rendering work by screen regions to perform the conditional query test on a region-by-region basis without checking the query results from other regions. Such a mode is useful for cases like split-frame SLI, where a frame is divided between multiple GPUs, each of which has its own occlusion query hardware.

New Procedures and Functions

```
    void BeginConditionalRenderNV(uint id, enum mode);
    void EndConditionalRenderNV(void);
```

New Tokens

Accepted by the <mode> parameter of BeginConditionalRenderNV:

QUERY_WAIT_NV	0x8E13
QUERY_NO_WAIT_NV	0x8E14
QUERY_BY_REGION_WAIT_NV	0x8E15
QUERY_BY_REGION_NO_WAIT_NV	0x8E16

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

(Incorporate the spec edits from the EXT_transform_feedback specification that move the "Occlusion Queries" Section 4.1.7 -- to between Section 2.11, Coordinate Transforms and Section 2.12, Clipping, and rename it to "Asynchronous Queries". Insert a new section immediately after the moved "Asynchronous Queries" section. If EXT_transform_feedback is incorporated, this section should be inserted prior the the "Transform Feedback" section.)

Section 2.X, Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```
    void BeginConditionalRenderNV(uint id, enum mode);
    void EndConditionalRenderNV(void);
```

<id> specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (SAMPLES_PASSED) of the query is zero, all rendering commands between BeginConditionalRenderNV and the corresponding EndConditionalRenderNV are discarded. In this case, Begin, End, all vertex array commands performing an implicit Begin and End, DrawPixels (section 3.6), Bitmap (section 3.7), Clear (section 4.2.3), Accum (section 4.2.4), CopyPixels (section 4.3.3), EvalMesh1, and EvalMesh2 (section 5.1) have no effect. The effect of commands setting current vertex state (e.g., Color, VertexAttrib) is undefined. If the result of the occlusion query is non-zero, such commands are not discarded.

<mode> specifies how BeginConditionalRenderNV interprets the results of the occlusion query given by <id>. If <mode> is QUERY_WAIT_NV, the GL waits for the results of the query to be available and then uses the results to determine if subsquent rendering commands are discarded. If <mode> is QUERY_NO_WAIT_NV, the GL may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If <mode> is QUERY_BY_REGION_WAIT_NV, the GL will also wait for occlusion query results and discard rendering commands if the result of the occlusion query is zero. If the query result is non-zero, subsequent rendering commands are executed, but the GL may discard the results of the

commands for any region of the framebuffer that did not contribute to the sample count in the specified occlusion query. Any such discarding is done in an implementation-dependent manner, but the rendering command results may not be discarded for any samples that contributed to the occlusion query sample count. If <mode> is QUERY_BY_REGION_NO_WAIT_NV, the GL operates as in QUERY_BY_REGION_WAIT_NV, but may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If BeginConditionalRenderNV is called while conditional rendering is in progress, or if EndConditionalRenderNV is called while conditional rendering is not in progress, the error INVALID_OPERATION is generated. The error INVALID_VALUE is generated if <id> is not the name of an existing query object query. The error INVALID_OPERATION is generated if <id> is the name of a query object with a target other than SAMPLES_PASSED, or <id> is the name of a query currently in progress.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

TBD.

Errors

INVALID_OPERATION is generated by BeginConditionalRenderNV if a previous BeginConditionalRenderNV command has been executed without a corresponding EndConditionalRenderNV command.

INVALID_OPERATION is generated by EndConditionalRenderNV if no corresponding BeginConditionalRenderNV command has been executed.

INVALID_VALUE is generated by BeginConditionalRenderNV if <id> is not the name of an existing occlusion query object.

INVALID_OPERATION is generated by BeginConditionalRenderNV if <id> is the name of a query object with a <target> other than SAMPLES_PASSED.

INVALID_OPERAITON is generated by BeginConditionalRenderNV if the query identified by <id> is still in progress.

Issues

(1) How should rendering commands other than "normal" Begin/End-style geometry be affected by conditional rendering?

RESOLVED: All rendering commands (DrawPixels, Bitmap, Clear, Accum, etc...) are performed conditionally.

(2) What does NO_WAIT do, and why would anyone care?

RESOLVED: Hardware OpenGL implementations are heavily pipelined. After vertices are transformed, they are assembled into primitives and rasterized. While a GPU is rasterizing a primitive, it may be simultaneously transforming the vertices of the next primitive provided to the GL. At the same time, the CPU may be preparing hardware commands to process primitives following that one.

Conditional rendering uses the results of rasterizing one primitive (an occlusion query) to determine whether it will process subsequent ones. In a pipelined implementation, the initial set of primitives may not be finished drawing by the time the GL needs the occlusion query results. Waiting for the query results will leave portions of the GPU temporarily idle. It may be preferable to avoid the idle time by proceeding with a conservative assumption that the primitives rendered during the occlusion query will hit at least one sample. The NO_WAIT <mode> parameter tells the driver move ahead in that case.

For best performance, applications should attempt to insert some amount of non-dependent rendering between an occlusion query and the conditionally-rendered primitives that depend on the query result.

(3) What does BY_REGION do, and why should anyone care?

RESOLVED: Conditional rendering may be used for a variety of effects. Some of these use conditional rendering only for performance. One common use would be to draw a bounding box for a primitive unconditionally with an occlusion query active, and then conditionally execute a DrawElements call to draw the full (complex) primitive. If the bounding box is not visible, any work needed to process the full primitive can be skipped in the conditional rendering pass.

In a split-screen SLI implementation, one GPU might draw the top half of the scene while a second might draw the bottom half. The results of the occlusion query would normally be obtained by combining individual occlusion query results from each half of the screen. However, it is not necessary to do this for the bounding box algorithm. We could skip this synchronization point, and each region could instead use only its local occlusion query results. If the bounding box hits only the bottom

half of the screen, the complex primitive need not be drawn on the top half, because that portion is known not to be visible. The bottom half would still be drawn, but the GPU used for the top half could skip it and start drawing the next primitive specified. The QUERY_BY_REGION_*_NV modes would be useful in that case.

However, some algorithms may require conditional rendering for correctness. For example, an application may want to render a post-processing effect that should be drawn if and only if a point is visible in the scene. Drawing only half of such an effect due to BY_REGION tests would not be desirable.

For QUERY_BY_REGION_NO_WAIT_NV, we expect that GL implementations using region-based rendering will discard rendering commands in any region where query results are available and the region's sample count is zero. Rendering would proceed normally in all other regions. The spec language doesn't require such behavior, however.

(4) Should the <mode> parameter passed to BeginConditionalRenderNV be specified as a hint instead?

RESOLVED: The "wait" or "don't wait" portion of the <mode> parameter could be a hint. But it doesn't fit nicely with the FASTEST or NICEST values that are normally passed to Hint. Providing this functionality via a <mode> parameter to BeginConditionalRenderNV seems to make the most sense. Note that the <mode> parameter is specified such that QUERY_NO_WAIT_NV can be implemented as though QUERY_WAIT_NV were specified, which makes the "NO_WAIT" part of the mode a hint.

The "BY_REGION" part is also effectively a hint. These modes may be implemented as though the equivalent non-BY_REGION mode were provided. Many OpenGL implementations will do all of their processing in a single region.

(5) What happens if BeginQuery is called while the specified occlusion query is begin used for conditional rendering?

RESOLVED: An INVALID_OPERATION error is generated.

(6) Should conditional rendering work with any type of query other than SAMPLES_PASSED (occlusion)?

RESOLVED: Not in this extension. The spec currently requires that <id> be the name of an occlusion query. There might be other query types where such an operation would make sense, but there aren't any in the current OpenGL spec.

(7) What is the effect on current state for immediate mode attribute calls (e.g., Color, VertexAttrib) made during conditional rendering if the corresponding occlusion query failed?

RESOLVED: The effect of these calls is undefined. If subsequent primitives depend on a vertex attribute set inside a conditional rendering block, and application should re-send the values after EndConditionalRenderNV.

(8) Should we provide any new query object types for conditional rendering?

RESOLVED: No. It may be useful to some GL implementations to provide an occlusion query type that only returns "zero" or "non-zero", or to provide a query type that is used only for conditional rendering but doesn't have to maintain results that can be returned to the application. However, performing conditional rendering using only the occlusion query mechanisms already in core OpenGL is sufficient for the platforms targeted by this extension.

(9) What happens if QUERY_BY_REGION_* is used, and the application switches between windows or FBOs between the occlusion query and conditional rendering blocks? The "regions" used for the two operations may not be identical.

RESOLVED: The spec language doesn't specifically address this issue, and implementations may choose to define regions arbitrarily in this case.

We strongly recommend that applications using QUERY_BY_REGION_* should not change windows or FBO configuration between the occlusion query and the dependent rendering.

Usage Example

```
GLuint queryID = 0x12345678;

// Use an occlusion query while rendering the bounding box of the real
// object.
glBeginQuery(GL_SAMPLES_PASSED, queryID);
   drawBoundingBox();
glEndQuery(GL_SAMPLES_PASSED);

// Do some unrelated rendering in hope that the query result will be
// available by the time we call glBeginConditionalRenderNV.

// Now conditionally render the real object if any portion of its
// bounding box is visible.
glBeginConditionalRenderNV(queryID, GL_QUERY_WAIT_NV);
   drawComplicatedObject();
glEndConditionalRenderNV();
```

Revision History

Rev.	Date	Author	Changes
2	11/29/07	ewerness	First public release
1			Internal revisions

Name

NV_depth_buffer_float

Name Strings

GL_NV_depth_buffer_float

Contributors

Pat Brown
Mike Strauss

Contact

Mike Strauss, NVIDIA Corporation (mstrauss 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 11/06/2006
NVIDIA Revision: 8

Number

334

Dependencies

OpenGL 2.0 is required.

ARB_color_buffer_float is required.

EXT_packed_depth_stencil is required.

EXT_framebuffer_object is required.

This extension modifies EXT_depth_bounds_test.

This extension modifies NV_copy_depth_to_color.

This extension is written against the OpenGL 2.0 specification.

Overview

This extension provides new texture internal formats whose depth components are stored as 32-bit floating-point values, rather than the normalized unsigned integers used in existing depth formats. Floating-point depth textures support all the functionality supported for fixed-point depth textures, including shadow mapping and rendering support via EXT_framebuffer_object. Floating-point depth textures can store values outside the range [0,1].

By default, OpenGL entry points taking depth values implicitly clamp the values to the range [0,1]. This extension provides new DepthClear, DepthRange, and DepthBoundsEXT entry points that allow applications to specify depth values that are not clamped.

Additionally, this extension provides new packed depth/stencil pixel formats (see EXT_packed_depth_stencil) that have 64-bit pixels consisting of a 32-bit floating-point depth value, 8 bits of stencil, and 24 unused bites. A packed depth/stencil texture internal format is also provided.

This extension does not provide support for WGL or GLX pixel formats with floating-point depth buffers. The existing (but not commonly used) WGL_EXT_depth_float extension could be used for this purpose.

New Procedures and Functions

```
void DepthRangedNV(double n, double f);
void ClearDepthdNV(double d);
void DepthBoundsdNV(double zmin, double zmax);
```

New Tokens

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, CopyTexImage2D, and RenderbufferStorageEXT, and returned in the <data> parameter of GetTexLevelParameter and GetRenderbufferParameterivEXT:

DEPTH_COMPONENT32F_NV	0x8DAB
DEPTH32F_STENCIL8_NV	0x8DAC

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D, TexSubImage2D, TexSubImage3D, and GetTexImage:

FLOAT_32_UNSIGNED_INT_24_8_REV_NV	0x8DAD

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

DEPTH_BUFFER_FLOAT_MODE_NV	0x8DAF

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.11.1 (Controling the Viewport), p. 41

(modify second paragraph) The factor and offset applied to z_d encoded by n and f are set using

```
void DepthRange(clampd n, clampd f);
void DepthRangedNV(double n, double f);
```

z_w is represented as either fixed-point or floating-point depending on whether the framebuffer's depth buffer uses fixed-point or floating-point representation. If the depth buffer uses fixed-point representation, we assume that the representation used represents each value k/(2^m - 1), where k is in {0,1,...,2^m-1}, as k (e.g. 1.0 is represented in binary as a

string of all ones). The parameters n and f are clamped to [0, 1] when using DepthRange, but not when using DepthRangedNV. When n and f are applied to z_d, they are clamped to the range appropriate given the depth buffer's representation.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

Modify Section 3.5.5 (Depth Offset), p. 112

(modify third paragraph) The minimum resolvable difference r is an implementation dependent parameter that depends on the depth buffer representation. It is the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_w values that differ by r, will have distinct depth values.

For fixed-point depth buffer representations, r is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, e, in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

r = 2^(e - n). (3.11)

(modify fourth paragraph) The offset value o for a polygon is

o = m * factor + r * units. (3.12)

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range [0, 1], and o is applied to depth values in the same range.

(modify last paragraph) For fixed-point depth buffers, fragment depth values are always limited to the range [0, 1], either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygons. Fragment depth values are not clamped when the depth buffer uses a floating-point representation.

Add a row to table 3.5, p. 128

type Parameter	GL Type	Special
...	...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	N/A	Yes
...	...	...

Modify Section 3.6.4 (Rasterization of Pixel Rectangles), p. 128

(modify second paragraph as updated by EXT_packed_depth_stencil) ... If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL_EXT,

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_EXT and <type> is not UNSIGNED_INT_24_8_EXT or FLOAT_32_UNSIGNED_INT_24_8_REV_NV, then the error INVALID_ENUM occurs. Some additional constraints on the combinations of <format> and <type> values that are accepted are discussed below.

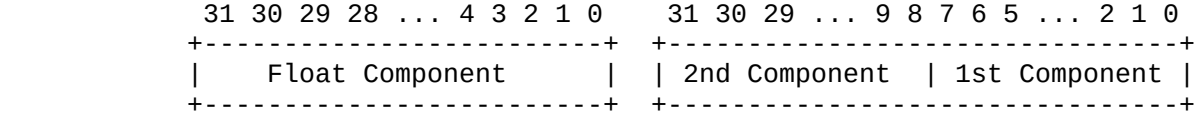
(modify fifth paragraph of "Unpacking," p 130. as updated by EXT_packed_depth_stencil) Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, ..., UNSIGNED_INT_2_10_10_10_REV, or UNSIGNED_INT_24_8_EXT is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If <type> is FLOAT_32_UNSIGNED_INT_24_8_REV_NV, the components of each group are two 32-bit words. The first word contains the float component. The second word contains packed 24-bit and 8-bit components.

Add two rows to table 3.8, p. 132

type Parameter	GL Type	Components	Pixel Formats
...	...	...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	N/A	2	DEPTH_STENCIL_EXT
...	...	...	...

Add a row to table 3.11, p. 134

FLOAT_32_UNSIGNED_INT_24_8_REV_NV:



(modify last paragraph of "Final Conversion," p. 136) For a depth component, an element is processed according to the depth buffer's representation. For fixed-point depth buffers, the element is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.11.1, Controling the Viewport). Clamping and conversion are not necessary when the depth buffer uses a floating-point representation.

Modify Section 3.8.1 (Texture Image Specification), p. 150

(modify the second paragraph, p. 151, as modified by ARB_color_buffer_float) The selected groups are processed exactly as for DrawPixels, stopping just before final conversion. Each R, G, B, A, or depth value so generated is clamped based on the component type in the <internalFormat>. Fixed-point components are clamped to [0, 1]. Floating-point components are clamped to the limits of the range representable by their format. 32-bit floating-point components are in the standard IEEE float format. 16-bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits. Stencil index values are masked by 2^n-1 where n is the number of stencil bits in the internal format resolution (see below). If the base internal format is

DEPTH_STENCIL_EXT and <format> is not DEPTH_STENCIL_EXT, then the values of the stencil index texture components are undefined.

Add two rows to table 3.16, p. 154

Sized Internal Format	Base InternalFormat	R bits	G bits	B bits	A bits	L bits	I bits	D bits	S bits
...	...	...	...	...	...	...	...	...	...
DEPTH_COMPONENT32F_NV	DEPTH_COMPONENT							f32	
DEPTH32F_STENCIL8_NV	DEPTH_STENCIL_EXT							f32	8
...	...	...	...	...	...	...	...	...	...

Modify Section 3.8.14 (Texture Comparison Modes), p. 185

(modify second paragraph of "Depth Texture Comparison Mode," p. 188) Let D_t be the depth texture value, and R be the interpolated texture coordinate. If the texture's internal format indicates a fixed-point depth texture, then D_t and R are clamped to [0, 1], otherwise no clamping is performed. The effective texture value L_t, I_t, or A_t is computed as follows:

Modify Section 3.11.2 (Shader Execution), p. 194

(modify first paragraph of "Shader Outputs," p, 196, as modified by ARB_color_buffer_float) The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled, the final fragment color values or the final fragment data values written by a fragment shader are clamped to the range [0, 1] and then may be converted to fixed-point as described in section 2.14.9. If fragment clamping is disabled, the final fragment color values or the final fragment data values are not modified. For fixed-point depth buffers the final fragment depth written by a fragment shader is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.11.1). Clamping and conversion are not applied for floating-point depth buffers. Note that the depth range computation is not applied here.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

(modify third paragraph in the introduction, p. 198, as modified by ARB_color_buffer_float) Color buffers consist of either unsigned integer color indices, R, G, B and optionally A unsigned integer values, or R, G, B, and optionally A floating-point values. Depth buffers consist of either unsigned integer values of the format described in section 2.11.1, or floating-point values. The number of bitplanes...

Modify Section 4.2.3 (Clearing the Buffers), p. 215

(modify fourth paragraph)

The functions

```
void ClearDepth(clampd d);
void ClearDepthdNV(double d);
```

are used to set the depth value used when clearing the depth buffer. ClearDepth takes a floating-point value that is clamped to the range [0, 1]. ClearDepthdNV takes a floating-point value that is not clamped. When clearing a fixed-point depth buffer, the depth clear value is clamped to the range [0, 1], and converted to fixed-point according to the rules for a window z value given in section 2.11.1. No clamping or conversion are applied when clearing a floating-point depth buffer.

Modify Section 4.3.1 (Writing to the Stencil Buffer), p. 218

(modify paragraph added by EXT_packed_depth_stencil, p. 219)
If the <format> is DEPTH_STENCIL_EXT, 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, then the error INVALID_OPERATION occurs. If the <type> parameter is not UNSIGNED_INT_24_8_EXT, or FLOAT_32_UNSIGNED_INT_24_8_NV then the error INVALID_ENUM occurs.

Modify Section 4.3.2 (Reading Pixels), p. 219

(modify "Conversion of Depth values," p. 222, as modified by EXT_packed_depth_stencil) This step only applies if <format> is DEPTH_COMPONENT or DEPTH_STENCIL_EXT and the depth buffer uses a fixed-point representation. An 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 2.11.1). No conversion is necessary if <format> is DEPTH_COMPONENT or DEPTH_STENCIL_EXT and the depth buffer uses a floating-point representation.

Add a row to table 4.6, p. 223

type Parameter	Index Mask
...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	2^8-1

Add a row to table 4.7, p. 224

type Parameter	GL Type	Component Conversion
...	...	...
FLOAT_32_UNSIGNED_INT_24_8_REV_NV	float	c = f (depth only)

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify DEPTH_RANGE entry in table 6.9 (Transformation State) p. 270

Get Value	Type	Get Command	Init Value	Description	Sec.	Attribute
DEPTH_RANGE	2xR	GetFloatv	0,1	Depth range near & far	2.11.1	viewport

Modify DEPTH_BOUNDS_EXT entry in table 6.19 (Pixel Operation) p. 280

Get Value	Type	Get Command	Init Value	Description	Sec	Attribute
DEPTH_BOUNDS_EXT	2xR	GetFloatv	0,1	Depth bounds zmin & zmax	4.1.X	depth-buffer

Modify DEPTH_CLEAR_VALUE entry in table 6.21 (Framebuffer Control) p. 280

Get Value	Type	Get Command	Init Value	Description	Sec	Attribute
DEPTH_CLEAR_VALUE	R	GetFloatv	1	Depth buffer clear value	4.2.3	depth-buffer

Add DEPTH_BUFFER_FLOAT_MODE entry to table 6.32 (Implementation Dependent Values) p. 293

Get Value	Type	Get Command	Init Value	Description	Sec	Attribute
DEPTH_BUFFER_FLOAT_MODE	B	GetBooleanv	-	True if depth buffer uses a floating-point represnetation	4	-

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Dependencies on EXT_depth_bounds_test:

Modify the definition of DepthBoundsEXT in section 4.1.x Depth Bounds Test.

Modify section 4.1.x (Depth Bounds Test)

(modify first paragraph) ...These values are set with

```
    void DepthBoundsEXT(clampd zmin, clampd zmax);
    void DepthBoundsdNV(double zmin, double zmax);
```

The paramerters to DepthBoundsEXT are clamped to the range [0, 1]. No clamping is applied to the parameters of DepthBoundsdNV. Each of zmin and zmax are subject to clamping to the range of the depth buffer at the time the depth bounds test is applied. For fixed-point depth buffers, the applied zmin and zmax are clamped to [0, 1]. For floating-point depth buffers, the applied zmin and zmax are unmodified. If zmin <= Zpixel <= zmax, then the depth bounds test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using Enable or Disable using the constant DEPTH_BOUNDS_TEST_EXT. When disabled, it is as if the depth bounds test always passes. If zmin is greater than zmax, then the error INVALID_VALUE is generated. The state required consists of two floating-point values and a bit indicating whether the test is enabled or disabled. In the initial state, zmin and zmax are set to 0.0 and 1.0 respectively; and the depth bounds test is disabled.

Errors

Modify the following error in the EXT_packed_depth_stencil specification by adding mention of FLOAT_32_UNSIGNED_INT_24_8_REV_NV:

The error INVALID_ENUM is generated if DrawPixels or ReadPixels is called where format is DEPTH_STENCIL_EXT and type is not UNSIGNED_INT_24_8_EXT, or FLOAT_32_UNSIGNED_INT_24_8_REV_NV.

Modify the following error in the EXT_packed_depth_stencil specification by adding mention of FLOAT_32_UNSIGNED_INT_24_8_REV_NV:

The error INVALID_OPERATION is generated if DrawPixels or ReadPixels is called where type is UNSIGNED_INT_24_8_EXT, or FLOAT_32_UNSIGNED_INT_24_8_REV_NV and format is not DEPTH_STENCIL_EXT.

Add the following error to the NV_copy_depth_to_color specification:

The error INVALID_OPERATION is generated if CopyPixels is called where type is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCL_TO_BGRA_NV and the depth buffer uses a floating point representation.

New State

None.

Issues

1. *Should this extension expose floating-point depth buffers through WGL/GLX "pixel formats?"*

   RESOLVED: No. The WGL_EXT_depth_float extension already provides a mechanism for requesting a floating-point depth buffer.

2. *How does an application access the full range of a floating-point depth buffer?*

   RESOLVED: New functions have been introduced that set existing GL state without clamping to the range [0, 1]. These functions are DepthRangedNV, ClearDepthdNV, and DepthBoundsdNV.

3. *Should we add a new state query to determine if the depth buffer is using a floating-point representation?*

   RESOLVED: Yes. An application can query DEPTH_FLOAT_MODE_NV to see if the depth buffer is using a floating-point representation.

4. *How does polygon offset work with floating-point depth buffers?*

   RESOLVED: The third paragraph of section 3.5.5 (Depth Offset) describes the minimum resolvable difference r as "the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer." The polygon offset value o is computed as a function of r. The minimum resolvable difference r makes sense for fixed-point depth values, and even floating-point depth values in the range [-1, 1]. For unclamped floating-point depth values, there is no constant minimum resolvable difference -- the minimum difference necessary to change the mantissa of a floating-point value by one bit depends on the exponent of the value being offset. To remedy this problem, the minimum resolvable difference is defined to be relative to the range of depth values for the given primitive when the depth buffer is floating-point.

5. *How does NV_copy_depth_to_color work with floating-point depth values?*

   RESOLVED: It isn't clear that there is any usefulness to copying the data for 32-bit floating-point depth values to a fixed-point color buffer. It is even less clear how copying packed data from a FLOAT_32_UNSIGNED_24_8_NV depth/stencil buffer to a fixed-point color buffer would be useful or even how it should be implemented. An error should be generated if CopyPixels is called where <type> is DEPTH_STENCIL_TO_RGBA_NV or DEPTH_STENCIL_TO_BGRA and the depth buffer uses a floating-point representation.

6. *Other OpenGL hardware implementations may be capable of supporting floating-point depth buffers. Why is this an NV extension?*

   RESOLVED: When rendering to floating-point depth buffers, we expect that other implementations may only be capable of supporting Z values in the range [0,1]. For such implementations, floating-point Z buffers do not improve the range of Z values supported, but do offer

increased precision than conventional 24-bit fixed-point Z buffers, particularly around zero.

This extension was initially proposed as an EXT, but we have changed it to an NV extension in the expectation that an EXT may be offered at some point in the not-too-distant future. We expect that the EXT could be supported by a larger range of vendors. NVIDIA would continue to support both extensions, where the NV extension could be thought of as taking the capability of the EXT version and extending it to support Z values outside the range [0,1].

Revision History

None

Name

NV_fragment_program4

Name Strings

(none)

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 11/06/2007
NVIDIA Revision: 4

Number

335

Dependencies

OpenGL 1.1 is required.

NV_gpu_program4 is required. This extension is supported if "GL_NV_gpu_program4" is found in the extension string.

ATI_draw_buffers and ARB_draw_buffers trivially affects the definition of this specification.

ARB_fragment_program_shadow trivially affects the definition of this specification.

NV_primitive_restart trivially affects the definition of this extension.

This extension is written against the OpenGL 2.0 specification.

Overview

This extension builds on the common assembly instruction set infrastructure provided by NV_gpu_program4, adding fragment program-specific features.

This extension provides interpolation modifiers to fragment program attributes allowing programs to specify that specified attributes be flat-shaded (constant over a primitive), centroid-sampled (multisample rendering), or interpolated linearly in screen space. The set of input and output bindings provided includes all bindings supported by ARB_fragment_program. Additional input bindings are provided to determine whether fragments were generated by front- or back-facing primitives ("fragment.facing"), to identify the individual primitive used to generate the fragment ("primitive.id"), and to determine distances to user clip

planes ("fragment.clip[n]"). Additionally generic input attributes allow a fragment program to receive a greater number of attributes from previous pipeline stages than possible using only the pre-defined fixed-function attributes.

By and large, programs written to ARB_fragment_program can be ported directly by simply changing the program header from "!!ARBfp1.0" to "!!NVfp4.0", and then modifying instructions to take advantage of the expanded feature set. There are a small number of areas where this extension is not a functional superset of previous fragment program extensions, which are documented in the NV_gpu_program4 specification.

New Procedures and Functions

None.

New Tokens

None.

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.X, GPU Programs

(insert after second paragraph)

Fragment Programs

Fragment programs are used to compute the transformed attributes of a fragment, in lieu of the set of fixed-function operations described in sections 3.8 through 3.10. Fragment programs are run on a single fragment at a time, and the state of neighboring fragments is not explicitly available. (In practice, fragment programs may be run on a block of fragments, and neighboring fragments' attributes may be used for texture LOD calculations or partial derivative approximation.) The inputs available to a fragment program are the interpolated attributes of a fragment, which include (among other things) window-space position, primary and secondary colors, and texture coordinates. The results of the program are one (or more) colors and possibly a new window Z coordinate. A fragment program can not modify the (X,Y) location of the fragment.

Modify Section 2.X.2, Program Grammar

(replace third paragraph)

Fragment programs are required to begin with the header string "!!NVfp4.0". This header string identifies the subsequent program body as being a fragment program and indicates that it should be parsed according to the base NV_gpu_program4 grammar plus the additions below. Program string parsing begins with the character immediately following the header string.

(add the following grammar rules to the NV_gpu_program4 base grammar)

```
<instruction>           ::= <SpecialInstruction>

<varModifier>           ::= <interpModifier>

<SpecialInstruction>    ::= "KIL" <opModifiers> <killCond>
                          | "DDX" <opModifiers> <instResult> ","
                            <instOperandV>
                          | "DDY" <opModifiers> <instResult> ","
                            <instOperandV>

<killCond>              ::= <instOperandV>

<interpModifier>        ::= "FLAT"
                          | "CENTROID"
                          | "NOPERSPECTIVE"

<attribBasic>           ::= <fragPrefix> "fogcoord"
                          | <fragPrefix> "position"
                          | <fragPrefix> "facing"
                          | <attribTexCoord> <optArrayMemAbs>
                          | <attribClip> <arrayMemAbs>
                          | <attribGeneric> <arrayMemAbs>
                          | "primitive" "." "id"

<attribColor>           ::= <fragPrefix> "color"

<attribMulti>           ::= <attribTexCoord> <arrayRange>
                          | <attribClip> <arrayRange>
                          | <attribGeneric> <arrayRange>

<attribTexCoord>        ::= <fragPrefix> "texcoord"

<attribClip>            ::= <fragPrefix> "clip"

<attribGeneric>         ::= <fragPrefix> "attrib"

<fragPrefix>            ::= "fragment" "."

<resultBasic>           ::= <resPrefix> "color" <resultOptColorNum>
                          | <resPrefix> "depth"

<resultOptColorNum>     ::= /* empty */

<resPrefix>             ::= "result" "."
```

(add the following subsection to section 2.X.3.1, Program Variable Types)

Explicitly declared fragment program attribute variables may have one or more interpolation modifiers that control how per-fragment values are computed.

An attribute variable declared as "FLAT" will be flat-shaded. For such variables, the value of the attribute will be constant over an entire primitive and will taken from the provoking vertex of the primitive, as described in Section 2.14.7. If "FLAT" is not specified, attributes will

be interpolated as described in Chapter 3, with the exception that attribute variables bound to colors will still be flat-shaded if the shade model (section 2.14.7) is FLAT. If an attribute variable declared as "FLAT" corresponds to a texture coordinate replaced by a point sprite (s,t) value (section 3.3.1), the value of the attribute is undefined.

An attribute variable declared as "CENTROID" will be interpolated using a point on or inside the primitive, if possible, when doing multisample line or polygon rasterization (sections 3.4.4 and 3.5.6). This method can avoid artifacts during multisample rasterization when some samples of a pixel are covered, but the sample location used is outside the primitive. Note that when centroid sampling, the sample points used to generate attribute values for adjacent pixels may not be evenly spaced, which can lead to artifacts when evaluating partial derivatives or performing texture LOD calculations needed for mipmapping. If "CENTROID" is not specified, attributes may be sampled anywhere inside the pixel as permitted by the specification, including at points outside the primitive.

An attribute variable declared as "NOPERSPECTIVE" will be interpolated using a method that is linear in screen space, as described in equation 3.7 and the appoximation that follows equation 3.8. If "NOPERSPECTIVE" is not specified, attributes must be interpolated with perspective correction, as described in equations 3.6 and 3.8. When clipping lines or polygons, an alternate method is used to compute the attributes of vertices introduced by clipping when they are specified as "NOPERSPECTIVE" (section 2.14.8).

Implicitly declared attribute variables (bindings used directly in a program instruction) will inherit the interpolation modifiers of any explicitly declared attribute variable using the same binding. If no such variable exists, default interpolation modes will be used.

For fragments generated by point primitives, DrawPixels, and Bitmap, interpolation modifiers have no effect.

Implementations are not required to support arithmetic interpolation of integer values written by a previous pipeline stage. Integer fragment program attribute variables must be flat-shaded; a program will fail to load if it declares a variable with the "INT" or "UINT" data type modifiers without the "FLAT" interpolation modifier.

There are several additional limitations on the use of interpolation modifiers. A fragment program will fail to load:

* if an interpolation modifier is specified when declaring a non-attribute variable,

* if the same interpolation modifier is specified more than once in a single declaration (e.g., "CENTROID CENTROID ATTRIB"),

* if the "FLAT" modifier is used together with either "CENTROID" or "NOPERSPECTIVE" in a single declaration,

* if any interpolation modifier is specified when declaring a variable bound to a fragment's position, face direction, fog coordinate, or any interpolated clip distance,

* if multiple attribute variables with different interpolation modifiers are bound to the same fragment attribute, or

* if one variable is bound to the fragment's primary color and a second variable with different interpolation modifiers is bound the the fragment's secondary color.

(add the following subsection to section 2.X.3.2, Program Attribute Variables)

Fragment program attribute variables describe the attributes of a fragment produced during rasterization. The set of available bindings is enumerated in Table X.X.

Most attributes correspond to per-vertex attributes that are interpolated over a primitive; such attributes are subject to the interpolation modifiers described in section 2.X.3.1. The fragment's position, facing, and primitive IDs are the exceptions, and are generated specially during rasterization. Since two-sided color selection occurs prior to rasterization, there are no distinct "front" or "back" colors available to fragment programs. A single set of colors is available, which corresponds to interpolated front or back vertex colors.

If geometry programs are enabled, attributes will be obtained by interpolating per-vertex outputs written by the geometry program. If geometry programs are disabled, but vertex programs are enabled, attributes will be obtained by interpolating per-vertex outputs written by the vertex program. In either case, the fragment program attributes should be read using the same component data type used to write the vertex output attributes in the geometry or vertex program. The value of any attribute corresponding to a vertex output not written by the geometry or vertex program is undefined.

If neither geometry nor vertex programs are used, attributes will be obtained by interpolating per-vertex values computed by fixed-function vertex processing. All interpolated fragment attributes should be read as floating-point values.

Fragment Attribute Binding	Components	Underlying State
fragment.color	(r,g,b,a)	primary color
fragment.color.primary	(r,g,b,a)	primary color
fragment.color.secondary	(r,g,b,a)	secondary color
fragment.texcoord	(s,t,r,q)	texture coordinate, unit 0
fragment.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
fragment.fogcoord	(f,-,-,-)	fog distance/coordinate
* fragment.clip[n]	(c,-,-,-)	interpolated clip distance n
fragment.attrib[n]	(x,y,z,w)	generic interpolant n
fragment.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
* fragment.clip[n..o]	(c,-,-,-)	clip distances n thru o
fragment.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
* fragment.position	(x,y,z,1/w)	window position
* fragment.facing	(f,-,-,-)	fragment facing
* primitive.id	(id,-,-,-)	primitive number

Table X.X: Fragment Attribute Bindings. The "Components" column indicates the mapping of the state in the "Underlying State" column. Bindings containing "[n]" require an integer value of <n> to select an individual item. Interpolation modifiers are not supported on variables that use bindings labeled with "*".

If a fragment attribute binding matches "fragment.color" or "fragment.color.primary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment's primary color.

If a fragment attribute binding matches "fragment.color.secondary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment's secondary color.

If a fragment attribute binding matches "fragment.texcoord" or "fragment.texcoord[n]", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the fragment texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is used.

If a fragment attribute binding matches "fragment.fogcoord", the "x" component of the fragment attribute variable is filled with either the fragment eye distance or the fog coordinate, depending on whether the fog source is set to FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT, respectively. The "y", "z", and "w" coordinates are undefined.

If a fragment attribute binding matches "fragment.clip[n]", the "x" component of the fragment attribute variable is filled with the interpolated value of clip distance <n>, as written by the vertex or geometry program. The "y", "z", and "w" components of the variable are undefined. If fixed-function vertex processing or position-invariant vertex programs are used with geometry programs disabled, clip distances are obtained by interpolating the per-clip plane dot product:

(p_1' p_2' p_3' p_4') dot (x_e y_e z_e w_e),

for clip plane <n> as described in section 2.12. The clip distance for clip plane <n> is undefined if clip plane <n> is disabled.

If a fragment attribute binding matches "fragment.attrib[n]", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "x", "y", "z", and "w" components of generic interpolant <n>. All generic interpolants will be undefined when used with fixed-function vertex processing with no geometry program enabled.

If a fragment attribute binding matches "fragment.texcoord[n..o]", "fragment.clip[n..o]", or "fragment.attrib[n..o]", a sequence of 1+<o>-<n> bindings is created. For texture coordinate bindings, it is as though the sequence "fragment.texcoord[n], fragment.texcoord[n+1], ... fragment.texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a fragment attribute binding matches "fragment.position", the "x" and "y" components of the fragment attribute variable are filled with the floating-point (x,y) window coordinates of the fragment center, relative to the lower left corner of the window. The "z" component is filled with the fragment's z window coordinate. If z window coordinates are represented internally by the GL as fixed-point values, the z window coordinate undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. The "w" component is filled with the reciprocal of the fragment's clip w coordinate.

If a fragment attribute binding matches "fragment.facing", the "x" component of the fragment attribute variable is filled with +1.0 or -1.0, depending on the orientation of the primitive producing the fragment. If the fragment is generated by a back-facing polygon (including point- and line-mode polygons), the facing is -1.0; otherwise, the facing is +1.0. The "y", "z", and "w" coordinates are undefined.

If a fragment attribute binding matches "primitive.id", the "x" component of the fragment attribute variable is filled with a single integer. If a geometry program is active, this value is obtained by taking the primitive ID value emitted by the geometry program for the provoking vertex. If no geometry program is active, the value is the number of primitives processed by the rasterizer since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is zero. The primitive ID is zero for fragments generated by DrawPixels or Bitmap. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter. The "y", "z", and "w" components of the variable are always undefined.

(add the following subsection to section 2.X.3.5, Program Results.)

Fragment programs produce final fragment values, and the set of result variables available to such programs correspond to the final attributes of a fragment. Fragment program result variables may not be declared as arrays.

The set of allowable result variable bindings is given in Table X.X.

Binding	Components	Description
result.color	(r,g,b,a)	color
result.color[n]	(r,g,b,a)	color output n
result.depth	(*,*,d,*)	depth coordinate

Table X.X: Fragment Result Variable Bindings.
Components labeled "*" are unused.

If a result variable binding matches "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment's output color.

If a result variable binding matches "result.color[n]" and the ARB_draw_buffers program option is specified, updates to the "x", "y", "z", and "w" components of the color result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment output color numbered <n>. If the ARB_draw_buffers program option is not specified, the "result.color[n]" binding is unavailable.

If a result variable binding matches "result.depth", updates to the "z" component of the result variable modify the fragment's output depth value. If the "result.depth" binding is not in used in a variable written to by any instruction in the fragment program, the interpolated depth value produced by rasterization is used as if fragment program mode is not enabled. Otherwise, the value written by the fragment program is used, and the fragment's final depth value is undefined if the program did not end up writing a depth value due to flow control or write masks. Writes to any component of depth other than the "z" component have no effect.

(modify Table X.13 in section 2.X.4, Program Instructions, to include the following.)

	Modifiers								
Instruction	F	I	C	S	H	D	Inputs	Out	Description
DDX	X	-	X	X	X	F	v	v	partial derivative relative to X
DDY	X	-	X	X	X	F	v	v	partial derivative relative to Y
KIL	X	X	-	-	X	F	vc	-	kill fragment

(add the following subsection to section 2.X.5, Program Options.)

Section 2.X.5.Y, Fragment Program Options

+ Fixed-Function Fog Emulation (ARB_fog_exp, ARB_fog_exp2, ARB_fog_linear)

If a fragment program specifies one of the options "ARB_fog_exp", "ARB_fog_exp2", or "ARB_fog_linear", the program will apply fog to the program's final color using a fog mode of EXP, EXP2, or LINEAR, respectively, as described in section 3.10.

When a fog option is specified in a fragment program, semantic restrictions are added to indicate that a fragment program will fail to load if the number of temporaries it contains exceeds the

implementation-dependent limit minus 1, if the number of attributes it contains exceeds the implementation-dependent limit minus 1, or if the number of parameters it contains exceeds the implementation-dependent limit minus 2.

Additionally, when the ARB_fog_exp option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 3. When the ARB_fog_exp2 option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 4. When the ARB_fog_linear option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 2.

Only one fog application option may be specified by any given fragment program. A fragment program that specifies more than one of the program options "ARB_fog_exp", "ARB_fog_exp2", and "ARB_fog_linear", will fail to load.

+ Precision Hints (ARB_precision_hint_fastest, ARB_precision_hint_nicest)

Fragment program computations are carried out at an implementation-dependent precision. However, some implementations may be able to perform fragment program computations at more than one precision, and may be able to trade off computation precision for performance.

If a fragment program specifies the "ARB_precision_hint_fastest" program option, implementations should select precision to minimize program execution time, with possibly reduced precision. If a fragment program specifies the "ARB_precision_hint_nicest" program option, implementations should maximize the precision, with possibly increased execution time.

Only one precision control option may be specified by any given fragment program. A fragment program that specifies both the "ARB_precision_hint_fastest" and "ARB_precision_hint_nicest" program options will fail to load.

+ Multiple Color Outputs (ARB_draw_buffers, ATI_draw_buffers)

If a fragment program specifies the "ARB_draw_buffers" or "ATI_draw_buffers" option, it will generate multiple output colors, and the result binding "result.color[n]" is allowed, as described in section 2.X.3.5. If this option is not specified, a fragment program that attempts to bind "result.color[n]" will fail to load, and only "result.color" will be allowed.

The multiple color outputs will typically be written to an ordered list of draw buffers in the manner described in the ARB_draw_buffers extension specification.

+ Fragment Program Shadows (ARB_fragment_program_shadow)

The ARB_fragment_program_shadow option introduced a set of "SHADOW" texture targets and made the results of depth texture lookups undefined unless the texture format and compare mode were consistent with the target provided in the fragment program instruction. This behavior is enabled by default in NV_gpu_program4; specifying the option is not illegal but has no additional effect.

(add the following subsection to section 2.X.8, Program Instruction Set.)

Section 2.X.8.Z, DDX: Partial Derivative Relative to X

The DDX instruction computes approximate partial derivatives of the four components of the single floating-point vector operand with respect to the X window coordinate to yield a result vector. The partial derivatives are evaluated at the sample location of the pixel.

```
f = VectorLoad(op0);
result = ComputePartialX(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives. Note also that the sample locations for attributes declared with the CENTROID interpolation modifier may not be evenly spaced, which can lead to artifacts in derivative calculations.

DDX supports only floating-point data type modifiers and is available only to fragment programs.

Section 2.X.8.Z, DDY: Partial Derivative Relative to Y

The DDY instruction computes approximate partial derivatives of the four components of the single operand with respect to the Y window coordinate to yield a result vector. The partial derivatives are evaluated at the center of the pixel.

```
f = VectorLoad(op0);
result = ComputePartialY(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives. Note also that the sample locations for attributes declared with the CENTROID interpolation modifier may not be evenly spaced, which can lead to artifacts in derivative calculations.

DDY supports only floating-point data type modifiers and is available only to fragment programs.

Section 2.X.8.Z, KIL: Kill Fragment

The KIL instruction evaluates a condition and kills a fragment if the test passes. A fragment killed by the KIL instruction is discarded, and will not be seen by subsequent stages of the pipeline.

A KIL instruction may be specified using either a floating-point vector operand or a condition code test.

If a floating-point vector is provided, the fragment is discarded if any of its components are negative:

```
  tmp = VectorLoad(op0);
  if ((tmp.x < 0) || (tmp.y < 0) ||
      (tmp.z < 0) || (tmp.w < 0))
  {
      exit;
  }
```

If a condition code test is provided, the fragment is discarded if any component of the test passes:

```
  if (TestCC(rc.c***) || TestCC(rc.*c**) ||
      TestCC(rc.**c*) || TestCC(rc.***c))
  {
     exit;
  }
```

KIL supports no data type modifiers. If a vector operand is provided, it must have floating-point components.

KIL is available only to fragment programs.

Replace Section 2.14.8, and rename it to "Vertex Attribute Clipping" (p. 70).

After lighting, clamping or masking and possible flatshading, vertex attributes, including colors, texture and fog coordinates, shader varying variables, and point sizes computed on a per vertex basis, are clipped. Those attributes associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the attributes assigned to vertices produced by clipping are produced by interpolating attributes along the clipped edge.

Let the attributes assigned to the two vertices P_1 and P_2 of an unclipped edge be a_1 and a_2. The value of t (section 2.12) for a clipped point P is used to obtain the attribute associated with P as

```
  a = t * a_1 + (1-t) * a_2
```

unless the attribute is specified to be interpolated without perspective correction in a fragment program. In that case, the attribute associated with P is

```
  a = t' * a_1 + (1-t') * a_2
```

where

```
  t' = (t * w_1) / (t * w_1 + (1-t) * w_2)
```

and w_1 and w_2 are the w clip coordinates of P_1 and P_2, respectively. If w_1 or w_2 is either zero or negative, the value of the associated attribute is undefined.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None

Additions to the AGL/GLX/WGL Specifications

None

Dependencies on ARB_draw_buffers and ATI_draw_buffers

If neither ARB_draw_buffers nor ATI_draw_buffers is supported, then the discussion of the ARB_draw_buffers option in section 2.X.5.Y should be removed, as well as the result bindings of the form "result.color[n]" and "result.color[n..o]".

Dependencies on ARB_fragment_program_shadow

If ARB_fragment_program_shadow is not supported, then the discussion of the ARB_fragment_program_shadow option in section 2.X.5.Y should be removed.

Dependencies on NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count. If NV_primitive_restart is not supported, references to that extension in the discussion of the "primitive.id" attribute should be removed.

Errors

None

New State

None

New Implementation Dependent State

None

Issues

(1) How should special interpolation controls be specified?

RESOLVED: As a special modifier to fragment program attribute variable declarations. It was decided that the fragment program was the most natural place to put the control. This wouldn't require making a large number of related state changes controlling interpolation whenever the fragment program used. The final mechanism using special interpolation modifiers was chosen because it fit well with the other variable modifiers (for data storage size and data type) provided by NV_gpu_program4. Examples:

```
FLAT ATTRIB texcoords[4] = { fragment.texcoord[0..3] };
CENTROID ATTRIB texcoord4 = fragment.texcoord[4];
CENTROID NOPERSPECTIVE ATTRIB
  attribs[3] = { fragment.attrib[0..2] };
```

There were a variety of options considered, including:

* special declarations in vertex or geometry programs to specify the interpolation type,

* special declarations in the fragment program to specify one or more interpolation type modifiers per binding, such as:

```
INTERPOLATE fragment.texcoord[0..3], FLAT;
INTERPOLATE fragment.texcoord[4], CENTROID;
INTERPOLATE fragment.attrib[0..2], CENTROID, NOPERSPECTIVE;
```

* fixed-function state specifying the interpolation mode

```
glInterpolateAttribNV(GL_TEXTURE0, GL_FLAT);
glInterpolateAttribNV(GL_GENERIC_ATTRIB0, GL_CENTROID_NV);
```

Recent updates to GLSL provide similar functionality (for centroid) with a similar approach, using a modifier on varying variable declarations.

(2) How should perspective-incorrect interpolation (linear in screen space) and clipping interact?

RESOLVED: Primitives with attributes specified to be perspective-incorrect should be clipped so that the vertices introduced by clipping should have attribute values consistent with the interpolation mode. We do not want to have large color shifts introduced by clipping a perspective-incorrect attribute. For example, a primitive that approaches, but doesn't cross, a frustum clip plane should look pretty much identical to a similar primitive that just barely crosses the clip plane.

Clipping perspective-incorrect interpolants that cross the W==0 plane is very challenging. The attribute clipping equation provided in the spec effectively projects all the original vertices to screen space while ignoring the X and Y frustum clip plane. As W approaches zero, the projected X/Y window coordinates become extremely large. When clipping an edge with one vertex inside the frustum and the other out near infinity (after projection, due to W approaching zero), the interpolated

attribute for the entire visible portion of the edge should almost exactly match the attribute value of the visible vertex.

If an outlying vertex approaches and then goes past W==0, it can be said to go "to infinity and beyond" in screen space. The correct answer for screen-linear interpolation is no longer obvious, at least to the author of this specification. Rather than trying to figure out what the "right" answer is or if one even exists, the results of clipping such edges is specified as undefined.

(3) If a shader wants to use interpolation modifiers without using declared variables, is that possible?

RESOLVED: Yes. If "dummy" variables are declared, all interpolants bound to that variable will get the variable's interpolation modifiers. In the following program:

```
FLAT ATTRIB tc02[3] = { fragment.texcoord[0..2] };
MOV R0, fragment.texcoord[1];
MOV R1, fragment.texcoord[3];
```

The variable R0 will get texture coordinate set 1, which will be flat-shaded due to the declaration of "tc02". The variable R1 will get texture coordinate set 3, which will be smooth shaded (default).

(4) Is it possible to read the same attribute with different interpolation modifiers?

RESOLVED: No. A program that tries to do that will fail to compile.

(5) Why can't fragment program results be declared as arrays?

RESOLVED: This is a limitation of the programming model. If an implementation needs to do run-time indexing of fragment program result variables (effectively writing to "result.color[A0.x]"), code such as the following can be used:

```
TEMP colors[4];
...
MOV colors[A0.x], R1;
MOV colors[3], 12.3;
...
# end of the program
MOV result.color[0], colors[0];
MOV result.color[1], colors[1];
MOV result.color[2], colors[2];
MOV result.color[3], colors[3];
```

(6) Do clip distances require that the corresponding clip planes be enabled to be read by a fragment program?

RESOLVED: No.

(7) How do primitive IDs work with fragment programs?

RESOLVED: If a geometry program is enabled, the primitive ID is consumed by the geometry program and is not automatically available to

the fragment program. If the fragment program needs a primitive ID in this case, the geometry program can write out a primitive ID using the "result.primid" binding, and the fragment program will see the primitive ID written for the provoking vertex.

If no geometry program is enabled, the primitive ID is automatically available, and specifies the number of primitives (points, lines, or triangles) processed by since the last explicit or implicit Begin call.

(8) What is the primitive ID for non-geometry commands that generate fragments, such as DrawPixels, Bitmap, and CopyPixels.

RESOLVED: Zero.

(9) How does the FLAT interpolation modifier interact with point sprite coordinate replacement?

RESOLVED: The value of such attributes are undefined. Specifying these two operations together is self-contradictory -- FLAT asks for an interpolant that is constant over a primitive, and point sprite coordinate interpolation asks for an interpolant that is non-constant over a point sprite.

Revision History

Rev.	Date	Author	Changes
4	11/06/07	pbrown	Documented interaction between the FLAT interpolation modifier and point sprite coordinate replacement.
1-3		pbrown	Internal spec development.

Name

NV_framebuffer_multisample_coverage

Name Strings

GL_NV_framebuffer_multisample_coverage

Contact

Mike Strauss, NVIDIA Corporation (mstrauss 'at' nvidia.com)

Status

Shipping in NVIDIA Release 95 drivers (November 2006)

Functionaltiy supported by GeForce 8800

Version

Last Modified Date: November 6, 2006
Revision #5

Number

336

Dependencies

Requires GL_EXT_framebuffer_object.

Requires GL_EXT_framebuffer_blit.

Requires GL_EXT_framebuffer_multisample.

Written based on the wording of the OpenGL 1.5 specification.

Overview

This extension extends the EXT_framebuffer_multisample specification by providing a new function, RenderBufferStorageMultisampleCoverageNV, that distinguishes between color samples and coverage samples.

EXT_framebuffer_multisample introduced the function RenderbufferStorageMultisampleEXT as a method of defining the storage parameters for a multisample render buffer. This function takes a <samples> parameter. Using rules provided by the specification, the <samples> parameter is resolved to an actual number of samples that is supported by the underlying hardware. EXT_framebuffer_multisample does not specify whether <samples> refers to coverage samples or color samples.

This extension adds the function RenderbufferStorageMultisamplCoverageNV, which takes a <coverageSamples> parameter as well as a <colorSamples> parameter.

These two parameters give developers more fine grained control over the quality of multisampled images.

New Procedures and Functions

```
void RenderbufferStorageMultisampleCoverageNV(
         enum target, sizei coverageSamples,
         sizei colorSamples, enum internalformat,
         sizei width, sizei height);
```

New Tokens

Accepted by the <pname> parameter of GetRenderbufferParameterivEXT:

RENDERBUFFER_COVERAGE_SAMPLES_NV	0x8CAB
RENDERBUFFER_COLOR_SAMPLES_NV	0x8E10

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

None.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

Modification to 4.4.2.1 (Renderbuffer Objects)

Add, just above the definition of RenderbufferStorageMultisampleEXT:

"The command

```
void RenderbufferStorageMultisampleCoverageNV(
    enum target, sizei coverageSamples,
    sizei colorSamples, enum internalformat,
    sizei width, sizei height);
```

establishes the data storage, format, dimensions, number of coverage samples, and number of color samples of a renderbuffer object's image. <target> must be RENDERBUFFER_EXT. <internalformat> must be RGB, RGBA, DEPTH_COMPONENT, STENCIL_INDEX, or one of the internal formats from table 3.16 or table 2.nnn that has a base internal format of RGB, RGBA, DEPTH_COMPONENT, or STENCIL_INDEX. <width> and <height> are the dimensions in pixels of the renderbuffer. If either <width> or <height> is greater than MAX_RENDERBUFFER_SIZE_EXT, the error INVALID_VALUE is generated. If the GL is unable to create a data store of the requested size, the error OUT_OF_MEMORY is generated.

Upon success, RenderbufferStorageMultisampleCoverageNV deletes any existing data store for the renderbuffer image and the contents of the data store after calling RenderbufferStorageMultisampleCoverageNV are undefined. RENDERBUFFER_WIDTH_EXT is set to <width>, RENDERBUFFER_HEIGHT_EXT

```
is set to <height>, and RENDERBUFFER_INTERNAL_FORMAT_EXT is set to
<internalformat>.

If <coverageSamples> is zero, then RENDERBUFFER_COVERAGE_SAMPLES_NV
is set to zero.  Otherwise <coverageSamples> represents a request
for a desired minimum number of coverage samples. Since different
implementations may support different coverage sample counts for
multisampled rendering, the actual number of coverage samples
allocated for the renderbuffer image is implementation dependent.
However, the resulting value for RENDERBUFFER_COVERAGE_SAMPLES_NV is
guaranteed to be greater than or equal to <coverageSamples> and no
more than the next larger coverage sample count supported by the
implementation.

If <colorSamples> is zero then RENDERBUFFER_COLOR_SAMPLES_NV is set
to zero.  Otherwise, <colorSamples> represents a request for a
desired minimum number of colors samples.  Since different
implementations may support different color sample counts for
multisampled rendering, the actual number of color samples
allocated for the renderbuffer image is implementation dependent.
Furthermore, a given implementation may support different color
sample counts for each supported coverage sample count.  The
resulting value for RENDERBUFFER_COLOR_SAMPLES_NV is determined
after resolving the value for RENDERBUFFER_COVERAGE_SAMPLES_NV.
If the requested color sample count exceeds the maximum number of
color samples supported by the implementation given the value of
RENDERBUFFER_COVERAGE_SAMPLES_NV, the implementation will set
RENDERBUFFER_COLOR_SAMPLES_NV to the highest supported value.
Otherwise, the resulting value for RENDERBUFFER_COLOR_SAMPLES_NV is
guaranteed to be greater than or equal to <colorSamples> and no
more than the next larger color sample count supported by the
implementation given the value of RENDERBUFFER_COVERAGE_SAMPLES_NV.

If <colorSamples> is greater than <coverageSamples>, the error
INVALID_VALUE is generated.

If <coverageSamples> or <colorSamples> is greater than
MAX_SAMPLES_EXT, the error INVALID_VALUE is generated.

If <coverageSamples> is greater than zero, and <colorSamples> is
zero, RENDERBUFFER_COLOR_SAMPLES_NV is set to an implementation
dependent value based on RENDERBUFFER_COVERAGE_SAMPLES_NV.

Modify the definition of RenderbufferStorageMultisampleEXT as
follows:

"The command

    void RenderbufferStorageMultisampleEXT(
        enum target, sizei samples,
        enum internalformat,
        sizei width, sizei height);

is equivalent to calling

    RenderbufferStorageMultisamplesCoverageNv(target, samples, 0,
        internalforamt, width, height).
```

Modification to 4.4.4.2 (Framebuffer Completeness)

Modify the RENDERBUFFER_SAMPLES_EXT entry in the bullet list:

* The value of RENDERBUFFER_COVERAGE_SAMPLES_NV is the same for all attached images.
  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

Add an entry to the bullet list:

* The value of RENDERBUFFER_COLOR_SAMPLES_NV is the same for all attached images.
  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE_EXT }

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Errors

The error INVALID_OPERATION is generated if RenderbufferStorageMultisampleCoverageNV is called and <colorSamples> is greater than <coverageSamples>

The error INVALID_VALUE is generated if RenderbufferStorageMultisampleCoverageNV is called and <coverageSamples> is greater than MAX_SAMPLES_EXT.

The error INVALID_VALUE is generated if RenderbufferStorageMultisampleCoverageNV is called and <colorSamples> is greater than MAX_SAMPLES_EXT.

New State

(add to table 8.nnn, "Renderbuffers (state per renderbuffer object)")

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_COVERAGE_SAMPLES_NV	Z+	GetRenderbufferParameterivEXT	0	Number of coverage samples used by the renderbuffer	4.4.2.1	-
RENDERBUFFER_COLOR_SAMPLES_NV	Z+	GetRenderbufferParameterivEXT	0	Number of color samples used by the renderbuffer	4.4.2.1	-

(modify RENDERBUFFER_SAMPLES_EXT entry in table 8.nnn)

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
RENDERBUFFER_SAMPLES_EXT	Z+	GetRenderbufferParameterivEXT	0	Alias for RENDERBUFFER_-COVERAGE_SAMPLES_NV	4.4.2.1	-

New Implementation Dependent State

None

Issues

(1) *How should RenderbufferStorageMultisampleEXT be layered on top of RenderbufferStorageMultisampleCoverageNV?*

RESOLVED. NVIDIA will expose this extension at the same time that EXT_framebuffer_multisample is exposed, so there will not be any issues with backward compatibility. However, some developers choose not to use vendor specific extensions. These developers should be able to make use of current and future hardware that differentiates between color and coverage samples. Since color samples are a subset of coverage samples, the <samples> parameter to RenderbufferStorageMultisampleEXT should be treated as a request for coverage samples. The implementation is free to choose the number of color samples used by the renderbuffer.

(2) *<coverageSamples> is rounded up to the next highest number of samples supported by the implementation. How should <colorSamples> be rounded given that an implementation may not support all combinations of <coverageSamples> and <colorSamples>?*

RESOLVED: It is a requirement that <coverageSamples> be compatible with the <samples> parameter to RenderbufferStorageMultisampleEXT. While it is desirable for <colorSamples> to resolve the same way as <coverageSamples>, this may not always be possible. An implementation may support a different maximum number of color samples for each coverage sample count. It would be confusing to set an error when <colorSamples> exceeds the maximum supported number of color samples for a given coverage sample count, because there is no mechanism to query or predict this behavior. Therefore, the implementation should round <colorSamples> down when it exceeds the maximum number of color samples supported with the given coverage sample count. Otherwise, <colorSamples> is rounded up to the next highest number of color samples supported by the implementation.

(3) *Should a new query function be added so that an application can determine the maximum number of color samples supported with a given value of <coverageSamples>?*

UNRESOLVED. Such a query would have to evaluate <coverageSamples>, and resolve it to an implementation

supported value. The query would then return the maximum number of color samples supported given the resolved value of <coverageSamples>. There is no precedent for supporting a query of an implementation dependent value that requires complex evaluation of a parameter to the query. Adding such a query is unlikely.

An alternative query mechanism might involve a pair of queries. One query returns the maximum number of unique combinations of coverage samples and color samples supported by the implementation. A second query is used to enumerate these combinations. In the event that no such query mechanism is added, an application can still determin the set of unique and valid combinations of coverage samples and color samples.

An application wishing to implement such a query can do so by creating a set of multisample renderbuffers and querying their properties. A renderbuffer can be created for each (<coverageSamples>, <colorSamples>) pair where <coverageSamples> is in [1, MAX_SAMPLES_EXT], and <colorSamples> is in [1, <coverageSamples>]. The application can query RENDERBUFFER_COVERAGE_SAMPLES_NV and RENDERBUFFER_COLOR_SAMPLES_NV for each renderbuffer, using the results to identify the set of unique (<coverageSamples>, <colorSamples>) pairs supported by the implementation.

Revision History

None

Name

NV_geometry_program4

Name Strings

(none)

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 11/06/2006
NVIDIA Revision: 6

Number

323

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 2.0 specification.

NV_gpu_program4 is required. This extension is supported if "GL_NV_gpu_program4" is found in the extension string.

EXT_framebuffer_object interacts with this extension.

EXT_framebuffer_blit interacts with this extension.

EXT_texture_array interacts with this extension.

ARB_texture_rectangle trivially affects the definition of this extension.

EXT_texture_buffer_object trivially affects the definition of this extension.

NV_primitive_restart trivially affects the definition of this extension.

Overview

NV_geometry_program4 defines a new type of program available to be run on the GPU, called a geometry program. Geometry programs are run on full primitives after vertices are transformed, but prior to flat shading and clipping.

A geometry program begins with a single primitive - a point, line, or triangle. Quads and polygons are allowed, but are decomposed into individual triangles prior to geometry program execution. It can read the

attributes of any of the vertex in the primitive and use them to generate new primitives. A geometry program has a fixed output primitive type, either a point, a line strip, or a triangle strip. It emits vertices (using the EMIT opcode) to define the output primitive. The attributes of emitted vertices are specified by writing to the same set of result bindings (e.g., "result.position") provided for vertex programs. Additionally, a geometry program can emit multiple disconnected primitives by using the ENDPRIM opcode, which is roughly equivalent to calling End and then Begin again. The primitives emitted by the geometry program are then clipped and then processed like an equivalent OpenGL primitive specified by the application.

This extension provides four additional primitive types: lines with adjacency, line strips with adjacency, separate triangles with adjacency, and triangle strips with adjacency. Some of the vertices specified in these new primitive types are not part of the ordinary primitives. Instead, they represent neighboring vertices that are adjacent to the two line segment end points (lines/strips) or the three triangle edges (triangles/tstrips). These "adjacency" vertices can be accessed by geometry programs and used to match up the outputs of the geometry program with those of neighboring primitives.

Additionally, geometry programs allow for layered rendering, where entire three-dimensional, cube map, or array textures (EXT_texture_array) can be bound to the current framebuffer. Geometry programs can use the "result.layer" binding to select a layer or cube map face to render to. Each primitive emitted by such a geometry program is rendered to the layer taken from its provoking vertex.

Since geometry programs expect a specific input primitive type, an error will occur if the application presents primtives of a different type. For example, if an enabled geometry program expects points, an error will occur at Begin() time, if a primitive mode of TRIANGLES is specified.

New Procedures and Functions

```
void ProgramVertexLimitNV(enum target, int limit);

void FramebufferTextureEXT(enum target, enum attachment,
                           uint texture, int level);
void FramebufferTextureLayerEXT(enum target, enum attachment,
                                uint texture, int level, int layer);
void FramebufferTextureFaceEXT(enum target, enum attachment,
                               uint texture, int level, enum face);
```

New Tokens

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

GEOMETRY_PROGRAM_NV	0x8C26

```
Accepted by the <pname> parameter of GetProgramivARB:

    MAX_PROGRAM_OUTPUT_VERTICES_NV                  0x8C27
    MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV          0x8C28
    GEOMETRY_VERTICES_OUT_EXT                       0x8DDA
    GEOMETRY_INPUT_TYPE_EXT                         0x8DDB
    GEOMETRY_OUTPUT_TYPE_EXT                        0x8DDC

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

    MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT            0x8C29

Accepted by the <mode> parameter of Begin, DrawArrays, MultiDrawArrays,
DrawElements, MultiDrawElements, and DrawRangeElements:

    LINES_ADJACENCY_EXT                             0xA
    LINE_STRIP_ADJACENCY_EXT                        0xB
    TRIANGLES_ADJACENCY_EXT                         0xC
    TRIANGLE_STRIP_ADJACENCY_EXT                    0xD

Returned by CheckFramebufferStatusEXT:

    FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT        0x8DA8
    FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT          0x8DA9

Accepted by the <pname> parameter of
GetFramebufferAttachmentParameterivEXT:

    FRAMEBUFFER_ATTACHMENT_LAYERED_EXT              0x8DA7
    FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT        0x8CD4

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

    PROGRAM_POINT_SIZE_EXT                          0x8642

(Note:  The "EXT" tokens above are shared with the EXT_geometry_shader4
extension.)

(Note:  FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is simply an alias for the
FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET_EXT token provided in
EXT_framebuffer_object.  This extension generalizes the notion of
"<zoffset>" to include layers of an array texture.)

(Note:  PROGRAM_POINT_SIZE_EXT is simply an alias for the
VERTEX_PROGRAM_POINT_SIZE token provided in OpenGL 2.0, which is itself an
alias for VERTEX_PROGRAM_POINT_SIZE_ARB provided by ARB_vertex_program.
Program-computed point sizes can be enabled if geometry programs are
enabled, even if no vertex program is used.)
```

Additions to Chapter 2 of the OpenGL 1.5 Specification (OpenGL Operation)

Modify Section 2.6.1 (Begin and End Objects), p. 13

(Add to end of section, p. 18)

(add figure)

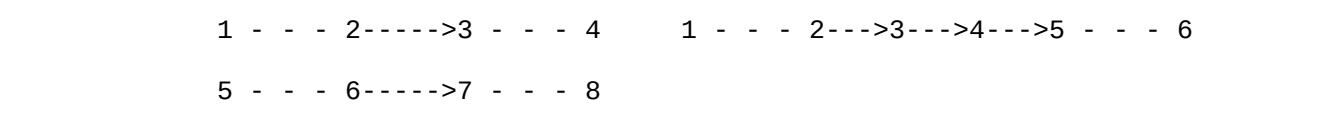


Figure X.1 (a) Lines with adjacency, (b) Line strip with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry program.

Lines with Adjacency

Lines with adjacency are independent line segments where each endpoint has a corresponding "adjacent" vertex that can be accessed by a geometry program (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

A line segment is drawn from the 4i + 2nd vertex to the 4i + 3rd vertex for each i = 0, 1, ... , n-1, where there are 4n+k vertices between the Begin and End. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the 4i + 1st and 4i + 4th vertices are considered adjacent to the 4i + 2nd and 4i + 3rd vertices, respectively. See Figure X.1.

Lines with adjacency are generated by calling Begin with the argument value LINES_ADJACENCY_EXT.

Line Strips with Adjacency

Line strips with adjacency are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by geometry programs (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

A line segment is drawn from the i + 2nd vertex to the i + 3rd vertex for each i = 0, 1, ..., n-1, where there are n+3 vertices between the Begin and End. If there are fewer than four vertices between a Begin and End, all vertices are ignored. For line segment i, the i + 1st and i + 4th vertices are considered adjacent to the i + 2nd and i + 3rd vertices, respectively. See Figure X.1.

Line strips with adjacency are generated by calling Begin with the argument value LINE_STRIP_ADJACENCY_EXT.

(add figure)

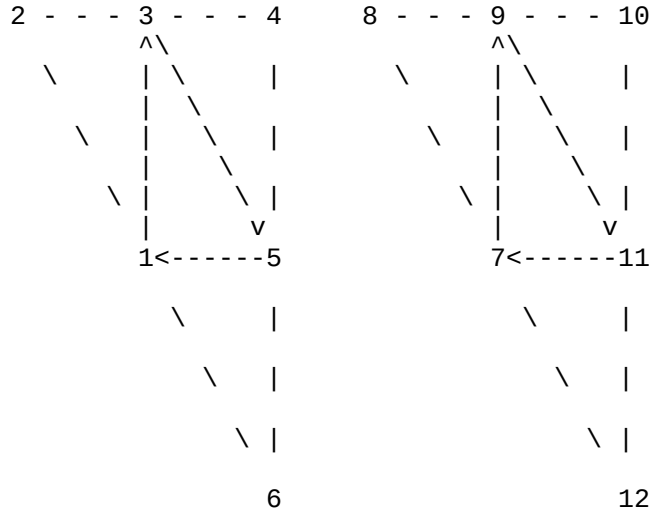


Figure X.2 Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry program.

Triangles with Adjacency

Triangles with adjacency are similar to separate triangles, except that each triangle edge has an adjacent vertex that can be accessed by geometry programs (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

The 6i + 1st, 6i + 3rd, and 6i + 5th vertices (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 6n+k vertices between the Begin and End. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the i + 2nd, i + 4th, and i + 6th vertices are considered adjacent to edges from the i + 1st to the i + 3rd, from the i + 3rd to the i + 5th, and from the i + 5th to the i + 1st vertices, respectively. See Figure X.2.

Triangles with adjacency are generated by calling Begin with the argument value TRIANGLES_ADJACENCY_EXT.

(add figure)

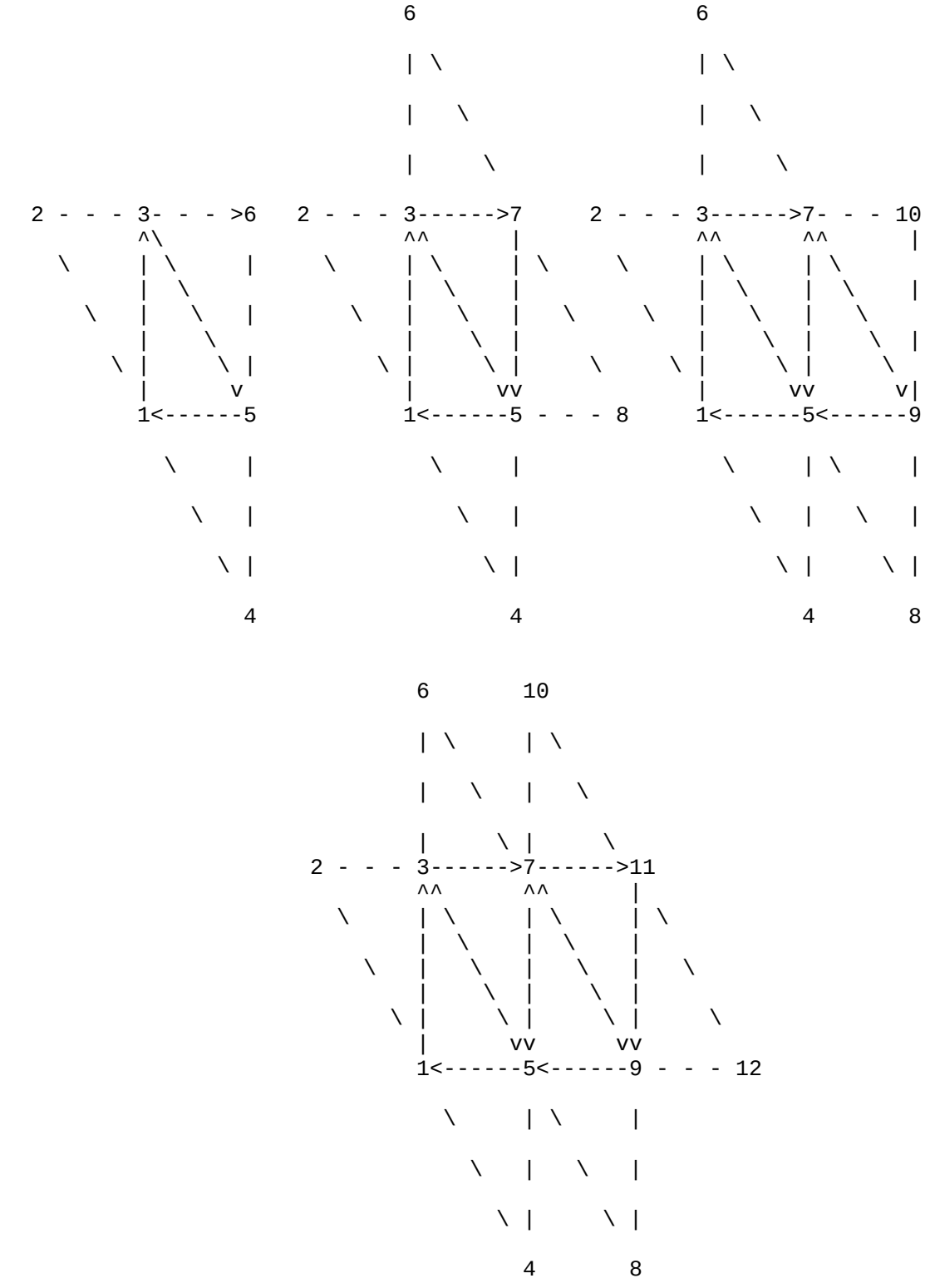


Figure X.3 Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry program.

Triangle Strips with Adjacency

Triangle strips with adjacency are similar to triangle strips, except that each triangle edge has an adjacent vertex that can be accessed by geometry programs (Section 2.15). If geometry programs are disabled, the "adjacent" vertices are ignored.

In triangle strips with adjacency, n triangles are drawn using 2 * (n+2) + k vertices between the Begin and End. k is either 0 or 1; if k is 1, the final vertex is ignored. If fewer than 6 vertices are specified between the Begin and End, the entire primitive is ignored. Table X.1 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle. See Figure X.3.

(add table)

	primitive vertices			adjacent vertices		
primitive	1st	2nd	3rd	1/2	2/3	3/1
only (i==0, n==1)	1	3	5	2	6	4
first (i==0)	1	3	5	2	7	4
middle (i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+7
middle (i even)	2i+1	2i+3	2i+5	2i-1	2i+7	2i+4
last (i==n-1, i odd)	2i+3	2i+1	2i+5	2i-1	2i+4	2i+6
last (i==n-1, i even)	2i+1	2i+3	2i+5	2i-1	2i+6	2i+4

Table X.1: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices in the "1st", "2nd", and "3rd" columns under "primitive vertices", in that order. The vertices in the "1/2", "2/3", and "3/1" columns under "adjacent vertices" are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to the six cases: the first and only triangle (i=0, n=1), the first triangle of several (i=0, n>0), "odd" middle triangles (i=1,3,5...), "even" middle triangles (i=2,4,6,...), and special cases for the last triangle inside the Begin/End, when i is either even or odd. For the purposes of this table, the first vertex specified after Begin is numbered "1" and the first triangle is numbered "0".

Triangle strips with adjacency are generated by calling Begin with the argument value TRIANGLE_STRIP_ADJACENCY_EXT.

Modify Section 2.14.1, Lighting (p. 59)

(modify fourth paragraph, p. 63) Additionally, vertex and geometry shaders and programs can operate in two-sided color mode, which is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE. When a vertex or geometry shader is active, the shaders can write front and back color values to the gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor outputs. When a vertex or geometry program is active, programs can write front and back colors using the available color result bindings. When a vertex or geometry shader or program is active and two-sided color mode is enabled, the GL chooses between front and back colors, as described below. If

two-sided color mode is disabled, the front color output is always selected.

Insert New Section 2.14.6, Geometry Programs (between 2.14.5, Color Index Lighting and 2.14.6, Clamping and Masking, p. 69)

Section 2.14.6, Geometry Programs

Each primitive may be optionally transformed by a geometry program. Geometry programs are enabled by calling Enable with the value GEOMETRY_PROGRAM_NV. A geometry program takes a single input primitive and generates vertices to be arranged into one or more output primitives. The original input primitive is discarded, and the output primitives are processed in order by the remainder of the GL pipeline.

Section 2.14.6.1, Geometry Program Input Primitives

A geometry program can operate on one of five input primitive types, as specified by the mandatory "PRIMITIVE_IN" declaration. Depending on the input primitive type, one to six vertices are available when the program is executed. A geometry program will fail to load unless it contains exactly one such declaration.

Each input primitive type supports only a subset of the primitives provided by the GL. If geometry programs are enabled, Begin, or any function that implicitly calls Begin, will produce an INVALID_OPERATION error if the <mode> parameter is incompatible with the input primitive type of the current geometry program.

The supported input primitive types are:

Points (POINTS)

Geometry programs that operate on points are valid only for the POINTS primitive type. There is a only a single vertex available for each program invocation: "vertex[0]" refers to the single point.

Lines (LINES)

Geometry programs that operate on line segments are valid only for the LINES, LINE_STRIP, and LINE_LOOP primitive types. There are two vertices available for each program invocation: "vertex[0]" and "vertex[1]" refer to the beginning and end of the line segment.

Lines with Adjacency (LINES_ADJACENCY)

Geometry programs that operate on line segments with adjacent vertices are valid only for the LINES_ADJACENCY_EXT and LINE_STRIP_ADJACENCY_EXT primitive types. There are four vertices available for each program invocation. "vertex[1]" and "vertex[2]" refer to the beginning and end of the line segment. "vertex[0]" and "vertex[3]" refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (TRIANGLES)

Geometry programs that operate on triangles are valid for the TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, and POLYGON primitive types.

When used with a geometry program that operates on triangles, QUADS, QUAD_STRIP, and POLYGON primitives are decomposed into triangles in an unspecified, implementation-dependent manner. For convex polygons (already required in the core GL specification), this decomposition satisfies three properties:

* the collection of triangles fully covers the area of the original primitive,

* no two triangles in the decomposition overlap, and

* the orientation of each triangle is consistent with the orientation of the original primitive.

For such primitives, the program is executed once for each triangle in the decomposition.

There are three vertices available for each program invocation. "vertex[0]", "vertex[1]", and "vertex[2]", refer to the first, second, and third vertex of the triangle, respectively.

Triangles with Adjacency (TRIANGLES_ADJACENCY)

Geometry programs that operate on triangles with adjacent vertices are valid for the TRIANGLES_ADJACENCY_EXT and TRIANGLE_STRIP_ADJACENCY_EXT primitive types. There are six vertices available for each program invocation. "vertex[0]", "vertex[2]", and "vertex[4]" refer to the first, second, and third vertex of the triangle respectively. "vertex[1]", "vertex[3]", and "vertex[5]" refer to the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

Section 2.14.6.2, Geometry Program Output Primitives

A geometry program can generate primitives of one of three types, as specified by the mandatory "PRIMITIVE_OUT" declaration. A geometry program will fail to load unless it contains exactly one such declaration.

The supported output primitive types are points (POINTS), line strips (LINE_STRIP), and triangle strips (TRIANGLE_STRIP). The vertices output by the geometry program are decomposed into points, lines, or triangles based on the output primitive type in the manner described in section 2.6.1.

Section 2.14.6.3, Geometry Program Execution Environment

Geometry programs execute using the instruction set documented in the GL_NV_gpu_program4 extension specification and in a manner similar to vertex programs. Each vertex attribute access must identify the vertex number being accessed. For example, "vertex[1].position" identifies the transformed position of "vertex[1]" as specified in teh description of the

input primitive type. Output vertices are specified by writing to vertex result variables in the same manner as done by vertex programs.

The special instruction "EMIT" specifies that a vertex is completed. A vertex is added to the current output primitive using the current values of the vertex result variables. The values of any unwritten result variables (or components) are undefined.

After an EMIT instruction is completed, the current values of all vertex result variables become undefined. If a program wants to ensure that the same result is used for every vertex written by the program, it is necessary to write the corresponding value once per vertex.

The special instruction "ENDPRIM" specifies that the current output primitive should be completed and a new output primitive should be started. A geometry program starts with an output primitive containing no vertices. When a geometry program terminates, the current output primitive is automatically completed. ENDPRIM has no effect if the geometry program's output primitive type is POINTS.

When a primitive generated by a geometry program is completed, the vertices added by the EMIT instruction are decomposed into points, lines, or triangles according to the output primitive type in the manner described in Section 2.8.1. The resulting primitives are then clipped and rasterized. If the number of vertices emitted by the geometry program is not sufficient to produce a single primitive, nothing is drawn.

Like vertex and fragment programs, geometry programs can access textures. The maximum number of texture image units that can be accessed by a geometry program is given by the value of MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT.

Section 2.14.6.4, Geometry Program Output Limits

A geometry program may not emit an limited in the number of vertices per invocation. Each geometry program must declare a vertex limit, which is the maximum number of vertices that the program can ever produce. The vertex limit is specified using the "VERTICES_OUT" declaration. A geometry program will fail to load unless it contains exactly one such declaration.

There are two implementation-dependent limits that limit the total number of vertices that a program can emit. First, the vertex limit may not exceed the value of MAX_PROGRAM_OUTPUT_VERTICES_NV. Second, product of the vertex limit and the number of result variable components written by the program (PROGRAM_RESULT_COMPONENTS_NV, as described in section 2.X.3.5 of NV_gpu_program4) may not exceed the value of MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV. A geometry program will fail to load if its maximum vertex count or maximum total component count exceeds the implementation-dependent limit. The limits may be queried by calling GetProgramiv with a <target> of GEOMETRY_PROGRAM_NV. Note that the maximum number of vertices that a geometry program can emit may be much lower than MAX_PROGRAM_OUTPUT_VERTICES_NV if the program writes a large number of result variable components.

After a geometry program is compiled, the vertex limit may be changed using the command

```
  void ProgramVertexLimitNV(enum target, int limit);
```

<target> must be GEOMETRY_PROGRAM_NV. <limit> is the new vertex limit, which must satisfy the two rules described above. The error INVALID_VALUE is generated if <limit> is less than or equal to zero, <limit> is greater than or equal to MAX_PROGRAM_OUTPUT_VERTICES_NV, or if the total number of components emitted would exceed MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV. The error INVALID_OPERATION is generated if the current geometry program has not been successfully loaded.

When a program executes, the number of vertices it emits should not exceed the vertex limit. Once a geometry program emits a number of vertices equal to the vertex limit, subsequent EMIT instructions may or may not have any effect.

Modify Section 2.X.2, Program Grammar

(replace third paragraph)

Geometry programs are required to begin with the header string "!!NVgp4.0". This header string identifies the subsequent program body as being a geometry program and indicates that it should be parsed according to the base NV_gpu_program4 grammar plus the additions below. Program string parsing begins with the character immediately following the header string.

(add the following grammar rules to the NV_gpu_program4 base grammar)

```
<declSequence>          ::= <declaration> <declSequence>

<instruction>           ::= <SpecialInstruction>

<attribUseV>            ::= <attribVarName> <arrayMem> <arrayMem>
                            <swizzleSuffix>

<attribUseS>            ::= <attribVarName> <arrayMem> <arrayMem>
                            <scalarSuffix>

<attribUseVNS>          ::= <attribVarName> <arrayMem> <arrayMem>

<resultUseW>            ::= <resultVarName> <arrayMem> <optWriteMask>
                          | <resultColor> <optWriteMask>
                          | <resultColor> "." <colorType> <optWriteMask>
                          | <resultColor> "." <faceType> <optWriteMask>
                          | <resultColor> "." <faceType> "." <colorType>
                            "." <optWriteMask>

<resultUseD>            ::= <resultColor>
                          | <resultColor> "." <colorType>
                          | <resultMulti>

<declaration>           ::= "PRIMITIVE_IN" <declPrimInType>
                          | "PRIMITIVE_OUT" <declPrimOutType>
                          | "VERTICES_OUT" <int>
```

```
<declPrimInType>        ::= "POINTS"
                          | "LINES"
                          | "LINES_ADJACENCY"
                          | "TRIANGLES"
                          | "TRIANGLES_ADJACENCY"

<declPrimOutType>       ::= "POINTS"
                          | "LINE_STRIP"
                          | "TRIANGLE_STRIP"

<SpecialInstruction>    ::= "EMIT"
                          | "ENDPRIM"

<attribBasic>           ::= <vtxPrefix> "position"
                          | <vtxPrefix> "fogcoord"
                          | <vtxPrefix> "pointsize"
                          | <attribTexCoord> <optArrayMemAbs>
                          | <attribClip> <arrayMemAbs>
                          | <attribGeneric> <arrayMemAbs>
                          | "primitive" "." "id"

<attribColor>           ::= <vtxPrefix> "color"

<attribMulti>           ::= <attribTexCoord> <arrayRange>
                          | <attribClip> <arrayRange>
                          | <attribGeneric> <arrayRange>

<attribTexCoord>        ::= <vtxPrefix> "texcoord"

<attribClip>            ::= <vtxPrefix> "clip"

<attribGeneric>         ::= <vtxPrefix> "attrib"

<vtxPrefix>             ::= "vertex" <optArrayMemAbs>

<resultBasic>           ::= <resPrefix> "position"
                          | <resPrefix> "fogcoord"
                          | <resPrefix> "pointsize"
                          | <resPrefix> "primid"
                          | <resPrefix> "layer"
                          | <resultTexCoord> <optArrayMemAbs>
                          | <resultClip> <arrayMemAbs>
                          | <resultGeneric> <arrayMemAbs>

<resultColor>           ::= <resPrefix> "color"

<resultMulti>           ::= <resultTexCoord> <arrayRange>
                          | <resultClip> <arrayRange>
                          | <resultGeneric> <arrayRange>

<resultTexCoord>        ::= <resPrefix> "texcoord"

<resultClip>            ::= <resPrefix> "clip"

<resultGeneric>         ::= <resPrefix> "attrib"
```

```
<resPrefix>             ::= "result" "."
```

(add the following subsection to section 2.X.3.2, Program Attribute Variables)

Geometry program attribute variables describe the attributes of each transformed vertex accessible to the geometry program. Most attributes correspond to the per-vertex results generated by vertex program execution or fixed-function vertex processing. The "primitive.id" attribute is generated specially, as described below.

If vertex programs are enabled, attributes will be obtained from the per-vertex outputs of the vertex program used to generate the vertex in question. Geometry program attributes should be read using the same component data type used to write the corresponding vertex program results. The value of any attribute corresponding to a vertex output not written by the vertex program is undefined.

If vertex programs are disabled, attributes will be obtained from the values computed by fixed-function vertex processing. All attributes, except for the primitive ID should be read as floating-point values in this case.

Geometry Vertex Binding	Components	Description
vertex[m].position	(x,y,z,w)	clip coordinates
vertex[m].color	(r,g,b,a)	front primary color
vertex[m].color.primary	(r,g,b,a)	front primary color
vertex[m].color.secondary	(r,g,b,a)	front secondary color
vertex[m].color.front	(r,g,b,a)	front primary color
vertex[m].color.front.primary	(r,g,b,a)	front primary color
vertex[m].color.front.secondary	(r,g,b,a)	front secondary color
vertex[m].color.back	(r,g,b,a)	back primary color
vertex[m].color.back.primary	(r,g,b,a)	back primary color
vertex[m].color.back.secondary	(r,g,b,a)	back secondary color
vertex[m].fogcoord	(f,-,-,-)	fog coordinate
vertex[m].pointsize	(s,-,-,-)	point size
vertex[m].texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex[m].texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex[m].attrib[n]	(x,y,z,w)	generic interpolant n
vertex[m].clip[n]	(d,-,-,-)	clip plane distance
vertex[m].texcoord[n..o]	(s,t,r,q)	array of texture coordinates
vertex[m].attrib[n..o]	(x,y,z,w)	array of generic interpolants
vertex[m].clip[n..o]	(d,-,-,-)	array of clip distances
vertex[m].id	(id,-,-,-)	vertex id
primitive.id	(id,-,-,-)	primitive number

Table X.2, Geometry Program Attribute Bindings. <m> refers to a vertex number, while <n>, and <o> refer to integer constants. Only the "vertex[m].texcoord" and "vertex.attrib" bindings are available in arrays.

For bindings that include "vertex[m]", <m> identifies the vertex number whose attributes are used for the binding. For bindings in explicit variable declarations, "[m]" is optional. If "[m]" is specified, <m> must be an integer constant and must be in the valid range of vertices supported for the input primitive type. If "[m]" is not specified, the

declared variable is accessed as an array, with the first array index specifying the vertex number. If such a variable is declared an array, it must have a second array index to identify the individual array element. For bindings used directly in instructions, "[m]" is required and must be an integer constant specifying a vertex number. The following examples illustrate various legal and illegal geometry program bindings and their meanings.

```
ATTRIB pos = vertex.position;
ATTRIB pos2 = vertex[2].position;
ATTRIB texcoords[] = { vertex.texcoord[0..3] };
ATTRIB tcoords1[4] = { vertex[1].texcoord[1..4] };
INT TEMP A0;
...
MOV R0, pos[1];                   # position of vertex 1
MOV R0, vertex[1].position;       # position of vertex 1
MOV R0, pos2;                     # position of vertex 2
MOV R0, texcoords[A0.x][1];       # texcoord 1 of vertex A0.x
MOV R0, texcoords[A0.x][A0.y];    # texcoord A0.y of vertex A0.x
MOV R0, tcoords1[2];              # texcoord 3 of vertex 1
MOV R0, vertex[A0.x].texcoord[1]; # ILLEGAL allowed -- vertex number
                                  #    must be constant here.
```

If a geometry attribute binding matches "vertex[m].position", the "x", "y", "z" and "w" components of the geometry attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of the transformed position of vertex <m>, in clip coordinates.

If a geometry attribute binding matches any binding in Table X.2 beginning with "vertex[m].color", the "x", "y", "z", and "w" components of the geometry attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding color of vertex <m>. Bindings containing "front" and "back" refer to the front and back colors, respectively. Bindings containing "primary" and "secondary" refer to primary and secondary colors, respectively. If face or color type is omitted in the binding, the binding is treated as though "front" and "primary", respectively, were specified.

If a geometry attribute binding matches "vertex[m].fogcoord", the "x" component of the geometry attribute variable is filled with the fog coordinate of vertex <m>. The "y", "z", and "w" components are undefined.

If a geometry attribute binding matches "vertex[m].pointsize", the "x" component of the geometry attribute variable is filled with the point size of vertex <m> computed by the vertex program. For fixed-function vertex processing, the point size attribute is undefined. The "y", "z", and "w" components are always undefined.

If a geometry attribute binding matches "vertex[m].texcoord" or "vertex[m].texcoord[n]", the "x", "y", "z", and "w" coordinates of the geometry attribute variable are filled with the "s", "t", "r", and "q" coordinates of texture coordinate set <n> of vertex <m>. If <n> is omitted, texture coordinate set zero is used.

If a geometry attribute binding matches "vertex[m].attrib[n]", the "x", "y", "z", and "w" components of the geometry attribute variable are filled with the "x", "y", "z", and "w" coordinates of generic interpolant <n> of

vertex <m>. All generic interpolants will be undefined when used with fixed-function vertex processing.

If a geometry attribute binding matches "vertex[m].clip[n]", the "x" component of the geometry attribute variable is filled the clip distance of vertex <m> for clip plane <n>, as written by the vertex program. If fixed-function vertex processing or position-invariant vertex programs are used, the clip distance is obtained by computing the per-clip plane dot product:

(p_1' p_2' p_3' p_4') dot (x_e y_e z_e w_e),

at the vertex location, as described in section 2.12. The clip distance for clip plane <n> is undefined if clip plane <n> is disabled. The "y", "z", and "w" components of the attribute are undefined.

If a geometry attribute binding matches "vertex[m].texcoord[n..o]", "vertex[m].attrib[n..o]", or "vertex[m].clip[n..o]", a sequence of 1+<o>-<n> texture coordinate bindings is created. For texture coordinate bindings, it is as though the sequence "vertex[m].texcoord[n], vertex[m].texcoord[n+1], ... vertex[m].texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a geometry attribute binding matches "vertex[m].id", the "x" component is filled with the vertex ID. If a vertex program is currently active, the attribute variable is filled with the vertex ID result written by the vertex program. If fixed-function vertex processing is used, the vertex ID is undefined. The "y", "z", and "w" components of the attribute are undefined.

If a geometry attribute binding matches "primitive.id", the "x" component is filled with the number of primitives received by the GL since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is zero. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter. The "y", "z", and "w" components of the variable are always undefined.

(add the following subsection to section 2.X.3.5, Program Results.)

Geometry programs emit vertices, and the set of result variables available to such programs correspond to the attributes of each emitted vertex. The set of allowable result variable bindings for geometry programs is given in Table X.3.

Binding	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing primary color
result.color.primary	(r,g,b,a)	front-facing primary color
result.color.secondary	(r,g,b,a)	front-facing secondary color
result.color.front	(r,g,b,a)	front-facing primary color
result.color.front.primary	(r,g,b,a)	front-facing primary color
result.color.front.secondary	(r,g,b,a)	front-facing secondary color
result.color.back	(r,g,b,a)	back-facing primary color
result.color.back.primary	(r,g,b,a)	back-facing primary color
result.color.back.secondary	(r,g,b,a)	back-facing secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
result.attrib[n]	(x,y,z,w)	generic interpolant n
result.clip[n]	(d,*,*,*)	clip plane distance
result.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
result.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
result.clip[n..o]	(d,*,*,*)	clip distances n thru o
result.primid	(id,*,*,*)	primitive id
result.layer	(l,*,*,*)	layer for cube/array/3D FBOs

Table X.3: Geometry Program Result Variable Bindings. Components labeled "*" are unused.

If a result variable binding matches "result.position", updates to the "x", "y", "z", and "w" components of the result variable modify the "x", "y", "z", and "w" components, respectively, of the transformed vertex's clip coordinates. Final window coordinates will be generated for the vertex as described in section 2.14.4.4.

If a result variable binding match begins with "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the corresponding vertex color attribute in Table X.3. Color bindings that do not specify "front" or "back" are consided to refer to front-facing colors. Color bindings that do not specify "primary" or "secondary" are considered to refer to primary colors.

If a result variable binding matches "result.fogcoord", updates to the "x" component of the result variable set the transformed vertex's fog coordinate. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.pointsize", updates to the "x" component of the result variable set the transformed vertex's point size. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.texcoord" or "result.texcoord[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "s", "t", "r" and "q" components, respectively, of the transformed vertex's texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is selected.

If a result variable binding matches "result.attrib[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "x", "y", "z", and "w" components of the generic interpolant <n>.

If a result variable binding matches "result.clip[n]", updates to the "x" component of the result variable set the clip distance for clip plane <n>.

If a result variable binding matches "result.texcoord[n..o]", "result.attrib[n..o]", or "result.clip[n..o]", a sequence of 1+<o>-<n> bindings is created. For texture coordinates, it is as though the sequence "result.texcoord[n], result.texcoord[n+1], ... result.texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a result variable binding matches "result.primid", updates to the "x" component of the result variable provide a single integer that serves as a primitive identifier. The written primitive ID is available to fragment programs using the "primitive.id" attribute binding. If a fragment program using primitive IDs is active and a geometry program is also active, the geometry program must write "result.primid" or the primitive ID number is undefined.

If a result variable binding matches "result.layer", updates to the "x" component of the result variable provide a single integer that serves as a layer selector for layered rendering (section 2.14.6.5). The layer must be written as an integer value; writing a floating-point layer number will produce undefined results.

(modify Table X.13 in section 2.X.4, Program Instructions, to include the following.)

	Modifiers								
Instruction	F	I	C	S	H	D	Inputs	Out	Description
EMIT	-	-	-	-	-	-	-	-	emit vertex
ENDPRIM	-	-	-	-	-	-	-	-	end of primitive

(add the following subsection to section 2.X.5, Program Options.)

Section 2.X.5.Y, Geometry Program Options

No options are supported at present for geometry programs.

(add the following subsection to section 2.X.6, Program Declarations.)

Section 2.X.6.Y, Geometry Program Declarations

Geometry programs support three types of declaration statements, as described below. Each of the three must be included exactly once in the geometry program.

- Input Primitive Type (PRIMITIVE_IN)

The PRIMITIVE_IN statement declares the type of primitives seen by a geometry program. The single argument must be one of "POINTS", "LINES", "LINES_ADJACENCY", "TRIANGLES", or "TRIANGLES_ADJACENCY".

- Output Primitive Type (PRIMITIVE_OUT)

The PRIMITIVE_OUT statement declares the type of primitive emitted by a geometry program. The single argument must be one of "POINTS", "LINE_STRIP", or "TRIANGLE_STRIP".

- Maximum Vertex Count (VERTICES_OUT)

The VERTICES_OUT statement declares the maximum number of vertices that may be emitted by a geometry program. The single argument must be a positive integer. A vertex program that emits more than the specified number of vertices may terminate abnormally.

(add the following subsections to section 2.X.7, Program Instruction Set.)

Section 2.X.7.Z, EMIT: Emit Vertex

The EMIT instruction emits a new vertex to be added to the current output primitive of a geometry program. The attributes of the emitted vertex are given by the current values of the vertex result variables. After the EMIT instruction completes, a new vertex is started and all result variables become undefined.

Section 2.X.7.Z, ENDPRIM: End of Primitive

A geometry program can emit multiple primitives in a single invocation. The ENDPRIM instruction is used in a geometry program to signify the end of the current primitive and the beginning of a new primitive of the same type. The effect of ENDPRIM is roughly equivalent to calling End followed by a new Begin, where the primitive mode is specified in the text of the geometry program.

Like End, the ENDPRIM instruction does not emit a vertex. Any result registers written prior to an ENDPRIM instruction are unchanged, and will be used in the vertex specified by the next EMIT instruction if they are not overwritten first.

When geometry program execution completes, the current primitive is automatically terminated. It is not necessary to include an ENDPRIM instruction if the geometry program writes only a single primitive.

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

Modify Section 3.3, Points (p. 95)

(replace all Section 3.3 text on p. 95) A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If no vertex or geometry program is active, the size of the point is controlled by

```
  void PointSize(float size);
```

<size> specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

```
   derived size = clamp(size * sqrt(1/(a+b*d+c*d^2)))
```

where d is the eye-coordinate distance from the eye, (0,0,0,1) in eye coordinates, to the vertex, and a, b, and c are distance attenuation function coefficients.

If a vertex or geometry program is active, the derived size depends on the per-vertex point size mode enable. Per-vertex point size mode is enabled or disabled by calling Enable or Disable with the symbolic value PROGRAM_POINT_SIZE_EXT. If per-vertex point size is enabled and a geometry program is active, the point size is taken from the point size emitted by the geometry program. If per-vertex point size is enabled an no geometry program is active, the point size is taken from the point size result of the vertex program. Otherwise, the point size is taken from the <size> value provided to PointSize, with no distance attenuation applied. In all cases, the point size is clamped to the implementation-dependent point size range.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width. ...

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.5 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when geometry program mode is enabled and the currently bound geometry program object does not contain a valid geometry program.

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when geometry program mode is enabled and:

* the input primitive type of the current geometry program is POINTS and <mode> is not POINTS,

* the input primitive type of the current geometry program is LINES and <mode> is not LINES, LINE_STRIP, or LINE_LOOP,

* the input primitive type of the current geometry program is TRIANGLES and <mode> is not TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, or POLYGON,

* the input primitive type of the current geometry program is LINES_ADJACENCY and <mode> is not LINES_ADJACENCY_EXT or LINE_STRIP_ADJACENCY_EXT, or

* the input primitive type of the current geometry program is TRIANGLES_ADJACENCY and <mode> is not TRIANGLES_ADJACENCY_EXT or TRIANGLE_STRIP_ADJACENCY_EXT.

The error INVALID_ENUM is generated if GetProgramivARB is called with a <pname> of MAX_PROGRAM_OUTPUT_VERTICES_NV or MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV and the target isn't GEOMETRY_PROGRAM_NV.

Dependencies on EXT_framebuffer_object

If EXT_framebuffer_object (or similar functionality) is not supported, the "result.layer" binding should be removed. "FramebufferTextureEXT" and "FramebufferTextureLayerEXT" should be removed from "New Procedures and Functions", and FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT, and FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT should be removed from "New Tokens".

Otherwise, this extension modifies EXT_framebuffer_object to add the notion of layered framebuffer attachments and framebuffers that can be used in conjunction with geometry programs to allow programs to direct primitives to a face of a cube map or layer of a three-dimensional texture or one- or two-dimensional array texture. The layer used for rendering can be selected by the geometry program at run time.

(insert before the end of Section 4.4.2, Attaching Images to Framebuffer Objects)

There are several types of framebuffer-attachable images:

* the image of a renderbuffer object, which is always two-dimensional,

* a single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one,

* a single level of a two-dimensional or rectangle texture,

* a single face of a cube map texture level, which is treated as a two-dimensional image, or

* a single layer of a one- or two-dimensional array texture or three-dimensional texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be layered.

(replace section 4.4.2.3, "Attaching Texture Images to a Framebuffer")

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines CopyTexImage{1D|2D}, and CopyTexSubImage{1D|2D|3D}. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

```
void FramebufferTextureEXT(enum target, enum attachment,
                           uint texture, int level);
```

<target> must be FRAMEBUFFER_EXT. <attachment> must be one of the attachment points of the framebuffer listed in table 1.nnn.

If <texture> is zero, any image or array of images attached to the attachment point named by <attachment> is detached, and the state of the attachment point is reset to its initial values. <level> is ignored if <texture> is zero.

If <texture> is non-zero, FramebufferTextureEXT attaches level <level> of the texture object named <texture> to the framebuffer attachment point named by <attachment>. The error INVALID_VALUE is generated if <texture> is not the name of a texture object, or if <level> is not a supported texture level number for textures of the type corresponding to <target>. The error INVALID_OPERATION is generated if <texture> is the name of a buffer texture.

If <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture, the texture level attached to

the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

The command

```
  void FramebufferTextureLayerEXT(enum target, enum attachment,
                                  uint texture, int level, int layer);
```

operates like FramebufferTextureEXT, except that only a single layer of the texture level, numbered <layer>, is attached to the attachment point. If <texture> is non-zero, the error INVALID_VALUE is generated if <layer> is negative, or if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional or one- or two-dimensional array texture.

The command

```
  void FramebufferTextureFaceEXT(enum target, enum attachment,
                                 uint texture, int level, enum face);
```

operates like FramebufferTextureEXT, except that only a single face of a cube map texture, given by <face>, is attached to the attachment point. <face> is one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z. If <texture> is non-zero, the error INVALID_VALUE is generated if <texture> is not the name of a texture object. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a cube map texture.

The command

```
  void FramebufferTexture1DEXT(enum target, enum attachment,
                               enum textarget, uint texture, int level);
```

operates identically to FramebufferTextureEXT, except for two additional restrictions. If <texture> is non-zero, the error INVALID_ENUM is generated if <textarget> is not TEXTURE_1D and the error INVALID_OPERATION is generated unless <texture> is the name of a one-dimensional texture.

The command

```
  void FramebufferTexture2DEXT(enum target, enum attachment,
                               enum textarget, uint texture, int level);
```

operates similarly to FramebufferTextureEXT. If <textarget> is TEXTURE_2D or TEXTURE_RECTANGLE_ARB, <texture> must be zero or the name of a two-dimensional or rectangle texture. If <textarget> is TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, <texture> must be zero or the name of a cube map texture. For cube map textures, only the single face of the cube map texture level given by <textarget> is attached. The error INVALID_ENUM is generated if <texture> is not zero and <textarget> is not one of the values enumerated above. The error INVALID_OPERATION is generated if <texture> is the name of a texture whose type does not match the texture type required by <textarget>.

The command

```
  void FramebufferTexture3DEXT(enum target, enum attachment,
                               enum textarget, uint texture,
                               int level, int zoffset);
```

behaves identically to FramebufferTextureLayerEXT, with the <layer> parameter set to the value of <zoffset>. The error INVALID_ENUM is generated if <textarget> is not TEXTURE_3D. The error INVALID_OPERATION is generated unless <texture> is zero or the name of a three-dimensional texture.

For all FramebufferTexture commands, if <texture> is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to <attachment> is updated based on the new attachment. FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is set to TEXTURE, FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT is set to <texture>, and FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to <level>. FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_FACE is set to <textarget> if FramebufferTexture2DEXT is called and <texture> is the name of a cubemap texture; otherwise, it is set to TEXTURE_CUBE_MAP_POSITIVE_X. FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT is set to <layer> or <zoffset> if FramebufferTextureLayerEXT or FramebufferTexture3DEXT is called; otherwise, it is set to zero. FRAMEBUFFER_ATTACHMENT_LAYERED_EXT is set to TRUE if FramebufferTextureEXT is called and <texture> is the name of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture; otherwise it is set to FALSE.

(modify Section 4.4.4.1, Framebuffer Attachment Completeness -- add to the conditions necessary for attachment completeness)

The framebuffer attachment point <attachment> is said to be "framebuffer attachment complete" if ...:

  * If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a three-dimensional texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the depth of the texture.

  * If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME_EXT names a one- or two-dimensional array texture, FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT must be smaller than the number of layers in the texture.

(modify section 4.4.4.2, Framebuffer Completeness -- add to the list of conditions necessary for completeness)

  * If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target (i.e., three-dimensional, cube map, or one- or two-dimensional array textures).
    { FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS_EXT }

  * If any framebuffer attachment is layered, all attachments must have the same layer count. For three-dimensional textures, the layer count is the depth of the attached volume. For cube map textures, the layer count is always six. For one- and two-dimensional array textures, the

```
    layer count is simply the number of layers in the array texture.
    { FRAMEBUFFER_INCOMPLETE_LAYER_COUNT_EXT }

The enum in { brackets } after each clause of the framebuffer completeness
rules specifies the return value of CheckFramebufferStatusEXT (see below)
that is generated when that clause is violated. ...

(add section 4.4.7, Layered Framebuffers)

A framebuffer is considered to be layered if it is complete and all of its
populated attachments are layered.  When rendering to a layered
framebuffer, each fragment generated by the GL is assigned a layer number.
The layer number for a fragment is zero if

  * the fragment is generated by DrawPixels, CopyPixels, or Bitmap,

  * geometry programs are disabled, or

  * the current geometry program does not contain an instruction that
    writes to the layer result binding.

Otherwise, the layer for each point, line, or triangle emitted by the
geometry program is taken from the layer output of the provoking vertex.
For line strips, the provoking vertex is the second vertex of each line
segment.  For triangle strips, the provoking vertex is the third vertex of
each individual triangles.  The per-fragment layer can be different for
fragments generated by each individual point, line, or triangle emitted by
a single geometry program invocation.  A layer number written by a
geometry program has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment's layer
number selects an image from the array of images at each attachment point
from which to obtain the destination R, G, B, A values for blending
(Section 4.1.8) and to which to write the final color values for that
attachment.  If the fragment's layer number is negative or greater than
the number of layers attached, the effects of the fragment on the
framebuffer contents are undefined.

When the Clear command is used to clear a layered framebuffer attachment,
all layers of the attachment are cleared.

When commands such as ReadPixels or CopyPixels read from a layered
framebuffer, the image at layer zero of the selected attachment is always
used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there
are six layers attached, numbered zero through five.  Each layer number is
mapped to a cube map face, as indicated in Table X.4.
```

layer number	cube map face
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

Table X.4, Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

(modify Section 6.1.3, Enumerated Queries -- Modify/add to list of <pname> values for GetFramebufferAttachmentParameterivEXT if FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE_EXT is TEXTURE)

If <pname> is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER_EXT and the attached image is a layer of a three-dimensional texture or one- or two-dimensional array texture, then <params> will contain the specified layer number. Otherwise, <params> will contain the value zero.

If <pname> is FRAMEBUFFER_ATTACHMENT_LAYERED_EXT, then <params> will contain TRUE if an entire level of a three-dimesional texture, cube map texture, or one- or two-dimensional array texture is attached to the <attachment>. Otherwise, <params> will contain FALSE.

(Modify the Additions to Chapter 5, section 5.4)

Add the commands FramebufferTextureEXT, FramebufferTextureLayerEXT, and FramebufferTextureFaceEXT to the list of commands that are not compiled into a display list, but executed immediately.

Dependencies on EXT_framebuffer_blit

If EXT_framebuffer_blit is supported, the EXT_framebuffer_object language should be further amended so that <target> values passed to FramebufferTextureEXT and FramebufferTextureLayerEXT can be DRAW_FRAMEBUFFER_EXT or READ_FRAMEBUFFER_EXT, and that those functions set/query state for the draw framebuffer if <target> is FRAMEBUFFER_EXT.

Dependencies on EXT_texture_array

If EXT_texture_array is not supported, the discussion array textures the layered rendering edits to EXT_framebuffer_object should be removed. Layered rendering to cube map and 3D textures would still be supported.

If EXT_texture_array is supported, the edits to EXT_framebuffer_object supersede those made in EXT_texture_array, except for language pertaining to mipmap generation of array textures.

There are no functional incompatibilities between the FBO support in these two specifications. The only differences are that this extension supports layered rendering and also rewrites certain sections of the core FBO specification more aggressively.

Dependencies on ARB_texture_rectangle

If ARB_texture_rectangle is not supported, all references to rectangle textures in the EXT_framebuffer_object spec language should be removed.

Dependencies on EXT_texture_buffer_object

If EXT_buffer_object is not supported, the reference to an INVALID_OPERATION error if a buffer texture is passed to FramebufferTextureEXT should be removed.

Dependencies on NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count). If NV_primitive_restart is not supported, references to that extension in the discussion of the "primitive.id" attribute should be removed.

New State

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
GEOMETRY_PROGRAM_NV	B	IsEnabled	FALSE	Geometry shader enable	2.14.6	enable/transform
FRAMEBUFFER_ATTACHMENT_LAYERED_EXT	nxB	GetFramebuffer-Attachment-ParameterivEXT	FALSE	Framebuffer attachment is layered	4.4.2.3	-
GEOMETRY_VERTICES_OUT_EXT	Z+	GetProgramivARB	0	vertex limit of the current geometry program	2.14.6.4	-
GEOMETRY_INPUT_TYPE_EXT	Z+	GetProgramivARB	0	input primitive type of the current geometry program	2.14.6.4	-
GEOMETRY_OUTPUT_TYPE_EXT	Z+	GetProgramivARB	0	output primitive type of the current geometry program	2.14.6.4	-

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attrib
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS_EXT	Z+	GetIntegerv	16	maximum number of texture image units accessible in a geometry program	2.14.6.3	-
MAX_PROGRAM_OUTPUT_VERTICES_NV	Z+	GetProgramivARB	256	maximum number of vertices that any geometry program could emit	2.14.6.4	-
MAX_PROGRAM_TOTAL_OUTPUT_COMPONENTS_NV	Z+	GetProgramivARB	1024	maximum number of result components (all vertices) that a geometry program can emit	2.14.6.4	-

NVIDIA Implementation Details

Because of a hardware limitation, some GeForce 8 series chips use the odd vertex of an incomplete TRIANGLE_STRIP_ADJACENCY_EXT primitive as a replacement adjacency vertex rather than ignoring it.

Issues

(1) How do geometry programs fit into the existing GL pipeline?

RESOLVED: The following diagram illustrates how geometry programs fit into the "vertex processing" portion of the GL (Chapter 2 of the OpenGL 2.0 Specification).

First, vertex attributes are specified via immediate-mode commands or through vertex arrays. They can be conventional attributes (e.g., glVertex, glColor, glTexCoord) or generic (numbered) attributes.

Vertices are then transformed, either using a vertex program or fixed-function vertex processing. Fixed-function vertex processing includes position transformation (modelview and projection matrices), lighting, texture coordinate generation, and other calculations. The results of either method are a "transformed vertex", which has a position (in clip coordinates), front and back colors, texture coordinates, generic attributes (vertex program only), and so on. Note that on many current GL implementations, vertex processing is performed by executing a "fixed function vertex program" generated by the driver.

After vertex transformation, vertices are assembled into primitives, according to the topology (e.g., TRIANGLES, QUAD_STRIP) provided by the call to glBegin(). Primitives are points, lines, triangles, quads, or polygons. Many GL implementations do not directly support quads or polygons, but instead decompose them into triangles as permitted by the spec.

After initial primitive assembly, a geometry program is executed on each individual point, line, or triangle primitive, if enabled. It can read the attributes of each transformed vertex, perform arbitrary computations, and emit new transformed vertices. These emitted vertices are themselves assembled into primitives according to the output primitive type of the geometry program.

Then, the colors of the vertices of each primitive are clamped to [0,1] (if color clamping is enabled), and flat shading may be performed by taking the color from the provoking vertex of the primitive.

Each primitive is clipped to the view volume, and to any enabled user-defined clip planes. Color, texture coordinate, and other attribute values are computed for each new vertex introduced by clipping.

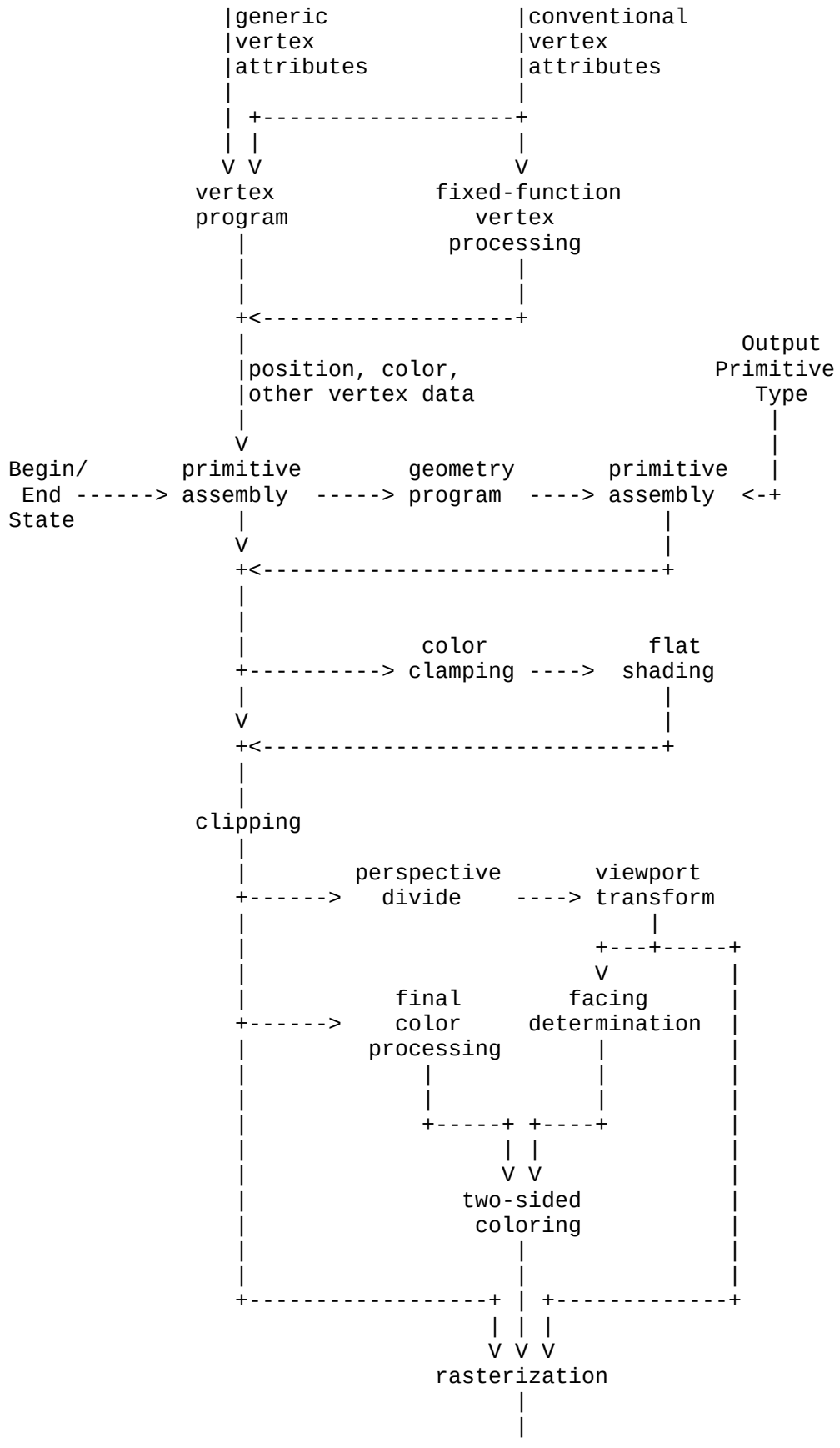
After clipping, the position of each vertex (in clip coordinates) is converted to normalized device coordinates in the perspective division (divide by w) step, and to window coordinates in the viewport transformation step.

At the same time, color values may be converted to normalized fixed-point values according to the "Final Color Processing" portion of the specification.

After the vertices of the primitive are transformed to window coordinate, the GL determines if the primitive is front- or back-facing. That information is used for two-sided color selection, where a single set of colors is selected from either the front or back colors associated with each transformed vertex.

When all this is done, the final transformed position, colors (primary and secondary), and other attributes are used for rasterization (Chapter 3 in the OpenGL 2.0 Specification).

When the raster position is specified (via glRasterPos), it goes through the entire vertex processing pipeline as though it were a point. However, geometry programs are never run on the raster position.


generic
vertex
attributes
conventional
vertex
attributes
vertex
program
fixed-function
vertex
processing
position, color,
other vertex data
Output
Primitive
Type
Begin/
End
State
primitive
assembly
geometry
program
primitive
assembly
color
clamping
flat
shading
clipping
perspective
divide
viewport
transform
final
color
processing
facing
determination
two-sided
coloring
rasterization

V

(2) Why is this called GL_NV_geometry_program4? There aren't any previous versions of this extension, let alone three?

RESOLVED: The instruction set for GPU programs of all types (vertex, fragment, and now geometry) have been unified in the GL_NV_gpu_program4 extension, and the "4" suffix in this extension name indicates the instruction set type. There are three previous NV_vertex_program variants (four if you count NV_vertex_program1_1), so "4" is the next available numeric suffix.

(3) Should the GL produce errors at Begin time if an application specifies a primitive mode that is "incompatible" with the geometry program? For example, if the geometry program operates on triangles and the application sends a POINTS primitive?

RESOLVED: Yes. Mismatches of app-specified primitive types and geometry program input primitive types seem like clear errors and would produce weird and wonderful effects.

(4) Can the input primitive type of a geometry program be changed at run time?

RESOLVED: Not in this extension. Each geometry program has a single input primitive type, and vertices are presented to the program in a specific order based on that type.

(5) Can the output primitive type of a geometry program be determined at run time?

RESOLVED: Not in this extension.

(6) Must the output primitive type of a geometry program match the input primitive type in any way?

RESOLVED: No, you can have a geometry program generate points out of triangles or triangles out of points. Some combinations are analogous to existing OpenGL operations: reading triangles and writing points or line strips can be used to emulate a subset of PolygonMode functionality. Reading points and writing triangle strips can be used to emulate point sprites.

(7) Are primitives emitted by a geometry program processed like any other OpenGL primitive?

RESOLVED: Yes. Antialiasing, stippling, polygon offset, polygon mode, culling, two-sided lighting and color selection, point sprite operations, and fragment processing all work as expected.

One limitation is that the only output primitive types supported are points, line strips, and triangle strips, none of which meaningfully support edge flags that are sometimes used in conjunction with the POINT and LINE polygon modes (edge flags are always ignored for line-mode triangle strips).

(8) Should geometry programs support additional input primitive types?

RESOLVED: Possibly in a future extension. It should be straightforward to build a future extension to support geometry programs that operate on quads. Other primitive types might be more demanding on hardware. Quads with adjacency would require 12 vertices per program execution. General polygons may require even more, since there is no fixed bound on the number of vertices in a polygon.

(9) Should geometry programs support additional output primitive types?

RESOLVED: Possibly in a future extension. Additional output types (e.g., independent lines, line loops, triangle fans, polygons) may be useful in the future; triangle fans/polygons seem particularly useful.

(10) Should we provide additional adjacency primitive types that can be used inside a Begin/End?

RESOLVED: Not in this extension. It may be desirable to add new primitive types (e.g., TRIANGLE_FAN_ADJACENCY) in a future extension.

(11) How do geometry programs interact with RasterPos?

RESOLVED: Geometry programs are ignored when specifying the raster position. While the raster position could be treated as a point, turning it into a triangle strip would be quite bizarre.

(12) How do geometry programs interact with pixel primitives (DrawPixels, Bitmap)?

RESOLVED: They do not. Fragments generated be DrawPixels and Bitmap are injected into the pipeline after the point where geometry program execution occurs.

(13) Is there a limit on the number of vertices that can be emitted by a geometry program?

RESOLVED: Unfortunately, yes. Besides practical hardware limits, there may also be practical performance advantages when applications guarantee a tight upper bound on the number of vertices a geometry shader will emit. GPUs frequently excecute programs in parallel, and there are substantial implementation challenges to parallel execution of geometry threads that can write an unbounded number of results, particular given that the all the primitives generated by the first geometry program invocation must be consumed before any of the primitives generated by the second program invocation. Limiting the amount of data a geometry program can write substantially eases the implementation burden.

A geometry program must declare a maximum number of vertices that can be emitted, called the vertex limit. There is an implementation-dependent limit on the total number of vertices a program can emit (256 minimum) and the product of the vertex limit and the number of active result components (1024 minimum). A program will fail to load if doesn't declare a limit or exceeds either of the two implementatoin-dependent limits.

It would be ideal if the limit could be inferred from the instructions in the program itself, and that would be possible for many programs, particularly ones with straight-line flow control. For programs with more complicated flow control (subroutines, data-dependent looping, and so on), it would be impossible to make such an inference and a "safe" limit would have to be used with adverse and possibly unexpected performance consequences.

The limit on the number of EMIT instructions that can be issued can not always be enforced at compile time, or even at Begin time. We specify that if a program tries to emit more vertices than the vertex limit allows, emits that exceed the limit may or may not have any effect.

(14) Should it be possible to change the limit on the number of vertices emitted by a geometry program after the program is specified?

RESOLVED: Yes, using the function ProgramVertexLimitNV(). Applications may want to tweak a piece of data that affects the number of vertices emitted, but doesn't necessarily require recompiling the entire program. Examples might be a "circular point sprite" program, that reads a single point, and draws a circle centered at that point with <N> vertices. An application could change the value <N> at run time, but it could require a change in the vertex limit. Another example might be a geometry program that does some fancy subdivision, where the relevant parameter might be a limit on how far the primitive is subdivided.

Ideally, this program object state should be set by a "program parameter" command, much like texture state is set by a "texture parameter" (TexParameter) command. Unfortunately, there are already several different "program parameter" functions:

```
ProgramEnvParameter4fARB()   -- sets global environment constants
ProgramLocalParameter4fARB() -- sets per-program constants
ProgramParameter4fNV()       -- also sets global environment constants
```

Additionally, GLSL and OpenGL 2.0 introduced "program objects" which are linked collections of vertex, fragment, and now geometry shaders. A GLSL vertex "shader" is equivalent to an ARB_vertex_program vertex "program", which is nothing like a GLSL program. As of OpenGL 2.0, GLSL programs do not have settable parameters, by subsequent extensions may want to add them (for example, EXT_geometry_shader4, which has this same functionality for GLSL). If that happens, they would want their own ProgramParameter API, but with a different prototype than this extension would want.

Naming this function "ProgramVertexLimitNV" sidesteps this issue for now.

(15) How do edge flags interact with adjacency primitives?

RESOLVED: If geometry programs are disabled, adjacency primitives are still supported. For TRIANGLES_ADJACENCY_EXT, edge flags will apply as they do for TRIANGLES. Such primitives are rendered as independent triangles as though the adjacency vertices were not provided. Edge flags for the "real" vertices are supported. For all other adjacency primitive types, edge flags are irrelevant.

(16) How do geometry programs interact with color clamping?

RESOLVED: Geometry program execution occurs prior to color clamping in the pipeline. This means the colors written by vertex programs or fixed-function vertex processing are not clamped to [0,1] before they are read by geometry programs. If color clamping is enabled, any vertex colors written by the geometry program will have their components clamped to [0,1].

(17) How are QUADS, QUAD_STRIP, and POLYGON primitives decomposed into triangles in the initial implementation of GL_NV_geometry_program4?

RESOLVED: The specification leaves the decomposition undefined, subject to a small number of rules. Assume that four vertices are specified in the order V0, V1, V2, V3.

For QUADS primitives, the quad V0->V1->V2->V3 is decomposed into the triangles V0->V1->V2, and V0->V2->V3. The provoking vertex of the quad (V3) is only found in the second triangle. If it's necessary to flat shade over an entire quad, take the attributes from V0, which will be the first vertex for both triangles in the decomposition.

For QUAD_STRIP primitives, the quad V0->V1->V3->V2 is decomposed into the triangles V0->V1->V3 and V2->V0->V3. This has the property of leaving the provoking vertex for the polygon (V3) as the third vertex for each triangle of the decomposition.

For POLYGON primitives, the polygon V0->V1->V2->V3 is decomposed into the triangles V1->V2->V0 and V2->V3->V0. This has the property of leaving the provoking vertex for the polygon (V0) as the third vertex for each triangle of the decomposition.

(18) Should geometry programs be able to select a layer of a 3D texture, cube map texture, or array texture at run time? If so, how?

RESOLVED: This extension provides a per-vertex result binding called "result.layer", which is an integer specifying the layer to render to. When an each individual point, line, or triangle is emitted by a geometry program, the layer number is taken from the provoking (last) vertex of the primitive and is used for all fragments generated by that primitive.

The EXT_framebuffer_object (FBO) extension is used for rendering to textures, but for cube maps and 3D textures, it only provides the ability to attach a single face or layer of such textures.

This extension generalizes FBO by creates new entry points to bind an entire texture level (FramebufferTextureEXT) or a single layer of a texture level (FramebufferTextureLayerEXT) to an attachment point. The existing FBO binding functions, FramebufferTexture[123]DEXT are retained, and are defined in terms of the more general new functions.

The new functions do not have a dimension in the function name or a <textarget> parameter, which can be inferred from the provided texture. They can do anything that the old functions can do, except attach a single face of a cube map texture. We considered adding a separate function FramebufferTextureFaceEXT to provide this functionality, but

decided that the existing FramebufferTexture2DEXT API was adequate. We also considered using FramebufferTextureLayerEXT for this purpose, but it was not clear whether a layer number (0-5) or face enum (e.g, TEXTURE_CUBE_MAP_POSITIVE_X) should be provided.

When an entire texel level of a cube map, 3D, or array texture is attached, that attachment is considered layered. The framebuffer is considered layered if any attachment is layered. When the framebuffer is layered, there are three additional completeness requirements:

* all attachments must be layered
* all color attachments must be from textures of identical type
* all attachments must have the same number of layers

We expect subsequent versions of the FBO spec to relax the requirement that all attachments must have the same width and height, and plan to relax the similar requirement for layer count at that time.

When rendering to a layered framebuffer, layer zero is used unless a geometry program that writes the layer result is enabled. When rendering to a non-layered framebuffer, any layer result emitted from geometry programs is ignored and the set of single-image attachments are used. When reading from a layered framebuffer (e.g., ReadPixels), layer zero is always used. When clearing a layered framebuffer, all layers are cleared to the corresponding clear values.

Several other approaches were considered, including leveraging existing FBO attachment functions and requiring the use of FramebufferTexture3D with a <zoffset> of zero to make a framebuffer attachment "layerable" (attaching layer zero means that the attachment could be used for either layered- or non-layered rendering). Whether rendering was layered or not could either be inferred from the active geometry program, or set as a new property of the framebuffer object. There is presently FramebufferParameter API to set a property of a framebuffer, so it would have been necessary to create new set/query APIs if this approach were chosen.

(19) *How can single-pass cube map rendering be done efficiently in a geometry program?*

UNRESOLVED: To do single-pass cubemap rendering, attach entire cube map textures to framebuffer attachment points using the new functions provided by this extension. The vertex program used should only transform the vertex position to eye coordinates (position relative to the center of the cube map). A geometry program should be used that effectively projects each input triangle onto each of the six faces of the cube map, emitting a triangle for each. Each of the projected vertices should be emitted with a "result.layer" value matching the face number (0-5). When the projected triangle is drawn, it is automatically drawn on the face corresponding to the emitted layer number.

It should be simple to skip projecting primitives onto faces they won't touch. For example, if all of the X eye coordinates are positive, there is no reason to project to the "negative X" cube map face.

An example should be provided for this issue.

(20) How should per-vertex point size work with geometry programs?

RESOLVED: We will generalize the existing VERTEX_PROGRAM_POINT_SIZE enable to control the point size behavior if either vertex or geometry programs are enabled.

If geometry programs are enabled, the point size is taken from the point size result of the emitted vertex if VERTEX_PROGRAM_POINT_SIZE is enabled, or from the PointSize state otherwise.

If no geometry program is enabled, it works like OpenGL 2.0. If a vertex program is active, it's taken from the point size result or PointSize state, depending on the VERTEX_PROGRAM_POINT_SIZE enable. If no program is enabled, normal fixed-function point size handling (including distance attenuation) is supported.

This extension creates a new alias for the VERTEX_PROGRAM_POINT_SIZE enum, called PROGRAM_POINT_SIZE_EXT, to reflect that the point size enable now covers multiple program types. Both enums have the same value.

(21) How do vertex IDs work with geometry programs?

RESOLVED: Vertex IDs are automatically provided to vertex programs when applicable, via the "vertex.id" binding. However, they are not automatically copied the transformed vertex results that are read by geometry programs.

Geometry programs can read the ID of vertex <n> via the "vertex[<n>].id" binding, but the vertex ID must have been copied by the vertex program using an instruction such as:

```
MOV result.id.x, vertex.id.x;
```

If a vertex program doesn't write vertex ID, or fixed-function vertex processing is used, the vertex ID visible to geometry programs is undefined.

(22) How do primitive IDs work with geometry programs?

RESOLVED: Primitive IDs are automatically available to geometry programs via the "primitive.id" binding and indicate the number of input primitives previously processed since the last explicit or implicit Begin call.

If a geometry program wants to make the primitive ID available to a fragment program, it should copy the appropriate value to the "result.primid" binding.

(23) How do primitive IDs work with primitives not supported directly by geometry program input topologies (e.g., QUADS, POLYGON)?

RESOLVED: QUADS are decomposed into two triangles. Both triangles will have the same primitive ID, which is the number of full quads previously processed. POLYGON primitives are decomposed into a series of triangles, and all of them will have the primitive ID -- zero.

(24) This is an NV extension (NV_geometry_program4). Why do some of the new tokens have an "EXT" extension?

RESOLVED: Some of the tokens are shared between this extension and the comparable high-level GLSL programmability extension (EXT_geometry_shader4). Rather than provide a duplicate set of tokens, we simply use the EXT versions here. The tokens specific to assembly shader uses retain an NV suffix.

Revision History

None

Name

NV_geometry_shader4

Name String

GL_NV_geometry_shader4

Contact

Pat Brown, NVIDIA (pbrown 'at' nvidia.com)
Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 01/10/2007
Author revision: 16

Number

338

Dependencies

OpenGL 1.1 is required.

EXT_geometry_shader4 is required.

This extension is written against the EXT_geometry_shader4 and OpenGL 2.0 specifications.

Overview

This extension builds upon the EXT_geometry_shader4 specification to provide two additional capabilities:

* Support for QUADS, QUAD_STRIP, and POLYGON primitive types when geometry shaders are enabled. Such primitives will be tessellated into individual triangles.

* Setting the value of GEOMETRY_VERTICES_OUT_EXT will take effect immediately. It is not necessary to link the program object in order for this change to take effect, as is the case in the EXT version of this extension.

New Procedures and Functions

None

New Tokens

None

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.16.1, Geometry shader Input Primitives, of the EXT_geometry_shader4 specification as follows:

Triangles (TRIANGLES)

Geometry shaders that operate on triangles are valid for the TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, and POLYGON primitive types.

When used with a geometry shader that operates on triangles, QUADS, QUAD_STRIP, and POLYGON primitives are decomposed into triangles in an unspecified, implementation-dependent manner. This decomposition satisfies three properties:

1. the collection of triangles fully covers the area of the original primitive,
2. no two triangles in the decomposition overlap, and
3. the orientation of each triangle is consistent with the orientation of the original primitive.

For such primitives, the shader is executed once for each triangle in the decomposition.

There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively. ...

Modify Section 2.16.4, Geometry Shader Execution Environment, of the EXT_geometry_shader4 specification as follows:

Geometry shader inputs

(modify the spec language for primitive ID, describing its interaction with QUADS, QUAD_STRIP, and POLYGON topologies) The built-in special variable gl_PrimitiveIDIn is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is undefined. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Geometry Shader outputs

(modify the vertex output limit language to allow changes to take effect immediately) A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit needs to be set as a parameter of the program object that contains the geometry shader. To do so, call ProgramParameteriEXT with

<pname> set to GEOMETRY_VERTICES_OUT_EXT and <value> set to the maximum number of vertices the geometry shader will emit in one invocation. Setting this limit will take effect immediately. If a geometry shader, in one invocation, emits more vertices than the value GEOMETRY_VERTICES_OUT_EXT, these emits may have no effect.

(modify the error checking language for values that are too large) There are two implementation-dependent limits on the value of GEOMETRY_VERTICES_OUT_EXT. First, the error INVALID_VALUE will be generated by ProgramParameteriEXT if the number of vertices specified exceeds the value of MAX_GEOMETRY_OUTPUT_VERTICES_EXT. Second, the product of the total number of vertices and the sum of all components of all active varying variables may not exceed the value of MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT. If <program> has already been successfully linked, the error INVALID_VALUE will be generated by ProgramParameteriEXT if the specified value causes this limit to be exceeded. Additionally, LinkProgram will fail if it determines that the total component limit would be violated.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None

Additions to the AGL/GLX/WGL Specifications

None

Interactions with NV_transform_feedback

If GL_NV_transform_feedback is not supported, the function GetActiveVaryingNV() needs to be added to this extension. This function can be used to count the number of varying components output by a geometry shader, and from that data the maximum value for GEOMETRY_VERTICES_OUT_EXT computed by the application.

GLX protocol

None required

Errors

The error INVALID_OPERATION is generated if Begin, or any command that implicitly calls Begin, is called when a geometry shader is active and:

* the input primitive type of the current geometry shader is POINTS and <mode> is not POINTS,

* the input primitive type of the current geometry shader is LINES and <mode> is not LINES, LINE_STRIP, or LINE_LOOP,

* the input primitive type of the current geometry shader is TRIANGLES and <mode> is not TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, or POLYGON,

* the input primitive type of the current geometry shader is LINES_ADJACENCY_EXT and <mode> is not LINES_ADJACENCY_EXT or LINE_STRIP_ADJACENCY_EXT, or

* the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY_EXT and <mode> is not TRIANGLES_ADJACENCY_EXT or TRIANGLE_STRIP_ADJACENCY_EXT.

* GEOMETRY_VERTICES_OUT_EXT is zero for the currently active program object.

New State

None

Issues

1. *Why is there a GL_NV_geometry_shader4 and a GL_EXT_geometry_shader4 extension?*

   RESOLVED: NVIDIA initially wrote the geometry shader extension, and worked with other vendors on a common extension. Most of the functionality of the original specification was retained, but a few functional changes were made, resulting in the GL_EXT_geometry_shader4 specification.

   Some of the functionality removed in this process may be useful to developers, so we chose to provide an NVIDIA extension to expose this extra functionality.

2. *Should it be possible to change the limit on the number of vertices emitted by a geometry shader after the program object, containing the shader, is linked?*

   RESOLVED: Yes. Applications may want to tweak a piece of data that affects the number of vertices emitted, but wouldn't otherwise require re-linking the entire program object. One simple example might be a "circular point sprite" shader, that reads a single point, and draws a circle centered at that point with <N> vertices, where <N> is provided as a uniform. An application could change the value <N> at run time, which would require a change in the vertex limit. Another example might be a geometry shader that does some fancy subdivision, where the

relevant parameter might be a limit on how far the primitive is subdivided. This limit can be changed using the function ProgramParameteriEXT with <pname> set to GEOMETRY_VERTICES_OUT_EXT.

3. *How are QUADS, QUAD_STRIP, and POLYGON primitives decomposed into triangles in the initial implementation?*

   RESOLVED: The specification leaves the decomposition undefined, subject to a small number of rules. Assume that four vertices are specified in the order V0, V1, V2, V3.

   For QUADS primitives, the quad V0->V1->V2->V3 is decomposed into the triangles V0->V1->V2, and V0->V2->V3. The provoking vertex of the quad (V3) is only found in the second triangle. If it's necessary to flat shade over an entire quad, take the attributes from V0, which will be the first vertex for both triangles in the decomposition.

   For QUAD_STRIP primitives, the quad V0->V1->V3->V2 is decomposed into the triangles V0->V1->V3 and V2->V0->V3. This has the property of leaving the provoking vertex for the polygon (V3) as the third vertex for each triangle of the decomposition.

   For POLYGON primitives, the polygon V0->V1->V2->V3 is decomposed into the triangles V1->V2->V0 and V2->V3->V0. This has the property of leaving the provoking vertex for the polygon (V0) as the third vertex for each triangle of the decomposition.

   The triangulation described here is not guaranteed to be used on all implementations of this extension, and subsequent implementations may use a more natural decomposition for QUAD_STRIP and POLYGON primitives. (For example, the triangulation of 4-vertex polygons might match that used for QUADS.)

Revision History

None

Name

NV_gpu_program4

Name Strings

GL_NV_gpu_program4

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 02/04/2008
NVIDIA Revision: 4

Number

322

Dependencies

This extension is written against to OpenGL 2.0 specification.

OpenGL 2.0 is not required, but we expect all implementations of this extension will also support OpenGL 2.0.

This extension is also written against the ARB_vertex_program specification, which provides the basic mechanisms for the assembly programming model used by this extension.

This extension serves as the basis for the NV_fragment_program4, NV_geometry_program4, and NV_vertex_program4, which all build on this extension to support fragment, geometry, and vertex programs, respectively. If "GL_NV_gpu_program4" is found in the extension string, all of these extensions are supported.

NV_parameter_buffer_object affects the definition of this extension.

ARB_texture_rectangle trivially affects the definition of this extension.

EXT_gpu_program_parameters trivially affects the definition of this extension.

EXT_texture_integer trivially affects the definition of this extension.

EXT_texture_array trivially affects the definition of this extension.

EXT_texture_buffer_object trivially affects the definition of this extension.

NV_primitive_restart trivially affects the definition of this extension.

Overview

This specification documents the common instruction set and basic functionality provided by NVIDIA's 4th generation of assembly instruction sets supporting programmable graphics pipeline stages.

The instruction set builds upon the basic framework provided by the ARB_vertex_program and ARB_fragment_program extensions to expose considerably more capable hardware. In addition to new capabilities for vertex and fragment programs, this extension provides a new program type (geometry programs) further described in the NV_geometry_program4 specification.

NV_gpu_program4 provides a unified instruction set -- all instruction set features are available for all program types, except for a small number of features that make sense only for a specific program type. It provides fully capable signed and unsigned integer data types, along with a set of arithmetic, logical, and data type conversion instructions capable of operating on integers. It also provides a uniform set of structured branching constructs (if tests, loops, and subroutines) that fully support run-time condition testing.

This extension provides several new texture mapping capabilities. Shadow cube maps are supported, where cube map faces can encode depth values. Texture lookup instructions can include an immediate texel offset, which can assist in advanced filtering. New instructions are provided to fetch a single texel by address in a texture map (TXF) and query the size of a specified texture level (TXQ).

By and large, vertex and fragment programs written to ARB_vertex_program and ARB_fragment_program can be ported directly by simply changing the program header from "!!ARBvp1.0" or "!!ARBfp1.0" to "!!NVvp4.0" or "!!NVfp4.0", and then modifying the code to take advantage of the expanded feature set. There are a small number of areas where this extension is not a functional superset of previous vertex program extensions, which are documented in this specification.

New Procedures and Functions

```
void ProgramLocalParameterI4iNV(enum target, uint index,
                                int x, int y, int z, int w);
void ProgramLocalParameterI4ivNV(enum target, uint index,
                                 const int *params);
void ProgramLocalParametersI4ivNV(enum target, uint index,
                                  sizei count, const int *params);
void ProgramLocalParameterI4uiNV(enum target, uint index,
                                 uint x, uint y, uint z, uint w);
void ProgramLocalParameterI4uivNV(enum target, uint index,
                                  const uint *params);
void ProgramLocalParametersI4uivNV(enum target, uint index,
                                   sizei count, const uint *params);
```

```
    void ProgramEnvParameterI4iNV(enum target, uint index,
                                  int x, int y, int z, int w);
    void ProgramEnvParameterI4ivNV(enum target, uint index,
                                   const int *params);
    void ProgramEnvParametersI4ivNV(enum target, uint index,
                                    sizei count, const int *params);
    void ProgramEnvParameterI4uiNV(enum target, uint index,
                                   uint x, uint y, uint z, uint w);
    void ProgramEnvParameterI4uivNV(enum target, uint index,
                                    const uint *params);
    void ProgramEnvParametersI4uivNV(enum target, uint index,
                                     sizei count, const uint *params);

    void GetProgramLocalParameterIivNV(enum target, uint index,
                                       int *params);
    void GetProgramLocalParameterIuivNV(enum target, uint index,
                                        uint *params);
    void GetProgramEnvParameterIivNV(enum target, uint index,
                                     int *params);
    void GetProgramEnvParameterIuivNV(enum target, uint index,
                                      uint *params);
```

New Tokens

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

MIN_PROGRAM_TEXEL_OFFSET_EXT	0x8904
MAX_PROGRAM_TEXEL_OFFSET_EXT	0x8905

(note: these tokens are shared with the EXT_gpu_shader4 extension.)

Accepted by the <pname> parameter of GetProgramivARB:

PROGRAM_ATTRIB_COMPONENTS_NV	0x8906
PROGRAM_RESULT_COMPONENTS_NV	0x8907
MAX_PROGRAM_ATTRIB_COMPONENTS_NV	0x8908
MAX_PROGRAM_RESULT_COMPONENTS_NV	0x8909
MAX_PROGRAM_GENERIC_ATTRIBS_NV	0x8DA5
MAX_PROGRAM_GENERIC_RESULTS_NV	0x8DA6

Additions to Chapter 2 of the OpenGL 1.5 Specification (OpenGL Operation)

(Modify "Section 2.14.1" of the ARB_vertex_program specification, describing program parameters.)

Each program object has an associated array of program local parameters. Program local parameters are four-component vectors whose components can hold floating-point, signed integer, or unsigned integer values. The data type of each local parameter is established when the parameter's values are assigned. If a program attempts to read a local parameter using a data type other than the one used when the parameter is set, the values returned are undefined. ... The commands

```
  void ProgramLocalParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
  void ProgramLocalParameter4fvARB(enum target, uint index,
                                   const float *params);
  void ProgramLocalParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
  void ProgramLocalParameter4dvARB(enum target, uint index,
                                   const double *params);

  void ProgramLocalParameterI4iNV(enum target, uint index,
                                  int x, int y, int z, int w);
  void ProgramLocalParameterI4ivNV(enum target, uint index,
                                   const int *params);
  void ProgramLocalParameterI4uiNV(enum target, uint index,
                                   uint x, uint y, uint z, uint w);
  void ProgramLocalParameterI4uivNV(enum target, uint index,
                                    const uint *params);
```

update the values of the program local parameter numbered <index> belonging to the program object currently bound to <target>. For the non-vector versions of these commands, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For the vector versions, the components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program local parameters supported by <target>.

The commands

```
  void ProgramLocalParameters4fvNV(enum target, uint index,
                                   sizei count, const float *params);
  void ProgramLocalParametersI4ivNV(enum target, uint index,
                                    sizei count, const int *params);
  void ProgramLocalParametersI4uivNV(enum target, uint index,
                                     sizei count, const uint *params);
```

update the values of the program local parameters numbered <index> through <index> + <count> - 1 with the array of 4 * <count> values pointed to by <params>. The error INVALID_VALUE is generated if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

When a program local parameter is updated, the data type of its components is assigned according to the data type of the provided values. If values provided are of type "float" or "double", the components of the parameter are floating-point. If the values provided are of type "int", the components of the parameter are signed integers. If the values provided are of type "uint", the components of the parameter are unsigned integers.

Additionally, each program target has an associated array of program environment parameters. Unlike program local parameters, program environment parameters are shared by all program objects of a given target. Program environment parameters are four-component vectors whose components can hold floating-point, signed integer, or unsigned integer values. The data type of each environment parameter is established when the parameter's values are assigned. If a program attempts to read an environment parameter using a data type other than the one used when the

parameter is set, the values returned are undefined. ... The commands

```
  void ProgramEnvParameter4fARB(enum target, uint index,
                                float x, float y, float z, float w);
  void ProgramEnvParameter4fvARB(enum target, uint index,
                                 const float *params);
  void ProgramEnvParameter4dARB(enum target, uint index,
                                double x, double y, double z, double w);
  void ProgramEnvParameter4dvARB(enum target, uint index,
                                 const double *params);
  void ProgramEnvParameterI4iNV(enum target, uint index,
                                int x, int y, int z, int w);
  void ProgramEnvParameterI4ivNV(enum target, uint index,
                                 const int *params);
  void ProgramEnvParameterI4uiNV(enum target, uint index,
                                 uint x, uint y, uint z, uint w);
  void ProgramEnvParameterI4uivNV(enum target, uint index,
                                  const uint *params);
```

update the values of the program environment parameter numbered <index> for the given program target <target>. For the non-vector versions of these commands, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For the vector versions, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program environment parameters supported by <target>.

The commands

```
  void ProgramEnvParameters4fvNV(enum target, uint index,
                                 sizei count, const float *params);
  void ProgramEnvParametersI4ivNV(enum target, uint index,
                                  sizei count, const int *params);
  void ProgramEnvParametersI4uivNV(enum target, uint index,
                                   sizei count, const uint *params);
```

update the values of the program environment parameters numbered <index> through <index> + <count> - 1 with the array of 4 * <count> values pointed to by <params>. The error INVALID_VALUE is generated if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

When a program environment parameter is updated, the data type of its components is assigned according to the data type of the provided values. If values provided are of type "float" or "double", the components of the parameter are floating-point. If the values provided are of type "int", the components of the parameter are signed integers. If the values provided are of type "uint", the components of the parameter are unsigned integers.

Insert New Section 2.X between Sections 2.Y and 2.Z:

Section 2.X, GPU Programs

The GL provides a number of different program targets that allow an application to either replace certain fixed-function pipeline stages with

a fully programmable model or use a program to control aspects of the GL pipeline that previously had only hard-wired behavior.

A common base instruction set is available for all program types, providing both integer and floating-point operations. Structured branching operations and subroutine calls are available. Texture mapping (loading data from external images) is supported for all program types. The main differences between the different program types are the set of available inputs and outputs, which are program type-specific, and a few instructions that are meaningful for only a subset of program types.

Section 2.X.2, Program Grammar

GPU program strings are specified as an array of ASCII characters containing the program text. When a GPU program is loaded by a call to ProgramStringARB, the program string is parsed into a set of tokens possibly separated by whitespace. 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 GPU programs. The set of valid tokens can be inferred from the grammar. A line containing "/* empty */" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any tokens or characters not defined in this specification.

Note that this extension is not a standalone extension and a small number of grammar rules are left to be defined in the extensions defining the specific vertex, fragment, and geometry program types.

```
<program>               ::= <optionSequence> <declSequence>
                            <statementSequence> "END"

<optionSequence>        ::= <option> <optionSequence>
                          | /* empty */

<option>                ::= "OPTION" <identifier> ";"

<declSequence>          ::= /* empty */

<statementSequence>     ::= <statement> <statementSequence>
                          | /* empty */

<statement>             ::= <instruction> ";"
                          | <namingStatement> ";"
                          | <instLabel> ":"

<instruction>           ::= <ALUInstruction>
                          | <TexInstruction>
                          | <FlowInstruction>
```

```
<ALUInstruction>         ::= <VECTORop_instruction>
                           | <SCALARop_instruction>
                           | <BINSCop_instruction>
                           | <BINop_instruction>
                           | <VECSCAop_instruction>
                           | <TRIop_instruction>
                           | <SWZop_instruction>

<TexInstruction>         ::= <TEXop_instruction>
                           | <TXDop_instruction>

<FlowInstruction>        ::= <BRAop_instruction>
                           | <FLOWCCop_instruction>
                           | <IFop_instruction>
                           | <REPop_instruction>
                           | <ENDFLOWop_instruction>

<VECTORop_instruction>   ::= <VECTORop> <opModifiers> <instResult> ","
                             <instOperandV>

<VECTORop>               ::= "ABS"
                           | "CEIL"
                           | "FLR"
                           | "FRC"
                           | "I2F"
                           | "LIT"
                           | "MOV"
                           | "NOT"
                           | "NRM"
                           | "PK2H"
                           | "PK2US"
                           | "PK4B"
                           | "PK4UB"
                           | "ROUND"
                           | "SSG"
                           | "TRUNC"

<SCALARop_instruction>   ::= <SCALARop> <opModifiers> <instResult> ","
                             <instOperandS>

<SCALARop>               ::= "COS"
                           | "EX2"
                           | "LG2"
                           | "RCC"
                           | "RCP"
                           | "RSQ"
                           | "SCS"
                           | "SIN"
                           | "UP2H"
                           | "UP2US"
                           | "UP4B"
                           | "UP4UB"

<BINSCop_instruction>    ::= <BINSCop> <opModifiers> <instResult> ","
                             <instOperandS> "," <instOperandS>

<BINSCop>                ::= "POW"
```

```
<VECSCAop_instruction>  ::= <VECSCAop> <opModifiers> <instResult> ","
                            <instOperandV> "," <instOperandS>

<VECSCAop>              ::= "DIV"
                          | "SHL"
                          | "SHR"
                          | "MOD"

<BINop_instruction>     ::= <BINop> <opModifiers> <instResult> ","
                            <instOperandV> "," <instOperandV>

<BINop>                 ::= "ADD"
                          | "AND"
                          | "DP3"
                          | "DP4"
                          | "DPH"
                          | "DST"
                          | "MAX"
                          | "MIN"
                          | "MUL"
                          | "OR"
                          | "RFL"
                          | "SEQ"
                          | "SFL"
                          | "SGE"
                          | "SGT"
                          | "SLE"
                          | "SLT"
                          | "SNE"
                          | "STR"
                          | "SUB"
                          | "XPD"
                          | "DP2"
                          | "XOR"

<TRIop_instruction>     ::= <TRIop> <opModifiers> <instResult> ","
                            <instOperandV> "," <instOperandV> ","
                            <instOperandV>

<TRIop>                 ::= "CMP"
                          | "DP2A"
                          | "LRP"
                          | "MAD"
                          | "SAD"
                          | "X2D"

<SWZop_instruction>     ::= <SWZop> <opModifiers> <instResult> ","
                            <instOperandVNS> "," <extendedSwizzle>

<SWZop>                 ::= "SWZ"

<TEXop_instruction>     ::= <TEXop> <opModifiers> <instResult> ","
                            <instOperandV> "," <texAccess>
```

```
<TEXop>                 ::= "TEX"
                          | "TXB"
                          | "TXF"
                          | "TXL"
                          | "TXP"
                          | "TXQ"

<TXDop_instruction>     ::= <TXDop> <opModifiers> <instResult> ","
                            <instOperandV> "," <instOperandV> ","
                            <instOperandV> "," <texAccess>

<TXDop>                 ::= "TXD"

<BRAop_instruction>     ::= <BRAop> <opModifiers> <instTarget>
                            <optBranchCond>

<BRAop>                 ::= "CAL"

<FLOWCCop_instruction>  ::= <FLOWCCop> <opModifiers> <optBranchCond>

<FLOWCCop>              ::= "RET"
                          | "BRK"
                          | "CONT"

<IFop_instruction>      ::= <IFop> <opModifiers> <ccTest>

<IFop>                  ::= "IF"

<REPop_instruction>     ::= <REPop> <opModifiers> <instOperandV>
                          | <REPop> <opModifiers>

<REPop>                 ::= "REP"

<ENDFLOWop_instruction> ::= <ENDFLOWop> <opModifiers>

<ENDFLOWop>             ::= "ELSE"
                          | "ENDIF"
                          | "ENDREP"

<opModifiers>           ::= <opModifierItem> <opModifiers>
                          | /* empty */

<opModifierItem>        ::= "." <opModifier>

<opModifier>            ::= "F"
                          | "U"
                          | "S"
                          | "CC"
                          | "CC0"
                          | "CC1"
                          | "SAT"
                          | "SSAT"
                          | "NTC"
                          | "S24"
                          | "U24"
                          | "HI"
```

```
    <texAccess>             ::= <texImageUnit> "," <texTarget>
                              | <texImageUnit> "," <texTarget> "," <texOffset>

    <texImageUnit>          ::= "texture" <optArrayMemAbs>

    <texTarget>             ::= "1D"
                              | "2D"
                              | "3D"
                              | "CUBE"
                              | "RECT"
                              | "SHADOW1D"
                              | "SHADOW2D"
                              | "SHADOWRECT"
                              | "ARRAY1D"
                              | "ARRAY2D"
                              | "SHADOWCUBE"
                              | "SHADOWARRAY1D"
                              | "SHADOWARRAY2D"

    <texOffset>             ::= "(" <texOffsetComp> ")"
                              | "(" <texOffsetComp> "," <texOffsetComp> ")"
                              | "(" <texOffsetComp> "," <texOffsetComp> ","
                                <texOffsetComp> ")"

    <texOffsetComp>         ::= <optSign> <int>

    <optBranchCond>         ::= /* empty */
                              | <ccMask>

    <instOperandV>          ::= <instOperandAbsV>
                              | <instOperandBaseV>

    <instOperandAbsV>       ::= <operandAbsNeg> "|" <instOperandBaseV> "|"

    <instOperandBaseV>      ::= <operandNeg> <attribUseV>
                              | <operandNeg> <tempUseV>
                              | <operandNeg> <paramUseV>
                              | <operandNeg> <bufferUseV>

    <instOperandS>          ::= <instOperandAbsS>
                              | <instOperandBaseS>

    <instOperandAbsS>       ::= <operandAbsNeg> "|" <instOperandBaseS> "|"

    <instOperandBaseS>      ::= <operandNeg> <attribUseS>
                              | <operandNeg> <tempUseS>
                              | <operandNeg> <paramUseS>
                              | <operandNeg> <bufferUseS>

    <instOperandVNS>        ::= <attribUseVNS>
                              | <tempUseVNS>
                              | <paramUseVNS>
                              | <bufferUseVNS>

    <operandAbsNeg>         ::= <optSign>

    <operandNeg>            ::= <optSign>
```

```
<instResult>            ::= <instResultCC>
                          | <instResultBase>

<instResultCC>          ::= <instResultBase> <ccMask>

<instResultBase>        ::= <tempUseW>
                          | <resultUseW>

<namingStatement>       ::= <varMods> <ATTRIB_statement>
                          | <varMods> <PARAM_statement>
                          | <varMods> <TEMP_statement>
                          | <varMods> <OUTPUT_statement>
                          | <varMods> <BUFFER_statement>
                          | <ALIAS_statement>

<ATTRIB_statement>      ::= "ATTRIB" <establishName> "=" <attribUseD>

<PARAM_statement>       ::= <PARAM_singleStmt>
                          | <PARAM_multipleStmt>

<PARAM_singleStmt>      ::= "PARAM" <establishName> <paramSingleInit>

<PARAM_multipleStmt>    ::= "PARAM" <establishName> <optArraySize>
                            <paramMultipleInit>

<paramSingleInit>       ::= "=" <paramUseDB>

<paramMultipleInit>     ::= "=" "{" <paramMultInitList> "}"

<paramMultInitList>     ::= <paramUseDM>
                          | <paramUseDM> "," <paramMultInitList>

<TEMP_statement>        ::= "TEMP" <varNameList>

<OUTPUT_statement>      ::= "OUTPUT" <establishName> "=" <resultUseD>

<varMods>               ::= <varModifier> <varMods>
                          | /* empty */

<varModifier>           ::= "SHORT"
                          | "LONG"
                          | "INT"
                          | "UINT"
                          | "FLOAT"

<ALIAS_statement>       ::= "ALIAS" <establishName> "=" <establishedName>

<BUFFER_statement>      ::= <bufferDeclType> <establishName> "="
                            <bufferSingleInit>
                          | <bufferDeclType> <establishName>
                            <optArraySize> "=" <bufferMultInit>

<bufferDeclType>        ::= "BUFFER"
                          | "BUFFER4"

<bufferSingleInit>      ::= "=" <bufferUseDB>
```

```
<bufferMultInit>        ::= "=" "{" <bufferMultInitList> "}"

<bufferMultInitList>    ::= <bufferUseDM>
                          | <bufferUseDM> "," <bufferMultInitList>

<varNameList>           ::= <establishName>
                          | <establishName> "," <varNameList>

<attribUseV>            ::= <attribBasic> <swizzleSuffix>
                          | <attribVarName> <swizzleSuffix>
                          | <attribVarName> <arrayMem> <swizzleSuffix>
                          | <attribColor> <swizzleSuffix>
                          | <attribColor> "." <colorType> <swizzleSuffix>

<attribUseS>            ::= <attribBasic> <scalarSuffix>
                          | <attribVarName> <scalarSuffix>
                          | <attribVarName> <arrayMem> <scalarSuffix>
                          | <attribColor> <scalarSuffix>
                          | <attribColor> "." <colorType> <scalarSuffix>

<attribUseVNS>          ::= <attribBasic>
                          | <attribVarName>
                          | <attribVarName> <arrayMem>
                          | <attribColor>
                          | <attribColor> "." <colorType>

<attribUseD>            ::= <attribBasic>
                          | <attribColor>
                          | <attribColor> "." <colorType>
                          | <attribMulti>

<paramUseV>             ::= <paramVarName> <optArrayMem> <swizzleSuffix>
                          | <stateSingleItem> <swizzleSuffix>
                          | <programSingleItem> <swizzleSuffix>
                          | <constantVector> <swizzleSuffix>
                          | <constantScalar>

<paramUseS>             ::= <paramVarName> <optArrayMem> <scalarSuffix>
                          | <stateSingleItem> <scalarSuffix>
                          | <programSingleItem> <scalarSuffix>
                          | <constantVector> <scalarSuffix>
                          | <constantScalar>

<paramUseVNS>           ::= <paramVarName> <optArrayMem>
                          | <stateSingleItem>
                          | <programSingleItem>
                          | <constantVector>
                          | <constantScalar>

<paramUseDB>            ::= <stateSingleItem>
                          | <programSingleItem>
                          | <constantVector>
                          | <signedConstantScalar>
```

```
<paramUseDM>            ::= <stateMultipleItem>
                          | <programMultipleItem>
                          | <constantVector>
                          | <signedConstantScalar>

<stateMultipleItem>     ::= <stateSingleItem>
                          | "state" "." <stateMatrixRows>

<stateSingleItem>       ::= "state" "." <stateMaterialItem>
                          | "state" "." <stateLightItem>
                          | "state" "." <stateLightModelItem>
                          | "state" "." <stateLightProdItem>
                          | "state" "." <stateFogItem>
                          | "state" "." <stateMatrixRow>
                          | "state" "." <stateTexGenItem>
                          | "state" "." <stateClipPlaneItem>
                          | "state" "." <statePointItem>
                          | "state" "." <stateTexEnvItem>
                          | "state" "." <stateDepthItem>

<stateMaterialItem>     ::= "material" "." <stateMatProperty>
                          | "material" "." <faceType> "."
                            <stateMatProperty>

<stateMatProperty>      ::= "ambient"
                          | "diffuse"
                          | "specular"
                          | "emission"
                          | "shininess"

<stateLightItem>        ::= "light" <arrayMemAbs> "." <stateLightProperty>

<stateLightProperty>    ::= "ambient"
                          | "diffuse"
                          | "specular"
                          | "position"
                          | "attenuation"
                          | "spot" "." <stateSpotProperty>
                          | "half"

<stateSpotProperty>     ::= "direction"

<stateLightModelItem>   ::= "lightmodel" "." <stateLModProperty>

<stateLModProperty>     ::= "ambient"
                          | "scenecolor"
                          | <faceType> "." "scenecolor"

<stateLightProdItem>    ::= "lightprod" <arrayMemAbs> "."
                            <stateLProdProperty>
                          | "lightprod" <arrayMemAbs> "." <faceType> "."
                            <stateLProdProperty>

<stateLProdProperty>    ::= "ambient"
                          | "diffuse"
                          | "specular"
```

```
<stateFogItem>          ::= "fog" "." <stateFogProperty>

<stateFogProperty>      ::= "color"
                          | "params"

<stateMatrixRows>       ::= <stateMatrixItem>
                          | <stateMatrixItem> "." <stateMatModifier>
                          | <stateMatrixItem> "." "row" <arrayRange>
                          | <stateMatrixItem> "." <stateMatModifier> "."
                            "row" <arrayRange>

<stateMatrixRow>        ::= <stateMatrixItem> "." "row" <arrayMemAbs>
                          | <stateMatrixItem> "." <stateMatModifier> "."
                            "row" <arrayMemAbs>

<stateMatrixItem>       ::= "matrix" "." <stateMatrixName>

<stateMatModifier>      ::= "inverse"
                          | "transpose"
                          | "invtrans"

<stateMatrixName>       ::= "modelview" <optArrayMemAbs>
                          | "projection"
                          | "mvp"
                          | "texture" <optArrayMemAbs>
                          | "program" <arrayMemAbs>

<stateTexGenItem>       ::= "texgen" <optArrayMemAbs> "."
                            <stateTexGenType> "." <stateTexGenCoord>

<stateTexGenType>       ::= "eye"
                          | "object"

<stateTexGenCoord>      ::= "s"
                          | "t"
                          | "r"
                          | "q"

<stateClipPlaneItem>    ::= "clip" <arrayMemAbs> "." "plane"

<statePointItem>        ::= "point" "." <statePointProperty>

<statePointProperty>    ::= "size"
                          | "attenuation"

<stateTexEnvItem>       ::= "texenv" <optArrayMemAbs> "."
                            <stateTexEnvProperty>

<stateTexEnvProperty>   ::= "color"

<stateDepthItem>        ::= "depth" "." <stateDepthProperty>

<stateDepthProperty>    ::= "range"

<programSingleItem>     ::= <progEnvParam>
                          | <progLocalParam>
```

```
<programMultipleItem>   ::= <progEnvParams>
                          | <progLocalParams>

<progEnvParams>         ::= "program" "." "env" <arrayMemAbs>
                          | "program" "." "env" <arrayRange>

<progEnvParam>          ::= "program" "." "env" <arrayMemAbs>

<progLocalParams>       ::= "program" "." "local" <arrayMemAbs>
                          | "program" "." "local" <arrayRange>

<progLocalParam>        ::= "program" "." "local" <arrayMemAbs>

<constantVector>        ::= "{" <constantVectorList> "}"

<constantVectorList>    ::= <signedConstantScalar>
                          | <signedConstantScalar> ","
                            <signedConstantScalar>
                          | <signedConstantScalar> ","
                            <signedConstantScalar> ","
                            <signedConstantScalar>
                          | <signedConstantScalar> ","
                            <signedConstantScalar> ","
                            <signedConstantScalar> ","
                            <signedConstantScalar>

<signedConstantScalar>  ::= <optSign> <constantScalar>

<constantScalar>        ::= <floatConstant>
                          | <intConstant>

<floatConstant>         ::= <float>

<intConstant>           ::= <int>

<tempUseV>              ::= <tempVarName> <swizzleSuffix>

<tempUseS>              ::= <tempVarName> <scalarSuffix>

<tempUseVNS>            ::= <tempVarName>

<tempUseW>              ::= <tempVarName> <optWriteMask>

<resultUseW>            ::= <resultBasic> <optWriteMask>
                          | <resultVarName> <optWriteMask>

<resultUseD>            ::= <resultBasic>

<bufferUseV>            ::= <bufferVarName> <optArrayMem> <swizzleSuffix>

<bufferUseS>            ::= <bufferVarName> <optArrayMem> <scalarSuffix>

<bufferUseVNS>          ::= <bufferVarName> <optArrayMem>

<bufferUseDB>           ::= <bufferBinding> <arrayMemAbs>
```

```
    <bufferUseDM>           ::= <bufferBinding> <arrayMemAbs>
                              | <bufferBinding> <arrayRange>
                              | <bufferBinding>

    <bufferBinding>         ::= "program" "." "buffer" <arrayMemAbs>

    <optArraySize>          ::= "[" "]"
                              | "[" <int> "]"

    <optArrayMem>           ::= /* empty */
                              | <arrayMem>

    <arrayMem>              ::= <arrayMemAbs>
                              | <arrayMemRel>

    <optArrayMemAbs>        ::= /* empty */
                              | <arrayMemAbs>

    <arrayMemAbs>           ::= "[" <int> "]"

    <arrayMemRel>           ::= "[" <arrayMemReg> <arrayMemOffset> "]"

    <arrayMemReg>           ::= <addrUseS>

    <arrayMemOffset>        ::= /* empty */
                              | "+" <int>
                              | "-" <int>

    <arrayRange>            ::= "[" <int> ".." <int> "]"

    <addrUseS>              ::= <addrVarName> <scalarSuffix>

    <ccMask>                ::= "(" <ccTest> ")"

    <ccTest>                ::= <ccMaskRule> <swizzleSuffix>
```

```
<ccMaskRule>            ::= "EQ"
                          | "GE"
                          | "GT"
                          | "LE"
                          | "LT"
                          | "NE"
                          | "TR"
                          | "FL"
                          | "EQ0"
                          | "GE0"
                          | "GT0"
                          | "LE0"
                          | "LT0"
                          | "NE0"
                          | "TR0"
                          | "FL0"
                          | "EQ1"
                          | "GE1"
                          | "GT1"
                          | "LE1"
                          | "LT1"
                          | "NE1"
                          | "TR1"
                          | "FL1"
                          | "NAN"
                          | "NAN0"
                          | "NAN1"
                          | "LEG"
                          | "LEG0"
                          | "LEG1"
                          | "CF"
                          | "CF0"
                          | "CF1"
                          | "NCF"
                          | "NCF0"
                          | "NCF1"
                          | "OF"
                          | "OF0"
                          | "OF1"
                          | "NOF"
                          | "NOF0"
                          | "NOF1"
                          | "AB"
                          | "AB0"
                          | "AB1"
                          | "BLE"
                          | "BLE0"
                          | "BLE1"
                          | "SF"
                          | "SF0"
                          | "SF1"
                          | "NSF"
                          | "NSF0"
                          | "NSF1"
```

```
    <optWriteMask>          ::= /* empty */
                              | <xyzwMask>
                              | <rgbaMask>

    <xyzwMask>              ::= "." "x"
                              | "." "y"
                              | "." "xy"
                              | "." "z"
                              | "." "xz"
                              | "." "yz"
                              | "." "xyz"
                              | "." "w"
                              | "." "xw"
                              | "." "yw"
                              | "." "xyw"
                              | "." "zw"
                              | "." "xzw"
                              | "." "yzw"
                              | "." "xyzw"

    <rgbaMask>              ::= "." "r"
                              | "." "g"
                              | "." "rg"
                              | "." "b"
                              | "." "rb"
                              | "." "gb"
                              | "." "rgb"
                              | "." "a"
                              | "." "ra"
                              | "." "ga"
                              | "." "rga"
                              | "." "ba"
                              | "." "rba"
                              | "." "gba"
                              | "." "rgba"

    <swizzleSuffix>         ::= /* empty */
                              | "." <component>
                              | "." <xyzwSwizzle>
                              | "." <rgbaSwizzle>

    <extendedSwizzle>       ::= <extSwizComp> "," <extSwizComp> ","
                                <extSwizComp> "," <extSwizComp>

    <extSwizComp>           ::= <optSign> <xyzwExtSwizSel>
                              | <optSign> <rgbaExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                              | "1"
                              | <xyzwComponent>

    <rgbaExtSwizSel>        ::= <rgbaComponent>

    <scalarSuffix>          ::= "." <component>

    <component>             ::= <xyzwComponent>
                              | <rgbaComponent>
```

```
<xyzwComponent>         ::= "x"
                          | "y"
                          | "z"
                          | "w"

<rgbaComponent>         ::= "r"
                          | "g"
                          | "b"
                          | "a"

<optSign>               ::= /* empty */
                          | "-"
                          | "+"

<faceType>              ::= "front"
                          | "back"

<colorType>             ::= "primary"
                          | "secondary"

<instLabel>             ::= <identifier>

<instTarget>            ::= <identifier>

<establishedName>       ::= <identifier>

<establishName>         ::= <identifier>
```

The <int> rule matches an integer constant. The integer consists of a sequence of one or more digits ("0" through "9"), or a sequence in hexadecimal form beginning with "0x" followed by a sequence of one or more hexadecimal digits ("0" through "9", "a" through "f", "A" through "F").

The <float> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of one or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing. Most grammar rules that allow floating-point values also allow integers matching the <int> rule.

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z"), digits ("0" through "9), underscores ("_"), or dollar signs ("$"); the first character must not be a number. Upper and lower case letters are considered different (names are case-sensitive). The following strings are reserved keywords and may not be used as identifiers: "fragment" (for fragment programs only), "vertex" (for vertex and geometry programs), "primitive" (for fragment and geometry programs), "program", "result", "state", and "texture".

The <tempVarName>, <paramVarName>, <attribVarName>, <resultVarName>, and <bufferName> rules match identifiers that have been previously established as names of temporary, program parameter, attribute, result, and program parameter buffer variables, respectively.

The <xyzwSwizzle> and <rgbaSwizzle> rules match any 4-character strings consisting only of the characters "x", "y", "z", and "w" (<xyzwSwizzle>) or "r", "g", "b", "a" (<rgbaSwizzle>).

The error INVALID_OPERATION is generated if a program fails to load because it is not syntactically correct or for one of the semantic restrictions described in the following sections.

A successfully loaded 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.X.4. A successfully loaded program string replaces the program string previously loaded into the specified program object. If the OUT_OF_MEMORY error is generated by ProgramStringARB, no change is made to the previous contents of the current program object.

Section 2.X.3, Program Variables

Programs may operate on a number of different variables during their execution. The following sections define the different classes of variables that can be declared and used by a program.

Some variable classes require variable bindings. Variable classes with bindings refer to state that is either generated or consumed outside the program. Examples of variable bindings include a vertex's normal, the position of a vertex computed by a vertex program, an interpolated texture coordinate, and the diffuse color of light 1. Variables that are used only during program execution do not have bindings.

Variables may be declared explicitly according to the <namingStatement> grammar rule. Explicit variable declarations allow a program to establish a variable name that can be used to refer to a specified resource in subsequent instructions. Variables may be declared anywhere in the program string, but must be declared prior to use. A program will fail to load if it declares the same variable name more than once, or if it refers to a variable name that has not been previously declared in the program string.

Variables may also be declared implicitly, simply by using a variable binding as an operand in a program instruction. Such uses are considered to automatically create a nameless variable using the specified binding. Only variable from classes with bindings can be declared implicitly.

Section 2.X.3.1, Program Variable Types

Explicit variable declarations may include one or more modifiers that specify additional information about the variable, such as the size and data type of the components of the variable. Variable modifiers are specified according to the <varModifier> grammar rule.

By default, variables are considered typeless. They can be used in instructions that read or write the variable as floating-point values, signed integers, or unsigned integers. If a variable is written using one data type but then read using a different one, the results of the operation are undefined. Variables with bindings are considered to be read or written when their values are produced or consumed; the data type used by the GL is specified in the description of each binding.

Explicitly declared variables may optionally have one data type modifier, which can be used to detect data type mismatch errors. Type modifers of "INT", "UINT", and "FLOAT" indicate that the components of the variable are stored as signed integers, unsigned integers, or floating-point values, respectively. A program will fail to load if it attempts to read or write a variable using a data type other than the one indicated by the data type modifier. Variables without a data type modifier can be read or written using any data type.

Explicitly declared variables may optionally have one storage size modifier. Variables decared as "SHORT" will be represented using at least 16 bits per component. "SHORT" floating-point values will have at least 5 bits of exponent and 10 bits of mantissa. Variables declared as "LONG" will be represented with at least 32 bits per component. "LONG" floating-point values will have at least 8 bits of exponent and 23 bits of mantissa. If no size modifier is provided, the GL will automatically select component sizes. Implementations are not required to support more than one component size, so "SHORT", "LONG", and the default could all refer to the same component size.

Each variable declaration can include at most one data type and one storage size modifier. A program will fail to load if it specifies multiple data type or multiple storage size modifiers in a single variable declaration.

(NOTE: Fragment programs also support the modifiers "FLAT", "CENTROID", and "NOPERSPECTIVE", which control how per-fragment attribute values are produced. These modifiers are described in detail in the NV_fragment_program4 specification.)

Explicitly declared variables of all types may be declared as arrays. An array variable has one or more members, numbered 0 through <n>-1, where <n> is the number of entries in the array. The total number of entries in the array can be declared using the <optArraySize> grammar rule. For variable classes without bindings, an array size must be specified in the program, and must be a positive integer. For variable classes with bindings, a declared size is optional, and is taken from the number of bindings assigned in the declaration if omitted. A program will fail to load if the declared size of an array variable does not match the number of assigned bindings.

When a variable is declared as an array, instructions that use the variable must specify an array member to access according to the <arrayMem> grammar rule. A program will fail to load if it contains an instruction that accesses an array variable without specifying an array member or an instruction that specifies an array member for a non-array variable.

Section 2.X.3.2, Program Attribute Variables

Program attribute variables represent per-vertex or per-fragment inputs to the program. All attribute variables have associated bindings, and are read-only during program execution. Attribute variables may be declared explicitly via the <ATTRIB_statement> grammar rule, or implicitly by using an attribute binding in an instruction.

The set of available attribute bindings depends on the program type, and is enumerated in the specifications for each program type.

The set of bindings allowed for attribute array variables is limited to attribute state grouped in arrays (e.g., texture coordinates, generic vertex attributes). Additionally, all bindings assigned to the array must be of the same binding type and must increase consecutively. Examples of valid and invalid binding lists include:

```
  vertex.attrib[1], vertex.attrib[2]      # valid, 2-entry array
  vertex.texcoord[0..3]                   # valid, 4-entry array
  vertex.attrib[1], vertex.attrib[3]      # invalid, skipped attrib 2
  vertex.attrib[2], vertex.attrib[1]      # invalid, wrong order
  vertex.attrib[1], vertex.texcoord[2]    # invalid, different types
```

Additionally, attribute bindings may be used in no more than one array variable accessed with relative addressing.

Implementations may have a limit on the total number of attribute binding components used by each program target (MAX_PROGRAM_ATTRIB_COMPONENTS). Programs that use more attribute binding components than this limit will fail to load. The method of counting used attribute binding components is implementation-dependent, but must satisfy the following properties:

* If an attribute binding is not referenced in a program, or is referenced only in declarations of attribute variables that are not used, none of its components are counted.

* An attribute binding component may be counted as used only if there exists an instruction operand where

  - the component is enabled for read by the swizzle pattern (Section 2.X.4.2), and

  - the attribute binding is

    - referenced directly by the operand,

    - bound to a declared variable referenced by the operand, or

    - bound to a declared array variable where another binding in the array satisfies one of the two previous conditions.

  Implementations are not required to optimize out unused elements of an attribute array or components that are used in only some elements of an array. The last of these rules is intended to cover the case where the same attribute binding is used in multiple variables.

  For example, an operand whose swizzle pattern selects only the x component may result in the x component of an attribute binding being counted, but may never result in the counting of the y, z, or w components of any attribute binding.

* Implementations are not required to determine that components read by an instruction are actually unused due to:

  - instruction write masks (for example, a component-wise ADD operation that only writes the "x" component doesn't have to read the "y", "z", and "w" components of its operands) or

  - any other properties of the instruction (for example, the DP3 instruction computes a 3-component dot product doesn't have to read the "w" component of its operands).

Section 2.X.3.3, Program Parameters

Program parameter variables are used as constants during program execution. All program parameter variables have associated bindings and are read-only during program execution. Program parameters retain their values across program invocations, although their values may change between invocations due to GL state changes. Program parameter variables may be declared explicitly via the <PARAM_statement> grammar rule, or implicitly by using a parameter binding in an instruction. Except where otherwise specified, program parameter bindings always specify floating-point values.

When declaring program parameter array variables, all bindings are supported and can be assigned to array members in any order. The only restriction is that no parameter binding may be used more than once in array variables accessed using relative addressing. A program will fail to load if any program parameter binding is used more than once in a single array accessed using relative addressing or used at least once in two or more arrays accessed using relative addressing.

Constant Bindings

If a program parameter binding matches the <constantScalar> or <signedConstantScalar> grammar rules, the corresponding program parameter variable is bound to the vector (X,X,X,X), where X is the value of the specified constant.

If a program parameter binding matches <constantVector>, the corresponding program parameter variable is bound to the vector (X,Y,Z,W), where X, Y, Z, and W are the values corresponding to the first, second, third, and fourth match of <signedConstantScalar>. If fewer than four constants are specified, Y, Z, and W assume the values 0, 0, and 1, if their respective constants are not specified.

Constant bindings can be interpreted as having signed integer, unsigned integer, or floating-point values, depending on how they are used in the program text. For constants in variable declarations, the components of the constant are interpreted according to the variable's component data type modifier. If no data type modifier is specified in a declaration, constants are interpreted as floating-point values. For constant bindings used directly in an instruction, the components of the constant are interpreted according to the required data type of the operand. A program will fail to load if it specifies a floating-point constant value (matching the <floatConstant> grammar rule) that should be interpreted as a signed or unsigned integer, or a negative integer constant value that should be interpreted as an unsigned integer.

If the value used to specify a floating-point constant can not be exactly represented, the nearest floating-point value will be used. If the value used to specify an integer constant is too large to be represented, the program will fail to load.

Program Environment/Local Parameter Bindings

Binding	Components	Underlying State
program.env[a]	(x,y,z,w)	program environment parameter a
program.local[a]	(x,y,z,w)	program local parameter a
program.env[a..b]	(x,y,z,w)	program environment parameters a through b
program.local[a..b]	(x,y,z,w)	program local parameters a through b

Table X.1: Program Environment/Local Parameter Bindings. <a> and <b> indicate parameter numbers, where <a> must be less than or equal to <b>.

If a program parameter binding matches "program.env[a]" or "program.local[a]", the four components of the program parameter variable are filled with the four components of program environment parameter <a> or program local parameter <a> respectively.

Additionally, for program parameter array bindings, "program.env[a..b]" and "program.local[a..b]" are equivalent to specifying program environment or local parameters <a> through <b> in order, respectively. A program using any of these bindings will fail to load if <a> is greater than <b>.

Program environment and local parameters are typeless, and may be specified as signed integer, unsigned integer, or floating-point variables. If a program environment parameter is read using a data type other than the one used to specify it, an undefined value is returned.

Material Property Bindings

Binding	Components	Underlying State
state.material.ambient	(r,g,b,a)	front ambient material color
state.material.diffuse	(r,g,b,a)	front diffuse material color
state.material.specular	(r,g,b,a)	front specular material color
state.material.emission	(r,g,b,a)	front emissive material color
state.material.shininess	(s,0,0,1)	front material shininess
state.material.front.ambient	(r,g,b,a)	front ambient material color
state.material.front.diffuse	(r,g,b,a)	front diffuse material color
state.material.front.specular	(r,g,b,a)	front specular material color
state.material.front.emission	(r,g,b,a)	front emissive material color
state.material.front.shininess	(s,0,0,1)	front material shininess
state.material.back.ambient	(r,g,b,a)	back ambient material color
state.material.back.diffuse	(r,g,b,a)	back diffuse material color
state.material.back.specular	(r,g,b,a)	back specular material color
state.material.back.emission	(r,g,b,a)	back emissive material color
state.material.back.shininess	(s,0,0,1)	back material shininess

Table X.3: Material Property Bindings. If a material face is not specified in the binding, the front property is used.

If a program parameter binding matches any of the material properties listed in Table X.3, the program parameter variable is filled according to the table. For ambient, diffuse, specular, or emissive colors, the "x", "y", "z", and "w" components are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding material color. For material shininess, the "x" component is filled with the material's specular exponent, and the "y", "z", and "w" components are filled with the floating-point constants 0, 0, and 1, respectively. Bindings containing ".back" refer to the back material; all other bindings refer to the front material.

Material properties can be changed inside a Begin/End pair, either directly by calling Material, or indirectly through color material. However, such property changes are not guaranteed to update program parameter bindings until the following End command. Program parameter variables bound to material properties changed inside a Begin/End pair are undefined until the following End command.

Light Property Bindings

Binding	Components	Underlying State
state.light[n].ambient	(r,g,b,a)	light n ambient color
state.light[n].diffuse	(r,g,b,a)	light n diffuse color
state.light[n].specular	(r,g,b,a)	light n specular color
state.light[n].position	(x,y,z,w)	light n position
state.light[n].attenuation	(a,b,c,e)	light n attenuation constants and spot light exponent
state.light[n].spot.direction	(x,y,z,c)	light n spot direction and cutoff angle cosine
state.light[n].half	(x,y,z,1)	light n infinite half-angle
state.lightmodel.ambient	(r,g,b,a)	light model ambient color
state.lightmodel.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel.front.scenecolor	(r,g,b,a)	light model front scene color
state.lightmodel.back.scenecolor	(r,g,b,a)	light model back scene color
state.lightprod[n].ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n].diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n].specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n].front.ambient	(r,g,b,a)	light n / front material ambient color product
state.lightprod[n].front.diffuse	(r,g,b,a)	light n / front material diffuse color product
state.lightprod[n].front.specular	(r,g,b,a)	light n / front material specular color product
state.lightprod[n].back.ambient	(r,g,b,a)	light n / back material ambient color product
state.lightprod[n].back.diffuse	(r,g,b,a)	light n / back material diffuse color product
state.lightprod[n].back.specular	(r,g,b,a)	light n / back material specular color product

Table X.4: Light Property Bindings. <n> indicates a light number.

If a program parameter binding matches "state.light[n].ambient", "state.light[n].diffuse", or "state.light[n].specular", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding light color.

If a program parameter binding matches "state.light[n].position", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "x", "y", "z", and "w" components, respectively, of the light position.

If a program parameter binding matches "state.light[n].attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic attenuation parameters of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the spot light exponent of the specified light.

If a program parameter binding matches "state.light[n].spot.direction", the "x", "y", and "z" components of the program parameter variable are filled with the "x", "y", and "z" components of the spot light direction of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the cosine of the spot light cutoff angle of the specified light.

If a program parameter binding matches "state.light[n].half", the "x", "y", and "z" components of the program parameter variable are filled with the x, y, and z components, respectively, of the normalized infinite half-angle vector

  h_inf = || P + (0, 0, 1) ||.

The "w" component is filled with 1.0. In the computation of h_inf, P consists of the x, y, and z coordinates of the normalized vector from the eye position P_e to the eye-space light position P_pli (section 2.13.1). h_inf is defined to correspond to the normalized half-angle vector when using an infinite light (w coordinate of the position is zero) and an infinite viewer (v_bs is FALSE). For local lights or a local viewer, h_inf is well-defined but does not match the normalized half-angle vector, which will vary depending on the vertex position.

If a program parameter binding matches "state.lightmodel.ambient", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components of the light model ambient color, respectively.

If a program parameter binding matches "state.lightmodel.scenecolor" or "state.lightmodel.front.scenecolor", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components respectively of the "front scene color"

  c_scene = a_cs * a_cm + e_cm,

where a_cs is the light model ambient color, a_cm is the front ambient material color, and e_cm is the front emissive material color. The "w" component of the program parameter variable is filled with the alpha component of the front diffuse material color. If a program parameter binding matches "state.lightmodel.back.scenecolor", a similar back scene color, computed using back-facing material properties, is used. The front and back scene colors match the values that would be assigned to vertices using conventional lighting if all lights were disabled.

If a program parameter binding matches anything beginning with "state.lightprod[n]", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components, respectively, of the corresponding light product. The three light product components are the products of the corresponding color components of the specified material property and the light color of the specified light (see Table X.4). The "w" component of the program parameter variable is filled with the alpha component of the specified material property.

Light products depend on material properties, which can be changed inside a Begin/End pair. Such property changes are not guaranteed to take effect until the following End command. Program parameter variables bound to

light products whose corresponding material property changes inside a Begin/End pair are undefined until the following End command.

Texture Coordinate Generation Property Bindings

Binding	Components	Underlying State
state.texgen[n].eye.s	(a,b,c,d)	TexGen eye linear plane coefficients, s coord, unit n
state.texgen[n].eye.t	(a,b,c,d)	TexGen eye linear plane coefficients, t coord, unit n
state.texgen[n].eye.r	(a,b,c,d)	TexGen eye linear plane coefficients, r coord, unit n
state.texgen[n].eye.q	(a,b,c,d)	TexGen eye linear plane coefficients, q coord, unit n
state.texgen[n].object.s	(a,b,c,d)	TexGen object linear plane coefficients, s coord, unit n
state.texgen[n].object.t	(a,b,c,d)	TexGen object linear plane coefficients, t coord, unit n
state.texgen[n].object.r	(a,b,c,d)	TexGen object linear plane coefficients, r coord, unit n
state.texgen[n].object.q	(a,b,c,d)	TexGen object linear plane coefficients, q coord, unit n

Table X.5: Texture Coordinate Generation Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches a set of TexGen plane coefficients, the "x", "y", "z", and "w" components of the program parameter variable are filled with the coefficients p1, p2, p3, and p4, respectively, for object linear coefficients, and the coefficents p1', p2', p3', and p4', respectively, for eye linear coefficients (section 2.10.4).

Fog Property Bindings

Binding	Components	Underlying State
state.fog.color	(r,g,b,a)	RGB fog color (section 3.10)
state.fog.params	(d,s,e,r)	fog density, linear start and end, and 1/(end-start) (section 3.10)

Table X.6: Fog Property Bindings

If a program parameter binding matches "state.fog.color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the fog color (section 3.10).

If a program parameter binding matches "state.fog.params", the "x", "y", and "z" components of the program parameter variable are filled with the fog density, linear fog start, and linear fog end parameters (section 3.10), respectively. The "w" component is filled with 1/(end-start), where end and start are the linear fog end and start parameters, respectively.

Clip Plane Property Bindings

Binding	Components	Underlying State
state.clip[n].plane	(a,b,c,d)	clip plane n coefficients

Table X.7: Clip Plane Property Bindings. <n> specifies the clip plane number, and is required.

If a program parameter binding matches "state.clip[n].plane", the "x", "y", "z", and "w" components of the program parameter variable are filled with the coefficients p1', p2', p3', and p4', respectively, of clip plane <n> (section 2.11).

Point Property Bindings

Binding	Components	Underlying State
state.point.size	(s,n,x,f)	point size, min and max size clamps, and fade threshold (section 3.3)
state.point.attenuation	(a,b,c,1)	point size attenuation consts

Table X.8: Point Property Bindings

If a program parameter binding matches "state.point.size", the "x", "y", "z", and "w" components of the program parameter variable are filled with the point size, minimum point size, maximum point size, and fade threshold, respectively (section 3.3).

If a program parameter binding matches "state.point.attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic point size attenuation parameters (a, b, and c), respectively (section 3.3). The "w" component is filled with 1.0.

Texture Environment Property Bindings

Binding	Components	Underlying State
state.texenv[n].color	(r,g,b,a)	texture environment n color

Table X.9: Texture Environment Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches "state.texenv[n].color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding texture environment color. Note that only "legacy" texture units, as queried by MAX_TEXTURE_UNITS, include texture environment state. Texture image units and texture coordinate sets do not have associated texture environment state.

Depth Property Bindings

Binding	Components	Underlying State
state.depth.range	(n,f,d,1)	Depth range near, far, and (far-near) (section 2.10.1)

Table X.10: Depth Property Bindings

If a program parameter binding matches "state.depth.range", the "x" and "y" components of the program parameter variable are filled with the mappings of near and far clipping planes to window coordinates, respectively. The "z" component is filled with the difference of the mappings of near and far clipping planes, far minus near. The "w" component is filled with 1.0.

Matrix Property Bindings

Binding	Underlying State
* state.matrix.modelview[n]	modelview matrix n
state.matrix.projection	projection matrix
state.matrix.mvp	modelview-projection matrix
* state.matrix.texture[n]	texture matrix n
state.matrix.program[n]	program matrix n

Table X.11: Base Matrix Property Bindings. The "[n]" syntax indicates a specific matrix number. For modelview and texture matrices, a matrix number is optional, and matrix zero will be used if the matrix number is omitted. These base bindings may further be modified by a inverse/transpose selector and a row selector.

If the beginning of a program parameter binding matches any of the matrix binding names listed in Table X.11, the binding corresponds to a 4x4 matrix. If the parameter binding is followed by ".inverse", ".transpose", or ".invtrans" (<stateMatModifier> grammar rule), the inverse, transpose, or transpose of the inverse, respectively, of the matrix specified in Table X.11 is selected. Otherwise, the matrix specified in Table X.11 is selected. If the specified matrix is poorly-conditioned (singular or nearly so), its inverse matrix is undefined. The binding name "state.matrix.mvp" refers to the product of modelview matrix zero and the projection matrix, defined as

```
MVP = P * M0,
```

where P is the projection matrix and M0 is modelview matrix zero.

If the selected matrix is followed by ".row[<a>]" (matching the <stateMatrixRow> grammar rule), the "x", "y", "z", and "w" components of the program parameter variable are filled with the four entries of row <a> of the selected matrix. In the example,

```
PARAM m0 = state.matrix.modelview[1].row[0];
PARAM m1 = state.matrix.projection.transpose.row[3];
```

the variable "m0" is set to the first row (row 0) of modelview matrix 1 and "m1" is set to the last row (row 3) of the transpose of the projection

matrix.

For program parameter array bindings, multiple rows of the selected matrix can be bound via the <stateMatrixRows> grammar rule. If the selected matrix binding is followed by ".row[<a>..<b>]", the result is equivalent to specifying matrix rows <a> through <b>, in order. A program will fail to load if <a> is greater than <b>. If no row selection is specified (<optMatrixRows> matches ""), matrix rows 0 through 3 are bound in order. In the example,

```
  PARAM m2[] = { state.matrix.program[0].row[1..2] };
  PARAM m3[] = { state.matrix.program[0].transpose };
```

the array "m2" has two entries, containing rows 1 and 2 of program matrix zero, and "m3" has four entries, containing all four rows of the transpose of program matrix zero.

Section 2.X.3.4, Program Temporaries

Program temporary variables are used to hold temporary results during program execution. Temporaries do not persist between program invocations, and are undefined at the beginning of each program invocation.

Temporary variables are declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Temporaries can not be declared implicitly. Temporaries can be declared using any component size ("SHORT" or "LONG") and type ("FLOAT" or "INT") modifier.

Temporary variables may be declared as arrays. Temporary variables declared as arrays may be stored in slower memory than those not declared as arrays, and it is recommended to use non-array variables unless array functionality is required.

Section 2.X.3.5, Program Results

Program result variables represent the per-vertex or per-fragment results of the program. All result variables have associated bindings, are write-only during program execution, and are undefined at the beginning of each program invocation. Any vertex or fragment attributes corresponding to unwritten result variables will be undefined in subsequent stages of the pipeline. Result variables may be declared explicitly via the <OUTPUT_statement> grammar rule, or implicitly by using a result binding in an instruction.

The set of available result bindings depends on the program type, and is enumerated in the specifications for each program type.

Result variables may generally be declared as arrays, but the set of bindings allowed for arrays is limited to state grouped in arrays (e.g., texture coordinates, clip distances, colors). Additionally, all bindings assigned to the array must be of the same binding type and must increase consecutively. Examples of valid and invalid binding lists for vertex programs include:

```
  result.clip[1], result.clip[2]          # valid, 2-entry array
```

```
  result.texcoord[0..3]                   # valid, 4-entry array
  result.texcoord[1], result.texcoord[3]  # invalid, skipped texcoord 2
  result.texcoord[2], result.texcoord[1]  # invalid, wrong order
  result.texcoord[1], result.clip[2]      # invalid, different types
```

Additionally, result bindings may be used in no more than one array addressed with relative addressing.

Implementations may have a limit on the total number of result binding components used by each program target (MAX_PROGRAM_RESULT_COMPONENTS_NV). Programs that require more result binding components than this limit will fail to load. The method of counting used result binding components is implementation-dependent, but must satisfy the following properties:

* If a result binding is not referenced in a program, or is referenced only in declarations of result variables that are not used, none of its components are counted.

* A result binding component may be counted as used only if there exists an instruction operand where

  - the component is enabled in the write mask (Section 2.X.4.3), and

  - the result binding is either

    - referenced directly by the operand,

    - bound to a declared variable referenced by the operand, or

    - bound to a declared array variable where another binding in the array satisfies one of the two previous conditions.

  Implementations are not required to optimize out unused elements of an result array or components that are used in only some elements of an array. The last of these rules is intended to cover the case where the same result binding is used in multiple variables.

  For example, an instruction whose write mask selects only the x component may result in the x component of a result binding being counted, but may never result in the counting of the y, z, or w components of any result binding.

Section 2.X.3.6, Program Parameter Buffers

Program parameter buffers are arrays consisting of single-component typeless values or four-component typeless vectors stored in a buffer object. The GL provides an implementation-dependent number of buffer object binding points for each program target, to which buffer objects can be attached. Program parameter buffer variables can be changed either by updating the contents of bound buffer objects, or simply by changing the buffer object attached to a binding point.

Program parameter buffer variables are used as constants during program execution. All program parameter buffer variables have an associated binding and are read-only during program execution. Program parameter buffers retain their values across program invocations, although their values may change as buffer object bindings or contents change. Program

parameter buffer variables must be declared explicitly via the <BUFFER_statement> grammar rule. Program parameter buffer bindings can not be used directly in executable instructions.

Program parameter buffer variables are treated as an array of single-component values if the <bufferDeclType> grammar rule matches "BUFFER" or as an array of four-component vectors if it matches "BUFFER4". A program will fail to load if a variable declared as "BUFFER" and another variable declared as "BUFFER4" use the same buffer binding point.

Program parameter buffer variables may be declared as arrays, but all bindings assigned to the array must use the same binding point and must increase consecutively.

Binding	Components	Underlying State
program.buffer[a][b]	(x,x,x,x)	program parameter buffer a, element b
program.buffer[a][b..c]	(x,x,x,x)	program parameter buffer a, elements b through c
program.buffer[a]	(x,x,x,x)	program parameter buffer a, all elements

Table X.12: Program Parameter Buffer Bindings. <a> indicates a buffer number, <b> and <c> indicate individual elements.

If a program parameter buffer binding matches "program.buffer[a][b]", the program parameter variable are filled with element <b> of the buffer object bound to binding point <a>. Each element of the bound buffer object is treated a one or four words of data that can hold integer or floating-point values. When a single-component binding is evaluated, the selected word is broadcast to all four components of the variable. When a four-component binding is evaluated, the four components of the buffer element are loaded into the variable. If no buffer object is bound to binding point <a>, or the bound buffer object is not large enough to hold an element <b>, the values used are undefined. The binding point <a> must be a nonnegative integer constant.

For program parameter buffer array declarations, "program.buffer[a][b..c]" is equivalent to specifying elements <b> through <c> of the buffer object bound to binding point <a> in order.

For program parameter buffer array declarations, "program.buffer[a]" is equivalent to specifying the entire buffer -- elements 0 through <n>-1, where <n> is either the size of the array (if declared) or the implementation-dependent maximum parameter buffer object size limit (if no size is declared).

Section 2.X.3.7, Program Condition Code Registers

The program condition code registers are four-component vectors. Each component of this register is a collection of single-bit flags, including a sign flag (SF), a zero flag (ZF), an overflow flag (OF), and a carry flag (CF). There are two condition code registers (CC0 and CC1), whose values are undefined at the beginning of program execution.

Most program instructions can optionally update one of the condition code registers, by designating the condition code to update in the instruction. When a condition code component is updated, the four flags of each component of the condition code are set according to the corresponding component of the instruction result. Full details on the condition code updates and tests can be found in Section 2.X.4.3.

The value of these four flags can be combined in various condition code tests, which can be used to mask writes to destination variables and to perform conditional branches or other condition operations.

Section 2.X.3.8, Program Aliases

Programs can create aliases by matching the <ALIAS_statement> grammar rule. Aliases allow programs to use multiple variable names to refer to a single underlying variable. For example, the statement

```
ALIAS var1 = var0
```

establishes a variable name of "var1". Subsequent references to "var1" in the program text are treated as references to "var0". The left hand side of an ALIAS statement must be a new variable name, and the right hand side must be an established variable name.

Aliases are not considered variable declarations, so do not count against the limits on the number of variable declarations allowed in the program text.

Section 2.X.3.9, Program Resource Limits

(see ARB_vertex_program specification, incorporates all the different limits on instruction counts, temporaries, attribute bindings, program parameters, and so on)

Section 2.X.4, Program Execution Environment

The set of instructions supported for GPU programs is given in Table X.13 below and is described in detail in Section 2.X.8. An instruction can use up to three operands when it executes, and most instructions can write a single result vector. Instructions may also specify one or more modifiers, according to the <opModifiers> grammar rule. Instruction modifiers affect how the specified operation is performed.

GPU programs may operate on signed integer, unsigned integer, or floating-point values; some instructions are capable of operating on any of the three types. However, the data type of the operands and the result are always determined based solely on the instruction and its modifiers. If any of the variables used in the instruction are typeless, they will be interpreted according to the data type derived from the instruction. If any variables with a conflicting data type are used in the instruction, the program will fail to load unless the "NTC" (no type checking) instruction modifier is specified.

	Modifiers								
Instruction	F	I	C	S	H	D	Out	Inputs	Description
ABS	X	X	X	X	X	F	v	v	absolute value
ADD	X	X	X	X	X	F	v	v,v	add
AND	-	X	X	-	-	S	v	v,v	bitwise and
BRK	-	-	-	-	-	-	-	c	break out of loop instruction
CAL	-	-	-	-	-	-	-	c	subroutine call
CEIL	X	X	X	X	X	F	v	vf	ceiling
CMP	X	X	X	X	X	F	v	v,v,v	compare
CONT	-	-	-	-	-	-	-	c	continue with next loop interation
COS	X	-	X	X	X	F	s	s	cosine with reduction to [-PI,PI]
DIV	X	X	X	X	X	F	v	v,s	divide vector components by scalar
DP2	X	-	X	X	X	F	s	v,v	2-component dot product
DP2A	X	-	X	X	X	F	s	v,v,v	2-comp. dot product w/scalar add
DP3	X	-	X	X	X	F	s	v,v	3-component dot product
DP4	X	-	X	X	X	F	s	v,v	4-component dot product
DPH	X	-	X	X	X	F	s	v,v	homogeneous dot product
DST	X	-	X	X	X	F	v	v,v	distance vector
ELSE	-	-	-	-	-	-	-	-	start if test else block
ENDIF	-	-	-	-	-	-	-	-	end if test block
ENDREP	-	-	-	-	-	-	-	-	end of repeat block
EX2	X	-	X	X	X	F	s	s	exponential base 2
FLR	X	X	X	X	X	F	v	vf	floor
FRC	X	-	X	X	X	F	v	v	fraction
I2F	-	X	X	-	-	S	vf	v	integer to float
IF	-	-	-	-	-	-	-	c	start of if test block
KIL	X	X	-	-	X	F	-	vc	kill fragment
LG2	X	-	X	X	X	F	s	s	logarithm base 2
LIT	X	-	X	X	X	F	v	v	compute lighting coefficients
LRP	X	-	X	X	X	F	v	v,v,v	linear interpolation
MAD	X	X	X	X	X	F	v	v,v,v	multiply and add
MAX	X	X	X	X	X	F	v	v,v	maximum
MIN	X	X	X	X	X	F	v	v,v	minimum
MOD	-	X	X	-	-	S	v	v,s	modulus vector components by scalar
MOV	X	X	X	X	X	F	v	v	move
MUL	X	X	X	X	X	F	v	v,v	multiply
NOT	-	X	X	-	-	S	v	v	bitwise not
NRM	X	-	X	X	X	F	v	v	normalize 3-component vector
OR	-	X	X	-	-	S	v	v,v	bitwise or
PK2H	X	X	-	-	-	F	s	vf	pack two 16-bit floats
PK2US	X	X	-	-	-	F	s	vf	pack two floats as unsigned 16-bit
PK4B	X	X	-	-	-	F	s	vf	pack four floats as signed 8-bit
PK4UB	X	X	-	-	-	F	s	vf	pack four floats as unsigned 8-bit
POW	X	-	X	X	X	F	s	s,s	exponentiate
RCC	X	-	X	X	X	F	s	s	reciprocal (clamped)
RCP	X	-	X	X	X	F	s	s	reciprocal
REP	X	X	-	-	X	F	-	v	start of repeat block
RET	-	-	-	-	-	-	-	c	subroutine return
RFL	X	-	X	X	X	F	v	v,v	reflection vector
ROUND	X	X	X	X	X	F	v	vf	round to nearest integer
RSQ	X	-	X	X	X	F	s	s	reciprocal square root
SAD	-	X	X	-	-	S	vu	v,v,vu	sum of absolute differences
SCS	X	-	X	X	X	F	v	s	sine/cosine without reduction
SEQ	X	X	X	X	X	F	v	v,v	set on equal
SFL	X	X	X	X	X	F	v	v,v	set on false
SGE	X	X	X	X	X	F	v	v,v	set on greater than or equal

	Modifiers								
Instruction	F	I	C	S	H	D	Out	Inputs	Description
SGT	X	X	X	X	X	F	v	v,v	set on greater than
SHL	-	X	X	-	-	S	v	v,s	shift left
SHR	-	X	X	-	-	S	v	v,s	shift right
SIN	X	-	X	X	X	F	s	s	sine with reduction to [-PI,PI]
SLE	X	X	X	X	X	F	v	v,v	set on less than or equal
SLT	X	X	X	X	X	F	v	v,v	set on less than
SNE	X	X	X	X	X	F	v	v,v	set on not equal
SSG	X	-	X	X	X	F	v	v	set sign
STR	X	X	X	X	X	F	v	v,v	set on true
SUB	X	X	X	X	X	F	v	v,v	subtract
SWZ	X	-	X	X	X	F	v	v	extended swizzle
TEX	X	X	X	X	-	F	v	vf	texture sample
TRUNC	X	X	X	X	X	F	v	vf	truncate (round toward zero)
TXB	X	X	X	X	-	F	v	vf	texture sample with bias
TXD	X	X	X	X	-	F	v	vf,vf,vf	texture sample w/partials
TXF	X	X	X	X	-	F	v	vs	texel fetch
TXL	X	X	X	X	-	F	v	vf	texture sample w/LOD
TXP	X	X	X	X	-	F	v	vf	texture sample w/projection
TXQ	-	-	-	-	-	S	vs	vs	texture info query
UP2H	X	X	X	X	-	F	vf	s	unpack two 16-bit floats
UP2US	X	X	X	X	-	F	vf	s	unpack two unsigned 16-bit ints
UP4B	X	X	X	X	-	F	vf	s	unpack four signed 8-bit ints
UP4UB	X	X	X	X	-	F	vf	s	unpack four unsigned 8-bit ints
X2D	X	-	X	X	X	F	v	v,v,v	2D coordinate transformation
XOR	-	X	X	-	-	S	v	v,v	exclusive or
XPD	X	-	X	X	X	F	v	v,v	cross product

Table X.13: Summary of NV_gpu_program4 instructions. The "Modifiers" columns specify the set of modifiers allowed for the instruction:

F = floating-point data type modifiers
I = signed and unsigned integer data type modifiers
C = condition code update modifiers
S = clamping (saturation) modifiers
H = half-precision float data type suffix
D = default data type modifier (F, U, or S)

The input and output columns describe the formats of the operands and results of the instruction.

v: 4-component vector (data type is inherited from operation)
vf: 4-component vector (data type is always floating-point)
vs: 4-component vector (data type is always signed integer)
vu: 4-component vector (data type is always unsigned integer)
s: scalar (replicated if written to a vector destination; data type is inherited from operation)
c: condition code test result (e.g., "EQ", "GT1.x")
vc: 4-component vector or condition code test

Section 2.X.4.1, Program Instruction Modifiers

There are several types of instruction modifiers available. A data type modifier specifies that an instruction should operate on signed integer, unsigned integer, or floating-point data, when multiple data types are

supported. A clamping modifier applies to instructions with floating-point results, and specifies the range to which the results should be clamped. A condition code update modifier specifies that the instruction should update one of the condition code variables. Several other special modifiers are also provided.

Instruction modifiers may be specified as stand-alone modifiers or as suffixes concatenated with the opcode name. A program will fail to load if it contains an instruction that

* specifies more than one modifier of any given type,

* specifies a clamping modifier on an instruction, unless it produces floating-point results, or

* specifies a modifier that is not supported by the instruction (see Table X.13 and the instruction description).

Stand-alone instruction modifiers are specified according to the <opModifiers> grammar rule using a ".<modifier>" syntax. Multiple modifers, separated by periods, may be specified. The set of supported modifiers is described in Table X.14.

Modifier	Description
F	Floating-point operation
U	Fixed-point operation, unsigned operands
S	Fixed-point operation, signed operands
CC	Update condition code register zero
CC0	Update condition code register zero
CC1	Update condition code register one
SAT	Floating-point results clamped to [0,1]
SSAT	Floating-point results clamped to [-1,1]
NTC	Disable type-checking on operands/results
S24	Signed multiply (24-bit operands)
U24	Unsigned multiply (24-bit operands)
HI	Multiplies two 32-bit integer operands, returns the 32 MSBs of the product

Table X.14, Instruction Modifers.

"F", "U", and "S" modifiers are data type modifiers and specify that the instruction should operate on floating-point, unsigned integer, or signed integer values, respectively. For example, "ADD.F", "ADD.U", and "ADD.S" specify component-wise addition of floating-point, unsigned integer, or signed integer vectors, respectively. These modifiers specify a data type, but do not specify a precision at which the operation is performed. Floating-point operations will be carried out with an internal precision no less than that used to represent the largest operand. Fixed-point operations will be carried out using at least as many bits as used to represent the largest operand. Operands represented with fewer bits than used to perform the instruction will be promoted to a larger data type. Signed integer operands will be sign-extended, where the most significant bits are filled with ones if the operand is negative and zero otherwise. Unsigned integer operands will be zero-extended, where the most significant bits are always filled with zeroes. For some instructions, the data type of some operands or the result are fixed; in

these cases, the data type modifier specifies the data type of the remaining values.

"CC", "CC0", and "CC1" are condition code update modifiers that specify that one of the condition code registers should be updated based on the result of the instruction, as described in section 2.X.4.3. "CC" and "CC0" specify that the condition code register CC0 be updated; "CC1" specifies an update to CC1. If no condition code update modifier is provided, the condition code registers will not be affected.

"SAT" and "SSAT" are clamping modifiers that specify that the floating-point components of the instruction result should be clamped to [0,1] or [-1,1], respectively, before updating the condition code and the destination variable. If no clamping suffix is specified, unclamped results will be used for condition code updates (if any) and destination variable writes. Clamping modifiers are not supported on instructions that do not produce floating-point results.

"NTC" (no type checking) disables data type checking on the instruction, and allows instructions to use operands or result variables whose data types are inconsistent with the expected data types of the instruction.

"S24", "U24", and "HI" are special modifiers that are allowed only for the MUL instruction, and are described in detail where MUL is documented. No more than one such modifier may be provided for any instruction.

If an instruction supports data type modifiers, but none is provided, a default data type will be chosen based on the instruction, as specified in Table X.13 and the instruction set description (Section 2.X.8). If condition code update or clamping modifiers are not specified, the corresponding operation will not be performed.

Additionally, each instruction name may have one or more suffixes, concatenated onto the base instruction name, that operate as instruction modifiers. For conciseness, these suffixes are not spelled out in the grammar -- the base opcode name is used as a placeholder for the opcode and all of its possible suffixes. Instruction suffixes are provided mainly for compatibility with prior GPU program instruction sets (e.g., NV_vertex_program3, NV_fragment_program2, and predecessors). The set of allowable suffixes, and their equivalent stand-alone modifiers, are listed in Table X.15.

Suffix	Modifier	Description
R	F	Floating-point operation, 32-bit precision
H	F(*)	Floating-point operation, at least 16-bit precision
C	CC0	Update condition code register zero
C0	CC0	Update condition code register zero
C1	CC1	Update condition code register one
_SAT	SAT	Floating-point results clamped to [0,1]
_SSAT	SSAT	Floating-point results clamped to [-1,1]

Table X.15, Instruction Suffixes.

The "R" and "H" suffixes specify floating-point operations and are equivalent to the "F" data type modifier. They additionally specify a minimum precision for the operations. Instructions with an "R" precision

modifier will be carried out at no less than IEEE single-precision floating-point (8 bits of exponent, 23 bits of mantissa). Instructions with an "H" precision modifier will be carried out at no less than 16-bit floating-point precision (5 bits of exponent, 10 bits of mantissa).

An instruction may have multiple suffixes, but they must appear in order, with data type suffixes first, followed by condition code update suffixes, followed by clamping suffixes. For example, "ADDR" carries out an add at 32-bit precision. "ADDH_SAT" carries out an add at 16-bit precision (or better) and clamps the results to [0,1]. "ADDRC1_SSAT" carries out an add at 32-bit floating-point precision, clamps the results to [-1,1], and updates condition code one based on the clamped result.

Section 2.X.4.2, Program Operands

Most program instructions operate on one or more scalar or vector operands. Each operand specifies an operand variable, which is either the name of a previously declared variable or an implicit variable declaration created by using a variable binding in the instruction. Attribute, parameter, or parameter buffer variables can be declared implicitly by using a valid binding name in an operand. Instruction operands are specified by the <instOperandV>, <instOperandS>, or <instOperandVNS> grammar rules.

If the operand variable is not an array, its contents are loaded directly. If the operand variable is an array, a single element of the array is loaded according to the <arrayMem> grammar rule. The elements of an array are numbered from 0 to <n>-1, where <n> is the number of entries in the array. Array members can be accessed using either absolute or relative addressing.

Absolute array addressing is used when the <arrayMemAbs> grammar rule is matched; the array member to load is specified by the matching integer. Out-of-bounds array absolute accesses are not allowed. If the specified member number is greater than or equal to the size of the array, the program will fail to load.

Relative array addressing is used when the <arrayMemRel> grammar rule is matched. This grammar rule allows the program to specify a scalar integer operand and an optional constant offset, according to the <arrayMemReg> and <arrayMemOffset> grammar rules. When performing relative addressing, the GL evaluates the specified integer scalar operand (according to the rules specified in this section) and adds the constant offset. The array member loaded is given by this sum. The constant offset is considered zero if an offset is omitted. If the sum is negative or exceeds the size of the array, the results of the access are undefined, but may not lead to program or GL termination. The set of constant offsets supported for relative addressing is limited to values in the range [0,<n>-1], where <n> is the size of the array. A program will fail to load if it specifies an offset outside that range. If offsets outside that range are required, they can be applied by using an integer ADD instruction writing to a temporary variable.

After the operand is loaded, its components can be rearranged according to the <swizzleSuffix> grammar rule, or it can be converted to a scalar operand according to the <scalarSuffix> grammar rule.

The <swizzleSuffix> grammar rule rearranges the components of a loaded vector to produce another vector. If the <swizzleSuffix> rule matches the <xyzwSwizzle> or <rgbaSwizzle> grammar rule, a pattern of the form ".????" is used, where each question mark is replaced with one of "x", "y", "z", "w", "r", "g", "b", or a". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. Swizzle components of "r", "g", "b", and "a" are equivalent to "x", "y", "z", and "w", respectively. For example, if the swizzle suffix is ".yzzx" or ".gbbr" and the specified source contains {2,8,9,0}, the result is the vector {8,9,9,2}. If the <swizzleSuffix> matches the <component> grammar rule, a pattern of the form ".?" is used. For this pattern, all four components of the operand are taken from the single component identified by the pattern. If the swizzle suffix is omitted, components are not rearranged and swizzling has no effect, as though ".xyzw" were specified.

The swizzle suffix rules do not allow mixing "x", "y", "z", or "w" selectors with "r", "g", "b", or "a" selectors. A program will fail to load if it contains a swizzle suffix with selectors from both of these sets.

The <scalarSuffix> grammar rule converts a vector to a scalar by selecting a single component. The <scalarSuffix> rule is similar to the swizzle selector, except that only a single component is selected. If the scalar suffix is ".y" and the specified source contains {2,8,9,0}, the value is the scalar value 8.

Next, a component-wise negate operation is performed on the operand if the <operandNeg> grammar rule matches "-". Negation is not performed if the operand has no sign prefix, or is prefixed with "+". For unsigned integer operands, the negate operand performs a two's complement operation.

Next, a component-wise absolute value operation is performed on the operand if the <instOperandAbsV> or <instOperandAbsS> grammar rule is matched, by surrounding the operand with two "|" characters. The result is optionally negated if the <operandAbsNeg> grammar rule matches "-". For unsigned integer operands, the absolute value operation has no effect.

Section 2.X.4.3, Program Destination Variable Update

Most program instructions perform computations that produce a result, which will be written to a variable. Each instruction that computes a result specifies a destination variable, which is either the name of a previously declared variable or an implicit variable declaration created by using a variable binding in the instruction. Result variables can be declared implicitly by using a valid program result binding name in the result portion of the instruction. Instruction results are specified according to the <instResult> grammar rule.

The destination variable may be a single member of an array. In this case, a single array member is specified using the <arrayMem> grammar rule, and the array member to update is computed in the exact same manner as done for operand loads. If the array member is computed at run time, and is negative or greater than or equal to the size of the array, the results of the destination variable update are undefined and could result in overwriting other program variables.

The results of the operation may be obtained at a different precision than that used to store the destination variable. If so, the results are converted to match the size of the destination variable. For floating-point values, the results are rounded to the nearest floating-point value that can be represented in the destination variable. If a result component is larger in magnitude than the largest representable floating-point value in the data type of the destination variable, an infinity encoding (+/-INF) is used. Signed or unsigned integer values are sign-extended or zero-extended, respectively, if the destination variable has more bits than the result, and have their most significant bits discarded if the destination variable has fewer bits.

Writes to individual components of a vector destination variable can be controlled at compile time by individual component write masks specified in the instruction. The component write mask is specified by the <optWriteMask> grammar rule, and is a string of up to four characters, naming the components to enable for writing. If no write mask is specified, all components are enabled for writing. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, a write mask mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not be written. The grammar requires that the destination register mask components must be listed in "xyzw" order. Additionally, write mask components of "r", "g", "b", and "a" are equivalent to "x", "y", "z", and "w", respectively. The grammar does not allow mixing "x", "y", "z", or "w" components with "r", "g", "b", and "a" ones.

Writes to individual components of a vector destination variable, or to a scalar destination variable, can also be controlled at run time using condition code write masks. The condition code write mask is specified by the <ccMask> grammar rule. If a mask is specified, a condition code variable is loaded according to the <ccMaskRule> grammar rule and tested as described in Table X.16 to produce a four-component vector of TRUE/FALSE values.

mask rule	test name	condition
EQ, EQ0, EQ1	equal	!SF && ZF
GE, GE0, GE1	greater than or equal	!(SF ^ OF)
GT, GT0, GT1	greater than	(!SF ^ OF) && !ZF
LE, LE0, LE1	less than or equal	SF ^ (ZF || OF)
LT, LT0, LT1	less than	(SF && !ZF) ^ OF
NE, NE0, NE1	not equal	SF || !ZF
FL, FL0, FL1	false	always false
TR, TR0, TR1	true	always true
NAN, NAN0, NAN1	not a number	SF && ZF
LEG, LEG0, LEG1	less, equal, or greater (anything but a NaN)	!SF || !ZF
CF, CF0, CF1	carry flag	CF
NCF, NCF0, NCF1	no carry flag	!CF
OF, OF0, OF1	overflow flag	OF
NOF, NOF0, NOF1	no overflow flag	!OF
SF, SF0, SF1	sign flag	SF
NSF, NSF0, NSF1	no sign flag	!SF
AB, AB0, AB1	above	CF && !ZF
BLE, BLE0, BLE1	below or equal	!CF || ZF

Table X.16, Condition Code Tests. The allowed rules are specified in the "mask rule" column. If "0" or "1" is appended to the rule name (e.g., "EQ1"), the corresponding condition code register (CC1 in this example) is loaded, otherwise CC0 is loaded. After loading, each component is tested, using the expression listed in the "condition" column.

After the condition code tests are performed, the four-component result can be swizzled according to the <swizzleSuffix> grammar rule. Individual components of the destination variable are written only if the corresponding component of the swizzled condition code test result is TRUE. If both a (compile-time) component write mask and a condition code write mask are specified, destination variable components are written only if the corresponding component is enabled in both masks.

A program instruction can also optionally update one of the two condition code registers if the "CC", "CC0", or "CC1" instruction modifier are specified. These instruction modifiers update condition code register CC0, CC0, or CC1, respectively. The instructions "ADD.CC" or "ADD.CC0" will perform an add and update condition code zero, "ADD.CC1" will add and update condition code one, and "ADD" will simply perform the add without a condition code update. The components of the selected condition code register are updated if and only if the corresponding component of the destination variable are enabled by both write masks. For the purposes of condition code update, a scalar destination variable is treated as a vector where the scalar result is written to "x" (if enabled in the write mask), and writes to the "y", "z", and "w" components are disabled.

When condition code components are written, the condition code flags are updated based on the corresponding component of the result. If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

For floating-point results, the sign flag (SF) is set if the result is less than zero or is a NaN (not a number) value. The zero flag (ZF) is set if the result is equal to zero or is a NaN.

For signed and unsigned integer results, the sign flag (SF) is set if the most significant bit of the value written to the result variable is set and the zero flag (ZF) is set if the result written is zero. For instructions other than those performing an integer add or subtract (ADD, MAD, SAD, SUB), the overflow and carry flags (OF and CF) are cleared.

For integer add or subtract operations, the overflow and carry flags by doing both signed and unsigned adds/subtracts as follows:

  The overflow flag (OF) is set by interpreting the two operands as signed integers and performing a signed add or subtract. If the result is representable as a signed integer (i.e., doesn't overflow), the overflow flag is cleared; otherwise, it is set.

  The carry flag (CF) is set by interpreting the two operands as unsigned integers and performing an unsigned add or subtract. If the result of an add is representable as an unsigned integer (i.e., doesn't overflow), the carry flag is cleared; otherwise, it is set. If the result of a subtract is greater than or equal to zero, the carry flag is set; otherwise, it is cleared.

For the purposes of condition code setting, negation modifiers turn add operations into subtracts and vice versa. If the operation is equivalent to an add with both operands negated (-A-B), the carry and overflow flags are both undefined.

Section 2.X.4.4, Program Texture Access

Certain program instructions may access texture images, as described in section 3.8. The coordinates, level-of-detail, and partial derivatives used for performing the texture lookup are derived from values provided in the program as described in the various sub-sections of Section 2.X.8. These descriptions use the function

```
result_t_vec
  TextureSample(float_vec coord, float lod, float_vec ddx,
                float_vec ddy, int_vec offset);
```

which obtains a filtered texel value <tau> as described in Section 3.8.8 and returns a 4-component vector (R,G,B,A) according to the format conversions specified in Table 3.21. The result vector is interpreted as floating-point, signed integer, or unsigned integer, according to the data type modifier of the instruction. If the internal format of the texture does not match the instruction's data type modifer, the results of the texture lookup are undefined.

(Note: For unextended OpenGL 2.0, all supported texture internal formats store integer values but return floating-point results in the range [0,1] on a texture lookup. The ARB_texture_float extension introduces floating-point internal format where components are both stored and returned as floating-point values. The EXT_texture_integer extension introduces formats that both store and return either signed or unsigned integer values.)

<coord> is a four-component floating-point vector from which the (s,t,r) texture coordinates used for the texture access, the layer used for array textures, and the reference value used for depth comparisons (section 3.8.14) are extracted according to Table X.17. If the texture is a cube map, (s,t,r) is projected to one of the six cube faces to produce a new (s,t) vector according to Section 3.8.6. For array textures, the layer used is derived by rounding the extracted floating-point component to the nearest integer and clamping the result to the range [0,<n>-1], where <n> is the number of layers in the texture.

<lod> specifies the level of detail parameter and replaces the value computed in equation 3.18. <ddx> and <ddy> specify partial derivatives (ds/dx, dt/dx, dr/dx, ds/dy, dt/dy, and dr/dy) for the texture coordinates, and may be used to derive footprint shapes for anisotropic texture filtering.

<offset> is a constant 3-component signed integer vector specified according to the <texOffset> grammar rule, which is added to the computed <u>, <v>, and <w> texel locations prior to sampling. One, two, or three components may be specified in the instruction; if fewer than three are specified, the remaining offset components are zero. A limited range of offset values are supported; the minimum and maximum <texOffset> values are implementation-dependent and given by MIN_PROGRAM_TEXEL_OFFSET_EXT and MAX_PROGRAM_TEXEL_OFFSET_EXT, respectively. A program will fail to load:

  * if the texture target specified in the instruction is 1D, ARRAY1D, SHADOW1D, or SHADOWARRAY1D, and the second or third component of the offset vector is non-zero,

  * if the texture target specified in the instruction is 2D, RECT, ARRAY2D, SHADOW2D, SHADOWRECT, or SHADOWARRAY2D, and the third component of the offset vector is non-zero,

  * if the texture target is CUBE or SHADOWCUBE, and any component of the offset vector is non-zero -- texel offsets are not supported for cube map or buffer textures, or

  * if any component of the offset vector is less than MIN_PROGRAM_TEXEL_OFFSET_EXT or greater than MAX_PROGRAM_TEXEL_OFFSET_EXT.

(NOTE: Texel offsets are a new feature provided by this extension and are described in more detail in edits to Section 3.8 below.)

The texture used by TextureSample() is one of the textures bound to the texture image unit whose number is specified in the instruction according to the <texImageUnit> grammar rule. The texture target accessed is specified according to the <texTarget> grammar rule and Table X.17. Fixed-function texture enables are always ignored when determining the texture to access in a program.

texTarget	Texture Type	coordinates used s t r	layer	shadow
1D	TEXTURE_1D	x - -	-	-
2D	TEXTURE_2D	x y -	-	-
3D	TEXTURE_3D	x y z	-	-
CUBE	TEXTURE_CUBE_MAP	x y z	-	-
RECT	TEXTURE_RECTANGLE_ARB	x y -	-	-
ARRAY1D	TEXTURE_1D_ARRAY_EXT	x - -	y	-
ARRAY2D	TEXTURE_2D_ARRAY_EXT	x y -	z	-
SHADOW1D	TEXTURE_1D	x - -	-	z
SHADOW2D	TEXTURE_2D	x y -	-	z
SHADOWRECT	TEXTURE_RECTANGLE_ARB	x y -	-	z
SHADOWCUBE	TEXTURE_CUBE_MAP	x y z	-	w
SHADOWARRAY1D	TEXTURE_1D_ARRAY_EXT	x - -	y	z
SHADOWARRAY2D	TEXTURE_2D_ARRAY_EXT	x y -	z	w
BUFFER	TEXTURE_BUFFER_EXT	<not supported>		

Table X.17: Texture types accessed for each of the <texTarget>, and coordinate mappings. The "SHADOW" and "ARRAY" targets are special pseudo-targets described below. The "coordinates used" column indicate the input values used for each coordinate of the texture lookup, the layer selector for array textures, and the reference value for texture comparisons. Buffer textures are not supported by normal texture lookup functions, but are supported by TXF and TXQ, described below.

Texture targets with "SHADOW" are used to access textures with a DEPTH_COMPONENT base internal format using depth comparisons (Section 3.8.14). Results of a texture access are undefined:

* if a "SHADOW" target is used, and the corresponding texture has a base internal format other than DEPTH_COMPONENT or a TEXTURE_COMPARE_MODE of NONE, or

* if a non-"SHADOW" target is used, and the corresponding texture has a base internal format of DEPTH_COMPONENT and a TEXTURE_COMPARE_MODE other than NONE.

If the texture being accessed is not complete (or cube complete for cubemap textures), no texture access is performed and the result is undefined.

A program will fail to load if it attempts to sample from multiple texture targets (including the SHADOW pseudo-targets) on the same texture image unit. For example, a program containing any two the following instructions will fail to load:

```
TEX out, coord, texture[0], 1D;
TEX out, coord, texture[0], 2D;
TEX out, coord, texture[0], ARRAY2D;
TEX out, coord, texture[0], SHADOW2D;
TEX out, coord, texture[0], 3D;
```

Additionally, multiple texture targets for a single texture image unit may not be used at the same time by the GL. The error INVALID_OPERATION is generated by Begin, RasterPos, or any command that performs an implicit Begin if an enabled program accesses one texture target for a texture unit

while another enabled program or fixed-function fragment processing accesses a different texture target for the same texture image unit.

Some texture instructions use standard methods to compute partial derivatives and/or the level-of-detail used to perform texture accesses. For fragment programs, the functions

```
float_vec ComputePartialsX(float_vec coord);
float_vec ComputePartialsY(float_vec coord);
```

compute approximate component-wise partial derivatives of the floating-point vector <coord> relative to the X and Y coordinates, respectively. For vertex and geometry programs, these functions always return (0,0,0,0). The function

```
float ComputeLOD(float_vec ddx, float_vec ddy);
```

maps partial derivative vectors <ddx> and <ddy> to ds/dx, dt/dx, dr/dx, ds/dy, dt/dy, and dr/dy and computes lambda_base(x,y) according to equation 3.18.

The TXF instruction provides the ability to extract a single texel from a specified texture image using the function

```
result_t_vec TexelFetch(uint_vec coord, int_vec offset);
```

The extracted texel is converted to an (R,G,B,A) vector according to Table 3.21. The result vector is interpreted as floating-point, signed integer, or unsigned integer, according to the data type modifier of the instruction. If the internal format of the texture is not compatible with the instruction's data type modifer, the extracted texel value is undefined.

<coord> is a four-component signed integer vector used to identify the single texel accessed. The (i,j,k) coordinates of the texel and the layer used for array textures are extracted according to Table X.18. The level of detail accessed is obtained by adding the w component of <coord> to the base level (level_base). <offset> is a constant 3-component signed integer vector added to the texel coordinates prior to the texel fetch as described above. In addition to the restrictions described above, non-zero offset components are also not supported for BUFFER targets.

The texture used by TexelFetch() is specified by the image unit and target parameters provided in the instruction, as for TextureSample() above. Single texel fetches can not perform depth comparisons or access cubemaps. If a program contains a TXF instruction specifying one of the "SHADOW" or "CUBE" targets, it will fail to load.

		coordinates used				
texTarget	supported	i	j	k	layer	lod
1D	yes	x	-	-	-	w
2D	yes	x	y	-	-	w
3D	yes	x	y	z	-	w
CUBE	no	-	-	-	-	-
RECT	yes	x	y	-	-	w
ARRAY1D	yes	x	-	-	y	w
ARRAY2D	yes	x	y	-	z	w
SHADOW1D	no	-	-	-	-	-
SHADOW2D	no	-	-	-	-	-
SHADOWRECT	no	-	-	-	-	-
SHADOWCUBE	no	-	-	-	-	-
SHADOWARRAY1D	no	-	-	-	-	-
SHADOWARRAY2D	no	-	-	-	-	-
BUFFER	yes	x	-	-	-	-

Table X.18, Mappings of texel fetch coordinates to texel location.

Single-texel fetches do not support LOD clamping or any texture wrap mode, and require a mipmapped minification filter to access any level of detail other than the base level. The results of the texel fetch are undefined:

* if the computed LOD is less than the texture's base level (level_base) or greater than the maximum level (level_max),

* if the computed LOD is not the texture's base level and the texture's minification filter is NEAREST or LINEAR,

* if the layer specified for array textures is negative or greater than the number of layers in the array texture,

* if the texel at (i,j,k) coordinates refer to a border texel outside the defined extents of the specified LOD, where

  i < -b_s, j < -b_s, k < -b_s,
  i >= w_s - b_s, j >= h_s - b_s, or k >= d_s - b_s,

  where the size parameters (w_s, h_s, d_s, and b_s) refer to the width, height, depth, and border size of the image, as in equations 3.15, 3.16, and 3.17, or

* if the texture being accessed is not complete (or cube complete for cubemaps).

Section 2.X.5, Program Flow Control

In addition to basic arithmetic, logical, and texture instructions, a number of flow control instructions are provided, which are described in detail in Section 2.X.8. Programs can contain several types of instruction blocks: IF/ELSE/ENDIF blocks, REP/ENDREP blocks, and subroutine blocks. IF/ELSE/ENDIF blocks are a set of instructions beginning with an "IF" instruction, ending with an "ENDIF" instruction, and possibly containing an optional "ELSE" instruction. REP/ENDREP blocks are a set of instructions beginning with a "REP" instruction and ending with an "ENDREP" instruction. Subroutine blocks begin with an instruction

label identifying the name of the subroutine and ending just before the next instruction label or the end of the program. Examples include the following:

```
    MOVC CC, R0;
    IF GT.x;
      MOV R0, R1;     # executes if R0.x > 0
    ELSE;
      MOV R0, R2;     # executes if R0.x <= 0
    ENDIF;

    REP repCount;
    ADD R0, R0, R1;
    ENDREP;

  square:             # subroutine to compute R0^2
    MUL R0, R0, R0;
    RET;
  main:
    MOV R0, 9.0;
    CAL square;       # compute 9.0^2 in R0
```

IF/ELSE/ENDIF and REP/ENDREP blocks may be nested inside each other, and inside subroutines. In all cases, each instruction block must be terminated with the appropriate instruction (ENDIF for IF, ENDREP for REP). Nested instruction blocks must be wholly contained within a block -- if a REP instruction is found between an IF and ELSE instruction, the corresponding ENDREP must also be present between the IF and ELSE. Subroutines may not be nested inside IF/ELSE/ENDIF or REP/ENDREP blocks, or inside other subroutines. A program will fail to load if any instruction block is terminated by an incorrect instruction, is not terminated before the block containing it, or contains an instruction label.

IF/ELSE/ENDIF blocks evaluate a condition to determine which instructions to execute. If the condition is true, all instructions between the IF and ELSE are executed. If the condition is false, all instructions between the ELSE and ENDIF are executed. The ELSE instruction is optional. If the ELSE is omitted, all instructions between the IF and ENDIF are executed if the condition is true, or skipped if the condition is false. A limited amount of nesting is supported -- a program will fail to load if an IF instruction is nested inside MAX_PROGRAM_IF_DEPTH_NV or more IF/ELSE/ENDIF blocks.

REP/ENDREP blocks are used to execute a sequence of instructions multiple times. The REP instruction includes an optional scalar operand to specify a loop count indicating the number of times the block of instructions should be repeated. If the loop count is omitted, the contents of a REP/ENDREP block will be repeated indefinitely until the loop is explicitly terminated. A limited amount of nesting is supported -- a program will fail to load if a REP instruction is nested inside MAX_PROGRAM_LOOP_DEPTH_NV or more REP/ENDREP blocks.

Within a REP/ENDREP block, the CONT instruction can be used to terminate the current iteration of the loop by effectively jumping to the ENDREP instruction. The BRK instruction can be used to terminate the entire loop by effectively jumping to the instruction immediately following the ENDREP

instruction. If CONT and BRK instructions are found inside multiply nested REP/ENDREP blocks, they apply to the innermost block. A program will fail to load if it includes a CONT or BRK instruction that is not contained inside a REP/ENDREP block.

A REP/ENDREP block without a specified loop count can result in an infinite loop. To prevent obvious infinite loops, a program will fail to load if it contains a REP/ENDREP block that contains neither a BRK instruction at the current nesting level or a RET instruction at any nesting level.

Subroutines are supported via the CAL and RET instructions. A subroutine block is identified by an instruction, which can be any valid identifier according to the <instLabel> grammar rule. The CAL instruction identifies a subroutine name to call according to the <instTarget> grammar rule. Instruction labels used in CAL instructions do not need to be defined in the program text that precedes the instruction, but a program will fail to load if it includes a CAL instruction that references an instruction label that is not defined anywhere in the program. When a CAL instruction is executed, it transfers control to the instruction immediately following the specified instruction label. Subsequent instructions in that subroutine are executed until a RET instruction is executed, or until program execution reaches another instruction label or the end of the program text. After the subroutine finishes, execution continues with the instruction immediately following the CAL instruction. When a RET instruction is issued, it will break out of any IF/ELSE/ENDIF or REP/ENDREP blocks that contain it.

Subroutines may call other subroutines before completing, up to an implementation-dependent maximum depth of MAX_PROGRAM_CALL_DEPTH_NV calls. Subroutines may call any subroutine in the program, including themselves, as long as the call depth limit is obeyed. The results of issuing a CAL instruction while MAX_PROGRAM_CALL_DEPTH subroutines have not completed has undefined results, including possible program termination.

Several flow control instructions include condition code tests. The IF instruction requires a condition test to determine what instructions are executed. The CONT, BRK, CAL, and RET instructions have an optional condition code test; if the test fails, the instructions are not executed. Condition code tests are specified by the <ccTest> grammar rule. The test is evaluated like the condition code write mask (section 2.X.4.3), and passes if and only if any of the four components passes.

If an instruction label named "main" is specified, GPU program execution begins with the instruction immediately following that label. Otherwise, it begins with the first instruction of the program. Instructions are executed in sequence until either a RET instruction is issued in the main subroutine or the end of the program text is reached.

Section 2.X.6, Program Options

Programs may specify a number of options to indicate that one or more extended language features are used by the program. All program options used by the program must be declared at the beginning of the program string. Each program option specified in a program string will modify the syntactic or semantic rules used to interpet the program and the execution environment used to execute the program. Features in program options

not declared by the program are ignored, even if the option is otherwise supported by the GL. Each option declaration consists of two tokens: the keyword "OPTION" and an identifier.

The set of available options depends on the program type, and is enumerated in the specifications for each program type. Some program types may not provide any options.

Section 2.X.7, Program Declarations

Programs may include a number of declaration statements to specify characteristics of the program. Each declaration statement is followed by one or more arguments, separated by commas.

The set of available declarations depends on the program type, and is enumerated in the specifications for each program type. Some program types may not provide declarations.

Section 2.X.8, Program Instruction Set

The following sections enumerate the set of instructions supported for GPU programs.

Some instructions allow the use of one of the three basic data type modifiers (floating point, signed integer, and unsigned integer). Unless otherwise mentioned:

* the result and all of the operands will be interpreted according to the specified data type, and

* if no data type modifier is specified, the instruction will operate as though a floating-point modifier ("F") were specified.

Some instructions will override one or both of these rules.

Section 2.X.8.Z, ABS: Absolute Value

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
tmp = VectorLoad(op0);
result.x = abs(tmp.x);
result.y = abs(tmp.y);
result.z = abs(tmp.z);
result.w = abs(tmp.w);
```

ABS supports all three data type modifiers. Taking the absolute value of an unsigned integer is not a useful operation, but is not illegal.

Section 2.X.8.Z, ADD: Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x + tmp1.x;
result.y = tmp0.y + tmp1.y;
result.z = tmp0.z + tmp1.z;
result.w = tmp0.w + tmp1.w;
```

ADD supports all three data type modifiers.

Section 2.X.8.Z, AND: Bitwise AND

The AND instruction performs a bitwise AND operation on the components of the two source vectors to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x & tmp1.x;
result.y = tmp0.y & tmp1.y;
result.z = tmp0.z & tmp1.z;
result.w = tmp0.w & tmp1.w;
```

AND supports only signed and unsigned integer data type modifiers. If no type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, BRK: Break out of Loop Instruction

The BRK instruction conditionally transfers control to the instruction immediately following the next ENDREP instruction. A BRK instruction has no effect if the condition code test evaluates to FALSE.

The following pseudocode describes the operation of the instruction:

```
if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.***c)) {
  continue execution at instruction following the next ENDREP;
}
```

Section 2.X.8.Z, CAL: Subroutine Call

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. It also pushes a reference to the instruction immediately following the CAL instruction onto the call stack, where execution will continue after executing the matching RET instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
      // undefined results
    } else {
      callStack[callStackDepth] = nextInstruction;
      callStackDepth++;
    }
    // continue execution at instruction following <instTarget>
  } else {
    // do nothing
  }
```

In the pseudocode, <instTarget> is the label specified in the instruction matching the <branchLabel> grammar rule, <callStackDepth> is the current depth of the call stack, <callStack> is an array holding the call stack, and <nextInstruction> is a reference to the instruction immediately following the CAL instruction in the program string.

If the call stack overflows, the results of the CAL instruction are undefined, and can result in immediate program termination.

An instruction label signifies the beginning of a new subroutine. Subroutines may not nest or overlap. If a CAL instruction is executed and subsequent program execution reaches an instruction label before a corresponding RET instruction is executed, the subroutine call returns immediately, as though an unconditional RET instruction were inserted immediately before the instruction label.

(Note: On previous vertex program extensions -- NV_vertex_program2 and NV_vertex_program3 -- instruction labels were also used as targets for branch (BRA) instructions. This unstructured branching functionality has been replaced with the structured branching constructs found in this instruction set.)

Section 2.X.8.Z, CEIL: Ceiling

The CEIL instruction loads a single vector operand and performs a component-wise ceiling operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  iresult.x = ceil(tmp.x);
  iresult.y = ceil(tmp.y);
  iresult.z = ceil(tmp.z);
  iresult.w = ceil(tmp.w);
```

The ceiling operation returns the nearest integer greater than or equal to the operand. For example ceil(-1.7) = -1.0, ceil(+1.0) = +1.0, and ceil(+3.7) = +4.0.

CEIL supports all three data type modifiers. The single operand is always treated as a floating-point vector, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, CMP: Compare

The CMP instructions performs a component-wise comparison of the first operand against zero, and copies the values of the second or third operands based on the results of the compare.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = (tmp0.x < 0) ? tmp1.x : tmp2.x;
result.y = (tmp0.y < 0) ? tmp1.y : tmp2.y;
result.z = (tmp0.z < 0) ? tmp1.z : tmp2.z;
result.w = (tmp0.w < 0) ? tmp1.w : tmp2.w;
```

CMP supports all three data type modifiers. CMP with an unsigned data type modifier is not a useful operation, but is not illegal.

Section 2.X.8.Z, CONT: Continue with Next Loop Iteration

The CONT instruction conditionally transfers control to the next ENDREP instruction. A CONT instruction has no effect if the condition code test evaluates to FALSE.

The following pseudocode describes the operation of the instruction:

```
if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.***c)) {
  continue execution at the next ENDREP;
}
```

Section 2.X.8.Z, COS: Cosine with Reduction to [-PI,PI]

The COS instruction approximates the trigonometric cosine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
tmp = ScalarLoad(op0);
result.x = ApproxCosine(tmp);
result.y = ApproxCosine(tmp);
result.z = ApproxCosine(tmp);
result.w = ApproxCosine(tmp);
```

COS supports only floating-point data type modifiers.

Section 2.X.8.Z, DDX: Partial Derivative Relative to X

The DDX instruction computes approximate partial derivatives of a vector operand with respect to the X window coordinate, and is only available to fragment programs. See the NV_fragment_program4 specification for more details.

Section 2.X.8.Z, DDY: Partial Derivative Relative to Y

The DDY instruction computes approximate partial derivatives of a vector operand with respect to the Y window coordinate, and is only available to fragment programs. See the NV_fragment_program4 specification for more details.

Section 2.X.8.Z, DIV: Divide Vector Components by Scalar

The DIV instruction performs a component-wise divide of the first vector operand by the second scalar operand to produce a 4-component result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x / tmp1;
  result.y = tmp0.y / tmp1;
  result.z = tmp0.z / tmp1;
  result.w = tmp0.w / tmp1;
```

DIV supports all three data type modifiers. For floating-point division, this instruction is not guaranteed to produce results identical to a RCP/MUL instruction sequence.

The results of an signed or unsigned integer division by zero are undefined.

Section 2.X.8.Z, DP2: 2-Component Dot Product

The DP2 instruction computes a two-component dot product of the two operands (using the first two components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP2 supports only floating-point data type modifiers.

Section 2.X.8.Z, DP2A: 2-Component Dot Product with Scalar Add

The DP2 instruction computes a two-component dot product of the two operands (using the first two components), adds the x component of the third operand, and replicates the result to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + tmp2.x;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP2A supports only floating-point data type modifiers.

Section 2.X.8.Z, DP3: 3-Component Dot Product

The DP3 instruction computes a three-component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP3 supports only floating-point data type modifiers.

Section 2.X.8.Z, DP4: 4-Component Dot Product

The DP4 instruction computes a four-component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DP4 supports only floating-point data type modifiers.

Section 2.X.8.Z, DPH: Homogeneous Dot Product

The DPH instruction computes a three-component dot product of the two operands (using the x, y, and z components), adds the w component of the second operand, and replicates the sum to all four components of the result vector. This is equivalent to a four-component dot product where the w component of the first operand is forced to 1.0.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + tmp1.w;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

DPH supports only floating-point data type modifiers.

Section 2.X.8.Z, DST: Distance Vector

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-vertex light attenuation calculations: a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

DST supports only floating-point data type modifiers.

Section 2.X.8.Z, ELSE: Start of If Test Else Block

The ELSE instruction signifies the end of the "execute if true" portion of an IF/ELSE/ENDIF block and the beginning of the "execute if false" portion.

If the condition evaluated at the IF statement was TRUE, when a program reaches the ELSE statement, it has completed the entire "execute if true"

portion of the IF/ELSE/ENDIF block. Execution will continue at the corresponding ENDIF instruction.

If the condition evaluated at the IF statement was FALSE, program execution would skip over the entire "execute if true" portion of the IF/ELSE/ENDIF block, including the ELSE instruction.

Section 2.X.8.Z, EMIT: Emit Vertex

The EMIT instruction emits a new vertex to be added to the current output primitive generated by a geometry program, and is only available to geometry programs. See the NV_geometry_program4 specification for more details.

Section 2.X.8.Z, ENDIF: End of If Test Block

The ENDIF instruction signifies the end of an IF/ELSE/ENDIF block. It has no other effect on program execution.

Section 2.X.8,Z, ENDPRIM: End of Primitive

A geometry program can emit multiple primitives in a single invocation. The ENDPRIM instruction is used in a geometry program to signify the end of the current primitive and the beginning of a new primitive of the same type. It is only available to geometry programs. See the NV_geometry_program4 specification for more details.

Section 2.X.8.Z, ENDREP: End of Repeat Block

The ENDREP instruction specifies the end of a REP block.

When used with in conjunction with a REP instruction with a loop count, ENDREP decrements the loop counter. If the decremented loop counter is greater than zero, ENDREP transfers control to the instruction immediately after the corresponding REP instruction. If the loop counter is less than or equal to zero, execution continues at the instruction following the ENDREP instruction. When used in conjunction with a REP instruction without loop count, ENDREP always transfers control to the instruction immediately after the REP instruction.

```
  if (REP instruction includes a loop count) {
    LoopCount--;
    if (LoopCount > 0) {
      continue execution at instruction following corresponding REP
        instruction;
    }
  } else {
    continue execution at instruction following corresponding REP
      instruction;
  }
```

Section 2.X.8.Z, EX2: Exponential Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates the approximation to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

EX2 supports only floating-point data type modifiers.

Section 2.X.8.Z, FLR: Floor

The FLR instruction loads a single vector operand and performs a component-wise floor operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

The floor operation returns the nearest integer less than or equal to the operand. For example floor(-1.7) = -2.0, floor(+1.0) = +1.0, and floor(+3.7) = +3.0.

FLR supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, FRC: Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.0, 1.0).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = fraction(tmp.x);
  result.y = fraction(tmp.y);
  result.z = fraction(tmp.z);
  result.w = fraction(tmp.w);
```

FRC supports only floating-point data type modifiers.

Section 2.X.8.Z, I2F: Integer to Float

The I2F instruction converts the components of an integer vector operand to floating-point to produce a floating-point result vector.

```
  tmp = VectorLoad(op0);
  result.x = (float) tmp.x;
  result.y = (float) tmp.y;
  result.z = (float) tmp.z;
  result.w = (float) tmp.w;
```

I2F supports only signed and unsigned integer data type modifiers. The single operand is interpreted according to the data type modifier. If no data type modifier is specified, the operand is treated as a signed integer vector. The result is always written as a float.

Section 2.X.8.Z, IF: Start of If Test Block

The IF instruction performs a condition code test to determine what instructions inside an IF/ELSE/ENDIF block are executed. If the test passes, execution continues at the instruction immediately following the IF instruction. If the test fails, IF transfers control to the instruction immediately following the corresponding ELSE instruction (if present) or the ENDIF instruction (if no ELSE is present).

Implementations may have a limited ability to nest IF blocks in any subroutine. If the number of IF/ENDIF blocks nested inside each other is MAX_PROGRAM_IF_DEPTH_NV or higher, a program will fail to compile.

```
  // Evaluate the condition.  If the condition is true, continue at the
  // next instruction.  Otherwise, continue at the
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    continue execution at the next instruction;
  } else if (IF block contains an ELSE statement) {
    continue execution at instruction following corresponding ELSE;
  } else {
    continue execution at instruction following corresponding ENDIF;
  }
```

(Note: Unlike the NV_fragment_program2 extension, there is no run-time limit on the maximum overall depth of IF/ENDIF nesting. As long as each individual subroutine of the program obeys the static nesting limits, there will be no run-time errors in the program. With the NV_fragment_program2 extension, a program could terminate abnormally if it called a subroutine inside a very deeply nested set of IF/ENDIF blocks and the called subroutine also contained deeply nested IF/ENDIF blocks. SUch an error could occur even if neither subroutine exceeded static limits.)

Section 2.X.8.Z, KIL: Kill Fragment

The KIL instruction conditionally kills a fragment, and is only available to fragment programs. See the NV_fragment_program4 specification for more details.

Section 2.X.8.Z, LG2: Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
tmp = ScalarLoad(op0);
result.x = ApproxLog2(tmp);
result.y = ApproxLog2(tmp);
result.z = ApproxLog2(tmp);
result.w = ApproxLog2(tmp);
```

If the scalar operand is zero or negative, the result is undefined.

LG2 supports only floating-point data type modifiers.

Section 2.X.8.Z, LIT: Compute Lighting Coefficients

The LIT instruction accelerates lighting computations by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the single operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
tmp = VectorLoad(op0);
if (tmp.x < 0) tmp.x = 0;
if (tmp.y < 0) tmp.y = 0;
if (tmp.w < -(128.0-epsilon)) tmp.w = -(128.0-epsilon);
else if (tmp.w > 128-epsilon) tmp.w = 128-epsilon;
result.x = 1.0;
result.y = tmp.x;
result.z = (tmp.x > 0) ? RoughApproxPower(tmp.y, tmp.w) : 0.0;
result.w = 1.0;
```

Since 0^0 is defined to be 1, RoughApproxPower(0.0, 0.0) will produce 1.0.

LIT supports only floating-point data type modifiers.

Section 2.X.8.Z, LRP: Linear Interpolation

The LRP instruction performs a component-wise linear interpolation between the second and third operands using the first operand as the blend factor.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
  result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
  result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
  result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

LRP supports only floating-point data type modifiers.

Section 2.X.8.Z, MAD: Multiply and Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

The multiplication and addition operations in this instruction are subject to the same rules as described for the MUL and ADD instructions.

MAD supports all three data type modifiers.

Section 2.X.8.Z, MAX: Maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp0.x : tmp1.x;
  result.y = (tmp0.y > tmp1.y) ? tmp0.y : tmp1.y;
  result.z = (tmp0.z > tmp1.z) ? tmp0.z : tmp1.z;
  result.w = (tmp0.w > tmp1.w) ? tmp0.w : tmp1.w;
```

MAX supports all three data type modifiers.

Section 2.X.8.Z, MIN: Minimum

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp1.x : tmp0.x;
  result.y = (tmp0.y > tmp1.y) ? tmp1.y : tmp0.y;
  result.z = (tmp0.z > tmp1.z) ? tmp1.z : tmp0.z;
  result.w = (tmp0.w > tmp1.w) ? tmp1.w : tmp0.w;
```

MIN supports all three data type modifiers.

Section 2.X.8.Z, MOD: Modulus

The MOD instruction performs a component-wise modulus operation on the first vector operand by the second scalar operand to produce a 4-component result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = tmp0.x % tmp1;
  result.y = tmp0.y % tmp1;
  result.z = tmp0.z % tmp1;
  result.w = tmp0.w % tmp1;
```

MOD supports both signed and unsigned integer data type modifiers. If no data type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, MOV: Move

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

MOV supports all three data type modifiers.

Section 2.X.8.Z, MUL: Multiply

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

MUL supports all three data type modifiers. The MUL instruction additionally supports three special modifiers.

The "S24" and "U24" modifiers specify "fast" signed or unsigned integer multiplies of 24-bit quantities, respectively. The results of such

multiplies are undefined if either operand is outside the range [-2^23,+2^23-1] for S24 or [0,2^24-1] for U24. If "S24" or "U24" is specified, the data type is implied and normal data type modifiers may not be provided.

The "HI" modifier specifies a 32-bit integer multiply that returns the 32 most significant bits of the 64-bit product. Integer multiplies without the "HI" modifier normally return the least significant bits of the product. If "HI" is specified, either of the "S" or "U" integer data type modifiers must also be specified.

Note that if condition code updates are performed on integer multiplies, the overflow or carry flags are always cleared, even if the product overflowed. If it is necessary to determine if the results of an integer multiply overflowed, the MUL.HI instruction may be used.

Section 2.X.8.Z, NOT: Bitwise Not

The NOT instruction performs a component-wise bitwise NOT operation on the source vector to produce a result vector.

```
tmp = VectorLoad(op0);
tmp.x = ~tmp.x;
tmp.y = ~tmp.y;
tmp.z = ~tmp.z;
tmp.w = ~tmp.w;
```

NOT supports only integer data type modifiers. If no type modifier is specified, the operand and the result are treated as signed integers.

Section 2.X.8.Z, NRM: Normalize 3-Component Vector

The NRM instruction normalizes the vector given by the x, y, and z components of the vector operand to produce the x, y, and z components of the result vector. The w component of the result is undefined.

```
tmp = VectorLoad(op0);
scale = ApproxRSQ(tmp.x * tmp.x + tmp.y * tmp.y + tmp.z * tmp.z);
result.x = tmp.x * scale;
result.y = tmp.y * scale;
result.z = tmp.z * scale;
result.w = undefined;
```

NRM supports only floating-point data type modifiers.

Section 2.X.8.Z, OR: Bitwise Or

The OR instruction performs a bitwise OR operation on the components of the two source vectors to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x | tmp1.x;
result.y = tmp0.y | tmp1.y;
result.z = tmp0.z | tmp1.z;
result.w = tmp0.w | tmp1.w;
```

OR supports only integer data type modifiers. If no type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, PK2H: Pack Two 16-bit Floats

The PK2H instruction converts the "x" and "y" components of the single floating-point vector operand into 16-bit floating-point format, packs the bit representation of these two floats into a 32-bit unsigned integer, and replicates that value to all four components of the result vector. The PK2H instruction can be reversed by the UP2H instruction below.

```
tmp0 = VectorLoad(op0);
/* result obtained by combining raw bits of tmp0.x, tmp0.y */
result.x = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
result.y = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
result.z = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
result.w = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
```

PK2H supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer results, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions.

A program will fail to load if it contains a PK2H instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, PK2US: Pack Two Floats as Unsigned 16-bit

The PK2US instruction converts the "x" and "y" components of the single floating-point vector operand into a packed pair of 16-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the two converted components are packed into a 32-bit unsigned integer, and that value is replicated to all four components of the result vector. The PK2US instruction can be reversed by the UP2US instruction below.

```
tmp0 = VectorLoad(op0);
if (tmp0.x < 0.0) tmp0.x = 0.0;
if (tmp0.x > 1.0) tmp0.x = 1.0;
if (tmp0.y < 0.0) tmp0.y = 0.0;
if (tmp0.y > 1.0) tmp0.y = 1.0;
us.x = round(65535.0 * tmp0.x);  /* us is a ushort vector */
us.y = round(65535.0 * tmp0.y);
/* result obtained by combining raw bits of us. */
result.x = ((us.x) | (us.y << 16));
result.y = ((us.x) | (us.y << 16));
result.z = ((us.x) | (us.y << 16));
result.w = ((us.x) | (us.y << 16));
```

PK2US supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer result variables, the

bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions.

A program will fail to load if it contains a PK2S instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, PK4B: Pack Four Floats as Signed 8-bit

The PK4B instruction converts the four components of the single floating-point vector operand into 8-bit signed quantities. The signed quantities are represented in a bit pattern where all '0' bits corresponds to -128/127 and all '1' bits corresponds to +127/127. The bit representations of the four converted components are packed into a 32-bit unsigned integer, and that value is replicated to all four components of the result vector. The PK4B instruction can be reversed by the UP4B instruction below.

```
tmp0 = VectorLoad(op0);
if (tmp0.x < -128/127) tmp0.x = -128/127;
if (tmp0.y < -128/127) tmp0.y = -128/127;
if (tmp0.z < -128/127) tmp0.z = -128/127;
if (tmp0.w < -128/127) tmp0.w = -128/127;
if (tmp0.x > +127/127) tmp0.x = +127/127;
if (tmp0.y > +127/127) tmp0.y = +127/127;
if (tmp0.z > +127/127) tmp0.z = +127/127;
if (tmp0.w > +127/127) tmp0.w = +127/127;
ub.x = round(127.0 * tmp0.x + 128.0);  /* ub is a ubyte vector */
ub.y = round(127.0 * tmp0.y + 128.0);
ub.z = round(127.0 * tmp0.z + 128.0);
ub.w = round(127.0 * tmp0.w + 128.0);
/* result obtained by combining raw bits of ub. */
result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

PK4B supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer result variables, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions. A program will fail to load if it contains a PK4B instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, PK4UB: Pack Four Floats as Unsigned 8-bit

The PK4UB instruction converts the four components of the single floating-point vector operand into a packed grouping of 8-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the four converted components are packed into a 32-bit unsigned integer, and that value is replicated to all four components of the result vector. The PK4UB instruction can be reversed by the UP4UB

instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  if (tmp0.z < 0.0) tmp0.z = 0.0;
  if (tmp0.z > 1.0) tmp0.z = 1.0;
  if (tmp0.w < 0.0) tmp0.w = 0.0;
  if (tmp0.w > 1.0) tmp0.w = 1.0;
  ub.x = round(255.0 * tmp0.x);  /* ub is a ubyte vector */
  ub.y = round(255.0 * tmp0.y);
  ub.z = round(255.0 * tmp0.z);
  ub.w = round(255.0 * tmp0.w);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

PK4UB supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. For integer result variables, the bits can be interpreted as described above. For floating-point result variables, the packed results do not constitute a meaningful floating-point variable and should only be used to feed future unpack instructions.

A program will fail to load if it contains a PK4UB instruction that writes its results to a variable declared as "SHORT".

Section 2.X.8.Z, POW: Exponentiate

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function may be implemented using the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

Note that a logarithm may be involved even for cases where the exponent is an integer. This means that it may not be possible to exponentiate correctly with a negative base. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

POW supports only floating-point data type modifiers.

Section 2.X.8.Z, RCC: Reciprocal (Clamped)

The RCC instruction approximates the reciprocal of the scalar operand, clamps the result to one of two ranges, and replicates the clamped result to all four components of the result vector.

If the approximated reciprocal is greater than 0.0, the result is clamped to the range [2^-64, 2^+64]. If the approximate reciprocal is not greater than zero, the result is clamped to the range [-2^+64, -2^-64].

```
  tmp = ScalarLoad(op0);
  result.x = ClampApproxReciprocal(tmp);
  result.y = ClampApproxReciprocal(tmp);
  result.z = ClampApproxReciprocal(tmp);
  result.w = ClampApproxReciprocal(tmp);
```

RCC supports only floating-point data type modifiers.

Section 2.X.8.Z, RCP: Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

RCP supports only floating-point data type modifiers.

Section 2.X.8.Z, REP: Start of Repeat Block

The REP instruction begins a REP/ENDREP block. The REP instruction supports an optional operand whose x component specifies the initial value for the loop count. The loop count indicates the number of times the instructions between the REP and corresponding ENDREP instruction will be executed. If the initial value of the loop count is not positive, the entire block is skipped and execution continues at the instruction following the corresponding ENDREP instruction. If the loop count is specified as a floating-point value, it is converted to the largest integer less than or equal to the specified value (i.e., taking its floor).

If no operand is provided to REP, the loop count is ignored and the corresponding ENDREP instruction unconditionally transfers control to the instruction immediately following the REP instruction. The only way to exit such a loop is with the BRK instruction. To prevent obvious infinite loops, a program that includes a REP/ENDREP block with no loop count will fail to compile unless it contains either a BRK instruction at the current nesting level or a RET instruction at any nesting level.

Implementations may have a limited ability to nest REP/ENDREP blocks. If the number of REP/ENDREP blocks nested inside each other is

```
MAX_PROGRAM_LOOP_DEPTH_NV or higher, a program will fail to compile.

  // Set up loop information for the new nesting level.
  tmp = VectorLoad(op0);
  LoopCount = floor(tmp.x);
  if (LoopCount <= 0) {
    continue execution at the corresponding ENDREP;
  }

REP supports all three data type modifiers.  The single operand is
interpreted according to the data type modifier.

(Note:  Unlike the NV_fragment_program2 extension, REP blocks in this
extension support fully general looping; the specified loop count can be
computed in the program itself.  Additionally, there is no run-time limit
on the maximum overall depth of REP/ENDREP nesting.  As long as each
individual subroutine of the program obeys the static nesting limits,
there will be no run-time errors in the program.  With the
NV_fragment_program2 extension, a program could terminate abnormally if it
called a subroutine inside a deeply nested set of REP/ENDREP blocks and
the called subroutine also contained deeply nested REP/ENDREP blocks.
Such an error could occur even if neither subroutine exceeded static
limits.)
```

Section 2.X.8.Z, RET: Subroutine Return

```
The RET instruction conditionally returns from a subroutine initiated by a
CAL instruction by popping an instruction reference off the top of the
call stack and transferring control to the referenced instruction.  The
following pseudocode describes the operation of the instruction:

  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate program
    } else {
      callStackDepth--;
      instruction = callStack[callStackDepth];
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }

In the pseudocode, <callStackDepth> is the depth of the call stack,
<callStack> is an array holding the call stack, and <instruction> is a
reference to an instruction previously pushed onto the call stack.

If the call stack is empty when RET executes, the program terminates
normally.
```

Section 2.X.8.Z, RFL: Reflection Vector

```
The RFL instruction computes the reflection of the second vector operand
(the "direction" vector) about the vector specified by the first vector
operand (the "axis" vector).  Both operands are treated as 3D vectors (the
```

w components are ignored). The result vector is another 3D vector (the "reflected direction" vector). The length of the result vector, ignoring rounding errors, should equal that of the second operand.

```
  axis = VectorLoad(op0);
  direction = VectorLoad(op1);
  tmp.w = (axis.x * axis.x + axis.y * axis.y + axis.z * axis.z);
  tmp.x = (axis.x * direction.x + axis.y * direction.y +
           axis.z * direction.z);
  tmp.x = 2.0 * tmp.x;
  tmp.x = tmp.x / tmp.w;
  result.x = tmp.x * axis.x - direction.x;
  result.y = tmp.x * axis.y - direction.y;
  result.z = tmp.x * axis.z - direction.z;
```

RFL supports only floating-point data type modifiers.

Section 2.X.8.Z, ROUND: Round to Nearest Integer

The ROUND instruction loads a single vector operand and performs a component-wise round operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  result.x = round(tmp.x);
  result.y = round(tmp.y);
  result.z = round(tmp.z);
  result.w = round(tmp.w);
```

The round operation returns the nearest integer to the operand. If the fractional portion of the operand is 0.5, round() selects the nearest even integer. For example round(-1.7) = -2.0, round(+1.0) = +1.0, and round(+3.7) = +4.0.

ROUND supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, RSQ: Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

If the operand is less than or equal to zero, the results of the instruction are undefined.

RSQ supports only floating-point data type modifiers.

Note that this instruction differs from the RSQ instruction in ARB_vertex_program in that it does not implicitly take the absolute value of its operand. The |abs| operator can be used to achieve equivalent semantics.

Section 2.X.8.Z, SAD: Sum of Absolute Differences

The SAD instruction performs a component-wise difference of the first two integer operands (subtracting the second from the first), and then does a component-wise add of the absolute value of the difference to the third unsigned integer operand to yield an unsigned integer result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = abs(tmp0.x - tmp1.x) + tmp2.x;
result.y = abs(tmp0.y - tmp1.y) + tmp2.y;
result.z = abs(tmp0.z - tmp1.z) + tmp2.z;
result.w = abs(tmp0.w - tmp1.w) + tmp2.w;
```

SAD supports signed and unsigned integer data type modifiers. The first two operands are interpreted according to the data type modifier. The third operand and the result are always unsigned integers.

Section 2.X.8.Z, SCS: Sine/Cosine without Reduction

The SCS instruction approximates the trigonometric sine and cosine of the angle specified by the scalar operand and places the cosine in the x component and the sine in the y component of the result vector. The z and w components of the result vector are undefined. The angle is specified in radians and must be in the range [-PI,PI].

```
tmp = ScalarLoad(op0);
result.x = ApproxCosine(tmp);
result.y = ApproxSine(tmp);
```

If the scalar operand is not in the range [-PI,PI], the result vector is undefined.

SCS supports only floating-point data type modifiers.

Section 2.X.8.Z, SEQ: Set on Equal

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is equal to that of the second, and a FALSE value otherwise.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x == tmp1.x) ? TRUE : FALSE;
result.y = (tmp0.y == tmp1.y) ? TRUE : FALSE;
result.z = (tmp0.z == tmp1.z) ? TRUE : FALSE;
result.w = (tmp0.w == tmp1.w) ? TRUE : FALSE;
```

SEQ supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data

types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SFL: Set on False

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to a FALSE value (described below).

```
result.x = FALSE;
result.y = FALSE;
result.z = FALSE;
result.w = FALSE;
```

SFL supports all data type modifiers. For floating-point data types, the FALSE value is 0.0. For signed and unsigned integer data types, the FALSE value is zero.

Section 2.X.8.Z, SGE: Set on Greater Than or Equal

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is greater than or equal to that of the second, and a FALSE value otherwise.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x >= tmp1.x) ? TRUE : FALSE;
result.y = (tmp0.y >= tmp1.y) ? TRUE : FALSE;
result.z = (tmp0.z >= tmp1.z) ? TRUE : FALSE;
result.w = (tmp0.w >= tmp1.w) ? TRUE : FALSE;
```

SGE supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SGT: Set on Greater Than

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is greater than that of the second, and a FALSE value otherwise.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x > tmp1.x) ? TRUE : FALSE;
result.y = (tmp0.y > tmp1.y) ? TRUE : FALSE;
result.z = (tmp0.z > tmp1.z) ? TRUE : FALSE;
result.w = (tmp0.w > tmp1.w) ? TRUE : FALSE;
```

SGT supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits

are ones) and the FALSE value is zero.

Section 2.X.8.Z, SHL: Shift Left

The SHL instruction performs a component-wise left shift of the bits of the first operand by the value of the second scalar operand to produce a result vector. The bits vacated during the shift operation are filled with zeroes.

```
tmp0 = VectorLoad(op0);
tmp1 = ScalarLoad(op1);
result.x = tmp0.x << tmp1;
result.y = tmp0.y << tmp1;
result.z = tmp0.z << tmp1;
result.w = tmp0.w << tmp1;
```

The results of a shift operation ("<<") are undefined if the value of the second operand is negative, or greater than or equal to the number of bits in the first operand.

SHL supports both signed and unsigned integer data type modifiers. If no modifier is provided, the operands and the result are treated as signed integers.

Section 2.X.8.Z, SHR: Shift Right

The SHR instruction performs a component-wise right shift of the bits of the first operand by the value of the second scalar operand to produce a result vector. The bits vacated during shift operation are filled with zeros if the operand is non-negative and ones otherwise.

```
tmp0 = VectorLoad(op0);
tmp1 = ScalarLoad(op1);
result.x = tmp0.x >> tmp1;
result.y = tmp0.y >> tmp1;
result.z = tmp0.z >> tmp1;
result.w = tmp0.w >> tmp1;
```

The results of a shift operation (">>") are undefined if the value of the second operand is negative, or greater than or equal to the number of bits in the first operand.

SHR supports both signed and unsigned integer data type modifiers. If no modifiers are provided, the operands and the result are treated as signed integers.

Section 2.X.8.Z, SIN: Sine with Reduction to [-PI,PI]

The SIN instruction approximates the trigonometric sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

SIN supports only floating-point data type modifiers.

Section 2.X.8.Z, SLE: Set on Less Than or Equal

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is less than or equal to that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y <= tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z <= tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w <= tmp1.w) ? TRUE : FALSE;
```

SLE supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SLT: Set on Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is less than that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y < tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z < tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w < tmp1.w) ? TRUE : FALSE;
```

SLT supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector returns a TRUE value (described below) if the corresponding component of the first operand is less than that of the second, and a FALSE value otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? TRUE : FALSE;
  result.y = (tmp0.y != tmp1.y) ? TRUE : FALSE;
  result.z = (tmp0.z != tmp1.z) ? TRUE : FALSE;
  result.w = (tmp0.w != tmp1.w) ? TRUE : FALSE;
```

SNE supports all data type modifiers. For floating-point data types, the TRUE value is 1.0 and the FALSE value is 0.0. For signed integer data types, the TRUE value is -1 and the FALSE value is 0. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones) and the FALSE value is zero.

Section 2.X.8.Z, SSG: Set Sign

The SSG instruction generates a result vector containing the signs of each component of the single vector operand. Each component of the result vector is 1.0 if the corresponding component of the operand is greater than zero, 0.0 if the corresponding component of the operand is equal to zero, and -1.0 if the corresponding component of the operand is less than zero.

```
  tmp = VectorLoad(op0);
  result.x = SetSign(tmp.x);
  result.y = SetSign(tmp.y);
  result.z = SetSign(tmp.z);
  result.w = SetSign(tmp.w);
```

SSG supports only floating-point data type modifiers.

Section 2.X.8.Z, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to a TRUE value (described below).

```
  result.x = TRUE;
  result.y = TRUE;
  result.z = TRUE;
  result.w = TRUE;
```

STR supports all data type modifiers. For floating-point data types, the TRUE value is 1.0. For signed integer data types, the TRUE value is -1. For unsigned integer data types, the TRUE value is the maximum integer value (all bits are ones).

Section 2.X.8.Z, SUB: Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

SUB supports all three data type modifiers.

Section 2.X.8.Z, SWZ: Extended Swizzle

The SWZ instruction loads the single vector operand, and performs a swizzle operation more powerful than that provided for loading normal vector operands to yield an instruction vector.

After the operand is loaded, the "x", "y", "z", and "w" components of the result vector are selected by the first, second, third, and fourth matches of the <extSwizComp> pattern in the <extendedSwizzle> rule.

A result component can be selected from any of the four components of the operand or the constants 0.0 and 1.0. The result component can also be optionally negated. The following pseudocode describes the component selection method. "operand" refers to the vector operand, "select" is an enumerant where the values ZERO, ONE, X, Y, Z, and W correspond to the <extSwizSel> rule matching "0", "1", "x", "y", "z", and "w", respectively. "negate" is TRUE if and only if the <optionalSign> rule in <extSwizComp> matches "-".

```
  float ExtSwizComponent(floatVec operand, enum select, boolean negate)
  {
      float result;
      switch (select) {
        case ZERO:  result = 0.0; break;
        case ONE:   result = 1.0; break;
        case X:     result = operand.x; break;
        case Y:     result = operand.y; break;
        case Z:     result = operand.z; break;
        case W:     result = operand.w; break;
      }
      if (negate) {
        result = -result;
      }
      return result;
  }
```

The entire extended swizzle operation is then defined using the following pseudocode:

```
  tmp = VectorLoad(op0);
  result.x = ExtSwizComponent(tmp, xSelect, xNegate);
  result.y = ExtSwizComponent(tmp, ySelect, yNegate);
  result.z = ExtSwizComponent(tmp, zSelect, zNegate);
  result.w = ExtSwizComponent(tmp, wSelect, wNegate);
```

"xSelect", "xNegate", "ySelect", "yNegate", "zSelect", "zNegate", "wSelect", and "wNegate" correspond to the "select" and "negate" values above for the four <extSwizComp> matches.

Since this instruction allows for component selection and negation for each individual component, the grammar does not allow the use of the normal swizzle and negation operations allowed for vector operands in other instructions.

SWZ supports only floating-point data type modifiers.

Section 2.X.8.Z, TEX: Texture Sample

The TEX instruction takes the four components of a single floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives and the level of detail are computed automatically.

```
  tmp = VectorLoad(op0);
  ddx = ComputePartialsX(tmp);
  ddy = ComputePartialsY(tmp);
  lambda = ComputeLOD(ddx, ddy);
  result = TextureSample(tmp, lambda, ddx, ddy, texelOffset);
```

TEX supports all three data type modifiers. The single operand is always treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TRUNC: Truncate (Round Toward Zero)

The TRUNC instruction loads a single vector operand and performs a component-wise truncate operation to generate a result vector.

```
  tmp = VectorLoad(op0);
  result.x = trunc(tmp.x);
  result.y = trunc(tmp.y);
  result.z = trunc(tmp.z);
  result.w = trunc(tmp.w);
```

The truncate operation returns the nearest integer to zero smaller in magnitude than the operand. For example trunc(-1.7) = -1.0, trunc(+1.0) = +1.0, and trunc(+3.7) = +3.0.

TRUNC supports all three data type modifiers. The single operand is always treated as a floating-point value, but the result is written as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier. If a value is not exactly

representable using the data type of the result (e.g., an overflow or writing a negative value to an unsigned integer), the result is undefined.

Section 2.X.8.Z, TXB: Texture Sample with Bias

The TXB instruction takes the four components of a single floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives and the level of detail are computed automatically, but the fourth component of the source vector is added to the computed LOD prior to sampling.

```
tmp = VectorLoad(op0);
ddx = ComputePartialsX(tmp);
ddy = ComputePartialsY(tmp);
lambda = ComputeLOD(ddx, ddy);
result = TextureSample(tmp, lambda + tmp.w, ddx, ddy, texelOffset);
```

The single source vector in the TXB instruction does not have enough coordinates to specify a lookup into a two-dimensional array texture or cube map texture with both an LOD bias and an explicit reference value for depth comparison. A program will fail to load if it contains a TXB instruction with a target of SHADOWCUBE or SHADOWARRAY2D.

TXB supports all three data type modifiers. The single operand is always treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXD: Texture Sample with Partials

The TXD instruction takes the four components of the first floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. The partial derivatives of the texture coordinates with respect to X and Y are specified by the second and third floating-point source vectors. The level of detail is computed automatically using the provided partial derivatives.

Note that for cube map texture targets, the provided partial derivatives are in the coordinate system used before texture coordinates are projected onto the appropriate cube face. The partial derivatives of the post-projection texture coordinates, which are used for level-of-detail and anisotropic filtering calculations, are derived from the original coordinates and partial derivatives in an implementation-dependent manner.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
lambda = ComputeLOD(tmp1, tmp2);
result = TextureSample(tmp0, lambda, tmp1, tmp2, texelOffset);
```

TXD supports all three data type modifiers. All three operands are always treated as floating-point vectors; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXF: Texel Fetch

The TXF instruction takes the four components of a single signed integer source vector and performs a single texel fetch as described in Section 2.X.4.4. The first three components provide the <i>, <j>, and <k> values for the texel fetch, and the fourth component is used to determine the LOD to access. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives are irrelevant for single texel fetches.

```
  tmp = VectorLoad(op0);
  result = TexelFetch(tmp, texelOffset);
```

TXF supports all three data type modifiers. The single vector operand is treated as a signed integer vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXL: Texture Sample with LOD

The TXL instruction takes the four components of a single floating-point source vector and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. The level of detail is taken from the fourth component of the source vector.

Partial derivatives are not computed by the TXL instruction and anisotropic filtering is not performed.

```
  tmp = VectorLoad(op0);
  ddx = (0,0,0);
  ddy = (0,0,0);
  result = TextureSample(tmp, tmp.w, ddx, ddy, texelOffset);
```

The single source vector in the TXL instruction does not have enough coordinates to specify a lookup into a 2D array or cube map texture with both an explicit LOD and a reference value for depth comparison. A program will fail to load if it contains a TXL instruction with a target of SHADOWCUBE or SHADOWARRAY2D.

TXL supports all three data type modifiers. The single vector operand is treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXP: Texture Sample with Projection

The TXP instruction divides the first three components of its single floating-point source vector by its fourth component, maps the results to s, t, and r, and performs a filtered texture access as described in Section 2.X.4.4. The returned (R,G,B,A) value is written to the floating-point result vector. Partial derivatives and the level of detail are computed automatically.

```
tmp0 = VectorLoad(op0);
tmp0.x = tmp0.x / tmp0.w;
tmp0.y = tmp0.y / tmp0.w;
tmp0.z = tmp0.z / tmp0.w;
ddx = ComputePartialsX(tmp);
ddy = ComputePartialsY(tmp);
lambda = ComputeLOD(ddx, ddy);
result = TextureSample(tmp, lambda, ddx, ddy, texelOffset);
```

The single source vector in the TXP instruction does not have enough coordinates to specify a lookup into a 2D array or cube map texture with both a Q coordinate and an explicit reference value for depth comparison. A program will fail to load if it contains a TXP instruction with a target of SHADOWCUBE or SHADOWARRAY2D.

TXP supports all three data type modifiers. The single vector operand is treated as a floating-point vector; the results are interpreted according to the data type modifier.

Section 2.X.8.Z, TXQ: Texture Size Query

The TXQ instruction takes the first component of the single integer vector operand, adds the number of the base level of the specified texture to determine a texture image level, and returns an integer result vector containing the size of the image at that level of the texture.

For one-dimensional and one-dimensional array textures, the "x" component of the result vector is filled with the width of the image(s). For two-dimensional, rectangle, cube map, and two-dimensional array textures, the "x" and "y" components are filled with the width and height of the image(s). For three-dimensional textures, the "x", "y", and "z" components are filled with the width, height, and depth of the image. Additionally, the number of layers in an array texture is returned in the "y" component of the result for one-dimensional array textures or the "z" component for two-dimensional array textures. All other components of the result vector is undefined. For the purposes of this instruction, the width, height, and depth of a texture do NOT include any border.

```
tmp0 = VectorLoad(op0);
tmp0.x = tmp0.x + texture[op1].target[op2].base_level;
result.x = texture[op1].target[op2].level[tmp0.x].width;
result.y = texture[op1].target[op2].level[tmp0.x].height;
result.z = texture[op1].target[op2].level[tmp0.x].depth;
```

If the level computed by adding the operand to the base level of the texture is less than the base level number or greater than the maximum level number, the results are undefined.

TXQ supports no data type modifiers; the scalar operand and the result vector are both interpreted as signed integers.

Section 2.X.8.Z, UP2H: Unpack Two 16-bit Floats

The UP2H instruction unpacks two 16-bit floats stored together in a 32-bit scalar operand. The first 16-bit float (stored in the 16 least significant bits) is written into the "x" and "z" components of the result vector; the second is written into the "y" and "w" components of the result vector.

This operation undoes the type conversion and packing performed by the PK2H instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (fp16) (RawBits(tmp) & 0xFFFF);
  result.y = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
  result.z = (fp16) (RawBits(tmp) & 0xFFFF);
  result.w = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
```

UP2H supports all three data type modifiers. The single operand is read as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier; the 32 least significant bits of the encoding are used for unpacking. For floating-point operand variables, it is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A program will fail to load if it contains a UP2H instruction whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, UP2US: Unpack Two Unsigned 16-bit Integers

The UP2US instruction unpacks two 16-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" and "z" components of the result vector are obtained from the 16 least significant bits of the operand; the "y" and "w" components are obtained from the 16 most significant bits.

This operation undoes the type conversion and packing performed by the PK2US instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.y = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
  result.z = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.w = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
```

UP2US supports all three data type modifiers. The single operand is read as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier; the 32 least significant bits of the encoding are used for unpacking. For floating-point operand variables, it is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A GPU program will fail to load if it contains a UP2S instruction whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, UP4B: Unpack Four Signed 8-bit Integers

The UP4B instruction unpacks four 8-bit signed values packed together in a 32-bit scalar operand. The signed quantities are encoded where a bit pattern of all '0' bits corresponds to -128/127 and a pattern of all '1' bits corresponds to +127/127. The "x" component of the result vector is the converted value corresponding to the 8 least significant bits of the operand; the "w" component corresponds to the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4B instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (((RawBits(tmp) >> 0) & 0xFF) - 128) / 127.0;
  result.y = (((RawBits(tmp) >> 8) & 0xFF) - 128) / 127.0;
  result.z = (((RawBits(tmp) >> 16) & 0xFF) - 128) / 127.0;
  result.w = (((RawBits(tmp) >> 24) & 0xFF) - 128) / 127.0;
```

UP2B supports all three data type modifiers. The single operand is read as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier; the 32 least significant bits of the encoding are used for unpacking. For floating-point operand variables, it is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A program will fail to load if it contains a UP4B instruction whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, UP4UB: Unpack Four Unsigned 8-bit Integers

The UP4UB instruction unpacks four 8-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" component of the result vector is obtained from the 8 least significant bits of the operand; the "w" component is obtained from the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4UB instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFF) / 255.0;
  result.y = ((RawBits(tmp) >> 8)  & 0xFF) / 255.0;
  result.z = ((RawBits(tmp) >> 16) & 0xFF) / 255.0;
  result.w = ((RawBits(tmp) >> 24) & 0xFF) / 255.0;
```

UP4UB supports all three data type modifiers. The single operand is read as a floating-point value, a signed integer, or an unsigned integer, as specified by the data type modifier; the 32 least significant bits of the encoding are used for unpacking. For floating-point operand variables, it

is expected (but not required) that the operand was produced by a previous pack instruction. The result is always written as a floating-point vector.

A program will fail to load if it contains a UP4UB instruction whose operand is a variable declared as "SHORT".

Section 2.X.8.Z, X2D: 2D Coordinate Transformation

The X2D instruction multiplies the 2D offset vector specified by the "x" and "y" components of the second vector operand by the 2x2 matrix specified by the four components of the third vector operand, and adds the transformed offset vector to the 2D vector specified by the "x" and "y" components of the first vector operand. The first component of the sum is written to the "x" and "z" components of the result; the second component is written to the "y" and "w" components of the result.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.y = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
  result.z = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.w = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
```

X2D supports only floating-point data type modifiers.

Section 2.X.8.Z, XOR: Exclusive Or

The XOR instruction performs a bitwise XOR operation on the components of the two source vectors to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x ^ tmp1.x;
  result.y = tmp0.y ^ tmp1.y;
  result.z = tmp0.z ^ tmp1.z;
  result.w = tmp0.w ^ tmp1.w;
```

XOR supports only integer data type modifiers. If no type modifier is specified, both operands and the result are treated as signed integers.

Section 2.X.8.Z, XPD: Cross Product

The XPD instruction computes the cross product using the first three components of its two vector operands to generate the x, y, and z components of the result vector. The w component of the result vector is undefined.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.y * tmp1.z - tmp0.z * tmp1.y;
  result.y = tmp0.z * tmp1.x - tmp0.x * tmp1.z;
  result.z = tmp0.x * tmp1.y - tmp0.y * tmp1.x;
```

XPD supports only floating-point data type modifiers.

Additions to Chapter 3 of the OpenGL 1.5 Specification (Rasterization)

Modify Section 3.8.1, Texture Image Specification, p. 150

(modify 4th paragraph, p. 151 -- add cubemaps to the list of texture targets that can be used with DEPTH_COMPONENT textures) Textures with a base internal format of DEPTH_COMPONENT are supported by texture image specification commands only if <target> is TEXTURE_1D, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_RECTANGLE_ARB, TEXTURE_1D_ARRAY_EXT, TEXTURE_2D_ARRAY_EXT, PROXY_TEXTURE_1D PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, PROXY_TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_1D_ARRAY_EXT, or PROXY_TEXTURE_2D_ARRAY_EXT. Using this format in conjunction with any other target will result in an INVALID_OPERATION error.

Delete Section 3.8.7, Texture Wrap Modes. (The language in this section is folded into updates to the following section, and is no longer needed here.)

Modify Section 3.8.8, Texture Minification:

(replace the last paragraph, p. 171): Let s(x,y) be the function that associates an s texture coordinate with each set of window coordinates (x,y) that lie within a primitive; define t(x,y) and r(x,y) analogously. Let

```
  u(x,y) = w_t * s(x,y) + offsetu_shader,
  v(x,y) = h_t * t(x,y) + offsetv_shader,
  w(x,y) = d_t * r(x,y) + offsetw_shader, and
```

where w_t, h_t, and d_t are as defined by equations 3.15, 3.16, and 3.17 with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is level_base. (offsetu_shader, offsetv_shader, offsetw_shader) is the texel offset specified in the vertex, geometry, or fragment program instruction used to perform the access. For fixed-function texture accesses, all three shader offsets are taken to be zero. For a one-dimensional texture, define v(x,y) == 0 and w(x,y) === 0; for two-dimensional textures, define w(x,y) == 0.

(start a new paragraph with "For a polygon, rho is given at a fragment with window coordinates...", and then continue with the original spec text.)

(replace text starting with the last paragraph on p. 172, continuing to the end of p. 174)

The (u,v,w) coordinates are then modified according the texture wrap modes, as specified in Table X.19, to generate a new set of coordinates (u',v',w').

TEXTURE_WRAP_S	Coordinate Transformation
CLAMP	u' = clamp(u, 0, w_t-0.5), if NEAREST filtering, clamp(u, 0, w_t), otherwise
CLAMP_TO_EDGE	u' = clamp(u, 0.5, w_t-0.5)
CLAMP_TO_BORDER	u' = clamp(u, -0.5, w_t+0.5)
REPEAT	u' = clamp(fmod(u, w_t), 0.5, w_t-0.5)
MIRROR_CLAMP_EXT	u' = clamp(fabs(u), 0.5, w_t-0.5), if NEAREST filtering, or = clamp(fabs(u), 0.5, w_t), otherwise
MIRROR_CLAMP_TO_EDGE_EXT	u' = clamp(fabs(u), 0.5, w_t-0.5)
MIRROR_CLAMP_TO_BORDER_EXT	u' = clamp(fabs(u), 0.5, w_t+0.5)
MIRRORED_REPEAT	u' = w_t - clamp(fabs(w_t - fmod(u, 2*w_t)), 0.5, w_t-0.5),

Table X.19: Texel coordinate wrap mode application. clamp(a,b,c) returns b if a<b, c if a>c, and a otherwise. fmod(a,b) returns a-b*floor(a/b), and fabs(a) returns the absolute value of a. For the v and w coordinates, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t, respectively, are used.

When lambda indicates minification, the value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When TEXTURE_MIN_FILTER is NEAREST, the texel in the image array of level level_base that is nearest (in Manhattan distance) to that specified by (s,t,r) is obtained. For a three-dimensional texture, the texel at location (i,j,k) becomes the texture value. For a two-dimensional texture, k is irrelevant, and the texel at location (i,j) becomes the texture value. For a one-dimensional texture, j and k are irrelevant, and the texel at location i becomes the texture value.

If the selected (i,j,k), (i,j), or i location refers to a border texel that satisfies any of the following conditions:

i < -b_s,
j < -b_s,
k < -b_s,
i >= w_l + b_s,
j >= h_l + b_s, or
j >= d_l + b_s,

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of TEXTURE_BORDER_COLOR 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 depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When TEXTURE_MIN_FILTER is LINEAR, a 2x2x2 cube of texels in the image array of level level_base is selected. Let:

```
i_0   = floor(u' - 0.5),
j_0   = floor(v' - 0.5),
k_0   = floor(w' - 0.5),
i_1   = i_0 + 1,
j_1   = j_0 + 1,
k_1   = k_0 + 1,
alpha = frac(u' - 0.5),
beta  = frac(v' - 0.5),
gamma = frac(w' - 0.5),
```

For a three-dimensional texture, the texture value tau is found as...

(replace last paragraph, p.174) For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify 2nd paragraph, p. 188, indicating that the Q texture coordinate is used for depth comparisons on cubemap textures)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture coordinate, clamped to the range [0, 1]. For texture lookups generated by a program instruction, let R be the reference value for depth comparisons provided in the instruction, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows:

Additions to Chapter 4 of the OpenGL 1.5 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.5 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.5 Specification (State and State Requests)

Modify Section 6.1.12 of the ARB_vertex_program specification.

(Add new integer program parameter queries, plus language that program environment or local parameter query results are undefined if the query specifies a data type incompatible with the data type of the parameter being queried.)

The commands

```
void GetProgramEnvParameterdvARB(enum target, uint index,
                                 double *params);
void GetProgramEnvParameterfvARB(enum target, uint index,
                                 float *params);
void GetProgramEnvParameterIivNV(enum target, uint index,
```

```
                                      int *params);
  void GetProgramEnvParameterIuivNV(enum target, uint index,
                                      uint *params);
```

obtain the current value for the program environment parameter numbered <index> for the given program target <target>, and places the information in the array <params>. The values returned are undefined if the data type of the components of the parameter is not compatible with the data type of <params>. Floating-point components are compatible with "double" or "float"; signed and unsigned integer components are compatible with "int" and "uint", respectively. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program environment parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program environment parameters for the program target.

...

The commands

```
  void GetProgramLocalParameterdvARB(enum target, uint index,
                                     double *params);
  void GetProgramLocalParameterfvARB(enum target, uint index,
                                     float *params);
  void GetProgramLocalParameterIivNV(enum target, uint index,
                                     int *params);
  void GetProgramLocalParameterIuivNV(enum target, uint index,
                                      uint *params);
```

obtain the current value for the program local parameter numbered <index> belonging to the program object currently bound to <target>, and places the information in the array <params>. The values returned are undefined if the data type of the components of the parameter is not compatible with the data type of <params>. Floating-point components are compatible with "double' or "float"; signed and unsigned integer components are compatible with "int" and "uint", respectively. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program local parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program local parameters for the program target.

...

The command

```
  void GetProgramivARB(enum target, enum pname, int *params);
```

obtains program state for the program target <target>, writing ...

(add new paragraphs describing the new supported queries)

If <pname> is PROGRAM_ATTRIB_COMPONENTS_NV or PROGRAM_RESULT_COMPONENTS_NV, GetProgramivARB returns a single integer holding the number of active attribute or result variable components, respectively, used by the program object currently bound to <target>.

If <pname> is MAX_PROGRAM_ATTRIB_COMPONENTS or MAX_PROGRAM_RESULT_COMPONENTS_NV, GetProgramivARB returns a single integer holding the maximum number of active attribute or result variable components, respectively, supported for programs of type <target>.

Additions to Appendix A of the OpenGL 1.5 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

The error INVALID_VALUE is generated by ProgramLocalParameter4fARB, ProgramLocalParameter4fvARB, ProgramLocalParameter4dARB, ProgramLocalParameter4dvARB, ProgramLocalParameterI4iNV, ProgramLocalParameterI4ivNV, ProgramLocalParameterI4uiNV, ProgramLocalParameterI4uivNV, GetProgramLocalParameter4fvARB, GetProgramLocalParameter4dvARB, GetProgramLocalParameterI4ivNV, and GetProgramLocalParameterI4uivNV if <index> is greater than or equal to the number of program local parameters supported by <target>.

The error INVALID_VALUE is generated by ProgramEnvParameter4fARB, ProgramEnvParameter4fvARB, ProgramEnvParameter4dARB, ProgramEnvParameter4dvARB, ProgramEnvParameterI4iNV, ProgramEnvParameterI4ivNV, ProgramEnvParameterI4uiNV, ProgramEnvParameterI4uivNV, GetProgramEnvParameter4fvARB, GetProgramEnvParameter4dvARB, GetProgramEnvParameterI4ivNV, and GetProgramEnvParameterI4uivNV if <index> is greater than or equal to the number of program environment parameters supported by <target>.

The error INVALID_VALUE is generated by ProgramLocalParameters4fvNV, ProgramLocalParametersI4ivNV, and ProgramLocalParametersI4uivNV if the sum of <index> and <count> is greater than the number of program local parameters supported by <target>.

The error INVALID_VALUE is generated by ProgramEnvParameters4fvNV, ProgramEnvParametersI4ivNV, and ProgramEnvParametersI4uivNV if the sum of <index> and <count> is greater than the number of program environment parameters supported by <target>.

Dependencies on NV_parameter_buffer_object

If NV_parameter_buffer_object is not supported, references to program parameter buffer variables and bindings should be removed.

Dependencies on ARB_texture_rectangle

If ARB_texture_rectangle is not supported, references to rectangle textures and the RECT and SHADOWRECT texture target identifiers should be removed.

Dependencies on EXT_gpu_program_parameters

If EXT_gpu_program_parameters is not supported, references to the Program{Local,Env}Parameters4fvNV commands, which set multiple program local or environment parameters in a single call, should be removed. These prototypes were included in this spec for completeness only.

Dependencies on EXT_texture_integer

If EXT_texture_integer is not supported, references to texture lookups returning integer values in Section 2.X.4.4 (Texture Access) should be removed, and all texture formats are considered to produce floating-point values.

Dependencies on EXT_texture_array

If EXT_texture_array is not supported, references to array textures in Section 2.X.4.4 (Texture Access) and elsewhere should be removed, as should all references to the "ARRAY1D", "ARRAY2D", "SHADOWARRAY1D", and "SHADOWARRAY2D" tokens.

Dependencies on EXT_texture_buffer_object

If EXT_texture_buffer_object is not supported, references to buffer textures in Section 2.X.4.4 (Texture Access) and elsewhere should be removed, as should all references to the "BUFFER" tokens.

Dependencies on NV_primitive_restart

If NV_primitive_restart is supported, index values causing a primitive restart are not considered as specifying an End command, followed by another Begin. Primitive restart is therefore not guaranteed to immediately update bindings for material properties changed inside a Begin/End. The spec language says they "are not guaranteed to update program parameter bindings until the following End command."

New State

Get Value	Type	Get Command	Initial Value	Description	Sec	Attrib
PROGRAM_ATTRIB_COMPONENTS_NV	Z+	GetProgramivARB	-	number of components used for attributes	6.1.12	-
PROGRAM_RESULT_COMPONENTS_NV	Z+	GetProgramivARB	-	number of components used for results	6.1.12	-

Table X.20. New Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attrib
MIN_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	-8	minimum texel offset allowed in lookup	2.x.4.4	-
MAX_PROGRAM_TEXEL_OFFSET_EXT	Z	GetIntegerv	+7	maximum texel offset allowed in lookup	2.x.4.4	-
MAX_PROGRAM_ATTRIB_COMPONENTS_NV	Z+	GetProgramivARB	(*)	maximum number of components allowed for attributes	6.1.12	-
MAX_PROGRAM_RESULT_COMPONENTS_NV	Z+	GetProgramivARB	(*)	maximum number of components allowed for results	6.1.12	-
MAX_PROGRAM_GENERIC_ATTRIBS_NV	Z+	GetProgramivARB	(*)	number of generic attribute vectors supported	6.1.12	-
MAX_PROGRAM_GENERIC_RESULTS_NV	Z+	GetProgramivARB	(*)	number of generic result vectors supported	6.1.12	-
MAX_PROGRAM_CALL_DEPTH_NV	Z+	GetProgramivARB	4	maximum program call stack depth	2.X.5	-
MAX_PROGRAM_IF_DEPTH_NV	Z+	GetProgramivARB	48	maximum program if nesting	2.X.5	-
MAX_PROGRAM_LOOP_DEPTH_NV	Z+	GetProgramivARB	4	maximum program loop nesting	2.X.5	-

Table X.21: New Implementation-Dependent Values Introduced by NV_gpu_program4. (*) means that the required minimum is program type-specific. There are separate limits for each program type.

Issues

(1) How does this extension differ from previous NV_vertex_program and NV_fragment_program extensions?

RESOLVED:

- This extension provides a uniform set of instructions and bindings. Unlike previous extensions, the set of instructions and bindings available is generally the same. The only exceptions are a small number of instructions and bindings that make sense for one specific program type.

- This extension supports integer data types and provides a full-fledged integer instruction set.

- This extension supports array variables of all types, including temporaries. Array variables can be accessed directly or indirectly (using integer temporaries as indices).

- This extension provides a uniform set of structured branching constructs (if tests, loops, subroutines) that fully support run-time condition testing. Previous versions of NV_vertex_program provided unstructured branching. Previous versions of NV_fragment_program provided structure branching constructs, but the support was more limited -- for example, looping constructs couldn't specify loop counts with values computed at run time.

- This extension supports geometry programs, which are described in more detail in the NV_geometry_program4 extension.

- This extension provides the ability to specify and use cubemap textures with a DEPTH_COMPONENT internal format. Shadow mapping is supported; the Q texture coordinate is used as the reference value for comparisons.

(2) Is this extension backward-compatible with previous NV_vertex_program and NV_fragment_program extensions? If not, what support has been removed?

RESOLVED: This extension is largely, but not completely, backward-compatible. Functionality removed includes:

- Unstructured branching: NV_vertex_program2 included a general branch instruction "BRA" that could be used to jump to an arbitrary instruction. The "CAL" instruction could "call" to an arbitrary instruction into code that was not necessarily structured as simple subroutine blocks. Arbitrary unstructured branching can be difficult to implement efficiently on highly parallel GPU architectures, while basic structured branching is not nearly as difficult.

  This extension retains the "CAL" instruction but treats each block of code between instruction labels as a separate subroutine. The "BRA" instruction and arbitrary branching has been removed. The structured branching constructs in this extension are sufficient to implement almost all of the looping/branching support in high-level languages ("goto" being the most obvious exception).

- Address registers: NV_vertex_program added the notion of address registers, which were effectively under-powered integer temporaries. The set of instructions used to manipulate address registers was severely limited. NV_vertex_program[23] extended the original scalars to vectors and added a few more instructions to manipulate address registers. Fragment programs had no address registers until NV_fragment_program2 added the loop counter, which was very similar in functionality to vertex program address registers, but even more limited. This extension adds true integer temporaries, which can accomplish everything old address registers could do, and much more. Address register support was removed to simplify the API.

- NV_fragment_program2 LOOP construct: NV_fragment_program2 added a LOOP instruction, which let you repeat a block of code <N> times, with a parallel loop counter that started at <A> and stepped by <B> on each iteration. This construct was signficantly limited in several ways -- the loop count had to be constant, and you could only access the innermost loop counter in a nested loop. This extension discards the support and retains the simpler "REP" construct to implement loops. If desired, a loop counter can be implemented by manipulating an integer temporary. The "BRK" instruction (conditional break) is retained, and a "CONT" instruction (conditional continue) is added. Additionally, the loop count need not be a constant.

- NV_vertex_program and ARB_vertex_program EXP and LOG instructions:
  NV_vertex_program provided EXP and LOG instructions that computed a rough approximation of 2^x or log_2(x) and provided some additional values that could help refine the approximation. Those opcodes were carried forward into ARB_vertex_program. Both ARB_vertex_program and NV_vertex_program2 provided EX2 and LG2 instructions that computed a better approximation. All fragment program extensions also provided EX2 and LG2, but did not bother to include EXP and LOG. On the hardware targeted by this extension, there is no advantage to using EXP and LOG, so these opcodes have been removed for simplicity.

- NV_vertex_program3 and NV_fragment_program2 provide the ability to do indirect addressing of inputs/outputs when using bindings in instructions -- for example:

```
MOV R0, vertex.attrib[A0.x+2];      # vertex
MOV result.texcoord[A0.y], R1;      # vertex
MOV R2, fragment.texcoord[A0.x];    # fragment
```

  This extension provides indexing capability, but using named array variables instead.

```
ATTRIB attribs[] = { vertex.attrib[2..5] };
MOV R0, attribs[A0.x];
OUTPUT outcoords[] = { result.texcoord[0..3] };
MOV outcoords[A0.y], R1;
ATTRIB texcoords[] = { fragment.texcoord[0..2] };
MOV R2, texcoords[A0.x];
```

  This approach makes the set of attribute and result bindings more regular. Additionally, it helps the assembler determine which vertex/fragment attributes are actually needed -- when the assembler sees constructs like "fragment.texcoord[A0.x]", it must treat *all* texture coordinates as live unless it can determine the range of values used for indexing. The named array variable approach explicitly identifies which attributes are needed when indexing is used.

Functionality altered includes:

- The RSQ instruction in the original NV_vertex_program and ARB_vertex_program extensions implicitly took the absolute value of their operand. Since the ARB extensions don't have numerics guarantees, computing the reciprocal square root of a negative value was not meaningful. To allow for the possibility of taking the reciprocal square root of a negative value (which should yield NaN -- "not a number"), the RSQ instruction in this instruction no longer implicitly takes the absolute value of its operand. Equivalent functionality can be achieved using the explicit |abs| absolute value operator on the operand to RSQ.

- The results of texture lookups accessing inconsistent textures are now undefined, instead of producing a fixed constant vector.

(3) What should this set of extensions be called?

RESOLVED: NV_gpu_program4, NV_vertex_program4, NV_fragment_program4, and NV_geometry_program4. Only NV_gpu_program4 will appear in the extension string; the other three specifications exist simply to define vertex, fragment, and geometry program-specific features.

The "gpu_program" name was chosen due to the common instruction set intended to run on GPUs. On previous chip generations, the vertex and fragment instruction sets were similar, but there were enough differences to package them separately.

The choice of "4" indicates that this is the fourth generation of programmable hardware from NVIDIA. The GeForce3 and GeForce4 series supported NV_vertex_program. The GeForce FX series supported NV_vertex_program2 and added fragment programmability with NV_fragment_program. Around this time, the OpenGL Architecture Review Board (ARB) approved ARB_vertex_program and ARB_fragment_program extensions, and NVIDIA added NV_vertex_program2_option and NV_fragment_program_option extensions exposing GeForce FX features using the ARB extensions' instruction set. The GeForce6 and GeForce7 series brought the NV_vertex_program3 and NV_fragment_program2 extensions, which extend the ARB extensions further. This extension adds geometry programs, and brings the "version number" for each of these extensions up to "4".

(4) This instruction adds integer data type support in programmable shaders that were previously float-centric. Should applications be able to pass integer values directly to the shaders, and if so, how does it work?

RESOLVED: The diagram at the bottom of this issue depicts data flows in the GL, as extended by this and related extensions.

This extension generalizes some state to be "typeless", instead of being strongly typed (and almost invariably floating-point) as in the core specification. We introduce a new set of functions to specify GL state as signed or unsigned integer values, instead of floating point values. These functions include:

* VertexAttribI*{i,ui}() -- Specify generic vertex attributes as integers. This extension does not create "integer" versions for fixed-function attribute functions (e.g., glColor, glTexCoord), which remain fully floating-point.

* Program{Env,Local}ParameterI*{i,ui}() -- Specify environment and local parameters as integers.

* TexImage*() with EXT_texture_integer internal formats -- Specify texture images as containing integer data whose values are not converted to floating-point values.

* EXT_parameter_buffer_object functions -- Bind (typeless) buffer object data stores for use as program parameters. These buffer objects can be loaded with either integer or floating-point data.

* EXT_texture_buffer_object functions -- Bind (typeless) buffer object data stores for use as textures. These buffer objects can be loaded with either integer or floating-point data.

Each type of program (using NV_gpu_program4 and related extension) can read attributes using any data type (float, signed integer, unsigned integer) and write result values used by subsequent stages using any data type.

Finally, there are several new places where integer data can be consumed by the GL:

* NV_transform_feedback -- Stream transformed vertex attribute components to a (typeless) buffer object. The transformed attributes can be written as signed or unsigned integers in vertex and geometry programs.

* EXT_texture_integer internal formats and framebuffer objects -- Provide support for rendering to integer texture formats, where final fragment values are treated as signed or unsigned integers, rather than floating-point values.

The diagram below represents a substantial portion of the GL pipeline. Each line connecting blocks represents an interface where data is "produced" from the GL state or by fixed-function or programmable pipeline stages and "consumed" by another pipeline stage. Each producer and consumer is labeled with a data type. For producers, the "(typeless)" designation generally means that the state and/or output can be written as floating-point values or as signed or unsigned integers. "(float)" means that the outputs are always written as floating-point. The same distinction applies to consumers -- "(typeless)" means that the consumer is capable of reading inputs using any data type, and "(float)" means that consumer always reads inputs as floating-point values.

To get sane results, applications must ensure that each value passed between pipeline stages is produced and consumed using the same data type. If a value is written in one stage as a floating-point value; it must be read as a floating-point value as well. If such a value is read as a signed or unsigned integer, its value is considered undefined. In practice, the raw bits used to represent the floating-point (IEEE single-precision floating-point encoding in the initial implementation of this spec) will be treated as an integer.

Type matching between stages is not enforced by the GL, because the overhead of doing so would be substantial. Such overhead would include:

* matching the inputs and outputs of each pipeline stage (fixed-function or programmable) every time the program configuration or fixed-function state changes,

* tracking the data type of each generic vertex attribute and checking it against the vertex program's inputs,

* tracking the data type of each program parameter and checking it against the manner the parameters were used in programs,

* matching color buffers against fragment program outputs.

Such error checking is certainly valuable, but the additional CPU overhead cost is substantial. Given that current CPUs often have a hard time keeping up with high-end GPUs, adding more overhead is a step in the wrong direction. We expect developer tools, such as instrumented drivers, to be able to provide type checking on most interfaces.

The diagram below depicts assembly programmability. Using vertex, geometry, and fragment shaders provided by the OpenGL Shading Language (GLSL) isn't substantially different from the assembly interface, except that the interfaces between programmable pipeline stages are more tightly coupled in GLSL (vertex, geometry, and fragment shaders are linked together into a single program object), and that shader variables are more strongly typed in GLSL than in the assembly interface.

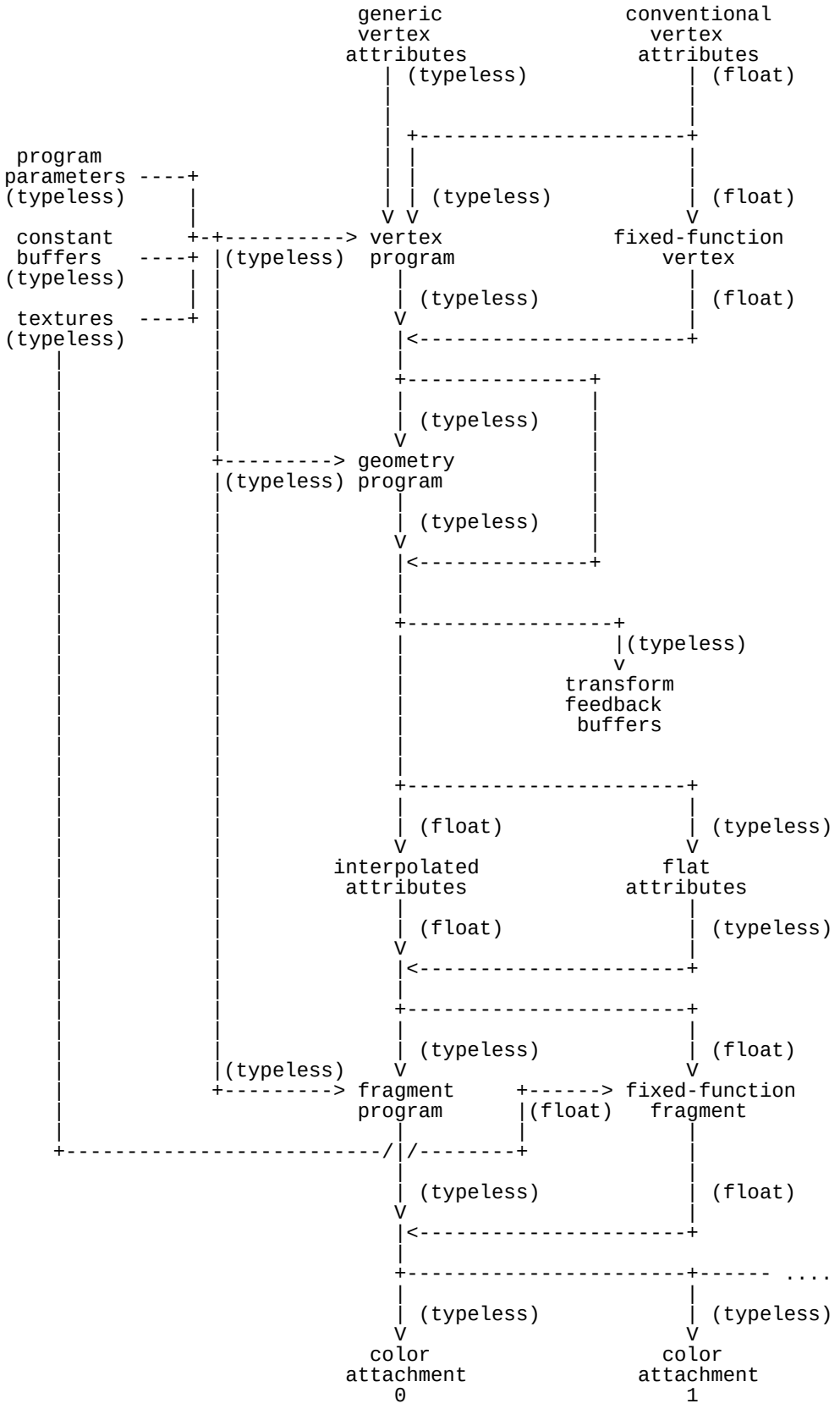
In the figure below, the first programmable stage is vertex program execution. For all inputs read by the vertex program, they must be specified in the GL vertex APIs (immediate mode or vertex arrays) using a data type matching the data type read by the shader. Additionally, vertex programs (and all other program types) can read program parameters, parameter buffers, and textures. In all cases the parameter, buffer, or texture data must be accessed in the shader using the same data type used to specify the data. If vertex programs are disabled, fixed-function vertex processing is used. Fixed-function vertex processing is fully floating-point, and all the conventional vertex attributes and state used by fixed-function are floating-point values.

After vertex processing, an optional geometry program can be executed, which reads attributes written by vertex programs (or fixed-functon) and writes out new vertex attributes. The vertex attributes it reads must have been written by the vertex program (or fixed-function) using a matching data type.

After geometry program execution, vertex attributes can optionally be written out to buffer objects using the NV_transform_feedback extension. The vertex attributes are written by the GL to the buffer objects using the same data type used to write the attribute in the geometry program (or vertex program if geometry programs are disabled).

Then, rasterization generates fragments based on transformed vertices. Most attributes written by vertex or geometry programs can be read by fragment programs, after the rasterization hardware "interpolates" them. This extension allows fragment programs to control how each attribute is interpolated. If an attribute is flat-shaded, it will be taken from the output attribute of the provoking vertex of the primitive using the same data type. If an attribute is smooth-shaded, the per-vertex attributes will be interpreted as a floating-point value, and a floating-point result. One necessary consequence of this is that any integer per-fragment attributes must be flat-shaded. To prevent some interpolation type errors, assembly and GLSL fragment shaders will not compile if they declare an integer fragment attribute that is not flat shaded. [NOTE: While point primitives generally have constant attributes, any integer attributes must still be flat-shaded; point rasterization may perform (degenerate) floating-point interpolation.]

```
Fragment programs must read attributes using data types matching the
outputs of the interpolation or flat-shading operations.  They may write
one or more color outputs using any data type, but the data type used
must match the corresponding framebuffer attachments.  Outputs directed
at signed or unsigned integer textures (EXT_texture_integer) must be
written using the appropriate integer data type; all other outputs must
be written as floating-point values.  Note that some of the
fixed-function per-fragment operations (e.g., blending, alpha test) are
specified as floating-point operations and are skipped when directed at
signed or unsigned integer color buffers.
```


generic
vertex
attributes
conventional
vertex
attributes
(typeless)
(float)
program
parameters
(typeless)
constant
buffers
(typeless)
textures
(typeless)
vertex
program
fixed-function
vertex
geometry
program
transform
feedback
buffers
fragment
program
fixed-function
fragment
interpolated
attributes
flat
attributes
color
attachment
0
color
attachment
1

(5) Instructions can operate on signed integer, unsigned integer, and floating-point values. Some operations make sense on all three data types? How is this supported, and what type checking support is provided by the assembler?

RESOLVED: One important property of the instruction set is that the data type for all operands and the result is fully specified by the instructions themselves. For instructions (such as ADD) that make sense for both integer and floating-point values, an optional data type modifier is provided to indicate which type of operation should be performed. For example, "ADD.S", "ADD.U", and "ADD.F", add signed integers, unsigned integers, or floating-point values, respectively. If no data type modifier is provided, ".F" is assumed if the instruction can apply to floating-point values and ".S" is assumed otherwise.

To help identify errors where the wrong data type is used -- for example, adding integer values in an ADD instruction that omits a data type modifier and thus defaults to "ADD.F" -- variables may be declared with optional data type modifiers. In the following code:

```
INT TEMP a;
UINT TEMP b;
FLOAT TEMP c;
TEMP d;
```

"a", "b", "c", and "d" are declared as temporary variables holding signed integer, unsigned integer, floating-point, and typeless values. Since each instruction fully specifies the data type of each operand and its result, these data types can be checked against the data type assigned to the variables operated on. If the types don't match, and the variable is not typeless, an error is reported. The opcode modifier ".NTC" can be used to ignore such errors on a per-opcode basis, if required.

Note that when bindings are used directly in instructions, they are always considered typeless for simplicity. Some fixed-function bindings have an obvious data type, but other bindings (e.g., program parameters) can hold either integer or floating-point values, depending on how they were specified.

Variable data types are optional. Typeless variables are provided because some programs may want to reuse the same variable in several places with different data types.

(6) Should both signed (INT) and unsigned integer (UINT) data types be provided?

RESOLVED: Yes. Signed and unsigned integer operations are supported. Providing both "INT" and "UINT" variable modifiers distinguish between signed and unsigned values for type checking purposes, to ensure that unsigned values aren't read as signed values and vice versa.

This specification says if a value is read a signed integer, but was written as an unsigned integer, the value returned is undefined. However, signed and unsigned integers are interchangeable in practice, except for very large unsigned integers (which can't be represented as signed values of the equivalent size) or negative signed integers.

If programs know that they won't generate negative or very large values, signed and unsigned integers can be used interchangeably. To avoid type errors in the assembler in this case, typeless variables can be used. Or the ".NTC" modifier can be used when appropriate.

(7) Integer and floating-point constants are supported in the instruction set. Integer constants might be interpreted to mean either "real integer" values or floating-point values. How are they supported?

RESOLVED: When an obvious floating point constant is specified (e.g., "3.0"), the developers' intent is clear. If you try to use a floating-point value in an instruction that wants an integer operand, or a declaration of an integer parameter variable, the program will fail to load. An integer constant used in an instruction isn't quite as clear. But its meaning can be easily inferred because the operand types of instructions are well-known at compile time. An integer multiply involving the constant "2" will interpret the "2" as an integer. A floating-point multiply involving the same constant "2" will interpret it as a floating-point value.

The only real problem is for a parameter declaration that is typeless. For typed variables, the intent is clear:

```
INT PARAM two = 2;              # use integer 2
FLOAT PARAM twoPt0 = 2;         # use floating-point 2.0
```

For typeless variables, there's no context to go on:

```
PARAM two = 2;                  # 2?  2.0?
```

This extension is intended to be largely upward-compatible with ARB_vertex_program, ARB_fragment_program, and the other extensions built on top of them. In all of these, the previous declaration is legal and means "2.0". For compatibility, we choose to interpret integer constants in this case as floating-point values. The assembler in the NVIDIA implementation will issue a warning if this case ever occurs.

This extension does not provide decoration of integer constant values -- we considered adding suffixed integers such as "2U" to mean "2, and don't even think about converting me to a float!". We expect that it will be sufficient to use the "INT" or "FLOAT" modifiers to disambiguate effectively.

(8) Should hexadecimal constants (e.g., 0x87A3 or 0xFFFFFFFF) be supported?

RESOLVED: Yes.

(9) Should we provide data type modifiers with explicit component sizes? For example, "INT8", "FLOAT16", or "INT32". If so, should we provide a mechanism to query the size (in bits) of a variable, or of different variable types/qualifiers?

RESOLVED: No.

(10) Should this extension provide better support for array variables?

RESOLVED: Yes; array variables of all types are allowed.

In ARB_vertex_program, program parameter (constant) variables could be addressed as arrays. Temporary variables, vertex attributes, and vertex results could not be declared as arrays.

In NV_vertex_program3 and NV_fragment_program2, relative addressing was supported in program bindings:

```
MOV R0, vertex.attrib[A0.x];            # vertex
MOV result.texcoord[A0.x], R0;          # vertex
MOV R0, fragment.texcoord[A0.x];        # fragment -- inside LOOP
```

Explicitly declared attribute or result arrays were not supported, and temporaries could also not be arrays.

This extension allows users to declare attribute, result, and temporary arrays such as:

```
ATTRIB attribs[] = { vertex.attrib[7..11] };
TEMP scratch[10];
RESULT texcoords[] = { result.texcoord[0..3] };
```

Additionally, the relative addressing mechanisms provided by NV_vertex_program3 and NV_fragment_program2 are NOT supported in this extension -- instead, declared array variables are the only way to get relative addressing. Using declared arrays allows the assembler to identify which attributes will actually be used. An expression like "vertex.texcoord[A0.x]" doesn't identify which texture coordinates are referenced, and the assembler must be conservative in this case and assume that they all are.

(11) Is relative addressing of temporaries allowed?

RESOLVED: Yes. However, arrays of temporaries may end up being stored in off-chip memory, and may be slower to access than non-array temporaries.

(12) Should this extension add bindings to pass generic attributes between vertex, geometry, and fragment programs, or are texture coordinates sufficient?

RESOLVED: While texture coordinates have been used in the past, generic attributes should be provided.

The assembler provides a large set of bindings and automatically eliminates generic attributes or components that are unused. At each interface between programs, there is an implementation-dependent limit on the number of attribute components that can be passed.

There are several reasons that this approach was chosen. First, if the number of attributes that can be passed between program stages exceeds the number of existing texture coordinate sets supported when specifying vertex, a second implementation-dependent number of texture coordinates would need to be exposed to cover the number supported between stages.

Second, the mechanisms described above reduce or eliminate the need to pack attributes into four component vectors. Third, "texture coordinates" that have been historically used for texture lookups don't need to be used to pass values that aren't used this way.

(13) The structured branching support in NV_fragment_program2 provides a REP instruction that says to repeat a block of code <N> times, as well as a LOOP instruction that does the same, but also provides a special loop counter variable. What sort of looping mechanism should we provide here?

RESOLVED: Provide only the REP instruction. The functionality provided by the LOOP instruction can be easily achieved by using an integer temporary as the loop index. This avoids two annoyances of the old LOOP models: (a) the loop index (A0.x) is a special variable name, while all other variables are declared normally and (b) instructions can only access the loop index of the innermost loop -- loop indices at higher nesting levels are not accessible.

One other option was a considered -- a "LOOPV" instruction (LOOP with a variable where the program specified a variable name and component to hold the loop index, instead of using the implicit variable name "A0.x". In the end, it was decided that using an integer temporary as a loop counter was sufficient.

(14) The structured branching support in NV_fragment_program2 provides a REP instruction that requires a loop count. Some looping constructs may not have a definite loop count, such as a "while" statement in C. Should this construct be supported, and if so, how?

RESOLVED: The REP instruction is extended to make the loop count optional. If no loop count is provided, the REP instruction specified a loop that can only be exited using the BRK (break) or RET instructions. To avoid obvious infinite loops, an error will be reported if a REP/ENDREP block contains no BRK instruction at the current nesting level and no RET instruction at any nesting level.

To implement a loop like "while (value < 7.0) ...", code such as the following can be used:

```
TEMP cc;                        # dummy variable
REP;
  SLT.CC cc.x, value.x, 7.0;    # compare value.x to 7.0, set CC0
  BRK NE.x;                     # break out if not true
  ...
  ...                           # presumably update value!
  ...
ENDREP;
```

(15) The structured branching support in NV_fragment_program2 provides a BRK instruction that operates like C's "break" statement. Should we provide something similar to C's "continue" statement, which skips to the next iteration of the loop?

RESOLVED: Yes, a new CONT opcode is provided for this purpose.

(16) Can the BRK or CONT instructions break out of multiple levels of nested loops at once?

RESOLVED: No. BRK and CONT only exit the current nesting level. To break out of multiple levels of nested loops, multiple BRK/CONT instructions are required.

(17) For REP instructions, is the loop counter reloaded on each iteration of the loop?

RESOLVED: No. The loop counter is loaded once at the top of the loop, compared to zero at the top of the loop, and decremented when each loop iteration completes. A program may overwrite the variable used to specify the initial value of the loop counter inside the loop without affecting the number of times the loop body is executed.

(18) How are floating-point values represented in this extension? What about floating-point arithmetic operations?

RESOLVED: In the initial hardware implementation of this extension, floating-point values are represented using the standard 32-bit IEEE single-precision encoding, consisting of a sign bit, 8 exponent bits, and 23 mantissa bits. Special encodings for NaN (not a number), +/-INF (infinity), and positive and negative zero are supported. Denorms (values less than 2^-126, which have an exponent encoding of "0" and no implied leading one) are supported, but may be flushed to zero, preserving the sign bit of the original value. Arithmetic operations are carried out at single-precision using normal IEEE floating-point rules, including special rules for generating infinities, NaNs, and zeros of each sign.

Floating-point temporaries declared as "SHORT" may be, but are not necessarily, stored as 16-bit "fp16" values (sign bit, five exponent bits, ten mantissa bits), as specified in the NV_float_buffer and ARB_half_float_pixel extensions.

(19) Should we provide a method to declare how fragment attributes are interpolated? It is possible to have flat-shaded attributes, perspective-corrected attributes, and centroid-sampled attributes.

RESOLVED: Yes. Fragment program attribute variable declarations may specify the "FLAT", "NOPERSPECTIVE", and "CENTROID" modifiers.

These modifiers are documented in detail in the NV_fragment_program4 specification.

(20) Should vertex and primitive identifiers be supported? If so, how?

RESOLVED: A vertex identifier is available as "vertex.id" in a vertex program. The vertex ID is equal to value effectively passed to ArrayElement when the vertex is specified, and is defined only if vertex arrays are used with buffer objects (VBOs).

A primitive identifier is available as "primitive.id" in a geometry or fragment program. The primitive ID is equal to the number of primitives processed since the last implicit or explicit call to glBegin().

See the NV_vertex_program4 spec for more information on vertex IDs, and the NV_geometry_program4 or NV_fragment_program4 specs for more information on primitive IDs.

(21) For integer opcodes, should a bitwise inversion operator "~" be provided, analogous to existing negation operator?

RESOLVED: No. If this operator were provided, it might allow a program to evaluate the expression "a&(~b)" using a single instruction:

```
AND.U a, a, ~b;
```

Instead, it is necessary to instead do something like:

```
UINT TEMP t;
NOT.U t, b;
AND.U a, a, t;
```

If necessary, this functionality could be added in a subsequent extension.

(22) What happens if you negate or take the absolute value of the biggest-magnitude negative integer?

RESOLVED: Signed integers are represented using two's complement representation. For 32-bit integers, the largest possible value is 2^31-1; the smallest possible value is -2^31. There is no way to represent 2^31, which is what these operators "should" return. The value returned in this case is the original value of -2^31.

(23) How do condition codes work? How are they different from those provided in previous NVIDIA extensions?

RESOLVED: There are two condition codes -- CC0 and CC1 -- each of which is a four-component vector. The condition codes are set based on the result of an instruction that specifies a condition code update modifier. Examples include:

```
ADD.S.CC  R0, R1, R2;       # add signed integers R1 and R2, update
                            #   CC0 based on the result, write the
                            #   final value to R0
ADD.F.CC1 R3, R4, R5;       # add floats R4 and R5, update CC1 based
                            #   on the result, write the final value
                            #   to R3
ADD.U.CC0 R6.xy, R7, R8;    # add unsigned integers R7 and R8, update
                            #   CC0 (x and y components) based on the
                            #   result, write the final value to R6
                            #   (x and y components)
```

Condition codes can be used for conditional writes, conditional branches, or other operations. The condition codes aren't used directly, but are instead used with a condition code test such as "LT" (less than) or "EQ" (equal to). Examples include:

```
  MOV R0 (GT.x), R1;          # move R1 to R0 only if the x component of
                              #   CC0 indicates a result of ">0"
  MOV R2 (NE1), R3;           # component-wise move of R3 to R2 if the
                              #   corresponding component of CC1
                              #   indicates a result of "!=0"
  IF LE0.xyxy;                # execute the block of code if the x or
    ...                       #   y components of CC0 indicate a result
  ENDIF;                      #   of "<=0"
  REP;
    ...
    BRK EQ1.xyzx;             # break out of loop if the x, y, or z
  ENDREP;                     #   components of CC1 indicate a result of
                              #   "==0".
```

Previous NVIDIA extensions provide eight tests, which are still supported here. The tests "EQ" (equal), "GE" (greater/equal), "GT" (greater than), "LE" (less/equal), "LT" (less than), and "NE" (not equal) can be used to determine the relation of the result used to set the condition code with zero. The tests "TR" (true) and "FL" (false), are special tests that always evaluate to true or false respectively.

For floating-point results, a NaN (not a number) encoding causes the "NE" condition to evaluate to TRUE and all other conditions to evaluate to FALSE. IEEE encodings for "negative" and "positive" zero are both treated as equal to zero.

Condition codes are implemented as a set of flags, which are set depending on the type of operation, as described in the spec.

For instructions that return floating-point or signed integer values, the normal condition code tests reliably indicate the relationship of the result to zero. For instructions that return unsigned values, the condition codes are a bit more complicated. For example, the sign flag is set if the most significant bit of the result written is set. As a result, very large unsigned integer values (e.g., 0x80000000 - 0xFFFFFFFF) are effectively treated as negative values. Condition code tests should be used with care with unsigned results -- to test if an unsigned integer is ">0", use a sequence like:

```
  MOV.U.CC R0, R1;            # move R1 to R0, set condition code
  IF NE;                      # test if the result is "!=0", a very
    ...                       #   large value might fail "GT"!
  ENDIF;
```

This extension provides a number of additional condition code tests useful for different floating-point or integer operations:

* NAN (not a number) is true if a floating-point result is a NaN. LEG (less, equal to, or greater) is the opposite of NAN.

* CF (carry flag) is true if an unsigned add overflows, or if an unsigned subtract produces a non-negative value. NCF (no carry flag) is the opposite of CF.

* OF (overflow flag) is true if a signed add or subtract overflows. NOF (no overflow flag) is the opposite of OF.

* SF (sign flag) is true if the sign flag is set. NSF (no sign flag) is the opposite of SF.

* AB (above) is true if an unsigned subtract produces a positive result. BLE (below or equal) is the opposite of AB, and is true if an unsigned subtract produces a negative result or zero. Note that CF can be used to test if the result is greater than or equal to zero, and NCF can be used to test if the result is less than zero.

(24) How do the "set on" instructions (SEQ, SGE, SGT, SLE, SLT, SNE) work with integer values and/or condition codes?

RESOLVED: "Set on" instructions comparing signed and unsigned values return zero if the condition is false, and an integer with all bits set if the condition is true. If the result is signed, it is interpreted as -1. If the result is unsigned, it is interpreted the largest unsigned value (0xFFFFFFFF for 32-bit integers). This is different from the floating-point "set on", which is defined to return 1.0.

This specific result encoding was chosen so that bitwise operators (NOT, AND, OR, XOR) can be used to evaluate boolean expressions.

When performing condition code tests on the results of an integer "set on" instruction, keep in mind that a TRUE result has the most significant bit set and will be interpreted as a negative value. To test if a condition is true, use "NE" (!=0). A condition code test of "GT" will always fail if the condition code was written by an integer "set on" instruction.

(25) What new texture functionality is provided?

RESOLVED: Several new features are provided.

First, the TXF (texel fetch) instruction allows programs to access a texture map like a normal array. Integer coordinates identifying an individual texel and LOD are provided, and the corresponding texture data is returned without filtering of any type.

Second, the TXQ (texture size query) instruction allows programs to query the size of a specified level of detail of a texture. This feature allows programs to perform computations dependent on the size of the texture without having to pass the size as a program parameter or via some other mechanism.

Third, applications may specify a constant texel offset in a texture instruction that moves the texture sample point by the specified number of texels. This offset can be used to perform custom texture filtering, and is also independent of the size of the texture LOD -- the same offsets are applied, regardless of the mipmap level.

Fourth, shadow mapping is supported for cube map textures. The first three coordinates are the normal (s,t,r) coordinates for a cube map texture lookup, and the fourth component is a depth reference value that can be compared to the depth value stored in the texture.

(26) What "consistency" requirements are in effect for textures accessed via the TXF (texel fetch) instruction?

UNRESOLVED: The texture must be usable for regular texture mapping operations -- if texture sizes or formats are inconsistent and a mipmapped min filter is used, the results are undefined.

(27) How does the TXF instruction work with bordered textures?

RESOLVED: The entire image can be accessed, including the border texels. For a 64x64 2D texture plus border (66x66 overall), the lower left border texel is accessed using the coordinates (-1,-1); the upper right border texel is accessed using the coordinates (64,64).

(28) What should TXQ (texture size query) return for "irrelevant" texture sizes (e.g., height of a 1D texture)? Should it return any other information at the same time?

RESOLVED: This specification leaves all "extra" components undefined.

(29) How do texture offsets interact with cubemap textures?

RESOLVED: They are not supported in this extension.

(30) How do texture offsets interact with mipmapped textures?

RESOLVED: The texture offsets are added after the (s,t,r) coordinates have been divided by q (if applicable) and converted to (u,v,w) coordinates by multiplying by the size of the selected texture level. The offsets are added to the (u,v,w) coordinates, and always move the sample point by an integral number of texel coordinates. If multiple mipmaps are accessed, the sample point in each mipmap level is moved by an identical offset. The applied offsets are independent of the selected mipmap level.

(31) How do shadow cube maps work?

UNRESOLVED: An application can define a cube map texture with a DEPTH_COMPONENT internal format, and then render a scene using the cube map faces as the depth buffer(s). When rendering the projection should be set up using the "center" of the cubemap as the eye, and using a normal projection matrix. When applying the shadow map, the fragment program read the (x,y,z) eye coordinates, compute the length of the major axis (MAX(|x|,|y|,|z|) and then transform this coordinate to [0,1] space using the same parameters used to derive Z in the projection matrix. A 4-component vector consisting of x, y, z, and this computed depth value should be passed to the texture lookup, and normal shadow mapping operations will be performed.

This issue should include the math needed to do this computation and sample code.

(32) Integer multiplies can overflow by a lot. Should there be some way to return the high part of both unsigned and signed integer multiplies?

RESOLVED: Yes. The ".HI" multipler is provided to do a return the 32 MSBs of a 32x32 integer multiply. The instruction sequence:

```
INT TEMP R0, R1, R2, R3;
MUL.S    R0, R2, R3;
MUL.S.HI R1, R2, R3;
```

will do a 32x32 signed integer multiply of R2 and R3, with the 32 LSBs of the 64-bit result in R0 and the 32 MSBs in R1.

(33) Should there be any other special multiplication modifiers?

RESOLVED: Yes. The ".S24" and ".U24" modifiers allow for signed and unsigned integer multiplies where both operands are guaranteed to fit in the least significant 24 bits. On some architectures supporting this extension, ".S24" and ".U24" integer multiplies may be faster than general-purpose ".S" and ".U" multiplies. If either value doesn't fit in 24 bits, the results of the operation are undefined -- implementations may, but are not required to, ignore the MSBs of the operands if ".S24" or ".U24" is specified.

(34) This extension provides subroutines, but doesn't provide a stack to push and pop parameters. How do we deal with this? NV_vertex_program3 supported PUSHA/POPA instructions to push and pop address registers.

RESOLVED: No explicit stack is required. A program can implement a stack by allocating a temporary array plus a single integer temporary to use as the stack "pointer". For example:

```
TEMP stack[256];                # 256 4-component vectors
INT TEMP sp;                    # sp.x == stack pointer
INT TEMP cc;                    # condition code results

function:
  SGE.S.CC cc.x, sp.x, 256;     # compute stackPointer >= 256
  RET NE.x;                     # return if TRUE
  MOV stack[sp], R0;            # push R0 onto the stack
  ADD.S sp.x, sp.x, 1;
  ...
  SUB.S sp.x, sp.x, 1;          # pop R0 off the stack
  MOV R0, stack[sp];
  RET
```

(35) Should we provide new vector semantics for previously-defined opcodes (e.g., LG2 computes a component-wise logarithm)?

RESOLVED: Not in this extension. The instructions we define here are compatible with the vector or scalar nature of previously defined opcodes. This simplifies the implementation of an assembler that needs to support both old and new instruction sets.

(36) Should it really be undefined to read from a register storing data of one type with an instruction of the other type (e.g., to read the bits of a floating-point number as an unsigned integer)?

RESOLVED: The spec describes undefined results for simplicity. In practice, mixing data types can be done, where signed integers are represented as two's complement integers and floating-point numbers are represented using IEEE single-precision representation. For example:

```
TEMP R0, R1;                    # typeless
MOV.U R0, 0x3F800000;           # R0 = 1.0
MOV.U R1, 0xBF800000;           # R1 = -1.0
MUL.F R0, R0, R1;               # R0 = -1 * 1 = -1 (0xBF800000)
XOR.U R0, R0, R1;               # R0 = 0xBF800000 ^ 0xBF800000 = 0
NOT.U R0, R0;                   # R0 = 0xFFFFFFFF
I2F.S R0, R0;                   # R0 = -1.0 (0xFFFFFFFF = -1 signed)
SEQ.F R0, R0, R1;               # R0 = 1.0 (-1.0 == -1.0)
```

(37) Buffer objects can be sourced as program parameters using the NV_parameter_buffer_object extension. How are they accessed in a program?

RESOLVED: The instruction set and existing program environment and local parameter bindings operate largely on four-component vectors. However, NV_parameter_buffer_object exposes the ability to reach into buffers consisting of user-generated data or data written to the buffer object by the GPU. Such data sets may not consist entirely four-component floating-point vectors, so a four-component vector API may be unnatural. An application might need to reformat its data set to deal with this issue. Or it might generate odd code to compensate for mis-alignment -- for example, reading an array of 3-component vectors by doing two four-component vector accesses and then rotating based on alignment. Neither approach is particularly satisfying.

Instead, this extension takes the approach of treating parameter buffers as array of scalar words. When an individual buffer element is read, the single word is replicated to produce a four-component vector. To access an array of 3-component vectors, code like the following can be used:

```
PARAM buffer[] = { program.buffer[0] };
INT TEMP index;
TEMP R0;
...
MUL.S index, index, 3;          # to read "vec3" #X, compute 3*X
MOV R0.x, buffer[index+0];
MOV R0.y, buffer[index+1];
MOV R0.z, buffer[index+2];
```

(38) Should recursion be allowed? If so, how is the total amount of recursion limited?

RESOLVED: Recursion is allowed, and a call stack is provided by the implementation. The size of the call stack is limited to the implementation-dependent constant MAX_PROGRAM_CALL_DEPTH, and when a the call stack is full, the results of further CAL instructions is undefined. In the initial implementation of this extension, such instructions will have no effect.

Note that no stack is provided to hold local registers; a program may implement its own via a temporary array and integer stack "pointer".

(39) Variables are all four-component vectors in previous extensions. Should scalar or small-vector variables be provided?

RESOLVED: It would be a useful feature, but it was left out for simplicity. In practice, a variable where only the X component is used will be equivalent to a scalar.

(40) The PK (pack) and UP* (unpack) instructions allow packing multiple components of data into a single component. The bit packing is well-defined. Should we require specific data types (e.g., unsigned integer) to hold packed values?*

RESOLVED: No. Previous instruction sets only allowed programs to write packed values to a floating-point variable (the only data type provided). We will allow packed results to be written to a variable of any data type. Integer instructions can be used to manipulate bits of packed data in place.

(41) What happens when converting integers to floats or vice versa if there is insufficient precision or range to represent the result?

RESOLVED: For integer-to-float conversions, the nearest representable floating-point value is used, and the least significant bits of the original integer value are lost. For float-to-integer conversions, out-of-range values are clamped to the nearest representable integer.

(42) Why are some of the grammar rules so bizarre (e.g., attribUseD, attribUseV, attribUseS, attribUseVNS)?

RESOLVED: This grammar is based upon the original ARB_vertex_program grammar, which has a number of "interesting" characteristics. For example, some of the bindings provided by ARB_vertex_program naturally require some amount of lookahead. For example, a vertex program can write an output color using any of the following:

```
MOV result.color, 0;            # primary color
MOV result.color.primary, 0;    # primary color again
MOV result.color.secondary, 0;  # secondary color this time
```

The pieces of the color binding are separated by "." tokens. However, writemasks are also supported, which also use "." before the write mask. So, we could also have something like:

```
MOV result.color.xyz, 0;        # primary color with W masked off
```

In this form, a parser needs to look at both the "." and the "xyz" to determine that the binding being used is "result.color" (and not "result.color.secondary").

Additionally, some checks that should probably be semantic errors (e.g., allowing different swizzle or scalar operand selectors per instruction, or disallowing both in the case of SWZ) we specified in the original grammar.

ARB_fragment_program and subsequent NVIDIA instructions built upon this, and the grammar for this extension was rewritten in the current form so it could be validated more easily.

(43) This is an NV extension (NV_gpu_program4). Why does the MAX_PROGRAM_TEXEL_OFFSET_EXT token has an "EXT" suffix?

RESOLVED: This token is shared between this extension and the comparable high-level GLSL programmability extension (EXT_gpu_shader4). Rather than provide a duplicate set of token names, we simply use the EXT version here.

Revision History

Rev.	Date	Author	Changes
4	02/04/08	pbrown	Fix errors in texture wrap mode handling. Added a missing clamp to avoid sampling border in REPEAT mode. Fixed incorrectly specified weights for LINEAR filtering.

Name

NV_parameter_buffer_object

Name Strings

None (impled by NV_GPU_program4)

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)
Eric Werness, NVIDIA Corporation (ewerness 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 04/18/2007
NVIDIA Revision: 7

Number

339

Dependencies

OpenGL 2.0 is required.

NV_gpu_program4 is required.

This extension is written against the OpenGL 2.0 specification.

NV_transform_feedback affects this extension.

Overview

This extension, in conjunction with NV_gpu_program4, provides a new type of program parameter than can be used as a constant during vertex, fragment, or geometry program execution. Each program target has a set of parameter buffer binding points to which buffer objects can be attached.

A vertex, fragment, or geometry program can read data from the attached buffer objects using a binding of the form "program.buffer[a][b]". This binding reads data from the buffer object attached to binding point <a>. The buffer object attached is treated either as an array of 32-bit words or an array of four-component vectors, and the binding above reads the array element numbered <b>.

The use of buffer objects allows applications to change large blocks of program parameters at once, simply by binding a new buffer object. It also provides a number of new ways to load parameter values, including readback from the frame buffer (EXT_pixel_buffer_object), transform feedback (NV_transform_feedback), buffer object loading functions such as MapBuffer and BufferData, as well as dedicated parameter buffer update functions provided by this extension.

New Procedures and Functions

```
    void BindBufferRangeNV(enum target, uint index, uint buffer,
                           intptr offset, sizeiptr size);
    void BindBufferOffsetNV(enum target, uint index, uint buffer,
                            intptr offset);
    void BindBufferBaseNV(enum target, uint index, uint buffer);
    void ProgramBufferParametersfvNV(enum target, uint buffer, uint index,
                                     sizei count, const float *params);
    void ProgramBufferParametersIivNV(enum target, uint buffer, uint index,
                                      sizei count, const int *params);
    void ProgramBufferParametersIuivNV(enum target, uint buffer, uint index,
                                       sizei count, const uint *params);
    void GetIntegerIndexedvEXT(enum value, uint index, boolean *data);
```

New Tokens

Accepted by the <pname> parameter of GetProgramivARB:

MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV	0x8DA0
MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV	0x8DA1

Accepted by the <target> parameter of ProgramBufferParametersfvNV, ProgramBufferParametersIivNV, and ProgramBufferParametersIuivNV, BindBufferRangeNV, BindBufferOffsetNV, BindBufferBaseNV, and BindBuffer and the <value> parameter of GetIntegerIndexedvEXT:

VERTEX_PROGRAM_PARAMETER_BUFFER_NV	0x8DA2
GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV	0x8DA3
FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV	0x8DA4

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify "Section 2.14.1" of the ARB_vertex_program specification.

(Add after the discussion of environment parameters.)

Additionally, each program target has an array of parameter buffer binding points, to which a buffer object (Section 2.9) can be bound. The number of available binding points is given by the implementation-dependent constant MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV. These binding points are shared by all programs of a given type. All bindings are initialized to the name zero, which indicates that no valid binding is present.

A program parameter binding is associated with a buffer object using BindBufferOffset with a <target> of VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV and <index> corresponding to the number of the desired binding point. The error INVALID_VALUE is generated if the value of <index> is greater than or equal to MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS.

Buffer objects are made to be sources of program parameter buffers by calling one of

```
  void BindBufferRangeNV(enum target, uint index, uint buffer,
                         intptr offset, sizeiptr size)
  void BindBufferOffsetNV(enum target, uint index, uint buffer,
                          intptr offset)
  void BindBufferBaseNV(enum target, uint index, uint buffer)
```

where <target> is set to VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV. Any of the three BindBuffer* commands perform the equivalent of BindBuffer(target, buffer). <buffer> specifies which buffer object to bind to the target at index number <index>. <index> must be less than the value of MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV. <offset> specifies a starting offset into the buffer object <buffer>. <size> specifies the number of elements in the bound portion of the buffer. Both <offset> and <size> are in basic machine units. The error INVALID_VALUE is generated if the value of <size> is less than or equal to zero. The error INVALID_VALUE is generated if <offset> or <size> are not word-aligned. For program parameter buffers, the error INVALID_VALUE is generated if <offset> is non-zero.

BindBufferBaseNV is equivalent to calling BindBufferOffsetNV with an <offset> of 0. BindBufferOffsetNV is the equivalent of calling BindBufferRangeNV with <size> = sizeof(buffer) - <offset> and rounding <size> down so that it is word-aligned.

All program parameter buffer parameters are either single-component 32-bit words or four-component vectors made up of 32-bit words. The program parameter buffers may hold signed integer, unsigned integer, or floating-point data. There is a limit on the maximum number of words of a buffer object that can be accessed using any single parameter buffer binding point, given by the implementation-dependent constant MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV. Buffer objects larger than this size may be used, but the results of accessing portions of the buffer object beyond the limit are undefined.

The commands

```
  void ProgramBufferParametersfvNV(enum target, uint buffer, uint index,
                                   sizei count, const float *params);
  void ProgramBufferParametersIivNV(enum target, uint buffer, uint index,
                                    sizei count, const int *params);
  void ProgramBufferParametersIuivNV(enum target, uint buffer, uint index,
                                    sizei count, const uint *params);
```

update words <index> through <index>+<count>-1 in the buffer object bound to the binding point numbered <buffer> for the program target <target>. The new data is referenced by <params>. The error INVALID_OPERATION is generated if no buffer object is bound to the binding point numbered <buffer>. The error INVALID_VALUE is generated if <index>+<count> is greater than either the number of words in the buffer object or the maximum parameter buffer size MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV. These functions perform an operation functionally equivalent to calling BufferSubData, but possibly with higher performance.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify the second paragraph of section 6.1.1 (Simple Queries) p. 244 to read as follows:

...<data> is a pointer to a scalar or array of the indicated type in which to place the returned data.

```
void GetIntegerIndexedvEXT(enum target, uint index,
                           boolean *data);
```

are used to query indexed state. <target> is the name of the indexed state and <index> is the index of the particular element being queried. <data> is a pointer to a scalar or array of the indicated type in which to place the returned data.

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

TBD

Dependencies on NV_transform_feedback

Both NV_transform_feedback and this extension define the behavior of BindBuffer{Range, Offset, Base}NV. Both definitions should be functionally identical.

Errors

The error INVALID_VALUE is generated by BindBufferRangeNV, BindBufferOffsetNV, or BindBufferBaseNV if <target> is VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV, and <index> is greater than or equal to MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS.

The error INVALID_VALUE is generated by BindBufferRangeNV or BindBufferOffsetNV if <offset> or <size> is not word-aligned.

The error INVALID_VALUE is generated by BindBufferRangeNV if <size> is less than zero.

The error INVALID_VALUE is generated by BindBufferRangeNV or BindBufferOffsetNV if <target> is VERTEX_PROGRAM_PARAMETER_BUFFER_NV, GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV, or FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV, and <offset> is non-zero.

The error INVALID_OPERATION is generated by ProgramBufferParametersfvNV, ProgramBufferParametersIivNV, or ProgramBufferParametersIuivNV if no buffer object is bound to the binding point numbered <buffer> for program target <target>.

The error INVALID_VALUE is generated by ProgramBufferParametersfvNV, ProgramBufferParametersIivNV, or ProgramBufferParametersIuivNV if the sum of <index> and <count> is greater than either the number of words in the buffer object boudn to <buffer> or the maximum parameter buffer size MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV.

New State

(Modify ARB_vertex_program, Table X.6 -- Program State)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_PROGRAM_PARAMETER_BUFFER_NV	Z+	GetIntegerv	0	Active vertex program buffer object binding	2.14.1	-
VERTEX_PROGRAM_PARAMETER_BUFFER_NV	nxZ+	GetInteger-IndexedvEXT	0	Buffer objects bound for vertex program use	2.14.1	-
GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV	Z+	GetIntegerv	0	Active geometry program buffer object binding	2.14.1	-
GEOMETRY_PROGRAM_PARAMETER_BUFFER_NV	nxZ+	GetInteger-IndexedvEXT	0	Buffer objects bound for geometry program use	2.14.1	-
FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV	Z+	GetIntegerv	0	Active fragment program buffer object binding	2.14.1	-
FRAGMENT_PROGRAM_PARAMETER_BUFFER_NV	nxZ+	GetInteger-IndexedvEXT	0	Buffer objects bound for fragment program use	2.14.1	-

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_PROGRAM_PARAMETER_BUFFER_BINDINGS_NV	Z	GetProgram-ivARB	8	size of program parameter binding tables	2.14.1	-
MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV	Z	GetProgram-ivARB	4096	maximum usable size of program parameter buffers	2.14.1	-

Examples

```
!!NVfp4.0
# Legal
BUFFER bones[] = { program.buffer[0] };
ALIAS funBone = bones[69];
MOV t, bones[1];
# Illegal
ALIAS numLights = program.buffer[5][6];
MOV t, program.buffer[3][x];
END
```

Issues

(1) PBO is already taken as an acronym? What do we call this?

RESOLVED: PaBO.

(2) How should the ability to simultaneously access multiple parameter buffers be exposed?

RESOLVED: In the program text (see NV_gpu_program4), the buffers are referred to using a buffer binding statement which is dereferenced in the instructions. In the rest of the APIs, an array of internal binding points is provided, which are dereferenced using the index parameter of BindBufferBase and associated functions.

(3) Should program parameter buffer bindings be provided per-target (i.e., environment parameters), per-program (i.e., local parameters), or some combination of the two?

RESOLVED: Per-target. That fits most naturally with the ARB program model, similar to textures. Having both per-program and per-target add complexity with no benefit.

(4) Should references to the parameter buffer be scalar or vector?

RESOLVED: Scalar. Having vector is more consistent with the legacy APIs, but is more difficult to build the arbitrary data structures that are interesting to store in a parameter buffer. A future extension can define an alternate keyword in the program text to specify accesses of a different size.

(5) Should parameter buffers be editable using the ProgramEnvParameter API?

RESOLVED: No. There is a new parallel API for the bindable buffers, including the ability to update multiple parameters at a time. These are more convenient than having to rebind for BufferData and potentially faster.

(6) Should parameter buffers be editable outside the ProgramBufferParameters API?

RESOLVED: Yes. The use of buffer objects allows the buffers to be naturally manipulated using normal buffer object mechanisms. That

includes CPU mapping, loading via BufferData or BufferSubData, and even reading data back using the ARB_pixel_buffer_object extension.

(7) Will buffer object updates from different sources cause potential synchronization problems? If so, how will they be resolved.

RESOLVED: If reads and write occur in the course of the same call (e.g. reading from a buffer using parameter buffer binding while writing to it using transform feedback. All other cases are allowed and occur in command order. Any synchronization is handled by the GL.

(8) Is there an implementation-dependent limit to the size of program parameter buffers?

RESOLVED: Yes, limited-size buffers are provided to reduce the complexity of the GPU design that supports program parameter buffer access and updates. However, the minimum limit is 16K scalar parameters, or 64KB. A larger buffer object can be provided, but only the first 64KB is accessible. The limit is queryable with GetProgramivARB with <pname> MAX_PROGRAM_PARAMETER_BUFFER_SIZE_NV.

(9) With scalar buffers, which parameter setting routines do we need?

UNRESOLVED: A function to set N scalars is very important. It might be nice to have convenience functions that take 1 or 4 parameters directly.

(10) Do we need GetProgramBufferParameter functions?

UNRESOLVED: Probably not - they aren't perf critical and offer no functionality beyond getting the buffer object data any of the standard ways.

(11) What happens if a value written using ProgramBufferParametersfNV is read as an integer or the other way around?

RESOLVED: Undefined - likely just a raw bit cast between whatever internal representations are used by the GL.

Revision History

Rev.	Date	Author	Changes
7	04/18/07	pbrown	Fixed state table to include the buffer object binding array for each program type.

Name

NV_transform_feedback

Name Strings

GL_NV_transform_feedback

Contributors

Cliff Woolley
Nick Carter

Contact

Barthold Lichtenbelt (blichtenbelt 'at' nvidia.com)
Pat Brown (pbrown 'at' nvidia.com)
Eric Werness (ewerness 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 02/04/2008
NVIDIA Revision: 14

Number

341

Dependencies

OpenGL 1.5 is required.

This extension interacts with EXT_timer_query.

NV_vertex_program4, NV_geometry_program4 and NV_gpu_program4 affect this extension.

EXT_geometry_shader4 trivially interacts with this extension.

This extension has an OpenGL Shading Language component. As such it interacts with ARB_shader_objects and OpenGL 2.0.

This extension is written against the OpenGL 2.0 specification.

Overview

This extension provides a new mode to the GL, called transform feedback, which records vertex attributes of the primitives processed by the GL. The selected attributes are written into buffer objects, and can be written with each attribute in a separate buffer object or with all attributes interleaved into a single buffer object. If a geometry program or shader is active, the primitives recorded are those emitted by the geometry program. Otherwise, transform feedback captures primitives whose

vertex are transformed by a vertex program or shader, or by fixed-function vertex processing. In either case, the primitives captured are those generated prior to clipping. Transform feedback mode is capable of capturing transformed vertex data generated by fixed-function vertex processing, outputs from assembly vertex or geometry programs, or varying variables emitted from GLSL vertex or geometry shaders.

The vertex data recorded in transform feedback mode is stored into buffer objects as an array of vertex attributes. The regular representation and the use of buffer objects allows the recorded data to be processed directly by the GL without requiring CPU intervention to copy data. In particular, transform feedback data can be used for vertex arrays (via vertex buffer objects), as the source for pixel data (via pixel buffer objects), as program constant data (via the NV_parameter_buffer_object or EXT_bindable_uniform extension), or via any other extension that makes use of buffer objects.

This extension introduces new query object support to allow transform feedback mode to operate asynchronously. Query objects allow applications to determine when transform feedback results are complete, as well as the number of primitives processed and written back to buffer objects while in transform feedback mode. This extension also provides a new rasterizer discard enable, which allows applications to use transform feedback to capture vertex attributes without rendering anything.

New Procedures and Functions

```
void BindBufferRangeNV(enum target, uint index, uint buffer,
                       intptr offset, sizeiptr size)
void BindBufferOffsetNV(enum target, uint index, uint buffer,
                        intptr offset)
void BindBufferBaseNV(enum target, uint index, uint buffer)
void TransformFeedbackAttribsNV(sizei count, const int *attribs,
                                enum bufferMode)
void TransformFeedbackVaryingsNV(uint program, sizei count,
                                 const int *locations,
                                 enum bufferMode)
void BeginTransformFeedbackNV(enum primitiveMode)
void EndTransformFeedbackNV()

int GetVaryingLocationNV(uint program, const char *name)
void GetActiveVaryingNV(uint program, uint index,
                        sizei bufSize, sizei *length, sizei *size,
                        enum *type, char *name)
void ActiveVaryingNV(uint program, const char *name)
void GetTransformFeedbackVaryingNV(uint program, uint index,
                                   int *location)

void GetIntegerIndexedvEXT(enum param, uint index, int *values);
void GetBooleanIndexedvEXT(enum param, uint index, boolean *values);
```

(Note: These indexed query functions are provided in the EXT_draw_buffers2 extension. The boolean query is not useful for any queryable value in this extension, but is supported for completeness and consistency with base GL typed "Get" functions.)

New Tokens

Accepted by the <target> parameters of BindBuffer, BufferData, BufferSubData, MapBuffer, UnmapBuffer, GetBufferSubData, GetBufferPointerv, BindBufferRangeNV, BindBufferOffsetNV and BindBufferBaseNV:

TRANSFORM_FEEDBACK_BUFFER_NV	0x8C8E

Accepted by the <param> parameter of GetIntegerIndexedvEXT and GetBooleanIndexedvEXT:

TRANSFORM_FEEDBACK_BUFFER_START_NV	0x8C84
TRANSFORM_FEEDBACK_BUFFER_SIZE_NV	0x8C85
TRANSFORM_FEEDBACK_RECORD_NV	0x8C86

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

TRANSFORM_FEEDBACK_BUFFER_BINDING_NV	0x8C8F

Accepted by the <bufferMode> parameter of TransformFeedbackAttribsNV and TransformFeedbackVaryingsNV:

INTERLEAVED_ATTRIBS_NV	0x8C8C
SEPARATE_ATTRIBS_NV	0x8C8D

Accepted by the <target> parameter of BeginQuery, EndQuery, and GetQueryiv:

PRIMITIVES_GENERATED_NV	0x8C87
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV	0x8C88

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

RASTERIZER_DISCARD_NV	0x8C89

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

MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV	0x8C8A
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV	0x8C8B
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS_NV	0x8C80
TRANSFORM_FEEDBACK_ATTRIBS_NV	0x8C7E

Accepted by the <pname> parameter of GetProgramiv:

ACTIVE_VARYINGS_NV	0x8C81
ACTIVE_VARYING_MAX_LENGTH_NV	0x8C82
TRANSFORM_FEEDBACK_VARYINGS_NV	0x8C83

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

TRANSFORM_FEEDBACK_BUFFER_MODE_NV	0x8C7F

Accepted by the <attribs> parameter of TransformFeedbackAttribsNV:

BACK_PRIMARY_COLOR_NV	0x8C77
BACK_SECONDARY_COLOR_NV	0x8C78
TEXTURE_COORD_NV	0x8C79
CLIP_DISTANCE_NV	0x8C7A
VERTEX_ID_NV	0x8C7B
PRIMITIVE_ID_NV	0x8C7C
GENERIC_ATTRIB_NV	0x8C7D
POINT_SIZE	0x0B11
FOG_COORDINATE	0x8451
SECONDARY_COLOR_NV	0x852D
PRIMARY_COLOR	0x8577
POSITION	0x1203
LAYER_NV	0x8DAA

(note: POINT_SIZE, FOG_COORDINATE, PRIMARY_COLOR, and POSITION are defined in the core OpenGL specification; SECONDARY_COLOR_NV is defined in NV_register_combiners.)

Returned by the <type> parameter of GetActiveVaryingNV:

UNSIGNED_INT_VEC2_EXT	0x8DC6
UNSIGNED_INT_VEC3_EXT	0x8DC7
UNSIGNED_INT_VEC4_EXT	0x8DC8

(note: All three of these are defined in the EXT_gpu_shader4 extension.)

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Insert three new sections between Sections 2.11, Coordinate Transforms and 2.12, Clipping:

(Move the "Asynchronous Queries" language out of Section 4.1.7)

Section 2.X, Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are two query types supported by the GL. Transform feedback queries (section 2.Y) returns information on the number of vertices and primitives processed by the GL and written to one or more buffer objects. Occlusion queries (section 4.1.7.1) count the number of fragments or samples that pass the depth test.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.12 provide mechanisms to determine

when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results will be written into the corresponding query object. If the active query object for a query type name is zero, no such information is being tracked.

A query object is created by calling

```
  void BeginQuery(enum target, uint id);
```

with an unused name <id>. <target> indicates the type of query to be performed; valid values of <target> are defined in subsequent sections. When a query object is created, the name <id> is marked as used and associated with a new query object.

BeginQuery sets the active query object name for the query type given by <target> to <id>. If BeginQuery is called with an <id> of zero, if the active query object name for <target> is non-zero, or if <id> is the active query object name for any query type, the error INVALID OPERATION is generated.

The command

```
  void EndQuery(enum target);
```

marks the end of the sequence of commands to be tracked for the query type given by <target>. The active query object for <target> is updated to indicate that query results are not available, and the active query object name for <target> is reset to zero. When the commands issued prior to EndQuery have completed and a final query result is available, the query object, active when EndQuery is, called is updated by the GL. The query object is updated to indicate that the query results are available and to contain the query result. If the active query object name for <target> is zero when EndQuery is called, the error INVALID_OPERATION is generated.

The command

```
  void GenQueries(sizei n, uint *ids);
```

returns <n> previously unused query object names in <ids>. These names are marked as used, but no object is associated with them until the first time they are used by BeginQuery.

Query objects are deleted by calling

```
  void DeleteQueries(sizei n, const uint *ids);
```

<ids> contains <n> names of query objects to be deleted. After a query object is deleted, its name is again unused. Unused names in <ids> are silently ignored.

Calling either GenQueries or DeleteQueries while any query of any target is active causes an INVALID_OPERATION error to be generated.

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The number of bits used to represent the query result is implementation-dependent. In the initial state of a query object, the result is available and its value is zero.

The necessary state for each query type is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress.

Section 2.Y, Transform Feedback

In 'transform feedback' mode the vertices of transformed primitives are written out to one or more buffer objects. The vertices are fed back after the geometry shader stage, if it exists, or otherwise after vertex processing right before clipping (section 2.12) but after color clamping. Optionally the transformed vertices can be discarded after being stored into one or more buffer objects, or they can be passed on down to the clipping stage for further processing.

Transform feedback is started and finished by calling

```
  void BeginTransformFeedbackNV(enum primitiveMode)
```

and

```
  void EndTransformFeedbackNV(),
```

respectively. Transform feedback is said to be active after a call to BeginTransformFeedbackNV and inactive after a call to EndTransformFeedbackNV. Transform feedback is initially inactive. Transform feedback is performed after color clamping, but immediately before clipping in the OpenGL pipeline. <primitiveMode> is one of TRIANGLES, LINES, or POINTS, and specifies the output type of primitives that will be recorded into the buffer objects bound for transform feedback (see below). <primitiveMode> places a restriction on the primitive types that may be rendered during an instance of transform feedback. See table X.1.

Transform Feedback primitiveMode	allowed render primitive modes
POINTS	POINTS
LINES	LINES, LINE_LOOP, and LINE_STRIP
TRIANGLES	TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, QUAD_STRIP, and POLYGON

Table X.1 Legal combinations between the transform feedback primitive mode, as passed to BeginTransformFeedbackNV and the current primitive mode.

If a geometry program or geometry shader is active, the output primitive type of the currently active program is used as the render primitive in table X.1, otherwise the Begin mode is used.

Quads and polygons will be tessellated and recorded as triangles (the order of tessellation within a primitive is undefined); primitives specified in strips or fans will be assembled and recorded as individual primitives. Incomplete primitives are not recorded. Begin or any operation that implicitly calls Begin (such as DrawElements) will generate INVALID_OPERATION if the begin mode is not an allowed begin mode for the current transform feedback buffer state. If a geometry program or geometry shader is active, its output primtive mode is used for the error check instead of the begin mode.

It is an invalid operation error to call BeginTransformFeedbackNV, TransformFeedbackBufferNV, TransformFeedbackVaryingsNV, TransformFeedbackAttribsNV, or UseProgram or LinkProgram on the currently active program object while transform feedback is active. It is an invalid operation error to call EndTransformFeedbackNV while transform feedback is inactive.

Transform feedback can operate in either INTERLEAVED_ATTRIBS_NV or SEPARATE_ATTRIBS_NV mode. In the INTERLEAVED_ATTRIBS_NV mode, several vertex attributes can be written, interleaved, into a single buffer object. In the SEPARATE_ATTRIBS_NV mode, vertex attributes are recorded, non-interleaved, into several buffer objects simultaneously.

It is an INVALID_OPERATION error to call BeginTransformFeedbackNV if there is no buffer object bound to index 0 (see the description of the BindBuffer* commands below) in INTERLEAVED_ATTRIBS_NV mode. It is also an INVALID_OPERATION error to call BeginTransformFeedbackNV if the number of buffer objects bound in SEPARATE_ATTRIBS_NV mode is less than the number of buffer objects required, as given by the current transform feedback state. It is also an INVALID_OPERATION error to call BeginTransformFeedbackNV if no attributes are specified to be captured in either separate or interleaved mode.

Buffer objects are made to be targets of transform feedback by calling one of

```
  void BindBufferRangeNV(enum target, uint index, uint buffer,
                         intptr offset, sizeiptr size)
  void BindBufferOffsetNV(enum target, uint index, uint buffer,
                          intptr offset)
  void BindBufferBaseNV(enum target, uint index, uint buffer)
```

where <target> is set to TRANSFORM_FEEDBACK_BUFFER_NV. Any of the three BindBuffer* commands perform the equivalent of BindBuffer(target, buffer). <buffer> specifies which buffer object to bind to the target at index number <index>. <index> exists for use with the SEPARATE_ATTRIBS_NV mode and must be less than the value of MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV. <offset> specifies a starting offset into the buffer object <buffer>. <size> specifies the number of elements that can be written during transform feedback mode. This is useful to prevent the GL from writing past a certain position in the buffer object. Both <offset> and <size> are in basic machine units. The error INVALID_VALUE is generated if the value of <size> is less than or

equal to zero. The error INVALID_VALUE is generated if <offset> or <size> are not word-aligned. The error INVALID_OPERATION is generated when any of the BindBuffer* commands is called while transform feedback is active.

BindBufferBaseNV is equivalent to calling BindBufferOffsetNV with an <offset> of 0. BindBufferOffsetNV is the equivalent of calling BindBufferRangeNV with <size> = sizeof(buffer) - <offset> and rounding <size> down so that it is word-aligned.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position <offset> + <size> - 1, as set by BindbufferRangeNV, then no vertices of the primitive are recorded, and the counter corresponding to the asynchronous query target TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV (see Section 2.Z) is not incremented.

Two methods exist to specify which transformed vertex attributes are streamed to one, or more, buffer objects in transform feedback mode. If an OpenGL Shading Language vertex and/or geometry shader is active, then the state set with the TransformFeedbackVaryingsNV() command determines which attributes to record. If neither a vertex nor geometry shader is active, the state set with the TransformFeedbackAttribsNV() command determines which attributes to record.

When a program object containing a vertex shader and/or geometry shader is active, the set of vertex attributes recorded in transform feedback mode is specified by

```
  void TransformFeedbackVaryingsNV(uint program, sizei count,
                                   const int *locations,
                                   enum bufferMode)
```

This command sets the transform feedback state for <program> and specifies which varying variables to record when transform feedback is active. The array <locations> contains <count> locations of active varying variables, as queried with GetActiveVaryingNV(), to stream to a buffer object. See section 2.15.3. <bufferMode> is one of INTERLEAVED_ATTRIBS_NV or SEPARATE_ATTRIBS_NV. The error INVALID_OPERATION is generated if any value in <locations> does not reference an active varying variable, or if any value in <locations> appears more than once in the array. The same error is generated if <program> has not been linked successfully. The program object's state value TRANSFORM_FEEDBACK_BUFFER_MODE_NV will be set to <bufferMode> and the program object's state value TRANSFORM_FEEDBACK_VARYINGS_NV set to <count>. These values can be queried with GetProgramiv (see section 6.1.14).

In the INTERLEAVED_ATTRIBS_NV mode, varying variables are written, interleaved, into one buffer object. This is the buffer object bound to index 0. Varying variables are written out to that buffer object in the order that they appear in the array <locations>. The error INVALID_OPERATION is generated if the total number of components of all varying variables specified in the array <locations> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

In the SEPARATE_ATTRIBS_NV mode, varying variables are recorded, non-interleaved, into several buffer objects simultaneously. The first

varying variable in the array <locations> is written to the buffer bound to index 0. The last varying variable is written to the buffer object bound to index <count> - 1. No more than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV buffer objects can be written to simultaneously. The error INVALID_VALUE is generated if <count> is greater than that limit. Furthermore, the number of components for each varying variable in the array <locations> cannot exceed MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS_NV. The error INVALID_VALUE is generated if any varying variable in <locations> exceeds this limit.

It is not necessary to (re-)link <program> after calling TransformFeedbackVaryingsNV(). Changes to the transform feedback state will be picked up right away after calling TransformFeedbackVaryingsNV().

The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex or geometry shader is undefined.

When neither a vertex nor geometry shader is active, the vertex attributes produced by fixed-function vertex processing or an assembly vertex or geometry program can be recorded in transform feedback mode. The set of attributes to record is specified by

```
void TransformFeedbackAttribsNV(sizei count, const int *attribs,
                                enum bufferMode)
```

This command specifies which attributes to record into one, or more, buffer objects. The value TRANSFORM_FEEDBACK_BUFFER_MODE_NV will be set to <bufferMode> and the value TRANSFORM_FEEDBACK_ATTRIBS_NV set to <count>. The array <attribs> contains an interleaved representation of the attributes desired to be fed back containing 3*count values. For attrib i, the value at 3*i+0 is the enum corresponding to the attrib, as given in table X.2. The value at 3*i+1 is the number of components of the provided attrib to be fed back and is between 1 and 4. The value at 3*i+2 is the index for attribute enumerants corresponding to more than one real attribute. For an attribute enumerant corresponding to only one attribute, the index is ignored. For an attribute enumerant corresponding to more than one attribute, the error INVALID_VALUE is generated if the index value is outside the allowable range for that attribute.

attrib	permitted sizes	index?	GPU_program_4 result name
POSITION	1,2,3,4	no	position
PRIMARY_COLOR	1,2,3,4	no	color.front.primary
SECONDARY_COLOR_NV	1,2,3,4	no	color.front.secondary
BACK_PRIMARY_COLOR_NV	1,2,3,4	no	color.back.primary
BACK_SECONDARY_COLOR_NV	1,2,3,4	no	color.back.secondary
FOG_COORDINATE	1	no	fogcoord
POINT_SIZE	1	no	pointsize
TEXTURE_COORD_NV	1,2,3,4	yes	texcoord[index]
CLIP_DISTANCE_NV	1	yes	clip[index]
VERTEX_ID_NV	1	no	vertexid
PRIMITIVE_ID_NV	1	no	primid
GENERIC_ATTRIB_NV	1,2,3,4	yes	attrib[index]
LAYER_NV	1	no	layer

Table X.2: Transform Feedback Attribute Specifiers.The 'attrib' column

specifies which attribute to record. The 'permitted sizes' column indicates how many components of the attribute can be recorded. The 'index' column indicates if the attribute is indexed. The 'gpu program 4' column shows which result variable of a vertex or geometry program corresponds to the attribute to record.

The TransformFeedbackAttribsNV() command sets transform feedback state which is used both when the GL is in fixed-function vertex processing mode, as well as when an assembly vertex and/or geometry program is active.

The parameter <bufferMode> has the same meaning as described for TransformFeedbackVaryingsNV(). Attributes are either written interleaved, or into separate buffer objects, in the same manner as described earlier for TransformFeedbackVaryingsNV().

In the INTERLEAVED_ATTRIBS_NV mode, the error INVALID_VALUE is generated if the sum of the values of elements 3*i+1 in the array <attribs> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

In the SEPARATE_ATTRIBS_NV mode, no more than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV buffer objects can be written to simultaneously. The error INVALID_VALUE is generated if <count> is greater than that limit.

The error INVALID_OPERATION is generated if any attribute appears more than once in the array <attribs>.

The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex or geometry program is undefined. The values of PRIMITIVE_ID_NV or LAYER_NV for a vertex is defined if and only if a geometry program is active and that program writes to the result variables "result.primid" or "result.layer", respectively. The value of VERTEX_ID_NV is only defined if and only if a vertex program is active, no geometry program is active, and the vertex program writes to the output attribute "result.id".

Section 2.Z, Primitive Queries

Primitive queries use query objects to track the number of primitives generated by the GL and to track the number of primitives written to transform feedback buffers.

When BeginQuery is called with a <target> of PRIMITIVES_GENERATED_NV, the primitives-generated count maintained by the GL is set to zero. When the generated primitive query is active, the primitives-generated count is incremented every time a primitive reaches the Discarding Rasterization stage (see Section 3.x) right before rasterization. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform-feedback stage, if active.

When BeginQuery is called with a <target> of TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV, the transform-feedback-primitives-written count maintained by the GL is set to zero. When the transform feedback primitive written query is active, the transform-feedback-primitives-written count is incremented every time a

primitive is recorded into a buffer object. If transform feedback is not active, this counter is not incremented. If the primitive does not fit in the buffer object, the counter is not incremented.

These two queries can be used together to determine if all primitives have been written to the bound feedback buffers; if both queries are run simultaneously and the query results are equal, all primitives have been written to the buffer(s). If the number of primitives written is less than the number of primitives generated, the buffer is full.

Modify section 2.15.3 "Shader Variables", page 75.

Change the second sentence in the first paragraph on p. 84 as follows:

. . . or read by a fragment shader, will count against this limit. The transformed vertex position (gl_Position) does not count against this limit.

Add the following paragraphs on p.84:

A varying variable is considered active if it is determined by the linker that the varying will actually be used when the executable code in a program object is executed. The linker will make this determination regardless of the transform-feedback state set with the TransformFeedbackVaryingsNV() command. In cases where the linker cannot make a conclusive determination, the varying will be considered active. It is possible to override this determination and force the linker to consider a varying variable as active by calling ActiveVaryingNV(). This can be useful in transform feedback mode if there are varying variables to be recorded but not otherwise needed.

To find the location of an active varying variable, call

```
int GetVaryingLocationNV(uint program, const char *name)
```

This command will return the location of varying variable <name>. <name> is a null-terminated string without whitespace. If <name> is not the name of an active varying variable in <program>, -1 is returned. Locations for both user-defined as well as built-in varying variables can be queried. If <program> has not been successfully linked, the error INVALID_OPERATION is generated. After a program is linked, the location will not change, unless the program is re- linked. A valid name cannot be any portion of a single vector or matrix, but can be a single element of an array or the whole array. Note that varying variables cannot be structures.

To determine the set of active varying variables used by a program object, and their data types, use the command:

```
void GetActiveVaryingNV(uint program, uint index,
                        sizei bufSize, sizei *length, sizei *size,
                        enum *type, char *name);
```

This command provides information about the varying selected by <index>. An <index> of 0 selects the first active varying variable, and an <index> of ACTIVE_VARYINGS_NV-1 selects the last active varying variable. The value of ACTIVE_VARYINGS_NV can be queried with GetProgramiv (see section 6.1.14). If <index> is greater than or equal to

ACTIVE_VARYINGS_NV, the error INVALID_VALUE is generated. The parameter <program> is the name of a program object for which the command LinkProgram has been issued in the past. It is not necessary for <program> to have been linked successfully. The link could have failed because the number of active varying variables exceeded the limit.

The name of the selected varying is returned as a null-terminated string in <name>. The actual number of characters written into <name>, excluding the null terminator, is returned in <length>. If <length> is NULL, no length is returned. The maximum number of characters that may be written into <name>, including the null terminator, is specified by <bufSize>. The returned varying name can be the name of a user defined varying variable or the name of a built- in varying (which begin with the prefix "gl_", see the OpenGL Shading Language specification for a complete list). The length of the longest varying name in program is given by ACTIVE_VARYING_MAX_LENGTH_NV, which can be queried with GetProgramiv (see section 6.1.14).

For the selected varying variable, its type is returned into <type>. The size of the varying is returned into <size>. The value in <size> is in units of the type returned in <type>. The type returned can be any of FLOAT, FLOAT_VEC2, FLOAT_VEC3, FLOAT_VEC4, INT, INT_VEC2, INT_VEC3, INT_VEC4, UNSIGNED_INT, UNSIGNED_INT_VEC2_EXT, UNSIGNED_INT_VEC3_EXT, UNSIGNED_INT_VEC4_EXT, FLOAT_MAT2, FLOAT_MAT3, or FLOAT_MAT4. If an error occurred, the return parameters <length>, <size>, <type> and <name> will be unmodified. This command will return as much information about active varying variables as possible. If no information is available, <length> will be set to zero and <name> will be an empty string. This situation could arise if GetActiveVaryingNV is issued after a failed link.

To force the linker to mark a varying variable as active, call

```
void ActiveVaryingNV(uint program, const char *name)
```

to specify that the varying variable <name> in <program> should be marked as active when the program is next linked. In particular, it does not modify the list of active varying variables in a program object that has already been linked. For any varying variable in <program> not passed to ActiveVaryingNV, the linker will determine their active status. <name> must be a null-terminated string without whitespace. A valid name cannot be an element of an array, or any portion of a single vector or matrix. ActiveVaryingNV may be issued before any shader objects are attached to <program>. Hence, <name> can contain any string, including a name that is never used as a varying variable in any shader object. Such names are ignored by the GL.

The application is advised to force any varying variable live that it needs for transform feedback purposes. The set of active varying variables are linker dependent.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

(Add new section 3.X, Discarding Rasterization)

Primitives can be optionally discarded before rasterization by calling Enable and Disable with RASTERIZER_DISCARD_NV. When enabled, primitives are discared right before the rasterization stage, but after the optional

transform feedback stage. When disabled, primitives are passed through to the rasterization stage to be processed normally. RASTERIZER_DISCARD_NV applies to the DrawPixels, CopyPixels, Bitmap, Clear and Accum commands as well.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

(Replace section 4.1.7, "Occlusion Queries", p. 204, with the following)

Occlusion queries use query objects to track the number of fragments or samples that pass the depth test. An occlusion query can be started and finished by calling BeginQuery and EndQuery, respectively, with a <target> of SAMPLES_PASSED.

When an occlusion query starts, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of SAMPLE BUFFERS is 0, then the samples- passed count is incremented by 1 for each fragment. If the value of SAMPLE BUFFERS is 1, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of SAMPLES if any sample in the fragment is covered. When an occlusion query finishes and all fragments generated by the commands issued prior to EndQuery have been generated, the samples-passed count is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

If the samples-passed count overflows, (i.e., exceeds the value 2^n - 1, where n is the number of bits in the samples-passed count), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at 2^n - 1 and incrementing no further.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

(Add to section 5.4, Display Lists p. 237)

On p. 241, add the following to the list of vertex buffer object commands not compiled into a display list: BindBufferRangeNV, BindBufferOffsetNV, BindBufferBaseNV, TransformFeedbackAttribsNV, TransformFeedbackVaryingsNV, and ActiveVaryingNV.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify the second paragraph of section 6.1.1 (Simple Queries) p244 to read as follows:

...<data> is a pointer to a scalar or array of the indicated type in which to place the returned data. The commands

```
void GetIntegerIndexedvEXT(enum param, uint index, int *values);
void GetBooleanIndexedvEXT(enum param, uint index, boolean *values);
```

are used to query indexed state. <target> is the name of the indexed

state and <index> is the index of the particular element being queried. <data> is a pointer to a scalar or array of the indicated type in which to place the returned data. In addition ...

(Replace Section 6.1.12, Occlusion Queries, p. 254)

Section 6.1.12, Asynchronous Queries

The command

```
boolean IsQuery(uint id);
```

returns TRUE if <id> is the name of a query object. If <id> is zero, or if <id> is a non-zero value that is not the name of a query object, IsQuery returns FALSE.

Information about a query target can be queried with the command

```
void GetQueryiv(enum target, enum pname, int *params);
```

<target> identifies the query target and can be SAMPLES_PASSED for occlusion queries or PRIMITIVES_GENERATED_NV and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV for primitive queries.

If <pname> is CURRENT_QUERY, the name of the currently active query for <target>, or zero if no query is active, will be placed in <params>.

If <pname> is QUERY_COUNTER_BITS, the implementation-dependent number of bits used to hold the query result for <target> will be placed in params. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (PRIMITIVES_GENERATED_NV and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV) if the number of bits is non-zero, the minimum number of bits allowed is 32.

For occlusion queries (SAMPLES_PASSED), if the number of bits is non-zero, the minimum number of bits allowed is a function of the implementation's maximum viewport dimensions (MAX_VIEWPORT_DIMS). The counter must be able to represent at least two overdraws for every pixel in the viewport. The formula to compute the allowable minimum value (where n is the minimum number of bits) is:

```
n = min(32, ceil(log_2(maxViewportWidth *
                       maxViewportHeight * 2))).
```

The state of a query object can be queried with the commands

```
void GetQueryObjectiv(uint id, enum pname, int *params);
void GetQueryObjectuiv(uint id, enum pname, uint *params);
```

If <id> is not the name of a query object, or if the query object named by <id> is currently active, then an INVALID_OPERATION error is generated.

If <pname> is QUERY_RESULT, then the query object's result value is returned as a single integer in <params>. If the value is so large in magnitude that it cannot be represented with the requested type, then the

nearest value representable using the requested type is returned. If the number of query counter bits for any <target> is zero, then the result is returned as a single integer with a value of 0.

There may be an indeterminate delay before the above query returns. If <pname> is QUERY_RESULT_AVAILABLE, FALSE is returned if such a delay would be required, TRUE is returned otherwise. It must always be true that if any query object returns a result available of TRUE, all queries of the same type issued prior to that query must also return TRUE.

Querying the state for any given query object forces the corresponding query to complete within a finite amount of time.

If multiple queries are issued using the same object name prior to calling GetQueryObject[u]iv, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same <target> and <id>.

(Add to Section 6.1.13, Buffer Objects, p. 255)

Add the following paragraph to the bottom of this section, p. 256.

To query which buffer objects are the target(s) when transform feedback is active, call GetIntegerIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_BUFFER_BINDING_NV. <index> has to be in the range 0 to MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV - 1, otherwise the error INVALID_VALUE is generated. The name of the buffer object bound to <index> is returned in <values>. If no buffer object is bound for <index>, zero is returned in <values>.

To query the starting offset or size of the range of each buffer object binding used for transform feedback, call GetIntegerIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_BUFFER_START_NV or TRANSFORM_FEEDBACK_BUFFER_SIZE_NV respectively. The error INVALID_VALUE is generated if <index> not in the range 0 to MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV - 1. If the parameter (starting offset or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to <index>, -1 is returned.

(Add a new Section 6.1.14 "Transform Feedback " and rename 6.1.14 to 6.1.15)

To query the attributes to stream to a buffer object when neither an OpenGL Shading Language vertex nor geometry shader is active, call GetIntegerIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_RECORD_NV. This will return three values in <values> for each <index>. The first value returned is the attribute. The second value the number of components of the attribute, and the third value the index of the attribute, if applicable. If the attribute is not indexed, the third component will return 0. The parameter <index> has to be in the range 0 to TRANSFORM_FEEDBACK_ATTRIBS_NV - 1, otherwise the error INVALID_VALUE is generated. If no data exists for <index> 0 is returned three times in <values>.

To query the attributes to stream to a buffer object when a vertex and/or geometry shader is active, use the command GetTransformFeedbackVaryingNV(), as explained in section 6.1.14.

(add to Section 6.1.14, Shader and Program Queries, p. 256)

Add the following paragraph to the bottom of page 257:

If <pname> is TRANSFORM_FEEDBACK_BUFFER_MODE_NV, the buffer mode, used when transform feedback is active, is returned. It can be one of SEPARATE_ATTRIBS_NV or INTERLEAVED_ATTRIBS_NV. If <pname> is TRANSFORM_FEEDBACK_VARINGS_NV, the number of varying variables to stream to one, or more, buffer objects are returned. If <pname> is ACTIVE_VARYINGS_NV, the number of active varying variables is returned. If no active varyings exist, 0 is returned. If <pname> is ACTIVE_VARYINGS_MAX_LENGTH_NV, the length of the longest active varying name, including a null terminator, is returned. If no active varying variable exists, 0 is returned.

The command

```
  void GetTransformFeedbackVaryingNV(uint program, uint index,
                                     int *location)
```

returns, for each <index>, the location of a varying variable to stream to a buffer object in <location>. The <index> element of the array <locations>, as passed to TransformFeedbackVaryingsNV, is returned. <index> has to be in the program object specific range 0 to TRANSFORM_FEEDBACK_VARYINGS_NV - 1, otherwise the error INVALID_VALUE is generated. If no location exists for <index>, -1 is returned. If <program> is not the name of a program object, or if program object has not been linked successfully, the error INVALID_OPERATION is generated.

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with EXT_timer_query

EXT_timer_query is the first extension to generalize the BeginQuery and EndQuery mechanism introduced by ARB_occlusion_query and OpenGL 1.5 to cover an additional query type. This extension is the second. This extension is written against the OpenGL 2.0 specification and uses most of the modifications in the EXT_timer_query specification. If EXT_timer_query is supported, timer queries need to be added as a third query type.

Dependencies on NV_geometry_program4 and EXT_geometry_shader4

If NV_geometry_program4 is not supported, delete the reference to the output primitive type in Section 2.Y. Delete the reference to PRIMITIVE_ID_NV and LAYER_NV.

If EXT_geometry_shader4 is not supported, delete any reference to a geometry shader.

Dependencies on NV_vertex_program4 and NV_gpu_program4

If NV_vertex_program4 is not supported, delete any reference to VERTEX_ID_NV. If NV_gpu_program4 is not supported, table X.2 needs to refer to the "result" variables defined in the ARB_vertex_program specification instead.

Interactions with ARB_shader_objects and OpenGL 2.0

If neither ARB_shader_objects nor OpenGL 2.0 is supported, all references to shader and program objects, as well as varying variables, should be removed. This also means that functions including TransformFeedbackVaryingsNV, GetVaryingLocationNV, GetActiveVaryingNV, ActiveVaryingNV, and GetTransformFeedbackVaryingNV will not be supported, and enums that are relevant only in the context of shader and program objects will not be accepted.

Errors

The error INVALID_OPERATION is generated by BeginQuery if called with an <id> of zero, if the active query object name for <target> is non- zero, or if <id> is the active query object name for any query type.

The error INVALID_OPERATION is generated by EndQuery if the active query object name for <target> is zero.

The error INVALID_OPERATION is generated if Begin, or any command that performs an explicit Begin, is called when:

* A geometry program or shader is not active AND the begin mode does not match the allowed begin modes for the current transform feedback state as given by table X.1.

* A geometry program or shader is active AND the output primitive type (of the geometry program / shader) does not match the allowed begin modes for the current transform feedback state as given by table X.1.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV if there is no buffer object bound to index 0 in INTERLEAVED_ATTRIBS_NV mode.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV if the number of buffer objects bound in SEPARATE_ATTRIBS_NV mode is less than the number of buffer objects required, as given by the current transform feedback state.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV if no attributes are specified to be captured.

The error INVALID_OPERATION is generated by BeginTransformFeedbackNV, TransformFeedbackBufferNV, TransformFeedbackVaryingsNV, TransformFeedbackAttribsNV, or UseProgram or LinkProgram, called on the currently in use program object, while transform feedback is active.

The error INVALID_OPERATION is generated by EndTransformFeedbackNV while transform feedback is inactive.

The error INVALID_OPERATION is generated by BindBufferRangeNV, BindBufferOffsetNV or BindBufferBaseNV if <index> is greater or equal than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV.

The error INVALID_VALUE is generated by BindBufferRangeNV if the value of <size> <= 0.

The error INVALID_VALUE is generated by BindBufferRangeNV or BindBufferOffsetNV if <start> or <end> are not word-aligned.

The error INVALID_OPERATION is generated when any of the BindBuffer* commands is called while transform feedback is active.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV commands if any location appears more than once in the array <locations.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV if any location in <locations> references a non-existing varying variable.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV if <program> has not been linked successfully.

The error INVALID_OPERATION is generated by TransformFeedbackVaryingsNV in INTERLEAVED_ATTRIBS_NV mode if the total number of components of all varying variables specified in the array <locations> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

The error INVALID_VALUE is generated by TransformFeedbackVaryingsNV or TransformFeedbackAttribsNV in SEPARATE_ATTRIBS_NV mode if <count> is greater than MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV.

The error INVALID_VALUE is generated by TransformFeedbackVaryingsNV in SEPARATE_ATTRIBS_NV mode if the number of components for each varying variable in the array <locations> is greater than MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS_NV.

The error INVALID_VALUE is generated by TransformFeedbackAttribsNV in INTERLEAVED_ATTRIBS_NV mode if the sum of the values of the components of the attributes in the array <attribs> is greater than MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS_NV.

The error INVALID_OPERATION is generated by TransformFeedbackAttribsNV if an enum value is specified more than once in the array <attribs>.

The error INVALID_OPERATION is generated by TransformFeedbackAttribsNV if the number of components for each attribute in the array <attribs> is outside the range [0,4].

The error INVALID_VALUE is generated by TransformFeedbackAttribsNV if the index value is in the array <attribs> is outside the allowable range for an attribute enumerant corresponding to more than one real attribute.

The error INVALID_OPERATION is generated by GetVaryingLocationNV if <program> is not the name of a program object or if <program> has not been linked successfully.

The error INVALID_OPERATION is generated by GetActiveVaryingNV or ActiveVaryingNV if <program> is not the name of a program object.

The error INVALID_VALUE is generated by GetActiveVaryingNV if <index> is greater than or equal to ACTIVE_VARYINGS_NV.

The error INVALID_VALUE is generated by GetIntegerIndexedvEXT() or GetBooleanIndexedv() with <param> set to TRANSFORM_FEEDBACK_RECORD_NV if <index> is greater than or equal to TRANSFORM_FEEDBACK_ATTRIBS_NV.

The error INVALID_VALUE is generated by GetIntegerIndexedvEXT() or GetBooleanIndexedvEXT() with <param> set to TRANSFORM_FEEDBACK_BUFFER_BINDING_NV if <index> is greater than or equal to MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS_NV.

The error INVALID_VALUE is generated by GetTransformFeedbackVaryingsNV if <index> is greater than the program object specific value TRANSFORM_FEEDBACK_VARYINGS_NV - 1.

The error INVALID_OPERATION is generated by GetTransformFeedbackVaryingsNV if <program> is not the name of a program object, or if program object has not been linked successfully.

New State

(Add a new table: Table 6.X, Transform Feedback State)

Get Value	Type	Get Command	Init. Value	Description	Sec	Attrib
TRANSFORM_FEEDBACK_BUFFER_MODE_NV	Z2	GetIntegerv	INTERLEAVED_ATTRIBS_NV	Transform feedback mode	2.Y	-
TRANSFORM_FEEDBACK_ATTRIBS_NV	Z2	GetIntegerv	0	Number of attributes to capture in transform feedback mode	2.Y	-
TRANSFORM_FEEDBACK_BUFFER_BINDING_NV	Z+	GetIntegerv	0	Buffer object bound to generic bind point for transform feedback.	6.1.13	-
TRANSFORM_FEEDBACK_RECORD_NV	nx3*Z+	GetIntegerIndexedvEXT	0	Name, component count, and index of each attribute captured	6.1.14	-
TRANSFORM_FEEDBACK_BUFFER_BINDING_NV	nxZ+	GetIntegerIndexedvEXT	0	Buffer object bound to each transform feedback attribute stream.	6.1.13	-
TRANSFORM_FEEDBACK_BUFFER_START_NV	nxZ+	GetIntegerIndexedvEXT	0	Start offset of binding range for each transform feedback attrib. stream	6.1.13	-
TRANSFORM_FEEDBACK_BUFFER_SIZE_NV	nxZ+	GetIntegerIndexedvEXT	0	Size of binding range for each transform feedback attrib. stream	6.1.13	-

(Modify Table 6.37, p 298, updating the query object state to cover transform feedback.)

Get Value	Type	Get Command	Init. Value	Description	Sec	Attribute
CURRENT_QUERY	3xZ+	GetQueryiv	0	Active query object name (occlusion, timer, xform feedback)	2.X	-
QUERY_RESULT	3xZ+	GetQueryObjectiv	0	Query object result (samples passed, Time elapsed, feedback data amount)	2.X	-
QUERY_RESULT_AVAILABLE	3xZ+	GetQueryObjectiv	TRUE	Query object result available?	2.X	-

(Modify Table 6.29, p. 290, Program Object State. Add the following state.)

Get Value	Type	Get Command	Init. Value	Description	Sec	Attribute
ACTIVE_VARYINGS_NV	Z+	GetProgramiv	0	Number of active varyings	2.15.3	-
ACTIVE_VARYING_MAX_LENGTH_NV	Z+	GetProgramiv	0	Maximum active varying name length	2.15.3	-
TRANSFORM_FEEDBACK_BUFFER_MODE_NV	Z2	GetProgramiv	INTERLEAVED_ATTRIBS_NV	Transform feedback mode for the program	6.1.14	-
TRANSFORM_FEEDBACK_VARYINGS_NV	Z+	GetProgramiv	0	Number of varyings to stream to buffer object(s)	6.1.14	-
-	nxZ+	GetVarying-LocationNV	-	Location of each active varying variable	2.15.3	-
-	Z+	GetActive-VaryingNV	-	Size of each active varying variable	2.15.3	-
-	Z+	GetActive-VaryingNV	-	Type of each active varying variable	2.15.3	-
-	0+x-char	GetActive-VaryingNV	-	Name of each active varying variable	2.15.3	-
-	Z+	GetTransform-Feedback-VaryingNV	-	Varying location for one of the multiple varyings to capture	6.1.14	-

New Implementation Dependent State

(Modify Table 6.34, p. 295. Update the query object state to cover transform feedback.)

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
QUERY_COUNTER_BITS	2xZ+	GetQueryiv	see 6.1.12	Asynchronous query counter bits (occlusion, timer, tranform feedback queries)	6.1.12	-

(Add a new table, Table 6.X. Transform Feedback State.)

NOTE: In the "GetValue" columns below, MXFB stands for "MAX_TRANSFORM_FEEDBACK".

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MXFB_INTERLEAVED_COMPONENTS_NV	Z+	GetIntegerv	64	Max number of components to write to a single buffer in interleaved mode	2.Y	-
MXFB_SEPARATE_ATTRIBS_NV	Z+	GetIntegerv	4	Max number of separate attributes or vayings that can be captured in transform feedback	2.Y	-
MXFB_SEPARATE COMPONENTS_NV	Z+	GetIntegerv	16	Max number of components per attribute or varying in separate mode	2.Y	-

Issues

1. How does transform feedback differ from core GL feedback?

* Transform feedback writes vertex data to buffer objects, which allows the data returned to be used directly by vertex pulling. GL feedback mode writes vertex data to a buffer in system memory.

* Transform feedback is done after transformation, but prior to clipping. The primitives returned contain the original transformed vertices produced by vertex or geometry program execution, and does not contain any primitives inserted by clipping.

* Transform feedback supports only a single basic output primitive type (points, lines, or triangles), while core GL feedback mode supports all primitive types. Since only one primitive type is supported, the data returned does not contain tokens describing each primitive being fed back. Primitive tokens make the data returned by GL feedback mode irregular and unsuitable for vertex pulling.

2. What should this extension be called?

RESOLVED: The current name is "NV_transform_feedback", playing off the fact that it is transformed primitives that are handled and the similarities to GL feedback mode.

3. What happens if you bind a buffer for transform feedback that is currently bound for other purposes? Should we somehow detect this case and produce an error?

!!! NBC I feel strongly that we should follow the precedent for Map/Unmap. The reason that MapBuffer and UnmapBuffer are a precedent here is because while a buffer object is in the mapped state, no GL commands are allowed to operate on the buffer object's data. So by analogy, while a buffer is being used for transform feedback, no other GL commands should be allowed to operate on the buffer object's data. This includes initiating any rendering which would cause the GL to source data from an active transform feedback buffer object.

UNRESOLVED

4. *Should this extension include any new buffer object binding targets, or should it overload ARRAY_BUFFER, or should we skip the binding target altogether in favor of a buffer object name accepted directly by the new GL commands?*

   RESOLVED: There are new binding points for XFB along with a new API (BindBufferBase etc) to set the internal binding points. A new binding point, TRANSFORM_FEEDBACK_BUFFER_NV is also introduced.

5. *Previous buffer object extensions provided a way to have existing GL commands reference a buffer object instead of a user-supplied buffer. Should the new commands introduced here allow referencing a user-supplied buffer in addition to a buffer object?*

   RESOLVED: No. A program can get the contents of the feedback buffer back to the CPU using MapBuffer and GetBufferSubData

6. *Is BeginTransformFeedback really necessary? Could the query just initiate the transform feedback mode?*

   RESOLUTION: Using BeginTransformFeedback and EndTransformFeedback gives a clean place to spec all of the transform-feedback-specific issues without cluttering up the query language. Also, the queries don't have to be done at the same time as beginning and ending the feedback process.

7. *What usage enums should be provided to glBufferData for use in conjunction with transform feedback?*

   RESOLVED: STREAM_COPY or STREAM_READ are expected to be the most common usages. If a buffer object is being written by the GL through transform feedback, and the contents of the buffer object are subsequently being consumed by the GL (e.g. by being used as a vertex buffer object), then this is a *_COPY usage. If the buffer object is being written by the GL through transform feedback, but is being consumed by the application (e.g. being mapped for read), this is a *_READ usage. The temporal (STREAM, STATIC, or DYNAMIC) component of the usage enum is determined by the ratio between how often the contents of the buffer object are modified and how often operations that source data from the buffer object occur.

8. *What should the behavior be when a buffer object is the active target of transform feedback, and it is deleted via DeleteBuffers?*

   RESOLVED: Deletion is deferred until the EndTransformFeedback if transform feedback is active.

9. *Should we allow more buffers to be bound than are used?*

   RESOLVED: Yes. The extra buffers are not in the way and can stay bound.

10. *Should we allow feedback to buffer lists with holes (i.e. 0 and 2 bound)?*

    RESOLVED: No. This makes for an ugly API with the potential for bugs, without any real benefit. The application can as well bind all buffers

needed to incremented indices. It is an invalid operation to not have a buffer bound where one is required.

11. Why only one feedback primitive mode per feedback invocation?

RESOLVED: Having primitive tokens breaks up the stream and makes it less amenable to being read back in as a vertex buffer. Also, mixing multiple primitive types makes the counting of primitives less clear for the application.

12. Is RasterPos fed back?

RESOLVED: No.

13. Is DrawPixels/CopyPixels/Bitmap fed back?

RESOLVED: No. Rasterization occurs as normal, but there is no output to the feedback buffer. This is consistent with taking a tap out of the pipe before clipping.

14. Why do we need new BindBuffer functions?*

RESOLVED: All previous buffer object extensions have been retrofits of existing pointer-based APIs. New extensions built assuming buffer objects don't have that history, so need a new API. The functionality of these new functions combines the functionality of BindBuffer, to set the external bind point used by calls like MapBuffer and BufferSubData, with the functionality to set an internal bind point like VertexAttribPointer does.

15. How do the transform feedback indices, passed to the BindBuffer commands, work with multiple bindings?*

RESOLVED: The same way that they work with vertex arrays. There is one external bind point, TRANSFORM_FEEDBACK_BUFFER_NV. There are n internal bind points, selected with the <index> parameter to the BindBuffer* commands, where n is some implementation dependent limit. The BindBuffer* commands take the buffer passed and bind it to the external bind point, as well as to the selected internal bind point.

For example:

```
BindBufferOffsetNV(TRANSFORM_FEEDBACK_BUFFER_NV, 0, 1, 12);
// XFB index 0 points at buffer 1 with offset 12

BindBuffer(TRANSFORM_FEEDBACK_BUFFER_NV, 2);
// Buffer 2 is now bound to the external bind point. XFB index 0 still
// points at buffer 1

MapBuffer(TRANSFORM_FEEDBACK_BUFFER_NV, ...);
// Maps buffer 2
```

16. How are quads/quadstrips/polygons tesselated into triangles?

RESOLVED: In an implementation-dependent manner. OpenGL doesn't define quads or polygons in terms of triangles, so there is no one correct way to do it, and different gpus may implement the behavior differently. A

quad may be split into two triangles in several different ways, and an application may not rely on this behavior.

17. How does this extension interact with display lists?

RESOLVED: Just like the VBO extension, none of the BindBuffer* commands are compiled into a display list.

18. Does polygon mode state affect the logic that determines if the transform feed back primitive mode and the render mode states are valid at the start of transform feedback mode?

RESOLVED: PolygonMode has no influence on the BeginTransFormFeedback primitiveMode check since it is performed later, in raster.

19. What to do with incomplete primitives?

RESOLVED: If there is no room to store one or more vertices of a primitive in a buffer object, none of the vertices in that primitive are written to the buffer. If a partial primitive enters transform feedback (i.e. only two vertices sent in triangles mode), none of the vertices in that primitive are written to the buffer object.

20. Why does TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV have a TRANSFORM_FEEDBACK prefix but PRIMITIVES_GENERATED_NV doesn't?

RESOLVED: The number of primitives generated is independent of any feedback that is active. The number of primitives that are written is only valid for transform feedback - another extension could conceivably have a different way of writing out primitives that would require a similar but distinct token.

21. When a GLSL vertex shader is active, what happens in transform feedback mode if non-active varying variables are specified?

DISCUSSION: Active varying variables are varying variables, declared in the shader, that the linker determined are actually needed. As an optimization, the linker can discard the ones declared, but not needed. If non-active varying variables need to be fed into a buffer object, the linker should not perform this optimization.

There are three suggested resolutions to this problem:

1. The set of varying variables that need to be streamed to a buffer object in transform feedback mode are set as a property of the program object, and are taken into account during the link step. This means that changing the set means the application will have to re-link the program object in order to have the change take effect.

2. The set of varying variables that need to be streamed to a buffer object in transform feedback mode are specified after the program object has been linked. This is the most flexible option from the applications perspective, but this might mean that a) specifying this set could force the GL to re-link 'under the covers', and b) could mean that the GL runs out of varying variable slots because the combined total of the set of active varyings and the varyings to stream in transform feedback mode is too large.

3. This solution is a hybrid of the above two approaches. The set of potential varying variables that need to be streamed to a buffer object are set as a property of the program object. These varying variables are marked as active by the application and therefore cannot be eliminated during the link step. However, a sub-set of varying variables to actually stream to a buffer object can be changed without the application having to re-link the program object. This approach gives the application flexibility to change the set of varying variables to stream, while it eliminates the need for the GL to compile 'under the covers'.

RESOLUTION: Option 3 offers a good compromise, and therefore we'll go with that.

22. Given option 3 in the previous resolution, how to specify that a varying variable has to be considered active by the linker?

DISCUSSION: There are two approaches to the application specifying which varying variables are active. We can either provide a simple flag that specifies that all varying variables are considered active, or we can provide a more complex mechanism where the application can specify an individual varying variable as being active.

RESOLUTION: RESOLVED. The 'all or nothing' flag is a simple idea, but has a drawback when used with a 'uber-shader' that implements many paths to achieve an effect, but only one path is used during any run of the shader. In this case, a lot more varying variables might be flagged as active then really is necessary, running the risk of running out of resources. Therefore, we'll provide a mechanism for the application to specify on a per varying variable basis if it is active.

23. Given the discussion in the previous issues, should a GetActiveVarying() command be added, modeled after the existing getActiveUniform() command?

DISCUSSION: Such a command will return the list of active uniforms, after the program object has been linked. As per issue 22's resolution, the complete set of varying variables that could be streamed to a buffer object needs to be specified before the program object is linked.

It can be useful to an application to stream out a subset of the active varying variables or to find out the whole set of active varyings, especially since the set can be implementation dependent.

RESOLUTION: YES.

24. What is proper use of the command ActiveVaryingNV()?

RESOLVED: The application is well advised to force any varying variable live that it needs for transform feedback purposes. The set of active varying variables are linker dependent. For example, if a program object has no fragment shader, then the LinkProgram command cannot typically determine which built-in varying variables, output by a geometry or vertex shader, are active. This is because the fragment processing state can change, and therefore such a determination cannot be made until a render command is issued. Furthermore, any user-defined varyings are

likely to be marked as non-active if there is no fragment shader because they are guaranteed to have no effect on fixed-function fragment processing. If there is both a vertex (or geometry) and fragment shader in a program object, the application can probably deduce what will be an active varying variable, or not. But beware of any (static) flow-control that the linker can use to do cross vertex- fragment optimization to cull any varying variables.

25. Are primitives sent down the pipeline after transform feedback, or discarded?

RESOLVED: Primitives can be optionally discarded before rasterization by calling Enable and Disable with RASTERIZER_DISCARD_NV. When enabled, primitives are discarded after vertex attributes are recorded into the buffer objects bound to transform feedback. When disabled, primitives are passed through to the rasterization stage to be clipped and rasterized normally. All rasterization operations are discarded, not just those that are fed back into the buffer.

This applies to DrawPixels, CopyPixels, Bitmap, Clear, Accum as well.

26. If a varying is declared as an array, is the whole array streamed out?

RESOLVED: No, the application has to specify which elements of an array it wants to stream out. Implementations might not be able to stream out a large number of components to a single buffer object. If that is the case, the application can stream each element of an array to a different buffer object in TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS mode.

27. Is it possible to capture attributes when using the fixed-function pipeline?

RESOLVED: Yes, there is nothing that precludes this. The application is responsible for sending down the needed vertex attributes and setting the GL state, as desired, for the attributes it wants to stream to a buffer object. Note that VERTEX_ID is not defined in fixed-function.

28. Is it possible to record hardware-generated primitive ID values that would be available to a pixel shader?

RESOLVED: Transform feedback can only record the primitive ID values emitted per-vertex by a geometry shader or program. While each primitive recorded for transform-feedback has a well-defined primitive ID, transform feedback is only capable of recording the attributes of individual vertices.

29. Does transform feedback support the ability to capture per-vertex layer outputs, as provided by EXT_geometry_shader4 and NV_geometry_program4?

RESOLVED: Yes. For GLSL shaders, it is sufficient to reference the built-in varying "gl_Layer". For assembly geometry programs, the original version of the spec did not provide an enum allowing you to name "result.layer" in TransformFeedbackAttribsNV. This was an oversight in the original spec, which was fixed by version 14. An updated driver will be required to take advantage of this capability; NVIDIA drivers supporting this extension published prior to February

2008 will not be able to capture "result.layer". The value captured for LAYER_NV will be undefined unless a geometry program that writes "result.layer" is active.

Revision History

Rev.	Date	Author	Changes
14	02/04/08	pbrown	Fixed a problem with the spec where we were unable to record "result.layer" using the assembly interface. Added a new enum to address.

Name

NV_vertex_program4

Name Strings

(none)

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Status

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 10/06/06
NVIDIA Revision: 5

Number

325

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 2.0 specification.

ARB_vertex_program is required.

NV_gpu_program4 is required. This extension is supported if "GL_NV_gpu_program4" is found in the extension string.

NVX_instanced_arrays affects the definition of this extension.

Overview

This extension builds on the common assembly instruction set infrastructure provided by NV_gpu_program4, adding vertex program-specific features.

This extension provides the ability to specify integer vertex attributes that are passed to vertex programs using integer data types, rather than being converted to floating-point values as in existing vertex attribute functions. The set of input and output bindings provided includes all bindings supported by ARB_vertex_program. This extension provides additional input bindings identifying the index of the vertex when vertex arrays are used ("vertex.id") and the instance number when instanced arrays are used ("vertex.instance", requires EXT_draw_instanced). It also provides output bindings allowing vertex programs to directly specify clip distances (for user clipping) plus a set of generic attributes that allow programs to pass a greater number of attributes to subsequent pipeline stages than is possible using only the pre-defined fixed-function vertex outputs.

```
    By and large, programs written to ARB_vertex_program can be ported
    directly by simply changing the program header from "!!ARBvp1.0" to
    "!!NVvp4.0", and then modifying instructions to take advantage of the
    expanded feature set.  There are a small number of areas where this
    extension is not a functional superset of previous vertex program
    extensions, which are documented in the NV_gpu_program4 specification.
```

New Procedures and Functions

```
    void VertexAttribI1iEXT(uint index, int x);
    void VertexAttribI2iEXT(uint index, int x, int y);
    void VertexAttribI3iEXT(uint index, int x, int y, int z);
    void VertexAttribI4iEXT(uint index, int x, int y, int z, int w);

    void VertexAttribI1uiEXT(uint index, uint x);
    void VertexAttribI2uiEXT(uint index, uint x, uint y);
    void VertexAttribI3uiEXT(uint index, uint x, uint y, uint z);
    void VertexAttribI4uiEXT(uint index, uint x, uint y, uint z, uint w);

    void VertexAttribI1ivEXT(uint index, const int *v);
    void VertexAttribI2ivEXT(uint index, const int *v);
    void VertexAttribI3ivEXT(uint index, const int *v);
    void VertexAttribI4ivEXT(uint index, const int *v);

    void VertexAttribI1uivEXT(uint index, const uint *v);
    void VertexAttribI2uivEXT(uint index, const uint *v);
    void VertexAttribI3uivEXT(uint index, const uint *v);
    void VertexAttribI4uivEXT(uint index, const uint *v);

    void VertexAttribI4bvEXT(uint index, const byte *v);
    void VertexAttribI4svEXT(uint index, const short *v);
    void VertexAttribI4ubvEXT(uint index, const ubyte *v);
    void VertexAttribI4usvEXT(uint index, const ushort *v);

    void VertexAttribIPointerEXT(uint index, int size, enum type,
                                 sizei stride, const void *pointer);

    void GetVertexAttribIivEXT(uint index, enum pname, int *params);
    void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);

    (note:  all these functions are shared with the EXT_gpu_shader4
    extension.)
```

New Tokens

```
    Accepted by the <pname> parameters of GetVertexAttribdv,
    GetVertexAttribfv, GetVertexAttribiv, GetVertexAttribIivEXT, and
    GetVertexAttribIuivEXT:

      VERTEX_ATTRIB_ARRAY_INTEGER_EXT                   0x88FD

    (note:  this token is shared with the EXT_gpu_shader4 extension.)
```

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.7 (Vertex Specification), p.20

(insert before last paragraph, p.22) The commands

```
  void VertexAttribI[1234]{i,ui}EXT(uint index, T values);
  void VertexAttribI[1234]{i,ui}vEXT(uint index, T values);
  void VertexAttribI4{b,s,ub,us}vEXT(uint index, T values);
```

specify fixed-point coordinates that are not converted to floating-point values, but instead are represented as signed or unsigned integer values. Vertex programs that use integer instructions may read these attributes using integer data types. A vertex program that attempts to read a vertex attribute as a float will get undefined results if the attribute was specified as an integer, and vice versa.

(modify second paragraph, p.23) Setting generic vertex attribute zero specifies a vertex; the four vertex coordinates are taken from the values of attribute zero. A Vertex2, Vertex3, or Vertex4 command is completely equivalent to the corresponding VertexAttrib* or VertexAttribI* command with an index of zero. ...

(insert at end of function list, p.24)

```
  void VertexAttribIPointerEXT(uint index, int size, enum type,
                               sizei stride, const void *pointer);
```

(modify last paragraph, p.24) The <index> parameter in the VertexAttribPointer and VertexAttribIPointerEXT commands identify the generic vertex attribute array being described. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS. Generic attribute arrays with integer <type> arguments can be handled in one of three ways: converted to float by normalizing to [0,1] or [-1,1] as specified in table 2.9, converted directly to float, or left as integer values. Data for an array specified by VertexAttribPointer will be converted to floating-point by normalizing if the <normalized> parameter is TRUE, and converted directly to floating-point otherwise. Data for an array specified by VertexAttribIPointerEXT will always be left as integer values.

(modify Table 2.4, p. 25)

Command	Sizes	Integer Handling	Types
VertexPointer	2,3,4	cast	...
NormalPointer	3	normalize	...
ColorPointer	3,4	normalize	...
SecondaryColorPointer	3	normalize	...
IndexPointer	1	cast	...
FogCoordPointer	1	n/a	...
TexCoordPointer	1,2,3,4	cast	...
EdgeFlagPointer	1	integer	...
VertexAttribPointer	1,2,3,4	flag	...
VertexAttribIPointerEXT	1,2,3,4	integer	byte, ubyte, short, ushort, int, uint

Table 2.4: Vertex array sizes (values per vertex) and data types. The "integer handling" column indicates how fixed-point data types are handled: "cast" means that they converted to floating-point directly, "normalize" means that they are converted to floating-point by normalizing to [0,1] (for unsigned types) or [-1,1] (for signed types), "integer" means that they remain as integer values, and "flag" means that either "cast" or "normalized" applies, depending on the setting of the <normalized> flag in VertexAttribPointer.

(modify end of pseudo-code, pp. 27-28)

```
  for (j = 1; j < genericAttributes; j++) {
    if (generic vertex attribute j array enabled) {
      if (generic vertex attribute j array is a pure integer array) {
        VertexAttribI[size][type]vEXT(j, generic vertex attribute j
                                      array element i);
      } else if (generic vertex attribute j array normalization flag
                 is set and <type> is not FLOAT or DOUBLE) {
        VertexAttrib[size]N[type]v(j, generic vertex attribute j
                                    array element i);
      } else {
        VertexAttrib[size][type]v(j, generic vertex attribute j
                                    array element i);
      }
    }
  }

  if (generic vertex attribute 0 array enabled) {
    if (generic vertex attribute 0 array is a pure integer array) {
      VertexAttribI[size][type]vEXT(0, generic vertex attribute 0
                                     array element i);
    } else if (generic vertex attribute 0 array normalization flag
               is set and <type> is not FLOAT or DOUBLE) {
      VertexAttrib[size]N[type]v(0, generic vertex attribute 0
                                  array element i);
    } else {
      VertexAttrib[size][type]v(0, generic vertex attribute 0
                                  array element i);
    }
  }
```

Modify Section 2.X, GPU Programs

(insert after second paragraph)

Vertex Programs

Vertex programs are used to compute the transformed attributes of a vertex, in lieu of the set of fixed-function operations described in sections 2.10 through 2.13. Vertex programs are run on a single vertex at a time, and the state of neighboring vertices is not available. The inputs available to a vertex program are the vertex attributes described in section 2.7. The results of the program are the attributes of a transformed vertex, which include (among other things) a transformed position, colors, and texture coordinates. The vertices transformed by a vertex program are then processed normally by the remainder of the GL pipeline.

Modify Section 2.X.2, Program Grammar

(replace third paragraph)

Vertex programs are required to begin with the header string "!!NVvp4.0". This header string identifies the subsequent program body as being a vertex program and indicates that it should be parsed according to the base NV_gpu_program4 grammar plus the additions below. Program string parsing begins with the character immediately following the header string.

(add the following grammar rules to the NV_gpu_program4 base grammar)

```
<resultUseW>            ::= <resultVarName> <arrayMem> <optWriteMask>
                          | <resultColor> <optWriteMask>
                          | <resultColor> "." <colorType> <optWriteMask>
                          | <resultColor> "." <faceType> <optWriteMask>
                          | <resultColor> "." <faceType> "." <colorType>
                            "." <optWriteMask>

<resultUseD>            ::= <resultColor>
                          | <resultColor> "." <colorType>
                          | <resultColor> "." <faceType>
                          | <resultColor> "." <faceType> "." <colorType>
                          | <resultMulti>

<attribBasic>           ::= <vtxPrefix> "position"
                          | <vtxPrefix> "weight" <optArrayMemAbs>
                          | <vtxPrefix> "normal"
                          | <vtxPrefix> "fogcoord"
                          | <attribTexCoord> <optArrayMemAbs>
                          | <attribGeneric> <arrayMemAbs>
                          | <vtxPrefix> "id"
                          | <vtxPrefix> "instance"

<attribColor>           ::= <vtxPrefix> "color"

<attribMulti>           ::= <attribTexCoord> <arrayRange>
                          | <attribGeneric> <arrayRange>
```

```
<attribTexCoord>        ::= <vtxPrefix> "texcoord"

<attribGeneric>         ::= <vtxPrefix> "attrib"

<vtxPrefix>             ::= "vertex" "."

<resultBasic>           ::= <resPrefix> "position"
                          | <resPrefix> "fogcoord"
                          | <resPrefix> "pointsize"
                          | <resultTexCoord> <optArrayMemAbs>
                          | <resultClip> <arrayMemAbs>
                          | <resultGeneric> <arrayMemAbs>
                          | <resPrefix> "id"

<resultColor>           ::= <resPrefix> "color"

<resultMulti>           ::= <resultTexCoord> <arrayRange>
                          | <resultClip> <arrayRange>
                          | <resultGeneric> <arrayRange>

<resultTexCoord>        ::= <resPrefix> "texcoord"

<resultClip>            ::= <resPrefix> "clip"

<resultGeneric>         ::= <resPrefix> "attrib"

<resPrefix>             ::= "result" "."
```

(add the following subsection to Section 2.X.3.2, Program Attribute Variables)

Vertex program attribute variables describe the attributes of the vertex being transformed, as specified by the application. The set of available bindings is enumerated in Table X.X. Except where otherwise noted, all vertex program attribute bindings are four-component floating-point vectors.

Vertex Attribute Binding	Components	Underlying State
vertex.position	(x,y,z,w)	object coordinates
vertex.normal	(x,y,z,1)	normal
vertex.color	(r,g,b,a)	primary color
vertex.color.primary	(r,g,b,a)	primary color
vertex.color.secondary	(r,g,b,a)	secondary color
vertex.fogcoord	(f,0,0,1)	fog coordinate
vertex.texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex.attrib[n]	(x,y,z,w)	generic vertex attribute n
vertex.id	(id,-,-,-)	vertex identifier (integer)
vertex.instance	(i,-,-,-)	primitive instance number (integer)
vertex.texcoord[n..o]	(x,y,z,w)	array of texture coordinates
vertex.attrib[n..o]	(x,y,z,w)	array of generic vertex attributes

Table X.X, Vertex Program Attribute Bindings. <n> and <o> refer to integer constants. Only the "vertex.texcoord" and "vertex.attrib" bindings are available in arrays.

NVIDIA Note: The "vertex.weight" and "vertex.matrixindex" bindings described in ARB_vertex_program use state provided only by extensions not supported by NVIDIA implementations and are not available.

If a vertex attribute binding matches "vertex.position", the "x", "y", "z" and "w" components of the vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of the vertex position.

If a vertex attribute binding matches "vertex.normal", the "x", "y", and "z" components of the vertex attribute variable are filled with the "x", "y", and "z" components, respectively, of the vertex normal. The "w" component is filled with 1.

If a vertex attribute binding matches "vertex.color" or "vertex.color.primary", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the vertex color.

If a vertex attribute binding matches "vertex.color.secondary", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the vertex secondary color.

If a vertex attribute binding matches "vertex.fogcoord", the "x" component of the vertex attribute variable is filled with the vertex fog coordinate. The "y", "z", and "w" coordinates are filled with 0, 0, and 1, respectively.

If a vertex attribute binding matches "vertex.texcoord" or "vertex.texcoord[n]", the "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the vertex texture coordinate set <n>. If "[n]" is omitted, texture coordinate set zero is used.

If a vertex attribute binding matches "vertex.instance", the "x" component of the vertex attribute variable is filled with the integer instance number for the primitive to which the vertex belongs. The "y", "z", and "w" components are undefined.

If a vertex attribute binding matches "vertex.attrib[n]", the "x", "y", "z" and "w" components of the generic vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of generic vertex attribute <n>. Note that "vertex.attrib[0]" and "vertex.position" are equivalent. Generic vertex attribute bindings are typeless, and can be interpreted as having floating-point, signed integer, or unsigned integer values, depending on how they are used in the program text. If a vertex attribute is read using a data type different from the one used to specify the generic attribute, the values corresponding to the binding are undefined.

As described in section 2.7, setting a generic vertex attribute may leave a corresponding conventional vertex attribute undefined, and vice versa. To prevent inadvertent use of attribute pairs with undefined attributes, a vertex program will fail to load if it binds both a conventional vertex attribute and a generic vertex attribute listed in the same row of Table X.X.

Conventional Attribute Binding	Generic Attribute Binding
vertex.position	vertex.attrib[0]
vertex.normal	vertex.attrib[2]
vertex.color	vertex.attrib[3]
vertex.color.primary	vertex.attrib[3]
vertex.color.secondary	vertex.attrib[4]
vertex.fogcoord	vertex.attrib[5]
vertex.texcoord	vertex.attrib[8]
vertex.texcoord[0]	vertex.attrib[8]
vertex.texcoord[1]	vertex.attrib[9]
vertex.texcoord[2]	vertex.attrib[10]
vertex.texcoord[3]	vertex.attrib[11]
vertex.texcoord[4]	vertex.attrib[12]
vertex.texcoord[5]	vertex.attrib[13]
vertex.texcoord[6]	vertex.attrib[14]
vertex.texcoord[7]	vertex.attrib[15]
vertex.texcoord[n]	vertex.attrib[8+n]

Table X.X: Invalid Vertex Attribute Binding Pairs. Vertex programs may not bind both attributes listed in any row. The <n> in the last row matches the number of any valid texture unit.

If a vertex attribute binding matches "vertex.texcoord[n..o]" or "vertex.attrib[n..o]", a sequence of 1+<o>-<n> texture coordinate bindings are created. For texture coordinates, it is as though the sequence "vertex.texcoord[n], vertex.texcoord[n+1], ... vertex.texcoord[o]" were specfied. These bindings are available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

When doing vertex array rendering using buffer objects, a vertex ID is also available. If a vertex attribute binding matches "vertex.id", the "x" component of this vertex attribute is filled with the integer index <i> implicitly passed to ArrayElement() to specify the vertex. The vertex ID is defined if and only if:

* the vertex comes from a vertex array command that specifies a complete primitive (e.g., DrawArrays, DrawElements),

* all enabled vertex arrays have non-zero buffer object bindings, and

* the vertex does not come from a display list (even if the display list was compiled using DrawArrays/DrawElements using buffer objects).

The "y", "z", and "w" components of the vertex attribute are always undefined.

(add the following subsection to section 2.X.3.5, Program Results.)

Vertex programs produce vertices, and the set of result variables available to such programs correspond to the attributes of a transformed vertex. The set of allowable result variable bindings for vertex and fragment programs is given in Table X.4.

Binding	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing primary color
result.color.primary	(r,g,b,a)	front-facing primary color
result.color.secondary	(r,g,b,a)	front-facing secondary color
result.color.front	(r,g,b,a)	front-facing primary color
result.color.front.primary	(r,g,b,a)	front-facing primary color
result.color.front.secondary	(r,g,b,a)	front-facing secondary color
result.color.back	(r,g,b,a)	back-facing primary color
result.color.back.primary	(r,g,b,a)	back-facing primary color
result.color.back.secondary	(r,g,b,a)	back-facing secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
result.attrib[n]	(x,y,z,w)	generic interpolant n
result.clip[n]	(d,*,*,*)	clip plane distance
result.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
result.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
result.clip[n..o]	(d,*,*,*)	clip distances n thru o
result.id	(id,*,*,*)	vertex id

Table X.4: Vertex Program Result Variable Bindings. Components labeled "*" are unused.

If a result variable binding matches "result.position", updates to the "x", "y", "z", and "w" components of the result variable modify the "x", "y", "z", and "w" components, respectively, of the transformed vertex's clip coordinates. Final window coordinates will be generated for the vertex as described in section 2.14.4.4.

If a result variable binding match begins with "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the corresponding vertex color attribute in Table X.4. Color bindings that do not specify "front" or "back" are consided to refer to front-facing colors. Color bindings that do not specify "primary" or "secondary" are considered to refer to primary colors.

If a result variable binding matches "result.fogcoord", updates to the "x" component of the result variable set the transformed vertex's fog coordinate. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.pointsize", updates to the "x" component of the result variable set the transformed vertex's point size. Updates to the "y", "z", and "w" components of the result variable have no effect.

If a result variable binding matches "result.texcoord" or "result.texcoord[n]", updates to the "x", "y", "z", and "w" components of the result variable set the "s", "t", "r" and "q" components, respectively, of the transformed vertex's texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is selected.

If a result variable binding matches "result.attrib[n]", updates to the

"x", "y", "z", and "w" components of the result variable set the "x", "y", "z", and "w" components of the generic interpolant <n>. Generic interpolants may be used by subsequent geometry or fragment program invocations, but are not available to fixed-function fragment processing.

If a result variable binding matches "result.clip[n]", updates to the "x" component of the result variable set the clip distance for clip plane <n>.

If a result variable binding matches "result.texcoord[n..o]", "result.attrib[n..o]", or "result.clip[n..o]", a sequence of 1+<o>-<n> bindings is created. For texture coordinates, it is as though the sequence "result.texcoord[n], result.texcoord[n+1], ... result.texcoord[o]" were specfied. This binding is available only in explicit declarations of array variables. A program will fail to load if <n> is greater than <o>.

If a result variable binding matches "result.id", updates to the "x" component of the result variable provide a integer vertex identifier available to geometry programs using the "vertex[m].id" attribute binding. If a geometry program using vertex IDs is active and a vertex program is active, the vertex program must write "result.id" or the vertex ID number is undefined.

(add the following subsection to section 2.X.5, Program Options.)

Section 2.X.5.Y, Vertex Program Options

+ Position-Invariant Vertex Programs (ARB_position_invariant)

If a vertex program specifies the "ARB_position_invariant" option, the program is used to generate all transformed vertex attributes except for position. Instead, clip coordinates are computed as specified in section 2.10. Additionally, user clipping is performed as described in section 2.11. Use of position-invariant vertex programs should generally guarantee that the transformed position of a vertex should be the same whether vertex program mode is enabled or disabled, allowing for correct mixed multi-pass rendering semantics.

When the position-invariant option is specified in a vertex program, vertex programs can no longer declare (explicitly or implicitly) a result variable bound to "result.position". A semantic restriction is added to indicate that a vertex program will fail to load if the number of instructions it contains exceeds the implementation-dependent limit minus four.

(add the following subsection to section 2.X.6, Program Declarations.)

Section 2.X.6.1, Vertex Program Declarations

No declarations are supported at present for vertex programs.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify Section 6.1.14, Shader and Program Queries (p. 256)

(modify 2nd paragraph, p.259) The commands

```
  ...
  void GetVertexAttribIivEXT(uint index, enum pname, int *params);
  void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);
```

obtain the... <pname> must be one of VERTEX_ATTRIB_ARRAY_ENABLED, ... VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER_EXT, or CURRENT_VERTEX_ATTRIB. ...

(split 3rd paragraph, p.259) ... The size, stride, type, normalized flag, and unconverted integer flag are set by the commands VertexAttribPointer and VertexAttribIPointerEXT. The normalized flag is always set to FALSE by by VertexAttribIPointerEXT. The unconverted integer flag is always set to FALSE by VertexAttribPointer and TRUE by VertexAttribIPointerEXT.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute <index>. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to integer values; GetVertexAttribIivEXT reads and returns them a signed integers; GetVertexAttribIuivEXT reads and returns them as unsigned integers. The results of the query are undefined if the current attribute values are read using one data type but specified using a different one. The error INVALID_OPERATION is generated if <index> is zero.

Additions to the AGL/GLX/WGL Specifications

None

GLX Protocol

TBD

Errors

None.

Dependencies on EXT_draw_instanced

If EXT_draw_instanced or a similar extension is not supported, references to the "vertex.instance" attribute binding and a primitive's instance number should be eliminated.

New State

(add to table 6.7, p. 268)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY_INTEGER_EXT	16+xB	GetVertexAttrib	FALSE	vertex attrib array has unconverted ints	2.8	vertex-array

New Implementation Dependent State

None.

Issues

(1) Should a new set of immediate-mode functions be provided for "real integer" attributes? If so, which ones should be provided?

RESOLVED: Yes, although an incomplete subset is provided. This extension provides vector and non-vector functions that accept 1-, 2-, 3-, and 4-component "int" and "uint" values. Additionally, we provide only 4-component vector versions of functions that accept "byte", "ubyte", "short", and "ushort" values. Note that the ARB_vertex_program extension provided a similar incomplete subset.

Since existing VertexAttrib functions include versions that take integer values and convert them to float, it was necessary to create a different way to specify integer values that are not converted. We created a new set of functions using capital letter "I" to denote "real integer" values.

This "I" approach is consistent with a similar choice made by ARB_vertex_program for the existing integer attribute functions. There are two methods of converting to floating point -- straight casts and normalization to [0,1] or [-1,+1]. The normalization version of the attribute functions use the capital letter "N" to denote normalization.

(2) For vertex arrays with "real integer" attributes, should we provide a new function to specify the array or re-use the existing one?

RESOLVED: Provide a new function, VertexAttribIPointerEXT. This function and VertexAttribPointer both set the same attribute state -- state set by VertexAttribPointer for a given <index> will be overwritten by VertexAttribIPointerEXT() and vice versa. There is one new piece of state per array (VERTEX_ATTRIB_ARRAY_INTEGER_EXT) which is set to TRUE for VertexAttribIPointerEXT() and FALSE by VertexAttribPointer. The use of a new function with capital "I" in the name is consistent with the choice made for immediate-mode integer attributes.

We considered reusing the existing VertexAttribPointer function by hijacking the <normalized> parameter, which specifies whether the provided arrays are converted to float by normalizing or a straight cast. It would have been possible to add a third setting to indicate unconverted integer values, but that has two problems: (a) it doesn't agree with the <normalized> flag being specified as a "boolean" (which only has two values), and (b) the enum value that would be used would be

outside the range [0,255] and "boolean" may be represented using single-byte data types.

One other possibility would have been to create a new set of <type> values to indicate integer values that are never converted to floating point -- for example, GL_INTEGER_INT.

(3) Should we provide a whole new set of generic integer vertex attributes?

RESOLVED: No. This extension makes the existing generic vertex attributes "typeless", where they can store either integer or floating-point data. This avoids the need to introduce new hardware resources for integer vertex attributes or software overhead in juggling integer and floating-point generic attributes.

Vertex programs and any queries that access these attributes are responsible for ensuring that they are read using the same data type that they were specified using, and will get undefined results on type mismatches. Checking for such mismatches would be an excellent feature for an instrumented OpenGL driver, or other debugging tool.

(4) Should we provide integer forms of existing conventional attributes?

RESOLVED: No. We could have provided "integer" versions of Color, TexCoord, MultiTexCoord, and other functions, but it didn't seem useful. The use of generic attributes for such values is perfectly acceptable, and fixed-function vertex processing paths won't know what to do with integer values for position, color, normal, and so on.

(5) With integers throughout the pipeline, should we provide automatic identifiers that can be read to get a "vertex number"? If so, how should this functionality be provided?

RESOLVED: The "vertex.id" binding provides an integer "vertex number" for each vertex called the "vertex ID".

When using vertex arrays in vertex buffer objects (VBOs), the vertex ID is defined to be the index of the vertex in the array -- the value implicitly passed to ArrayElement() when DrawArrays() or DrawElements() is called. In practice, vertex arrays in buffer objects will be stored in memory that is directly accessible by the GPU. When functions such as DrawArrays() or DrawElements() are called, a set of vertex indices are passed to the GPU to identify the vertices to pull out of the buffer objects. These same indices can be easily passed to the vertex program.

Vertex IDs can be used by applications in a variety of ways, for example to compute or look up some property of the vertex based on its position in the data set.

Note: The EXT_texture_buffer_object extension can be used to bind a buffer object as a texture resource, which can the be used for lookups in a vertex program. If the amount of memory required for each vertex is very large or is variable, the existing vertex array model might not work very well. However, with TexBOs (texture buffer objects), the vertex program can be used to compute an offset into the buffer object holding the vertex data and fetch the data needed using texture lookups.

This approach blurs the line between texture and vertex pulling, and treats the "texture" in question as a simple array.

(6) Should vertex IDs be provided for vertices in immediate mode? Vertices in display lists? Vertex arrays compiled into a display list?

RESOLVED: No to all.

A different definition would be needed for immediate mode vertices, since the vertex attributes are not specified with an index. It would have been possible to implement some sort of counter where the vertex ID indicates that the vertex is the <N>th one since the previous Begin command.

Vertex arrays compiled into a display list are an even more complicated problem, since either the "array element" definition or the alternate "immediate mode" definition could be used. If the "array element" definition were used, it would additionally be necessary to compile the array element values into the display list. This would introduce additional overhead into the display list, and the storage space for the array element numbers would be wasted if no program using vertex ID were ever used.

While such functionality may be useful, it is left to a subsequent extension.

If such functionality is required, immediate mode VertexAttribI1i() calls can be used to specify the desired "vertex ID" values as integer generic attributes. In this case, the vertex program needs to refer to the specified generic attribute, and not "vertex.id".

(7) Should vertex identifiers be provided for non-VBO vertex arrays? For vertex arrays that are a mix of VBO and non-VBO arrays?

RESOLVED: Non-VBO arrays are generally not stored in memory directly accessible by the GPU; the data are instead copied from the application's memory as they are passed to the GPU. Additionally, the index ordering may not be preserved by the copy. For example, if a DrawElements() call passes vertices numbered 30, 20, 10, and 0 in order, the GPU might see vertex 30's data first, immediately followed by vertex 20's data, and so on.

It would be possible for the driver to provide per-vertex ID values to the GPU during the copy, but defining such functionality is left to a subsequent extension.

For vertices with a mix of VBO arrays and non-VBO arrays, the non-VBO arrays still have the same copy issues, so the automatic vertex ID is not provided.

If such functionality is required, a generic vertex attribute array can be set up using VertexAttribIPointerEXT(), holding integer values 0 through <maxSize>-1, where <maxSize> is the maximum vertex index used. For each vertex, the appropriate "vertex ID" value will be taken from this array. In this case, the vertex program needs to refer to the specified generic attribute, and not "vertex.id".

(8) Should vertex IDs be available to geometry programs, and if so, how?

RESOLVED: Yes, vertex IDs can be passed to geometry programs using the "result.id" binding in a vertex program. Note there is no requirement that the "result.id" written for a vertex must match the "vertex.id" originally provided.

Vertex IDs are not automatically provided to geometry programs; if a vertex program doesn't write to "result.id" or if fixed-function vertex processing is used, the vertex ID visible to the geometry program is undefined.

(9) For instanced arrays (EXT_draw_instanced), should a vertex program be able to read the instance number? If so, how?

RESOLVED: Yes, instance IDs are available to vertex programs using the "vertex.instance" attribute. The instance ID is available in the "x" component and should be read as an integer.

(10) Should instance IDs be available to geometry and fragment programs, and if so, how?

UNRESOLVED: No. If a geometry or fragment program needs the instance ID, the value read in the vertex program can be passed down using a generic integer vertex attribute.

It would be possible to provide a named output binding (e.g., "result.instance") that could be used to pass the instance ID to the next pipeline stage. Using such a binding would have no functional differences from using a generic attribute, except for a name.

In any event, instance IDs are not automatically available to geometry or fragment programs; they must be passed from earlier pipeline stages.

(11) This is an NV extension (NV_vertex_program4). Why do all the new functions and tokens have an "EXT" extension?

RESOLVED: These functions and tokens are shared between this extension and the comparable high-level GLSL programmability extension (EXT_gpu_shader4). Rather than provide a duplicate set of functions, we simply use the EXT version here.

Revision History

None