

NVIDIA OpenGL Extension Specifications for the CineFX 3.0 Architecture (NV4x)

NVIDIA Corporation

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

May 19, 2004

Copyright NVIDIA Corporation, 2001, 2002, 2003, 2004.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation.

This document is an abridged collection of OpenGL extension specifications limited to those extensions for new OpenGL functionality introduced by the CineFX 3.0 (NV4x) 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/>

Table of Contents

Table of NVIDIA OpenGL Extension Support	4
ARB_texture_non_power_of_two.....	8
ATI_draw_buffers.....	20
ATI_texture_float.....	26
ATI_texture_mirror_once.....	30
EXT_blend_equation_separate.....	33
EXT_texture_mirror_clamp.....	39
NV_fragment_program2.....	45
NV_vertex_program3.....	62
WGL_ATI_pixel_format_float.....	76

Table of NVIDIA OpenGL Extension Support

Extension	RIVA 128	RIVA TNT	NV1x	NV2x	NV3x	NV4x	Notes
ARB_depth_texture			Em	R25+	X	X	1.4 functionality
ARB_fragment_program			Em	Em	X	X	
ARB_fragment_program_shadow			Em	Em	R55	X	
ARB_fragment_shader			Em	Em	R60	X	GLSL
ARB_imaging		R10	R10	X	X	X	1.2 imaging subset
ARB_multisample				X	X	X	1.3 functionality
ARB_multitexture		X	X	X	X	X	1.3 functionality
ARB_occlusion_query			Em	R50	R50	R50	
ARB_point_parameters		R35	R35	R35	X	X	1.4 functionality
ARB_point_sprite			R50	R50	R50	X	
ARB_shader_objects			R60	R60	R60	X	GLSL
ARB_shading_language_100			R60	R60	R60	X	GLSL
ARB_shadow			Em	R25+	X	X	1.4 functionality
ARB_texture_border_clamp			Em	X	X	X	1.3 functionality
ARB_texture_compression			X	X	X	X	1.3 functionality
ARB_texture_cube_map			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	1.3 functionality
ARB_texture_env_crossbar							see explanation
ARB_texture_env_dot3			X	X	X	X	1.3 functionality
ARB_texture_mirrored_repeat		R40	R40	R40	X	X	1.4, same as IBM
ARB_texture_non_power_of_two			Em	Em	Em	X	
ARB_transpose_matrix		X	X	X	X	X	1.3 functionality
ARB_vertex_program			R40+	R40+	X	X	
ARB_vertex_shader			R60	R60	R60	R60	GLSL
ARB_window_pos		R40	R40	R40	X	X	1.4 functionality
ATI_draw_buffers			Em	Em	Em	X	
ATI_texture_float			Em	Em	Em	X	
ATI_texture_mirror_once			Em	Em	Em	X	use EXT_texture_mirror_clamp
EXT_abgr	X	X	X	X	X	X	
EXT_bgra	X	X	X	X	X	X	1.2 functionality
EXT_blend_color			X	X	X	X	1.4 functionality
EXT_blend_equation_separate			Em	Em	Em	R60	
EXT_blend_func_separate			Em	Em	X	X	1.4 functionality
EXT_blend_minmax			X	X	X	X	1.4 functionality
EXT_blend_subtract			X	X	X	X	1.4 functionality
EXT_Cg_shader			R60	R60	R60	R60	Cg through GLSL API
EXT_clip_volume_hint			R20+				
EXT_compiled_vertex_array		X	X	X	X	X	
EXT_depth_bounds_test			Em	Em	R50	X	NV35, NV36, NV4x in hw only
EXT_draw_range_elements		R20	R20	R20	X	X	1.2 functionality
EXT_fog_coord		X	X	X	X	X	1.4 functionality
EXT_multi_draw_arrays		R25	R25	R25	X	X	1.4 functionality
EXT_packed_pixels	X	X	X	X	X	X	1.2 functionality
EXT_paletted_texture			X	X	X		no NV4x hw support
EXT_point_parameters	X	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			Em	R25+	X	X	
EXT_shared_texture_palette			X	X	X		no NV4x hw support
EXT_stencil_two_side			Em	Em	X	X	
EXT_stencil_wrap	X	X	X	X	X	X	1.4 functionality
EXT_texture3D		sw	sw	X	X	X	1.2 functionality
EXT_texture_compression_s3tc			X	X	X	X	
EXT_texture_cube_map			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	1.3 functionality
EXT_texture_filter_anisotropic			X	X	X	X	

Table of NVIDIA OpenGL Extension Support NVIDIA OpenGL Extension Specifications for CineFX 3.0

Extension	RIVA 128	RIVA TNT	NV1x	NV2x	NV3x	NV4x	Notes
EXT_texture_lod			X	X	X	X	1.2 functionality; no spec
EXT_texture_lod_bias		R10	X	X	X	X	1.4 functionality
EXT_texture_mirror_clamp			Em	Em	Em	X	
EXT_texture_object	X	X	X	X	X	X	1.1 functionality
EXT_vertex_array	X	X	X	X	X	X	1.1 functionality
EXT_vertex_weighting		X	X	X			Discontinued
KTX_buffer_region	X	X	X	X	X	X	
HP_occlusion_test			Em	R25	X	X	
IBM_rasterpos_clip		R40+	R40+	R40+	R40+	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_copy_depth_to_color			Em	R20	X	X	
NV_depth_clamp			Em	R25+	X	X	
NV_evaluators		R10	R10	X			Discontinued
NV_fence			X	X	X	X	
NV_float_buffer			Em	Em	X	X	
NV_fog_distance		X	X	X	X	X	
NV_fragment_program			Em	Em	X	X	
NV_fragment_program_option			Em	Em	R55	X	NV_fp features for ARB_fp
NV_fragment_program2			Em	Em	Em	X	
NV_half_float			Em	Em	X	X	
NV_light_max_exponent		X	X	X	X	X	
NV_multisample_filter_hint				X	X	X	
NV_occlusion_query			Em	R25	X	X	
NV_packed_depth_stencil		R10+	R10+	R10+	X	X	
NV_pixel_data_range			R40	R40	X	X	
NV_point_sprite			R35+	R25	X	X	
NV_primitive_restart			Em	Em	X	X	
NV_register_combiners			X	X	X	X	
NV_register_combiners2			Em	X	X	X	
NV_texgen_emboss			X				Discontinued
NV_texgen_reflection	X	X	X	X	X	X	use 1.3 functionality
NV_texture_compression_vtc			Em	X	X	X	
NV_texture_env_combine4		X	X	X	X	X	
NV_texture_expand_normal			Em	Em	X	X	
NV_texture_rectangle			X	X	X	X	
NV_texture_shader			Em	X	X	X	
NV_texture_shader2			Em	X	X	X	
NV_texture_shader3			Em	R25	X	X	only NV25 and up in HW
NV_vertex_array_range			X	X	X	X	
NV_vertex_array_range2			R10	R10	X	X	
NV_vertex_program			R10	X	X	X	
NV_vertex_program1_1			R25	R25	X	X	
NV_vertex_program2			Em	Em	X	X	
NV_vertex_program2_option			Em	Em	R55	X	
NV_vertex_program3			Em	Em	Em	X	
S3_s3tc			X	X	X	X	no spec; use EXT_t_c_s3tc
SGIS_generate_mipmap			R10	X	X	X	1.4 functionality
SGIS_multitexture		X	X	X			use 1.3 version
SGIS_texture_lod			X	X	X	X	1.2 functionality
SGIX_depth_texture			Em	X	X	X	use 1.4 version
SGIX_shadow			Em	X	X	X	use 1.4 version
SUN_slice_accum		R50	R50	R50	R50	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	R55	X	
WGL_ARB_multisample				X	X	X	see ARB_multisample
WGL_ARB_pixel_format		R10	R10	X	X	X	Win32
WGL_ARB_pbuffer		R10	R10	X	X	X	Win32
WGL_ARB_render_texture			R25	R25	X	X	Win32
WGL_ATI_pixel_format_float			Em	Em	Em	X	Win32
WGL_EXT_extensions_string		X	X	X	X	X	Win32

NVIDIA OpenGL Extension Specifications for CineFX 3.0 Table of NVIDIA OpenGL Extension Support

Extension	RIVA 128	RIVA TNT	NV1x	NV2x	NV3x	NV4x	Notes
WGL_EXT_swap_control		X	X	X	X	X	Win32
WGL_NV_float_buffer			Em	Em	X	X	Win32, see NV_float_buffer
WGL_NV_render_depth_texture			Em	R25	X	X	Win32
WGL_NV_render_texture_rectangle			R25	R25	X	X	Win32
WIN_swap_hint	X	X	X	X	X	X	Win32, no spec

Key for table entries:

X = supported

sw = supported by software rasterization (expect poor performance)

Em = like sw, but only supported when a sufficient level of hardware emulation is enabled; by default, no hardware emulation is enabled

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

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

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

Name

ARB_texture_non_power_of_two

Name Strings

GL_ARB_texture_non_power_of_two

Notice

Copyright to be assigned to the ARB.

Status

Approved by the ARB on June 11, 2003.

Version

Date: May 14, 2004

Revision: 1.0

Number

ARB Extension #34

Dependencies

Written based on the OpenGL 1.4 specification.

ARB_texture_mirrored_repeat (and IBM_texture_mirrored_repeat) affects the definition of this extension.

ARB_texture_border_clamp affects the definition of this extension.

EXT_texture_compression_s3tc and NV_texture_compression_vtc affect the definition of this extension.

Overview

Conventional OpenGL texturing is limited to images with power-of-two dimensions and an optional 1-texel border. ARB_texture_non_power_of_two extension relaxes the size restrictions for the 1D, 2D, cube map, and 3D texture targets.

There is no additional procedural or enumerant api introduced by this extension except that an implementation which exports the extension string will allow an application to pass in texture dimensions for the 1D, 2D, cube map, and 3D targets that may or may not be a power of two.

An implementation which supports relaxing traditional GL's power-of-two size restrictions across all texture targets will export the extension string: "ARB_texture_non_power_of_two".

When this extension is supported, mipmapping, automatic mipmap generation, and all the conventional wrap modes are supported for non-power-of-two textures

Issues*1. What should this extension be called?*

STATUS: RESOLVED

RESOLUTION: ARB_texture_non_power_of_two. Conventional OpenGL textures are restricted to size dimensions that are powers of two.

The phrases POT (power of two) and NPOT (non-power of two) textures are used in the Overview and Issues section of this specification, but notice these terms are never required in the actual extension language to amend the core specification.

2. Should any enable or other state change be required to relax the texture dimension restrictions?

STATUS: RESOLVED

RESOLUTION: No. The restrictions on texture dimensions in the core OpenGL specification are enforced by errors. Extensions are free to make legal and defined the error behavior of extensions. This extension is really no different in that respect.

The argument for having an enable to "unlock" more generalized texture dimensions is that it avoids developers accidentally releasing applications developed on an OpenGL implementation supporting this extension and unintentionally using NPOT textures. This situation exists in theory with other extensions that do not require new entry points or enumerants to operate (think of NV_blend_square). The real responsibility falls on developers to not use extensions unless the implementation advertises support for the extension and do proper testing to ensure this is really the case.

An additional issue with not having an enable to "unlock" this feature concerns the cases where existing apps might actually be relying on the current error condition to tell them what to do, but might not be able to handle the "new" success this extension would create. However, this seems to be limited to apps that are explicitly checking for implementation correctness (like a conformance test) and this does not seem to be a typical problem for "real-world" applications. The working group members agreed that it is acceptable to require those few apps which fall into this category to be updated in the context of this extension.

3. Should this extension be limited to a subset of conventional texture targets?

STATUS: RESOLVED

SUGGESTION: No. This extension should apply to 1D, 2D, 3D, and cube map textures (all supported by OpenGL 1.4) but this extension does NOT extend or otherwise affect the EXT_texture_rectangle extension's TEXTURE_RECTANGLE_EXT target.

One early point of debate was whether we should have a single unified extension which lifted the power of two restrictions from all targets, or whether we should have individual target specific extensions. For example, one could imagine separate extensions for ARB_texture_non_power_of_two_2d, ARB_texture_non_power_of_two_3d, ARB_texture_non_power_of_two_cube_map.

The advantages of the separate extension approach are to allow IHV's to choose which pieces of functionality to support independently. The advantages of the single extension approach is to have a simpler and more forward looking extension.

4. *Are cube map texture images still required to be square when this extension is supported?*

STATUS: RESOLVED

RESOLUTION: Yes. But while the width and height of each level must be equal, they can be NPOT.

5. *How is a conventional NPOT target different from the texture rectangle target?*

STATUS: RESOLVED

RESOLUTION:

The biggest practical difference is that conventional targets use normalized texture coordinates (ie, [0..1]) while the texture rectangle target uses unnormalized (ie, [0..w]x[0..h]) texture coordinates.

Differences include:

- + In ARB_texture_non_power_of_two:
 - * mipmapping is allowed, default filter remains unchanged.
 - * all wrap modes are allowed, default wrap mode remains unchanged.
 - * borders are supported.
 - * paletted textures are not unsupported.
 - * texture coordinates are addressed parametrically [0..1], [0..1]
- + In EXT_texture_rectangle:
 - * mipmapping is not allowed, default filter is changed to LINEAR.
 - * only CLAMP* wrap modes are allowed, default is CLAMP_TO_EDGE.
 - * borders are not supported.
 - * paletted textures are unsupported.
 - * texture coordinates are addressed non-parametrically [0..w], [0..h].

6. What is the dimension reduction rule for each successively smaller mipmap level?

STATUS: RESOLVED

RESOLUTION: Each successively smaller mipmap level is half the size of the previous level, but if this half value is a fractional value, you should round down to the next largest integer. Essentially:

$$\begin{aligned} & \max(1, \text{floor}(w_b / 2^i)) \times \\ & \max(1, \text{floor}(h_b / 2^i)) \times \\ & \max(1, \text{floor}(d_b / 2^i)) \end{aligned}$$

where i is the i th level beyond the 0th level (the base level).

This is a "floor" convention. An alternative is a "ceiling" convention.

The primary reason to favor the floor convention is that Direct3D uses the floor convention.

Also, the "ceiling" convention potentially implies one more mipmap level than the "floor" convention.

Some regard the "ceiling" convention to have nicer properties with respect to making sure that each level samples at at least 2x the frequency of the next level. This can reduce the chances of sampling artifacts. However, it's probably not worth diverging from the Direct3D convention just for this. A more sophisticated downsampling algorithm (using a larger kernel perhaps) during mipmap level generation can help reduce artifacts related to using the "floor" convention.

The "floor" convention has a relatively straightforward way to evaluate (with integer math) means to determine how many mipmap levels are required for a complete pyramid:

$$\text{numLevels} = 1 + \text{floor}(\log_2(\max(w, h, d)))$$

The "floor" convention can be evaluated incrementally with the following recursion:

$$\text{nextLODdim} = \max(1, \text{currentLODdim} \gg 1)$$

where currentLODdim is the dimension of a level N and nextLODdim is the dimension of level $N+1$. The recursion stops when level $\text{numLevels}-1$ is reached.

Other compromise rules exist such as "round" ($\text{floor}(x+0.5)$). Such a hybrid approach make it more difficult to compute how many mipmap levels are required for a complete pyramid.

Note that this extension is compatible with supporting other rules because it merely relaxes the error and completeness conditions for mipmaps. At the same time, it makes sense to provide developers a single consistent rule since developers are unlikely to want to generate mipmaps for different rules unnecessarily. One reasonable

rule is sufficient and preferable, and the "floor" convention is the best choice.

7. *Should the LOD for filtering (ρ) be computed differently for NPOT textures?*

STATUS: RESOLVED

RESOLUTION: No (though, ideally, the answer would be "yes slightly somehow"). The core OpenGL specification already allows that the ideal computation of ρ (even for POT textures) is "often impractical to implement". The "ceiling" convention adds one more mipmap level for NPOT textures so at extreme minification, the "ceiling" convention may be somewhat sharper than ideal (whereas "floor" would be blurrier).

This excess bluriness should only be significant at the smallest (blurriest) mipmap levels where it should be quite difficult to notice for properly downsampled mipmap images.

8. *Should there be any restrictions on the wrap modes supported for NPOT textures?*

STATUS: RESOLVED

RESOLUTION: No restrictions; all existing wrap modes (GL_REPEAT, GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, and GL_MIRRORED_REPEAT) should "just work" with NPOT textures.

The difficult part of this requirement is to compute " $\text{mod } w_i$ " (or h_i or d_i) rather than simply " $\text{mod } 2^n$ " (or 2^m or 2^l) for the GL_REPEAT wrap mode (GL_MIRRORED_REPEAT may also be an issue, but as defined by OpenGL 1.4, no "mod" math is required to implement the mirrored repeat wrap mode). REPEAT is too commonly used (indeed it is the default wrap mode) to exclude it for NPOT textures.

9. *How does this extension interact with ARB_texture_compression?*

STATUS: RESOLVED

RESOLUTION: It does not. ARB_texture_compression doesn't technically require that any compressed formats be supported. Implementations can choose to compress or not compress any particular texture.

While implementations may choose an internal component resolution and compressed format, the OpenGL 1.4 requires that the choice be a function only of the TexImage parameters. If an implementation chose not to compress NPOT textures, it might get into a situation where a 7x7 image wasn't compressed but its 4x4, 2x2, and 1x1 mipmaps were compressed. The result would be an inconsistent mipmap chain since the internal format of each level would not be the same.

Therefore, an implementation must be able to handle the case where decisions it makes during image specification can be corrected appropriately at render time. This may mean that an implementation such as the one described above may need to temporarily keep

compressed and uncompressed images internally until the full mipmap stack can be examined or may need to decompress previously compressed images in order to recover.

10. How does this extension interact with specific texture compression extensions such as EXT_texture_compression_s3tc?

STATUS: RESOLVED

RESOLUTION: It does not. If both this extension and EXT_texture_compression_s3tc are supported, applications can safely load NPOT S3TC-compressed textures.

Textures are still decomposed into an array of 4x4 blocks. The compressed data for any texels outside the specified image dimensions are irrelevant and are effectively ignored, just as they are for the 1x1 and 2x2 mipmaps of a POT S3TC-compressed texture.

11. How is automatic mipmap generation affected by this extension?

STATUS: RESOLVED

RESOLUTION: It is not directly affected. If an implementation supports automatic mipmap generation, then mipmap generation must be supported even for NPOT textures.

Note however, that the OpenGL 1.4 specification recommends a "2x2 box filter" for the default filter. This is typo since a 2x2 box filter would be incorrect for 1D and 3D textures. With support for NPOT textures, this "2x2 box filter" becomes even more inappropriate. The wording should be changed to simply recommend a box filter where the dimensionality and filter size is assumed appropriate for the texture image dimensionality and size.

12. Are any edits required for Section 3.8.10 "Texture Completeness"?

STATUS: RESOLVED

RESOLUTION: No. This section references Section 3.8.8 for the allowed sequence of dimensions for completeness (rather than stating the requirements explicitly in Section 3.8.10). The only difference between NPOT and POT textures is the allowable sequence of mipmap sizes, and in both cases, a smaller level is half the size of the larger (modulo rounding).

As with POT textures, a mipmap chain is consistent only if the correct sequence of sizes is found. As with POT textures, an attempt to load a mipmap that could never be part of a consistent mipmap chain should fail. For example, if an implementation supports textures with dimensions only up to 1024, an attempt to load level 2 with a 257x114 texture will fail because the smallest possible corresponding level 0 texture would have to be 1028x456.

13. The `WGL_ARB_render_texture` extension allows creating a `pbuffer` with the `WGL_PBUFFER_LARGEST_ARB` attribute. If this extension is present, should this attribute potentially return a NPOT `pbuffer`?

STATUS: UNRESOLVED

SUGGESTION: The `WGL_ARB_render_texture` specification appears to anticipate NPOT textures with this statement: "e.g. Both the width and height will be a power of 2 if the implementation only supports power of 2 textures." so I think the right thing to do is allow NPOT textures (of the proper aspect ratio) to be returned.

It is not entirely clear if this behavior is "safe" for preexisting applications that might not be aware of NPOT textures. The safe thing would be to add a `WGL_PBUFFER_LARGEST_NPOT_ARB` enumerant that could return NPOT textures and require that the existing `WGL_PBUFFER_LARGEST_ARB` enumerant always return POT textures.

New Procedures and Functions

None

New Tokens

None

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

-- Section 3.8.1 "Texture Image Specification"

Replace the discussion of the border parameter with:

"The border argument to `TexImage3D` is a border width. The significance of borders is described below. The border width affect the dimensions of the texture image; it must be the case that

$$w_s = w_i + 2 b_s \quad (3.13)$$

$$h_s = h_i + 2 b_s \quad (3.14)$$

$$d_s = d_i + 2 b_s \quad (3.15)$$

where w_s , h_s , and d_s are the specified image width, height, and depth, and w_i , h_i , and d_i are the dimensions of the texture image internal to the border. If w_i , h_i , or d_i are less than zero, then the error `INVALID_VALUE` is generated.

-- Section 3.8.8 "Texture Minification"

In the subsection "Scale Factor and Level of Detail"...

Replace the sentence defining the u, v, and w functions with:

"Let $u(x,y) = w_i * s(x,y)$, $v(x,y) = h_i * t(x,y)$, and $w(x,y) = d_i * r(x,y)$, where w_i , h_i , and d_i are as defined by equations 3.13, 3.14, and 3.15 with w_s , h_s , and d_s equal to the width, height, and depth of the image array whose level is TEXTURE_BASE_LEVEL."

Replace 2^n , 2^m , and 2^l with w_i , h_i , and d_i in Equations 3.19, 3.20, and 3.21.

$$i = \begin{cases} \text{floor}(u), & s < 1 \\ w_i - 1, & s = 1 \end{cases} \quad (3.19)$$

$$j = \begin{cases} \text{floor}(v), & t < 1 \\ h_i - 1, & t = 1 \end{cases} \quad (3.20)$$

$$k = \begin{cases} \text{floor}(w), & r < 1 \\ d_i - 1, & r = 1 \end{cases} \quad (3.21)$$

Replace 2^n , 2^m , and 2^l with w_i , h_i , and d_i in the equations for computing i_0 , j_0 , k_0 , i_1 , j_1 , and k_1 used for LINEAR filtering.

$$i_0 = \begin{cases} \text{floor}(u - 1/2) \bmod w_i, & \text{TEXTURE_WRAP_S is REPEAT} \\ \text{floor}(u - 1/2), & \text{otherwise} \end{cases}$$

$$j_0 = \begin{cases} \text{floor}(v - 1/2) \bmod h_i, & \text{TEXTURE_WRAP_T is REPEAT} \\ \text{floor}(v - 1/2), & \text{otherwise} \end{cases}$$

$$k_0 = \begin{cases} \text{floor}(w - 1/2) \bmod d_i, & \text{TEXTURE_WRAP_R is REPEAT} \\ \text{floor}(w - 1/2), & \text{otherwise} \end{cases}$$

$$i_1 = \begin{cases} (i_0 + 1) \bmod w_i, & \text{TEXTURE_WRAP_S is REPEAT} \\ i_0 + 1, & \text{otherwise} \end{cases}$$

$$j_1 = \begin{cases} (j_0 + 1) \bmod h_i, & \text{TEXTURE_WRAP_T is REPEAT} \\ j_0 + 1, & \text{otherwise} \end{cases}$$

$$k_1 = \begin{cases} (k_0 + 1) \bmod d_i, & \text{TEXTURE_WRAP_R is REPEAT} \\ k_0 + 1, & \text{otherwise} \end{cases}$$

In the subsection "Mipmapping"...

Replace the last sentence of the first paragraph with:

"If the image array of level `level_base`, excluding its border, has dimensions `w_b` x `h_b` x `d_b`, then there are $\text{floor}(\log_2(\max(w_b, h_b, d_b))) + 1$ image arrays in the mipmap. Numbering the levels such that level `level_base` is the 0th level, the `i`th array has dimensions

$$\begin{aligned} &\max(1, \text{floor}(w_b / 2^i)) \times \\ &\quad \max(1, \text{floor}(h_b / 2^i)) \times \\ &\quad \quad \max(1, \text{floor}(d_b / 2^i)) \end{aligned}$$

until the last array is reached with dimension 1 x 1 x 1."

Replace the second sentence of the second paragraph with:

"Level-of-detail numbers proceed from `level_base` for the original texture array through $p = \text{floor}(\log_2(\max(w_b, h_b, d_b))) + \text{level_base}$ 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."

In the subsection "Automatic Mipmap Generation"...

Replace the second sentence of the third paragraph with:

"No particular filter algorithm is required, though a box filter is recommended as the default filter."

-- Section 3.8.10 "Texture Completeness"**In the subsection "Effects of Completeness on Texture Image Specification"...**

Replace the last sentence with:

"A mipmap complete set of arrays is equivalent to a complete set of arrays where `level_base` = 0 and `level_max` = 1000, and where, excluding borders, the dimensions of the image array being created are understood to be half the corresponding dimensions of the next lower numbered array (rounded down to the next integer if fractional)."

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

None

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

None

Additions to the GLX Specification

None

Additions to the EXT_texture_compression_s3tc and NV_texture_compression_vtc Specification

Add this paragraph:

"For a compressed texture where $w_i \neq 2^m$ OR $h_i \neq 2^n$ OR $d_i \neq 2^l$ for some integer value of m , n , and l , the 4x4 tiles are assumed to be aligned to $u=0$, $v=0$, $w=0$ origin in texel space. For such compressed textures, this implies that texels in regions of tiles beyond the edges $u=w_i$, $v=h_i$, and $w=d_i$ will not be sampled explicitly."

GLX Protocol

None

Errors

Various errors are ELIMINATED when this extension is supported as noted.

INVALID_VALUE is NO LONGER generated by TexImage1D or glCopyTexImage1D if width is not zero or cannot be represented as $2^{n+2}(\text{border})$ for some integer value of n .

INVALID_VALUE is NO LONGER generated by TexImage2D or glCopyTexImage2D if width or height is not zero or cannot be represented as $2^{n+2}(\text{border})$ for some integer value of n .

INVALID_VALUE is NO LONGER generated by TexImage3D if width, height, or depth is not zero or cannot be represented as $2^{n+2}(\text{border})$ for some integer value of n .

New State

None

New Implementation Dependent State

None

Revision History

Date 05/14/2004

Revision: 1.0

- Formated text for 72 column convention
- Fixed date for last revision
- fix "Image2d" typo

Date: 03/23/2004

Revision: 1.0

- Formulas for computing the dimensions of mipmap sizes based on the base level size should involve 2^i (not i^2)

Date: 09/11/2003

Revision: 1.0

- allow zero (instead of just positive values before) when specifying the width, height, and depth of texture image dimensions; this is to avoid an inconsistency with the sample implementation

Date: 05/29/2003

Revision: 0.10

- removed "@" language for target specific behavior, the spec now treats all targets uniformly

Date: 05/21/2003

Revision: 0.9

- fixed typo: ARB/IBM_mirrored_repeat should have been ARB/IBM_texture_mirrored_repeat
- fixed various other minor typos, duplicated words, etc.
- added a line to issue #6 regarding suggesting use of a larger kernel when downsampling using the floor convention
- coalesced the equations that used 3 2-term max equations into single 3-term max equations for clarity
- fixed two more typos where "ceil" should have been "floor"
- refer to ARB_texture_rectangle as EXT_texture_rectangle (this may change back when/if back extension becomes ARB'ified)

Date: 05/10/2003

Revision: 0.8

- additional additional names to contributors list
- clarified language describing resolution of issues #9,10,11

Date: 05/08/2003

Revision: 0.7

- very minor language update to overview section regarding exporting of ARB_texture_non_power_of_two string
- fixed another two places where it said we should round up instead of down (in section 3.8.10 "Texture Completeness", and in section 3.8.8 "Texture Minification")
- mark the regions of the spec affected by the decision to use separate strings per texture target with the "@" symbol. This is temporary until issue #3 is resolved.
- resolved issues 9,10,11,12

Date: 05/08/2003

Revision: 0.6

- updated revision history and coalesced revision notes from various specs
- fixed typo in issue #5 ("2d" --> "non_power_of_two")
- clarified the discussion in issue #3 as the language was a little confusing in parts.
- explicitly refer to the cube map targets in section 3.8.1 instead of using the "made up" target TEXTURE_CUBE_MAP.

Date: 05/06/2003

Revision: 0.5

- changed name of extension from ARB_texture_np2 to ARB_texture_non_power_of_two
- added target specific extension strings
- added more discussion to several issues based on feedback from the working group meetings
- fixed several typos where INVALID_VALUE was INVALID_VALID
- addressed typo in issue #6, it said you should round up, but really we agreed to round down when describing the mipmap stack (floor vs ceil convention).
- resolved issues 1 - 8.

Date: 04/24/2003

Revision: 0.4 (jsandmel)

- numbered issues list
- additional discussion of several issues
- added more explicit comparison of texture_rectangle and this proposal

Date: 04/10/2003

Revision: 0.3 (mjk)

- integrates input from the ARB_texture_2d_np2 proposals.

Date: 03/25/2003

Revision: 0.1 (jsandmel)

- draft proposal
- deals with 2d targets only
- named: ARB_texture_2d_np2

Name

ATI_draw_buffers

Name Strings

GL_ATI_draw_buffers

Status

Complete.

Version

Last Modified Date: December 30, 2002

Revision: 8

Number

277

Dependencies

The extension is written against the OpenGL 1.3 Specification.

OpenGL 1.3 is required.

ARB_fragment_program affects the definition of this extension.

Overview

This extension extends ARB_fragment_program to allow multiple output colors, and provides a mechanism for directing those outputs to multiple color buffers.

Issues

(1) *How many GL_DRAW_BUFFER#_ATI enums should be reserved?*

RESOLVED: We only need 4 currently, but for future expandability it would be nice to keep the enums in sequence. We'll specify 16 for now, which will be more than enough for a long time.

New Procedures and Functions

```
void DrawBuffersATI(sizei n, const enum *bufs);
```

New Tokens

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

MAX_DRAW_BUFFERS_ATI	0x8824
DRAW_BUFFER0_ATI	0x8825
DRAW_BUFFER1_ATI	0x8826
DRAW_BUFFER2_ATI	0x8827
DRAW_BUFFER3_ATI	0x8828
DRAW_BUFFER4_ATI	0x8829
DRAW_BUFFER5_ATI	0x882A
DRAW_BUFFER6_ATI	0x882B
DRAW_BUFFER7_ATI	0x882C
DRAW_BUFFER8_ATI	0x882D
DRAW_BUFFER9_ATI	0x882E
DRAW_BUFFER10_ATI	0x882F
DRAW_BUFFER11_ATI	0x8830
DRAW_BUFFER12_ATI	0x8831
DRAW_BUFFER13_ATI	0x8832
DRAW_BUFFER14_ATI	0x8833
DRAW_BUFFER15_ATI	0x8834

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)**Modify Section 3.11.2, Fragment Program Grammar and Semantic Restrictions**

(replace <resultBinding> grammar rule with these rules)

```
<resultBinding>      ::= "result" "." "color" <optOutputColorNum>
                       | "result" "." "depth"

<optOutputColorNum>  ::= ""
                       | "[" <outputColorNum> "]"

<outputColorNum>     ::= <integer> from 0 to MAX_DRAW_BUFFERS_ATI-1
```

Modify Section 3.11.3.4, Fragment Program Results

(modify Table X.3)

Binding	Components	Description
result.color[n]	(r,g,b,a)	color n
result.depth	(*,**,d)	depth coordinate

Table X.3: Fragment Result Variable Bindings. Components labeled "*" are unused. "[n]" is optional -- color <n> is used if specified; color 0 is used otherwise.

(modify third paragraph) If a result variable binding matches "result.color[n]", 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 corresponding output color. If "result.color[n]" is not both bound by the fragment program and written by some instruction of the program, the output color <n> of the fragment program is undefined.

Add a new Section 3.11.4.5.3

3.11.4.5.3 Draw Buffers Program Option

If a fragment program specifies the "ATI_draw_buffers" option, it will generate multiple output colors, and the result binding "result.color[n]" is allowed, as described in section 3.11.3.4, and with modified grammar rules as set forth in section 3.11.2. 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.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Modify Section 4.2.1, Selecting a Buffer for Writing (p. 168)

(modify the title and first paragraph, p. 168)

4.2.1 Selecting Color Buffers for Writing

The first such operation is controlling the color buffers into which each of the output colors are written. This is accomplished with either DrawBuffer or DrawBuffersATI. DrawBuffer defines the set of color buffers to which output color 0 is written.

(insert paragraph between first and second paragraph, p. 168)

DrawBuffer will set the draw buffer for output colors other than 0 to NONE. DrawBuffersATI defines the draw buffers to which all output colors are written.

```
void DrawBuffersATI(sizei n, const enum *bufs);
```

<n> specifies the number of buffers in <bufs>. <bufs> is a pointer to an array of symbolic constants specifying the buffer to which each output color is written. The constants may be NONE, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, and AUX0 through AUXn, where n + 1 is the number of available auxiliary buffers. The draw buffers being defined correspond in order to the respective output colors. The draw buffer for output colors beyond <n> is set to NONE.

If the "ATI_draw_buffers" fragment program option, is not being used then DrawBuffersATI specifies a set of draw buffers into which output color 0 is written.

The maximum number of draw buffers is implementation dependent and must be at least 1. The number of draw buffers supported can be queried with the state `MAX_DRAW_BUFFERS_ATI`.

The constants `FRONT`, `BACK`, `LEFT`, `RIGHT`, and `FRONT_AND_BACK` that refer to multiple buffers are not valid for use in `DrawBuffersATI` and will result in the error `INVALID_OPERATION`.

If `DrawBuffersATI` is supplied with a constant (other than `NONE`) that does not indicate any of the color buffers allocated to the GL context, the error `INVALID_OPERATION` will be generated. If `<n>` is greater than `MAX_DRAW_BUFFERS_ATI`, the error `INVALID_OPERATION` will be generated.

(replace last paragraph, p. 169)

The state required to handle color buffer selection is an integer for each supported output color. In the initial state, draw buffer for output color 0 is `FRONT` if there are no back buffers; otherwise it is `BACK`. The initial state of draw buffers for output colors other than 0 is `NONE`.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Dependencies on ARB_fragment_program

If `ARB_fragment_program` is not supported then all changes to section 3.11 are removed.

Interactions with possible future extensions

If there is some other future extension that defines multiple color outputs then this extension and `glDrawBuffersATI` could be used to define the destinations for those outputs. This extension need not be used only with `ARB_fragment_program`.

Errors

The error `INVALID_OPERATION` is generated by `DrawBuffersATI` if a color buffer not currently allocated to the GL context is specified.

The error `INVALID_OPERATION` is generated by `DrawBuffersATI` if `<n>` is greater than the state `MAX_DRAW_BUFFERS_ATI`.

The error `INVALID_OPERATION` is generated by `DrawBuffersATI` if value in `<bufs>` does not correspond to one of the allowed buffers.

New State

(table 6.19, p227) add the following entry:

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
DRAW_BUFFERi_ATI	Z10*	GetIntegerv	see 4.2.1	Draw buffer selected for output color i	4.2.1	color-buffer

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_DRAW_BUFFERS_ATI	Z+	GetIntegerv	1	Maximum number of active draw buffers	4.2.1	-

Revision History

Date: 12/30/2002

Revision: 8

- Clarified that DrawBuffersATI will set the set of draw buffers to write color output 0 to when the "ATI_draw_buffer" fragments program option is not in use.

Date: 9/27/2002

Revision: 7

- Fixed confusion between meaning of color buffer and draw buffer in last revision.
- Fixed mistake in when an error is generated based on the <n> argument of DrawBuffersATI.

Date: 9/26/2002

Revision: 6

- Cleaned up and put in sync with latest ARB_fragment_program revision (#22). Some meaningless changes made just in the name of consistency.

Date: 9/11/2002

Revision: 5

- Added section 3.11.4.5.3.
- Added enum numbers to New Tokens.

Date: 9/9/2002

Revision: 4

- Changed error from MAX_OUTPUT_COLORS to MAX_DRAW_BUFFERS_ATI.
- Changed 3.10 section numbers to 3.11 to match change to ARB_fragment_program spec.
- Changed ARB_fragment_program from required to affects, and added section on interactions with it and future extensions that define multiple color outputs.

Date: 9/6/2002

Revision: 3

- Changed error to INVALID OPERATION.
- Cleaned up typos.

Date: 8/19/2002

Revision: 2

- Added a paragraph that specifically points out that the constants that refer to multiple buffers are not allowed with DrawBuffersATI.
- Changed bufs to <bufs> in a couple of places.

Date: 8/16/2002

Revision: 1

- First draft for circulation.

Name

ATI_texture_float

Name Strings

GL_ATI_texture_float

Status

Complete.

Version

Last Modified Date: December 4, 2002

Revision: 4

Number

280

Dependencies

OpenGL 1.1 or EXT_texture is required.

The extension is written against the OpenGL 1.3 Specification.

Overview

This extension adds texture internal formats with 32 and 16 bit floating-point components. The 32 bit floating-point components are in the standard IEEE float format. The 16 bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits. Floating-point components are clamped to the limits of the range representable by their format.

Issues

1. *Should we expose a GL_FLOAT16_ATI pixel type so that the 16 bit float textures can be directly loaded?*

RESOLUTION: This will be exposed in a separate extension.

New Procedures and Functions

None

New Tokens

Accepted by the <internalFormat> parameter of TexImage1D, TexImage2D, and TexImage3D:

RGBA_FLOAT32_ATI	0x8814
RGB_FLOAT32_ATI	0x8815
ALPHA_FLOAT32_ATI	0x8816
INTENSITY_FLOAT32_ATI	0x8817
LUMINANCE_FLOAT32_ATI	0x8818
LUMINANCE_ALPHA_FLOAT32_ATI	0x8819
RGBA_FLOAT16_ATI	0x881A
RGB_FLOAT16_ATI	0x881B
ALPHA_FLOAT16_ATI	0x881C
INTENSITY_FLOAT16_ATI	0x881D
LUMINANCE_FLOAT16_ATI	0x881E
LUMINANCE_ALPHA_FLOAT16_ATI	0x881F

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Add a new Section 2.1.2, (p. 6):

2.1.2 16 Bit Floating-Point

A 16 bit floating-point number has 1 sign bit (s), 5 exponent bits (e), and 10 mantissa bits (m). The value (v) of a 16 bit floating-point number is determined by the following pseudo code:

```

if (e != 0)
    v = (-1)^s * 2^(e-15) * 1.m # normalized
else if (f == 0)
    v = (-1)^s * 0 # zero
else
    v = (-1)^s * 2^(e-14) * 0.m # denormalized

```

It is acceptable for an implementation to treat denormalized 16 bit floating-point numbers as zero.

There are no NAN or infinity values for 16 bit floating-point.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Section 3.8.1, (p. 116), change the last sentence on the page to:

Each R, G, B, and A 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.

Section 3.8.1, (p. 119), add the following to table 3.16:

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
RGBA_FLOAT32_ATI	RGBA	f32	f32	f32	f32		
RGB_FLOAT32_ATI	RGB	f32	f32	f32			
ALPHA_FLOAT32_ATI	ALPHA				f32		
INTENSITY_FLOAT32_ATI	INTENSITY						f32
LUMINANCE_FLOAT32_ATI	LUMINANCE					f32	
LUMINANCE_ALPHA_FLOAT32_ATI	LUMINANCE_ALPHA				f32	f32	
RGBA_FLOAT16_ATI	RGBA	f16	f16	f16	f16		
RGB_FLOAT16_ATI	RGB	f16	f16	f16			
ALPHA_FLOAT16_ATI	ALPHA				f16		
INTENSITY_FLOAT16_ATI	INTENSITY						f16
LUMINANCE_FLOAT16_ATI	LUMINANCE					f16	
LUMINANCE_ALPHA_FLOAT16_ATI	LUMINANCE_ALPHA				f16	f16	

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Errors

None

New State

None

New Implementation Dependent State

None

Revision History

Date: 12/4/2002

Revision: 4

- Added Section 2.1.2 16 Bit Floating-Point.

Date: 9/11/2002

Revision: 3

- Changed description of float clamping to be consistent with WGL_ATI_pixel_format_float.

Date: 9/6/2002

Revision: 2

- Changed unsigned integer components to fixed-point components.
- Resolved GL_FLOAT16_ATI issue.
- Cleaned up typos.

Date: 8/18/2002

Revision: 1

- First draft for circulation.

Name

ATI_texture_mirror_once

Name Strings

GL_ATI_texture_mirror_once

Version

Last Modified Date: 11/14/2000 Revision: 0.30

Number

221

Dependencies

EXT_texture3D

Overview

ATI_texture_mirror_once extends the set of texture wrap modes to include two modes (GL_MIRROR_CLAMP_ATI, GL_MIRROR_CLAMP_TO_EDGE_ATI) that effectively use a texture map twice as large as the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching "mirror image". This mode allows the texture to be mirrored only once in the negative s, t, and r directions.

Issues

None known

New Procedure and Functions

None

New Tokens

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

MIRROR_CLAMP_ATI	0x8742
MIRROR_CLAMP_TO_EDGE_ATI	0x8743

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

None

Additions to Chapter 3 if the OpenGL 1.2.1 Specification (Rasterization):

- (3.8.3, p. 124) Change first three entries in table:

TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, MIRROR_CLAMP_ATI, MIRROR_CLAMP_TO_EDGE_ATI
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, MIRROR_CLAMP_ATI, MIRROR_CLAMP_TO_EDGE_ATI
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, MIRROR_CLAMP_ATI, MIRROR_CLAMP_TO_EDGE_ATI

- (3.8.4, p. 125) Added after second paragraph:

"If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R_EXT is set to MIRROR_CLAMP_ATI or MIRROR_CLAMP_TO_EDGE_ATI, the s (or t or r) coordinate is clamped to [-1, 1] and then converted to:

$$\begin{array}{l} s \quad 0 \leq s \leq 1 \\ -s \quad -1 \leq s < 0 \end{array}$$

Like the CLAMP wrap mode, with MIRROR_CLAMP_ATI the texels from the border can be used by the texture filter. MIRROR_CLAMP_TO_EDGE_ATI clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel."

- (3.8.5, p.127) Change last paragraph to:

"When TEXTURE_MIN_FILTER is LINEAR, a 2 x 2 x 2 cube of texels in the image array of level TEXTURE_BASE_LEVEL is selected. This cube is obtained by first clamping texture coordinates as described above under Texture Wrap Modes (if the wrap mode for a coordinate is CLAMP, CLAMP_TO_EDGE, MIRROR_CLAMP_ATI, or MIRROR_CLAMP_TO_EDGE_ATI) and computing..."

Additions to Chapter 4:

None

Additions to Chapter 5:

None

Additions to Chapter 6:

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

Dependencies on EXT_texture3D

If EXT_texture3D is not implemented, then the references to clamping of 3D textures in this file are invalid, and references to TEXTURE_WRAP_R_EXT should be ignored.

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
-----	-----	---	-----	-----
TEXTURE_WRAP_S	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z5	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z5	REPEAT	texture

New Implementation Dependent State

None

Name

EXT_blend_equation_separate

Name Strings

GL_EXT_blend_equation_separate

Notice

Copyright NVIDIA Corporation, 2003.

Version

Date: 12/23/2003 Version 1.0

Status

Shipping as of May 2004 for GeForce6.

Number

299

Dependencies

Written based on the wording of the OpenGL 1.5 specification.

OpenGL 1.4 (or ARB_imaging, or EXT_blend_minmax and/or EXT_blend_subtract) is required for blend equation support.

EXT_blend_func_separate is presumed but not required.

EXT_blend_logic_op interacts with this extension.

Overview

EXT_blend_func_separate introduced separate RGB and alpha blend factors. EXT_blend_minmax introduced a distinct blend equation for combining source and destination blend terms. (EXT_blend_subtract & EXT_blend_logic_op added other blend equation modes.) OpenGL 1.4 integrated both functionalities into the core standard.

While there are separate blend functions for the RGB and alpha blend factors, OpenGL 1.4 provides a single blend equation that applies to both RGB and alpha portions of blending.

This extension provides a separate blend equation for RGB and alpha to match the generality available for blend factors.

IP Status

No known IP issues.

Issues

Why not use ATI_blend_equation_separate?

Apple supports this extension in OS X 10.2 but the extension lacks a specification and, as explained in subsequent issues, the token naming is inconsistent with OpenGL conventions.

What should the token names be?

RESOLVED: Follow the precedent of EXT_blend_equation_separate. For example, GL_BLEND_DST becomes GL_BLEND_DST_RGB and GL_BLEND_DST_ALPHA. So GL_BLEND_EQUATION becomes GL_BLEND_EQUATION_RGB (same value as GL_BLEND_EQUATION) and GL_BLEND_EQUATION_ALPHA.

This is different from the ATI_blend_equation_separate approach which introduces the single name GL_ALPHA_BLEND_EQUATION_ATI (no RGB name is introduced). The existing OpenGL convention (example: ARB_texture_env_combine) is to use _RGB and _ALPHA as a suffix for enumerants, not a prefix.

How should get token values be assigned?

RESOLVED: GL_BLEND_EQUATION_RGB_EXT has the same value as GL_BLEND_EQUATION. See "Compatibility" section.

For compatibility with ATI_blend_equation_separate, GL_BLEND_EQUATION_ALPHA_EXT shares the same value (0x883D) with the ATI_blend_equation_separate's GL_ALPHA_BLEND_EQUATION_ATI token. The GL_BLEND_EQUATION_ALPHA_EXT name uses the suffixing convention (rather than prefixing) for adding _ALPHA addition as done by ARB_texture_env_combine and EXT_blend_func_separate.

New Procedures and Functions

```
void BlendEquationSeparateEXT(enum modeRGB,
                             enum modeAlpha);
```

New Tokens

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

BLEND_EQUATION_RGB_EXT	0x8009 (same as BLEND_EQUATION)
BLEND_EQUATION_ALPHA_EXT	0x883D

Additions to Chapter 2 of the 1.5 GL Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.5 GL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.5 GL Specification (Per-Fragment Operations and the Framebuffer)

Replace the "Blend Equation" discussion in section 4.1.7 (Blending) with the following:

"The equations used to control blending are determined by the blend equations. Blend equations are specified with the commands:

```
void BlendEquation(enum mode);
void BlendEquationSeparateEXT(enum modeRGB, enum modeAlpha);
```

BlendEquationSeparateEXT arguments modeRGB determines the RGB blend function while modeAlpha determines the alpha blend equation. BlendEquation argument mode determines both the RGB and alpha blend equations. modeRGB and modeAlpha must each be one of FUNC_ADD, FUNC_SUBTRACT, FUNC_REVERSE_SUBTRACT, MIN, or MAX.

Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme in section 2.13.9 (Final Color Processing), as are source (fragment) components. Constant color components are taken to be floating point values. [ed: paragraph unchanged except that floating-point is hyphenated.]

Prior to blending, each fixed-point color component undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Blending components are treated as if carried out in floating-point. [ed: paragraph unchanged except that floating-point is hyphenated.]

Table 4.blendeq provides the corresponding per-component blend equations for each mode, whether acting on RGB components for modeRGB or the alpha component for modeAlpha.

In the table, the "s" subscript on a color component abbreviation (R, G, B, or A) refers to the source color component for an incoming fragment, the "d" subscript on a color component abbreviation refers to the destination color component at the corresponding framebuffer location, and the "c" subscript on a color component abbreviation refers to the constant blend color component. A color component abbreviation without a subscript refers to the new color component resulting from blending. Additionally, Sr, Sg, Sb, and Sa are the red, green, blue, and alpha components of the source weighting factors determined by the source blend function, and Dr, Dg, Db, and Da are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.

Mode	RGB components	Alpha component
----- FUNC_ADD	$R_c = R_s * S_r + R_d * D_r$ $G_c = G_s * S_g + G_d * D_g$ $B_c = B_s * S_b + B_d * D_b$	$A_c = A_s * S_a + A_d * D_a$
----- FUNC_SUBTRACT	$R_c = R_s * S_r - R_d * D_r$ $G_c = G_s * S_g - G_d * D_g$ $B_c = B_s * S_b - B_d * D_b$	$A_c = A_s * S_a - A_d * D_a$
----- FUNC_REVERSE_SUBTRACT	$R_c = R_d * S_r - R_s * D_r$ $G_c = G_d * S_g - G_s * D_g$ $B_c = B_d * S_b - B_s * D_b$	$A_c = A_d * S_a - A_s * D_a$
----- MIN	$R_c = \min(R_s, R_d)$ $G_c = \min(G_s, G_d)$ $B_c = \min(B_s, B_d)$	$A_c = \min(A_s, A_d)$
----- MAX	$R_c = \max(R_s, R_d)$ $G_c = \max(G_s, G_d)$ $B_c = \max(B_s, B_d)$	$A_c = \max(A_s, A_d)$
-----	-----	-----

Table 4.blendeq: RGB and alpha blend equations are their per-component equations controlling the color components resulting from blending for each mode."

In the "Blending State" paragraph, replace the initial lines with...

"The state required for blending is two integers for the RGB and alpha blend equations, four integer indicating the source and destination RGB and alpha blending functions, four floating-point values to store the RGBA constant blend color, and a bit indicating whether blending is enabled or disabled. The initial blending equations for RGB and alpha are FUNC_ADD. ..."

Additions to Chapter 5 of the 1.5 GL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.5 GL Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

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

```
BlendEquationSeparateEXT
  2          12          rendering command length
  2          4228         rendering command opcode
  4          ENUM         modeRGB
  4          ENUM         modeAlpha
```

Dependencies on EXT_blend_logic_op

If EXT_blend_logic_op and EXT_blend_equation_separate are both supported, the logic op blend equation should be supported separately for RGB and alpha as with the other blend equation modes.

And add to the table 4.blendeq this line:

Mode	RGB components	Alpha component
LOGIC_OP	Rc = Rs OP Rd Gc = Gs OP Gd Bc = Bs OP Bd	Ac = As OP Ad

where OP denotes the logical operation controlled by LogicOp (see table 4.2).

Note: there is no support for a distinct RGB logical operation and alpha logical operation (that could be provided by another extension).

Errors

INVALID_ENUM is generated if either the modeRGB or modeAlpha parameter of BlendEquationSeparateEXT is not one of FUNC_ADD, FUNC_SUBTRACT, FUNC_REVERSE_SUBTRACT, MAX, or MIN.

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

New State

Get Value	Get Command	Type	Initial Value	Attribute
BLEND_EQUATION_RGB_EXT	GetIntegerv	Z	FUNC_ADD	color-buffer
BLEND_EQUATION_ALPHA_EXT	GetIntegerv	Z	FUNC_ADD	color-buffer

[remove BLEND_EQUATION from the table, add a note "v1.5 BLEND_EQUATON" beside BLEND_EQUATION_RGB_EXT to note the legacy name.]

New Implementation Dependent State

None

Compatibility

The BLEND_EQUATION_RGB_EXT query token has the same value as the legacy BLEND_EQUATION query token. This means querying the legacy BLEND_EQUATION state is identical to querying the RGB blend equation state.

This is a different approach than taken by the EXT_blend_func_separate extension, but matches the approach taken by other "split" OpenGL state such as the SMOOTH_POINT_SIZE_RANGE and ALIASED_POINT_SIZE_RANGE values split from POINT_SIZE_RANGE.

In the EXT_blend_func_separate case, four new token names (BLEND_DST_RGB, BLEND_SRC_RGB, BLEND_DST_ALPHA, and BLEND_DST_RGB) with four new token values (0x80C8, 0x80C9, 0x80CA, and 0x80CB respectively) were added. Querying the legacy BLEND_DST (0x0BE0) and BLEND_RGB (0x0BE1) returns the same value as querying BLEND_SRC_RGB and BLEND_DST_RGB respectively but this was never explicitly documented.

In the case of the point size ranges, SMOOTH_POINT_SIZE_RANGE was given the same value as POINT_SIZE_RANGE (0x0B12) and a single new token ALIASED_POINT_SIZE_RANGE (0x846D).

The point size ranges approach is preferable because it minimizes the confusion about how the legacy name should be treated by implementations because the legacy name shares its value with the new name. This is less prone to confusion by developers and implementers and less effort to implement.

For token value compatibility with ATI_blend_equation_separate, GL_BLEND_EQUATION_ALPHA_EXT shares the same value (0x883D) with the ATI_blend_equation_separate's GL_ALPHA_BLEND_EQUATION_ATI token.

Name

EXT_texture_mirror_clamp

Name Strings

GL_EXT_texture_mirror_clamp

Status

Shipping as of May 2004 for GeForce6.

Version

Last Modified Date: \$Date: 2004/05/17 \$

NVIDIA Revision: \$Revision: #4 \$

Number

298

Issues

How does EXT_texture_mirror_clamp extend ATI_texture_mirror_once?

This EXT extension provides the two wrap modes that ATI_texture_mirror_once adds but also adds a third new wrap mode (GL_MIRROR_CLAMP_TO_BORDER_EXT). This extension uses the same enumerant values for the ATI_texture_mirror_once modes.

Why is the GL_MIRROR_CLAMP_TO_BORDER_EXT mode more interesting than the two other modes?

Rather than clamp to 100% of the edge of the texture (GL_MIRROR_CLAMP_TO_EDGE_EXT) or to 50% of the edge and border color (GL_MIRROR_CLAMP), it is preferable to clamp to 100% of the border color (GL_MIRROR_CLAMP_TO_BORDER_EXT). This avoids "bleeding" at smaller mipmap levels.

Consider a texture that encodes a circular fall-off pattern such as for a projected spotlight. A circular pattern is bi-symmetric so a "mirror clamp" wrap modes can reduce the memory footprint of the texture by a fourth. Far outside the spotlight pattern, you'd like to sample 100% of the border color (typically black for a spotlight texture). The way to achieve this without any bleeding of edge texels is with GL_MIRROR_CLAMP_TO_BORDER_EXT.

Does this extension complete the orthogonality of the current five OpenGL 1.5 wrap modes?

Yes. There are two ways for repetition to operate (repeated & mirrored) and four ways for texture coordinate clamping to operate (unclamped, clamp, clamp to edge, & clamp to border). The complete table of all 8 modes looks like this:

	Repeat	Mirror
Unclamped	REPEAT	MIRRORED_REPEAT
Clamp	CLAMP	MIRROR_CLAMP
Clamp to edge	CLAMP_TO_EDGE	MIRROR_CLAMP_TO_EDGE
Clamp to border	CLAMP_TO_BORDER	MIRROR_CLAMP_TO_BORDER

OpenGL 1.0 introduced REPEAT & CLAMP.

OpenGL 1.2 introduced CLAMP_TO_EDGE

OpenGL 1.3 introduced CLAMP_TO_BORDER

OpenGL 1.4 introduced MIRRORED_REPEAT

ATI_texture_mirror_once introduced MIRROR_CLAMP & MIRROR_CLAMP_TO_EDGE

EXT_texture_mirror_clamp introduces MIRROR_CLAMP_TO_BORDER

Do these three new wrap modes work with 1D, 2D, 3D, and cube map texture targets?

RESOLUTION: Yes.

Do these three new wrap modes work with ARB_texture_non_power_of_two functionality?

RESOLUTION: Yes.

Do these three new wrap modes interact with NV_texture_rectangle?

RESOLUTION: Mirroring wrap modes are not supported by GL_TEXTURE_RECTANGLE_NV textures. Conventional mirroring is already not supported for texture rectangles so supporting clamped mirroring modes should not be supported either.

Does the specification of MIRROR_CLAMP_EXT & MIRROR_CLAMP_TO_EDGE_EXT match the ATI_texture_mirror_once specification?

I believe yes. The ATI_texture_mirror_once specification is somewhat vague what happens to texture coordinates at or very near (within half a texel of) zero. The presumption is that a CLAMP_TO_EDGE behavior is used. This specification is quite explicit that values near zero are clamped to plus or minus 1/(2*N) respectively so that the CLAMP_TO_EDGE behavior is explicit.

What should this extension be called?

Calling the extension EXT_texture_mirror_once might cause confusion since this extension has additional functionality. Also, "once" never appears in the specification. EXT_texture_mirror_clamp is a good name because it implies support for all the clamped versions of mirroring.

There is GL_MIRRORED_REPEAT and then GL_MIRROR_CLAMP_EXT, GL_MIRROR_CLAMP_TO_EDGE_EXT, and GL_MIRROR_CLAMP_TO_BORDER_EXT. Why does the first enumerant name say "MIRRORED" while the other three say "MIRROR"?

This extension follows the naming precedent set by the ATI_texture_mirror_once specification.

Moreover, MIRRORED_REPEAT uses "mirrored" to help that the mirroring repeats infinitely. For the other three modes, there is just one mirror that occurs and then a clamp.

Dependencies

Written based on the wording of the OpenGL 1.4.

Extends ATI_texture_mirror_once by adding GL_MIRROR_CLAMP_TO_BORDER_EXT.

NV_texture_rectangle trivially affects the definition of this extension.

Overview

EXT_texture_mirror_clamp extends the set of texture wrap modes to include three modes (GL_MIRROR_CLAMP_EXT, GL_MIRROR_CLAMP_TO_EDGE_EXT, GL_MIRROR_CLAMP_TO_BORDER_EXT) that effectively use a texture map twice as large as the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching "mirror image". This mode allows the texture to be mirrored only once in the negative s, t, and r directions.

New Procedure and Functions

None

New Tokens

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

MIRROR_CLAMP_EXT	0x8742 (same value as MIRROR_CLAMP_ATI)
MIRROR_CLAMP_TO_EDGE_EXT	0x8743 (same value as MIRROR_CLAMP_TO_EDGE_ATI)
MIRROR_CLAMP_TO_BORDER_EXT	0x8912

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

None

Additions to Chapter 3 if the OpenGL 1.2.1 Specification (Rasterization):

- (3.8.4, page 136, as amended by the NV_texture_rectangle extension)

Add the 3 new wrap modes to the list of wrap modes unsupported for the TEXTURE_RECTANGLE_NV texture target.

"Certain texture parameter values may not be specified for textures with a target of TEXTURE_RECTANGLE_NV. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_EDGE_EXT, and MIRROR_CLAMP_TO_BORDER_EXT. The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and the TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted). The error INVALID_ENUM is generated if the target is TEXTURE_RECTANGLE_NV and TEXTURE_BASE_LEVEL is set to any value other than zero."

- Table 3.19, page 137: Change first three entries in table:

TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_BORDER, CLAMP_TO_EDGE, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_BORDER_EXT, MIRROR_CLAMP_TO_EDGE_EXT, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_BORDER, CLAMP_TO_EDGE, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_BORDER_EXT, REPEAT
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_BORDER, CLAMP_TO_EDGE, MIRRORED_REPEAT, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_BORDER_EXT, MIRROR_CLAMP_TO_EDGE_EXT, REPEAT"

- (3.8.7, page 140) After the last paragraph of the section add:

"Wrap Mode MIRROR_CLAMP_EXT

Wrap mode MIRROR_CLAMP_EXT mirrors and clamps the texture coordinate, where mirroring and clamping a value f computes

$$\text{mirrorClamp}(f) = \min(1, \max(1/(2*N), \text{abs}(f)))$$

where N is the size of the one-, two-, or three-dimensional texture image in the direction of wrapping.

Wrap Mode MIRROR_CLAMP_TO_EDGE_EXT

Wrap mode MIRROR_CLAMP_TO_EDGE_EXT mirrors and clamps to edge the texture coordinate, where mirroring and clamping to edge a value f computes

$$\text{mirrorClampToEdge}(f) = \min(1-1/(2*N), \max(1/(2*N), \text{abs}(f)))$$

where N is the size of the one-, two-, or three-dimensional texture image in the direction of wrapping.

Wrap Mode MIRROR_CLAMP_TO_BORDER_EXT

Wrap mode MIRROR_CLAMP_TO_BORDER_EXT mirrors and clamps to border the texture coordinate, where mirroring and clamping to border a value f computes

$$\text{mirrorClampToBorder}(f) = \min(1+1/(2*N), \max(1/(2*N), \text{abs}(f)))$$

where N is the size of the one-, two-, or three-dimensional texture image in the direction of wrapping."

- (3.8.8, page 142) Delete this phrase because it is out of date and unnecessary given the current way section 3.8.7 is written:

"(if the wrap mode for a coordinate is CLAMP or CLAMP_TO_EDGE)"

Additions to Chapter 4:

None

Additions to Chapter 5:

None

Additions to Chapter 6:

None

Additions to the GLX Specification

None

Dependencies on NV_texture_rectangle

If NV_texture_rectangle is not supported, ignore the statement that the initial value for the S, T, and R wrap modes is CLAMP_TO_EDGE for rectangular textures.

Ignore the error for a texture target of TEXTURE_RECTANGLE_NV.

GLX Protocol

None

Errors

INVALID_ENUM is generated when TexParameter is called with a target of TEXTURE_RECTANGLE_NV and the TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set to REPEAT, MIRROR_REPEAT_IBM, MIRROR_CLAMP_EXT, MIRROR_CLAMP_TO_EDGE_EXT, or MIRROR_CLAMP_TO_BORDER_EXT.

New State

(table 6.15, p230) amend the following entries [Z5 changed to Z8]:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_WRAP_S	n*Z8	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode S	3.8.7	texture
TEXTURE_WRAP_T	n*Z8	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode T	3.8.7	texture
TEXTURE_WRAP_R	n*Z8	GetTexParameter	REPEAT except for rectangular which is CLAMP_TO_EDGE	Texture wrap mode R	3.8.7	texture

New Implementation Dependent State

None

Name

NV_fragment_program2

Name Strings

GL_NV_fragment_program2

Status

Shipping.

Version

Last Modified: \$Date: 2004/05/17 \$
NVIDIA Revision: 6

Number

Unassigned

Dependencies

ARB_fragment_program is required.
NV_fragment_program_option is required.

Overview

This extension, like the NV_fragment_program_option extension, provides additional fragment program functionality to extend the standard ARB_fragment_program language and execution environment. ARB programs wishing to use this added functionality need only add:

```
OPTION NV_fragment_program2;
```

to the beginning of their fragment programs.

New functionality provided by this extension, above and beyond that already provided by the NV_fragment_program_option extension, includes:

- * structured branching support, including data-dependent IF tests, loops supporting a fixed number of iterations, and a data-dependent loop exit instruction (BRK),
- * subroutine calls,
- * instructions to perform vector normalization, divide vector components by a scalar, and perform two-component dot products (with or without a scalar add),
- * an instruction to perform a texture lookup with an explicit LOD,
- * a loop index register for indirect access into the texture coordinate attribute array, and
- * a facing attribute that indicates whether the fragment is generated from a front- or back-facing primitive.

Issues

- * *Should this extension expose projective forms of the LOD-modifying texture instructions?*

RESOLVED: No. The user can manually add a DIV instruction to achieve the same effect.

- * *Should this extension expose precision explicitly?*

RESOLVED: Only for storage using the SHORT TEMP and LONG TEMP syntax (similar to NV_fragment_program_option).

- * *How are resources (such as registers and condition codes) scoped?*

RESOLVED: All resources are globally scoped. This means that if, for instance, a subroutine modifies a condition code, that modification effects both the caller and the callee.

- * *How is the scope determined for instructions required to be within a specific loop construct?*

RESOLVED: The scope is determined statically at compile time. This means that calling BRK and using A0 from a subroutine called within a loop is a compile error.

New Procedures and Functions

None.

New Tokens

Accepted by the <pname> parameter of GetProgramivARB:

MAX_PROGRAM_EXEC_INSTRUCTIONS_NV	0x88F4
MAX_PROGRAM_CALL_DEPTH_NV	0x88F5
MAX_PROGRAM_IF_DEPTH_NV	0x88F6
MAX_PROGRAM_LOOP_DEPTH_NV	0x88F7
MAX_PROGRAM_LOOP_COUNT_NV	0x88F8

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

None.

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)**Modify Section 3.11 of ARB_fragment_program (Fragment Program):**

Delete the sentence referring to the lack of branching or looping.

Modify Section 3.11.2 of ARB_fragment_program (Fragment Program Grammar and Restrictions):

(mostly add to existing grammar rules, as extended by NV_fragment_program_option)

```

<optionName> ::= "NV_fragment_program2"

<statement> ::= <branchLabel> ":"

<instruction> ::= <FlowInstruction>

<ALUInstruction> ::= <VECSCAop_instruction>

<FlowInstruction> ::= <BRAop_instruction>
                    | <FLOWCCop_instruction>
                    | <IFop_instruction>
                    | <LOOPop_instruction>
                    | <ENDFLOWop_instruction>

<VECTORop> ::= "NRM"

<VECSCAop_instruction> ::= <VECSCAop> <instResult> ", " <instOperandV> ", "
                          <instOperands>

<VECSCAop> ::= "DIV"

<BINop> ::= "DP2"

<TRIOp> ::= "DP2A"

<TEXop> ::= "TXL"

<BRAop_instruction> ::= <BRAop> <branchLabel> <optBranchCond>

<BRAop> ::= "CAL"

<FLOWCCop_instruction> ::= <FLOWCCop> <optBranchCond>

<FLOWCCop> ::= "RET"
              | "BRK"

<IFop_instruction> ::= <IFop> <ccTest>

<IFop> ::= "IF"

<LOOPop_instruction> ::= <LOOPop> <instOperandV>

<LOOPop> ::= "LOOP"
            | "REP"

<ENDFLOWop_instruction> ::= <ENDFLOWop>

<ENDFLOWop> ::= "ELSE"
              | "ENDIF"
              | "ENDLOOP"
              | "ENDREP"

<optBranchCond> ::= /* empty */
                  | <ccMask>

<branchLabel> ::= <identifier>

```

```

<attribFragBasic>      ::= "texcoord" "[" <arrayMemRel> "]"
                        | "facing"

<arrayMemRel>         ::= <addrUseS> <arrayMemRelOffset>

<arrayMemRelOffset>   ::= /* empty */
                        | "+" <addrRegPosOffset>

<addrRegPosOffset>    ::= <integer> from 0 to 9

<addrUseS>            ::= <addrVarName> <scalarAddrSuffix>

<scalarAddrSuffix>    ::= "." <addrComponent>

<addrComponent>       ::= "x"
    
```

Note: This extension provides a pre-defined address register (A0) that matches the <addrVarName> grammar rule and can be used as a loop counter (Section 3.11.3.Y). It is not possible to declare additional address register variables.

Modify Section 3.11.3.1, Fragment Attributes

(add new bindings to binding table)

Fragment Attribute Binding	Components	Underlying State
...		
fragment.texcoord[A0.x+n]	(s,t,r,q)	indexed texture coordinate
fragment.facing	(f,0,0,1)	fragment facing

If a fragment attribute binding matches "fragment.texcoord[A0.x+n]", a texture coordinate number <c> is computed by adding the current value of the "A0.x" address register (the loop index -- Section 3.11.3.Y) and <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 coordinate set <c>. If <c> is negative or greater than or equal to MAX_TEXTURE_COORDS_ARB, the fragment attribute variable is undefined.

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 filled with 0, 0, and 1, respectively.

Add New Section 3.11.3.Y, Fragment Program Address Register (insert after Section 3.11.3.X, Condition Code Register)

Fragment program address register variables are a set of four-component signed integer vectors where only the "x" component of the address registers is currently accessible. Address registers are used as indices when performing relative addressing in the "fragment.texcoord" attribute array (section 3.11.3.1).

Fragment program address registers can not be declared in a fragment program. There is only a single built-in address register, "A0.x" (loop index), which is available inside LOOP/ENDLOOP blocks. A fragment program that accesses A0.x outside a LOOP/ENDLOOP block will fail to load.

A0.x is initialized in by the LOOP instruction and updated by the ENDLOOP instruction. When LOOP blocks are nested, each block has its own value for A0.x, but only the A0.x value for the innermost block can be used. The value of A0.x is clamped to be greater than or equal to 0.

Modify Section 3.11.4, Fragment Program Execution Environment

(modify instruction table) There are sixty-seven fragment program instructions....

Instr.	Modifiers					Inputs	Output	Description
	R	H	X	C	S			
ABS	X	X	X	X	X	v	v	absolute value
ADD	X	X	X	X	X	v,v	v	add
BRK	-	-	-	-	-	c	-	break out of loop instruction
CAL	-	-	-	-	-	c	-	subroutine call
CMP	-	-	-	X	X	v,v,v	v	compare
COS	X	X	-	X	X	s	ssss	cosine with reduction to [-PI,PI]
DDX	X	X	-	X	X	v	v	partial derivative relative to X
DDY	X	X	-	X	X	v	v	partial derivative relative to Y
DIV	X	X	-	X	X	v,s	v	divide vector components by scalar
DP2	X	X	X	X	X	v,v	ssss	2-component dot product
DP2A	X	X	X	X	X	v,v,v	ssss	2-comp. dot product w/scalar add
DP3	X	X	X	X	X	v,v	ssss	3-component dot product
DP4	X	X	X	X	X	v,v	ssss	4-component dot product
DPH	X	X	X	X	X	v,v	ssss	homogeneous dot product
DST	X	X	-	X	X	v,v	v	distance vector
ELSE	-	-	-	-	-	-	-	start if test else block
ENDIF	-	-	-	-	-	-	-	end if test block
ENDLOOP	-	-	-	-	-	-	-	end of loop block
ENDREP	-	-	-	-	-	-	-	end of repeat block
EX2	X	X	-	X	X	s	ssss	exponential base 2
FLR	X	X	X	X	X	v	v	floor
FRC	X	X	X	X	X	v	v	fraction
IF	-	-	-	-	-	c	-	start of if test block
KIL	-	-	-	-	-	v or c	v	kill fragment
LG2	X	X	-	X	X	s	ssss	logarithm base 2
LIT	X	X	-	X	X	v	v	compute light coefficients
LOOP	-	-	-	-	-	v	-	start of loop block
LRP	X	X	X	X	X	v,v,v	v	linear interpolation
MAD	X	X	X	X	X	v,v,v	v	multiply and add
MAX	X	X	X	X	X	v,v	v	maximum
MIN	X	X	X	X	X	v,v	v	minimum
MOV	X	X	X	X	X	v	v	move
MUL	X	X	X	X	X	v,v	v	multiply
NRM	X	X	-	X	X	v	v	normalize 3-component vector
PK2H	-	-	-	-	-	v	ssss	pack two 16-bit floats
PK2US	-	-	-	-	-	v	ssss	pack two unsigned 16-bit scalars
PK4B	-	-	-	-	-	v	ssss	pack four signed 8-bit scalars
PK4UB	-	-	-	-	-	v	ssss	pack four unsigned 8-bit scalars

Instr.	Modifiers					Inputs	Output	Description
	R	H	X	C	S			
POW	X	X	-	X	X	s,s	ssss	exponentiate
RCP	X	X	-	X	X	s	ssss	reciprocal
REP	-	-	-	-	-	v	-	start of repeat block
RET	-	-	-	-	-	c	-	subroutine return
RFL	X	X	-	X	X	v	v	reflection vector
RSQ	X	X	-	X	X	s	ssss	reciprocal square root
SCS	X	X	-	X	X	s	ss--	sine/cosine without reduction
SEQ	X	X	X	X	X	v,v	v	set on equal
SFL	X	X	X	X	X	v,v	v	set on false
SGE	X	X	X	X	X	v,v	v	set on greater than or equal
SGT	X	X	X	X	X	v,v	v	set on greater than
SIN	X	X	-	X	X	s	ssss	sine with reduction to [-PI,PI]
SLE	X	X	X	X	X	v,v	v	set on less than or equal
SLT	X	X	X	X	X	v,v	v	set on less than
SNE	X	X	X	X	X	v,v	v	set on not equal
STR	X	X	X	X	X	v,v	v	set on true
SUB	X	X	X	X	X	v,v	v	subtract
SWZ	X	X	-	X	X	v	v	extended swizzle
TEX	-	-	-	X	X	v	v	texture sample
TXB	-	-	-	X	X	v	v	texture sample with bias
TXD	-	-	-	X	X	v,v,v	v	texture sample w/partial derivatives
TXL	-	-	-	X	X	v	v	texture same w/explicit LOD
TXP	-	-	-	X	X	v	v	texture sample with projection
UP2H	-	-	-	X	X	s	v	unpack two 16-bit floats
UP2US	-	-	-	X	X	s	v	unpack two unsigned 16-bit scalars
UP4B	-	-	-	X	X	s	v	unpack four signed 8-bit scalars
UP4UB	-	-	-	X	X	s	v	unpack four unsigned 8-bit scalars
X2D	X	X	-	X	X	v,v,v	v	2D coordinate transformation
XPD	X	X	-	X	X	v,v	v	cross product

Table X.5: Summary of fragment program instructions. The columns "R", "H", "X", "C", and "S" indicate whether the "R", "H", or "X" precision modifiers, the C condition code update modifier, and the "_SAT"/"_SSAT" saturation modifiers, respectively, are supported for the opcode. In the input/output columns, "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "ss--" indicates two scalar outputs in the first two components, and "c" indicates a condition code test. Instructions describe as "texture sample" also specify a texture image unit identifier and a texture target.

Modify Section 3.11.4.3, Fragment Program Destination Register Update

(modify saturation discussion) If the instruction opcode has the "_SAT" suffix, requesting saturated result vectors, each component of the result vector is clamped to the range [0,1] before updating the destination register. If the instruction opcode has the "_SSAT" suffix, requesting signed saturation, each component of the result vector is clamped to the range [-1,1] before updating the destination register.

Add Section 3.11.4.X, Fragment Program Branching (before Section 3.11.4.4, Fragment Program Result Processing)

Fragment programs support a limited model of branching. Fragment programs can specify one of several types of instruction blocks: IF/ELSE/ENDIF blocks, LOOP/ENDLOOP blocks, and REP/ENDREP blocks. Examples include the following:

```

LOOP {5, 0, 1};      # 5 iterations with loop index at 0,1,2,3,4
ADD R0, R0, R1;
ENDLOOP;

REP repCount;
ADD R0, R0, R1;
ENDREP;

MOVC CC, R0;
IF GT.x;
    MOV R0, R1; # executes if R0.x > 0
ELSE;
    MOV R0, R2; # executes if R0.x <= 0
ENDIF;

```

Instruction blocks may be nested -- for example, a LOOP block may be contained inside an IF/ELSE/ENDIF block. In all cases, each instruction block must be terminated with the appropriate instruction (ENDIF for IF, ENDLOOP for LOOP, ENDREP for REP). Nested instruction blocks must be wholly contained within a block -- if a LOOP instruction is found between an IF and ELSE instruction, the ENDLOOP must also be present between the IF and ELSE. A fragment program will fail to load if any instruction block is terminated by an incorrect instruction or is not terminated before the block containing it.

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 fragment program will fail to load if an IF instruction is nested inside MAX_PROGRAM_IF_DEPTH_NV or more IF/ELSE/ENDIF blocks.

The condition of an IF test is specified by the <ccTest> grammar rule and may depend on the contents of the condition code register. Branch conditions are evaluated by evaluating a condition code write mask in exactly the same manner as done for register writes (section 2.14.2.2). If any of the four components of the condition code write mask are enabled, the branch is taken and execution continues with the instruction following the label specified in the instruction. Otherwise, the instruction is ignored and fragment program execution continues with the next instruction. In the following example code,

```

MOVC CC, c[0];      # c[0]=(-2, 0, 2, NaN), CC gets (LT,EQ,GT,UN)
CAL label1 (LT.xyzw); # call taken
CAL label2 (LT.wyzw); # call not taken

```

the first CAL instruction loads a condition code of (LT,EQ,GT,UN) while the second CAL instruction loads a condition code of (UN,EQ,GT,UN). The first call will be made because the "x" component evaluates to LT; the second call will not be made because no component evaluates to LT.

LOOP/ENDLOOP and REP/ENDREP blocks involve a loop counter that indicates the number of times the instructions between the LOOP/REP and ENDLOOP/ENDREP are executed. Looping blocks have a number of significant limitations. First, the loop counter can not be computed at run time; it must be specified as a program parameter. Second, the number of loop iterations is limited to the value MAX_PROGRAM_LOOP_COUNT_NV, which must be at least 255. Third, only a limited amount of nesting is supported -- a fragment program will fail to load if a LOOP or REP instruction is nested inside MAX_PROGRAM_LOOP_DEPTH_NV or more LOOP/ENDLOOP or REP/ENDREP blocks.

The BRK instruction is available to terminate a loop block early. A BRK instruction can be conditional; the condition is evaluated in the same manner as the condition of an IF instruction, and the loop is terminated if the condition is true. A fragment program will fail to load if it contains a BRK instruction that is not nested inside a LOOP/ENDLOOP or REP/ENDREP block.

Fragment programs can contain one or more instruction labels, matching the grammar rule <branchLabel>. An instruction label can be referred to explicitly in subroutine call (CAL) instructions. Instruction labels can be used at any point in the body of a program, and can be used in instructions before being defined in the program string. Instruction labels can be defined anywhere in the program, except inside an IF/ELSE/ENDIF, LOOP/ENDLOOP, or REP/ENDREP instruction block. A fragment program will fail to load if it contains an instruction label inside an instruction block.

Fragment programs can also specify subroutine calls. When a subroutine call (CAL) instruction is executed, a reference to the instruction immediately following the CAL instruction is pushed onto the call stack. When a subroutine return (RET) instruction is executed, an instruction reference is popped off the call stack and program execution continues with the popped instruction. A fragment program will terminate if a CAL instruction is executed with MAX_PROGRAM_CALL_DEPTH_NV entries already in the call stack or if a RET instruction is executed with an empty call stack. Subroutine calls may be conditional; the condition is specified by the <optBranchCond> grammar rule and evaluated in the same way as the condition of the IF instruction. If no condition is specified, it is as though "(TR)" were specified -- the branch is unconditional.

If a fragment program has an instruction label "main", program execution begins with the instruction immediately following the instruction label. Otherwise, program execution begins with the first instruction of the program. Instructions will be executed sequentially in the order specified in the program, although branch instructions will affect the instruction execution order, as described above. A fragment program will terminate after executing a RET instruction with an empty call stack. A fragment program will also terminate after executing the last instruction in the program, unless that instruction was a taken branch.

A fragment program will fail to load if an instruction refers to a label that is not defined in the program string.

A fragment program will terminate abnormally if a subroutine call instruction produces a call stack overflow. Additionally, a fragment program will terminate abnormally after executing `MAX_PROGRAM_EXEC_INSTRUCTIONS` instructions to prevent hangs caused by infinite loops in the program.

When a fragment program terminates, normally or abnormally, it will emit a fragment whose attributes are taken from the final values of the fragment program result variables (section 3.11.3.4).

Add to Section 3.11.4.5 of ARB_fragment_program (Fragment Program Options):

Section 3.11.4.5.3, NV_fragment_program2 Option

If a fragment program specifies the "NV_fragment_program2" option, the ARB_fragment_program grammar and execution environment are extended to take advantage of all the features of the "NV_fragment_program" option, plus the following features:

- * structured branching support, including data-dependent IF tests, loops supporting a fixed number of iterations, and a data-dependent loop exit instruction (BRK),
- * subroutine calls,
- * several new instructions:
 - * NRM -- vector normalization
 - * DIV -- divide vector components by a scalar
 - * DP2 -- two-component dot product
 - * DP2A -- two-component dot product with scalar add
 - * TXL -- texture lookup with explicit LOD specified
 - * IF/ELSE/ENDIF -- conditional execution blocks
 - * REP/ENDREP -- loop block
 - * LOOP/ENDLOOP -- loop block using index register
 - * BRK -- break out of loop block
 - * CAL -- subroutine call
 - * RET -- subroutine return
- * a loop index register inside LOOP/ENDLOOP blocks that can be used for indirect access into the texture coordinate attribute array, and
- * a facing attribute that indicates whether the fragment is generated from a front- or back-facing primitive.

Modify Section 3.11.5, Fragment Program ALU Instruction Set**Section 3.11.5.48, DIV: Divide (Vector Components by Scalar)**

The DIV instruction divides each component 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;
```

This instruction may not produce results identical to a RCP/MUL instruction sequence.

Section 3.11.5.49, 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;
```

Section 3.11.5.50, DP2A: 2-Component Dot Product w/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;
```

Section 3.11.5.51, NRM: 3-Component Vector Normalize

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;
```

Note that the normalization uses an approximate scale and may be carried at lower precision than a corresponding sequence of DP3, RSQ, and MUL instructions.

Add Section 3.11.6.6, TXL: Texture Lookup with Explicit LOD

The TXD instruction takes the x, y, and z components of the vector operand and maps them to s, t, and r, respectively. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters.

The level of detail is computed as specified in section 3.8.8, except that $\log_2(\rho(x,y))$ is given by 2^w , where w is the w component of the vector operand.

The resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector.

```
tmp = VectorLoad(op0);
result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

Add Section 3.11.X, Fragment Program Flow Control Instruction Set

(immediately after Section 3.11.6, Fragment Program Texture Instruction Set)

3.11.X.1, BRK: Break

The BRK instruction conditionally transfers control to the instruction immediately following the next ENDLOOP or 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 ENDLOOP or
    ENDREP;
}
```

3.11.X.2, CAL: Subroutine Call

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. A CAL instruction has no effect if the condition code test evaluates to FALSE.

When executed, the CAL instruction pushes a reference to the instruction immediately following the CAL instruction onto the call stack. When a matching RET instruction is executed, execution will continue at that instruction after executing the matching RET instruction.

Implementations may have a limited call stack. If the number of CAL instructions that have been performed without returning is MAX_PROGRAM_CALL_DEPTH_NV, a CAL instruction will cause the call stack to overflow and the fragment program to terminate.

The following pseudocode describes the operation of the instruction:

```

if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.**c)) {

    // Check for call stack overflow.
    if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
        terminate fragment program;
    }

    push instruction following the CAL instruction on the call stack;
    continue execution at instruction following <branchLabel>;
}

```

3.11.X.3, ELSE: Beginning of ELSE Block

The ELSE instruction signifies the end of the "execute if true" portion of an IF/ELSE/ENDIF block.

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.

3.11.X.4, ENDIF: End of IF/ELSE Block

The ENDIF instruction signifies the end of an IF/ELSE/ENDIF block. It has no other effect on program execution.

3.11.X.5, ENDLOOP: End of LOOP Block

The ENDLOOP instruction specifies the end of a LOOP block. When an ENDLOOP instruction executes, the loop count is decremented and the loop index increment value is added to the loop index (A0.x). If the decremented loop count is greater than zero, execution continues at the top of the LOOP block.

```

LoopCount--;
LoopIndex += LoopIncr;
if (LoopCount > 0) {
    continue execution at instruction following corresponding LOOP
    instruction;
}

```

3.11.X.6, ENDREP: End of REP Block

The ENDREP instruction specifies the end of a REP block. When an ENDREP instruction executes, the loop count is decremented. If the decremented loop count is greater than zero, execution continues at the top of the REP block.

```

LoopCount--;
if (LoopCount > 0) {
    continue execution at instruction following corresponding LOOP
    instruction;
}

```

3.11.X.7, IF: Beginning of IF Block

The IF instruction conditionally transfers control to the instruction immediately following the corresponding ELSE instruction (if present) or ENDIF instruction (if no ELSE is present).

Implementations may have a limited ability to nest IF blocks at run time. If the number of IF/ENDIF blocks that are currently active is MAX_PROGRAM_IF_DEPTH_NV, an IF instruction will cause the fragment program to terminate. If an IF instruction is executed inside a subroutine, any active IF/ENDIF blocks in the calling code count against this limit.

```

if (IF block nested too deeply) {
    terminate fragment program;
}

// 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;
}

```

3.11.X.8, LOOP: Beginning of LOOP Block

The LOOP instruction begins a LOOP block. The x, y, and z components of the single vector operand specify the initial values for the loop count, loop index, and loop index increment, respectively.

The loop count indicates the number of times the instructions between the LOOP and corresponding ENDLOOP 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 corresponding ENDLOOP instruction.

The loop index (A0.x) can be used for indirect addressing in the set of texture coordinate fragment attributes. A fragment program can only use the loop index of the current LOOP block; loop indices for containing LOOP blocks are not available.

Implementations may have a limited ability to nest LOOP and REP blocks at run time. If the number of LOOP/ENDLOOP and REP/ENDREP blocks that have not completed is MAX_PROGRAM_LOOP_DEPTH_NV, a LOOP instruction will cause the fragment program to terminate. If a LOOP instruction is executed inside a subroutine, any active LOOP/ENDLOOP or REP/ENDREP blocks in the calling code count against this limit.

```

if (LOOP block nested too deeply) {
    terminate fragment program;
}

// Set up loop information for the new nesting level.
tmp = VectorLoad(op0);
LoopCount = floor(op0.x);
LoopIndex = floor(op0.y);
LoopIncr = floor(op0.z);
if (LoopCount <= 0) {
    continue execution at the corresponding ENDLOOP;
}

```

LOOP blocks do not support fully general branching -- a fragment program will fail to load if the vector operand is not a program parameter.

3.11.X.9, REP: Beginning of REP Block

The REP instruction begins a REP block. The x component of the single vector operand specifies the initial value for the loop count. REP blocks are completely identical to LOOP blocks except that they don't use the loop index at all.

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.

Implementations may have a limited ability to nest LOOP and REP blocks at run time. If the number of LOOP/ENDLOOP and REP/ENDREP blocks that have not completed is MAX_PROGRAM_LOOP_DEPTH_NV, a REP instruction will cause the fragment program to terminate. If a REP instruction is executed inside a subroutine, any active LOOP/ENDLOOP or REP/ENDREP blocks in the

calling code count against this limit.

```

if (REP block nested too deeply) {
    terminate fragment program;
}

// Set up loop information for the new nesting level.
tmp = VectorLoad(op0);
LoopCount = floor(op0.x);
if (LoopCount <= 0) {
    continue execution at the corresponding ENDREP;
}

```

REP blocks do not support fully general branching -- a fragment program will fail to load if the vector operand is not a program parameter.

3.11.X.10, RET: Subroutine Return

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction. A RET instruction has no effect if the condition code test evaluates to FALSE.

When executed, the RET instruction pops a reference to the instruction immediately following the corresponding CAL instruction onto the call stack and continues execution at that instruction.

If a RET instruction is issued when the call stack is empty, the fragment program is terminated.

```

if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.**c)) {

    if (callStackDepth <= 0) {
        terminate fragment program;
    }

    pop instruction following the CAL instruction off the call stack;
    continue execution at that instruction;
}

```

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.4 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on ARB_fragment_program

ARB_fragment_program is required.

This specification and NV_fragment_program_option are based on a modified version of the grammar published in the ARB_fragment_program specification. This modified grammar includes a few structural changes to better accommodate new functionality from this and other extensions, but should be functionally equivalent to the ARB_fragment_program grammar. See NV_fragment_program_option for details on the base grammar.

Dependencies on NV_fragment_program2_option

NV_fragment_program_option is required.

If the NV_fragment_program2 program option is specified, all the functionality described in both this extension and the NV_fragment_program_option specification is available.

GLX Protocol

None.

Errors

None.

New State

None.

New Implementation Dependent State

Get Value	Type	Get Command	Min Value	Description	Sec	Attrib
MAX_PROGRAM_EXEC_INSTRUCTIONS_NV	Z+	GetProgramivARB	65536	maximum program execution instruction count	3.11.4.X	-
MAX_PROGRAM_CALL_DEPTH_NV	Z+	GetProgramivARB	4	maximum program call stack depth	3.11.4.X	-
MAX_PROGRAM_IF_DEPTH_NV	Z+	GetProgramivARB	48	maximum program if nesting	3.11.4.X	-
MAX_PROGRAM_LOOP_DEPTH_NV	Z+	GetProgramivARB	4	maximum program loop nesting	3.11.4.X	-
MAX_PROGRAM_LOOP_COUNT_NV	Z+	GetProgramivARB	255	maximum program initial loop count	3.11.4.X	-

(add to Table X.10. New Implementation-Dependent Values Introduced by ARB_fragment_program. Values queried by GetProgramivARB require a <pname> of FRAGMENT_PROGRAM_ARB.)

Revision History

Rev.	Date	Author	Changes
6	05/16/04	pbrown	Documented that "A0" is a pre-defined address register variable for the purposes of the grammar, and that no other address register variables can be declared.
5	-----	pbrown	Internal pre-release revisions.

Name

NV_vertex_program3

Name Strings

GL_NV_vertex_program3

Status

Shipping.

Version

Last Modified Data: \$Date: 2004/05/17 \$
NVIDIA Revision: 1

Number

Unassigned

Dependencies

ARB_vertex_program is required.
NV_vertex_program2_option is required.

Overview

This extension, like the NV_vertex_program2_option extension, provides additional vertex program functionality to extend the standard ARB_vertex_program language and execution environment. ARB programs wishing to use this added functionality need only add:

```
OPTION NV_vertex_program3;
```

to the beginning of their vertex programs.

New functionality provided by this extension, above and beyond that already provided by NV_vertex_program2_option extension, includes:

- * texture lookups in vertex programs,
- * ability to push and pop address registers on the stack,
- * address register-relative addressing for vertex attribute and result arrays, and
- * a second four-component condition code.

Issues

Should we provided a separate "!!VP3.0" program type, like the "!!VP2.0" type defined in NV_vertex_program2?

RESOLVED: No. Since ARB_vertex_program has been fully defined (it wasn't in the !!VP2.0 time-frame), we will simply define language extensions to !!ARBvp1.0 that expose new functionality.

The NV_vertex_program2_option specification followed this same pattern for the NV3X family (GeForce FX, Quadro FX).

Should this be called "NV_vertex_program3_option"?

RESOLVED: No. The similar extension to !!ARBvp1.0 called "NV_vertex_program2_option" got that name only because the simpler "NV_vertex_program2" name had already been used.

Is there a limit on the number of texture units that can be accessed by a vertex program?

RESOLVED: Yes -- same as MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB from the ARB_vertex_shader extension. !!!

Since vertices don't have screen space partial derivatives, how is the LOD used for texture accesses defined?

RESOLVED: The TXL instruction allows a program to explicitly set an LOD; the LOD for all other texture instructions is zero. The texture LOD bias specified in the texture object and environment do apply to all vertex texture lookups.

New Procedures and Functions

None.

New Tokens

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

MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB 0x8B4C

Additions to Chapter 2 of the OpenGL 1.4 Specification (OpenGL Operation)

Modify Section 2.14.2, Vertex Program Grammar and Restrictions

(mostly add to existing grammar rules, as extended by NV_vertex_program2_option)

```

<optionName>                    ::= "NV_vertex_program3"

<instruction>                   ::= <TexInstruction>

<ALUInstruction>                ::= <ASTACKop_instruction>

<TexInstruction>                ::= <TEXop_instruction>

<ASTACKop_instruction>         ::= <ASTACKop> <instOperandAddrVNS>

<ASTACKop>                      ::= "PUSHA"
                                 | "POPA"

<TEXop_instruction>            ::= <TEXop> <instResult> ", " <instOperandV> ", "
                                 <texTarget>

```

```

<TEXop>                ::= "TEX"
                        | "TXP"
                        | "TXB"
                        | "TXL"

<texTarget>            ::= <texImageUnit> "," <texTargetType>

<texImageUnit>         ::= "texture" <optTexImageUnitNum>

<optTexImageUnitNum>  ::= /* empty */
                        | "[" <texImageUnitNum> "]"

<texImageUnitNum>     ::= <integer>
                        /* [0,MAX_TEXTURE_IMAGE_UNITS_ARB-1] */

<texTargetType>       ::= "1D"
                        | "2D"
                        | "3D"
                        | "CUBE"
                        | "RECT"

<attribVtxBasic>      ::= "texcoord" "[" <arrayMemRel> "]"
                        | "attrib" "[" <arrayMemRel> "]"

<resultVtxBasic>      ::= "texcoord" "[" <arrayMemRel> "]"

<ccMaskRule>          ::= "EQ0"
                        | "GEO"
                        | "GTO"
                        | "LEO"
                        | "LTO"
                        | "NEO"
                        | "TRO"
                        | "FLO"
                        | "EQ1"
                        | "GE1"
                        | "GT1"
                        | "LE1"
                        | "LT1"
                        | "NE1"
                        | "TR1"
                        | "FL1"

```

(modify description of reserved identifiers)

... The following strings are reserved keywords and may not be used as identifiers:

```

ABS, ADD, ADDRESS, ALIAS, ARA, ARL, ARR, ATTRIB, BRA, CAL, COS,
DP3, DP4, DPH, DST, END, EX2, EXP, FLR, FRC, LG2, LIT, LOG, MAD,
MAX, MIN, MOV, MUL, OPTION, OUTPUT, PARAM, POPA, POW, PUSHA, RCC,
RCP, RET, RSQ, SEQ, SFL, SGE, SGT, SIN, SLE, SLT, SNE, SUB, SSG,
STR, SWZ, TEMP, TEX, TXB, TXL, TXP, XPD, program, result, state,
and vertex.

```

Modify Section 2.14.3.1, Vertex Attributes

(add new bindings to binding table)

Vertex Attribute Binding	Components	Underlying State
...		
vertex.texcoord[A+n]	(s,t,r,q)	indexed texture coordinate
vertex.attrib[A+n]	(x,y,z,w)	indexed generic vertex attribute

If a vertex attribute binding matches "vertex.texcoord[A+n]", where "A" is a component of an address register (Section 2.14.3.5), a texture coordinate number <c> is computed by adding the current value of the address register component and <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 coordinates for texture unit <c>. If <c> is negative or greater than or equal to MAX_TEXTURE_COORDS_ARB, the vertex attribute variable is undefined.

If a vertex attribute binding matches "vertex.attrib[A+n]", where "A" is a component of an address register (Section 2.14.3.5), a vertex attribute number <a> is computed by adding the current value of the address register component and <n>. The "x", "y", "z", and "w" components of the vertex attribute variable are filled with the "x", "y", "z", and "w" components, respectively, of generic vertex attribute <a>. If <a> is negative or greater than or equal to MAX_VERTEX_ATTRIBS_ARB, the vertex attribute variable is undefined.

Modify Section 2.14.3.4, Vertex Program Results

(add new binding to binding table)

Binding	Components	Description
...		
result.texcoord[A+n]	(s,t,r,q)	indexed texture coordinate

If a result variable binding matches "result.texcoord[A+n]", where "A" is a component of an address register (Section 2.14.3.5), a texture coordinate number <c> is computed by adding the current value of the address register component and <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 <c>. If <c> is negative or greater than or equal to MAX_TEXTURE_COORDS_ARB, the effects of updates to vertex attribute variable are undefined and may overwrite other programs results.

Modify Section 2.14.3.X, Condition Code Registers (added in NV_Vertex_program2_option)

The vertex program condition code registers are two four-component vectors, called CC0 and CC1. Each component of this register is one of four enumerated values: GT (greater than), EQ (equal), LT (less than), or UN (unordered). The condition code register can be used to mask writes to registers and to evaluate conditional branches.

Most vertex program instructions can optionally update one of the two condition code registers. When a vertex program instruction updates a condition code register, a condition code component is set to LT if the corresponding component of the result is less than zero, EQ if it is equal to zero, GT if it is greater than zero, and UN if it is NaN (not a number).

The condition code registers are initialized to vectors of EQ values each time a vertex program executes.

Modify Section 2.14.4, Vertex Program Execution Environment

(modify instruction table) There are forty-eight vertex program instructions. Vertex program instructions may have up to eight variants, including a suffix of "C" or "C0" to allow an update of condition code register zero (section 2.14.3.X), a suffix of "C1" to allow an update of condition code register one, and a suffix of "_SAT" to clamp the result vector components to the range [0,1]. For example, the eight forms of the "ADD" instruction are "ADD", "ADDC", "ADDC0", "ADDC1", "ADD_SAT", "ADDC_SAT", "ADDC0_SAT", and "ADDC1_SAT". The instructions and their respective input and output parameters are summarized in Table X.5.

Instruction	Modifiers		Inputs	Output	Description
	C	S			
ABS	X	X	v	v	absolute value
ADD	X	X	v,v	v	add
ARA	X	-	a	a	address register add
ARL	X	-	s	a	address register load
ARR	X	-	v	a	address register load (round)
BRA	-	-	c	-	branch
CAL	-	-	c	-	subroutine call
COS	X	X	s	ssss	cosine
DP3	X	X	v,v	ssss	3-component dot product
DP4	X	X	v,v	ssss	4-component dot product
DPH	X	X	v,v	ssss	homogeneous dot product
DST	X	X	v,v	v	distance vector
EX2	X	X	s	ssss	exponential base 2
EXP	X	X	s	v	exponential base 2 (approximate)
FLR	X	X	v	v	floor
FRC	X	X	v	v	fraction
LG2	X	X	s	ssss	logarithm base 2
LIT	X	X	v	v	compute light coefficients
LOG	X	X	s	v	logarithm base 2 (approximate)
MAD	X	X	v,v,v	v	multiply and add
MAX	X	X	v,v	v	maximum
MIN	X	X	v,v	v	minimum
MOV	X	X	v	v	move
MUL	X	X	v,v	v	multiply
POPA	-	-	-	a	pop address register
POW	X	X	s,s	ssss	exponentiate
PUSHA	-	-	a	-	push address register
RCC	X	X	s	ssss	reciprocal (clamped)
RCP	X	X	s	ssss	reciprocal
RET	-	-	c	-	subroutine return

Instruction	Modifiers		Inputs	Output	Description
	C	S			
RSQ	X	X	s	ssss	reciprocal square root
SEQ	X	X	v,v	v	set on equal
SFL	X	X	v,v	v	set on false
SGE	X	X	v,v	v	set on greater than or equal
SGT	X	X	v,v	v	set on greater than
SIN	X	X	s	ssss	sine
SLE	X	X	v,v	v	set on less than or equal
SLT	X	X	v,v	v	set on less than
SNE	X	X	v,v	v	set on not equal
SSG	X	X	v	v	set sign
STR	X	X	v,v	v	set on true
SUB	X	X	v,v	v	subtract
SWZ	X	X	v	v	extended swizzle
TEX	X	X	v	v	texture lookup
TXB	X	X	v	v	texture lookup with LOD bias
TXL	X	X	v	v	texture lookup with explicit LOD
TXP	X	X	v	v	projective texture lookup
XPD	X	X	v,v	v	cross product

Table X.5: Summary of vertex program instructions. The columns "C" and "S" indicate whether the "C", "C0", and "C1" condition code update modifiers, and the "_SAT" saturation modifiers, respectively, are supported for the opcode. "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "a" indicates a vector address register, and "c" indicates a condition code test.

Rewrite Section 2.14.4.3, Vertex Program Destination Register Update

A vertex program instruction can optionally clamp the results of a floating-point result vector to the range [0,1]. The components of the result vector are clamped to [0,1] if the saturation suffix "_SAT" is present in the instruction.

Most vertex program instructions write a 4-component result vector to a single temporary or vertex result register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction.

The component write mask is specified by the <optionalMask> rule found in the <maskedDstReg> rule. If the optional mask is "", all components are enabled. Otherwise, the optional mask names the individual components to enable. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be enabled for writing but the y component should not. The grammar requires that the destination register mask components must be listed in "xyzw" order. The condition code write mask is specified by the <ccMask> rule found in the <instResultCC> and <instResultAddrCC> rules. Otherwise, the selected condition code register is loaded and swizzled according to the swizzle codes specified by <swizzleSuffix>. Each component of the swizzled condition code is tested according to the rule given by <ccMaskRule>.

<ccMaskRule> may have the values "EQ", "NE", "LT", "GE", "LE", or "GT", which mean to enable writes if the corresponding condition code field evaluates to equal, not equal, less than, greater than or equal, less than or equal, or greater than, respectively. Comparisons involving condition codes of "UN" (unordered) evaluate to true for "NE" and false otherwise. For example, if the condition code is (GT,LT,EQ,GT) and the condition code mask is "(NE.zyxw)", the swizzle operation will load (EQ,LT,GT,GT) and the mask will thus will enable writes on the y, z, and w components. In addition, "TR" always enables writes and "FL" always disables writes, regardless of the condition code. If the condition code mask is empty, it is treated as "(TR)".

Each component of the destination register is updated with the result of the vertex program instruction if and only if the component is enabled for writes by both the component write mask and the condition code write mask. Otherwise, the component of the destination register remains unchanged.

A vertex program instruction can also optionally update the condition code register. The condition code is updated if the condition code register update suffix "C" is present in the instruction. The instruction "ADDC" will update the condition code; the otherwise equivalent instruction "ADD" will not. If condition code updates are enabled, each component of the destination register enabled for writes is compared to zero. The corresponding component of the condition code is set to "LT", "EQ", or "GT", if the written component is less than, equal to, or greater than zero, respectively. Condition code components are set to "UN" if the written component is NaN (not a number). Values of -0.0 and +0.0 both evaluate to "EQ". If a component of the destination register is not enabled for writes, the corresponding condition code component is also unchanged.

In the following example code,

```
# R1=(-2, 0, 2, NaN)          R0          CC
MOVC R0, R1;                # ( -2,  0,  2, NaN) (LT,EQ,GT,UN)
MOVC R0.xyz, R1.yzwx;       # (  0,  2, NaN, NaN) (EQ,GT,UN,UN)
MOVC R0 (NE), R1.zywx;      # (  0,  0, NaN, -2) (EQ,EQ,UN,LT)
```

the first instruction writes (-2,0,2,NaN) to R0 and updates the condition code to (LT,EQ,GT,UN). The second instruction, only the "x", "y", and "z" components of R0 and the condition code are updated, so R0 ends up with (0,2,NaN,NaN) and the condition code ends up with (EQ,GT,UN,UN). In the third instruction, the condition code mask disables writes to the x component (its condition code field is "EQ"), so R0 ends up with (0,0,NaN,-2) and the condition code ends up with (EQ,EQ,UN,LT).

The following pseudocode illustrates the process of writing a result vector to the destination register. In the pseudocode, "instrSaturate" is TRUE if and only if result saturation is enabled, "instrMask" refers to the component write mask given by the <optWriteMask> rule. "ccMaskRule" refers to the condition code mask rule given by <ccMask> and "updatecc" is TRUE if and only if condition code updates are enabled. "result", "destination", and "cc" refer to the result vector, the register selected by <dstRegister> and the condition code, respectively. Condition codes do not exist

in the VP1 execution environment.

```
boolean TestCC(CondCode field) {
    switch (ccMaskRule) {
        case "EQ": return (field == "EQ");
        case "NE": return (field != "EQ");
        case "LT": return (field == "LT");
        case "GE": return (field == "GT" || field == "EQ");
        case "LE": return (field == "LT" || field == "EQ");
        case "GT": return (field == "GT");
        case "TR": return TRUE;
        case "FL": return FALSE;
        case "": return TRUE;
    }
}

enum GenerateCC(float value) {
    if (value == NaN) {
        return UN;
    } else if (value < 0) {
        return LT;
    } else if (value == 0) {
        return EQ;
    } else {
        return GT;
    }
}
```

```

void UpdateDestination(floatVec destination, floatVec result)
{
    floatVec merged;
    ccVec    mergedCC;

    // Clamp result components to [0,1] if requested in the instruction.
    if (instrSaturate) {
        if (result.x < 0)    result.x = 0;
        else if (result.x > 1) result.x = 1;
        if (result.y < 0)    result.y = 0;
        else if (result.y > 1) result.y = 1;
        if (result.z < 0)    result.z = 0;
        else if (result.z > 1) result.z = 1;
        if (result.w < 0)    result.w = 0;
        else if (result.w > 1) result.w = 1;
    }

    // Merge the converted result into the destination register, under
    // control of the compile- and run-time write masks.
    merged = destination;
    mergedCC = cc;
    if (instrMask.x && TestCC(cc.c***)) {
        merged.x = result.x;
        if (updateecc) mergedCC.x = GenerateCC(result.x);
    }
    if (instrMask.y && TestCC(cc.*c**)) {
        merged.y = result.y;
        if (updateecc) mergedCC.y = GenerateCC(result.y);
    }
    if (instrMask.z && TestCC(cc.**c*)) {
        merged.z = result.z;
        if (updateecc) mergedCC.z = GenerateCC(result.z);
    }
    if (instrMask.w && TestCC(cc.***c)) {
        merged.w = result.w;
        if (updateecc) mergedCC.w = GenerateCC(result.w);
    }

    // Write out the new destination register and condition code.
    destination = merged;
    cc = mergedCC;
}

```

While this rule describes floating-point results, the same logic applies to the integer results generated by the ARA, ARL, and ARR instructions.

Add to Section 2.14.4.5, Vertex Program Options**Section 2.14.4.5.3, NV_vertex_program3 Program Option**

If a vertex program specifies the "NV_vertex_program3" option, the ARB_vertex_program grammar and execution environment are extended to take advantage of all the features of the "NV_vertex_program2" option, plus the following features:

- * several new instructions:
 - * POPA -- pop address register off stack
 - * PUSHA -- push address register onto stack
 - * TEX -- texture lookup
 - * TXB -- texture lookup w/LOD bias
 - * TXL -- texture lookup w/explicit LOD
 - * TXP -- projective texture lookup
- * address register-relative addressing for vertex texture coordinate and generic attribute arrays,
- * address register-relative addressing for vertex texture coordinate result array, and
- * a second four-component condition code.

Add to Section 2.14.5, Vertex Program Instruction Set**Section 2.14.5.43, POPA: Pop Address Register Stack**

The POPA instruction generates a integer result vector by popping an entry off of the call stack.

```

if (callStackDepth <= 0) {
    terminate vertex program;
} else {
    callStackDepth--;
    if (callStack[callStackDepth] is an address register) {
        irestult = callStack[callStackDepth];
    } else {
        terminate vertex program;
    }
}

```

In the pseudocode, <callStackDepth> is the current depth of the call stack and <callStack> is an array holding the call stack.

The vertex program terminates abnormally if it executes a POPA instruction when the call stack is empty, or when the entry at the top of the call stack is not an address register pushed by PUSHA.

Section 2.14.5.44, PUSHA: Push Address Register Stack

The PUSHA instruction pushes the address register operand onto the call stack, which is also used for subroutine calls. The PUSHA instruction does not generate a result vector.

```
tmp = AddrVectorLoad(op0);
if (callStackDepth >= MAX_PROGRAM_CALL_STACK_DEPTH_NV) {
    terminate vertex program;
} else {
    callStack[callStackDepth] = tmp;
    callStackDepth++;
}
```

In the pseudocode, <callStackDepth> is the current depth of the call stack and <callStack> is an array holding the call stack.

The vertex program terminates abnormally if it executes a PUSHA instruction when the call stack is full.

Component swizzling is not supported when the operand is loaded.

Section 2.14.5.45, TEX: Texture Lookup

The TEX instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,1), where x, y, and z are components of the vector operand.

```
tmp = VectorLoad(op0);
result = TextureSample(tmp.x, tmp.y, tmp.z, 1.0, 0.0, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

Since partial derivatives of the texture coordinates are not defined, the base LOD value for vertex texture lookups is defined to be zero. The value of λ' used in equation 3.16 will be simply $\text{clamp}(\text{texobj_bias} + \text{texunit_bias})$.

Section 2.14.5.46, TXB: Texture Lookup (With LOD Bias)

The TXB instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,1), where x, y, and z are components of the vector operand. The w component of the operand is used as an additional LOD bias.

```
tmp = VectorLoad(op0);
result = TextureSample(tmp.x, tmp.y, tmp.z, 1.0, tmp.w, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

Since partial derivatives of the texture coordinates are not defined, the base LOD value for vertex texture lookups is defined to be zero. The value of lambda' used in equation 3.16 will be simply clamp(texobj_bias + texunit_bias + tmp.w).

Since the base LOD value is zero, the TXB instruction is completely equivalent to the TXL instruction, where the w component contains an explicit base LOD value.

Section 2.14.5.47, TXL: Texture Lookup (With Explicit LOD)

The TXL instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,1), where x, y, and z are components of the vector operand. The w component of the operand is used as the base LOD for the texture lookup.

```
tmp = VectorLoad(op0);
result = TextureSampleLOD(tmp.x, tmp.y, tmp.z, 1.0, tmp.w, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

The value of lambda' used in equation 3.16 will be simply tmp.w + clamp(texobj_bias + texunit_bias), where tmp.w is the base LOD.

Section 2.14.5.48, TXP: Texture Lookup (Projective)

The TXP instruction uses the single vector operand to perform a lookup in the specified texture map, yielding a 4-component result vector containing filtered texel values. The (s,t,r,q) coordinates used for the texture lookup are (x,y,z,w), where x, y, z, and w are the four components of the vector operand.

```
tmp = VectorLoad(op0);
result = TextureSample(tmp.x, tmp.y, tmp.z, tmp.w, 0.0, unit, target);
```

where <unit> and <target> are the texture image unit number and target type, matching the <texImageUnitNum> and <texTargetType> grammar rules.

The resulting sample is mapped to RGBA as described in Table 3.21, and the R, G, B, and A values are written to the x, y, z, and w components, respectively, of the result vector.

Since partial derivatives of the texture coordinates are not defined, the base LOD value for vertex texture lookups is defined to be zero. The value of lambda' used in equation 3.16 will be simply clamp(texobj_bias + texunit_bias).

Additions to Chapter 3 of the OpenGL 1.4 Specification (Rasterization)

None.

Additions to Chapter 4 of the OpenGL 1.4 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the OpenGL 1.4 Specification (Special Functions)

None.

Additions to Chapter 6 of the OpenGL 1.4 Specification (State and State Requests)

None.

Additions to Appendix A of the OpenGL 1.4 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Dependencies on ARB_vertex_program

ARB_vertex_program is required.

This specification and NV_vertex_program2_option are based on a modified version of the grammar published in the ARB_vertex_program specification. This modified grammar includes a few structural changes to better accommodate new functionality from this and other extensions, but should be functionally equivalent to the ARB_vertex_program grammar. See NV_vertex_program2_option for details on the base grammar.

Dependencies on NV_vertex_program2_option

NV_vertex_program2_option is required.

If the NV_vertex_program3 program option is specified, all the functionality described in both this extension and the NV_vertex_program2_option specification is available.

Errors

None.

New State

None.

Revision History

None

Name

WGL_ATI_pixel_format_float

Name Strings

WGL_ATI_pixel_format_float

Contact

Rob Mace, ATI Research (mace 'at' ati.com)

Status

Complete.

Version

Last Modified Date: December 4, 2002
Revision: 5

Number

278

Dependencies

WGL_ARB_pixel_format is required.

This extension is written against the OpenGL 1.3 Specification.

Overview

This extension adds pixel formats with floating-point RGBA color components.

The size of each float components is specified using the same WGL_RED_BITS_ARB, WGL_GREEN_BITS_ARB, WGL_BLUE_BITS_ARB and WGL_ALPHA_BITS_ARB pixel format attributes that are used for defining the size of fixed-point components. 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.

In standard OpenGL RGBA color components are normally clamped to the range [0,1]. The color components of a float buffer are clamped to the limits of the range representable by their format.

Issues

1. Should we expose a GL_FLOAT16_ATI pixel type?

RESOLUTION: This will be exposed in a separate extension.

New Procedures and Functions

None

New Tokens

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

```

    RGBA_FLOAT_MODE_ATI                0x8820
    COLOR_CLEAR_UNCLAMPED_VALUE_ATI    0x8835

```

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_FLOAT_ATI            0x21A0

```

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

Add a new Section 2.1.2, (p. 6):

2.1.2 16 Bit Floating-Point

A 16 bit floating-point number has 1 sign bit (s), 5 exponent bits (e), and 10 mantissa bits (m). The value (v) of a 16 bit floating-point number is determined by the following pseudo code:

```

    if (e != 0)
        v = (-1)^s * 2^(e-15) * 1.m # normalized
    else if (f == 0)
        v = (-1)^s * 0 # zero
    else
        v = (-1)^s * 2^(e-14) * 0.m # denormalized

```

It is acceptable for an implementation to treat denormalized 16 bit floating-point numbers as zero.

There are no NAN or infinity values for 16 bit floating-point.

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Section 3.6.4, (p. 92), Add to figure 3.7 a block to "final conversion" for "RGBA float pixel data out" that says "clamp to float format range".

Section 3.6.4, (p. 102), change the first paragraph of the "Final Conversion" to:

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 RGBA components the conversion is based on whether the components in the destination color buffer are fixed-point or floating-point. For fixed-point destination buffers components are clamped to [0,1]. The resulting values are converted to fixed-point according to the rules given in section 2.13.9 (Final Color Processing). For floating-point

destination buffers components are clamped to the limits of the range representable by the destination format.

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Chapter 4 Introduction, (p. 156), change the first line of the third paragraph to:

Color buffers consist of either unsigned integer color indices, RGB and optionally A unsigned integer values, of RGBA floating-point values.

Section 4.1.7, (p. 162), change the third paragraph of the page to:

Fixed-point destination (framebuffer) components and source (fragment) components are taken to be values represented according to the scheme given in section 2.13.9 (Final Color Processing). Floating-point destination and source components are taken as is. Constant color components are taken to be floating-point values.

Section 4.1.7, (p. 163), change the fourth line of the second paragraph of "Using BlendFunc" to:

If destination color components are fixed-point, each floating-point value in this quadruplet is clamped to [0,1] and converted back to a fixed-point value in the manner described in section 2.13.9.

Section 4.1.8, (p. 165), insert after the first sentence:

Dithering has no effect if the destination color buffer components are floating-point.

Section 4.1.9, (p. 165), insert after the first sentence:

Logical operation has no effect if the destination color buffer components are floating-point.

Section 4.2.3, (p. 170), change the third paragraph to:

```
void ClearColor(float r, float g, float b, float a);
```

sets the clear value for the color buffers in RGBA mode. When clearing a fixed-point color buffer each of the specified components is clamped to [0; 1] and converted to fixed-point according to the rules of section 2.13.9. When clearing a floating-point color buffer the specified components are not clamped.

Section 4.3.2, (p. 176), change the "Conversion of RGBA values" to:

This step applies only if the GL is in RGBA mode, and then only if format is neither STENCIL INDEX nor DEPTH COMPONENT. The R, G, B, and A values form a group of elements. When reading from 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 2.13.9).

Section 4.3.2, (p. 177), change the second paragraph of the "Final Conversion" to:

For a fixed-point RGBA color buffer, each component is first clamped to [0,1]. For floating-point RGBA color buffer, components are not clamped if the <type> is FLOAT, clamped to [0,1] if the <type> is unsigned, and clamped to [-1,1] if the <type> is signed. After clamping the appropriate conversion formula from table 4.7 is applied to the component.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Additions to the GLX Specification

This specification is written for WGL.

GLX Protocol

This specification is written for WGL.

Additions to the WGL Specification

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, or WGL_TYPE_COLORINDEX_ARB.

Dependencies on WGL_ARB_pixel_format

The WGL_ARB_pixel_format extension must be used to determine a pixel format with float components.

Dependencies on WGL_ARB_extensions_string

Because this extension is a WGL extension, it is not included in the GL_EXTENSIONS string. Its existence can be determined with the WGL_ARB_extensions_string extension.

Errors

None

New State

(table 6.19, p227) modify COLOR_CLEAR_VALUE and add COLOR_CLEAR_UNCLAMPED_VALUE:

Get Value	Type	Get Command	Initial Value	Description	Section	Attribute
COLOR_CLEAR_VALUE	C	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode) clamped to [0,1]	4.2.3	color-buffer
COLOR_CLEAR_UNCLAMPED_VALUE_ATI	4 x R	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode) unclamped	4.2.3	color-buffer

(table 6.28, p236) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Section	Attribute
RGBA_FLOAT_MODE_ATI	B	GetBooleanv	-	True if RGBA components are floats	2.7	-

New Implementation Dependent State

None

Revision History

Date: 12/4/2002

Revision: 5

- Added Section 2.1.2 16 Bit Floating-Point.

Date: 9/12/2002

Revision: 4

- Fixed typo, CLEAR_COLOR_VALUE is really COLOR_CLEAR_VALUE.

Date: 9/11/2002

Revision: 3

- Added enum numbers to New Tokens.
- Added CLEAR_COLOR_UNCLAMPED_VALUE_ATI and defined behavior of CLEAR_COLOR_VALUE.
- Added description of change to figure 3.7.
- Clarified float clamping in section 3.6.4.

Date: 9/9/2002

Revision: 2

- Changed wording of how float clamping is described in Overview.

Date: 9/6/2002

Revision: 1

- First draft for circulation.