# NVIDIA OpenGL Extension Specifications for the CineFX Architecture (NV3*x*)

*November 13, 2006*

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

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

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

Other OpenGL extension specifications can be found at:

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

**Corrections?** Email opengl-specs@nvidia.com

**Table of Contents**

## Table of NVIDIA OpenGL Extension Support

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

| Extension | NV1*x* | NV2*x* | *NV3x* | *NV4x* | *G8x* | Notes |
|---|---|---|---|---|---|---|
| EXT_paletted_texture | X | X | X | | | no NV4x hw support |
| EXT_pixel_buffer_object | R55 | R55 | R55 | X | X | 2.1 functionality |
| EXT_point_parameters | X | X | X | X | X | 1.4 functionality |
| EXT_rescale_normal | X | X | X | X | X | 1.2 functionality |
| EXT_secondary_color | X | X | X | X | X | 1.4 functionality |
| EXT_separate_specular_color | X | X | X | X | X | 1.2 functionality |
| EXT_shadow_funcs | | R25+ | X | X | X | 1.5 functionality |
| EXT_shared_texture_palette | X | X | X | | | no NV4x hw support |
| EXT_stencil_clear_tag | | | | R70 | | NV44 only |
| EXT_stencil_two_side | | | X | X | X | 2.0 functionality |
| EXT_stencil_wrap | X | X | X | X | X | 1.4 functionality |
| EXT_texture3D | sw | X | X | X | X | 1.2 functionality |
| EXT_texture_array | | | | | X | |
| EXT_texture_buffer_object | | | | | X | |
| EXT_texture_compression_latc | | | | | X | |
| EXT_texture_compression_rgtc | | | | | X | |
| EXT_texture_compression_s3tc | X | X | X | X | X | |
| EXT_texture_cube_map | X | X | X | X | X | 1.2 functionality |
| EXT_texture_edge_clamp | X | X | X | X | X | 1.2 functionality |
| EXT_texture_env_add | X | X | X | X | X | 1.3 functionality |
| EXT_texture_env_combine | X | X | X | X | X | 1.3 functionality |
| EXT_texture_env_dot3 | X | X | X | X | X | 1.3 functionality |
| EXT_texture_filter_anisotropic | X | X | X | X | X | |
| EXT_texture_integer | | | | | X | |
| EXT_texture_lod | X | X | X | X | X | 1.2 functionality; no spec |
| EXT_texture_lod_bias | X | X | X | X | X | 1.4 functionality |
| EXT_texture_mirror_clamp | | | | X | X | |
| EXT_texture_object | X | X | X | X | X | 1.1 functionality |
| EXT_texture_shared_exponent | | | | | X | |
| EXT_texture_sRGB | | | | X | X | 2.1 functionality |
| EXT_timer_query | | R80 | R80 | R80 | X | |
| EXT_vertex_array | X | X | X | X | X | 1.1 functionality |
| EXT_vertex_weighting | X | X | | | | Discontinued |
| KTX_buffer_region | X | X | X | X | X | |
| HP_occlusion_test | | R25 | X | X | X | |
| IBM_rasterpos_clip | R40+ | R40+ | R40+ | X | X | |
| IBM_texture_mirrored_repeat | X | X | X | X | X | 1.4 functionality |
| KTX_buffer_region | X | X | X | X | X | use ARB_buffer_region |
| NV_blend_square | X | X | X | X | X | 1.4 functionality |
| NV_copy_depth_to_color | | R20 | X | X | X | |
| NV_depth_buffer_float | | | | | X | |
| NV_depth_clamp | | R25+ | X | X | X | |
| NV_evaluators | R10 | X | | | | Discontinued |
| NV_fence | X | X | X | X | X | |
| NV_float_buffer | | | X | X | X | |
| NV_fog_distance | X | X | X | X | X | |
| NV_fragment_program | | | X | X | X | |
| NV_fragment_program_option | | | R55 | X | X | NV_fp features for ARB_fp |
| NV_fragment_program2 | | | | X | X | |
| NV_fragment_program4 | | | | | X | See NV_gpu_program4 |
| NV_framebuffer_multisample_coverage | | | Nf | Nf | X | FBO extension |
| NV_geometry_program4 | | | | | X | See NV_gpu_program4 |
| NV_gpu_program4 | | | | | X | |
| NV_half_float | | | X | X | X | |
| NV_light_max_exponent | X | X | X | X | X | |
| NV_multisample_filter_hint | | X | X | X | X | |
| NV_occlusion_query | | R25 | X | X | X | |
| NV_packed_depth_stencil | R10+ | R10+ | X | X | X | |
| NV_parameter_buffer_object | | | | | X | See NV_gpu_program4 |
| NV_pixel_data_range | R40 | R40 | X | X | X | |
| NV_point_sprite | R35+ | R25 | X | X | X | |
| NV_primitive_restart | | | X | X | X | |
| NV_register_combiners | X | X | X | X | X | |
| NV_register_combiners2 | | X | X | X | X | |

| Extension | NV1*x* | NV2*x* | *NV3x* | *NV4x* | *G8x* | Notes |
|---|---|---|---|---|---|---|
| NV_texgen_emboss | X | | | | | Discontinued |
| NV_texgen_reflection | X | X | X | X | X | use 1.3 functionality |
| NV_texture_compression_vtc | | X | X | X | X | |
| NV_texture_env_combine4 | X | X | X | X | X | |
| NV_texture_expand_normal | | | X | X | X | |
| NV_texture_rectangle | X | X | X | X | X | |
| NV_texture_shader | | X | X | X | X | |
| NV_texture_shader2 | | X | X | X | X | |
| NV_texture_shader3 | | R25 | X | X | X | only NV25 and up in HW |
| NV_transform_feedback | | | | | X | |
| NV_vertex_array_range | X | X | X | X | X | |
| NV_vertex_array_range2 | R10 | R10 | X | X | X | |
| NV_vertex_program | R10 | X | X | X | X | |
| NV_vertex_program1_1 | R25 | R25 | X | X | X | |
| NV_vertex_program2 | | | X | X | X | |
| NV_vertex_program2_option | | | R55 | X | X | |
| NV_vertex_program3 | | | | X | X | |
| NV_vertex_program4 | | | | | X | See NV_gpu_program4 |
| S3_s3tc | X | X | X | X | X | no spec; use EXT_t_c_s3tc |
| SGIS_generate_mipmap | R10 | X | X | X | X | 1.4 functionality |
| SGIS_multitexture | X | X | | | | use 1.3 version |
| SGIS_texture_lod | X | X | X | X | X | 1.2 functionality |
| SGIX_depth_texture | | X | X | X | X | use 1.4 version |
| SGIX_shadow | | X | X | X | X | use 1.4 version |
| SUN_slice_accum | R50 | R50 | R50 | X | X | accelerated on NV3x/NV4x |
| WGL_ARB_buffer_region | X | X | X | X | X | Win32 |
| WGL_ARB_extensions_string | X | X | X | X | X | Win32 |
| WGL_ARB_make_current_read | R55 | R55 | R55 | X | X | |
| WGL_ARB_multisample | | X | X | X | X | see ARB_multisample |
| WGL_ARB_pixel_format | R10 | X | X | X | X | Win32 |
| WGL_ARB_pbuffer | R10 | X | X | X | X | Win32 |
| WGL_ARB_render_texture | R25 | R25 | X | X | X | Win32 |
| WGL_ATI_pixel_format_float | | | | X | X | Win32 |
| WGL_EXT_extensions_string | X | X | X | X | X | Win32 |
| WGL_EXT_swap_control | X | X | X | X | X | Win32 |
| WGL_NV_float_buffer | | | X | X | X | Win32, see NV_float_buffer |
| WGL_NV_gpu_affinity | | | | R95 | X | Win32 SLI |
| WGL_NV_render_depth_texture | | R25 | X | X | X | Win32 |
| WGL_NV_render_texture_rectangle | R25 | R25 | X | X | X | Win32 |
| WIN_swap_hint | X | X | X | X | X | Win32, no spec |

**Key for table entries:**

**X** = *supported*

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

**Nf** = *Extension advertised but rendering functionality not available*

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

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

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

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

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

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

**R40** = Detonator 40 release, August 2002.

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

**R50** = Detonator 50 release

**R55** = Detonator 55 release

**R60** = Detonator 60 release, May 2004

**R65** = Release 65

**R70** = Release 70

**R80** = Release 80

**R95** = Release 95

**no spec** = *no suitable specification available*

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

**Notices:**

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

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

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

**Name**

    ARB_fragment_program

**Name Strings**

    GL_ARB_fragment_program

**IP Status**

    Microsoft claims to own intellectual property related to this
    extension.

**Status**

    Complete.  Approved by ARB on September 18, 2002

**Version**

    Last Modified Date: August 22, 2003
    Revision: 26

**Number**

    ARB Extension #27

**Dependencies**

    The extension is written against the OpenGL 1.3 Specification.

    OpenGL 1.3 is required.

    EXT_texture_lod_bias or OpenGL 1.4 is required.

    OpenGL 1.4 affects the definition of this extension.

    ARB_vertex_blend and EXT_vertex_weighting affect the definition of
    this extension.

    ARB_matrix_palette affects the definition of this extension.

    ARB_transpose_matrix affects the definition of this extension.

    EXT_fog_coord affects the definition of this extension.

    EXT_texture_rectangle affects the definition of this extension.

    ARB_shadow interacts with this extension.

    ARB_vertex_program interacts with this extension.

    ATI_fragment_shader interacts with this extension.

    NV_fragment_program interacts with this extension.

**Overview**

Unextended OpenGL mandates a certain set of configurable per-fragment computations defining texture application, texture environment, color sum, and fog operations. Several extensions have added further per-fragment computations to OpenGL. For example, extensions have defined new texture environment capabilities (ARB_texture_env_add, ARB_texture_env_combine, ARB_texture_env_dot3, ARB_texture_env_crossbar), per-fragment depth comparisons (ARB_depth_texture, ARB_shadow, ARB_shadow_ambient, EXT_shadow_funcs), per-fragment lighting (EXT_fragment_lighting, EXT_light_texture), and environment mapped bump mapping (ATI_envmap_bumpmap).

Each such extension adds a small set of relatively inflexible per-fragment computations.

This inflexibility is in contrast to the typical flexibility provided by the underlying programmable floating point engines (whether micro-coded fragment engines, DSPs, or CPUs) that are traditionally used to implement OpenGL's texturing computations. The purpose of this extension is to expose to the OpenGL application writer a significant degree of per-fragment programmability for computing fragment parameters.

For the purposes of discussing this extension, a fragment program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters (not specific to an individual fragment) and an input set of per-fragment parameters are transformed to a set of per-fragment result parameters.

The per-fragment computations for standard OpenGL given a particular set of texture and fog application modes (along with any state for extensions defining per-fragment computations) is, in essence, a fragment program. However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining fragment program instruction sequences for application-defined fragment programs. In order to define such fragment programs, this extension defines a fragment programming model including a floating-point 4-component vector instruction set and a relatively large set of floating-point 4-component registers.

The extension's fragment programming model is designed for efficient hardware implementation and to support a wide variety of fragment programs. By design, the entire set of existing fragment programs defined by existing OpenGL per-fragment computation extensions can be implemented using the extension's fragment programming model.

**Issues**

This extension is closely related to ARB_vertex_program, and is in sync with revision 36 of that spec. ARB_fragment_program will continue to track changes made to ARB_vertex_program.

*(1) Should we provide precision queries?*

RESOLVED: We've decided not to include precision queries. Implementations are expected to meet or exceed the precision guidelines set forth in the core GL spec, section 2.1.1, p. 6, as ammended by this extension.

To summarize section 2.1.1, the maximum representable magnitude of colors must be at least 2^10, while the maximum representable magnitude of other floating-point values must be at least 2^32. The individual results of floating-point operations must be accurate to about 1 part in 10^5.

Here are the reasons why precision queries were not included:

1. It is unclear what the queries should be:
   a) min, max, [0,1) granularity
   b) min +, max +, min -, max -, [0,1) granularity
   c) IEEE mantissa bits, IEEE exponent bits
2. Due to instruction emulation, there is no way to query the actual precision that can be expected. Should the query return the best-case or worst-case precision?
3. Implementations may support multiple precisions, on a per-instruction basis or across the board. How would this be exposed?
4. Current implementations are able to meet the minimum requirements specified in the core GL, thanks to its sufficiently loose wording "... so that the individual results of floating-point operations are accurate to ABOUT 1 part in 10^5." (Emphasis added.)
5. A conformance test can act as watchdog to ensure implementations are not cutting corners on precision.
6. Adding precision queries would require a new entrypoint.

See issue 22 regarding reduced-precision modes.

*(2) Should the LOD biased texture sample be optional?*

RESOLVED: TXB support is mandatory. This exposes useful functionality which enables blurring and sharpening effects. It will be more useful to entirely override derivatives (scale factor) rather than just biasing the level-of-detail. This would be a future extension to fragment programs.

It should be noted here that the bias introduced per-fragment by TXB is added to any per-object or per-stage LOD bias. If per-fragment LOD bias is not necessary, using the per-object and/or per-stage LOD biases may perform better.

*(3) Should we include the ability to bind to the color matrix? How about others? Program matrices?*

RESOLVED: We will not specifically add anything that depends on the ARB_imaging subset. So we have not included matrix bindings to the color matrix (or parameter bindings to the color biases, etc.). However, we have included matrix binding support and support for all of the matrices present in ARB_vertex_program.

*(4) Should we include the ability to bind to just a texcoord attribute's S,T components? (Or just S, or S,T,P for that matter?)*

RESOLVED: No. Issue #15 below obviates this issue by making the texture coordinate usage within a program explicit, thereby making optimizations to reduce the number of interpolated texture coordinates something an implementation can do at compile time instead of having to do it during every texture target change.

*(5) What other instructions should be added? Should any be removed?*

RESOLVED: The differences between the ARB_vertex_program instruction set and the ARB_fragment_program instruction set are minimal. ARB_fragment_program removes the LOG and EXP rough approximation instructions and the ARL address register load instruction. ARB_fragment_program adds the SIN/COS/SCS trigonometric instructions, the LRP linear interpolation instruction, the CMP compare instruction, and the TEX/TXP/TXB/KIL texture instructions.

*(6) Should depth output be a program option or a mandatory feature?*

RESOLVED: Depth output capability should be mandatory.

*(6a) How should per-vertex geometric depth clipping be handled when replacing depth in a fragment program?*

RESOLVED: Per-vertex geometric depth clipping should be performed by the GL as usual, so no spec change is required. The ideal behavior would be to disable near and far clipping planes when replacing depth, but not all implementations can natively support disabling individual clip planes.

*(6b) How should depth output from the fragment program be further processed before being handed to the per-fragment operations?*

RESOLVED: Depth gets clamped by GL to [0,1]. App has access to depth range as a bindable parameter if it wants to either scale and bias its depth to fall within the depth range, or to kill fragments outside the depth range.

*(7) If a fragment program does not write a color value, what should be the final color of the fragment*?

RESOLVED: The final fragment color is undefined. Note that it may be perfectly reasonable to have a program that computes depth values but not colors. Fragment colors are often irrelevant if color writes are disabled (via ColorMask).

*(7a) If a fragment program does not write a depth value, what should be the final depth value of the fragment?*

RESOLVED: "Depth fly-over" (using the conventional depth produced by rasterization) should happen whenever a depth-replacing program is not in use. A depth-replacing program is defined as a program that writes to result.depth in at least one instruction. The presence of a depth declaration alone DOES NOT designate a depth-replacing program. The intention is that a future extension introducing conditional execution will still consider a program to be depth-replacing even if the instruction(s) writing to result.depth do(es) not execute.

Other considered definitions of depth-replacing program:

1. The presence of a depth declaration -OR- the use of result.depth as an instruction destination anywhere in the program designates a depth-replacing program.
2. Every program is a depth-replacing program, but the GL initializes the depth output to be the depth produced by rasterization. The app may then overwrite the depth output.
3. Every program is a depth-replacing program, and the app is solely responsible for copying the depth input to depth output if desired.

*(8) Should relative addressing, like that defined in ARB_vertex_program, be supported in this spec?*

RESOLVED: No, relative addressing won't be included in this spec.

*(9) Should full-featured operand component swizzling, like that defined in ARB_vertex_program, be supported in this spec?*

RESOLVED: Yes, full swizzling is mandatory.

*(10) Should texture instructions contain specific limitations on operations that can be performed? For example, should write masks or operand component swizzling be disallowed?*

RESOLVED: Texture instructions are specified to be very similar to ALU instructions. They have been given 3-letter names, they allow writemasking and saturation (which would be useful for floating-point texture formats), source swizzles and negates, and the ability to use parameters as sources.

*(11) Should we standardize options for stencil or aux data buffer outputs?*

RESOLVED: Stencil and aux data buffers will be saved for a possible future extension to fragment programs.

*(12) Should depth output be pulled from the 3rd or 4th component?*

RESOLVED: 3rd component, as the 3rd component is also used for depth input from the "fragment.position" attribute.

*(13) Which stages are subsumed by fragment programs?*

RESOLVED: Texturing, color sum, and fog.

*(14) What should the minimum resource limits be?*

RESOLVED: 10 attributes, 24 parameters, 4 texture indirections, 48 ALU instructions, 24 texture instructions, and 16 temporaries.

*(15) OpenGL provides a hierarchy of texture enables (cube map, 3D, 2D, 1D). Should the texture sampling instructions here override that hierarchy and select specific texture targets?*

RESOLVED: Yes. This removes a potential pitfall for developers: leaving the hierarchy of enables in an undesired state. It makes programs more readable as the intent of the sample is more obvious. Finally, it allows compilers to be more aggressive as to which texcoord components are "don't cares" without having to recompile programs when fixed-function texenables change. One drawback is that programs cannot be reused for both 2D and 3D texturing, for example, by simply changing the texture enables.

Texture sampling can be specified by instructions like

```
TEX myTexel, fragment.texcoord[1], texture[2], 3D;
```

which would indicate to use texture coordinate set number 1 to sample from the texture object bound to the TEXTURE_3D target on texture image unit 2.

Each texture unit can have only one "active" target. Programs are not allowed to reference different texture targets in the same texture image unit. In the example above, any other texture instructions using texture image unit 2 must specify the 3D texture target.

Note that every texture image unit always has a texture bound to every texture target, whether it is a named texture object or a default texture. However, the texture may not be complete as defined in section 3.8.9 of the core GL spec. See issue 23.

*(16) Should aux texture units be additional units on top of existing full-featured texture units, or should this spec fully deprecate "legacy" texture units and only expose texture coordinate sets and texture image units?*

Background: Some implementations are able to expose more "texture image units" (texture maps and associated parameters) than "texture coordinate sets" (current texcoords, texgen, and texture matrices). A conventional GL "texture unit" encompasses both a texture image unit and a texture coordinate set as well as texture environment state.

RESOLVED: Yes, deprecate "legacy" texture units. This is a more flexible model.

*(17) Should fragment programs affect all fragments, or just those produced by the rasterization of points, lines, and triangles?*

RESOLVED: Every fragment generated by the GL is subject to fragment program mode. This includes point, line, and polygon primitives as well as pixel rectangles and bitmaps.

*(18) Should per-fragment position and fogcoord be bindable as fragment attributes?*

RESOLVED: Yes, interpolated fogcoord will make per-fragment fog application possible, in addition to full fog stage subsummation. Interpolated window position, especially depth, enables interesting depth-replacing algorithms.

*(19) What characters should be used to identify individual components in swizzle selectors and write masks?*

RESOLVED: ARB_vertex_program provides "xyzw". This extension supports "xyzw" and also provides "rgba" for better readability when dealing with RGBA color values. Adding support for special identifiers for dealing with texture coordinates was considered and rejected. "strq" could be used to identify texture coordinate components, but the "r" would conflict with the "r" from "rgba". "stpq" would be another possibility, but could be a source of confusion.

*(20) Should implementations be required to support all programs that fit within the exported limits on the number of resources (e.g., instructions, temporaries) that can be present in a program, even if it means falling back to software? Should implementations be required to reject programs that could never be accelerated?*

RESOLVED: No and no. An implementation is allowed to fail ProgramStringARB due to the program exceeding native resources. Note that this failure must be invariant with respect to all other OpenGL state. In other words, a program cannot succeed to load with default state, but then fail to load when certain GL state is altered. However, an implementation is not required to fail when a program would exceed native resources, and is in fact encouraged to fallback to a software path. See issue 21 for a way of determining if this has happened.

This notable departure from ARB_vertex_program was made as an accommodation to vendors who could not justify implementing a software fallback path which would be relatively slow even compared to an ARB_vertex_program software fallback path.

Two issues with this decision:

1. The API limits become hints, and one can no longer tell by visual inspection whether or not a program will load on every implementation.
2. Program loading will now depend on the optimizer, which may vary from release to release of an implementation. A program that succeeded to load when an ISV first wrote it may fail to load in a future driver version, and vice versa.

*(21) How can applications determine if their programs are too large to run on the native (likely hardware) implementation, and therefore may run with reduced performance?*

RESOLVED: The following code snippet uses a native resource query to guarantee a program is loaded natively (or not at all):

```
GLboolean ProgramStringIsNative(GLenum target, GLenum format,
                                GLsizei len, const GLvoid *string)
{
    GLint errorPos, isNative;
    glProgramStringARB(target, format, len, string);
    glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, &errorPos);
    glGetProgramivARB(GL_FRAGMENT_PROGRAM_ARB,
        GL_PROGRAM_UNDER_NATIVE_LIMITS_ARB, &isNative);
    if ((errorPos == -1) && (isNative == 1))
        return GL_TRUE;
    else
        return GL_FALSE;
}
```

Note that a program that successfully loads, and falls under the native limits, is still not guaranteed to execute in hardware. Lack of other resources (e.g., texture memory) or the use of other OpenGL features not natively supported by the implementation (e.g., textures with borders) may also prevent the program from executing in hardware.

*(22) Should we provide applications with a method to control the level of precision used to carry out fragment program computations?*

RESOLVED: Yes. The GL implementation ultimately has control over the level of precision used for fragment program computations. However, the "ARB_precision_hint_fastest" and "ARB_precision_hint_nicest" program options allow applications to guide the GL implementation in its precision selection. The "fastest" option encourages the GL to minimize execution time, with possibly reduced precision. The "nicest" option encourages the GL to maximize precision, with possibly increased execution time.

If the precision hint is not "fastest", GL implementations should perform low-precision operations only if they could not appreciably affect the final results of the program. Regardless of the precision hint, GL implementations are discouraged from reducing the precision of computations so aggressively that final rendering results could be seriously compromised due to overflow of intermediate values or insufficient number of mantissa bits.

Some implementations may provide only a single level of precision, in which case these hints may have no effect. However, all implementations will accept these options, even if they are silently ignored.

More explicit control of precision, such as provided in "C" with data types such as "short", "int", "float", "double", may also be a desirable feature, but this level of detail is left to a separate extension.

*(23) What is the result of a sample from an incomplete texture? The definition of texture completeness can be found in section 3.8.9 of the core GL spec.*

RESOLVED: The result of a sample from an incomplete texture is the constant vector (0,0,0,1). The benefit of defining the result to be a constant is that broken apps are guaranteed to generate unexpected (black) results from their bad samples. If we were to leave the result undefined, some implementations may generate expected results some of the time, for example when magfiltering, giving app developers a false sense of correctness in their apps.

*(24) What is a texture indirection, and how is it counted?*

RESOLVED: On some implementations, fragment programs that have complex texture dependency chains may not be supported, even if the instruction counts fit within the exported limits. A texture dependency occurs when a texture instruction depends on the result of a previous instruction (ALU or texture) for use as its texture coordinate.

A texture indirection can be considered a node in the texture dependency chain. Each node contains a set of texture instructions which execute in parallel, followed by a sequence of ALU instructions. A dependent texture instruction is one that uses a temporary as an input coordinate rather than an attribute or a parameter. A program with no dependent texture instructions (or no texture instructions at all) will have a single node in its texture dependency chain, and thus a single indirection.

API-level texture indirections are counted by keeping track of which temporaries are read and written within the current node in the texture dependency chain. When a texture instruction is encountered, an indirection may be added and a new node started if either of the following two conditions is true:

1. the source coordinate of the texture instruction is a temporary that has already been written in the current node, either by a previous texture instruction or ALU instruction;

2. the result of the texture instruction is a temporary that has already been read or written in the current node by an ALU instruction.

The texture instruction provoking a new indirection and all subsequent instructions are added to the new node. This process is repeated until the end of the program is encountered. Below is some pseudo-code to describe this:

```
indirections = 1;
tempsOutput = 0;
aluTemps = 0;
while (i = getInst())
{
  if (i.type == TEX)
  {
    if (((i.input.type == TEMP) &&
          (tempsOutput & (1 << i.input.index))) ||
        ((i.op != KILL) && (i.output.type == TEMP) &&
          (aluTemps & (1 << i.output.index))))
    {
      indirections++;
      tempsOutput = 0;
      aluTemps = 0;
    }
  } else {
    if (i.input1.type == TEMP)
      aluTemps |= (1 << i.input1.index);
    if (i.input2 && i.input2.type == TEMP)
      aluTemps |= (1 << i.input2.index);
    if (i.input3 && i.input3.type == TEMP)
      aluTemps |= (1 << i.input3.index);
    if (i.output.type == TEMP)
      aluTemps |= (1 << i.output.index);
  }
  if ((i.op != KILL) && (i.output.type == TEMP))
    tempsOutput |= (1 << i.output.index);
}
```

For example, the following programs would have 1, 2, and 3 texture indirections, respectively:

```
  !!ARBfp1.0
  # No texture instructions, but always 1 indirection
  MOV result.color, fragment.color;
  END

  !!ARBfp1.0
  # A simple dependent texture instruction, 2 indirections
  TEMP myColor;
  MUL myColor, fragment.texcoord[0], fragment.texcoord[1];
  TEX result.color, myColor, texture[0], 2D;
  END

  !!ARBfp1.0
  # A more complex example with 3 indirections
  TEMP myColor1, myColor2;
  TEX myColor1, fragment.texcoord[0], texture[0], 2D;
  MUL myColor1, myColor1, myColor1;
  TEX myColor2, fragment.texcoord[1], texture[1], 2D;
  # so far we still only have 1 indirection
  TEX myColor2, myColor1, texture[2], 2D;      # This is #2
  TEX result.color, myColor2, texture[3], 2D;  # And #3
  END
```

Note that writemasks for the temporaries written and swizzles for the temporaries read are not taken into consideration when counting indirections. This makes hand-counting of indirections by a developer an easier task.

Native texture indirections may be counted differently by an implementation to reflect its exact restrictions, to reflect the true dependencies taking into account writemasks and swizzles, and to reflect optimizations such as instruction reordering.

For implementations with no restrictions on the number of indirections, the maximum indirection count will equal the maximum texture instruction count.

*(25) How can a program reduce SCS's scalar operand to the fundamental period [-PI,PI]?*

RESOLVED: Unlike the individual SIN and COS instructions, SCS requires that its argument be reduced ahead of time to the fundamental period. The reason SCS doesn't perform this operation automatically is that it may make unnecessary redundant work for programs that already have their operand in the correct range. Other programs that do need to reduce their operand simply need to add a block of code before the SCS instruction:

```
PARAM myParams = { 0.5, -3.14159, 6.28319, 0.15915 };
MAD myOperand.x, myOperand.x, myParams.w, myParams.x; # a = (a/(2*PI))+0.5
FRC myOperand.x, myOperand.x;                         # a = frac(a)
MAD myOperand.x, myOperand.x, myParams.z, myParams.y  # a = (a*2*PI)-PI
...
SCS myResult, myOperand.x;
```

*(26) Is depth output from a fragment program guaranteed to be invariant with respect to depth produced via conventional rasterization?*

RESOLVED: No. The floating-point representation of depth values output from a fragment program may lead to the output of depth with less precision than the depth output by convention GL rasterization. For example, a floating-point representation with 16 bits of mantissa will certainly produce depth with lesser precision than that of conventional rasterization used in conjunction with a 24-bit depth buffer, where all values are maintained as integers. Be aware of this when mixing conventional GL rendering with fragment program rendering.

*(27) How can conventional GL fog application be achieved within a fragment program?*

RESOLVED: Program options have been introduced that allow a program to request fog to be applied to the final clamped fragment color before being passed along to the antialiasing application stage. This makes it easy for:
1. developers to request conventional fog behavior
2. implementations with dedicated fog hardware to use it
3. implementations without dedicated fog hardware, so they need not track fog state after compilation, and constantly recompile when fog state changes.

The three mandatory options are ARB_fog_exp, ARB_fog_exp2, and ARB_fog_linear. As these options are mutually exclusive by nature, specifying more than one is not useful. If more than one is specified, the last one encountered in the <optionSequence> will be the one to actually modify the execution environment.

*(28) Why have all of the enums, entrypoints, GLX protocol, and spec language shared with ARB_vertex_program been reproduced here?*

RESOLVED: The two extensions are independent of one another, in so far as an implementation need not support both of them in order to support one of them. Everything needed to implement or make use of ARB_fragment_program is present in this spec without the need to refer to the ARB_vertex_program spec. When and if these two extensions are incorporated into the core OpenGL, the significant overlap of the two will be collapsed into a single instance of the shared parts.

*(29) How might an implementation implement the fog options? To What does the extra resource consumption described in 3.11.4.5.1 correspond?*

RESOLVED: The following code snippets reflect possible implementations of the fog options. While an implementation may use other instruction sequences to achieve the same result, or may use external fog hardware if available, all implementations must enforce the API-level resource consumption as described: 2 params, 1 temp, 1 attribute, and 3, 4, or 2 instructions. "finalColor" in the examples below is the color that would otherwise be

```
      "result.color", with components clamped to the range [0,1].
      "result.color.a" is assumed to have already been written, as fog
      blending does not affect the alpha component.

      EXP:
        # Exponential fog
        # f = exp(-d*z)
        #
        PARAM p = {DENSITY/LN(2), NOT USED, NOT USED, NOT USED};
        PARAM fogColor = state.fog.color;
        TEMP fogFactor;
        ATTRIB fogCoord = fragment.fogcoord.x;
        MUL fogFactor.x, p.x, fogCoord.x;
        EX2_SAT fogFactor.x, -fogFactor.x;
        LRP result.color.rgb, fogFactor.x, finalColor, fogColor;

      EXP2:
        #
        # 2nd-order Exponential fog
        # f = exp(-(d*z)^2)
        #
        PARAM p = {DENSITY/SQRT(LN(2)), NOT USED, NOT USED, NOT USED};
        PARAM fogColor = state.fog.color;
        TEMP fogFactor;
        ATTRIB fogCoord = fragment.fogcoord.x;
        MUL fogFactor.x, p.x, fogCoord.x;
        MUL fogFactor.x, fogFactor.x, fogFactor.x;
        EX2_SAT fogFactor.x, -fogFactor.x;
        LRP result.color.rgb, fogFactor.x, finalColor, fogColor;

      LINEAR:
        #
        # Linear fog
        # f = (end-z)/(end-start)
        #
        PARAM p = {-1/(END-START), END/(END-START), NOT USED, NOT USED};
        PARAM fogColor = state.fog.color;
        TEMP fogFactor;
        ATTRIB fogCoord = fragment.fogcoord.x;
        MAD_SAT fogFactor.x, p.x, fogCoord.x, p.y;
        LRP result.color.rgb, fogFactor.x, finalColor, fogColor;
```

*(30) Why is the order of operands for the CMP instruction different than the order used by another popular graphics API?*

RESOLVED: No other graphics API was used as a basis for the design of ARB_fragment_program except ARB_vertex_program, which did not have a CMP instruction. This independent evolution naturally led to differences in minor details such as order of operands. This discrepancy is noted here to help developers familiar with the other API to avoid this potential pitfall.

*(31) Is depth offset applied to the window z value before it enters the fragment program?*

RESOLVED: As in the base OpenGL specification, the depth offset generated by polygon offset is added during polygon rasterization.

```
    The depth value provided to shaders in the fragment.position.z
    attribute already includes polygon offset, if enabled.  If the
    depth value is replaced by a fragment program, the polygon offset
    value will NOT be recomputed and added back after fragment program
    execution.

    NOTE: This is probably not desirable for fragment programs that
    modify depth values since the partials used to generate the offset
    may not match the partials of the computed depth value.
```

**New Procedures and Functions**

```
    void ProgramStringARB(enum target, enum format, sizei len,
                          const void *string);

    void BindProgramARB(enum target, uint program);

    void DeleteProgramsARB(sizei n, const uint *programs);

    void GenProgramsARB(sizei n, uint *programs);

    void ProgramEnvParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
    void ProgramEnvParameter4dvARB(enum target, uint index,
                                   const double *params);
    void ProgramEnvParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
    void ProgramEnvParameter4fvARB(enum target, uint index,
                                   const float *params);

    void ProgramLocalParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
    void ProgramLocalParameter4dvARB(enum target, uint index,
                                     const double *params);
    void ProgramLocalParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
    void ProgramLocalParameter4fvARB(enum target, uint index,
                                     const float *params);

    void GetProgramEnvParameterdvARB(enum target, uint index,
                                     double *params);
    void GetProgramEnvParameterfvARB(enum target, uint index,
                                     float *params);

    void GetProgramLocalParameterdvARB(enum target, uint index,
                                       double *params);
    void GetProgramLocalParameterfvARB(enum target, uint index,
                                       float *params);

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

    void GetProgramStringARB(enum target, enum pname, void *string);

    boolean IsProgramARB(uint program);
```

**New Tokens**

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of ProgramStringARB, BindProgramARB, ProgramEnvParameter4[df][v]ARB, ProgramLocalParameter4[df][v]ARB, GetProgramEnvParameter[df]vARB, GetProgramLocalParameter[df]vARB, GetProgramivARB and GetProgramStringARB.

| | |
|---|---|
| FRAGMENT_PROGRAM_ARB | 0x8804 |

Accepted by the <format> parameter of ProgramStringARB:

| | |
|---|---|
| PROGRAM_FORMAT_ASCII_ARB | 0x8875 |

Accepted by the <pname> parameter of GetProgramivARB:

| | |
|---|---|
| PROGRAM_LENGTH_ARB | 0x8627 |
| PROGRAM_FORMAT_ARB | 0x8876 |
| PROGRAM_BINDING_ARB | 0x8677 |
| PROGRAM_INSTRUCTIONS_ARB | 0x88A0 |
| MAX_PROGRAM_INSTRUCTIONS_ARB | 0x88A1 |
| PROGRAM_NATIVE_INSTRUCTIONS_ARB | 0x88A2 |
| MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB | 0x88A3 |
| PROGRAM_TEMPORARIES_ARB | 0x88A4 |
| MAX_PROGRAM_TEMPORARIES_ARB | 0x88A5 |
| PROGRAM_NATIVE_TEMPORARIES_ARB | 0x88A6 |
| MAX_PROGRAM_NATIVE_TEMPORARIES_ARB | 0x88A7 |
| PROGRAM_PARAMETERS_ARB | 0x88A8 |
| MAX_PROGRAM_PARAMETERS_ARB | 0x88A9 |
| PROGRAM_NATIVE_PARAMETERS_ARB | 0x88AA |
| MAX_PROGRAM_NATIVE_PARAMETERS_ARB | 0x88AB |
| PROGRAM_ATTRIBS_ARB | 0x88AC |
| MAX_PROGRAM_ATTRIBS_ARB | 0x88AD |
| PROGRAM_NATIVE_ATTRIBS_ARB | 0x88AE |
| MAX_PROGRAM_NATIVE_ATTRIBS_ARB | 0x88AF |
| MAX_PROGRAM_LOCAL_PARAMETERS_ARB | 0x88B4 |
| MAX_PROGRAM_ENV_PARAMETERS_ARB | 0x88B5 |
| PROGRAM_UNDER_NATIVE_LIMITS_ARB | 0x88B6 |
| PROGRAM_ALU_INSTRUCTIONS_ARB | 0x8805 |
| PROGRAM_TEX_INSTRUCTIONS_ARB | 0x8806 |
| PROGRAM_TEX_INDIRECTIONS_ARB | 0x8807 |
| PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB | 0x8808 |
| PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB | 0x8809 |
| PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB | 0x880A |
| MAX_PROGRAM_ALU_INSTRUCTIONS_ARB | 0x880B |
| MAX_PROGRAM_TEX_INSTRUCTIONS_ARB | 0x880C |
| MAX_PROGRAM_TEX_INDIRECTIONS_ARB | 0x880D |
| MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB | 0x880E |
| MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB | 0x880F |
| MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB | 0x8810 |

Accepted by the <pname> parameter of GetProgramStringARB:

| | |
|---|---|
| PROGRAM_STRING_ARB | 0x8628 |

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

| | |
|---|---|
| PROGRAM_ERROR_POSITION_ARB | 0x864B |
| CURRENT_MATRIX_ARB | 0x8641 |
| TRANSPOSE_CURRENT_MATRIX_ARB | 0x88B7 |
| CURRENT_MATRIX_STACK_DEPTH_ARB | 0x8640 |
| MAX_PROGRAM_MATRICES_ARB | 0x862F |
| MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB | 0x862E |
| | |
| MAX_TEXTURE_COORDS_ARB | 0x8871 |
| MAX_TEXTURE_IMAGE_UNITS_ARB | 0x8872 |

Accepted by the <name> parameter of GetString:

| | |
|---|---|
| PROGRAM_ERROR_STRING_ARB | 0x8874 |

Accepted by the <mode> parameter of MatrixMode:

| | |
|---|---|
| MATRIX0_ARB | 0x88C0 |
| MATRIX1_ARB | 0x88C1 |
| MATRIX2_ARB | 0x88C2 |
| MATRIX3_ARB | 0x88C3 |
| MATRIX4_ARB | 0x88C4 |
| MATRIX5_ARB | 0x88C5 |
| MATRIX6_ARB | 0x88C6 |
| MATRIX7_ARB | 0x88C7 |
| MATRIX8_ARB | 0x88C8 |
| MATRIX9_ARB | 0x88C9 |
| MATRIX10_ARB | 0x88CA |
| MATRIX11_ARB | 0x88CB |
| MATRIX12_ARB | 0x88CC |
| MATRIX13_ARB | 0x88CD |
| MATRIX14_ARB | 0x88CE |
| MATRIX15_ARB | 0x88CF |
| MATRIX16_ARB | 0x88D0 |
| MATRIX17_ARB | 0x88D1 |
| MATRIX18_ARB | 0x88D2 |
| MATRIX19_ARB | 0x88D3 |
| MATRIX20_ARB | 0x88D4 |
| MATRIX21_ARB | 0x88D5 |
| MATRIX22_ARB | 0x88D6 |
| MATRIX23_ARB | 0x88D7 |
| MATRIX24_ARB | 0x88D8 |
| MATRIX25_ARB | 0x88D9 |
| MATRIX26_ARB | 0x88DA |
| MATRIX27_ARB | 0x88DB |
| MATRIX28_ARB | 0x88DC |
| MATRIX29_ARB | 0x88DD |
| MATRIX30_ARB | 0x88DE |
| MATRIX31_ARB | 0x88DF |

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

### Modify Section 2.1.1, Floating-Point Computation (p. 6)

(modify first paragraph, p. 6) ... The maximum representable magnitude of a floating-point number used to represent position, normal, or texture coordinates must be at least 2^32; the maximum representable magnitude for colors must be at least 2^10. ...

### Modify Section 2.7, Vertex Specification (p. 19)

(modify second paragraph, p. 20) Implementations support more than one set of texture coordinates. The commands

```
  void MultiTexCoord{1234}{sifd}(enum texture, T coords);
  void MultiTexCoord{1234}{sifd}v(enum texture, T coords);
```

take the coordinate set to be modified as the <texture> parameter. <texture> is a symbolic constant of the form TEXTUREi, indicating that texture coordinate set i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to k-1, where k is the implementation-dependent number of texture units defined by MAX_TEXTURE_COORDS_ARB).

### Modify Section 2.8, Vertex Arrays (p. 21)

(modify first paragraph, p. 21) ... The client may specify up to 5 plus the value of MAX_TEXTURE_COORDS_ARB arrays: one each to store vertex coordinates...

(modify first paragraph, p. 23) The command

```
  void ClientActiveTexture(enum texture);
```

is used to select the vertex array client state parameters to be modified by the TexCoordPointer command and the array affected by EnableClientState and DisableClientState with parameter TEXTURE_COORD_ARRAY. This command sets the client state variable CLIENT_ACTIVE_TEXTURE. Each texture coordinate set has a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects the texture coordinate set state used for queries of client state.

(modify first paragraph, p. 28) If the number of supported texture coordinate sets (the value of MAX_TEXTURE_COORDS_ARB) is k, ...

### Modify Section 2.10.2, Matrices (p. 31)

(modify first paragraph, p. 31) The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIX<i>_ARB as the argument. In the case of MATRIX<i>_ARB, <i> is an integer between 0 and <n>-1 indicating one of <n> program matrices where <n> is the value of the implementation defined constant MAX_PROGRAM_MATRICES_ARB. Such program matrices are described in section 3.11.7. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix.

(modify first paragraph, p. 34) For each texture coordinate set, a 4x4 matrix is applied to the corresponding texture coordinates...

(modify first and second paragraphs, p. 35) The command

```
  void ActiveTexture(enum texture);
```

specifies the active texture unit selector, ACTIVE_TEXTURE. Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit (consisting of a texture matrix stack and texture coordinate generation state) and a texture image unit (consisting of all the texture state defined in Section 3.8). In implementations with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector specifies the texture coordinate set accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX_MODE is TEXTURE), TexGen (section 2.10.4), Enable/Disable (if any texture coordinate generation enum is selected), as well as queries of the current texture coordinates and current raster texture coordinates. If the texture coordinate set number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_COORDS_ARB, the error INVALID_OPERATION is generated by any such command.

The active texture unit selector also selects the texture image unit accessed by commands involving texture image processing (section 3.8). Such commands include all variants of TexEnv, TexParameter, and TexImage commands, BindTexture, Enable/Disable for any texture target (e.g., TEXTURE_2D), and queries of all such state. If the texture image unit number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_IMAGE_UNITS_ARB, the error INVALID_OPERATION is generated by any such command.

ActiveTexture generates the error INVALID_ENUM if an invalid <texture> is specified. <texture> is a symbolic constant of the form TEXTUREi, indicating that texture unit i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to k-1, where k is the larger of the MAX_TEXTURE_COORDS_ARB and MAX_TEXTURE_IMAGE_UNITS_ARB). For compatibility with old OpenGL specifications, the implementation-dependent constant MAX_TEXTURE_UNITS specifies the number of conventional texture units supported by the implementation.  Its value must be no larger than the minimum of MAX_TEXTURE_COORDS_ARB and MAX_TEXTURE_IMAGE_UNITS_ARB.

(modify last paragraph, p. 35) The state required to implement transformations consists of a <n>-value integer indicating the current matrix mode (where <n> is 4 + the number of supported texture and program matrices), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, <n> stacks (where <n> is at least 8) of at least one 4x4 matrix for each MATRIX<i>_ARB with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW.  Initially, there is only one matrix on each stack, and all matrices are set to the identity.  The initial matrix mode is MODELVIEW.  The initial value of ACTIVE_TEXTURE is TEXTURE0.

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

Modify Chapter 3, Introduction (p. 58)

(modify first paragraph, p. 58) ... Figure 3.1 diagrams the rasterization process.  The color value assigned to a fragment is initially determined by the rasterization operations (sections 3.3 through 3.7) and modified by either the execution of the texturing, color sum, and fog operations as defined in sections 3.8, 3.9, and 3.10, or of a fragment program defined in section 3.11.  The final depth value is initially determined by the rasterization operations and may be modified or replaced by a fragment program.

(modify Figure 3.1)

```
             _ +---------------+   FRAGMENT_PROGRAM_ARB
             /||    Point      |          enable
            /  | Rasterization |\           |
           /   +---------------+ \          V  o-------------+
   From   /    +---------------+  \                          |
Primitive ---> |     Line      |---+++--->o    o             |
 Assembly \    | Rasterization |  / ||         |             |
           \   +---------------+ /  ||         |             |
            \  +---------------+/   ||   +-----+-----+  +----+-----+
             \||    Polygon    |    ||   | Texturing |  | Fragment |
             - | Rasterization |   / |   +-----+-----+  | Program  |
               +---------------+  /  |         |        +----+-----+
               +---------------+ /   |   +-----+-----+       |
               |     Pixel     |/    |   | Color Sum |       |
DrawPixels --> |   Rectangle   |    /    +-----+-----+       |
               | Rasterization |   /           |             V
               +---------------+  /      +-----+-----+
               +---------------+ /       |    Fog    |---> Fragments
  Bitmap ----> |    Bitmap     |/        +-----------+
               | Rasterization |
               +---------------+
```

**Modify Section 3.3, Points (p. 63)**

(modify first and second paragraphs, p. 64) All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point. (delete reference to divide by q)

If antialiasing is enabled, then ... The data associated with each fragment are otherwise the data associated with the point being rasterized. (delete reference to divide by q)

**Modify Section 3.4.1, Basic Line Segment Rasterization (p. 66)**

(modify first paragraph, p. 68) ... (Note that t=0 at p_a and t=1 at p_b). The value of an associated datum f from the fragment center, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode) or the s, t, r, or q texture coordinate or the clip w coordinate (the depth value, window z, must be found using equation 3.3, below), is found as

$$f = \frac{(1-t)*(f\_a/w\_a) + t*(f\_b/w\_b)}{(1-t)*(1/w\_a) + t*(1/w\_b)} \tag{3.2}$$

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. Note that linear interpolation would use

$$f = (1-t)*f\_a + t*f\_b. \tag{3.3}$$

... A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to inacceptable distortion effects when interpolating texture coordinates or clip w coordinates.

**Modify Section 3.5.1, Basic Polygon Rasterization (p. 73)**

(modify third and fourth paragraphs, p. 74) Denote a datum at p_a, p_b, or p_c as f_a, f_b, or f_c, respectively. Then the value f of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{a*(f\_a/w\_a) + b*(f\_b/w\_b) + c*(f\_c/w\_c)}{a*(1/w\_a) + b*(1/w\_b) + c*(1/w\_c)} \tag{3.4}$$

where w_a, w_b, and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the ... at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

$$f = a*f\_a + b*f\_b + c*f\_c;$$

this may yield ... for texture coordinates or clip w coordinates.

**Modify Section 3.6.4, Rasterization of Pixel Rectangles (p. 91)**

(modify third paragraph, p. 103) A fragment arising from a group ... the color and texture coordinates are given by those associated with the current raster position. (delete reference to divide by q) Groups arising from DrawPixels...

**Modify Section 3.7, Bitmaps (p. 113)**

(modify third paragraph, p. 114) Otherwise, a rectangular array ... The associated data for each fragment are those associated with the current raster position. (delete reference to divide by q) Once the fragments have been produced ...

**Modify Section 3.8, Texturing (p. 115)**

(add new paragraphs before first paragraph, p. 115) Texture coordinate sets are mapped to RGBA colors for application to primitives in one of two modes. The first mode, described in this and subsequent sections, is GL's conventional multitexture pipeline, describing texture environment and texture application. The second mode, referred to as fragment program mode and described in section 3.11, applies textures, color sum, and fog as specified in an application-supplied fragment program.

The fragment program mode is enabled and disabled using the generic Enable and Disable commands, respectively, with the symbolic constant FRAGMENT_PROGRAM_ARB. The required state is one bit indicating whether the fragment program mode is enabled or disabled. In the initial state, the fragment program mode is disabled. When fragment program mode is enabled, texturing, color sum, and fog application stages are ignored and a general purpose program is executed instead.

(modify first and second paragraph, p. 115) Conventional texturing is employed when fragment program mode is disabled. Texturing maps ... color of an image at the location indicated by a fragment's texture coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color.

An implementation may support texturing using more than one image at a time. In this case the fragment carries multiple sets of texture coordinates which are used to index ...

(add paragraph before 1st paragraph, p. 116) Except when in fragment program mode (section 3.11), the (s,t,r) texture coordinates used for texturing are the values s/q, t/q, and r/q, respectively, where s, t, r, and q are the texture coordinates associated with the fragment. When in fragment program mode, the (s,t,r) texture coordinates are specified by the program. If q is less than or equal to zero, the results of texturing are undefined.

**Modify Section 3.8.7, Texture Minification (p. 135)**

(add new paragraph after first paragraph, p. 137) When fragment program mode is enabled, the derivatives of the coordinates may be ill-defined or non-existent. As a result, the implementation is free to approximate these derivatives with such techniques as differencing. The only requirement is that texture samples be equivalent across the two modes. In other words, the texture sample chosen for a fragment of a primitive must be invariant between fragment program mode and conventional mode subject to the rules set forth in Appendix A, Invariance.

**Modify Section 3.8.13, Texture Application (p. 149)**

(modify fourth paragraph, p. 152) Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by MAX_TEXTURE_UNITS may be incomplete and are always treated as disabled.

Insert a new Section 3.11, (p. 154), between existing sections 3.10 and 3.11. Renumber 3.11, Antialiasing Application, to 3.12.

## 3.11 Fragment Programs

The conventional GL texturing model described in section 3.8 is a configurable but essentially hard-wired sequence of per-fragment computations based on a canonical set of per-fragment parameters and texturing-related state such as texture images, texture parameters, and texture environment parameters. The general success and utility of the conventional GL texturing model reflects its basic correspondence to the typical texturing requirements of 3D applications.

However when the conventional GL texturing model is not sufficient, the fragment program mode provides a substantially more flexible model for generating fragment colors. The fragment program mode permits applications to define their own fragment programs.

A fragment program is a character string that specifies a sequence of operations to perform. Fragment program instructions are typically 4-component vector operations that operate on per-fragment attributes and program parameters. Fragment programs execute on a per-fragment basis and operate on each fragment completely independently from any other fragments. Fragment programs execute a finite fixed sequence of instructions with no branching or looping. Fragment programs execute without data hazards so results computed in one instruction can be used immediately afterwards. The result of a fragment program is a set of fragment result registers that becomes the color used by antialiasing application and/or a depth value used in place of the interpolated depth value generated by conventional rasterization.

In fragment program mode, the color sum is subsumed by the fragment program. An application desiring the primary and secondary colors to be summed must explicitly include this operation in its program.

Fragment programs are defined to operate only in RGBA mode. The results of fragment program execution are undefined if the GL is in color index mode.

**3.11.1 Program Objects**

The GL provides one or more program targets, each identifying a portion of the GL that can be controlled through application-specified programs. The program target for fragment programs is FRAGMENT_PROGRAM_ARB. Each program target has an associated program object, called the current program object. Each program target also has a default program object, which is initially the current program object.

Each program object has an associated program string. The command

```
  ProgramStringARB(enum target, enum format, sizei len,
                   const void *string);
```

updates the program string for the current program object for <target>. <format> describes the format of the program string, which must currently be PROGRAM_FORMAT_ASCII_ARB. <string> is a pointer to the array of bytes representing the program string being loaded, which need not be null-terminated. The length of the array is given by <len>. If <string> is null-terminated, <len> should not include the terminator.

When a program string is loaded, it is interpreted according to syntactic and semantic rules corresponding to the program target specified by <target>. If a program violates the syntactic or semantic restrictions of the program target, ProgramStringARB generates the error INVALID_OPERATION. An implementation may also generate the error INVALID_OPERATION if the program would exceed the native resource limits defined in section 6.1.12. A program which fails to load due to exceeding native resource limits must always fail, regardless of any other GL state.

Additionally, ProgramString will update the program error position (PROGRAM_ERROR_POSITION_ARB) and error string (PROGRAM_ERROR_STRING_ARB). If a program fails to load, the value of the program error position is set to the ubyte offset into the specified program string indicating where the first program error was detected. If the program fails to load because of a semantic restriction that is not detected until the program is fully scanned, the error position is set to the value of <len>. If a program loads successfully, the error position is set to the value negative one. The implementation-dependent program error string contains one or more error or warning messages. If a program loads succesfully, the error string may either contain warning messages or be empty.

Each program object has an associated array of program local parameters. The number and type of program local parameters is target- and implementation-dependent. For fragment programs, program local parameters are four-component floating-point vectors. The number of vectors is given by the implementation-dependent constant MAX_PROGRAM_LOCAL_PARAMETERS_ARB, which must be at least 24. The commands

```
      void ProgramLocalParameter4fARB(enum target, uint index,
                                      float x, float y, float z, float w);
      void ProgramLocalParameter4fvARB(enum target, uint index,
                                       const float *params);
      void ProgramLocalParameter4dARB(enum target, uint index,
                                      double x, double y, double z, double w);
      void ProgramLocalParameter4dvARB(enum target, uint index,
                                       const double *params);
```

update the values of the program local parameter numbered <index> belonging to the program object currently bound to <target>. For ProgramLocalParameter4fARB and ProgramLocalParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramLocalParameter4fvARB and ProgramLocalParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program local parameters supported by <target>.

Additionally, each program target has an associated array of program environment parameters. Unlike program local parameters, program environment parameters are shared by all program objects of a given target. The number and type of program environment parameters is target- and implementation-dependent. For fragment programs, program environment parameters are four-component floating-point vectors. The number of vectors is given by the implementation-dependent constant MAX_PROGRAM_ENV_PARAMETERS_ARB, which must be at least 24. The commands

```
      void ProgramEnvParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
      void ProgramEnvParameter4fvARB(enum target, uint index,
                                     const float *params);
      void ProgramEnvParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
      void ProgramEnvParameter4dvARB(enum target, uint index,
                                     const double *params);
```

update the values of the program environment parameter numbered <index> for the given program target <target>. For ProgramEnvParameter4fARB and ProgramEnvParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramEnvParameter4fvARB and ProgramEnvParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of program environment parameters supported by <target>.

Each program target has a default program object. Additionally, named program objects can be created and operated upon. The name space for program objects is the positive integers and is shared by programs of all targets. The name zero is reserved by the GL.

A named program object is created by binding an unused program object name to a valid program target. The binding is effected by calling

```
  BindProgramARB(enum target, uint program);
```

with <target> set to the desired program target and <program> set to the unused program name. The resulting program object has a program target given by <target> and is assigned target-specific default values (see section 3.11.8 for fragment programs). BindProgramARB may also be used to bind an existing program object to a program target. If <program> is zero, the default program object for <target> is bound. If <program> is the name of an existing program object whose associated program target is <target>, the named program object is bound. The error INVALID_OPERATION is generated if <program> names an existing program object whose associated program target is anything other than <target>.

Programs objects are deleted by calling

```
  void DeleteProgramsARB(sizei n, const uint *programs);
```

<programs> contains <n> names of programs to be deleted. After a program object is deleted, its name is again unused. If a program object that is bound to any target is deleted, it is as though BindProgramARB is first executed with same target and a <program> of zero. Unused names in <programs> are silently ignored, as is the value zero.

The command

```
  void GenProgramsARB(sizei n, uint *programs);
```

returns <n> currently unused program names in <programs>. These names are marked as used, for the purposes of GenProgramsARB only, but objects are created only when they are first bound using BindProgramARB.

### 3.11.2  Fragment Program Grammar and Semantic Restrictions

Fragment program strings are specified as an array of ASCII characters containing the program text. When a fragment program is loaded by a call to ProgramStringARB, the program string is parsed into a set of tokens possibly separated by whitespace. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for fragment programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

A fragment program is required to begin with the header string "!!ARBfp1.0", without any preceding whitespace. This string

```
    identifies the subsequent program text as a fragment program
    (version 1.0) that should be parsed according to the following
    grammar and semantic rules.  Program string parsing begins with the
    character immediately following the header string.

    <program>               ::= <optionSequence> <statementSequence> "END"

    <optionSequence>        ::= <optionSequence> <option>
                              | ""

    <option>                ::= "OPTION" <identifier> ";"

    <statementSequence>     ::= <statementSequence> <statement>
                              | ""

    <statement>             ::= <instruction> ";"
                              | <namingStatement> ";"

    <instruction>           ::= <ALUInstruction>
                              | <TexInstruction>

    <ALUInstruction>        ::= <VECTORop_instruction>
                              | <SCALARop_instruction>
                              | <BINSCop_instruction>
                              | <BINop_instruction>
                              | <TRIop_instruction>
                              | <SWZ_instruction>

    <TexInstruction>        ::= <SAMPLE_instruction>
                              | <KIL_instruction>

    <VECTORop_instruction> ::= <VECTORop> <maskedDstReg> ","
                                <vectorSrcReg>

    <VECTORop>              ::= "ABS" | "ABS_SAT"
                              | "FLR" | "FLR_SAT"
                              | "FRC" | "FRC_SAT"
                              | "LIT" | "LIT_SAT"
                              | "MOV" | "MOV_SAT"

    <SCALARop_instruction> ::= <SCALARop> <maskedDstReg> ","
                                <scalarSrcReg>

    <SCALARop>              ::= "COS" | "COS_SAT"
                              | "EX2" | "EX2_SAT"
                              | "LG2" | "LG2_SAT"
                              | "RCP" | "RCP_SAT"
                              | "RSQ" | "RSQ_SAT"
                              | "SIN" | "SIN_SAT"
                              | "SCS" | "SCS_SAT"

    <BINSCop_instruction>  ::= <BINSCop> <maskedDstReg> ","
                                <scalarSrcReg> "," <scalarSrcReg>

    <BINSCop>               ::= "POW" | "POW_SAT"
    <BINop_instruction>    ::= <BINop> <maskedDstReg> ","
                                <vectorSrcReg> "," <vectorSrcReg>

    <BINop>                ::= "ADD" | "ADD_SAT"
                              | "DP3" | "DP3_SAT"
                              | "DP4" | "DP4_SAT"
                              | "DPH" | "DPH_SAT"
                              | "DST" | "DST_SAT"
                              | "MAX" | "MAX_SAT"
                              | "MIN" | "MIN_SAT"
                              | "MUL" | "MUL_SAT"
                              | "SGE" | "SGE_SAT"
                              | "SLT" | "SLT_SAT"
                              | "SUB" | "SUB_SAT"
                              | "XPD" | "XPD_SAT"

    <TRIop_instruction>    ::= <TRIop> <maskedDstReg> ","
                                <vectorSrcReg> "," <vectorSrcReg> ","
                                <vectorSrcReg>

    <TRIop>                ::= "CMP" | "CMP_SAT"
                              | "LRP" | "LRP_SAT"
                              | "MAD" | "MAD_SAT"

    <SWZ_instruction>      ::= <SWZop> <maskedDstReg> ","
                                <srcReg> "," <extendedSwizzle>

    <SWZop>                ::= "SWZ" | "SWZ_SAT"

    <SAMPLE_instruction>   ::= <SAMPLEop> <maskedDstReg> ","
                                <vectorSrcReg> "," <texImageUnit> ","
                                <texTarget>

    <SAMPLEop>             ::= "TEX" | "TEX_SAT"
                              | "TXP" | "TXP_SAT"
                              | "TXB" | "TXB_SAT"

    <KIL_instruction>      ::= "KIL" <vectorSrcReg>

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

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

    <optTexImageUnitNum>   ::= ""
                              | "[" <texImageUnitNum> "]"

    <texImageUnitNum>      ::= <integer> from 0 to
                                MAX_TEXTURE_IMAGE_UNITS_ARB-1

    <scalarSrcReg>         ::= <optionalSign> <srcReg> <scalarSuffix>

    <vectorSrcReg>         ::= <optionalSign> <srcReg> <optionalSuffix>
    <maskedDstReg>          ::= <dstReg> <optionalMask>

    <extendedSwizzle>       ::= <xyzwExtendedSwizzle>
                              | <rgbaExtendedSwizzle>

    <xyzwExtendedSwizzle>   ::= <xyzwExtSwizComp> "," <xyzwExtSwizComp> ","
                                <xyzwExtSwizComp> "," <xyzwExtSwizComp>

    <rgbaExtendedSwizzle>   ::= <rgbaExtSwizComp> "," <rgbaExtSwizComp> ","
                                <rgbaExtSwizComp> "," <rgbaExtSwizComp>

    <xyzwExtSwizComp>       ::= <optionalSign> <xyzwExtSwizSel>

    <rgbaExtSwizComp>       ::= <optionalSign> <rgbaExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                              | "1"
                              | <xyzwComponent>

    <rgbaExtSwizSel>        ::= "0"
                              | "1"
                              | <rgbaComponent>

    <srcReg>                ::= <fragmentAttribReg>
                              | <temporaryReg>
                              | <progParamReg>

    <dstReg>                ::= <temporaryReg>
                              | <fragmentResultReg>

    <fragmentAttribReg>     ::= <establishedName>
                              | <fragAttribBinding>

    <temporaryReg>          ::= <establishedName>

    <progParamReg>          ::= <progParamSingle>
                              | <progParamArray> "[" <progParamArrayAbs> "]"
                              | <paramSingleItemUse>

    <progParamSingle>       ::= <establishedName>

    <progParamArray>        ::= <establishedName>

    <progParamArrayAbs>     ::= <integer>

    <fragmentResultReg>     ::= <establishedName>
                              | <resultBinding>

    <scalarSuffix>          ::= "." <component>

    <optionalSuffix>        ::= ""
                              | "." <component>
                              | "." <xyzwComponent> <xyzwComponent>
                                    <xyzwComponent> <xyzwComponent>
                              | "." <rgbaComponent> <rgbaComponent>
                                    <rgbaComponent> <rgbaComponent>
    <component>             ::= <xyzwComponent>
                              | <rgbaComponent>

    <xyzwComponent>         ::= "x" | "y" | "z" | "w"

    <rgbaComponent>         ::= "r" | "g" | "b" | "a"

    <optionalMask>          ::= ""
                              | <xyzwMask>
                              | <rgbaMask>

    <xyzwMask>              ::= "." "x"
                              | "." "y"
                              | "." "xy"
                              | "." "z"
                              | "." "xz"
                              | "." "yz"
                              | "." "xyz"
                              | "." "w"
                              | "." "xw"
                              | "." "yw"
                              | "." "xyw"
                              | "." "zw"
                              | "." "xzw"
                              | "." "yzw"
                              | "." "xyzw"

    <rgbaMask>              ::= "." "r"
                              | "." "g"
                              | "." "rg"
                              | "." "b"
                              | "." "rb"
                              | "." "gb"
                              | "." "rgb"
                              | "." "a"
                              | "." "ra"
                              | "." "ga"
                              | "." "rga"
                              | "." "ba"
                              | "." "rba"
                              | "." "gba"
                              | "." "rgba"

    <namingStatement>       ::= <ATTRIB_statement>
                              | <PARAM_statement>
                              | <TEMP_statement>
                              | <OUTPUT_statement>
                              | <ALIAS_statement>

    <ATTRIB_statement>      ::= "ATTRIB" <establishName> "="
                                  <fragAttribBinding>

    <fragAttribBinding>     ::= "fragment" "." <fragAttribItem>
    <fragAttribItem>       ::= "color" <optColorType>
                             | "texcoord" <optTexCoordNum>
                             | "fogcoord"
                             | "position"

    <PARAM_statement>      ::= <PARAM_singleStmt>
                             | <PARAM_multipleStmt>

    <PARAM_singleStmt>     ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>   ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                   <paramMultipleInit>

    <optArraySize>         ::= ""
                             | <integer> from 1 to MAX_PROGRAM_PARAMETERS_ARB
                                 (maximum number of allowed program
                                  parameter bindings)

    <paramSingleInit>      ::= "=" <paramSingleItemDecl>

    <paramMultipleInit>    ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>    ::= <paramMultipleItem>
                             | <paramMultipleItem> "," <paramMultInitList>

    <paramSingleItemDecl>  ::= <stateSingleItem>
                             | <programSingleItem>
                             | <paramConstDecl>

    <paramSingleItemUse>   ::= <stateSingleItem>
                             | <programSingleItem>
                             | <paramConstUse>

    <paramMultipleItem>    ::= <stateMultipleItem>
                             | <programMultipleItem>
                             | <paramConstDecl>

    <stateMultipleItem>    ::= <stateSingleItem>
                             | "state" "." <stateMatrixRows>

    <stateSingleItem>      ::= "state" "." <stateMaterialItem>
                             | "state" "." <stateLightItem>
                             | "state" "." <stateLightModelItem>
                             | "state" "." <stateLightProdItem>
                             | "state" "." <stateTexEnvItem>
                             | "state" "." <stateFogItem>
                             | "state" "." <stateDepthItem>
                             | "state" "." <stateMatrixRow>

    <stateMaterialItem>    ::= "material" <optFaceType> "." <stateMatProperty>

    <stateMatProperty>     ::= "ambient"
                             | "diffuse"
                             | "specular"
                             | "emission"
                             | "shininess"
    <stateLightItem>       ::= "light" "[" <stateLightNumber> "]" "."
                                 <stateLightProperty>

    <stateLightProperty>   ::= "ambient"
                             | "diffuse"
                             | "specular"
                             | "position"
                             | "attenuation"
                             | "spot" "." <stateSpotProperty>
                             | "half"

    <stateSpotProperty>    ::= "direction"

    <stateLightModelItem>  ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>    ::= "." "ambient"
                             | <optFaceType> "." "scenecolor"

    <stateLightProdItem>   ::= "lightprod" "[" <stateLightNumber> "]"
                                 <optFaceType> "." <stateLProdProperty>

    <stateLProdProperty>   ::= "ambient"
                             | "diffuse"
                             | "specular"

    <stateLightNumber>     ::= <integer> from 0 to MAX_LIGHTS-1

    <stateTexEnvItem>      ::= "texenv" <optLegacyTexUnitNum> "."
                                 <stateTexEnvProperty>

    <stateTexEnvProperty>  ::= "color"

    <optLegacyTexUnitNum>  ::= ""
                             | "[" <legacyTexUnitNum> "]"

    <legacyTexUnitNum>     ::= <integer> from 0 to MAX_TEXTURE_UNITS-1

    <stateFogItem>         ::= "fog" "." <stateFogProperty>

    <stateFogProperty>     ::= "color"
                             | "params"

    <stateDepthItem>       ::= "depth" "." <stateDepthProperty>

    <stateDepthProperty>   ::= "range"

    <stateMatrixRow>       ::= <stateMatrixItem> "." "row" "["
                                  <stateMatrixRowNum> "]"

    <stateMatrixRows>      ::= <stateMatrixItem> <optMatrixRows>

    <optMatrixRows>        ::= ""
                             | "." "row" "[" <stateMatrixRowNum> ".."
                                  <stateMatrixRowNum> "]"

    <stateMatrixItem>      ::= "matrix" "." <stateMatrixName>
                               <stateOptMatModifier>
    <stateOptMatModifier>  ::= ""
                             | "." <stateMatModifier>

    <stateMatModifier>     ::= "inverse"
                             | "transpose"
                             | "invtrans"

    <stateMatrixRowNum>    ::= <integer> from 0 to 3

    <stateMatrixName>      ::= "modelview" <stateOptModMatNum>
                             | "projection"
                             | "mvp"
                             | "texture" <optTexCoordNum>
                             | "palette" "[" <statePaletteMatNum> "]"
                             | "program" "[" <stateProgramMatNum> "]"

    <stateOptModMatNum>    ::= ""
                             | "[" <stateModMatNum> "]"

    <stateModMatNum>       ::= <integer> from 0 to MAX_VERTEX_UNITS_ARB-1

    <optTexCoordNum>       ::= ""
                             | "[" <texCoordNum> "]"

    <texCoordNum>          ::= <integer> from 0 to MAX_TEXTURE_COORDS_ARB-1

    <statePaletteMatNum>   ::= <integer> from 0 to MAX_PALETTE_MATRICES_ARB-1

    <stateProgramMatNum>   ::= <integer> from 0 to MAX_PROGRAM_MATRICES_ARB-1

    <programSingleItem>    ::= <progEnvParam>
                             | <progLocalParam>

    <programMultipleItem>  ::= <progEnvParams>
                             | <progLocalParams>

    <progEnvParams>        ::= "program" "." "env"
                                 "[" <progEnvParamNums> "]"

    <progEnvParamNums>     ::= <progEnvParamNum>
                             | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>         ::= "program" "." "env"
                                 "[" <progEnvParamNum> "]"

    <progLocalParams>      ::= "program" "." "local"
                                 "[" <progLocalParamNums> "]"

    <progLocalParamNums>   ::= <progLocalParamNum>
                             | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>       ::= "program" "." "local"
                                 "[" <progLocalParamNum> "]"

    <progEnvParamNum>      ::= <integer> from 0 to
                               MAX_PROGRAM_ENV_PARAMETERS_ARB - 1
    <progLocalParamNum>    ::= <integer> from 0 to 
                               MAX_PROGRAM_LOCAL_PARAMETERS_ARB - 1

    <paramConstDecl>       ::= <paramConstScalarDecl>
                             | <paramConstVector>

    <paramConstUse>        ::= <paramConstScalarUse>
                             | <paramConstVector>

    <paramConstScalarDecl> ::= <signedFloatConstant>

    <paramConstScalarUse>  ::= <floatConstant>

    <paramConstVector>     ::= "{" <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> "}"
                             | "{" <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> ","
                                   <signedFloatConstant> "}"

    <signedFloatConstant>  ::= <optionalSign> <floatConstant>

    <floatConstant>        ::= see text

    <optionalSign>         ::= ""
                             | "-"
                             | "+"

    <TEMP_statement>       ::= "TEMP" <varNameList>

    <varNameList>          ::= <establishName>
                             | <establishName> "," <varNameList>

    <OUTPUT_statement>     ::= "OUTPUT" <establishName> "="
                                 <resultBinding>

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

    <optFaceType>          ::= ""
                             | "." "front"
                             | "." "back"

    <optColorType>         ::= ""
                             | "." "primary"
                             | "." "secondary"

    <ALIAS_statement>      ::= "ALIAS" <establishName> "="
                                 <establishedName>

    <establishName>        ::= <identifier>
```

```
<establishedName>      ::= <identifier>

<identifier>           ::= see text
```

The <integer> rule matches an integer constant. The integer consists of a sequence of one or more digits ("0" through "9").

The <floatConstant> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of one or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing.

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z"), digits ("0" through "9), underscores ("_"), or dollar signs ("$"); the first character must not be a number. Upper and lower case letters are considered different (names are case-sensitive). The following strings are reserved keywords and may not be used as identifiers:

```
ABS, ABS_SAT, ADD, ADD_SAT, ALIAS, ATTRIB, CMP, CMP_SAT, COS,
COS_SAT, DP3, DP3_SAT, DP4, DP4_SAT, DPH, DPH_SAT, DST, DST_SAT,
END, EX2, EX2_SAT, FLR, FLR_SAT, FRC, FRC_SAT, KIL, LG2,
LG2_SAT, LIT, LIT_SAT, LRP, LRP_SAT, MAD, MAD_SAT, MAX, MAX_SAT,
MIN, MIN_SAT, MOV, MOV_SAT, MUL, MUL_SAT, OPTION, OUTPUT, PARAM,
POW, POW_SAT, RCP, RCP_SAT, RSQ, RSQ_SAT, SIN, SIN_SAT, SCS,
SCS_SAT, SGE, SGE_SAT, SLT, SLT_SAT, SUB, SUB_SAT, SWZ, SWZ_SAT,
TEMP, TEX, TEX_SAT, TXB, TXB_SAT, TXP, TXP_SAT, XPD, XPD_SAT,
fragment, program, result, state, and texture.
```

The error INVALID_OPERATION is generated if a fragment program fails to load because it is not syntactically correct or for one of the semantic restrictions described in the following sections.

A successfully loaded fragment program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 3.11.5.

A successfully loaded program string replaces the program string previously loaded into the specified program object. If the OUT_OF_MEMORY error is generated by ProgramStringARB, no change is made to the previous contents of the current program object.

### 3.11.3 Fragment Program Variables

Fragment programs may access a number of different variables during their execution. The following sections define the variables that can be declared and used by a fragment program.

Explicit variable declarations allow a fragment program to establish a variable name that can be used to refer to a specified resource in subsequent instructions. A fragment program will fail to load if it declares the same variable name more than once or if it refers to a variable name that has not been previously declared in the program string.

Implicit variable declarations allow a fragment program to use the name of certain available resources by name.

**3.11.3.1 Fragment Attributes**

Fragment program attribute variables are a set of four-component floating-point vectors holding the attributes of the fragment being processed. Fragment attribute variables are read-only during fragment program execution.

Fragment attribute variables can be declared explicitly using the <ATTRIB_statement> grammar rule, or implicitly using the <fragAttribBinding> grammar rule in an executable instruction.

Each fragment attribute variable is bound to a single item of fragment state according to the <fragAttrBinding> grammar rule. The set of GL state that can be bound to a fragment attribute variable is given in Table X.1. Fragment attribute variables are initialized at each fragment program invocation with the current values of the bound state.

| Fragment Attribute Binding | Components | Underlying State |
|---|---|---|
| fragment.color | (r,g,b,a) | primary color |
| fragment.color.primary | (r,g,b,a) | primary color |
| fragment.color.secondary | (r,g,b,a) | secondary color |
| fragment.texcoord | (s,t,r,q) | texture coordinate, unit 0 |
| fragment.texcoord[n] | (s,t,r,q) | texture coordinate, unit n |
| fragment.fogcoord | (f,0,0,1) | fog distance/coordinate |
| fragment.position | (x,y,z,1/w) | window position |

Table X.1: Fragment Attribute Bindings. The "Components" column indicates the mapping of the state in the "Underlying State" column. Bindings containing "[n]" require an integer value of <n> to select an individual item.

If a fragment attribute binding matches "fragment.color" or "fragment.color.primary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment color. Each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant.

If a fragment attribute binding matches "fragment.color.secondary", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "r", "g", "b", and "a" components, respectively, of the fragment secondary color. Each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant.

If a fragment attribute binding matches "fragment.texcoord" or "fragment.texcoord[n]", the "x", "y", "z", and "w" components of the fragment attribute variable are filled with the "s", "t", "r", and "q" components, respectively, of the fragment texture coordinates for texture unit <n>. If "[n]" is omitted, texture unit zero is used.

If a fragment attribute binding matches "fragment.fogcoord", the "x" component of the fragment attribute variable is filled with either the fragment eye distance or the fog coordinate, depending on whether the fog source is set to FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT, respectively. The "y", "z", and "w" coordinates are filled with 0, 0, and 1, respectively.

If a fragment attribute binding matches "fragment.position", the "x" and "y" components of the fragment attribute variable are filled with the (x,y) window coordinates of the fragment center, relative to the lower left corner of the window. The "z" component is filled with the fragment's z window coordinate. This z window coordinate undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. The "w" component is filled with the reciprocal of the fragment's clip w coordinate.

On some implementations, the components of fragment.position may be generated by interpolating per-vertex position values. This may produce x and y window coordinates that don't exactly match those of the fragment center and z window coordinates that do not exactly match those generated by fixed-function rasterization. Therefore, there is no guaranteed invariance between the final z window coordinates of fragments processed by fragment programs that write depth values and fragments processed by any other means, even if the fragment programs in question simply copy the z value from the fragment.position binding.

### 3.11.3.2 Fragment Program Parameters

Fragment program parameter variables are a set of four-component floating-point vectors used as constants during fragment program execution. Fragment program parameters retain their values across fragment program invocations, although their values can change between invocations due to GL state changes.

Single program parameter variables and arrays of program parameter variables can be declared explicitly using the <PARAM_statement> grammar rule. Single program parameter variables can also be declared implicitly using the <paramSingleItemUse> grammar rule in an executable instruction.

Each single program parameter variable is bound to a constant vector or to a GL state vector according to the <paramSingleInit> grammar rule. Individual items of a program parameter array are bound to constant vectors or GL state vectors according to the <programMultipleInit> grammar rule. The set of GL state that can be bound to program parameter variables are given in Tables X.2.1 through X.2.4.

**Constant Bindings**

A program parameter variable can be bound to a scalar or vector constant using the <paramConstDecl> grammar rule (explicit declarations) or the <paramConstUse> grammar rule (implicit declarations).

If a program parameter binding matches the <paramConstScalarDecl> or <paramConstScalarUse> grammar rules, the corresponding program parameter variable is bound to the vector (X,X,X,X), where X is the value of the specified constant. Note that the <paramConstScalarUse> grammar rule, used only in implicit declarations, allows only non-negative constants. This disambiguates cases like "-2", which could conceivably be taken to mean either the vector "(2,2,2,2)" with all components negated or "(-2,-2,-2,-2)" without negation. Only the former interpretation is allowed by the grammar.

If a program parameter binding matches <paramConstVector>, the corresponding program parameter variable is bound to the vector (X,Y,Z,W), where X, Y, Z, and W are the values corresponding to the first, second, third, and fourth match of <signedFloatConstant>. If fewer than four constants are specified, Y, Z, and W assume the values 0.0, 0.0, and 1.0, if their respective constants are not specified.

Program parameter variables initialized to constant values can never be modified.

**Program Environment/Local Parameter Bindings**

| Binding | Components | Underlying State |
|---|---|---|
| program.env[a] | (x,y,z,w) | program environment parameter a |
| program.local[a] | (x,y,z,w) | program local parameter a |
| program.env[a..b] | (x,y,z,w) | program environment parameters a through b |
| program.local[a..b] | (x,y,z,w) | program local parameters a through b |

Table X.2.1: Program Environment/Local Parameter Bindings. <a> and <b> indicate parameter numbers, where <a> must be less than or equal to <b>.

If a program parameter binding matches "program.env[a]" or "program.local[a]", the four components of the program parameter variable are filled with the four components of program environment parameter <a> or program local parameter <a>, respectively.

Additionally, for program parameter array bindings, "program.env[a..b]" and "program.local[a..b]" are equivalent to specifying program environment parameters <a> through <b> in order or program local parameters <a> through <b> in order, respectively. In either case, a program will fail to load if <a> is greater than <b>.

**Material Property Bindings**

| Binding | Components | Underlying State |
|---|---|---|
| state.material.ambient | (r,g,b,a) | front ambient material color |
| state.material.diffuse | (r,g,b,a) | front diffuse material color |
| state.material.specular | (r,g,b,a) | front specular material color |
| state.material.emission | (r,g,b,a) | front emissive material color |
| state.material.shininess | (s,0,0,1) | front material shininess |
| state.material.front.ambient | (r,g,b,a) | front ambient material color |
| state.material.front.diffuse | (r,g,b,a) | front diffuse material color |
| state.material.front.specular | (r,g,b,a) | front specular material color |
| state.material.front.emission | (r,g,b,a) | front emissive material color |
| state.material.front.shininess | (s,0,0,1) | front material shininess |
| state.material.back.ambient | (r,g,b,a) | back ambient material color |
| state.material.back.diffuse | (r,g,b,a) | back diffuse material color |
| state.material.back.specular | (r,g,b,a) | back specular material color |
| state.material.back.emission | (r,g,b,a) | back emissive material color |
| state.material.back.shininess | (s,0,0,1) | back material shininess |

Table X.2.2: Material Property Bindings. If a material face is not specified in the binding, the front property is used.

If a program parameter binding matches any of the material properties listed in Table X.2.2, the program parameter variable is filled according to the table. For ambient, diffuse, specular, or emissive colors, the "x", "y", "z", and "w" components are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding material color. For material shininess, the "x" component is filled with the material's specular exponent, and the "y", "z", and "w" components are filled with 0, 0, and 1, respectively. Bindings containing ".back" refer to the back material; all other bindings refer to the front material.

Material properties can be changed inside a Begin/End pair, either directly by calling Material, or indirectly through color material. However, such property changes are not guaranteed to update program parameter bindings until the following End command. Program parameter variables bound to material properties changed inside a Begin/End pair are undefined until the following End command.

**Light Property Bindings**

| Binding | Components | Underlying State |
|---|---|---|
| state.light[n].ambient | (r,g,b,a) | light n ambient color |
| state.light[n].diffuse | (r,g,b,a) | light n diffuse color |
| state.light[n].specular | (r,g,b,a) | light n specular color |
| state.light[n].position | (x,y,z,w) | light n position |
| state.light[n].attenuation | (a,b,c,e) | light n attenuation constants and spot light exponent |
| state.light[n].spot.direction | (x,y,z,c) | light n spot direction and cutoff angle cosine |
| state.light[n].half | (x,y,z,1) | light n infinite half-angle |
| state.lightmodel.ambient | (r,g,b,a) | light model ambient color |
| state.lightmodel.scenecolor | (r,g,b,a) | light model front scene color |
| state.lightmodel . front.scenecolor | (r,g,b,a) | light model front scene color |
| state.lightmodel . back.scenecolor | (r,g,b,a) | light model back scene color |
| state.lightprod[n].ambient | (r,g,b,a) | light n / front material ambient color product |
| state.lightprod[n].diffuse | (r,g,b,a) | light n / front material diffuse color product |
| state.lightprod[n].specular | (r,g,b,a) | light n / front material specular color product |
| state.lightprod[n]. front.ambient | (r,g,b,a) | light n / front material ambient color product |
| state.lightprod[n]. front.diffuse | (r,g,b,a) | light n / front material diffuse color product |
| state.lightprod[n]. front.specular | (r,g,b,a) | light n / front material specular color product |
| state.lightprod[n]. back.ambient | (r,g,b,a) | light n / back material ambient color product |
| state.lightprod[n]. back.diffuse | (r,g,b,a) | light n / back material diffuse color product |
| state.lightprod[n]. back.specular | (r,g,b,a) | light n / back material specular color product |

Table X.2.3: Light Property Bindings. <n> indicates a light number.

If a program parameter binding matches "state.light[n].ambient", "state.light[n].diffuse", or "state.light[n].specular", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding light color.

If a program parameter binding matches "state.light[n].position", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "x", "y", "z", and "w" components, respectively, of the light position.

If a program parameter binding matches "state.light[n].attenuation", the "x", "y", and "z" components of the program parameter variable are filled with the constant, linear, and quadratic attenuation parameters of the specified light, respectively (section 2.13.1).

The "w" component of the program parameter variable is filled with the spot light exponent of the specified light.

If a program parameter binding matches "state.light[n].spot.direction", the "x", "y", and "z" components of the program parameter variable are filled with the "x", "y", and "z" components of the spot light direction of the specified light, respectively (section 2.13.1). The "w" component of the program parameter variable is filled with the cosine of the spot light cutoff angle of the specified light.

If a program parameter binding matches "state.light[n].half", the "x", "y", and "z" components of the program parameter variable are filled with the x, y, and z components, respectively, of the normalized infinite half-angle vector

```
  h_inf = || P + (0, 0, 1) ||.
```

The "w" component is filled with 1. In the computation of h_inf, P consists of the x, y, and z coordinates of the normalized vector from the eye position P_e to the eye-space light position P_pli (section 2.13.1). h_inf is defined to correspond to the normalized half-angle vector when using an infinite light (w coordinate of the position is zero) and an infinite viewer (v_bs is FALSE). For local lights or a local viewer, h_inf is well-defined but does not match the normalized half-angle vector, which will vary depending on the vertex position.

If a program parameter binding matches "state.lightmodel.ambient", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components of the light model ambient color, respectively.

If a program parameter binding matches "state.lightmodel.scenecolor" or "state.lightmodel.front.scenecolor", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components respectively of the "front scene color"

```
  c_scene = a_cs * a_cm + e_cm,
```

where a_cs is the light model ambient color, a_cm is the front ambient material color, and e_cm is the front emissive material color. The "w" component of the program parameter variable is filled with the alpha component of the front diffuse material color. If a program parameter binding matches "state.lightmodel.back.scenecolor", a similar back scene color, computed using back-facing material properties, is used. The front and back scene colors match the values that would be assigned to vertices using conventional lighting if all lights were disabled.

If a program parameter binding matches anything beginning with "state.lightprod[n]", the "x", "y", and "z" components of the program parameter variable are filled with the "r", "g", and "b" components, respectively, of the corresponding light product. The three light product components are the products of the corresponding color components of the specified material property and the light color of the specified light (see Table X.2.3). The "w" component of the program parameter variable is filled with the alpha component of the specified material property.

Light products depend on material properties, which can be changed inside a Begin/End pair. Such property changes are not guaranteed to take effect until the following End command. Program parameter variables bound to light products whose corresponding material property changes inside a Begin/End pair are undefined until the following End command.

**Texture Environment Property Bindings**

| Binding | Components | Underlying State |
|---|---|---|
| state.texenv[n].color | (r,g,b,a) | texture environment n color |

Table X.2.4: Texture Environment Property Bindings. "[n]" is optional -- texture unit <n> is used if specified; texture unit 0 is used otherwise.

If a program parameter binding matches "state.texenv[n].color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the corresponding texture environment color. Note that only "legacy" texture units, as queried by MAX_TEXTURE_UNITS, include texture environment state. Texture image units and texture coordinate sets do not have associated texture environment state.

**Fog Property Bindings**

| Binding | Components | Underlying State |
|---|---|---|
| state.fog.color | (r,g,b,a) | RGB fog color (section 3.11) |
| state.fog.params | (d,s,e,r) | fog density, linear start and end, and 1/(end-start) (section 3.11) |

Table X.2.5: Fog Property Bindings

If a program parameter binding matches "state.fog.color", the "x", "y", "z", and "w" components of the program parameter variable are filled with the "r", "g", "b", and "a" components, respectively, of the fog color (section 3.11).

If a program parameter binding matches "state.fog.params", the "x", "y", and "z" components of the program parameter variable are filled with the fog density, linear fog start, and linear fog end parameters (section 3.11), respectively. The "w" component is filled with 1/(end-start), where end and start are the linear fog end and start parameters, respectively.

**Depth Property Bindings**

| Binding | Components | Underlying State |
|---|---|---|
| state.depth.range | (n,f,d,1) | Depth range near, far, and (far-near) (section 2.10.1) |

Table X.2.6: Depth Property Bindings

If a program parameter binding matches "state.depth.range", the "x" and "y" components of the program parameter variable are filled with the mappings of near and far clipping planes to window coordinates, respectively. The "z" component is filled with the difference of the mappings of near and far clipping planes, far minus near. The "w" component is filled with 1.

**Matrix Property Bindings**

| Binding | Underlying State |
|---|---|
| * state.matrix.modelview[n] | modelview matrix n |
| state.matrix.projection | projection matrix |
| state.matrix.mvp | modelview-projection matrix |
| * state.matrix.texture[n] | texture matrix n |
| state.matrix.palette[n] | modelview palette matrix n |
| state.matrix.program[n] | program matrix n |

Table X.2.7: Base Matrix Property Bindings. The "[n]" syntax indicates a specific matrix number. For modelview and texture matrices, a matrix number is optional, and matrix zero will be used if the matrix number is omitted. These base bindings may further be modified by a inverse/transpose selector and a row selector.

If the beginning of a program parameter binding matches any of the matrix binding names listed in Table X.2.7, the binding corresponds to a 4x4 matrix. If the parameter binding is followed by ".inverse", ".transpose", or ".invtrans" (<stateMatModifier> grammar rule), the inverse, transpose, or transpose of the inverse, respectively, of the matrix specified in Table X.2.7 is selected. Otherwise, the matrix specified in Table X.2.7 is selected. If the specified matrix is poorly-conditioned (singular or nearly so), its inverse matrix is undefined. The binding name "state.matrix.mvp" refers to the product of modelview matrix zero and the projection matrix, defined as

```
MVP = P * M0,
```

where P is the projection matrix and M0 is modelview matrix zero.

If the selected matrix is followed by ".row[<a>]" (matching the <stateMatrixRow> grammar rule), the "x", "y", "z", and "w" components of the program parameter variable are filled with the four entries of row <a> of the selected matrix. In the example,

```
PARAM m0 = state.matrix.modelview[1].row[0];
PARAM m1 = state.matrix.projection.transpose.row[3];
```

the variable "m0" is set to the first row (row 0) of modelview matrix 1 and "m1" is set to the last row (row 3) of the transpose of the projection matrix.

For program parameter array bindings, multiple rows of the selected matrix can be bound via the <stateMatrixRows> grammar rule. If the selected matrix binding is followed by ".row[<a>..<b>]", the result is equivalent to specifying matrix rows <a> through <b>, in order. A program will fail to load if <a> is greater than <b>. If no row selection is specified (<optMatrixRows> matches ""), matrix rows 0 through 3 are bound in order. In the example,

```
  PARAM m2[] = { state.matrix.program[0].row[1..2] };
  PARAM m3[] = { state.matrix.program[0].transpose };
```

the array "m2" has two entries, containing rows 1 and 2 of program matrix zero, and "m3" has four entries, containing all four rows of the transpose of program matrix zero.

**Program Parameter Arrays**

A program parameter array variable can be declared explicitly by matching the <PARAM_multipleStmt> grammar rule. Programs can optionally specify the number of individual program parameters in the array, using the <optArraySize> grammar rule. Program parameter arrays may not be declared implicity.

Individual parameter variables in a program parameter array are bound to GL state vectors or constant vectors as specified by the grammar rule <paramMultInitList>. Each individual parameter in the array is bound in turn as described above.

The total number of entries in the array is equal to the number of parameters bound in the initializer list. A fragment program that specifies an array size (<optArraySize> matches <integer>) that does not match the number of parameter bindings in the initialization list will fail to load.

Program parameter array variables may only be accessed using absolute addressing by matching the <progParamArrayAbs> grammar rule. Array accesses are checked against the limits of the array. If any fragment program instruction accesses a program parameter array with an out-of-range index (greater than or equal to the size of the array), the fragment program will fail to load.

Individual state vectors can have no more than one unique binding in any given program. The GL will automatically combine multiple bindings of the same state vector into a single unique binding.

**3.11.3.3 Fragment Program Temporaries**

Fragment program temporary variables are a set of four-component floating-point vectors used to hold temporary results during fragment program execution. Temporaries do not persist between program invocations, and are undefined at the beginning of each fragment program invocation.

Fragment program temporary variables can be declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Fragment program temporary variables can not be declared implicitly.

**3.11.3.4 Fragment Program Results**

Fragment program result variables are a set of four component floating-point vectors used to hold the final results of a fragment program. Fragment program result variables are write-only during fragment program execution.

Fragment program result variables can be declared explicitly using the <OUTPUT_statement> grammar rule, or implicitly using the <resultBinding> grammar rule in an executable instruction. Each fragment program result variable is bound to a fragment attribute used in subsequent back-end processing. The set of fragment program result variable bindings is given in Table X.3.

| Binding | Components | Description |
|---|---|---|
| result.color | (r,g,b,a) | color |
| result.depth | (*,*,d,*) | depth coordinate |

Table X.3: Fragment Result Variable Bindings. Components labeled "*" are unused.

If a result variable binding matches "result.color", updates to the "x", "y", "z", and "w" components of the result variable modify the "r", "g", "b", and "a" components, respectively, of the fragment's output color. If "result.color" is not both bound by the fragment program and written by some instruction of the program, the output color of the fragment program is undefined.

If a result variable binding matches "result.depth", updates to the "z" component of the result variable modify the fragment's output depth value. If "result.depth" is not both bound by the fragment program and written by some instruction of the program, the interpolated depth value produced by rasterization is used as if fragment program mode is not enabled. Writes to any component of depth other than the "z" component have no effect.

**3.11.3.5 Fragment Program Aliases**

Fragment programs can create aliases by matching the <ALIAS_statement> grammar rule. Aliases allow programs to use multiple variable names to refer to a single underlying variable. For example, the statement

```
ALIAS var1 = var0
```

establishes a variable name named "var1". Subsequent references to "var1" in the program text are treated as references to "var0". The left hand side of an ALIAS statement must be a new variable name, and the right hand side must be an established variable name.

Aliases are not considered variable declarations, so do not count against the limits on the number of variable declarations allowed in the program text.

**3.11.3.6 Fragment Program Resource Limits**

The fragment program execution environment provides implementation-dependent resource limits on the number of ALU instructions, texture instructions, total instructions (ALU or texture), temporary variable declarations, program parameter bindings, or texture indirections. A program that exceeds any of these resource limits will fail to load. The resource limits for fragment programs can be queried by calling GetProgramiv (section 6.1.12) with a target of FRAGMENT_PROGRAM_ARB.

The limit on fragment program ALU instructions can be queried with a <pname> of MAX_PROGRAM_ALU_INSTRUCTIONS_ARB, and must be at least 48. Each ALU instruction in the program (matches of the <ALUInstruction> grammar rule) counts against this limit.

The limit on fragment program texture instructions can be queried with a <pname> of MAX_PROGRAM_TEX_INSTRUCTIONS_ARB, and must be at least 24. Each texture instruction in the program (matches of the <TexInstruction> grammar rule) counts against this limit.

The limit on fragment program total instructions can be queried with a <pname> of MAX_PROGRAM_INSTRUCTIONS_ARB, and must be at least 72. Each instruction in the program (matching the <instruction> grammar rule) counts against this limit. Note that the limit on total instructions is not necessarily equal to the sum of the limits on ALU instructions and texture instructions.

The limit on fragment program texture indirections can be queried with a <pname> of MAX_PROGRAM_TEX_INDIRECTIONS_ARB, and must be at least 4. Texture indirections are described in 3.11.6. If an implementation has no limit on texture indirections, the limit will be equal to the limit on texture instructions.

The limit on fragment program temporary variable declarations can be queried with a <pname> of MAX_PROGRAM_TEMPORARIES_ARB, and must be at least 16. Each temporary declared in the program, using the <TEMP_statement> grammar rule, counts against this limit. Aliases of declared temporaries do not.

The limit on fragment program attribute bindings can be queried with a <pname> of MAX_PROGRAM_ATTRIBS_ARB and must be at least 10. Each distinct vertex attribute bound explicitly or implicitly in the program counts against this limit; vertex attributes bound multiple times count only once.

The limit on fragment program parameter bindings can be queried with a <pname> of MAX_PROGRAM_PARAMETERS_ARB, and must be at least 24. Each distinct GL state vector bound explicitly or implicitly in the program counts against this limit; GL state vectors bound multiple times count only once. Every other constant vector bound in the program is counted if and only if an identical constant vector has not already been counted. Two constant vectors are considered identical if the four component values are numerically equivalent. Recall that scalar constants bound in a program are treated as vector constants with the scalar value replicated.

In addition to the limits described above, the GL provides a similar set of implementation-dependent native resource limits. These limits, specified in Section 6.1.12, provide guidance as to whether the program is small enough to use a "native" mode where fragment programs may be executed with higher performance. The native resource limits and usage counts are implementation-dependent and may not exactly correspond to limits and counts described above. A program's native resource consumption may be reduced by program optimizations performed by the GL. Native resource consumption may be increased due to emulation of instructions or any other program features not natively supported by an implementation. Notably, an additional texture indirection may be consumed due to an implementation's lack of native support for texture instructions with source coordinate swizzles or parameter source coordinates, which may require emulation by prepending ALU instructions. An implementation may also fail to natively support all combinations of attributes described in Table X.1, even if the total number of bound attributes is fewer than the native attribute limit. In this case the program is still considered to exceed the native resource limits, as queried by PROGRAM_UNDER_NATIVE_LIMITS_ARB (section 6.1.12).

To assist in resource counting, the GL additionally provides GetProgram queries to determine the resource usage and native resource usage of the currently bound program, and to determine whether the bound program exceeds any native resource limit.

Programs that exceed any native resource limit may or may not load depending on the implementation.

## 3.11.4 Fragment Program Execution Environment

If fragment program mode is enabled, the currently bound fragment program is executed when any fragment is produced by rasterization.

If fragment program mode is enabled and the currently bound program object does not contain a valid fragment program, the error INVALID_OPERATION will be generated by Begin, RasterPos, and any command that implicitly calls Begin (e.g., DrawArrays).

Fragment programs execute a sequence of instructions without branching. Fragment programs begin by executing the first instruction in the program, and execute instructions in the order specified in the program until the last instruction is completed.

There are 33 fragment program instructions. The instructions and their respective input and output parameters are summarized in Table X.5.

| Instruction | Inputs | Output | Description |
|---|---|---|---|
| ABS | v | v | absolute value |
| ADD | v,v | v | add |
| CMP | v,v,v | v | compare |
| COS | s | ssss | cosine with reduction to [-PI,PI] |
| DP3 | v,v | ssss | 3-component dot product |
| DP4 | v,v | ssss | 4-component dot product |
| DPH | v,v | ssss | homogeneous dot product |
| DST | v,v | v | distance vector |
| EX2 | s | ssss | exponential base 2 |
| FLR | v | v | floor |
| FRC | v | v | fraction |
| KIL | v | v | kill fragment |
| LG2 | s | ssss | logarithm base 2 |
| LIT | v | v | compute light coefficients |
| LRP | v,v,v | v | linear interpolation |
| MAD | v,v,v | v | multiply and add |
| MAX | v,v | v | maximum |
| MIN | v,v | v | minimum |
| MOV | v | v | move |
| MUL | v,v | v | multiply |
| POW | s,s | ssss | exponentiate |
| RCP | s | ssss | reciprocal |
| RSQ | s | ssss | reciprocal square root |
| SCS | s | ss-- | sine/cosine without reduction |
| SGE | v,v | v | set on greater than or equal |
| SIN | s | ssss | sine with reduction to [-PI,PI] |
| SLT | v,v | v | set on less than |
| SUB | v,v | v | subtract |
| SWZ | v | v | extended swizzle |
| TEX | v,u,t | v | texture sample |
| TXB | v,u,t | v | texture sample with bias |
| TXP | v,u,t | v | texture sample with projection |
| XPD | v,v | v | cross product |

Table X.5: Summary of fragment program instructions. "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, "u" indicates a texture image unit identifier, and "t" indicates a texture target.

### 3.11.4.1 Fragment Program Operands

Most fragment program instructions operate on floating-point vectors or scalars, as indicated by the grammar rules <vectorSrcReg> and <scalarSrcReg>, respectively.

Vector and scalar operands can be obtained from fragment attribute, program parameter, or temporary registers, as indicated by the <srcReg> rule. For scalar operands, a single vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w", or "r", "g", "b", and "a" select the first, second, third, and fourth components, respectively, of the vector.

Vector operands can be swizzled according to the <optionalSuffix> rule. In its most general form, the <optionalSuffix> rule matches the pattern ".????" where each question mark is replaced with one of "x", "y", "z", "w", "r", "g", "b", or "a". For such patterns, the first, second, third, and fourth components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" or ".gbbr" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}.

If the <optionalSuffix> rule matches "", it is treated as though it were ".xyzw". If the <optionalSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively. Likewise, if the <optionalSuffix> rule matches ".r", ".g", ".b", or ".a", these are treated the same as ".rrrr", ".gggg", ".bbbb", and ".aaaa" respectively.

Floating-point scalar or vector operands can optionally be negated according to the <optionalSign> rule in <scalarSrcReg> and <vectorSrcReg>. If the <optionalSign> matches "-", each operand or operand component is negated.

The following pseudo-code spells out the operand generation process. In the example, "float" is a floating-point scalar type, while "floatVec" is a four-component vector. "source" refers to the register used for the operand, matching the <srcReg> rule. "negate" is TRUE if the <optionalSign> rule in <scalarSrcReg> or <vectorSrcReg> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation; the ".c" modifier refers to the single component selected for a scalar load.

```
floatVec VectorLoad(floatVec source)
{
    floatVec operand;

    operand.x = source.c***;
    operand.y = source.*c**;
    operand.z = source.**c*;
    operand.w = source.***c;
    if (negate) {
       operand.x = -operand.x;
       operand.y = -operand.y;
       operand.z = -operand.z;
       operand.w = -operand.w;
    }

    return operand;
}
```

```
  float ScalarLoad(floatVec source)
  {
      float operand;

      operand = source.c;
      if (negate) {
        operand = -operand;
      }

      return operand;
  }
```

**3.11.4.2  Fragment Program Parameter Arrays**

A fragment program can load a single element of a program parameter array using only absolute addressing.  Program parameter arrays are accessed when the <progParamArrayAbs> rule is matched.  The offset of the selected entry in the array is given by the number matching <progParamRegNum>.  If the offset exceeds the size of the array, the results of the access are undefined, but may not lead to program or GL termination.

**3.11.4.3  Fragment Program Destination Register Update**

Fragment program instructions write a 4-component result vector to a single temporary or fragment result register.  Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction. Optional clamping of each component of the destination register to the range [0,1] is controlled by an opcode modifier.

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", or "r", "g", "b", and "a" match the first, second, third, and fourth 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", or "rgba" order.  Component names from one set (xyzw or rgba) cannot be mixed with component names from another set.  For example, ".rgw" is not a valid writemask.

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

If the instruction opcode has the "_SAT" suffix, requesting saturated result vectors, each component of the result vector enabled in the writemask is clamped to the range [0,1] before being updated in the destination register.

The following pseudocode illustrates the process of writing a result vector to the destination register.  In the pseudocode, "instrmask" refers to the component write mask given by the <optionalMask> rule. "clamp" is TRUE if the instruction specifies that the result should be clamped. "result" and "destination" refer to the result vector and the register selected by <dstReg>, respectively.

```
  void UpdateDestination(floatVec destination, floatVec result)
  {
      floatVec merged;

      // Clamp the result vector components to [0,1], if requested.
      if (instrClamp) {
          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-time write mask.
      merged = destination;
      if (instrMask.x) {
          merged.x = result.x;
      }
      if (instrMask.y) {
          merged.y = result.y;
      }
      if (instrMask.z) {
          merged.z = result.z;
      }
      if (instrMask.w) {
          merged.w = result.w;
      }

      // Write out the new destination register.
      destination = merged;
  }
```

#### 3.11.4.4 Fragment Program Result Processing

As a fragment program executes, it will write to either one or two result registers that are mapped to the fragment's color and depth.

The fragment's color components are first clamped to the range [0,1] then converted to fixed point as in section 2.13.9. If the fragment program does not write result.color, the color will be undefined in subsequent stages.

If the fragment program contains an instruction to write to result.depth, the fragment's depth is replaced by the value of the "z" component of result.depth. This z value is first clamped to the range [0,1] then converted to fixed-point as if it were a window z value (section 2.10.1). If the fragment program does not write result.depth, the fragment's original depth is unmodified.

**3.11.4.5  Fragment Program Options**

The <optionSequence> grammar rule provides a mechanism for programs to indicate that one or more extended language features are used by the program.  All program options used by the program must be declared at the beginning of the program string.  Each program option specified in a program string will modify the syntactic or semantic rules used to interpet the program and the execution environment used to execute the program.  Program options not present in the program string are ignored, even if they are supported by the GL.

The <identifier> token in the <option> rule must match the name of a program option supported by the implementation.  To avoid option name conflicts, option identifiers are required to begin with a vendor prefix.  A program will fail to load if it specifies a program option not supported by the GL.

Fragment program options should confine their semantic changes to the domain of fragment programs.  Support for a fragment program option should not change the specification and behavior of fragment programs not requesting use of that option.

**3.11.4.5.1  Fog Application Fragment Program Options**

If a fragment program specifies one of the options "ARB_fog_exp", "ARB_fog_exp2", or "ARB_fog_linear", the program will apply fog to the program's final clamped color using a fog mode of EXP, EXP2, or LINEAR, respectively, as described in section 3.10.

When a fog option is specified in a fragment program, semantic restrictions are added to indicate that a fragment program will fail to load if the number of temporaries it contains exceeds the implementation-dependent limit minus 1, if the number of attributes it contains exceeds the implementation-dependent limit minus 1, or if the number of parameters it contains exceeds the implementation-dependent limit minus 2.

Additionally, when the ARB_fog_exp option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 3.  When the ARB_fog_exp2 option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 4.  When the ARB_fog_linear option is specified in a fragment program, a semantic restriction is added to indicate that a fragment program will fail to load if the number of instructions or ALU instructions it contains exceeds the implementation-dependent limit minus 2.

Only one fog application option may be specified by any given fragment program.  A fragment program that specifies more than one of the program options "ARB_fog_exp", "ARB_fog_exp2", and "ARB_fog_linear", will fail to load.

**3.11.4.5.2 Precision Hint Options**

Fragment program computations are carried out at an implementation-dependent precision. However, some implementations may be able to perform fragment program computations at more than one precision, and may be able to trade off computation precision for performance.

If a fragment program specifies the "ARB_precision_hint_fastest" program option, implementations should select precision to minimize program execution time, with possibly reduced precision. If a fragment program specifies the "ARB_precision_hint_nicest" program option, implementations should maximize the precision, with possibly increased execution time.

Only one precision control option may be specified by any given fragment program. A fragment program that specifies both the "ARB_precision_hint_fastest" and "ARB_precision_hint_nicest" program options will fail to load.

**3.11.5 Fragment Program ALU Instruction Set**

The following sections describe the set of supported fragment program instructions. Each section contains pseudocode describing the instruction. Instructions will have up to three operands, referred to as "op0", "op1", and "op2". The operands are loaded using the mechanisms specified in section 3.11.4.1. The variables "tmp", "tmp0", "tmp1", and "tmp2" describe scalars or vectors used to hold intermediate results in the instruction. Instructions will generate a result vector called "result". The result vector is then written to the destination register specified in the instruction as described in section 3.11.4.3.

**3.11.5.1 ABS: Absolute Value**

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
  tmp = VectorLoad(op0);
  result.x = fabs(tmp.x);
  result.y = fabs(tmp.y);
  result.z = fabs(tmp.z);
  result.w = fabs(tmp.w);
```

**3.11.5.2  ADD:  Add**

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x + tmp1.x;
result.y = tmp0.y + tmp1.y;
result.z = tmp0.z + tmp1.z;
result.w = tmp0.w + tmp1.w;
```

The following rules apply to addition:

1. <x> + <y> == <y> + <x>, for all <x> and <y>.
2. <x> + 0.0 == <x>, for all <x>.

**3.11.5.3  CMP: Compare**

The CMP instructions performs a component-wise comparison of the first operand against zero, and copies the values of the second or third operands based on the results of the compare.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = (tmp0.x < 0.0) ? tmp1.x : tmp2.x;
result.y = (tmp0.y < 0.0) ? tmp1.y : tmp2.y;
result.z = (tmp0.z < 0.0) ? tmp1.z : tmp2.z;
result.w = (tmp0.w < 0.0) ? tmp1.w : tmp2.w;
```

**3.11.5.4  COS:  Cosine**

The COS instruction approximates the trigonometric cosine of the angle specified by the scalar operand and replicates it to all four components of the result vector.  The angle is specified in radians and does not have to be in the range [-PI,PI].

```
tmp = ScalarLoad(op0);
result.x = ApproxCosine(tmp);
result.y = ApproxCosine(tmp);
result.z = ApproxCosine(tmp);
result.w = ApproxCosine(tmp);
```

**3.11.5.5  DP3:  Three-Component Dot Product**

The DP3 instruction computes a three-component dot product of the two operands (using the first three components) and replicates the dot product to all four components of the result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + (tmp0.z * tmp1.z);
result.x = dot;
result.y = dot;
result.z = dot;
result.w = dot;
```

**3.11.5.6  DP4:  Four-Component Dot Product**

The DP4 instruction computes a four-component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

**3.11.5.7  DPH:  Homogeneous Dot Product**

The DPH instruction computes a three-component dot product of the two operands (using the x, y, and z components), adds the w component of the second operand, and replicates the sum to all four components of the result vector.  This is equivalent to a four-component dot product where the w component of the first operand is forced to 1.0.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
        (tmp0.z * tmp1.z) + tmp1.w;
  result.x = dot;
  result.y = dot;
  result.z = dot;
  result.w = dot;
```

**3.11.5.8  DST:  Distance Vector**

The DST instruction computes a distance vector from two specially-formatted operands.  The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length.  If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-fragment light attenuation calculations:  a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

**3.11.5.9  EX2:  Exponential Base 2**

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates the approximation to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

**3.11.5.10  FLR:  Floor**

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector.  The floor of a value is defined as the largest integer less than or equal to the value.  The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

**3.11.5.11  FRC:  Fraction**

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector.  The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.0, 1.0).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3.  0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = fraction(tmp.x);
  result.y = fraction(tmp.y);
  result.z = fraction(tmp.z);
  result.w = fraction(tmp.w);
```

**3.11.5.12 LG2: Logarithm Base 2**

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

If the scalar operand is zero or negative, the result is undefined.

**3.11.5.13 LIT: Light Coefficients**

The LIT instruction accelerates per-fragment lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the single operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (tmp.x < 0) tmp.x = 0;
  if (tmp.y < 0) tmp.y = 0;
  if (tmp.w < -(128.0-epsilon)) tmp.w = -(128.0-epsilon);
  else if (tmp.w > 128-epsilon) tmp.w = 128-epsilon;
  result.x = 1.0;
  result.y = tmp.x;
  result.z = (tmp.x > 0) ? RoughApproxPower(tmp.y, tmp.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation function may be defined in terms of the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions, where

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

In particular, the approximation may not be any more accurate than the underlying EX2 and LG2 operations.

Also, since 0^0 is defined to be 1, RoughApproxPower(0.0, 0.0) will produce 1.0.

**3.11.5.14 LRP: Linear Interpolation**

The LRP instruction performs a component-wise linear interpolation between the second and third operands using the first operand as the blend factor.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
  result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
  result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
  result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

**3.11.5.15 MAD: Multiply and Add**

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

The multiplication and addition operations in this instruction are subject to the same rules as described for the MUL and ADD instructions.

**3.11.5.16 MAX: Maximum**

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp0.x : tmp1.x;
  result.y = (tmp0.y > tmp1.y) ? tmp0.y : tmp1.y;
  result.z = (tmp0.z > tmp1.z) ? tmp0.z : tmp1.z;
  result.w = (tmp0.w > tmp1.w) ? tmp0.w : tmp1.w;
```

**3.11.5.17 MIN: Minimum**

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? tmp1.x : tmp0.x;
  result.y = (tmp0.y > tmp1.y) ? tmp1.y : tmp0.y;
  result.z = (tmp0.z > tmp1.z) ? tmp1.z : tmp0.z;
  result.w = (tmp0.w > tmp1.w) ? tmp1.w : tmp0.w;
```

**3.11.5.18 MOV: Move**

The MOV instruction copies the value of the operand to yield a result vector.

```
  result = VectorLoad(op0);
```

**3.11.5.19 MUL: Multiply**

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

The following rules apply to multiplication:

```
  1. <x> * <y> == <y> * <x>, for all <x> and <y>.
  2. +/-0.0 * <x> = +/-0.0, at least for all <x> that correspond to
     representable numbers (IEEE "not a number" and "infinity"
     encodings may be exceptions).
  3. +1.0 * <x> = <x>, for all <x>.
```

Multiplication by zero and one should be invariant, as it may be used to evaluate conditional expressions without branching.

**3.11.5.20 POW: Exponentiate**

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function may be implemented using the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

Note that a logarithm may be involved even for cases where the exponent is an integer. This means that it may not be possible to exponentiate correctly with a negative base. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

**3.11.5.21 RCP: Reciprocal**

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The following rule applies to reciprocation:

```
  1. ApproxReciprocal(+1.0) = +1.0.
```

**3.11.5.22 RSQ: Reciprocal Square Root**

The RSQ instruction approximates the reciprocal of the square root of the absolute value of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = fabs(ScalarLoad(op0));
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

**3.11.5.23 SCS: Sine/Cosine**

The SCS instruction approximates the trigonometric sine and cosine of the angle specified by the scalar operand and places the cosine in the x component and the sine in the y component of the result vector.  The z and w components of the result vector are undefined. The angle is specified in radians and must be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxSine(tmp);
```

If the scalar operand is not in the range [-PI,PI], the result vector is undefined.

**3.11.5.24 SGE: Set On Greater or Equal Than**

The SGE instruction performs a component-wise comparison of the two operands.  Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
```

**3.11.5.25 SIN: Sine**

The SIN instruction approximates the trigonometric sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [-PI,PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

**3.11.5.26 SLT: Set On Less Than**

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
```

**3.11.5.27 SUB: Subtract**

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

**3.11.5.28 SWZ: Extended Swizzle**

The SWZ instruction loads the single vector operand, and performs a swizzle operation more powerful than that provided for loading normal vector operands to yield an instruction vector.

After the operand is loaded, the "x", "y", "z", and "w" components of the result vector are selected by the first, second, third, and fourth matches of the <xyzwExtSwizComp> or <rgbaExtSwizComp> pattern in the <extendedSwizzle> rule.

A result component can be selected from any of the four components of the operand or the constants 0.0 and 1.0. The result component can also be optionally negated. The following pseudocode describes the component selection method. "operand" refers to the vector operand. "select" is an enumerant where the values ZERO, ONE, X, Y, Z, and W correspond to the <xyzwExtSwizSel> rule matching "0", "1", "x", "y", "z", and "w", respectively, or the <rgbaExtSwizSel> rule matching "0", 1", "r", "g", "b", and "a", respectively. "negate" is TRUE if and only if the <optionalSign> rule in <xyzwExtSwizComp> or <rgbaExtSwizComp> matches "-".

```
  float ExtSwizComponent(floatVec operand, enum select, boolean negate)
  {
      float result;
      switch (select) {
        case ZERO:  result = 0.0; break;
        case ONE:   result = 1.0; break;
        case X:     result = operand.x; break;
        case Y:     result = operand.y; break;
        case Z:     result = operand.z; break;
        case W:     result = operand.w; break;
      }
      if (negate) {
        result = -result;
      }
      return result;
  }
```

The entire extended swizzle operation is then defined using the following pseudocode:

```
  tmp = VectorLoad(op0);
  result.x = ExtSwizComponent(tmp, xSelect, xNegate);
  result.y = ExtSwizComponent(tmp, ySelect, yNegate);
  result.z = ExtSwizComponent(tmp, zSelect, zNegate);
  result.w = ExtSwizComponent(tmp, wSelect, wNegate);
```

"xSelect", "xNegate", "ySelect", "yNegate", "zSelect", "zNegate", "wSelect", and "wNegate" correspond to the "select" and "negate" values above for the four <xyzwExtSwizComp> or <rgbaExtSwizComp> matches.

Since this instruction allows for component selection and negation for each individual component, the grammar does not allow the use of the normal swizzle and negation operations allowed for vector operands in other instructions.

**3.11.5.29 XPD: Cross Product**

The XPD instruction computes the cross product using the first three components of its two vector operands to generate the x, y, and z components of the result vector. The w component of the result vector is undefined.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.y * tmp1.z - tmp0.z * tmp1.y;
  result.y = tmp0.z * tmp1.x - tmp0.x * tmp1.z;
  result.z = tmp0.x * tmp1.y - tmp0.y * tmp1.x;
```

**3.11.6 Fragment Program Texture Instruction Set**

The first three texture instructions described below specify the mapping of 4-tuple vectors to colors of an image. The sampling of the texture works as described in section 3.8, except that texture environments and texture functions are not applicable, and the texture enables hierarchy is replaced by explicit references to the desired texture target (i.e., 1D, 2D, 3D, cube map, rectangle). These texture instructions specify how the 4-tuple is mapped into the coordinates used for sampling. The following function is used to describe the texture sampling in the descriptions below:

```
  vec4 TextureSample(float s, float t, float r, float lodBias,
                     int texImageUnit, enum texTarget);
```

Note that not all three texture coordinates, s, t, and r, are used by all texture targets. In particular, 1D texture targets only use the s component, and 2D and rectangle (non-power-of-two) texture targets only use the s and t components. The descriptions of the texture instructions below supply all three components, as would be the case with 3D or cube map targets.

If a fragment program samples from a texture target on a texture image unit where the bound texture object is not complete, as defined in section 3.8.9, the result will be the vector (R, G, B, A) = (0, 0, 0, 1).

A fragment program will fail to load if it attempts to sample from multiple texture targets on the same texture image unit. For example, the following program would fail to load:

```
  !!ARBfp1.0
  TEX result.color, fragment.texcoord[0], texture[0], 2D;
  TEX result.depth, fragment.texcoord[1], texture[0], 3D;
  END
```

The fourth texture instruction described below, KIL, does not sample from a texture, but rather prevents further processing of the current fragment if any component of its 4-tuple vector is less than zero.

A dependent texture instruction is one that samples using a texture coordinate residing in a temporary, rather than in an attribute or a parameter. A program may have a chain of dependent texture instructions, where the result of the first texture instruction is used as the coordinate for a second texture instruction, which is in turn used as the coordinate for a third texture instruction, and so on. Each node in this chain is termed an indirection, and can be thought of as a set of texture samples that execute in parallel followed by a sequence of ALU instructions.

Some implementations may have limitations on how long the dependency chain may be, and so indirections are counted as a resource just like instructions or temporaries are counted. All programs have at least one indirection, or one node in this chain, even if the program performs no texture operation. Each instruction encountered is included in this node until a texture instruction is encountered

- whose texture coordinate is a temporary that has been previously written in the current node; or

- whose result vector is a temporary that is also the operand or result vector of a previous ALU instruction in the current node.

A new node is then started, including the texture instruction and all subsequent instructions, and the process repeats for all instructions in the program. Note that for simplicity in counting, result writemasks and operand suffixes are not taken into consideration when counting indirections.

**3.11.6.1 TEX: Map coordinate to color**

The TEX instruction takes the first three components of its source vector, and maps them to s, t, and r. 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 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);
```

**3.11.6.2 TXP: Project coordinate and map to color**

The TXP instruction divides the first three components of its source vector by the fourth component and maps the results to s, t, and r. 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 resulting sample is mapped to RGBA as described in table 3.21 and written to the result vector. If the value of the fourth component of the source vector is less than or equal to zero, the result vector is undefined.

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

**3.11.6.3 TXB: Map coordinate to color while biasing its LOD**

The TXB instruction takes the first three components of its source vector and maps them to s, t, and r. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. Additionally, the fourth component of the source vector is applied to equation 3.14 as fragment_bias below to further bias the level of detail.

```
 lambda'(x,y) = log2[p(x,y)] +
               clamp(texobj_bias + texunit_bias + fragment_bias)
```

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, tmp.w, op1, op2);
```

**3.11.6.4 KIL: Kill fragment**

Rather than mapping a coordinate set to a color, this function prevents a fragment from receiving any future processing. If any component of its source vector is negative, the processing of this fragment will be discontinued and no further outputs to this fragment will occur. Subsequent stages of the GL pipeline will be skipped for this fragment.

```
  tmp = VectorLoad(op0);
  if ((tmp.x < 0) || (tmp.y < 0) ||
      (tmp.z < 0) || (tmp.w < 0))
  {
      exit;
  }
```

**3.11.7 Program Matrices**

In addition to GL's conventional matrices, several additional program matrices are available for use as program parameters. These matrices have names of the form MATRIX<i>_ARB where <i> is between zero and <n>-1 where <n> is the value of the implementation-dependent constant MAX_PROGRAM_MATRICES_ARB. The MATRIX<i>_ARB constants obey MATRIX<i>_ARB = MATRIX0_ARB + <i>. The value of MAX_PROGRAM_MATRICES_ARB must be at least eight. The maximum stack depth for program matrices is defined by the MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB and must be at least 1.

**3.11.8 Required Fragment Program State**

The state required to support program objects of all targets consists of:

  an integer for the program error position, initially -1;

  an array of ubytes for the program error string, initially empty;

and the state that must be maintained to indicate which integers are currently in use as program object names.

The state required to support the fragment program target consists of:

a bit indicating whether or not fragment program mode is enabled, initially disabled;

a set of MAX_PROGRAM_ENV_PARAMETERS_ARB four-component floating-point program environment parameters, initially set to (0,0,0,0);

an unsigned integer naming the currently bound fragment program, initially zero;

The state required for each fragment program object consists of:

an unsigned integer indicating the program object name;

an array of type ubyte containing the program string, initially empty;

an unsigned integer holding the length of the program string, initially zero;

an enum indicating the program string format, initially PROGRAM_FORMAT_ASCII_ARB;

a bit indicating whether or not the program exceeds the native limits;

six unsigned integers holding the number of instruction (ALU, texture, and total), texture indirection, temporary variable, and program parameter binding resources used by the program, initially all zero;

six unsigned integers holding the number of native instruction (ALU, texture, and total), texture indirection, temporary variable, and program parameter binding resources used by the program, initially all zero;

and a set of MAX_PROGRAM_LOCAL_PARAMETERS_ARB four-component floating-point program local parameters, initially set to (0,0,0,0).

Initially, no fragment program objects exist.

## 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)

### Modify Section 5.4, Display Lists (p. 191)

(modify third paragraph, p. 195) ... These are IsList, GenLists, ..., IsProgramARB, GenProgramsARB, and DeleteProgramsARB, as well as IsEnabled and all the Get commands (chapter 6).

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

### Modify Section 6.1.2, Data Conversions (p. 198)

(add before last paragraph, p. 198) The matrix selected by the current matrix mode can be queried by calling GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev with <pname> set to CURRENT_MATRIX_ARB; the matrix will be returned in transposed form with <pname> set to TRANSPOSE_CURRENT_MATRIX_ARB. The depth of the selected matrix stack can be queried with <pname> set to CURRENT_MATRIX_STACK_DEPTH_ARB. Querying CURRENT_MATRIX_ARB and CURRENT_MATRIX_STACK_DEPTH_ARB is the only means for querying the matrix and matrix stack depth of the program matrices described in section 3.11.7.

(add to end of last paragraph, p. 199) Queries of texture state variables corresponding to texture coordinate processing unit (namely, TexGen state and enables, and matrices) will produce an INVALID_OPERATION error if the value of ACTIVE_TEXTURE is greater than or equal to MAX_TEXTURE_COORDS_ARB. All other texture state queries will result in an INVALID_OPERATION error if the value of ACTIVE_TEXTURE is greater than or equal to MAX_TEXTURE_IMAGE_UNITS_ARB.

### Modify Section 6.1.11, Pointer and String Queries (p. 206)

(modify last paragraph, p. 206) ... The possible values for <name> are VENDOR, RENDERER, VERSION, EXTENSIONS, and PROGRAM_ERROR_STRING_ARB.

(add after last paragraph of section, p. 207) Queries of PROGRAM_ERROR_STRING_ARB return a pointer to an implementation-dependent program load error string. If the last call to ProgramStringARB failed to load a program, the returned string describes at least one reason why the program failed to load. If the last call to ProgramStringARB successfully loaded a program, the returned string may be empty (containing only a zero terminator) or may contain one or more implementation-dependent warning messages. The contents of the error string are guaranteed to remain constant only until the next ProgramStringARB command, which may overwrite the error string.

Insert a new Section 6.1.12, Program Queries (p. 207), between existing sections 6.1.11 and 6.1.12.

**6.1.12  Program Queries**

The commands

```
  void GetProgramEnvParameterdvARB(enum target, uint index,
                                   double *params);
  void GetProgramEnvParameterfvARB(enum target, uint index,
                                   float *params);
```

obtain the current value for the program environment parameter numbered <index> for the given program target <target>, and places the information in the array <params>.  The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program environment parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program environment parameters for the program target.

When <target> is FRAGMENT_PROGRAM_ARB, each program parameter returned is an array of four values.

The commands

```
  void GetProgramLocalParameterdvARB(enum target, uint index,
                                     double *params);
  void GetProgramLocalParameterfvARB(enum target, uint index,
                                     float *params);
```

obtain the current value for the program local parameter numbered <index> belonging to the program object currently bound to <target>, and places the information in the array <params>.  The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support program local parameters.  The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported program local parameters for the program target.

When the program target type is FRAGMENT_PROGRAM_ARB, each program local parameter returned is an array of four values.

The command

```
  void GetProgramivARB(enum target, enum pname, int *params);
```

obtains program state for the program target <target>, writing the state into the array given by <params>.  GetProgramivARB can be used to determine the properties of the currently bound program object or implementation limits for <target>.

If <pname> is PROGRAM_LENGTH_ARB, PROGRAM_FORMAT_ARB, or PROGRAM_BINDING_ARB, GetProgramivARB returns one integer holding the program string length (in bytes), program string format, and program name, respectively, for the program object currently bound to <target>.

If <pname> is MAX_PROGRAM_LOCAL_PARAMETERS_ARB or MAX_PROGRAM_ENV_PARAMETERS_ARB, GetProgramivARB returns one integer holding the maximum number of program local parameters or program environment parameters, respectively, supported for the program target <target>.

If <pname> is MAX_PROGRAM_INSTRUCTIONS_ARB, MAX_PROGRAM_ALU_INSTRUCTIONS_ARB, MAX_PROGRAM_TEX_INSTRUCTIONS_ARB, MAX_PROGRAM_TEX_INDIRECTIONS_ARB, MAX_PROGRAM_TEMPORARIES_ARB, MAX_PROGRAM_PARAMETERS_ARB, or MAX_PROGRAM_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the maximum number of total instructions, ALU instructions, texture instructions, texture indirections, temporaries, parameters, and attributes that can be used by a program of type <target>. If <pname> is PROGRAM_INSTRUCTIONS_ARB, PROGRAM_ALU_INSTRUCTIONS_ARB, PROGRAM_TEX_INSTRUCTIONS_ARB, PROGRAM_TEX_INDIRECTIONS_ARB, PROGRAM_TEMPORARIES_ARB, PROGRAM_PARAMETERS_ARB, or PROGRAM_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the number of total instructions, ALU instructions, texture instructions, texture indirections, temporaries, parameters, and attributes used by the current program for <target>.

If <pname> is MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB, MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB, MAX_PROGRAM_NATIVE_TEMPORARIES_ARB, MAX_PROGRAM_NATIVE_PARAMETERS_ARB, or MAX_PROGRAM_NATIVE_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the maximum number of native instruction, ALU instruction, texture instruction, texture indirection, temporary, parameter, and attribute resources available to a program of type <target>. If <pname> is PROGRAM_NATIVE_INSTRUCTIONS_ARB, PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB, PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB, PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB, PROGRAM_NATIVE_TEMPORARIES_ARB, PROGRAM_NATIVE_PARAMETERS_ARB, or PROGRAM_NATIVE_ATTRIBS_ARB, GetProgramivARB returns a single integer giving the number of native instruction, ALU instruction, texture instruction, texture indirection, temporary, parameter, and attribute resources consumed by the program currently bound to <target>. Native resource counts will reflect the results of implementation-dependent scheduling and optimization algorithms applied by the GL, as well as emulation of non-native features. If <pname> is PROGRAM_UNDER_NATIVE_LIMITS_ARB, GetProgramivARB returns 0 if the native resource consumption of the program currently bound to <target> exceeds the number of available resources for any resource type, and 1 otherwise.

The command

```
  void GetProgramStringARB(enum target, enum pname, void *string);
```

obtains the program string for the program object bound to <target> and places the information in the array <string>. <pname> must be PROGRAM_STRING_ARB. <n> ubytes are returned into the array program where <n> is the length of the program in ubytes, as returned by GetProgramivARB when <pname> is PROGRAM_LENGTH_ARB. The program string is always returned using the format given when the program string was specified.

The command

```
  boolean IsProgramARB(uint program);
```

returns TRUE if <program> is the name of a program object. If <program> is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramARB returns FALSE. A name returned by GenProgramsARB, but not yet bound, is not the name of a program object.

**Modify Section 6.2, State Tables (p. 216)**

(add to caption of Table 6.5) When accessing the current texture coordinates (CURRENT_TEXTURE_COORDS) or the texture coordinates associated with raster position (CURRENT_RASTER_TEXTURE_COORDS), the active texture unit selector (ACTIVE_TEXTURE) must be less than the implementation dependent maximum number of texture coordinate sets (MAX_TEXTURE_COORDS_ARB).

(add to caption of Table 6.8) When accessing the texture matrix stack (TEXTURE_MATRIX, TRANSPOSE_TEXTURE_MATRIX) or the texture matrix stack pointer (TEXTURE_STACK_DEPTH), the active texture unit selector (ACTIVE_TEXTURE) must be less than the implementation dependent maximum number of texture coordinate sets (MAX_TEXTURE_COORDS_ARB).

(split Table 6.17 into two tables, Texture Environment and Texture Coordinate Generation; move active texture unit selector and texture coordinate generation state to table 6.18; renumber subsequent tables)

(add to captions of Tables 6.14, 6.15, 6.16) The active texture unit selector (ACTIVE_TEXTURE) identifies which texture object is accessed, and must be less than the implementation dependent maximum number of texture image units (MAX_TEXTURE_IMAGE_UNITS_ARB).

(add to caption of Table 6.18) With the exception of ACTIVE_TEXTURE, the active texture unit selector (ACTIVE_TEXTURE) identifies which texture coordinate set is accessed, and must be less than the implementation dependent maximum number of texture coordinate sets (MAX_TEXTURE_COORDS_ARB).

## Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

**Add to end of Section A.3 (p. 242):**

Rule 4. Fragment program instructions not relevant to the calculation of any result must have no effect on that result.

Rule 5. Fragment program instructions relevant to the calculation of any result must always produce the identical result.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between fragment colors generated by conventional GL texturing mode and fragment colors generated by fragment program mode.  Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional GL fragment texturing mode with fragment program mode for different rendering passes.  However, such algorithms will operate correctly if the algorithms limit themselves to a single mode of fragment color generation.

There is no guaranteed invariance between the final z window coordinates of fragments processed by fragment programs that write depth values and fragments processed by any other means, even if the fragment programs in question simply copy the z value from the "fragment.position" binding.  Multi-pass rendering algorithms that use depth-replacing fragment programs should use depth-replacing fragment programs on each pass to guarantee identical z values.

The texture sample chosen for a fragment of a primitive must be invariant between fragment program mode and conventional texture application mode subject to these conditions:

1. All state with the exception of fragment program state is identical

2. The primitives generating the fragments are identical

3. The sample in the fragment program mode is the result of a 'TEX' instruction (or a 'TXP' instruction with a unity q)

4. The texture coordinate operand for the texture instruction uses the same texture coordinate set as the conventional mode sample

5. The texture coordinate operand for the texture instruction has not been the result of any other operations in the fragment program

**Additions to the AGL/GLX/WGL Specifications**

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists.  No change is made to the AGL/GLX/WGL API.

Changes to program objects shared between multiple rendering contexts will be serialized (i.e., the changes will occur in a specific order).

Changes to a program object made by one rendering context are not guaranteed to take effect in another rendering context until the other calls BindProgram to bind the program object.

When a program object is deleted by one rendering context, the object itself is not destroyed until it is no longer the current program object in any context.  However, the name of the deleted object is removed from the program object name space, so the next attempt to bind a program using the same name will create a new program object.  Recall that destroying a program object bound in

    the current rendering context effectively unbinds the object being
    destroyed.

**Dependencies on OpenGL 1.4**

    If OpenGL 1.4 is not supported, the modified equation for the
    calculation of level of detail by the TXB instruction in 3.11.6.3
    should read

```
      lambda'(x,y) = log2[p(x,y)] +
                     clamp(texunit_bias + fragment_bias)
```

**Dependencies on EXT_vertex_weighting and ARB_vertex_blend**

    If EXT_vertex_weighting and ARB_vertex_blend are both not supported,
    all discussions of multiple modelview matrices should be removed.

    In particular, the line in the grammar

```
      <stateMatrixName>      ::= "modelview" <stateOptModMatNum>
```

    should be changed to

```
      <stateMatrixName>      ::= "modelview"
```

    and the rules <stateOptModMatNum> and <stateModMatNum> should be
    deleted.  The first line of Table X.2.7 should be modified to read:

```
      Binding                              Underlying State
      ------------------------------------  --------------------------
        state.matrix.modelview              modelview matrix
```

    The caption for Table X.2.7 should be modified to exclude optional
    modelview matrix number.  Subsequent references to "modelview matrix
    zero" and "modelview matrix 1" should be changed to "modelview
    matrix" and the example "state.matrix.modelview[1].row[0]" should be
    changed to "state.matrix.modelview.row[0]".

**Dependencies on ARB_matrix_palette:**

    If ARB_matrix_palette is not supported, all discussions of the
    matrix palette should be removed.

    In particular, the line

```
      "palette" "[" <statePaletteMatNum> "]"
```

    should be removed from the <stateMatrixName> grammar rule, and the
    <statePaletteMatNum> grammar rule should be removed entirely.
    "state.matrix.palette[n]" should be removed from Table X.2.7.

**Dependencies on ARB_transpose_matrix**

    If ARB_transpose_matrix is not supported, the discussion of
    TRANSPOSE_CURRENT_MATRIX_ARB in the edits to section 6.1.2 should be
    removed.

**Dependencies on EXT_fog_coord**

If EXT_fog_coord is not supported, references to "fog coordinate" in the definition of the "fragment.fogcoord" attribute should be removed.

**Dependencies on NV_texture_rectangle**

If NV_texture_rectangle is not supported, the discussion of the rectangle (non-power-of-two) texture target in section 3.11.6 should be removed, and the line

```
  "RECT"
```

should be removed from the <texTarget> grammar rule.

**Interactions with ARB_shadow**

The texture comparison introduced by ARB_shadow can be expressed in terms of a fragment program, and in fact use the same internal resources on some implementations.  Therefore, if fragment program mode is enabled, the GL behaves as if TEXTURE_COMPARE_MODE_ARB is NONE.

**Interactions with ARB_vertex_program**

The program object management entrypoints described in sections 2.14.1 (for vertex programs) and 3.11.1 (for fragment programs) are shared by both program targets.  The PROGRAM_ERROR_STRING_ARB and program queries in sections 6.1.11 and 6.1.12 are also shared, as are all common tokens.

The Errors section should be modified to generate INVALID_OPERATION from the Get command with argument CURRENT_MATRIX_ARB, TRANSPOSE_CURRENT_MATRIX_ARB, and CURRENT_MATRIX_STACK_DEPTH_ARB when the current matrix mode is TEXTURE.

In the presence of ARB_vertex_program, ARB_fragment_program must recognize and return appropriate values for the GetProgram <pname> tokens introduced in that spec but not otherwise shared by ARB_fragment_program:

```
    PROGRAM_ADDRESS_REGISTERS_ARB                   0x88B0
    MAX_PROGRAM_ADDRESS_REGISTERS_ARB               0x88B1
    PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB            0x88B2
    MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB        0x88B3
```

The following tables list new program object state and implementation-dependent state:

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attrib |
|---|---|---|---|---|---|---|
| PROGRAM_ADDRESS_REGISTERS_ARB | Z+ | GetProgramivARB | 0 | bound program address registers | 6.1.12 | - |
| PROGRAM_NATIVE_ADDRESS_ REGISTERS_ARB | Z+ | GetProgramivARB | 0 | bound program native address registers | 6.1.12 | - |

Table X.7. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

| Get Value | Type | Get Command | Minimum Value | Description | Sec. | Attrib |
|---|---|---|---|---|---|---|
| MAX_PROGRAM_ADDRESS_REGISTERS_ARB | Z+ | GetProgramivARB | 0 | maximum program address registers | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_ADDRESS_ REGISTERS_ARB | Z+ | GetProgramivARB | 0 | maximum program native address registers | 6.1.12 | - |

Table X.10. New Implementation-Dependent Values Introduced by ARB_vertex_program.

In the presence of ARB_fragment_program, ARB_vertex_program must recognize and return appropriate values for the GetProgram <pname> tokens introduced in this spec. The following tables list new program object state and implementation-dependent state:

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attrib |
|---|---|---|---|---|---|---|
| PROGRAM_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program ALU instructions | 6.1.12 | - |
| PROGRAM_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program texture instructions | 6.1.12 | - |
| PROGRAM_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program texture indirections | 6.1.12 | - |
| PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program native ALU instructions | 6.1.12 | - |
| PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program native texture instructions | 6.1.12 | - |
| PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program native texture indirections | 6.1.12 | - |

Table X.7. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.

| Get Value | Type | Get Command | Minimum Value | Description | Sec. | Attrib |
|---|---|---|---|---|---|---|
| MAX_PROGRAM_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | Number of frag. prg. ALU instructions | 6.1.12 | - |
| MAX_PROGRAM_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | Number of frag. prg. texture instructions | 6.1.12 | - |
| MAX_PROGRAM_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 0 | Number of frag. prg. texture indirections | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program native ALU instructions | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program native texture instructions | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 0 | maximum program native texture indirections | 6.1.12 | - |

Table X.10. New Implementation-Dependent Values Introduced by ARB_fragment_program.

**Interactions with ATI_fragment_shader**

The existing ATI_fragment_shader extension, if supported, also provides a similar fragment programming model. Mixing the two models in a single application is possible but not recommended. FRAGMENT_PROGRAM_ARB has priority over FRAGMENT_SHADER_ATI if both are enabled.

**Interactions with NV_fragment_program**

The NV_fragment_program extension, if supported, also provides a similar programming model. This extension is incompatible with NV_fragment_program in a number of different ways. Mixing the two models in a single application is possible but not recommended. The interactions between the extensions are defined below.

Functions, enumerants, and programs defined in NV_fragment_program are called "NV functions", "NV enumerants", and "NV programs," respectively. Functions, enumerants, and programs defined in ARB_fragment_program are called "ARB functions", "ARB enumerants", and "ARB programs," respectively.

The following GL state is identical in the two extensions:

- Fragment program mode enable. The NV and ARB enumerants have different values, but the same effect.

- Program error position.

- Program error string.

- NV_fragment_program and ARB_fragment_program "program local parameters."

- Fragment program names, targets, formats, program string, program string lengths, and residency information. The ARB and NV query functions operate differently. The ARB query function does not allow queries of target (passed in to the query) and residency information. The NV query function does not allow queries of program name (passed in to the query) or format. The format of NV programs is always PROGRAM_FORMAT_ASCII_ARB.

- Program object name space. Program objects are created differently in the NV and ARB specs. Under the NV spec, program objects are created by calling LoadProgramNV. Under the ARB spec, program objects are created by calling BindProgramARB with an unused program name.

The following state is provided only by ARB_fragment_program:

- Program environment parameters.

- Implementation-dependent limits on the number of instructions, ALU instructions, texture instructions, texture indirections, program parameters, fragment attributes, resource counts, and native resource counts. The instruction limit is baked into the NV spec. Implementations supporting NV_fragment_program have no specific restrictions on the number of ALU instructions, texture instructions, texture indirections, or fragment attributes used. Such implementations also have no limit on program parameters used, except that no more than one may be used by any single program instruction.

The following state is provided only by NV_fragment_program:

- Named program parameters (variables defined in the program text and updated by name).

The following are additional functional differences between ARB_fragment_program and NV_fragment_program:

- NV programs use a set of register names, with no support for user-defined variables (other than parameters in the program). ARB programs provide no support for fixed variable names; all variables must be declared, explicitly or implicitly, in the program.

- ARB programs support parameter variables that can be bound to selected GL state variables, and are updated automatically when the underlying state changes. NV programs provide no such support; applications must set program parameters themselves.

- ARB_fragment_program doesn't provide explicit support for multiple data types (fx12, fp16, fp32) described in NV_fragment_program, and provides no mechanism for controlling the precision used to carry out arithmetic operations.

- ARB_fragment_program doesn't support condition codes, conditional writemasks, or the "C" instruction suffix that specifies a condition code update.

- ARB_fragment_program doesn't support an absolute value operator that can be applied to a source vector as it is loaded.

- ARB_fragment_program doesn't define behavior for many floating-point special cases. On platforms where NV_fragment_program is supported, ARB programs will have the same special-case behavior.

```
- Language to declare program parameters is slightly different
  (NV_fragment_program has "DECLARE" and "DEFINE";
  ARB_fragment_program has "PARAM").

- NV_fragment_program provides a number of instructions not found
  in ARB_fragment_program:

    * DDX, DDY:  partial derivatives relative to x and  y.

    * "PK*" and "UP*":  packing and unpacking instructions.

    * RFL:  reflection vector.

    * SEQ, SFL, SGT, SLE, SNE, STR:  set on equal, false, greater
      than, less than or equal, not equal, and true, respectively.

    * TXD:  texture lookup w/partials.

    * X2D:  2D coordinate transformation.

- ARB_fragment_program provides several instructions not found in
  NV_fragment_program, and there are a few instruction set
  differences:

    * ABS:  absolute value.  ABS instructions are unnecessary in
        NV_fragment_program because of the free absolute value on
        input operator.  Equivalent to:

           MOV dst, |src|;

    * CMP:  compare.  Roughly equivalent to the following
        sequence, but may be optimized further:

           SLT tmp, src0;
           LRP dst, tmp, src1, src2;

    * DPH:  homogenous dot product.  Equivalent to:

           DP3 tmp, src0, src1;
           ADD dst, tmp, src0.w;

    * KIL:  kill fragment.  Both extensions support this
        instruction, but the ARB instruction takes a vector
        operand rather than a condition code.

    * SCS:  sine/cosine.  Emulated using the separate SIN and COS
        instructions in NV_fragment_program, which also have no
        restriction on the input values.

    * SWZ:  extended swizzle.  On NV_fragment_program platforms,
        this instruction will be emulated using a single MAD
        instruction and a program parameter constant.

    * TXB:  texture sample with bias.  Not exposed in the
        NV_fragment_program API.
    * XPD:  cross product.  Emulated using a MUL and a MAD
        instruction.
```

**GLX Protocol**

```
    The following rendering commands are sent to the server as part of
    a glXRender request:

       BindProgramArB
           2           12              rendering command length
           2           4180            rendering command opcode
           4           ENUM            target
           4           CARD32          program

       ProgramEnvParameter4fvARB
           2           32              rendering command length
           2           4184            rendering command opcode
           4           ENUM            target
           4           CARD32          index
           4           FLOAT32         params[0]
           4           FLOAT32         params[1]
           4           FLOAT32         params[2]
           4           FLOAT32         params[3]

       ProgramEnvParameter4dvARB
           2           44              rendering command length
           2           4185            rendering command opcode
           4           ENUM            target
           4           CARD32          index
           8           FLOAT64         params[0]
           8           FLOAT64         params[1]
           8           FLOAT64         params[2]
           8           FLOAT64         params[3]

       ProgramLocalParameter4fvARB
           2           32              rendering command length
           2           4215            rendering command opcode
           4           ENUM            target
           4           CARD32          index
           4           FLOAT32         params[0]
           4           FLOAT32         params[1]
           4           FLOAT32         params[2]
           4           FLOAT32         params[3]

       ProgramLocalParameter4dvARB
           2           44              rendering command length
           2           4216            rendering command opcode
           4           ENUM            target
           4           CARD32          index
           8           FLOAT64         params[0]
           8           FLOAT64         params[1]
           8           FLOAT64         params[2]
           8           FLOAT64         params[3]

    The ProgramStringARB is potentially large, and hence can be sent in
    a glXRender or glXRenderLarge request.
```

```
        ProgramStringARB
            2           16+len+p        rendering command length
            2           4217            rendering command opcode
            4           ENUM            target
            4           ENUM            format
            4           sizei           len
            len         LISTofBYTE      program
            p                           unused, p=pad(len)

         If the command is encoded in a glxRenderLarge request, the
         command opcode and command length fields above are expanded to
         4 bytes each:

            4           16+len+p        rendering command length
            4           4217            rendering command opcode

    The remaining commands are non-rendering commands.  These commands
    are sent separately (i.e., not as part of a glXRender or
    glXRenderLarge request), using the glXVendorPrivateWithReply
    request:

        DeleteProgramsARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+n             request length
            4           1294            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
            n*4         LISTofCARD32    programs

        GenProgramsARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1295            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           n               reply length
            24                          unused
            n*4         LISTofCARD322   programs
        **GetProgramEnvParameterfvARB**
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1296            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n (number of parameter components)

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        **GetProgramEnvParameterdvARB**
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1297            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            target
            4           CARD32          index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n*2)
            4                           unused
            4           CARD32          n (number of parameter components)

            if (n=1) this follows:

            8           FLOAT64         params
            8                           unused

            otherwise this follows:

            16                          unused
            n*8         LISTofFLOAT64   params
```

**GetProgramLocalParameterfvARB**
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1305            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n (number of parameter components)

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
```

**GetProgramLocalParameterdvARB**
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1306            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n (number of parameter components)

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params
```

**GetProgramivARB**
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1307            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params
```

**GetProgramStringARB**
```
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1308            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           (n+p)/4         reply length
    4                           unused
    4           CARD32          n
    16                          unused
    n           STRING          program
    p                           unused, p=pad(n)
```

```
        IsProgramARB
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1304            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
```

**Errors**

The error INVALID_OPERATION is generated by ProgramStringARB if the program string <string> is syntactically incorrect or violates any semantic restriction of the execution environment of the specified program target <target>. The error INVALID_OPERATION may also be generated by ProgramStringARB if the specified program would exceed native resource limits of the implementation.

The error INVALID_OPERATION is generated by BindProgramARB if <program> is the name of a program whose target does not match <target>.

The error INVALID_VALUE is generated by commands ProgramEnvParameter{fd}ARB, ProgramEnvParameter{fd}vARB, and GetProgramEnvParameter{fd}vARB if <index> is greater than or equal to the value of MAX_PROGRAM_ENV_PARAMETERS_ARB corresponding to the program target <target>.

The error INVALID_VALUE is generated by commands ProgramLocalParameter4{fd}ARB, ProgramLocalParameter4{fd}vARB, and GetProgramLocalParameter{fd}vARB if <index> is greater than or equal to the value of MAX_PROGRAM_LOCAL_PARAMETERS_ARB corresponding to the program target <target>.

The error INVALID_OPERATION is generated if Begin, RasterPos, or any command that performs an explicit Begin is called when fragment program mode is enabled and the currently bound fragment program object does not contain a valid fragment program.

The error INVALID_OPERATION is generated by any command accessing texture coordinate processing state if the texture unit number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_COORDS_ARB. Such commands include: GetTexGen{if}v; TexGen{ifd}, TexGen{ifd}v; Disable, Enable, IsEnabled with argument TEXTURE_GEN_{STRQ}; Get with argument CURRENT_TEXTURE_COORDS, CURRENT_RASTER_TEXTURE_COORDS, TEXTURE_STACK_DEPTH, TEXTURE_MATRIX, TRANSPOSE_TEXTURE_MATRIX; when the current matrix mode is TEXTURE, Frustum, LoadIdentity, LoadMatrix{fd}, LoadTransposeMatrix{fd}, MultMatrix{fd}, MultTransposeMatrix{fd}, Ortho, PopMatrix, PushMatrix, Rotate{fd}, Scale{fd}, Translate{fd}.

```
    The error INVALID_OPERATION is generated by any command accessing
    texture image processing state if the texture unit number
    corresponding to the current value of ACTIVE_TEXTURE is greater than
    or equal to the implementation-dependent constant
    MAX_TEXTURE_IMAGE_UNITS_ARB.  Such commands include: BindTexture;
    GetCompressedTexImage, GetTexEnv{if}v, GetTexImage,
    GetTexLevelParameter{if}v, GetTexParameter{if}v; TexEnv{if},
    TexEnv{if}v, TexParameter{if}, TexParameter{if}v; Disable, Enable,
    IsEnabled with argument TEXTURE_{123}D, TEXTURE_CUBE_MAP; Get with
    argument TEXTURE_BINDING_{123}D, TEXTURE_BINDING_CUBE_MAP;
    CompressedTexImage{123}D, CompressedTexSubImage{123}D,
    CopyTexImage{12}D, CopyTexSubImage{123}D, TexImage{123}D,
    TexSubImage{123}D.
```

**New State**

| Get Value | Type | Get Command | Initial Value | Description | Section | Attribute |
|---|---|---|---|---|---|---|
| FRAGMENT_PROGRAM_ARB | B | IsEnabled | False | fragment program enable | 3.8 | enable |
| - | 24+xR4 | GetProgramEnv-ParameterARB | (0,0,0,0) | program environment parameters | 3.11.1 | - |
| PROGRAM_ERROR_POSITION_ARB | Z | GetIntegerv | -1 | last program error position | 3.11.1 | - |
| PROGRAM_ERROR_STRING_ARB | 0+xub | GetString | "" | last program error string | 3.11.1 | - |

**Table X.6.  New Accessible State Introduced by ARB_fragment_program.**

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attrib |
|---|---|---|---|---|---|---|
| PROGRAM_BINDING_ARB | Z+ | GetProgramivARB | object-specific | bound program name | 6.1.12 | - |
| PROGRAM_LENGTH_ARB | Z+ | GetProgramivARB | 0 | bound program length | 6.1.12 | - |
| PROGRAM_FORMAT_ARB | Z1 | GetProgramivARB | PROGRAM_FORMAT_ ASCII_ARB | bound program format | 6.1.12 | - |
| PROGRAM_STRING_ARB | ubxn | GetProgramStringARB | (empty) | bound program string | 6.1.12 | - |
| PROGRAM_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program total instructions | 6.1.12 | - |
| PROGRAM_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program ALU instructions | 6.1.12 | - |
| PROGRAM_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program texture instructions | 6.1.12 | - |
| PROGRAM_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program texture indirections | 6.1.12 | - |
| PROGRAM_TEMPORARIES_ARB | Z+ | GetProgramivARB | 0 | bound program temporaries | 6.1.12 | - |
| PROGRAM_PARAMETERS_ARB | Z+ | GetProgramivARB | 0 | bound program parameter bindings | 6.1.12 | - |
| PROGRAM_ATTRIBS_ARB | Z+ | GetProgramivARB | 0 | bound program attribute bindings | 6.1.12 | - |
| PROGRAM_NATIVE_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program native instructions | 6.1.12 | - |
| PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program native ALU instructions | 6.1.12 | - |
| PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program native texture instructions | 6.1.12 | - |
| PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 0 | bound program native texture indirections | 6.1.12 | - |
| PROGRAM_NATIVE_TEMPORARIES_ARB | Z+ | GetProgramivARB | 0 | bound program native temporaries | 6.1.12 | - |
| PROGRAM_NATIVE_PARAMETERS_ARB | Z+ | GetProgramivARB | 0 | bound program native parameter bindings | 6.1.12 | - |
| PROGRAM_NATIVE_ATTRIBS_ARB | Z+ | GetProgramivARB | 0 | bound program native attribute bindings | 6.1.12 | - |
| PROGRAM_UNDER_NATIVE_LIMITS_ARB | B | GetProgramivARB | 0 | bound program under native resource limits | 6.1.12 | - |
| - | 24+xR4 | GetProgramLocal-ParameterARB | (0,0,0,0) | bound program local parameter value | 3.11.1 | - |

**Table X.7. Program Object State. Program object queries return attributes of the program object currently bound to the program target <target>.**

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| - | 16+xR4 | - | undefined | temporary registers | 3.11.3.3 | - |
| - | 2xR4 | - | undefined | fragment result registers | 3.11.3.4 | - |

**Table X.8. Fragment Program Per-fragment Execution State. All per-fragment execution state registers are uninitialized at the beginning of program execution.**

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| CURRENT_MATRIX_ARB | m*n*xM^4 | GetFloatv | Identity | current matrix | 6.1.2 | - |
| CURRENT_MATRIX_STACK_DEPTH_ARB | m*Z+ | GetIntegerv | 1 | current stack depth | 6.1.2 | - |

**Table X.9. Current matrix state where m is the total number of matrices including texture matrices and program matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.**

**New Implementation Dependent State**

| Get Value | Type | Get Command | Minimum Value | Description | Sec. | Attrib |
|---|---|---|---|---|---|---|
| MAX_TEXTURE_COORDS_ARB | Z+ | GetIntegerv | 2 | number of texture coordinate sets | 2.7 | - |
| MAX_TEXTURE_IMAGE_UNITS_ARB | Z+ | GetIntegerv | 2 | number of texture image units | 2.10.2 | - |
| MAX_PROGRAM_ENV_PARAMETERS_ARB | Z+ | GetProgramivARB | 24 | maximum program env parameters | 3.11.1 | - |
| MAX_PROGRAM_LOCAL_PARAMETERS_ARB | Z+ | GetProgramivARB | 24 | maximum program local parameters | 3.11.1 | - |
| MAX_PROGRAM_MATRICES_ARB | Z+ | GetIntegerv | 8 (not to exceed 32) | maximum number of program matrices | 3.11.7 | - |
| MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB | Z+ | GetIntegerv | 1 | maximum program matrix stack depth | 3.11.7 | - |
| MAX_PROGRAM_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 72 | maximum program total instructions | 6.1.12 | - |
| MAX_PROGRAM_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 48 | number of frag. prg. ALU instructions | 6.1.12 | - |
| MAX_PROGRAM_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | 24 | number of frag. prg. texture instructions | 6.1.12 | - |
| MAX_PROGRAM_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | 4 | number of frag. prg. texture indirections | 6.1.12 | - |
| MAX_PROGRAM_TEMPORARIES_ARB | Z+ | GetProgramivARB | 16 | maximum program temporaries | 6.1.12 | - |
| MAX_PROGRAM_PARAMETERS_ARB | Z+ | GetProgramivARB | 24 | maximum program parameter bindings | 6.1.12 | - |
| MAX_PROGRAM_ATTRIBS_ARB | Z+ | GetProgramivARB | 10 | maximum program attribute bindings | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | - | maximum program native total instructions | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | - | maximum program native ALU instructions | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB | Z+ | GetProgramivARB | - | maximum program native texture instructions | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB | Z+ | GetProgramivARB | - | maximum program native texture indirections | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_TEMPORARIES_ARB | Z+ | GetProgramivARB | - | maximum program native temporaries | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_PARAMETERS_ARB | Z+ | GetProgramivARB | - | maximum program native parameter bindings | 6.1.12 | - |
| MAX_PROGRAM_NATIVE_ATTRIBS_ARB | Z+ | GetProgramivARB | - | maximum program native attribute bindings | 6.1.12 | - |

**Table X.10. New Implementation-Dependent Values Introduced by ARB_fragment_program. Values queried by GetProgram require a <pname> of FRAGMENT_PROGRAM_ARB.**

**Sample Usage**

The following program shows how to perform a simple modulation between the interpolated color and a single texture:

```
!!ARBfp1.0
# Simple program to show how to code up the default texture environment
ATTRIB tex = fragment.texcoord;      #first set of texture coordinates
ATTRIB col = fragment.color.primary; #diffuse interpolated color
OUTPUT outColor = result.color;
TEMP tmp;
TXP tmp, tex, texture, 2D;           #sample the texture
MUL outColor, tmp, col;              #perform the modulation
END
```

    The following is an example the simulates a chrome surface:

```
      !!ARBfp1.0
      ########################
      # Input Textures:
      #-----------------------
      # Texture 0 contains the default 2D texture used for general mapping
      # Texture 2 contains a 1D pointlight falloff map
      # Texture 3 contains a 2D map for calculating specular lighting
      # Texture 4 contains normalizer cube map
      # Input Texture Coordinates:
      #-----------------------
      # TexCoord1 contains the calculated normal
      # TexCoord2 contains the light to vertex vector
      # TexCoord3 contains the half-vector in tangent space
      # TexCoord4 contains the light vector in tangent space
      # TexCoord5 contains the eye vector in tangent space
      ########################
      TEMP     NdotH, lV, L;
      ALIAS    diffuse  = L;
      PARAM       half  = { 0.5, 0.5, 0.5, 0.5 };
      ATTRIB   norm_tc  = fragment.texcoord[1];
      ATTRIB     lv_tc  = fragment.texcoord[2];
      ATTRIB   half_tc  = fragment.texcoord[3];
      ATTRIB  light_tc  = fragment.texcoord[4];
      ATTRIB    eye_tc  = fragment.texcoord[5];
      OUTPUT      oCol  = result.color;
      TEX     L, light_tc, texture[4], CUBE; # Sample cube map normalizer
      # Calculate diffuse lighting (N.L)
      SUB     L, L, half;             # Bias L and then multiply by 2
      ADD     L, L, L;
      DP3     diffuse, norm_tc, L;     # N.L
      # Calculate specular lighting component { (N.H), |H|^2 }
      DP3     NdotH.x, norm_tc, half_tc;
      DP3     NdotH.y, half_tc, half_tc;
      DP3     lV.x, lv_tc, lv_tc;     # lV = (|light to vertex|)^2
      #############
      # Pass 2
      #############
      TEMP     base, specular;
      ALIAS    atten  = lV;
      TEX base, eye_tc, texture[0], 2D; # sample enviroment map using eye vector
      TEX atten,    lV,     texture[2], 1D; # Sample attenuation map
      TEX specular, NdotH,  texture[3], 2D; # Sample specular NHHH map=(N.H)^256
      # specular = (N.H)^256 * (N.L)
      # this ensures a pixel is only lit if facing the light (since the specular
      # exponent makes negative N.H positive we must do this)
      MUL      specular, specular, diffuse;
      # specular = specular * environment map
      MUL      specular, base, specular;
      # diffuse = diffuse * environment map
      MUL      diffuse, base, diffuse;
      # outColor = (specular * environment map) + (diffuse * environment map)
      ADD      base, specular, diffuse;
      # Apply point light attenutaion
      MUL      oCol, base, atten.r;
      END
```

**Revision History**

```
    Date: 8/22/2003
    Revision: 26
       - Added list of commands generating errors when active texture
         unit selector is out of range.
       - Fixed typo in <stateMatrixItem> rule.
       - Clarified behavior of fragment.position.z with respect to depth
         offset.

    Date: 2/26/2003
    Revision: 25
       - Fixed description of KIL instruction to reflect less than zero
         test, not less than or equal to zero.
       - Clarified the processing of incoming and outgoing depths and
         colors to reflect the conversion to floating-point on input and
         the conversion to fixed-point on output.

    Date: 1/10/2003
    Revision: 24
       - Fixed bug where "state.matrix.mvp" was specified incorrectly.
         It should be P*M0 rather than M0*P.
       - Added issue warning about CMP opcode's order of operands.

    Date: 10/22/2002
    Revision: 23
       - Fixed reference to <extSwizComp> rule in 3.11.5.28.  Instead
         reference both <xyzwExtSwizComp> and <rgbaExtSwizComp> rules.

    Date: 10/02/2002
    Revision: 22
       - Fixed typo in section 3.11.1, where 8 program environment and
         8 program local parameters are listed as the minimums instead
         of 24 of each.  Table X.10 had the correct values.
       - Fixed <stateTexEnvItem> to refer to legacy texture units.
       - Fixed typos in issue 29 pseudo-code, added some clarification.

    Date: 9/19/2002
    Revision: 21
       - Added clarifying paragraph for native texture indirection
         counting, offering examples of possible cases where texture
         indirections may be increased.
       - Fixed typos in issues 25 and 29.

    Date: 9/16/2002
    Revision: 20
       - Added precision hint program options.
       - Fixed various typos, reworded some parts for consistency.
       - Updated issues list.
```

```
Date: 9/13/2002
Revision: 19
   - Promoted minimum precision of texture coordinates in 2.1.1.
   - Added ARB_fog_* program options.
   - Removed modification to 3.9, put clamps in 3.11.4.4.
   - Made 'texture' a reserved keyword in the grammar.
   - Fixed various typos.
   - Updated section 3.11.6.
   - Updated issues list.

Date: 9/11/2002
Revision: 18
   - Updated for consistency with ARB_vertex_program revision 36.
   - Depth output moved to 3rd component of result.depth.
   - Fixed various typos, reworded things in many places.
   - Added NV_fragment_program interactions.
   - Updated issues list.

Date: 9/09/2002
Revision: 17
   - Added fogcoord and position attributes.
   - Moved fragment program section to 3.11, after fog.
   - Changed MAX_TEXTURE_UNITS/MAX_AUX_TEXTURE_UNITS to
     MAX_TEXTURE_COORDS/MAX_TEXTURE_IMAGE_UNITS.
   - Removed TRC and MOD instructions.
   - Added SIN and COS instructions.
   - Added more clarity to resource consumption wording.
   - Added invariance wording concerning depth-replacement.
   - Added rule that a program that fails to load must always fail to
     load, regardless of GL state.
   - Updated issues list.

Date: 8/30/2002
Revision: 16
   - Improved texture indirection description.
   - Defined result of sample from incomplete texture as (0,0,0,1).
   - Removed PROGRAMS_LOAD_OVER_NATIVE_LIMITS_ARB per-target query.
   - Allowed ProgramStringARB to fail on non-native programs.
   - Updated issues list.

Date: 8/28/2002
Revision: 15
   - Updated for consistency with ARB_vertex_program revision 35.
   - Added PROGRAMS_LOAD_OVER_NATIVE_LIMITS_ARB per-target query.
   - Changed MAX_AUX_TEXTURE_UNITS_ARB enum value.
   - Updated issues list.

Date: 8/22/2002
Revision: 14
   - Added sine/cosine instruction (SCS).
   - Updated texture sample grammar, replaced texenables hierarchy.
   - Added EXT_vertex_weighting and ARB_vertex_blend dependency.
   - Updated issues list.
```

```
Date: 8/14/2002
Revision: 13
   - Fixed <paramConstant> grammar rule.
   - Updated issues list.

Date: 8/06/2002
Revision: 12
   - Fixed various typos.
   - Updated issues list.
   - Added wording to 3.10.3.6 to reflect that native resource
     consumption may increase due to emulated instructions.

Date: 7/29/2002
Revision: 11
   - Updated for consistency with ARB_vertex_program revision 34.
   - Added support for matrix binding.
   - Removed precision queries.
   - Updated issues list.

Date: 7/16/2002
Revision: 10
   - Updated for consistency with ARB_vertex_program revision 31.
   - Added fog params and depth range bindings to grammar.
   - Removed stpq writemasks and swizzles from grammar.
   - Required swizzle components to come from same set, xyzw or rgba.

Date: 7/10/2002
Revision: 9
   - Made fog params and depth range bindable.
   - Changed texture instruction names to match 3-letter format.
   - Made texture instructions more consistent with ALU instructions.
   - Increased minimums for implementation-dependent values.
   - Re-introduced 4-components swizzles and the SWZ instruction.
   - Updated issues list.

Date: 7/03/2002
Revision: 8
   - Fixed typos.
   - Added DST, LIT, SGE, SLT instructions.
   - Changed FRC definition to match ARB_vertex_program, added MOD
     instruction to expose fmod(arg, 1.0) behavior.

Date: 6/25/2002
Revision: 7
   - Updated for consistency with ARB_vertex_program revision 29.

Date: 6/19/2002
Revision: 6
   - Updated for consistency with ARB_vertex_program revision 28.
   - Changed from ATI to ARB prefix/suffix.
   - Started using single integer revision number.
   - Added a few more issues to the list.

Date: 6/14/2002
Revision: 1.4
   - Updated for consistency with ARB_vertex_program revision 27.
   - Added a few more issues to the list.
```

```
Date: 6/05/2002
Revision: 1.3
   - Updated for consistency with ARB_vertex_program revision 26.
   - Incorporated program object management, removing dependency on
     ARB_vertex_program.
   - Added interaction with ARB_shadow.

Date: 6/03/2002
Revision: 1.2
   - Updated for consistency with ARB_vertex_program revision 25.
   - Fixed TexInstructions to use <texSrcReg>, i.e. no parameters.
   - Added TRC, POW, DPH instructions, updated FRC and LRP.
   - Added fog color parameter binding.

Date: 5/23/2002
Revision: 1.1
   - Updated for consistency with ARB_vertex_program revision 24.
   - Added GetProgramfvATI entrypoint for querying precision values.

Date: 5/10/2002
Revision: 1.0
   - First draft for circulation.
```

**Name**

    ARB_fragment_program_shadow

**Name Strings**

    GL_ARB_fragment_program_shadow

**IP Status**

    Unknown, but Microsoft claims to own intellectual property
    related to ARB_fragment_program.  This extension is
    an extension to ARB_fragment_program.

**Status**

    Complete.  Approved by ARB on December 16, 2003

**Version**

    Last Modified Date: December 8, 2003
    Revision: 5

**Number**

    ARB Extension #36

**Dependencies**

    The extension is written against the OpenGL 1.3 Specification.

    ARB_fragment_program is required.

    ARB_shadow is required.

    EXT_texture_rectange affects the definition of this extension.

**Overview**

    This extension extends ARB_fragment_program to remove
    the interaction with ARB_shadow.

    This extension defines the program option
    "ARB_fragment_program_shadow".

    If a fragment program specifies the option
    "ARB_fragment_program_shadow"

        SHADOW1D, SHADOW2D, SHADOWRECT

    are added as texture targets.  When shadow map comparisons are
    desired, specify the SHADOW1D, SHADOW2D, or SHADOWRECT texture
    targets in texture instructions.

    Programs must assure that the comparison mode for each depth
    texture (TEXTURE_COMPARE_MODE) and/or the internal texture
    format (DEPTH_COMPONENT) and the targets of the texture lookup

```
    instructions match.  Otherwise, if the comparison mode
    and/or the internal texture format are inconsistent with the
    texture target, the results of the texture lookup are undefined.
```

**Issues**

*(1) What should this extension be called?*

```
      RESOLVED:  ARB_fragment_program_shadow.  Shadow support
      is the only new feature.  The name ARB_fragment_program2
      should be used for a far more major revision to
      ARB_fragment_program.  ARB_fragment_program1_1 is
      less descriptive.
```

*(2) Should this extension use the header string "!!ARBfp1.1" or a program option "ARB_fragment_program_shadow"?*

```
      RESOLVED: Program option "ARB_fragment_program_shadow".
```

*(3) What form should the ARB_fragment_program_shadow option take?*

```
        a.  New sampler instructions.
            SHX result.color.a, fragment.texcoord[1], texture[0], 2D;

        b.  New texture modifiers.
            TEX result.color.a, fragment.texcoord[1], texture[0], 2D,SHADOW;

        c.  New texture targets.
            TEX result.color.a, fragment.texcoord[1], texture[0], SHADOW2D;

        d.  New sampler instructions AND new texture modifiers.
            SHX result.color.a, fragment.texcoord[1], texture[0], 2D,SHADOW;

        e.  New sampler instructions AND new texture targets.
            SHX result.color.a, fragment.texcoord[1], texture[0], SHADOW2D;

    RESOLVED:  Choose the simplest option c, add new texture targets.

    All of the above forms are functionally equivalent.

    An earlier draft proposed option a, adding six new shadow
    instructions.  The required shadow instructions are
    three variants of shadow instruction (non-projective, projective,
    and biased), and the same instructions with the modifier _SAT.

    Option b adds texture modifiers but requires additional semantic
    restrictions.

    Option c adds texture targets only.  It is a sufficient
    and simple change to one grammar rule.

    Option d and e are listed for completeness.  They require
    additional instructions and additional semantic restrictions.

    Note that option e is most similar to the resolution of this issue
    by ARB_fragment_shader and the OpenGL Shading Language.  The OpenGL
    Shading Language has both built-in texture and shadow functions and
```

sampler types, analogous to texture instructions and texture targets. The resolution here drops the added reduntancy and potential error checking in favor of simplicity, but is otherwise consistent. This resolution is also consistent with the precident already established in ARB_fragment_program, since we have a TEX instruction, not a TEX1D, TEX2D, TEXCUBE, TEX3D, TEXRECT instructions.

*(4) How should ARB_fragment_program_shadow function?*

a. Simply remove the interaction with ARB_shadow so that TEXTURE_COMPARE_MODE behaves exactly as specified in the OpenGL 1.4 specification.

b. Add "SHADOW" targets to texture lookup instructions. TEXTURE_COMPARE_MODE is ignored. For samples from a SHADOW target TEXTURE_COMPARE_MODE is treated as COMPARE_R_TO_TEXTURE; otherwise, it is treated as NONE.

c. Like (b), but with undefined results if TEXTURE_COMPARE_MODE and/or the internal format of the texture does not match the target.

d. A hybrid of (a) and (b), where the SHADOW target means to use the TEXTURE_COMPARE_MODE state.

RESOLVED - Option c, undefined behavior when the target and mode do not match.

Program text is not simply loaded, it is compiled, optimized and then loaded. Options a and d would remove information from the optimizer. Which components of the texture coordinate are required for the sample? Specifically, is the r component of the texture coordinate required? Options b and c are both sufficient and retain the information required by optimizers. Option c is consistent with the resolution chosen by ARB_fragment_shader.

*(5) What if additional texture compare modes are added by future extensions to ARB_SHADOW?*

We do not anticipate future extensions adding additional texture compare modes. Only the additional mode COMPARE_T_TO_TEXTURE has even marginal utility, and then only for SHADOW1D targets. However, a future extension adding additional texture compare modes is not precluded. The language in this specification is carefully, if somewhat awkwardly, written to say the TEXTURE_COMPARE_MODE either "is NONE" or "is not NONE.

*(6) Does EXT_shadow_funcs interact with this extension?*

RESOLVED: It doesn't. ARB_shadow supports LEQUAL or GEQUAL comparison functions. EXT_shadow_funcs simply adds the additional functions LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, and NEVER. Whichever function is specified will be used for the comparison function.

*(7) Does ARB_shadow_ambient interact with this extension?*

```
RESOLVED:  It doesn't.  ARB_shadow returns a result
in the range [0,1].  ARB_shadow_ambient simply
maps this range to [TEXTURE_COMPARE_FAIL_ARB, 1].
The result will be returned in the specified range.
```

*(8) How would an existing fragment program be ported to use the program option ARB_fragment_program_shadow?*

```
RESOLVED:  Fairly simply, but with a caveat on undefined behavior.

!!ARBfp1.0
# A simple example of shadow map (R <= Dt)
#
# SHOULD make sure that the 2D texture bound to texture unit 0:
#    texture format of DEPTH_COMPONENT (for highest quality comparison)
#    TEXTURE_MAG_FILTER is NEAREST
#    TEXTURE_MIN_FILTER is NEAREST or NEAREST_MIPMAP_NEAREST
# Assumes DEPTH_TEXTURE_MODE is LUMINANCE or INTENSITY
#
TEMP Result;
ALIAS Dt = Result;
TEX Dt, fragment.texcoord[0], texture[0], 2D;
SGE Result, Dt.x, fragment.texcoord[0].z;      # R <= Dt

!!ARBfp1.0
OPTION ARB_fragment_program_shadow;
# A simple example of shadow map (R<= Dt)
#
# MUST make sure that the 2D texture bound to texture unit 0:
#    texture format of DEPTH_COMPONENT and a
#    TEXTURE_COMPARE_MODE of COMPARE_R_TO_TEXTURE
# Otherwise, the Result is undefined.
#
# Remember also that to get R <= Dt to set:
#    TEXTURE_COMPARE_FUNC of LEQUAL
#
# A single compare equivalent to the above example will result if:
#    TEXTURE_MAG_FILTER is NEAREST
#    TEXTURE_MIN_FILTER is NEAREST or NEAREST_MIPMAP_NEAREST
# Otherwise, percent closer filtering may be applied.
#
TEMP Result;
TEX Result, fragment.texcoord[0], texture[0], SHADOW2D;
```

## New Procedures and Functions

None

## New Tokens

None

## 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 <texTarget> grammar rule with

```
<texTarget>            ::= "1D"
                         | "2D"
                         | "3D"
                         | "CUBE"
                         | "RECT"
                         | <shadowTarget> (if program option is present)

<shadowTarget>         ::= "SHADOW1D"
                         | "SHADOW2D"
                         | "SHADOWRECT"
```

**Add Section 3.11.4.5.3 Fragment Program Shadow Option**

If a fragment program specifies the "ARB_fragment_program_shadow" program option, the <texTarget> rule is modified to add the texture targets SHADOW1D, SHADOW2D and SHADOWRECT (See Section 3.11.2).

**Modify Section 3.11.6 Fragment Program Texture Instruction Set**

(replace 1st through 4th paragraphs with the following paragraphs)

The first three texture instructions described below specify the mapping of 4-tuple input vectors to 4-tuple output vectors. The sampling of the texture works as described in section 3.8, except that texture environments and texture functions are not applicable, and the texture enables hierarchy is replaced by explicit references to the desired texture target (i.e., 1D, 2D, 3D, cube map, rectangle). These texture instructions specify how the 4-tuple is mapped into the coordinates used for sampling. The following function is used to describe the texture sampling in the descriptions below:

```
  vec4 TextureSample(float s, float t, float r, float lodBias,
                     int texImageUnit, enum texTarget);
```

Note that not all three texture coordinates, s, t, and r, are used by all texture targets. In particular, 1D texture targets only use the s component. 2D and RECT (non-power-of-two) texture targets only use the s and t components. SHADOW1D texture targets only use the s and r components. The descriptions of the texture instructions below supply all three components, as would be the case with CUBE, 3D, SHADOW2D, and SHADOWRECT targets.

If a fragment program samples from a texture target on a texture image unit where the bound texture object is not complete, as defined in section 3.8.9, the result will be the vector (R, G, B, A) = (0, 0, 0, 1).

If a fragment program does not specify the "ARB_fragment_program_shadow" program option, and if a fragment

```
    program samples from a texture target of 1D, 2D, or RECT, it is as
    if TEXTURE_COMPARE_MODE_ARB is NONE.

    If a fragment program specifies the "ARB_fragment_program_shadow"
    program option, the result returned of a sample from a texture target
    on a texture image unit is undefined if:

      the texture target is 1D, 2D, or RECT, and
      the texture object's internal format is DEPTH_COMPONENT_ARB, and
      the TEXTURE_COMPARE_MODE_ARB is not NONE;

    or

      the texture target is SHADOW1D, SHADOW2D, SHADOWRECT, and
        the texture object's internal format is DEPTH_COMPONENT_ARB, and
      the TEXTURE_COMPARE_MODE_ARB is NONE;

    or

      the texture target is SHADOW1D, SHADOW2D, SHADOWRECT, and
        the texture object's internal format is not DEPTH_COMPONENT_ARB.

    A fragment program will fail to load if it attempts to sample from
    multiple texture targets on the same texture image unit.  For example,
    the following programs would fail to load:

      !!ARBfp1.0
      TEX result.color.rgb, fragment.texcoord[0], texture[0], 2D;
      TEX result.color.a,   fragment.texcoord[1], texture[0], 3D;
      END

      !!ARBfp1.0
      OPTION ARB_fragment_program_shadow;
      TEX result.color.rgb, fragment.texcoord[0], texture[0], 2D;
      TEX result.color.a,   fragment.texcoord[1], texture[0], SHADOW2D;
      END
```

## 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

## Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None

**Additions to the AGL/GLX/WGL Specifications**

None

**Dependencies on EXT_texture_rectangle**

If EXT_texture_rectangle is not supported:

Section 3.11.2 should be modified by removing the line:

```
  | "SHADOWRECT"
```

from the <shadowTarget> grammar rule;

and Section 3.11.6 should be modified by removing the discussion of the rectangle shadow texture target.

## Name

```
    EXT_blend_func_separate
```

## Name Strings

```
    GL_EXT_blend_func_separate
```

## Version

```
    Date: 04/06/1999  Version 1.3
```

## Number

```
    173
```

## Dependencies

```
    None
```

## Overview

```
    Blending capability is extended by defining a function that allows
    independent setting of the RGB and alpha blend factors for blend
    operations that require source and destination blend factors.  It
    is not always desired that the blending used for RGB is also applied
    to alpha.
```

## New Procedures and Functions

```
    void BlendFuncSeparateEXT(enum sfactorRGB,
                              enum dfactorRGB,
                              enum sfactorAlpha,
                              enum dfactorAlpha);
```

## New Tokens

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

        BLEND_DST_RGB_EXT                  0x80C8
        BLEND_SRC_RGB_EXT                  0x80C9
        BLEND_DST_ALPHA_EXT                0x80CA
        BLEND_SRC_ALPHA_EXT                0x80CB
```

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

```
    None
```

## Additions to Chapter 3 of the 1.2 GL Specification (Rasterization)

```
    None
```

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

The RGB and alpha blend factors are separate. The function BlendFuncSeparateEXT allows the specification of the four factors. Table 4.1 and Table 4.2 are modified as follows:

| Value | RGB Factors | Alpha Factors |
|---|---|---|
| ZERO | (0, 0, 0) | 0 |
| ONE | (1, 1, 1) | 1 |
| DST_COLOR | (Rd/Kr, Gd/Kg, Bd/Kb) | Ad/Ka |
| ONE_MINUS_DST_COLOR | (1-Rd/Kr, 1-Gd/Kg, 1-Bd/Kb) | 1-Ad/Ka |
| SRC_ALPHA | (As/Ka, As/Ka, As/Ka) | As/Ka |
| ONE_MINUS_SRC_ALPHA | (1-As/Ka, 1-As/Ka, 1-As/Ka) | 1-As/Ka |
| DST_ALPHA | (Ad/Ka, Ad/Ka, Ad/Ka) | Ad/Ka |
| ONE_MINUS_DST_ALPHA | (1-Ad/Ka, 1-Ad/Ka, 1-Ad/Ka) | 1-Ad/Ka |
| CONSTANT_COLOR | (Rc, Gc, Bc) | Ac |
| ONE_MINUS_CONSTANT_COLOR | (1-Rc, 1-Gc, 1-Bc) | 1-Ac |
| CONSTANT_ALPHA | (Ac, Ac, Ac) | Ac |
| ONE_MINUS_CONSTANT_ALPHA | (1-Ac, 1-Ac, 1-Ac) | 1-Ac |
| SRC_ALPHA_SATURATE | (f, f, f) | 1 |

| Value | RGB Factors | Alpha Factors |
|---|---|---|
| ZERO | (0, 0, 0) | 0 |
| ONE | (1, 1, 1) | 1 |
| SRC_COLOR | (Rs/Kr, Gs/Kg, Bs/Kb) | As/Ka |
| ONE_MINUS_SRC_COLOR | (1-Rs/Kr, 1-Gs/Kg, 1-Bs/Kb) | 1-As/Ka |
| SRC_ALPHA | (As/Ka, As/Ka, As/Ka) | As/Ka |
| ONE_MINUS_SRC_ALPHA | (1-As/Ka, 1-As/Ka, 1-As/Ka) | 1-As/Ka |
| DST_ALPHA | (Ad/Ka, Ad/Ka, Ad/Ka) | Ad/Ka |
| ONE_MINUS_DST_ALPHA | (1-Ad/Ka, 1-Ad/Ka, 1-Ad/Ka) | 1-Ad/Ka |
| CONSTANT_COLOR | (Rc, Gc, Bc) | Ac |
| ONE_MINUS_CONSTANT_COLOR | (1-Rc, 1-Gc, 1-Bc) | 1-Ac |
| CONSTANT_ALPHA | (Ac, Ac, Ac) | Ac |
| ONE_MINUS_CONSTANT_ALPHA | (1-Ac, 1-Ac, 1-Ac) | 1-Ac |
| SRC_ALPHA_SATURATE | (f, f, f) | 1 |

The commands that control blending are

```
void BlendFunc(enum src, enum dst)
void BlendFuncSeparateEXT(enum sfactorRGB, enum dfactorRGB,
                          enum sfactorAlpha, enum dfactorAlpha);
```

The BlendFunc command sets both source factors (RGB and alpha) and destination factors (RGB and alpha) while BlendFuncSeparateEXT sets the RGB factors independently from the alpha factors.

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

None

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

The state required is four integers indicating the source and destination blending functions for RGB and alpha. The initial state for both source functions is ONE. The initial state for both destination functions is ZERO.

## 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:

**BlendFuncSeparateEXT**

| | | |
|---|---|---|
| 2 | 20 | rendering command length |
| 2 | 4134 | rendering command opcode |
| 4 | ENUM | sfactorRGB |
| 4 | ENUM | dfactorRGB |
| 4 | ENUM | sfactorAlpha |
| 4 | ENUM | dfactorAlpha |

## Errors

GL_INVALID_ENUM is generated if either sfactorRGB, dfactorRGB, sfactorAlpha, or dfactorAlpha is not an accepted value.

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

## New State

The get values BLEND_SRC and BLEND_DST return the RGB source and destination factor, respectively.

| Get Value | Get Command | Type | Initial Value | Attribute |
|---|---|---|---|---|
| BLEND_SRC_RGB_EXT | GetFloatv | Z | ONE | color-buffer |
| BLEND_DST_RGB_EXT | GetFloatv | Z | ZERO | color-buffer |
| BLEND_SRC_ALPHA_EXT | GetFloatv | Z | ONE | color-buffer |
| BLEND_DST_ALPHA_EXT | GetFloatv | Z | ZERO | color-buffer |

## New Implementation Dependent State

None

## Name

    EXT_depth_bounds_test

## Name Strings

    GL_EXT_depth_bounds_test

## Notice

## Status

Implemented in GeForce FX 5900 (NV35) drivers as of June 2003.

Also supported by GeForce FX 5700 (NV36) and GeForce6 (NV4x).

## Version

```
Last Modified Date:  $Date: 2004/05/17 $
NVIDIA Revision: $Revision: #5 $
```

## Number

297

## Dependencies

Written based on the wording of the OpenGL 1.3 specification.

## Overview

This extension adds a new per-fragment test that is, logically, after the scissor test and before the alpha test.  The depth bounds test compares the depth value stored at the location given by the incoming fragment's (xw,yw) coordinates to a user-defined minimum and maximum depth value.  If the stored depth value is outside the user-defined range (exclusive), the incoming fragment is discarded.

Unlike the depth test, the depth bounds test has NO dependency on the fragment's window-space depth value.

This functionality is useful in the context of attenuated stenciled shadow volume rendering.  To motivate the functionality's utility in this context, we first describe how conventional scissor testing can be used to optimize shadow volume rendering.

If an attenuated light source's illumination can be bounded to a rectangle in XY window-space, the conventional scissor test can be used to discard shadow volume fragments that are guaranteed to be outside the light source's window-space XY rectangle.  The stencil increments and decrements that would otherwise be generated by these scissored fragments are inconsequential because the light source's illumination can pre-determined to be fully attenuated outside the scissored region.  In other words, the scissor test can be used to discard shadow volume fragments rendered outside the scissor, thereby improving performance, without affecting the ultimate illumination of these pixels with respect to the attenuated light source.

This scissoring optimization can be used both when rendering the stenciled shadow volumes to update stencil (incrementing and decrementing the stencil buffer) AND when adding the illumination contribution of attenuated light source's.

In a similar fashion, we can compute the attenuated light source's window-space Z bounds (zmin,zmax) of consequential illumination. Unless a depth value (in the depth buffer) at a pixel is within the range [zmin,zmax], the light source's illumination can be pre-determined to be inconsequential for the pixel. Said another way, the pixel being illuminated is either far enough in front of or behind the attenuated light source so that the light source's illumination for the pixel is fully attenuated. The depth bounds test can perform this test.

**Issues**

*Where should the depth bounds test take place in the OpenGL fragment processing pipeline?*

RESOLUTION: After scissor test, before alpha test. In practice, this is a logical placement of the test. An implementation is free to perform the test in a manner that is consistent with the specified ordering.

Importantly, the depth bounds test occurs before any fragment operation that has a side-effect such as stencil and/or depth buffer writes (ie, the stencil or depth test). This makes it possible to discard incoming fragment's without concern for preserving such side-effects.

*Is the depth bounds test consistent with early depth rejection?*

Yes. If an OpenGL implementation supports some conservative bounds on depth values in subregions of the depth buffer (hierarchical depth buffers, etc), the depth bounds test can reject fragments based on these conservative bounds.

*How are the depth bounds specified?*

RESOLUTION: Normalized window-space depth values. This means the depth values are specified in the range [0.0, 1.0] similar to glDepthRange.

*Can the zmin bound be greater than the zmax bound?*

RESOLUTION: zmin must be less than or equal to zmax or an INVALID_VALUE error is generated.

Another way to interpret this situation is to have zmin>zmax reject all fragments where the corresponding pixel's depth value is between zmin and zmax. But this does not seem useful enough to specify.

*What should the glDepthBoundsEXT routine mimic?*

RESOLUTION: glDepthBoundsEXT should mimic glDepthRange in parameter types and clamping, excepting that zmin must be less than zmax.

*Do the depth bounds have anything to do with the depth range?*

RESOLUTION: No. These are totally independent pieces of state. To reinforce the point, having a depth range and depth bounds with no overlap is perfectly well-defined (even if a little odd).

*What push/pop attrib bits should affect the depth bounds test enable?*

RESOLUTION: GL_ENABLE_BIT and GL_DEPTH_BUFFER_BIT.

*How does depth bounds testing interact with polygon offset or depth replace operations (say from ARB_fragment_program, NV_texture_shader, or NV_fragment_program)?*

RESOLUTION: There are NO interactions. The depth bounds test has NO dependency on the incoming fragment's depth value so it doesn't matter if there is a polygon offset or depth replace operation.

*Does depth bounds testing affect bitmap/draw/copy pixels operations involving depth component pixels?*

RESOLUTION: Yes, depth bounds testing affects all rasterized primitives (just like all other fragment operations).

*How does depth bounds test interact with multisampling?*

RESOLUTION: The depth bounds test is performed per-sample when multisampling is active, just like the depth test.

*At what precision is the depth bounds test carried out?*

RESOLUTION: For the purposes of the test, the bounds are converted to fixed-point as though they were to be written to the depth buffer, and the comparison uses those quantized bounds.

*Can you have the depth test disabled and still have the depth bounds test enabled?*

RESOLUTION: Yes. The two tests operate independently.

*How does the depth bounds test operate if there is no depth buffer?*

RESOLUTION: It is as if the depth bounds test always passes (analogous to the depth test).

## New Procedures and Functions

```
void DepthBoundsEXT(clampd zmin, clampd zmax);
```

**New Tokens**

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

        DEPTH_BOUNDS_TEST_EXT                       0x8890

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

        DEPTH_BOUNDS_EXT                            0x8891
```

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

None

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

None

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

**-- Figure 4.1  Per-fragment operations**

Add a block for the "depth bounds test" after the scissor and before the alpha test.

**-- Section 4.1.X  Depth Bounds Test (following Section 4.1.2 Scissor Test)**

"The depth bounds test determines whether the depth value (Zpixel) stored at the location given by the incoming fragment's (xw,yw) location lies within the depth bounds range defined by two values. These values are set with

```
        void DepthBoundsEXT(clampd zmin, clampd zmax);
```

Each of zmin and zmax are clamped to lie within [0,1] (being of type clampd).  If zmin <= Zpixel <= zmax, then the depth bounds test passes.  Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using Enable or Disable using the constant DEPTH_BOUNDS_TEST_EXT.  When disabled, it is as if the depth bounds test always passes.  If zmin is greater than zmax, then the error INVALID_VALUE is generated.  The state required consists of two floating-point values and a bit indicating whether the test is enabled or disabled.  In the initial state, zmin and zmax are set to 0.0 and 1.0 respectively; and the depth bounds test is disabled.

If there is no depth buffer, it is as if the depth bounds test always passes."

**-- Section 4.10  Additional Multisample Fragment Operations**

Add depth bounds test to the list of operations affected by multisampling.  Amend the 1st and 2nd sentences in the 2nd paragraph to read:

"If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, the depth bounds test, alpha test, depth test, blending, and dithering operations are performed for each pixel sample, rather than just once for each fragment. Failure of the depth bounds, alpha, stencil, or depth test results in termination of the processing of the sample, rather than discarding of the fragment."

Amend the 1st sentence in the 3nd paragraph to read:

"Depth bounds, stencil, depth, blending, and dithering operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1."

Amend the 3rd sentence in the 4th paragraph to read:

"An implementation may choose to identify a centermost sample, and to perform depth bounds, alpha, stencil, and depth tests on only that sample."

**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 AGL/GLX/WGL Specifications**

None

**GLX Protocol**

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

```
DepthBoundsEXT
    2          12              rendering command length
    2          4229            rendering command opcode
    4          FLOAT32         zmin
    4          FLOAT32         zmax
```

**Errors**

If zmin is greater than zmax, then the error INVALID_VALUE is generated.

**New State**

(table 6.15 "Pixel Operation)

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| DEPTH_BOUNDS_TEST_EXT | B | IsEnabled | False | Depth bounds test enable | 4.1.X | depth-buffer/enable |
| DEPTH_BOUNDS_EXT | 2xR+ | GetFloatv | 0,1 | Depth bounds zmin & zmax | 4.1.X | depth-buffer |

**New Implementation Dependent State**

None

**Revision History**

NVIDIA exposed a functionally and enumerant identical version of this extension under the name NV_depth_bounds_test.  NVIDIA drivers after May 2003 support the EXT_depth_bounds_test name only.

Mesa and NVIDIA agreed to make this an EXT extension in April 2003.

8/27/2003 - GLX protocol specification added.

**Name**

    EXT_stencil_two_side

**Name Strings**

    GL_EXT_stencil_two_side

**Notice**

    Copyright NVIDIA Corporation, 2001-2002.

**Status**

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

**Version**

    Last Modified Date:  $Date: 2003/01/08 $
    $Id: //sw/main/docs/OpenGL/specs/GL_EXT_stencil_two_side.txt#6 $

**Number**

    268

**Dependencies**

    Written based on the OpenGL 1.3 specification.

    NV_packed_depth_stencil affects the definition of this extension.

**Overview**

    This extension provides two-sided stencil testing where the
    stencil-related state (stencil operations, reference value, compare
    mask, and write mask) may be different for front- and back-facing
    polygons.  Two-sided stencil testing may improve the performance
    of stenciled shadow volume and Constructive Solid Geometry (CSG)
    rendering algorithms.

**Issues**

    *Is this sufficient for shadow volume stencil update in a single pass?*

      RESOLUTION:  Yes.

      An application that wishes to increment the stencil value for
      rasterized depth-test passing fragments of front-facing polygons and
      decrement the stencil value for rasterized fragments of depth-test
      passing back-facing polygons in a single pass can use the following
      configuration:

```
          glDepthMask(0);
          glColorMask(0,0,0,0);
          glDisable(GL_CULL_FACE);
          glEnable(GL_STENCIL_TEST);
          glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);

          glActiveStencilFaceEXT(GL_BACK);
          glStencilOp(GL_KEEP,            // stencil test fail
                      GL_KEEP,            // depth test fail
                      GL_DECR_WRAP_EXT);  // depth test pass
          glStencilMask(~0);
          glStencilFunc(GL_ALWAYS, 0, ~0);

          glActiveStencilFaceEXT(GL_FRONT);
          glStencilOp(GL_KEEP,            // stencil test fail
                      GL_KEEP,            // depth test fail
                      GL_INCR_WRAP_EXT);  // depth test pass
          glStencilMask(~0);
          glStencilFunc(GL_ALWAYS, 0, ~0);

          renderShadowVolumePolygons();
```

Notice the use of EXT_stencil_wrap to avoid saturating decrements losing the shadow volume count. An alternative, using the conventional GL_INCR and GL_DECR operations, is to clear the stencil buffer to one half the stencil buffer value range, say 128 for an 8-bit stencil buffer. In the case, a pixel is "in shadow" if the final stencil value is greater than 128 and "out of shadow" if the final stencil value is 128. This does still create a potential for stencil value overflow if the stencil value saturates due to an increment or decrement. However saturation is less likely with two-sided stencil testing than the conventional two-pass approach because front- and back-facing polygons are mixed together, rather than processing batches of front-facing then back-facing polygons.

Contrast the two-sided stencil testing approach with the more or less equivalent approach using facingness-independent stencil testing:

```
  glDepthMask(0);
  glColorMask(0,0,0,0);
  glEnable(GL_CULL_FACE);
  glEnable(GL_STENCIL_TEST);

  glStencilMask(~0);
  glStencilFunc(GL_ALWAYS, 0, ~0);

  // Increment for front faces
  glCullFace(GL_BACK);
  glStencilOp(GL_KEEP,   // stencil test fail
              GL_KEEP,   // depth test fail
              GL_INCR);  // depth test pass

  renderShadowVolumePolygons();

  // Decrement for back faces
  glCullFace(GL_FRONT);
  glStencilOp(GL_KEEP,   // stencil test fail
              GL_KEEP,   // depth test fail
              GL_DECR);  // depth test pass

  renderShadowVolumePolygons();
```

Notice that all the render work implicit in renderShadowVolumePolygons is performed twice with the conventional approach, but only once with the two-sided stencil testing approach.

*Should there be just front and back stencil test state, or should the stencil write mask also have a front and back state?*

RESOLUTION: Both the stencil test and stencil write mask state should have front and back versions.

The shadow volume application for two-sided stencil testing does not require differing front and back versions of the stencil write mask, but we anticipate other applications where front and back write masks may be useful.

For example, it may be useful to draw a convex polyhedra such that (assuming the stencil bufer is cleared to the binary value 1010):

1) front-facing polygons that pass the depth test set stencil bit 0

2) front-facing polygons that fail the depth test zero stencil bit 1

3) back-facing polygons that pass the depth test set stencil bit 2

4) back-facing polygons that fail the depth test zero stencil bit 3

This could be accomplished in a single rendering pass using:

```
  glStencilMask(~0);
  glStencilClear(0xA);
  glClear(GL_STENCIL_BUFFER_BIT);

  glDepthMask(0);
  glColorMask(0,0,0,0);
  glDisable(GL_CULL_FACE);
  glEnable(GL_STENCIL_TEST);
  glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);

  glActiveStencilFaceEXT(GL_BACK);
  glStencilOp(GL_KEEP,      // stencil test fail
              GL_ZERO,      // depth test fail
              GL_REPLACE);  // depth test pass
  glStencilMask(0xC);
  glStencilFunc(GL_ALWAYS, 0x4, ~0);

  glActiveStencilFaceEXT(GL_FRONT);
  glStencilOp(GL_KEEP,      // stencil test fail
              GL_ZERO,      // depth test fail
              GL_REPLACE);  // depth test pass
  glStencilMask(0x3);
  glStencilFunc(GL_ALWAYS, 0x1, ~0);

  renderConvexPolyhedra();
```

*Is there a performance advantage to using two-sided stencil testing?*

RESOLUTION: It depends.

In a fill-rate limited situation, rendering front-facing primitives, then back-facing primitives in two passes will generate the same number of rasterized fragments as rendering front- and back-facing primitives in a single pass.

However, in other situations that are CPU-limited, transform-limited, or setup-limited, two-sided stencil testing can be faster than the conventional two-pass face culling rendering approaches. For example, if a lengthy vertex program is executed for every shadow volume vertex, rendering the shadow volume with a single two-sided stencil testing pass is advantageous.

Often applications using stencil shadow volume techniques require substantial CPU resources to determine potential silhouette boundaries to project shadow volumes from. If the shadow volume geometry generated by the CPU is only required to be sent to the GL once per-frame (rather than twice with the conventional technique), that can ease the CPU burden required to implement stenciled shadow volumes.

*Should GL_FRONT_AND_BACK be accepted by glActiveStencilFaceEXT?*

RESOLUTION: No.

GL_FRONT_AND_BACK is useful when materials are being updated for two-sided lighting because the front and back material are often identical and may change frequently (glMaterial calls are allowed within glBegin/glEnd pairs).

Two-sided stencil has no similiar performance justification.

It is also likely that forcing implementations to support this mode would increase the amount of overhead required to set stencil state, even for applications that don't use two-sided stencil.

*How should the two-sided stencil enable operate?*

RESOLUTION: It should be modeled after the way two-sided lighting works. There is a GL_LIGHTING enable and then an additional two-sided lighting mode. Unlike two-sided lighting which is a light model boolean, the two-sided stencil testing is a standard enable named GL_STENCIL_TEST_TWO_SIDE_EXT.

Here is the pseudo-code for the stencil testing enables:

```
if (glIsEnabled(GL_STENCIL_TEST)) {
  if (glIsEnabled(GL_STENCIL_TEST_TWO_SIDE_EXT) && primitiveType == polygon) {
    use two-sided stencil testing
  } else {
    use conventional stencil testing
  }
} else {
  no stencil testing
}
```

*How should the two-sided stencil interact with glPolygonMode?*

RESOLUTION: Primitive type is determined by the begin mode so GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_QUADS, GL_TRIANGLE_FAN, and GL_POLYGON generate polygon primitives. If the polygon mode is set such that lines or points are rasterized, two-sided stencil testing still operates based on the original polygon facingness if stencil testing and two-sided stencil testing are enabled.

This is consistent with how two-sided lighting and face culling interact with glPolygonMode.

**New Procedures and Functions**

```
void ActiveStencilFaceEXT(enum face);
```

**New Tokens**

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

```
STENCIL_TEST_TWO_SIDE_EXT                0x8910
```

Accepted by the <face> parameter of ActiveStencilFaceEXT:

```
FRONT
BACK
```

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

```
ACTIVE_STENCIL_FACE_EXT                  0x8911
```

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

None

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

None

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

**-- Section 4.1.5 "Stencil test"**

Replace the first paragraph in the section with:

"The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (xw,yw) and a reference value.

The test is enabled or disabled with the Enable and Disable commands, using the symbolic constant STENCIL_TEST. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

Stencil testing may operate in a two-sided mode. Two-sided stencil testing is enabled or disabled with the Enable and Disable commands, using the symbolic constant STENCIL_TEST_TWO_SIDE_EXT. When stencil testing is disabled, the state of two-sided stencil testing does not affect fragment processing.

There are two sets of stencil-related state, the front stencil state set and the back stencil state set. When two-sided stencil testing is enabled, stencil tests and writes use the front set of stencil state when processing fragments rasterized from non-polygon primitives (points, lines, bitmaps, image rectangles) and front-facing polygon primitives while the back set of stencil state is used when processing fragments rasterized from back-facing polygon primitives. For the purposes of two-sided stencil testing, a primitive is still considered a polygon even if the polygon is to be rasterized as points or lines due to the current polygon mode. Whether a polygon is front- or back-facing is determined in the same manner used for two-sided lighting and face culling (see sections 2.13.1 and 3.5.1). When two-sided stencil testing is disabled, the front set of stencil state is always used when stencil testing fragments.

The active stencil face determines whether stencil-related commands update the front or back stencil state. The active stencil face is set with:

```
  void ActiveStencilFace(enum face);
```

where face is either FRONT or BACK. Stencil commands (StencilFunc, StencilOp, and StencilMask) that update the stencil state update the front stencil state if the active stencil face is FRONT and the back stencil state if the active stencil face is BACK. Additionally, queries of stencil state return the front or back stencil state depending on the current active stencil face.

The stencil test state is controlled with

```
   void StencilFunc(enum func, int ref, uint mask);
   void StencilOp(enum sfail, enum dpfail, enum dppass);"
```

Replace the third and second to the last sentence in the last paragraph in section 4.1.5 with:

"In the initial state, stencil testing and two-sided stencil testing are both disabled, the front and back stencil reference values are both zero, the front and back stencil comparison functions are ALWAYS, and the front and back stencil mask are both all ones. Initially, both the three front and the three back stencil operations are KEEP."

**-- Section 4.2.2 "Fine Control of Buffer Updates"**

Replace the last sentence of the third paragraph with:

"The initial state is for both the front and back stencil plane mask to be all ones. The clear operation always uses the front stencil write mask when clearing the stencil buffer."

**-- Section 4.3.1 "Writing to the Stencil Buffer or to the Depth and Stencil Buffers"**

Replace the final sentence in the first paragraph with:

"Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current front stencil mask state (set with StencilMask), and if a depth component is present, if the setting of DepthMask is not FALSE, it is also written to the framebuffer; the setting of DepthTest is ignored."

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

None

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

None

## Additions to the GLX, WGL, and AGL 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:

```
    ActiveStencilFaceEXT
        2           8               rendering command length
        2           4220            rendering command opcode
        4           ENUM            face
```

## Errors

None

## New State

(table 6.15, page 205) amend the following entries:

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| STENCIL_FUNC | 2xZ8 | GetIntegerv | ALWAYS | Stencil function | 4.1.4 | stencil-buffer |
| STENCIL_VALUE_MASK | 2xZ+ | GetIntegerv | 1's | Stencil mask | 4.1.4 | stencil-buffer |
| STENCIL_REF | 2xZ+ | GetIntegerv | 0 | Stencil reference value | 4.1.4 | stencil-buffer |
| STENCIL_FAIL | 2xZ6 | GetIntegerv | KEEP | Stencil fail action | 4.1.4 | stencil-buffer |
| STENCIL_PASS_DEPTH_FAIL | 2xZ6 | GetIntegerv | KEEP | Stencil depth buffer fail action | 4.1.4 | stencil-buffer |
| STENCIL_PASS_DEPTH_PASS | 2xZ6 | GetIntegerv | KEEP | Stencil depth buffer pass action | 4.1.4 | stencil-buffer |

[Type field is amended with "2x" prefix.]

(table 6.15, page 205) add the following entries:

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| STENCIL_TEST_TWO_SIDE_EXT | B | IsEnabled | False | Two-sided stencil test enable | 4.1.4 | stencil-buffer/enable |
| ACTIVE_STENCIL_FACE_EXT | Z2 | GetIntegerv | FRONT | Active stencil face selector | 4.1.4 | stencil-buffer |

(table 6.16, page 205) ammend the following entry:

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| STENCIL_WRITE_MASK | 2xZ+ | GetIntegerv | 1's | Stencil buffer writemask | 4.2.2 | stencil-buffer |

[Type field is amended with "2x" prefix.]

**Revision History**

    None

**Name**

    NV_float_buffer

**Name Strings**

    GL_NV_float_buffer
    WGL_NV_float_buffer

**Notice**

**Status**

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

**Version**

```
    Last Modified:      $Date: 2003/06/16 $
    NVIDIA Revision:    Revision: #16
```

**Number**

    281

**Dependencies**

Written based on the wording of the OpenGL 1.3 specification and the WGL_ARB_pixel_format extension specification.

The following extensions are required:

* NV_fragment_program
* NV_texture_rectangle
* WGL_ARB_pixel_format
* WGL_ARB_render_texture
* WGL_NV_render_texture_rectangle

EXT_paletted_texture trivially affects the definition of this extension.

SGIX_depth_texture trivially affects the definition of this extension.

NV_texture_shader trivially affects the definition of this extension.

NV_half_float trivially affects the definition of this extension.

**Overview**

This extension builds upon NV_fragment_program to provide a framebuffer and texture format that allows fragment programs to read and write unconstrained floating point data.

In unextended OpenGL, most computations dealing with color or depth buffers are typically constrained to operate on values in the range [0,1]. Computational results are also typically clamped to the range [0,1].

Color, texture, and depth buffers themselves also hold values mapped to the range [0,1].

The NV_fragment_program extension provides a general computational model that supports floating-point numbers constrained only by the precision of the underlying data types.  The quantites computed by fragment programs do not necessarily correspond in number or in range to conventional attributes such as RGBA colors or depth values.  Because of the range and precision constraints imposed by conventional fixed-point color buffers, it may be difficult (if not impossible) to use them to implement certain multi-pass algorithms.

To enhance the extended range and precision available through fragment programs, this extension provides floating-point RGBA color buffers that can be used instead of conventional fixed-point RGBA color buffers.  A floating-point RGBA color buffer consists of one to four floating-point components stored in the 16- or 32-bit floating-point formats (fp16 or fp32) defined in the NV_half_float and NV_fragment_program extensions.

When a floating-point color buffer is used, the results of fragment programs, as written to the "x", "y", "z", and "w" components of the o[COLR] or o[COLH] output registers, are written directly to the color buffer without any clamping or modification.  Certain per-fragment operations are bypassed when rendering to floating-point color buffers.

A floating-point color buffer can also be used as a texture map, either by reading back the contents and then using conventional TexImage calls, or by using the buffer directly via the ARB_render_texture extension.

This extension has many uses.  Some possible uses include:

(1) Multi-pass algorithms with arbitrary intermediate results that don't have to be artifically forced into the range [0,1].  In addition, intermediate results can be written without having to worry about out-of-range values.

(2) Deferred shading algorithms where an expensive fragment program is executed only after depth testing is fully complete.  Instead, a simple program is executed, which stores the parameters necessary to produce a final result.  After the entire scene is rendered, a second pass is executed over the entire frame buffer to execute the complex fragment program using the results written to the floating-point color buffer in the first pass.  This will save the cost of applying complex fragment programs to fragments that will not appear in the final image.

(3) Use floating-point texture maps to evaluate functions with arbitrary ranges.  Arbitrary functions with a finite domain can be approximated using a texture map holding sample results and piecewise linear approximation.

There are several significant limitations on the use of floating-point color buffers.  First, floating-point color buffers do not support frame buffer blending.  Second, floating-point texture maps do not support mipmapping or any texture filtering other than NEAREST.  Third, floating-point texture maps must be 2D, and must use the NV_texture_rectangle extension.

**Issues**

*Should the extension create a separate non-RGBA pixel formats or simply extend existing RGBA formats?*

RESOLVED: Extend existing RGBA formats. Since fragment programs generally build on RGBA semantics, it's cleaner to avoid creating a separate "XYZW" mode. There are several special semantics that need to be added: clear color state is now not clamped, and ReadPixels will clamp to [0,1] only if the source data comes from fixed-point color buffers.

Fragment programs can be written that store data completely unrelated to color into a floating-point "RGBA" buffer.

*Can floating-point color buffers be displayed? If so, how?*

RESOLVED: Not in this extension. Floating-point color buffers can be used only as pbuffers. Hardware necessary to display floating-point color buffers would be expensive and consume significant memory bandwidth.

*Is it possible to encode more than four distinct values in a floating-point color buffer?*

RESOLVED: Yes. The NV_fragment_program extension contains pack and unpack instructions (PK2H, PK2US, PK4B, PK4UB, PK4UBG, UP2H, UP2US, UP4B, UP4UB, UP4UBG) that allow fragment programs to encode multiple values into a single 32-bit component. In particular, it is possible to pack two half-precision floats, two normalized unsigned shorts, or four normalized signed or unsigned bytes into a single 32-bit component.

A program can use a pack instruction to pack multiple values into a single 32-bit component and then write the resulting component to a floating-point color buffer with 32-bit components. On a subsequent rendering pass, a program can read back the stored data (using texture mapping) and use the equivalent unpack instruction to restore the original values. The only data lost in this process comes from the loss of precision or clamping in the packing operation, where the original values are converted to data types with lower precision or a smaller data range.

*What happens when rendering to an floating-point color buffer if fragment program mode is disabled? Or when fragment program mode is enabled, but no program is loaded?*

RESOLVED: Fragment programs are required to use floating-point color buffers. An INVALID_OPERATION error is generated by any GL command that generates fragments if FRAGMENT_PROGRAM_NV is disabled. The same behavior already exists for conventional frame buffers if FRAGMENT_PROGRAM_NV is enabled but the bound fragment program is invalid.

*Should alpha test be supported with floating-point color buffers?*

RESOLVED: No. It is trivial to implement an alpha test in a fragment program using the KIL instruction, which requires no dedicated frame buffer logic.

*Should blending be supported with floating-point color buffers?*

RESOLVED: Not in this extension. While blending would clearly be useful, full-precision floating-point blenders are expensive. In addition, a computational model more general than traditional blending (with its 1-x operations and clamping) is desirable. The traditional OpenGL blending model would not be the most suitable computational model for future blend-enabled floating-point color buffers.

An alternative to conventional blending (operating at a coarser granularity) is to (1) render a pass into the color buffer, (2) bind the color buffer as a texture rectangle using this extension and ARB_render_texture, (3) perform texture lookups in a fragment program using the TEX instruction with f[WPOS].xy as a 2D texture coordinate, and (4) perform the necessary blending between the passes using the same fragment program.

*Should we provide accumulation buffers for pixel formats with floating-point color buffers?*

RESOLVED: No. Accumulation operations contents can be achieved using fragment programs to perform the accumulation, which requires no dedicated frame buffer logic.

*Should fragment program color results be converted to match the format of the frame buffer, or should an error result? For example, what if we write to o[COLR] but have a 16-bit frame buffer?*

RESOLVED: Conversions can be performed simply in hardware, so no error semantics are required. This mechanism also allows the same programs to be shared between contexts with different pixel formats.

Applications should be aware that if color components contain packed data, a data type mismatch may result in a floating-point data conversion that corrupts the packed data.

*How should floating-point color buffers interact with multisampling? For normal color buffers, the multiple samples for each pixel are required to be filtered down to a single pixel in the color buffer. Similar filtering on floating-point color buffers does not necessarily make sense. Should there even be a normal color buffer in this case?*

RESOLVED: The initial implementation of this extension does not provide floating-point color buffers that support multisampling.

Multisample fragment operations (e.g., SAMPLE_COVERAGE) are explicitly not supported by extension. This extension does not modify the portion of the spec where multiple samples are resolved to a single color value. So if floating-point color buffers were provided, the multiple samples are filtered down to a single result value, most likely by computing a per-component average value.

*Conventional RGBA primitive antialiasing multiplies coverage by the alpha component of the fragment's color, with the assumption that alpha blending will be performed. How does antialiasing work with floating-point color buffers?*

RESOLVED: It doesn't. The computed coverage is not accessible to fragment programs and is discarded. Note also that conventional antialiasing requires alpha blending, which does not work for floating-point color buffers.

*What are the semantics for ReadPixels when using an floating-point color buffer?*

RESOLVED: ReadPixels from a floating-point color buffer works like any other RGBA read, except that the final results are not clamped to the range [0,1]. This ensures that we can save and restore floating-point color buffers using ReadPixels/DrawPixels.

*What are the semantics for Bitmap when using an floating-point color buffer?*

RESOLVED: Bitmap generates fragments using the current raster attributes, which are then passed to fragment programs like any other fragments. Bitmaps will be drawn using the color of the current raster position, whose components are clamped to [0,1] when the raster position is sent.

*What are the semantics for DrawPixels when using a floating-point color buffer? How about CopyPixels?*

RESOLVED: DrawPixels generates fragments with the originally specified color values; components are not clamped to [0,1]. For fixed-point color buffers, DrawPixels will generate fragments with clamped color components.

CopyPixels is defined in the spec as a ReadPixels followed by a DrawPixels, and will operate similarly.

This mechanism allows applications to write floating-point data directly into a floating-point color buffer without any clamping. Since DrawPixels and CopyPixels generate fragments and fragment programs are required to render to floating-point color buffers, a fragment program is still required to load a floating-point color buffer using DrawPixels.

*What are the semantics for Clear when using an floating-point color buffer?*

RESOLVED: Clears work as normal, except that values outside the range [0,1] can be written to the color buffer. The core spec is modified so that clear color values are not clamped to [0,1]. Instead, for fixed-point color buffers, clear colors are clamped to [0,1] at clear time.

For compatibility with conventional OpenGL, queries of CLEAR_COLOR_VALUE will clamp components to [0,1]. A separate

FLOAT_CLEAR_COLOR_VALUE_NV query is added to query unclamped color clear values.

*Why don't floating-point textures support filtering? What can be done to achieve texture filtering?*

RESOLVED: Extended OpenGL texture filtering (including mipmapping and support for anisotropic filters) is very computationally expensive. Even simple linear filtering for floating-point textures with large components is expensive.

Linear filters can be implemented in fragment programs by doing multiple lookups into the same texture. Since fragment programs allow the use of arbitrary coordinates into arbitrary texture maps, this type of operation can be easily done.

A 1D linear filter can be implemented using an nx1 texture rectangle with the following (untested) fragment program, assuming the 1D coordinate is in f[TEX0].x:

```
ADDR H2.xy, f[TEX0].x, {0.0, 1.0};
FRCH H3.x, R1.x;             # compute the blend factor
TEX  H0, H2.x, TEX0, RECT;   # lookup 1st sample
TEX  H1, H2.y, TEX0, RECT;   # lookup 2nd sample
LRPH H0, H3.x, H1, H0;       # blend
```

A 2D linear filter can be implemented similarly, assuming the 2D coordinate is in f[TEX0].xy:

```
ADDH H2, f[TEX0].xyxy, {0.0, 0.0, 1.0, 1.0};
FRCH H3.xy, H2.xyxy;         # base weights
ADDH H3.zw, 1.0, -H3.xyxy;   # 1-base weights
MULH H3, H3.xzxz, H3.yyww;   # bilinear filter weights
TEX H1, R2.xyxy, TEX0, RECT; # lookup 1st sample
MULH H0, H1, H3.x;           # blend
TEX H1, R2.zyzy, TEX0, RECT; # lookup 2nd sample
MADH H0, H1, H3.y, H0;       # blend
TEX H0, R2.xwxw, TEX0, RECT; # lookup 3rd sample
MADH H0, H1, H3.z, H0;       # blend
TEX H1, R2.zwzw, TEX0, RECT; # lookup 4th sample
MADH H0, H1, H3.w, H0;       # blend
```

Fragment programs can be used to perform more-or-less arbitrary filtering using similar methods, and the DDX and DDY instructions can be used to refine the shape of the filter.

*Why must the NV_texture_rectangle extension be used in order to use floating-point texture maps?*

RESOLVED: On many graphics hardware platforms, texture maps are stored using a special memory encodings designed to optimize rendering performance. In current hardware, conventional texture maps usually top out at 32 bits per texel. The logic required to encode and decode 128-bit texels (and frame buffer pixels) optimally is substantially more complex.

*What happens if you try to use an floating-point texture without a fragment program?*

RESOLVED: No error is generated, but that texture is effectively disabled. This is similar to the behavior if an application tried to use a normal texture having an inconsistent set of mipmaps.

*How does NV_float_buffer interact with the OpenGL 1.2 imaging subset?*

RESOLVED: The imaging subset as specified should work properly with floating-point color buffers, but is not modified by this extension. There are imaging operations (e.g., color tables, histograms) that expect the components they operate on to be in the range [0,1], and this extension makes no attempt to extend such functionality.

*How does NV_float_buffer interact with SGIS_generate_mipmap?*

RESOLVED: Since this extension supports only texture rectangles (which have no mipmaps), this issue is moot.

In the general case, mipmaps should be generated using an appropriate downsample filter, where floating-point component values are averaged. Components should not be clamped during any such mipmap generation.

*What is the deal with the names of the clear color query tokens?*

RESOLVED: The "normal" OpenGL clear color (clamped to [0,1]) is queried using the token COLOR_CLEAR_VALUE. This extension provides a new query for unclamped values, using the token FLOAT_CLEAR_COLOR_VALUE_NV. Notice that "CLEAR" and "COLOR" are reversed due to a mistake made when the spec was first written. This spec lists the core query token, and originally had "CLEAR" and "COLOR" reversed there, too.

Then again, the core specification is inconsistent since the queried state is set by calling glClearColor(), with "Clear" before "Color".

*What performance issues exist with this functionality?*

See the "NV3x Implementation Issues" section of the specification.

*How should the texture border color (values) be handled for float textures?*

RESOLVED: Clamp the texture border color (values) to [0,1] when sampling a float texture's border. In core OpenGL 1.0, the texture border color components are clamped to the range [01,]. The NV_texture_shader extension added support for signed texture components. We decided to provide GL_TEXTURE_BORDER_VALUES as a way of specifying a version of the texture border color whose components were not clamped to [0,1] when set. This was to provide a way of specifying negative texture border components.

In practice, that has not proven particularly useful. No real applications are known to have specified negative texture border values components.

```
        Ideally, the unclamped GL_TEXTURE_BORDER_VALUES state could
        provide an unclamped (unmassaged) set of floating-point color
        components for the texture border color.  This requires an
        additional 96 bits of state per texture unit to support this,
        and based on the experience with NV_texture_shader's support for
        texture border values outside the [0,1] range, it is simply not
        worth it.

        For compatibility with the NV_texture_shader extension, we
        provide language saying that floating-point textures clamp
        the components of the TEXTURE_BORDER_VALUES vector [0,1] when
        sampling the border color.
```

**New Procedures and Functions**

```
    None.
```

**New Tokens**

```
    Accepted by the <internalformat> parameter of TexImage2D and
    CopyTexImage2D:

        FLOAT_R_NV                                      0x8880
        FLOAT_RG_NV                                     0x8881
        FLOAT_RGB_NV                                    0x8882
        FLOAT_RGBA_NV                                   0x8883
        FLOAT_R16_NV                                    0x8884
        FLOAT_R32_NV                                    0x8885
        FLOAT_RG16_NV                                   0x8886
        FLOAT_RG32_NV                                   0x8887
        FLOAT_RGB16_NV                                  0x8888
        FLOAT_RGB32_NV                                  0x8889
        FLOAT_RGBA16_NV                                 0x888A
        FLOAT_RGBA32_NV                                 0x888B

    Accepted by the <pname> parameter of GetTexLevelParameterfv and
    GetTexLevelParameteriv:

        TEXTURE_FLOAT_COMPONENTS_NV                     0x888C

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

        FLOAT_CLEAR_COLOR_VALUE_NV                      0x888D
        FLOAT_RGBA_MODE_NV                              0x888E

    Accepted in the <piAttributes> array of wglGetPixelFormatAttribivARB and
    wglGetPixelFormatAttribfvARB and in the <piAttribIList> and
    <pfAttribFList> arrays of wglChoosePixelFormatARB:

        WGL_FLOAT_COMPONENTS_NV                         0x20B0
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV        0x20B1
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV       0x20B2
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV      0x20B3
        WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV     0x20B4
```

Accepted in the <piAttribIList> array of wglCreatePbufferARB and returned in the <value> parameter of wglQueryPbufferARB when <iAttribute> is WGL_TEXTURE_FORMAT_ARB:

```
    WGL_TEXTURE_FLOAT_R_NV                          0x20B5
    WGL_TEXTURE_FLOAT_RG_NV                         0x20B6
    WGL_TEXTURE_FLOAT_RGB_NV                        0x20B7
    WGL_TEXTURE_FLOAT_RGBA_NV                       0x20B8
```

**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 3.6.4, Rasterization of Pixel Rectangles (p. 91)**

(modify first paragraph of "Final Conversion", p. 102) ... For RGBA components, the final conversion depends on the format of the color buffer. If the components of the color buffer are fixed-point, each element is clamped to [0,1] and converted to fixed-point according to the rules given in section 2.13.9 (Final Color Processing). If the components of the color buffer are floating-point, the elements are not modified.

**Modify Section 3.8.1, Texture Image Specification (p. 116)**

(modify last paragaph, p. 116) The selected groups are processed exactly as for DrawPixels stopping just before final conversion. For textures with fixed-point RGBA internal formats, each R, G, B, A component is clamped to [0,1].

(modify first paragraph, p. 117) Components are then selected from the resulting pixel groups to obtain a texture with the base internal format specified by (or derived from) <internalformat>. Table 3.15 summarizes the mapping of pixel group values to texture components, ...

(add to end of first paragraph, p. 117) Specifying a value of <format> incompatible with <internalformat> produces the error INVALID_OPERATION. A pixel format and texture internal format are compatible if the pixel format can generate a pixel group of the type listed in the "Pixel Group Type" column of Table 3.15 in the row corresponding to the base internal format.

(add between first and second paragraphs, p.117) Textures with a base internal format of FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, and FLOAT_RGBA_NV are known as floating-point textures. Floating-point textures are only supported for the TEXTURE_RECTANGLE_NV target. Specifying an floating-point texture with any other target will produce an INVALID_OPERATION error.

(modify last paragraph, p. 117) The internal component resolution is the number of bits allocated to each component in a texture image. If internalformat is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the memory allocation per texture component is assigned by the GL to match the allocations listed in Table 3.16 as closely as possible. ...

(modify Table 3.15, p. 118 -- Respecify this table with all extensions relevant to texture formats supported by NVIDIA. For this extension, add four base internal formats.)

| | Base Internal Format | Pixel Group Type | Component Values | Internal Components |
|---|---|---|---|---|
| | ALPHA | RGBA | A | A |
| | LUMINANCE | RGBA | R | L |
| | LUMINANCE_ALPHA | RGBA | R,A | L,A |
| | INTENSITY | RGBA | R | I |
| | RGB | RGBA | R,G,B | R,G,B |
| | RGBA | RGBA | R,G,B,A | R,G,B,A |
| * | COLOR_INDEX | CI | CI | CI |
| * | DEPTH_COMPONENT | DEPTH | DEPTH | DEPTH |
| * | HILO_NV | HILO | HI,LO | HI,LO |
| * | DSDT_NV | TEXOFF | DS,DT | DS,DT |
| * | DSDT_MAG_NV | TEXOFF | DS,DT,MAG | DS,DT,MAG |
| * | DSDT_MAG_INTENSITY_NV | TEXOFF or RGBA | DS,DT,MAG,VIB | DS,DT,MAG,I |
| | FLOAT_R_NV | RGBA | R | R (float) |
| | FLOAT_RG_NV | RGBA | R,G | R,G (float) |
| | FLOAT_RGB_NV | RGBA | R,G,B | R,G,B (float) |
| | FLOAT_RGBA_NV | RGBA | R,G,B,A | R,G,B,A (float) |

Table 3.15: Conversion from pixel groups to internal texture components. "Pixel Group Type" defines the type of pixel group required for the specified internal format. All internal components are stored as unsigned-fixed point numbers, except for DS/DT (signed fixed-point numbers) and floating-point R,G,B,A (signed floating-point numbers). See Section 3.8.12 for a description of texture components R, G, B, A, L, and I. See NV_texture_shader spec (Section 3.8.13) for a description of texture components HI, LO, DS, DT, and MAG.

* - indicates formats found in other extension specs: COLOR_INDEX in EXT_paletted texture; DEPTH_COMPONENT in SGIX_depth_texture; and HILO_NV, DSDT_NV, DSDT_MAG_NV, DSDT_MAG_INTENSITY_NV in NV_texture_shader.

(modify Table 3.16, p. 119 -- Respecify this table with all extensions relevant to sized texture internal formats supported by NVIDIA. For this extension, add eight sized internal formats.)

| Sized Int. Format | Base Int. Format | Component Name / Type-Size | | | |
|---|---|---|---|---|---|
| ALPHA4 | ALPHA | A/U4 | | | |
| ALPHA8 | ALPHA | A/U8 | | | |
| ALPHA12 | ALPHA | A/U12 | | | |
| ALPHA16 | ALPHA | A/U16 | | | |
| LUMINANCE4 | LUMINANCE | L/U4 | | | |
| LUMINANCE8 | LUMINANCE | L/U8 | | | |
| LUMINANCE12 | LUMINANCE | L/U12 | | | |
| LUMINANCE16 | LUMINANCE | L/U16 | | | |
| LUMINANCE4_ALPHA4 | LUMINANCE_ALPHA | A/U4 | L/U4 | | |
| LUMINANCE6_ALPHA2 | LUMINANCE_ALPHA | A/U2 | L/U6 | | |
| LUMINANCE8_ALPHA8 | LUMINANCE_ALPHA | A/U8 | L/U8 | | |
| LUMINANCE12_ALPHA4 | LUMINANCE_ALPHA | A/U4 | L/U12 | | |
| LUMINANCE12_ALPHA12 | LUMINANCE_ALPHA | A/U12 | L/U12 | | |
| LUMINANCE16_ALPHA16 | LUMINANCE_ALPHA | A/U16 | L/U16 | | |
| INTENSITY4 | INTENSITY | I/U4 | | | |
| INTENSITY8 | INTENSITY | I/U8 | | | |
| INTENSITY12 | INTENSITY | I/U12 | | | |
| INTENSITY16 | INTENSITY | I/U16 | | | |
| R3_G3_B2 | RGB | R/U3 | G/U3 | B/U2 | |
| RGB4 | RGB | R/U4 | G/U4 | B/U4 | |
| RGB5 | RGB | R/U5 | G/U5 | B/U5 | |
| RGB8 | RGB | R/U8 | G/U8 | B/U8 | |
| RGB10 | RGB | R/U10 | G/U10 | B/10 | |
| RGB12 | RGB | R/U12 | G/U12 | B/U12 | |
| RGB16 | RGB | R/U16 | G/U16 | B/U16 | |
| RGBA2 | RGBA | R/U2 | G/U2 | B/U2 | A/U2 |
| RGBA4 | RGBA | R/U4 | G/U4 | B/U4 | A/U4 |
| RGB5_A1 | RGBA | R/U5 | G/U5 | B/U5 | A/U1 |
| RGBA8 | RGBA | R/U8 | G/U8 | B/U8 | A/U8 |
| RGB10_A2 | RGBA | R/U10 | G/U10 | B/U10 | A/U2 |
| RGBA12 | RGBA | R/U12 | G/U12 | B/U12 | A/U12 |
| RGBA16 | RGBA | R/U16 | G/U16 | B/U16 | A/U16 |
| * COLOR_INDEX1_EXT | COLOR_INDEX | CI/U1 | | | |
| * COLOR_INDEX2_EXT | COLOR_INDEX | CI/U2 | | | |
| * COLOR_INDEX4_EXT | COLOR_INDEX | CI/U4 | | | |
| * COLOR_INDEX8_EXT | COLOR_INDEX | CI/U8 | | | |
| * COLOR_INDEX16_EXT | COLOR_INDEX | CI/U16 | | | |
| * DEPTH_COMPONENT16_SGIX | DEPTH_COMPONENT | Z/U16 | | | |
| * DEPTH_COMPONENT24_SGIX | DEPTH_COMPONENT | Z/U24 | | | |
| * DEPTH_COMPONENT32_SGIX | DEPTH_COMPONENT | Z/U32 | | | |
| * HILO16_NV | HILO | HI/U16 | LO/U16 | | |
| * SIGNED_HILO16_NV | HILO | HI/S16 | LO/S16 | | |
| * SIGNED_RGBA8_NV | RGBA | R/S8 | G/S8 | B/S8 | A/S8 |
| * SIGNED_RGB8_UNSIGNED_ALPHA8_NV | RGBA | R/S8 | G/S8 | B/S8 | A/U8 |
| * SIGNED_RGB8_NV | RGB | R/S8 | G/S8 | B/S8 | |
| * SIGNED_LUMINANCE8_NV | LUMINANCE | L/S8 | | | |
| * SIGNED_LUMINANCE8_ALPHA8_NV | LUMINANCE_ALPHA | L/S8 | A/S8 | | |
| * SIGNED_ALPHA8_NV | ALPHA | A/S8 | | | |
| * SIGNED_INTENSITY8_NV | INTENSITY | I/S8 | | | |
| * DSDT8_NV | DSDT_NV | DS/S8 | DT/S8 | | |
| * DSDT8_MAG8_NV | DSDT_MAG_NV | DS/S8 | DT/S8 | MAG/U8 | |
| * DSDT8_MAG8_INTENSITY8_NV | DSDT_MAG_INTENSITY_NV | DS/S8 | DT/S8 | MAG/U8 | I/U8 |
| FLOAT_R16_NV | FLOAT_R_NV | R/F16 | | | |
| FLOAT_R32_NV | FLOAT_R_NV | R/F32 | | | |
| FLOAT_RG16_NV | FLOAT_RG_NV | R/F16 | G/F16 | | |
| FLOAT_RG32_NV | FLOAT_RG_NV | R/F32 | G/F32 | | |
| FLOAT_RGB16_NV | FLOAT_RGB_NV | R/F16 | G/F16 | B/F16 | |
| FLOAT_RGB32_NV | FLOAT_RGB_NV | R/F32 | G/F32 | B/F32 | |
| FLOAT_RGBA16_NV | FLOAT_RGBA_NV | R/F16 | G/F16 | B/F16 | A/F16 |
| FLOAT_RGBA32_NV | FLOAT_RGBA_NV | R/F32 | G/F32 | B/F32 | A/F32 |

Table 3.16: Sized Internal Formats. Describes the correspondence of sized internal formats to base internal formats, and desired component resolutions. Component resolution descriptions are of the form "<NAME>/<TYPE><SIZE>", where NAME specifies the component name in Table 3.15, TYPE is "U" for unsigned fixed-point, "S" for signed fixed-point, and "F" for unsigned floating-point. <SIZE> is the number of requested bits per component.

* - indicates formats found in other extension specs: COLOR_INDEX in EXT_paletted texture; DEPTH_COMPONENT in SGIX_depth_texture; and HILO_NV, DSDT_NV, DSDT_MAG_NV, DSDT_MAG_INTENSITY_NV in NV_texture_shader.

**Modify Section 3.8,7, Minification (p. 141)**

Change the last paragraph (as modified by the NV_texture_shader extension) to read (only the last sentence changes from the NV_texture_shader version):

"If any of the selected tauijk, tauij, or taui in the above equations refer to a border texel with i < -bs, j < bs, k < -bs, i >= ws-bs, j >= hs-bs, or k >= ds-bs, then the border values given by the current setting of TEXTURE_BORDER_VALUES is used instead of the unspecified value or values. If the texture contains color components, the components of the TEXTURE_BORDER_VALUES vector are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. If the texture contains HILO components, the first and second components of the TEXTURE_BORDER_VALUES vector are interpreted as the hi and lo components respectively. If the texture contains texture offset group components, the first, second, third, and fourth components of the TEXTURE_BORDER_VALUES vector are interpreted as ds, dt, mag, and vib components respectively. Additionally, the texture border values are clamped appropriately depending on the signedness of each particular component. Unsigned components and components of floating-point textures are clamped to [0,1]; signed components (not including floating-point textures) are clamped to [-1,1]."

(Add after the last paragraph in the section) Floating-point textures (those with a base internal format of FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV) do not support texture filters other than NEAREST. For such textures, NEAREST filtering is applied regardless of the setting of TEXTURE_MIN_FILTER.

**Modify Section 3.8.8, Magnification (p. 141)**

(Add after the last paragraph in the section) Floating-point textures (those with a base internal format of FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV) do not support texture filters other than NEAREST. For such textures, NEAREST filtering is applied regardless of the setting of TEXTURE_MAG_FILTER.

**Modify Section 3.8.13, Texture Environments and Texture Functions (p. 147)**

(Add paragraph after discussion of all the values used in the miscellaneous tables in this section.) If the base internal format is HILO_NV, DSDT_NV, DSDT_MAG_NV, DSDT_MAG_INTENSITY_NV, FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV, the texture lookup results are not supported using conventional OpenGL texture functions. In this case, the corresponding texture function is NONE (Cv = Cf, Av = Af), and it is as though texture mapping were disabled for that texture unit.

**Modify Section 3.11, Antialiasing Application (p. 155)**

Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value may be applied to the fragment. In RGBA mode with fixed-point frame buffers, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value. In RGBA mode with floating-point frame buffers, the coverage value is simply discarded. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.2.

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

**Modify Chapter 4 Introduction (p. 156)**

(replace next-to-last paragraph)

The GL provides three types of color buffers: color index, fixed-point RGBA, or floating-point RGBA. Color index buffers consist of unsigned integer color indices. Fixed-point RGBA buffers consist of R, G, B, and optionally, A unsigned integer values. Floating-point RGBA buffers consist of R, and optionally, G, B, and A floating-point component values, corresponding to the X, Y, Z, and W outputs, respectively, of a fragment program. The number of bitplanes in each of the color buffers, the depth buffer, ...

**Modify Section 4.1.3, Multisample Fragment Operations (p. 158)**

This step applies only for fixed-point RGBA color buffers. Otherwise, proceed to the next step. ...

**Modify Section 4.1.4, Alpha Test (p. 159)**

This step applies only for fixed-point RGBA color buffers. Otherwise, proceed to the next step. ...

**Modify Section 4.1.7, Blending (p. 161)**

(modify second paragraph)

This blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only for fixed-point RGBA color buffers; otherwise, it is bypassed. ...

**Modify Section 4.1.8, Dithering (p. 165)**

Dithering selects between two color values or indices. Dithering does not apply to floating-point RGBA color buffers. ...

**Modify Section 4.1.9, Logical Operation (p. 165)**

Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the frame buffer. Logical operations do not apply to floating-point color buffers. ...

**Modify Section 4.2.3, Clearing the Buffers (p. 171)**

...

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

sets the clear value for RGBA color buffers. When a fixed-point color buffer is cleared, the effective clear color is derived by clamping each component to [0,1] and converting to fixed-point according to the rules in section 2.13.9. When a floating-point color buffer is cleared, the components of the clear value are used directly without being clamped.

**Modify Section 4.2.4, The Accumulation Buffer (p. 172)**

(modify last paragraph) ... If there is no accumulation buffer, or if color buffer is not fixed-point RGBA, Accum generates the error INVALID_OPERATION.

**Modify Section 4.3.2, Reading Pixels**

(modify "Conversion of RGBA Values", p. 176) 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. If the color buffer has fixed-point format, 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).

(add to end of "Final Conversion", p. 177) ... For an RGBA color, components are clamped depending on the data type of the buffer being read. For fixed-point buffers, each component is clamped to [0.1]. For floating-point buffers, if <type> is not FLOAT or HALF_FLOAT_NV, each component is clamped to [0,1] if <type> is unsigned or [-1,1] if <type> is signed and then converted according to Table 4.7.

## 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)

**Modify Section 6.1.4, Texture Queries (p. 200)**

Modify Table 6.1 (add new rows, corresponding to new internal formats, p. 202)

| Base Internal Format | R | G | B | A |
|---|---|---|---|---|
| FLOAT_R_NV | R | 0 | 0 | 1 |
| FLOAT_RG_NV | R | G | 0 | 1 |
| FLOAT_RGB_NV | R | G | B | 1 |
| FLOAT_RGBA_NV | R | G | B | A |

## Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None.

## Additions to the WGL Specification

First, close your eyes and pretend that a WGL specification actually existed. Maybe if we all concentrate hard enough, one will magically appear.

Modify/add to the description of <piAttributes> in
wglGetPixelFormatAttribivARB and <pfAttributes> in
wglGetPixelFormatAttribfvARB:

```
  WGL_FLOAT_COMPONENTS_NV
    True if the R, G, B, and A components of each color buffer are
    represented as (unclamped) floating-point numbers.

  WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV
  WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV
  WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV
  WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV
    True if the pixel format describes a floating-point color that can be
    bound to a texture rectangle with internal formats of FLOAT_R_NV,
    FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV, respectively.  Currently
    only pbuffers can be bound as textures so this attribute will only be
    TRUE if WGL_DRAW_TO_PBUFFER is also TRUE.  Additionally,
    floating-point color buffers can not be bound to texture targets other
    than TEXTURE_RECTANGLE_NV.
```

Add new table entries for pixel format attribute matching in
wglChoosePixelFormatARB.

| Attribute | Type | Match Criteria |
|---|---|---|
| WGL_FLOAT_COMPONENTS_NV | boolean | exact |
| WGL_BIND_TO_TEXTURE_ RECTANGLE_FLOAT_R_NV | boolean | exact |
| WGL_BIND_TO_TEXTURE_ RECTANGLE_FLOAT_RG_NV | boolean | exact |
| WGL_BIND_TO_TEXTURE_ RECTANGLE_FLOAT_RGB_NV | boolean | exact |
| WGL_BIND_TO_TEXTURE_ RECTANGLE_FLOAT_RGBA_NV | boolean | exact |

(In the wglCreatePbufferARB section, modify the attribute list)

```
  WGL_TEXTURE_FORMAT_ARB

    This attribute indicates the base internal format of the texture that
    will be created when a color buffer of a pbuffer is bound to a texture
    map.  It can be set to WGL_TEXTURE_RGB_ARB (indicating an internal
    format of RGB), WGL_TEXTURE_RGBA_ARB (indicating a base internal
    format of RGBA), WGL_TEXTURE_FLOAT_R_NV (indicating a base internal
    format of FLOAT_R_NV), WGL_TEXTURE_FLOAT_RG_NV (indicating a base
    internal format of FLOAT_RG_NV), WGL_TEXTURE_FLOAT_RGB_NV (indicating
    a base internal format of FLOAT_RGB_NV), WGL_TEXTURE_FLOAT_RGBA_NV
    (indicating a base internal format of FLOAT_RGBA_NV), or
    WGL_NO_TEXTURE_ARB. The default value is WGL_NO_TEXTURE_ARB.
```

(In the wglCreatePbufferARB section, modify the discussion of what happens
to the depth/stencil/accum buffers when switching between mipmap levels or
cube map faces.)

For pbuffers with a texture format of WGL_TEXTURE_RGB_ARB,
WGL_TEXTURE_RGBA_ARB, WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_FLOAT_RG_NV,

```
    WGL_TEXTURE_FLOAT_RGB_NV, or WGL_TEXTURE_FLOAT_RGBA_NV, there will be a
    separate set of color buffers for each mipmap level and cube map face in
    the pbuffer.  Otherwise, the WGL implementation is free to share a single
    set of color, auxillary, and accumulation buffers between levels or faces.


    (In the wglCreatePbufferARB section, modify the error list)

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_R_NV,
                                WGL_TEXTURE_FLOAT_RG_NV,
                                WGL_TEXTURE_FLOAT_RGB_NV, or
                                WGL_TEXTURE_FLOAT_RGBA_NV, and
                                WGL_TEXTURE_TARGET_ARB is not
                                WGL_TEXTURE_RECTANGLE_NV.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_R_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV
                                attribute is not set in the pixel format.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_RG_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV
                                attribute is not set in the pixel format.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_RGB_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV
                                attribute is not set in the pixel format.

        ERROR_INVALID_DATA      WGL_TEXTURE_FORMAT_ARB is
                                WGL_TEXTURE_FLOAT_RGBA_NV,
                                WGL_TEXTURE_TARGET_ARB is
                                WGL_TEXTURE_RECTANGLE_NV, and the
                                WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV
                                attribute is not set in the pixel format.

    Modify wglBindTexImageARB:

    ...

        The pbuffer attribute WGL_TEXTURE_FORMAT_ARB determines the base
        internal format of the texture. The format-specific component sizes
        are also determined by pbuffer attributes as shown in the table below.
        The component sizes are dependent on the format of the texture.

        Component           Size                    Format
        ---------    -----------------------    ----------------------------
            R        WGL_RED_BITS_ARB           RGB, RGBA, FLOAT_R, FLOAT_RG,
                                                FLOAT_RGB, FLOAT_RGBA
```

|  |  |  |
|---|---|---|
| G | WGL_GREEN_BITS_ARB | RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA |
| B | WGL_BLUE_BITS_ARB | RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA |
| A | WGL_ALPHA_BITS_ARB | RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA |

**Additions to the AGL/GLX Specification**

None.

**Dependencies on EXT_paletted_texture, SGIX_depth_texture, and NV_texture_shader**

If any of these extensions are not supported, the rows in Tables 3.15 and 3.16 corresponding to texture formats defined by the unsupported extension should be removed.

If NV_texture_shader is not supported, ignore the amended paragraph from the NV_texture_shader specificiaton describing TEXTURE_BORDER_VALUES clamping in favor of the original OpenGL specification language.

**Dependencies on NV_half_float**

If GL_NV_half_float is not supported, all references to HALF_FLOAT_NV should be deleted.

**GLX Protocol**

None.

**Errors**

INVALID_OPERATION is generated by Begin, DrawPixels, Bitmap, CopyPixels, or a command that performs an explicit Begin if the color buffer has a floating-point RGBA format and FRAGMENT_PROGRAM_NV is disabled.

INVALID_OPERATION is generated by TexImage3D, TexImage2D, TexImage1D, TexSubImage3D, TexSubImage2D, or TexSubImage1D if the pixel group type corresponding to <format> is not compatible with the base internal format of the texture.

INVALID_OPERATION is generated by TexImage3D, TexImage1D, or CopyTexImage1D if the base internal format corresponding to <internalformat> is FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV.

INVALID_OPERATION is generated by TexImage2D or CopyTexImage2D if the base internal format corresponding to <internalformat> is FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, or FLOAT_RGBA_NV and <target> is not TEXTURE_RECTANGLE_NV.

INVALID_OPERATION is generated by Accum if the color buffer has a color index or floating-point RGBA format.

ERROR_INVALID_DATA is generated by wglCreatePbufferARB if WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_FLOAT_RG_NV, WGL_TEXTURE_FLOAT_RGB_NV, or WGL_TEXTURE_FLOAT_RGBA_NV, and WGL_TEXTURE_TARGET_ARB is not WGL_TEXTURE_RECTANGLE_NV.

ERROR_INVALID_DATA is generated by wglCreatePbufferARB if WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_TARGET_ARB is WGL_TEXTURE_RECTANGLE_NV, and the WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV attribute is not set in the pixel format.

ERROR_INVALID_DATA is generated by wglCreatePbufferARB if WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_RG_NV, WGL_TEXTURE_TARGET_ARB is WGL_TEXTURE_RECTANGLE_NV, and the WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV attribute is not set in the pixel format.

ERROR_INVALID_DATA is generated by wglCreatePbufferARB if WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_RGB_NV, WGL_TEXTURE_TARGET_ARB is WGL_TEXTURE_RECTANGLE_NV, and the WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV attribute is not set in the pixel format.

ERROR_INVALID_DATA is generated by wglCreatePbufferARB if WGL_TEXTURE_FORMAT_ARB is WGL_TEXTURE_FLOAT_RGBA_NV, WGL_TEXTURE_TARGET_ARB is WGL_TEXTURE_RECTANGLE_NV, and the WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV attribute is not set in the pixel format.

## New State

(Modify Table 6.15, Texture Objects (cont.), p. 223)

| Get Value | Type | Get Command | Init. Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| TEXTURE_FLOAT_COMPONENTS_NV | n x B | GetTexLevel- | 0 | True if texture holds unclamped floating-point values | 3.8 | - |

(Modify Table 6.19, Framebuffer Control, p. 227)

| Get Value | Type | Get Command | Init. Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| COLOR_CLEAR_VALUE | C | GetFloatv | 0,0,0,0 | Color buffer clear value (RGBA mode), each value clamped to [0,1]. | 4.2.3 | color-buffer |
| FLOAT_CLEAR_COLOR_VALUE_NV | 4xR | GetFloatv | 0,0,0,0 | Color buffer clear value (RGBA mode), each value unclamped. | 4.2.3 | color-buffer |

**New Implementation Dependent State**

(Modify Table 6.28, Implementation Dependent Values, p. 236)

| Get Value | Type | Get Command | Init. Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| FLOAT_RGBA_MODE_NV | B | GetBooleanv | - | True if color buffers store floating-point data | 4 | - |

**NV3x Implementation Details**

NV3x GPUs (GeForce FX, etc.) support hardware acceleration for float textures with two or more components only when the repeat mode state (S and T) is GL_CLAMP_TO_EDGE. If you use either the GL_CLAMP or GL_CLAMP_TO_BORDER repeat modes with a float texture with two or more components, the software rasterizer is used.

However, if you use a single-component float texture (GL_FLOAT_R_NV, etc.), all clamping repeat modes (GL_CLAMP, GL_CLAMP_TO_EDGE, and GL_CLAMP_TO_BORDER) are available with full hardware acceleration.

The two-, three-, and four-component texture formats all use the same amount of texture memory storage (128 bits per texel for the GL_FLOAT_x32 formats, and 64 bits per texel for the GL_FLOAT_x16 formats). Future GPUs will likely store two and three component float textures more efficiently.

The GL_FLOAT_R32_NV and GL_FLOAT_R16_NV texture formats each use 32 bits per texel. Future GPUs will likely store GL_FLOAT_R16_NV more efficiently.

NVIDIA treats the unsized internal formats GL_FLOAT_R_NV, GL_FLOAT_RGBA_NV, etc. the same as GL_FLOAT_R32_NV, GL_FLOAT_RGBA32_NV, etc.

**Revision History**

| Rev. | Date | Author | Changes |
|---|---|---|---|
| 16 | 06/16/03 | pbrown | Corrected the usage of WGL_TEXTURE_FLOAT_R_NV and related enums in the list of enumerants. |
| 15 | 01/23/03 | mjk | Document texture border color (values) behavior for float textures. See issue. |
| 14 | 01/20/03 | mjk | Added NV3x Implementation Details section. |

**Name**

    NV_fragment_program

**Name Strings**

    GL_NV_fragment_program

**Notice**

    Copyright NVIDIA Corporation, 2001-2002.

**IP Status**

    NVIDIA Proprietary.

**Status**

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

**Version**

    Last Modified Date:  $Date: 2003/05/12 $
    NVIDIA Revision:     70

**Number**

    282

**Dependencies**

    Written based on the wording of the OpenGL 1.2.1 specification and
    requires OpenGL 1.2.1.

    Requires support for the ARB_multitexture extension with at least
    two texture units.

    NV_vertex_program affects the definition of this extension.  The only
    dependency is that both extensions use the same mechanisms for defining
    and binding programs.

    NV_texture_shader trivially affects the definition of this extension.

    NV_texture_rectangle trivially affects the definition of this extension.

    ARB_texture_cube_map trivially affects the definition of this extension.

    EXT_fog_coord trivially affects the definition of this extension.

    NV_depth_clamp affects the definition of this extension.

    ARB_depth_texture and SGIX_depth_texture affect the definition of this
    extension.

NV_float_buffer affects the definition of this extension.

ARB_vertex_program affects the definition of this extension.

ARB_fragment_program affects the definition of this extension.

**Overview**

OpenGL mandates a certain set of configurable per-fragment computations defining texture lookup, texture environment, color sum, and fog operations. Each of these areas provide a useful but limited set of fixed operations. For example, unextended OpenGL 1.2.1 provides only four texture environment modes, color sum, and three fog modes. Many OpenGL extensions have either improved existing functionality or introduced new configurable fragment operations. While these extensions have enabled new and interesting rendering effects, the set of effects is limited by the set of special modes introduced by the extension. This lack of flexibility is in contrast to the high-level of programmability of general-purpose CPUs and other (frequently software-based) shading languages. The purpose of this extension is to expose to the OpenGL application writer an unprecedented degree of programmability in the computation of final fragment colors and depth values.

This extension provides a mechanism for defining fragment program instruction sequences for application-defined fragment programs. When in fragment program mode, a program is executed each time a fragment is produced by rasterization. The inputs for the program are the attributes (position, colors, texture coordinates) associated with the fragment and a set of constant registers. A fragment program can perform mathematical computations and texture lookups using arbitrary texture coordinates. The results of a fragment program are new color and depth values for the fragment.

This extension defines a programming model including a 4-component vector instruction set, 16- and 32-bit floating-point data types, and a relatively large set of temporary registers. The programming model also includes a condition code vector which can be used to mask register writes at run-time or kill fragments altogether. The syntax, program instructions, and general semantics are similar to those in the NV_vertex_program and NV_vertex_program2 extensions, which provide for the execution of an arbitrary program each time the GL receives a vertex.

The fragment program execution environment is designed for efficient hardware implementation and to support a wide variety of programs. By design, the entire set of existing fragment programs defined by existing OpenGL per-fragment computation extensions can be implemented using the extension's programming model.

The fragment program execution environment accesses textures via arbitrarily computed texture coordinates. As such, there is no necessary correspondence between the texture coordinates and texture maps previously lumped into a single "texture unit". This extension separates the notion of "texture coordinate sets" and "texture image units" (texture maps and associated parameters), allowing implementations with a different number of each. The initial implementation of this extension will support 8 texture coordinate sets and 16 texture image units.

**Issues**

*What limitations exist in this extension?*

RESOLVED: Very few. Programs can not exceed a maximum program length (which is no less than 1024 instructions), and can use no more than 32-64 temporary registers. Programs can not access more than one fragment attribute or program parameter (constant) per instruction, but can work around this restriction using temporaries. The number of textures that can be used by a program is limited to the number of texture image units provided by the implementation (16 in the initial implementation of this extension).

These limits are fairly high. Additionally, there is no limit on the total number of texture lookups that can be performed by a program. There is no limit on the length of a texture dependency chain -- one can write a program that performs over 1000 consecutive dependent texture lookups. There is no restrictions on dependencies between texture mapping instructions and arithmetic instructions. Texture lookups can be performed using arbitrarily computed texture coordinates. Applications can carry out their calculations with full 32-bit single precision, although two lower-precision modes are also available.

*How does texture mapping work with fragment programs?*

RESOLVED: This extension provides three instructions used to perform texture lookups.

The "TEX" instruction performs a lookup with the (s,t,r) values taken from an interpolated texture coordinate, an arbitrarily computed vector, or even a program constant. The "TXP" instruction performs a similar lookup, except that it uses the fourth component of the source vector to performs a perspective divide, using (s/q, t/q, r/q). In both cases, the GL will automatically compute partial derivatives used for filter and LOD selection.

The "TXD" instruction operates like "TEX", except that it allows the program to explicitly specify two additional vectors containing the partial derivatives of the texture coordinate with respect to x and y window coordinates.

All three instructions write a filtered texel value to a temporary or output register. Other than the computation of texture coordinates and partial derivatives, texture lookups not performed any differently in fragment program mode. In particular, any applicable LOD biases, wrap modes, minification and magnification filters, and anisotropic filtering controls are still applied in fragment program mode.

The results of the texture lookup are available to be used arbitrarily by subsequent fragment program instructions. Fragment programs are allowed to access any texture map arbitrarily many times.

*Can fragment programs be used to compute depth values?*

RESOLVED: Yes. A fragment program can perform arbitrary computations to compute a final value for the fragment, which it should write to the "z" component of the o[DEPR] register. The "z" value written should be in the range [0,1], regardless of the size of the depth buffer.

To assist in the computation of the final Z value, a fragment program can access the interpolated depth of the fragment (prior to any displacement) by reading the "z" component of the f[WPOS] attribute register.

*How should near and far plane clipping work in fragment program mode if the current fragment program computes a depth value?*

RESOLVED: Geometric clipping to the near and far clip plane should be disabled. Clipping should be done based on the depth values computed per-fragment. The rationale is that per-fragment depth displacement operations may effectively move portions of a primitive initially outside the clip volume inside, and vice versa.

Note that under the NV_depth_clamp extension, geometric clipping to the near and far clip planes is also disabled, and the fragment depth values are clamped to the depth range. If depth clamp mode is enabled when using a fragment program that computes a depth value, the computed depth value will be clamped to the depth range.

*Should fragment programs be allowed to use multiple precisions for operands and operations?*

RESOLVED: Yes. Low-precision operands are generally adequate for representing colors. Allowing low-precision registers also allows for a larger number of temporary registers (at lower precision). Low-precision operations also provide the opportunity for a higher level of performance.

Applications are free to use only high-precision operations or mix high- and low-precision operations as necessary.

*What levels of precision are supported in arithmetic operations?*

RESOLVED: Arithmetic operations can be performed at three different precisions. 32-bit floating point precision (fp32) uses the IEEE single-precision standard with a sign bit, 8 exponent bits, and 23 mantissa bits. 16-bit floating-point precision (fp16) uses a similar floating-point representation, but with 5 exponent bits and 10 mantissa bits. Additionally, many arithmetic operations can also be carried out at 12-bit fixed point precision (fx12), where values in the range [-2,+2) are represented as signed values with 10 fraction bits.

*How should the precision with which operations are carried out be specified? Should we infer the precision from the types of the operands or result vectors? Or should it be an attribute of the instruction?*

RESOLVED: Applications can optionally specify the precision of individual instructions by adding a suffix of "R", "H", and "X" to instruction names to select fp32, fp16, and fx12 precision, respectively.

By default, instructions will be carried out using the precision of the destination register. Always inferring the precision from the operands has a number of issues. First, there are a number of operations (e.g., TEX/TXP/TXD) where result type has little to no correspondance to the type of the operands. In these cases, precision suffixes are not supported. Second, one could have instructions automatically cast operands and compute results using the type of the highest precision operand or result. This behavior would be problematic since all fragment attribute registers and program parameters are kept at full precision, but full precision may not be needed by the operation.

The choice of precision level allows programs to trade off precision for potentially higher performance. Giving the program explicit control over the precision also allows it to dictate precision explicitly and eliminate any uncertainty over type casting.

*For instructions whose specified precision is different than the precision of the operands or the result registers, how are the operations performed? How are the condition codes updated?*

RESOLVED: Operations are performed with operands and results at the precision specified by the instruction. After the operation is complete, the result is converted to the precision of the destination register, after which the condition code is generated.

In an alternate approach, the condition code could be generated from the result. However, in some cases, the register contents would not match the condition code. In such cases, it may not be reliable to use the condition code to prevent division by zero or other special cases.

*How does this extension interact with the ARB_multisample extension? In the ARB_multisample extension, each fragment has multiple depth values. In this extension, a single interpolated depth value may be modified by a fragment program.*

RESOLVED: The depth values for the extra samples are generated by computing partials of the computed depth value and using these partials to derive the depth values for each of the extra samples.

*How does this extension interact with polygon offset? Both extensions modify fragment depth values.*

RESOLVED: As in the base OpenGL spec, the depth offset generated by polygon offset is added during polygon rasterization. The depth value provided to programs in f[WPOS].z already includes polygon offset, if enabled. If the depth value is replaced by a fragment program, the polygon offset value will NOT be recomputed and added back after program execution.

This is probably not desirable for fragment programs that modify depth values since the partials used to generate the offset may not match the partials of the computed depth value. Polygon offset for filled polygons can be approximated in a fragment program using the depth partials obtained by the DDX and DDY instructions. This will not work properly for line- and point-mode polygons, since the partials used for offset are computed over the polygon, while the partials resulting from the DDX and DDY instructions are computed along the line (or are zero for point-mode polygons). In addition, separate treatment of points, line segments, and polygons is not possible in a fragment program.

*Should depth component replacement be an property of the fragment program or a separate enable?*

RESOLVED: It should be a program property. Using the output register notation simplifies matters: depth components are replaced if and only if the DEPR register is written to. This alleviates the application and driver burden of maintaining separate state.

*How does this extension affect the handling of q texture coordinates in the OpenGL spec?*

RESOLVED: Fragment programs are allowed to access an associated q texture coordinate, so this attribute must be produced by rasterization. In unextended OpenGL 1.2, the q coordinate is eliminated in the rasterization portions of the spec after dividing each of s, t, and r by it. This extension updates the specification to pass q coordinates through at least to conventional texture mapping. When fragment program mode are disabled, q coordinates will be eliminated there in an identical manner. This modification has the added benefit of simplifying the equations used for attribute interpolation.

*How should clip w coordinates be handled by this extension?*

RESOLVED: Fragment programs are allowed to access the reciprocal of the clip w coordinate, so this attribute must be produced by rasterization. The OpenGL 1.2 spec doesn't explictly enumerate the attributes associated with the fragment, but we add treatment of the w clip coordinate in the appropriate locations.

The reciprocal of the clip w coordinate in traditional graphics hardware is produced by screen-space linear interpolation of the reciprocals of the clip w coordinates of the vertices. However, this spec says the clip w coordinate is produced by perspective-correct interpolation of the (non-reciprocated) clip w vertex coordinates. These two formulations turn out to be equivalent, and the latter is more convenient since the core OpenGL spec already contains formulas for perspective-correct interpolation of vertex attributes.

*What is produced by the TEX/TXP/TXD instructions if the requested texture image is inconsistent?*

RESOLVED: The result vector is specified to be (0,0,0,0). This behavior is consistent with the NV_texture_shader extension. Note that like in NV_texture_shader, these instructions ignore the standard hierarchy of texture enables and programs can access textures that are not specifically "enabled".

*Should a minimum precision be specified for certain fragment attribute registers (in particular COL0, COL1) that may not be generated with full fp32 precision?*

RESOLVED: No. It is expected that the precision of COL0/COL1 should generally be at least as high as that of the frame buffer.

*Fragment color components (f[COL0] and f[COL1]) are generally low-precision fixed-point values in the range [0,1]. Is it possible to pass unclamped or high-precision color components to fragment programs?*

RESOLVED: Yes, although you can't exactly call them "colors". High-precision per-vertex color values can be written into any unused texture coordinate set, either via a MultiTexCoord call or using a vertex program. These "texture coordinates" will be interpolated during rasterization, and can be used arbitrarily by a fragment program.

In particular, there is no requirement that per-fragment attributes called "texture coordinates" be used for texture mapping.

*Should this specification guarantee that temporary registers are initialized to zero?*

RESOLVED: Yes. This will allow for the modular construction of programs that accumulate results in registers. For example, per-fragment lighting may use MAD instructions to accumulate color contributions at each light. Without zero-initialization, the program would require an explicit MOV instruction to load 0 or the use of the MUL instruction for the first light.

*Should this specification support Unicode program strings?*

RESOLVED: Not necessary.

*Programs defined by NV_vertex_program begin with "!!VP1.0". Should fragment programs have a similar identifier?*

RESOLVED: Yes, "!!FP1.0", identifying the first revision of this fragment program language.

*Should per-fragment attributes have equivalent integer names in the program language, as per-vertex attributes do in NV_vertex_program?*

RESOLVED: No. In NV_vertex_program, "generic" vertex attributes could be specified directly by an application using only an attribute number. Those numbers may have no necessary correlation with the conventional attribute names, although conventional vertex attributes are mapped to attribute numbers. However, conventional attributes are the only outputs of vertex programs and of rasterization. Therefore, there is no need for a similar input-by-number functionality for fragment programs.

*Should we provide the ability to issue instructions that do not update temporary or output registers?*

RESOLVED: Yes. Programs may issue instructions whose only purpose is to update the condition code register, and requiring such instructions to write to a temporary may require the use of an additional temporary and/or defeat possible program optimizations. We accomplish this by adding two write-only temporary pseudo-registers ("RC" and "HC") that can be specified as destination registers.

*Do the packing and unpacking instructions in this extension make any sense?*

RESOLVED: Yes. They are useful for packing and unpacking multiple components in a single channel of a floating-point frame buffer. For example, a 128-bit "RGBA" frame buffer could pack 16 8-bit quantities or 8 16-bit quantities, all of which could be used in later rasterization passes. See the NV_float_buffer extension for more information.

*Should we provide a method for specifying an fp16 depth component output value?*

RESOLVED: No. There is no good reason for supporting half-precision Z outputs. Even with 16-bit Z buffers, the 10-bit mantissa of the half-precision float is rather limiting. There would effectively be only 11 good bits in the back half of the Z buffer.

*Should RequestResidentProgramsNV (or a new equivalent function) take a target? Dealing with working sets of different program types is a bit messy. Should we document some limitation if we get programs of different types?*

RESOLVED: In retrospect, it may have been a good idea to attach a target to this command, but there isn't a good reason to mess with something that already works for vertex programs. The driver is responsible for ensuring consistent results when the program types specified are mixed.

*What happens on data type conversions where the original value is not exactly representable in the new data type, either due to overflow or insufficient precision in the destination type?*

RESOLVED: In case of overflow, the original value is clamped to the +/-INF (fp16 or fp32) or the nearest representable value (fx12). In case of imprecision, the conversion is either to round or truncate to the nearest representable value.

*Should this extension support IEEE-style denorms? For 32-bit IEEE floating point, denorms are numbers smaller in absolute value than 2^-126. For 16-bit floats used by this extension, denorms are numbers smaller in absolute value than 2^-14.*

RESOLVED: For 32-bit data types, hardware support for denorms was considered too expensive relative to the benefit provided. Computational results that would otherwise produce denorms are flushed to zero. For 16-bit data types, hardware denorm support will be present. The expense of hardware denorm support is lower and the potential precision benefit is greater for 16-bit data types.

*OpenGL provides a hierarchy of texture enables. The texture lookup operations in NV_texture_shader effectively override the texture enable hierarchy and select a specific texture to enable. What should be done by this extension?*

RESOLVED: This extension will build upon NV_texture_shader and reduce the driver overhead of validating the texture enables. Texture lookups can be specified by instructions like "TEX H0, f[TEX2], TEX2, 3D", which would indicate to use texture coordinate set number 2 to do a lookup in the texture object bound to the TEXTURE_3D target in texture image unit 2.

Each texture unit can have only one "active" target. Programs are not allowed to reference different texture targets in the same texture image unit. In the example above, any other texture instructions using texture image unit 2 must specify the 3D texture target.

*What is the interaction with NV_register_combiners?*

RESOLVED: Register combiners are not available when fragment programs are enabled.

Previous version of this specification supported the notion of combiner programs, where the result of fragment program execution was a set of four "texture lookup" values that fed the register combiners.

*For convenience, should we include pseudo-instructions not present in the hardware instruction set that are trivially implementable? For example, absolute value and subtract instructions could fall in this category. An "ABS R1,R0" instruction would be equivalent to "MAX R1,R0,-R0", and a "SUB R2,R0,R1" would be equivalent to "ADD R2,R0,-R1"*

RESOLVED: In general, yes. A SUB instruction is provided for convenience. This extension does not provide a separate ABS instruction because it supports absolute value operations of each operand.

*Should there be a '+' in the <optionalSign> portion of the grammar? There isn't one in the GL_NV_vertex_program spec.*

RESOLVED: Yes, for orthogonality/readability. A '+' obviously adds no functionality. In NV_vertex_program, an <optionalSign> of "-" was always a negation operator. However, in fragment programs, it can also be used as a sign for a constant value.

*Can the same fragment attribute register, program parameter register, or constants be used for multiple operands in the same instruction? If so, can it be used with different swizzle patterns?*

RESOLVED: Yes and yes.

*This extension allows different limits for the number of texture coordinate sets and the number of texture image units (i.e., texture maps and associated data). The state in ActiveTextureARB affects both* *coordinate sets (TexGen, matrix operations) and image units (TexParameter, TexEnv). How should we deal with this?*

RESOLVED: Continue to use ActiveTextureARB and emit an INVALID_OPERATION if the active texture refers to an unsupported coordinate set/image unit. Other options included creating dummy (unusable) state for unsupported coordinate sets/image units and continue to use ActiveTextureARB normally, or creating separate state and state-setting commands for coordinate sets and image units. Separate state is the cleanest solution, but would add more calls and potentially cause more programmer confusion. Dummy state would avoid additional error checks, but the demands of dummy state could grow if the number of texture image units and texture coordinate sets increases.

The current OpenGL spec is vague as to what state is affected by the active texture selector and has no distination between coordinate-related and image-related state. The state tables could use a good clean-up in this area.

*The LRP instruction is defined so that the result of "LRP R0, R0, R1, R2" is R0*R1+(1-R0)*R2. There are conflicting precedents here. The definition here matches the "lrp" instruction in the DirectX 8.0 pixel shader language. However, an equivalent RenderMan lerp operation would yield a result of (1-R0)*R1+R0*R2. Which ordering should be implemented?*

RESOLVED: NVIDIA hardware implements the former operand ordering, and there is no good reason to specify a different ordering. To convert a "LRP" using the latter ordering to NV_fragment_program, swap the third and fourth arguments.

*Should this extension provide tracking of matrices or any other state, similar to that provided in NV_vertex_program?*

RESOLVED: No.

*Should this extension provide global program parameters -- values shared between multiple fragment programs?*

RESOLVED: No.

*Should this extension provide program parameters specific to a program? If so, how?*

RESOLVED: Yes. These parameters will be called "local parameters". This extension will provide both named and numbered local parameters. Local parameters can be managed by the driver and eliminate the need for applications to manage a global name space.

Named local parameters work much like standard variable names in most programming languages. They are created using the "DECLARE" instruction within the fragment program itself. For example:

```
DECLARE color = {1,0,0,1};
```

Named local parameters are used simply by referencing the variable name. They do not require the array syntax like the global parameters in the NV_vertex_program extension. They can be updated using the commands ProgramNamedParameter4[f,fv]NV.

Numbered local parameters are not declared. They are used by simply referencing an element of an array called "p". For example,

```
MOV R0, p[12];
```

loads the value of numbered local parameter 12 into register R0. Numbered local parameters can be updated using the commands ProgramLocalParameter4[d,dv,f,fv]ARB.

The numbered local parameter APIs were added to this extension late in its development, and are provided for compatibility with the ARB_vertex_program extension, and what will likely be supported in ARB_fragment_program as well. Providing this mechanism allows programs to use the same mechanisms to set local parameters in both extension.

*Why are the APIs for setting named and numbered local parameters different?*

RESOLVED: The named parameter API was created prior to ARB_vertex_program (and the possible future ARB_fragment_program) and uses conventions borrowed from NV_vertex_program. A slightly different API was chosen during the ARB standardization process; see the ARB_vertex_program specification for more details.

The named parameter API takes a program ID and a parameter name, and sets the parameter for the program with the specified ID. The specified program does not need to be bound (via BindProgramNV) in order to modify the values of its named parameters. The numbered parameter API takes a program target enum (FRAGMENT_PROGRAM_NV) and a parameter number and modifies the corresponding numbered parameter of the currently bound program.

*What should be the initial value of uninitialized local parameters?*

RESOLVED: (0,0,0,0). This choice is somewhat arbitrary, but matches previous extensions (e.g., NV_vertex_program).

*Should this extension support program parameter arrays?*

RESOLVED: No hardware support is present. Note that from the point of view of a fragment program, a texture map can be used as a 1-, 2-, or 3-dimensional array of constants.

*Should this extension provide support constants in fragment programs? If so, how?*

RESOLVED: Yes. Scalar or vector constants can be defined inline (e.g., "1.0" or "{1,2,3,4}"). In addition, named constants are supported using the "DEFINE" instruction, which allow programmers to change the values of constants used in multiple instructions simply be changing the value assigned to the named constant.

Note that because this extension uses program strings, the floating-point value of any constants generated on the fly must be printed to the program string. An alternate method that avoids the need to print constants is to declare a named local program parameter and initialize it with the ProgramNamedParameter4[f,fv]() calls.

*Should named constants be allowed to be redefined?*

RESOLVED: No. If you want to redefine the values of constants, you can create an equivalent named program parameter by changing the "DEFINE" keyword to "DECLARE".

*Should functions used to update or query named local parameters take a zero-terminated string (as with most strings in the C programming language), or should they require an explicit string length? If the former, should we create a version of LoadProgramNV that does not require a string length.*

RESOLVED: Stick with explicit string length. Strings that are defined as constants can have the length computed at compile-time. Strings read from files will have the length known in advance. Programs to build strings at run-time also likely keep the length up-to-date. Passing an explicit length saves time, since the driver doesn't have to do a strlen().

*What is the deal with the alpha of the secondary color?*

RESOLVED: In unextended OpenGL 1.2, the alpha component of the secondary color is forced to 0.0. In the EXT_secondary_color extension, the alpha of the per-vertex secondary colors is defined to be 0.0. NV_vertex_program allows vertex programs to produce a per-vertex alpha component, but it is forced to zero for the purposes of the color sum. In the NV_register_combiners extension, the alpha component of the secondary color is undefined. What a mess.

In this extension, the alpha of the secondary color is well-defined and can be used normally. When in vertex program mode

*Why are fragment program instructions involving f[FOGC] or f[TEX0] through f[TEX7] automatically carried out at full precision?*

RESOLVED: This is an artifact of the method that these interpolants are generated the NVIDIA graphics hardware. If such instructions absolutely must be carried out at lower precision, the requirement can be met by first loading the interpolants into a temporary register.

*With a different number of texture coordinate sets and texture image units, how many copies of each kind of texture state are there?*

RESOLVED: The intention is that texture state be broken into three groups. (1) There are MAX_TEXTURE_COORDS_NV copies of texture coordinate set state, which includes current texture coordinates, TexGen state, and texture matrices. (2) There are MAX_TEXTURE_IMAGE_UNITS_NV copies of texture image unit state, which include texture maps, texture parameters, LOD bias parameters. (3) There are MAX_TEXTURE_UNITS_ARB copies of legacy OpenGL texture unit state (e.g., texture enables, TexEnv blending state), all of which are unused when in fragment program mode.

It is not necessary that MAX_TEXTURE_UNITS_ARB be equal to the minimum of MAX_TEXTURE_COORDS_NV and MAX_TEXTURE_IMAGE_UNITS -- implementations may choose not to extend fixed-function OpenGL texture mapping modes beyond a certain point.

*The GLX protocol for LoadProgramNV (and ProgramNamedParameterNV) may end up with programs >64KB. This will overflow the limits of the GLX Render protocol, resulting in the need to use RenderLarge path. This is an issue with vertex programs, also.*

RESOLVED: Yes, it is.

*Should textures used by fragment programs be declared? For example, "TEXTURE TEX3, 2D", indicating that the 2D texture should be used for all accesses to texture unit 3. The dimension could be dropped from the TEX family of instructions, and some of the compile-time error checking could be dropped.*

RESOLVED: Maybe it should be, but for better or worse, it isn't.

*It is not all that uncommon to have negative q values with projective texture mapping, but results are undefined if any q values are negative in this specification. Why?*

RESOLVED: This restriction carries on a similar one in the initial OpenGL specification. The motivation for this restriction is that when interpolating, it is possible for a fragment to have an interpolated q coordinate at or near 0.0. Since the texture coordinates used for projective texture mapping are s/q, t/q, and r/q, this will result in a divide-by-zero error or suffer from significant numerical instability. Results will be inaccurate for such fragments.

Other than the numerical stability issue above, NVIDIA hardware should have no problems with negative q coordinates.

*Should programs that replace depth have their own special program type, Such as "!!FPD1.0" and "!!FPDC1.0"?*

RESOLVED: No. If a program has an instruction that writes to o[DEPR], the final fragment depth value is taken from o[DEPR].z. Otherwise, the fragment's original depth value is used.

*What fx12 value should NaN map to?*

RESOLVED: For the lack of any better choice, 0.0.

*How are special-case encodings (-INF, +INF, -0.0, +0.0, NaN) handled for arithmetic and comparison operations?*

RESOLVED: The special cases for all floating-point operations are designed to match the IEEE specification for floating-point numbers as closely as possible. The results produced by special cases should be enumerated in the sections of this spec describing the operations. There are some cases where the implemented fragment program behavior does not match IEEE conventions, and these cases should be noted in this specification.

*How can condition codes be used to mask out register writes? How about killing fragments? What other things can you do?*

RESOLVED: The following example computes a component wise |R1-R2|:

```
  SUBC R0, R1, R2;      # "C" suffix means update condition code
  MOV  R0 (LT), -R0;    # Conditional write mask in parentheses
```

The first instruction computes a component-wise difference between R1 and R2, storing R1-R2 in register R0. The "C" suffix in the instruction means to update the condition code based on the sign of the result vector components. The second instruction inverts the sign of the components of R0. However the "(LT)" portion says that the destination register should be updated only if the corresponding condition code component is LT (negative). This means that only those components of R0

To kill a fragment if the red (x) component of a texture lookup returns zero:

```
  TEXC R0, f[TEX0], TEX0, 2D;
  KIL EQ.x;
```

To kill based on the green (y) component, use "EQ.y" instead. To kill if any of the four components is zero, use "EQ.xyzw" or just "EQ".

Fragment programs do not support boolean expressions. These can generally be achieved using conditional write mask.

To evaluate the expression "(R0.x == 0) && (R1.x == 0)":

```
  MOVC RC.x, R0.x;
  MOVC RC.x (EQ), R1.x;
```

To evaluate the expression "(R0.x == 0) || (R1.x == 0)":

```
  MOVC RC.x, R0.x;
  MOVC RC.x (NE), R1.x;
```

In both cases, the x component of the condition code will contain "EQ" if and only if the condition is TRUE.

*How can fragment programs be used to implement non-standard texture filtering modes?*

RESOLVED: As one example, consider a case where you want to do linear filtering in a 2D texture map, but only horizontally. To achieve this, first set the texture filtering mode to NEAREST. For a 16 x n texture, you might do something like:

```
  DEFINE halfTexel = { 0.03125, 0 };   # 1/32 (1/2 a texel)
  ADD R0, f[TEX0], -halfTexel;         # coords of left sample
  ADD R1, f[TEX0], +halfTexel;         # coords of right sample
  TEX R0, R0, TEX0, 2D;                # lookup left sample
  TEX R1, R1, TEX0, 2D;                # lookup right sample
  MUL R2.x, R0.x, 16;                  # scale X coords to texels
```

```
          FRC R2.x, R2.x;                  # get fraction, filter weight
          LRP R0, R2, R1, R0;              # blend samples based on weight
```

There are plenty of other interesting things that can be done.

*Should this specification provide more examples?*

RESOLVED: Yes, it should.

*Is the OpenGL ARB working on a multi-vendor standard for fragment programmability? Will there be an ARB_fragment_program extension? If so, how will this extension interact with the ARB standard?*

RESOLVED: Yes, as of July 2002, there was a multi-vendor working group and a draft specification. The ARB extension is expected to have several features not present in this extension, such as state tracking and global parameters (called "program environment parameters"). It will also likely lack certain features found in this extension.

*Why does the HEMI mapping apply to the third component of signed HILO textures, but not to unsigned HILO textures?*

RESOLVED: This behavior matches the behavior of NV_texture_shader (e.g., the DOT_PRODUCT_NV mode). The HEMI mapping will construct the third component of a unit vector whose first two components are encoded in the HILO texture.

**New Procedures and Functions**

```
    void ProgramNamedParameter4fNV(uint id, sizei len, const ubyte *name,
                                   float x, float y, float z, float w);
    void ProgramNamedParameter4dNV(uint id, sizei len, const ubyte *name,
                                   double x, double y, double z, double w);
    void ProgramNamedParameter4fvNV(uint id, sizei len, const ubyte *name,
                                    const float v[]);
    void ProgramNamedParameter4dvNV(uint id, sizei len, const ubyte *name,
                                    const double v[]);
    void GetProgramNamedParameterfvNV(uint id, sizei len, const ubyte *name,
                                      float *params);
    void GetProgramNamedParameterdvNV(uint id, sizei len, const ubyte *name,
                                      double *params);

    void ProgramLocalParameter4dARB(enum target, uint index,
                                    double x, double y, double z, double w);
    void ProgramLocalParameter4dvARB(enum target, uint index,
                                     const double *params);
    void ProgramLocalParameter4fARB(enum target, uint index,
                                    float x, float y, float z, float w);
    void ProgramLocalParameter4fvARB(enum target, uint index,
                                     const float *params);
    void GetProgramLocalParameterdvARB(enum target, uint index,
                                       double *params);
    void GetProgramLocalParameterfvARB(enum target, uint index,
                                       float *params);
```

**New Tokens**

Accepted by the <cap> parameter of Disable, Enable, and IsEnabled, by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindProgramNV, LoadProgramNV, ProgramLocalParameter4dARB, ProgramLocalParameter4dvARB, ProgramLocalParameter4fARB, ProgramLocalParameter4fvARB, GetProgramLocalParameterdvARB, and GetProgramLocalParameterfvARB:

```
    FRAGMENT_PROGRAM_NV                           0x8870
```

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

```
    MAX_TEXTURE_COORDS_NV                         0x8871
    MAX_TEXTURE_IMAGE_UNITS_NV                    0x8872
    FRAGMENT_PROGRAM_BINDING_NV                   0x8873
    MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV      0x8868
```

Accepted by the <name> parameter of GetString:

```
    PROGRAM_ERROR_STRING_NV                       0x8874
```

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

**Modify Section 2.11, Clipping (p.39)**

(replace the first paragraph of the section, p. 39)  Primitives are clipped to the clip volume.  In clip coordinates, the view volume is defined by

```
    -w_c <= x_c <= w_c,
    -w_c <= y_c <= w_c, and
    -w_c <= z_c <= w_c.
```

Clipping to the near and far clip planes is ignored if fragment program mode (section 3.11) or texture shaders (see NV_texture_shader specification) are enabled, if the current fragment program or texture shader computes per-fragment depth values.  In this case, the view volume is defined by:

```
    -w_c <= x_c <= w_c and
    -w_c <= y_c <= w_c.
```

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

**Modify Chapter 3 introduction (p. 57)**

(p.57, modify 1st paragraph) ... Figure 3.1 diagrams the rasterization process.  The color value assigned to a fragment is initially determined by the rasterization operations (Sections 3.3 through 3.7) and modified by either the execution of the texturing, color sum, and fog operations as defined in Sections 3.8, 3.9, and 3.10, or of a fragment program defined in Section 3.11.  The final depth value is initially determined by the rasterization operations and may be modified by a fragment program.

*note:  Antialiasing Application is renumbered from Section 3.11 to Section*

*3.12.*

**Modify Figure 3.1 (p.58)**

```
                         Primitive Assembly
                                  |
          +-----------+-----------+-----------+-----------+
          |           |           |           |           |
          |           |           |        Pixel          |
        Point       Line       Polygon     Rectangle   Bitmap
       Raster-     Raster-     Raster-     Raster-     Raster-
       ization     ization     ization     ization     ization
          |           |           |           |           |
          +-----------+-----------+-----------+-----------+
                                  |
                                  |
                +-----------------+-----------------+
                |                 |                 |
          Conventional         Texture          Fragment
          Texture Fetch        Shaders          Programs
                |                 |                 |
                |  +--------------+                 |
                |  |                                |
    TEXTURE_    o  o                                |
    SHADER_NV                                       |
    enable      o                                   |
                |                                   |
                +-------------+                     |
                |             |                     |
           Conventional   Register                  |
              TexEnv      Combiners                 |
                |             |                     |
            Color Sum         |                     |
                |             |                     |
               Fog            |                     |
                |             |                     |
                |  +----------+                     |
                |  |                                |
    REGISTER_   o  o                                |
    COMBINERS_                                      |
    NV enable   o                                   |
                |                                   |
                +-----------------+  +--------------+
                                  |  |
                       FRAGMENT_  o  o
                       PROGRAM_
                       NV enable  o
                                  |
                                  |
                               Coverage
                              Application
                                  |
                                  v
                        to fragment processing
```

**Modify Section 3.3, Points (p.61)**

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex cooresponding to the point. (delete reference to divide by q).

If anitialiasing is enabled, then ... The data associated with each fragment are otherwise the data associated with the point being rasterized. (delete reference to divide by q)

**Modify Section 3.4.1, Basic Line Segment Rasterization (p.66)**

(Note that t=0 at p_a and t=1 at p_b). The value of an associated datum f from the fragment, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode), the s, t, r, or q texture coordinate, or the clip w coordinate (the depth value, window z, must be found using equation 3.3, below), is found as

$$f = \frac{(1-t) * f\_a / w\_a + t * f\_b / w\_b}{(1-t) / w\_a + t / w\_b} \tag{3.2}$$

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments respectively. Note that linear interpolation would use

$$f = (1-t) * f\_a + t * f\_b. \tag{3.3}$$

... A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to unacceptable distortion effects when interpolating texture coordinates or clip w coordinates.

**Modify Section 3.5.1, Basic Polygon Rasterization (p.71)**

Denote a datum at p_a, p_b, or p_c ... is given by

$$f = \frac{a * f\_a / w\_a + b * f\_b / w\_b + c * f\_c / w\_c}{a / w\_a + b / w\_b + c / w\_c} \tag{3.4}$$

where w_a, w_b, and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the exact coordinates ... at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

$$f = a * f\_a + b * f\_b + c * f\_c; \tag{3.5}$$

this may yield ... for texture coordinates or clip w coordinates.

**Modify Section 3.6.4, Rasterization of Pixel Rectangles (p.100)**

A fragment arising from a group ... are given by those associated with the current raster position. (delete reference to divide by q)

**Modify Section 3.7, Bitmaps (p.111)**

Otherwise, a rectangular array ... The associated data for each fragment are those associated with the current raster position. (delete reference to divide by q) Once the fragments have been produced ...

**Modify Section 3.8, Texturing (p.112)**

... an image at the location indicated by a fragment's texture coordinates to modify the fragments primary RGBA color. Texturing does not affect the secondary color.

Texturing is specified only for RGBA mode; its use in color index mode is undefined.

Except when in fragment program mode (Section 3.11), the (s,t,r) texture coordinates used for texturing are the values s/q, t/q, and r/q, respectively, where s, t, r, and q are the texture coordinates associated with the fragment. When in fragment program mode, the (s,t,r) texture coordinates are specified by the program. If q is less than or equal to zero, the results of texturing are undefined.

**Add new Section 3.11, Fragment Programs (p.140)**

Fragment program mode is enabled and disabled with the Enable and Disable commands using the symbolic constant FRAGMENT_PROGRAM_NV. When fragment program mode is enabled, standard and extended texturing, color sum, and fog application stages are ignored and a general purpose program is executed instead.

A fragment program is a sequence of instructions that execute on a per-fragment basis. In fragment program mode, the currently bound fragment program is executed as each fragment is generated by the rasterization operations. Fragment programs execute a finite fixed sequence of instructions with no branching or looping, and operate independently from the processing of other fragments. Fragment programs are used to compute new color values to be associated with each fragment, and can optionally compute a new depth value for each fragment as well.

Fragment program mode is not available in color index mode and is considered disabled, regardless of the state of FRAGMENT_PROGRAM_NV. When fragment program mode is enabled, texture shaders and register combiners (NV_texture_shader and NV_register_combiners extension) are disabled, regardless of the state of TEXTURE_SHADER_NV and REGISTER_COMBINERS_NV.

**Section 3.11.1, Fragment Program Registers**

Fragment programs operate on a set of program registers. Each program register is a 4-component vector, whose components are referred to as "x", "y", "z", and "w" respectively. The components of a fragment register are always referred to in this manner, regardless of the meaning of their contents.

The four components of each fragment program register have one of two different representations: 32-bit floating-point (fp32) or 16-bit floating-point (fp16). More details on these representations can be found in Section 3.11.4.1.

There are several different classes of program registers. Attribute registers (Table X.1) correspond to the fragment's associated data produced by rasterization. Temporary registers (Table X.2) hold intermediate results generated by the fragment program. Output registers (Table X.3) hold the final results of a fragment program. The single condition code register is used to mask writes to other registers or to determine if a fragment should be discarded.

**Section 3.11.1.1, Fragment Program Attribute Registers**

The fragment program attribute registers (Table X.1) hold the location of the fragment and the data associated with the fragment produced by rasterization.

| Fragment Attribute Register Name | Description | Component Interpretation |
|---|---|---|
| f[WPOS] | Position of the fragment center. | (x,y,z,1/w) |
| f[COL0] | Interpolated primary color | (r,g,b,a) |
| f[COL1] | Interpolated secondary color | (r,g,b,a) |
| f[FOGC] | Interpolated fog distance/coord | (z,0,0,0) |
| f[TEX0] | Texture coordinate (unit 0) | (s,t,r,q) |
| f[TEX1] | Texture coordinate (unit 1) | (s,t,r,q) |
| f[TEX2] | Texture coordinate (unit 2) | (s,t,r,q) |
| f[TEX3] | Texture coordinate (unit 3) | (s,t,r,q) |
| f[TEX4] | Texture coordinate (unit 4) | (s,t,r,q) |
| f[TEX5] | Texture coordinate (unit 5) | (s,t,r,q) |
| f[TEX6] | Texture coordinate (unit 6) | (s,t,r,q) |
| f[TEX7] | Texture coordinate (unit 7) | (s,t,r,q) |

**Table X.1: Fragment Attribute Registers. The component interpretation column describes the mapping of attribute values to register components. For example, the "x" component of f[COL0] holds the red color component, and the "x" component of f[TEX0] holds the "s" texture coordinate for texture unit 0. The entries "0" and "1" indicate that the attribute register components hold the constants 0 and 1, respectively.**

f[WPOS].x and f[WPOS].y hold the (x,y) window coordinates of the fragment center, and relative to the lower left corner of the window. f[WPOS].z holds the associated z window coordinate, normally in the range [0,1]. f[WPOS].w holds the reciprocal of the associated clip w coordinate.

f[COL0] and f[COL1] hold the associated RGBA primary and secondary colors of the fragment, respectively.

f[FOGC] holds the associated eye distance or fog coordinate normally used for fog computations.

f[TEX0] through f[TEX7] hold the associated texture coordinates for texture coordinate sets 0 through 7, respectively.

All attribute register components are treated as 32-bit floats. However, the components of primary and secondary colors (f[COL0] and f[COL1]) may be generated with reduced precision.

The contents of the fragment attribute registers may not be modified by a fragment program. In addition, each fragment program instruction can use at most one unique attribute register.

**Section 3.11.1.2, Fragment Program Temporary Registers**

The fragment temporary registers (Table X.2) hold intermediate values used during the execution of a fragment program. There are 96 temporary register names, but not all can be used simultaneously.

| Fragment Temporary Register Name | Description |
|---|---|
| R0-R31 | Four 32-bit (fp32) floating point values (s.e8.m23) |
| H0-H63 | Four 16-bit (fp16) floating point values (s.e5.m10) |

**Table X.2: Fragment Temporary Registers.**

In addition to the normal temporary registers, there are two temporary pseudo-registers, "RC" and "HC". RC and HC are treated as unnumbered, write-only temporary registers. The components of RC have an fp32 data type; the components of HC have an fp16 data type. The sole purpose of these registers is to permit instructions to modify the condition code register (section 3.11.1.4) without overwriting the values in any temporary register.

Fragment program instructions can read and write temporary registers. There is no restriction on the number of temporary registers that can be accessed by any given instruction.

All temporary registers are initialized to (0,0,0,0) each time a fragment program executes.

**Section 3.11.1.3, Fragment Program Output Registers**

The fragment program output registers hold the final results of the fragment program. The possible final results of a fragment program are an RGBA fragment color, a fragment depth value, and up to four texture values used by the NV_register_combiners extension.

| Output Register Name | Description |
|---|---|
| o[COLR] | Final RGBA fragment color, fp32 format (color programs) |
| o[COLH] | Final RGBA fragment color, fp16 format (color programs) |
| o[TEX0] | TEXTURE0 output, fp16 format (combiner programs) |
| o[TEX1] | TEXTURE1 output, fp16 format (combiner programs) |
| o[TEX2] | TEXTURE2 output, fp16 format (combiner programs) |
| o[TEX3] | TEXTURE3 output, fp16 format (combiner programs) |
| o[DEPR] | Final fragment depth value, fp32 format |

**Table X.3: Fragment Program Output Registers.**

o[COLR] and o[COLH] are used by color fragment programs to specify the color of a fragment. These two registers are identical, except for the associated data type of the components. The R, G, B, and A components of the fragment color are taken from the x, y, z, and w components respectively of the o[COLR] or o[COLH]. A fragment program will fail to load if it writes to both o[COLR] and o[COLH].

o[TEX0], o[TEX1], o[TEX2], and o[TEX3] are used by combiner fragment programs to generate the initial texture register values for the register combiners. After a combiner fragment program is executed, register combiner operations are performed and can use these computed values. The R, G, B, and A components of the combiner registers are taken from the x, y, z, and w components of the corresponding output registers.

o[DEPR] can be used to replace the associated depth value of a fragment. The new depth value is taken from the z component of o[DEPR]. If a fragment program does not write to o[DEPR], the associated depth value is unmodified.

A fragment program will fail to load if it does not write to at least one output register. A color fragment program will fail to load if it writes to o[TEX0], o[TEX1], o[TEX2], or o[TEX3]. A combiner fragment program will fail to load if it writes to o[COLR] or o[COLH], or if it does not write to any of o[TEX0], o[TEX1], o[TEX2], or o[TEX3].

The fragment program output registers may not be read by a fragment program, but may be written to multiple times.

The values of all fragment program output registers are initially undefined.

**Section 3.11.1.4, Fragment Program Condition Code Register**

The condition code register (CC) is a single four-component vector. 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 fragment data register components or to terminate processing of a fragment altogether (via the KIL instruction).

Most fragment program instructions can optionally update the condition code register. When a fragment program instruction updates the condition code register, a condition code component is set to LT if the corresponding component of the result vector 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 register is initialized to a vector of EQ values each time a fragment program executes.

**Section 3.11.2, Fragment Program Parameters**

In addition to using the registers defined in Section 3.11.1, fragment programs may also use fragment program parameters in their computation. Fragment program parameters are constant during the execution of fragment programs, but some parameters may be modified outside the execution of a fragment program.

There are five different types of program parameters: embedded scalar constants, embedded vector constants, named constants, named local parameters, and numbered local parameters.

Embedded scalar constants are written as standard floating-point numbers with an optional sign designator ("+" or "-") and optional scientific notation (e.g., "E+06", meaning "times 10^6").

Embedded vector constants are written as a comma-separated array of one to four scalar constants, surrounded by braces (like a C/C++ array initializer). Vector constants are always treated as 4-component vectors: constants with fewer than four components are expanded to 4-components by filling missing y and z components with 0.0 and missing w components with 1.0. Thus, the vector constant "{2}" is equivalent to "{2,0,0,1}", "{3,4}" is equivalent to "{3,4,0,1}", and "{5,6,7}" is equivalent to "{5,6,7,1}".

Named constants allow fragment program instructions to define scalar or vector constants that can be referenced by name. Named constants are created using the DEFINE instruction:

```
DEFINE pi = 3.1415926535;
DEFINE color = {0.2, 0.5, 0.8, 1.0};
```

The DEFINE instruction associates a constant name with a scalar or vector constant value. Subsequent fragment program instructions that use the constant name are equivalent to those using the corresponding constant value.

Named local parameters are similar to named vector constants, but their values can be modified after the program is loaded. Local parameters are created using the DECLARE instruction:

```
DECLARE fog_color1;
DECLARE fog_color2 = {0.3, 0.6, 0.9, 0.1};
```

The DECLARE instruction creates a 4-component vector associated with the local parameter name. Subsequent fragment program instructions referencing the local parameter name are processed as though the current value of the local parameter vector were specified instead of the parameter name. A DECLARE instruction can optionally specify an initial value for the local parameter, which can be either a scalar or vector constant. Scalar constants are expanded to 4-component vectors by replicating the scalar value in each component. The initial value of local parameters not initialized by the program is (0,0,0,0).

A named local parameter for a specific program can be updated using the calls ProgramNamedParameter4fNV or ProgramNamedParameter4fvNV (section 5.7). Named local parameters are accessible only by the program in which they are defined. Modifying a local parameter affects the only the associated program and does not affect local parameters with the same name that are found in any other fragment program.

Numbered local parameters are similar to named local parameters, except that they are referred to by number and are not declared in fragment programs. Each fragment program object has an array of four-component floating-point vectors that can be used by the program. The number of vectors is given by the implementation-dependent constant MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV, and must be at least 64. A numbered local parameter is accessed by a fragment program as members of an array called "p". For example, the instruction

```
    MOV R0, p[31];
```

copies the contents of numbered local parameter 31 into temporary register R0.

Constant and local parameter names can be arbitrary strings consisting of letters (upper or lower-case), numbers, underscores ("_"), and dollar signs ("$"). Keywords defined in the grammar (including instruction names) can not be used as constant names, nor can strings that start with numbers, or strings that specify valid temporary register or texture numbers (e.g., "R0"-"R31", "H0"-"H63"", "TEX0"-"TEX15"). A fragment program will fail to load if a DEFINE or DECLARE instruction specifies an invalid constant or local parameter name.

A fragment program will fail to load if an instruction contains a named parameter not specified in a previous DEFINE or DECLARE instruction. A fragment program will also fail to load if a DEFINE or DECLARE instruction attempts to re-define a named parameter specified in a previous DEFINE or DECLARE instruction.

The contents of the fragment program parameters may not be modified by a fragment program. In addition, each fragment program instruction can normally use at most one unique program parameter. The only exception to this rule is if all program parameter references specify named or embedded constants that taken together contain no more than four unique scalar values. For such instructions, the GL will automatically generate an equivalent instruction that references a single merged vector constant. This merging allows programs to specify instructions like the following:

```
    Instruction              Equivalent Instruction
    ---------------------    ---------------------------------------
    MAD R0, R1, 2, -1;       MAD R0, R1, {2,-1,0,0}.x, {2,-1,0,0}.y;
    ADD R0, {1,2,3,4}, 4;    ADD R0, {1,2,3,4}.xyzw, {1,2,3,4}.w;
```

Before counting the number of unique values, any named constants are first converted to the equivalent embedded constants. When generating a combined vector constant, the GL does not perform swizzling, component selection, negation, or absolute value operations. The following instructions are invalid, as they contain more than four unique scalar values.

```
    Invalid Instructions
    -----------------------------------
    ADD R0, {1,2,3,4}, -4;
    ADD R0, {1,2,3,4}, |-4|;
    ADD R0, {1,2,3,4}, -{-1,-2,-3,-4};
    ADD R0, {1,2,3,4}, {4,5,6,7}.x;
```

**Section 3.11.3, Fragment Program Specification**

Fragment programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program. The command LoadProgramNV loads a fragment program when the target parameter is FRAGMENT_PROGRAM_NV. The command BindProgramNV enables a fragment program for execution.

```
    At program load time, the program is parsed into a set of tokens possibly
    separated by white space.  Spaces, tabs, newlines, carriage returns, and
    comments are considered whitespace.  Comments begin with the character "#"
    and are terminated by a newline, a carriage return, or the end of the
    program array.  Fragment programs are case-sensitive -- upper and lower
    case letters are treated differently.  The proper choice of case can be
    inferred from the grammar.

    The Backus-Naur Form (BNF) grammar below specifies the syntactically valid
    sequences for fragment programs.  The set of valid tokens can be inferred
    from the grammar.  The token "" represents an empty string and is used to
    indicate optional rules.  A program is invalid if it contains any
    undefined tokens or characters.

    <program>              ::= <progPrefix> <instructionSequence> "END"

    <progPrefix>           ::= <colorProgPrefix>
                             | <combinerProgPrefix>

    <colorProgPrefix>      ::= "!!FP1.0"

    <combinerProgPrefix>   ::= "!!FCP1.0"

    <instructionSequence>  ::= <instructionSequence> <instructionStatement>
                             | <instructionStatement>

    <instructionStatement> ::= <instruction> ";"
                             | <constantDefinition> ";"
                             | <localDeclaration> ";"

    <instruction>          ::= <VECTORop-instruction>
                             | <SCALARop-instruction>
                             | <BINSCop-instruction>
                             | <BINop-instruction>
                             | <TRIop-instruction>
                             | <KILop-instruction>
                             | <TEXop-instruction>
                             | <TXDop-instruction>

    <VECTORop-instruction> ::= <VECTORop> <maskedDstReg> ","
                               <vectorSrc>
    <VECTORop>              ::= "DDX"    | "DDX_SAT"
                             | "DDXR"   | "DDXR_SAT"
                             | "DDXH"   | "DDXH_SAT"
                             | "DDXC"   | "DDXC_SAT"
                             | "DDXRC"  | "DDXRC_SAT"
                             | "DDXHC"  | "DDXHC_SAT"
                             | "DDY"    | "DDY_SAT"
                             | "DDYR"   | "DDYR_SAT"
                             | "DDYH"   | "DDYH_SAT"
                             | "DDYC"   | "DDYC_SAT"
                             | "DDYRC"  | "DDYRC_SAT"
                             | "DDYHC"  | "DDYHC_SAT"
                             | "FLR"    | "FLR_SAT"
                             | "FLRR"   | "FLRR_SAT"
                             | "FLRH"   | "FLRH_SAT"
                             | "FLRX"   | "FLRX_SAT"
                             | "FLRC"   | "FLRC_SAT"
                             | "FLRRC"  | "FLRRC_SAT"
                             | "FLRHC"  | "FLRHC_SAT"
                             | "FLRXC"  | "FLRXC_SAT"
                             | "FRC"    | "FRC_SAT"
                             | "FRCR"   | "FRCR_SAT"
                             | "FRCH"   | "FRCH_SAT"
                             | "FRCX"   | "FRCX_SAT"
                             | "FRCC"   | "FRCC_SAT"
                             | "FRCRC"  | "FRCRC_SAT"
                             | "FRCHC"  | "FRCHC_SAT"
                             | "FRCXC"  | "FRCXC_SAT"
                             | "LIT"    | "LIT_SAT"
                             | "LITR"   | "LITR_SAT"
                             | "LITH"   | "LITH_SAT"
                             | "LITC"   | "LITC_SAT"
                             | "LITRC"  | "LITRC_SAT"
                             | "LITHC"  | "LITHC_SAT"
                             | "MOV"    | "MOV_SAT"
                             | "MOVR"   | "MOVR_SAT"
                             | "MOVH"   | "MOVH_SAT"
                             | "MOVX"   | "MOVX_SAT"
                             | "MOVC"   | "MOVC_SAT"
                             | "MOVRC"  | "MOVRC_SAT"
                             | "MOVHC"  | "MOVHC_SAT"
                             | "MOVXC"  | "MOVXC_SAT"
                             | "PK2H"
                             | "PK2US"
                             | "PK4B"
                             | "PK4UB"

    <SCALARop-instruction> ::= <SCALARop> <maskedDstReg> ","
                               <scalarSrc>
    <SCALARop>              ::= "COS"     | "COS_SAT"
                             | "COSR"    | "COSR_SAT"
                             | "COSH"    | "COSH_SAT"
                             | "COSC"    | "COSC_SAT"
                             | "COSRC"   | "COSRC_SAT"
                             | "COSHC"   | "COSHC_SAT"
                             | "EX2"     | "EX2_SAT"
                             | "EX2R"    | "EX2R_SAT"
                             | "EX2H"    | "EX2H_SAT"
                             | "EX2C"    | "EX2C_SAT"
                             | "EX2RC"   | "EX2RC_SAT"
                             | "EX2HC"   | "EX2HC_SAT"
                             | "LG2"     | "LG2_SAT"
                             | "LG2R"    | "LG2R_SAT"
                             | "LG2H"    | "LG2H_SAT"
                             | "LG2C"    | "LG2C_SAT"
                             | "LG2RC"   | "LG2RC_SAT"
                             | "LG2HC"   | "LG2HC_SAT"
                             | "RCP"     | "RCP_SAT"
                             | "RCPR"    | "RCPR_SAT"
                             | "RCPH"    | "RCPH_SAT"
                             | "RCPC"    | "RCPC_SAT"
                             | "RCPRC"   | "RCPRC_SAT"
                             | "RCPHC"   | "RCPHC_SAT"
                             | "RSQ"     | "RSQ_SAT"
                             | "RSQR"    | "RSQR_SAT"
                             | "RSQH"    | "RSQH_SAT"
                             | "RSQC"    | "RSQC_SAT"
                             | "RSQRC"   | "RSQRC_SAT"
                             | "RSQHC"   | "RSQHC_SAT"
                             | "SIN"     | "SIN_SAT"
                             | "SINR"    | "SINR_SAT"
                             | "SINH"    | "SINH_SAT"
                             | "SINC"    | "SINC_SAT"
                             | "SINRC"   | "SINRC_SAT"
                             | "SINHC"   | "SINHC_SAT"
                             | "UP2H"    | "UP2H_SAT"
                             | "UP2HC"   | "UP2HC_SAT"
                             | "UP2US"   | "UP2US_SAT"
                             | "UP2USC"  | "UP2USC_SAT"
                             | "UP4B"    | "UP4B_SAT"
                             | "UP4BC"   | "UP4BC_SAT"
                             | "UP4UB"   | "UP4UB_SAT"
                             | "UP4UBC"  | "UP4UBC_SAT"

    <BINSCop-instruction> ::=  <BINSCop> <maskedDstReg> ","
                               <scalarSrc> "," <scalarSrc>

    <BINSCop>               ::= "POW"   | "POW_SAT"
                             | "POWR"  | "POWR_SAT"
                             | "POWH"  | "POWH_SAT"
                             | "POWC"  | "POWC_SAT"
                             | "POWRC" | "POWRC_SAT"
                             | "POWHC" | "POWHC_SAT"

    <BINop-instruction>     ::= <BINop> <maskedDstReg> ","
                               <vectorSrc> "," <vectorSrc>
    <BINop>                ::= "ADD"    | "ADD_SAT"
                             | "ADDR"   | "ADDR_SAT"
                             | "ADDH"   | "ADDH_SAT"
                             | "ADDX"   | "ADDX_SAT"
                             | "ADDC"   | "ADDC_SAT"
                             | "ADDRC"  | "ADDRC_SAT"
                             | "ADDHC"  | "ADDHC_SAT"
                             | "ADDXC"  | "ADDXC_SAT"
                             | "DP3"    | "DP3_SAT"
                             | "DP3R"   | "DP3R_SAT"
                             | "DP3H"   | "DP3H_SAT"
                             | "DP3X"   | "DP3X_SAT"
                             | "DP3C"   | "DP3C_SAT"
                             | "DP3RC"  | "DP3RC_SAT"
                             | "DP3HC"  | "DP3HC_SAT"
                             | "DP3XC"  | "DP3XC_SAT"
                             | "DP4"    | "DP4_SAT"
                             | "DP4R"   | "DP4R_SAT"
                             | "DP4H"   | "DP4H_SAT"
                             | "DP4X"   | "DP4X_SAT"
                             | "DP4C"   | "DP4C_SAT"
                             | "DP4RC"  | "DP4RC_SAT"
                             | "DP4HC"  | "DP4HC_SAT"
                             | "DP4XC"  | "DP4XC_SAT"
                             | "DST"    | "DST_SAT"
                             | "DSTR"   | "DSTR_SAT"
                             | "DSTH"   | "DSTH_SAT"
                             | "DSTC"   | "DSTC_SAT"
                             | "DSTRC"  | "DSTRC_SAT"
                             | "DSTHC"  | "DSTHC_SAT"
                             | "MAX"    | "MAX_SAT"
                             | "MAXR"   | "MAXR_SAT"
                             | "MAXH"   | "MAXH_SAT"
                             | "MAXX"   | "MAXX_SAT"
                             | "MAXC"   | "MAXC_SAT"
                             | "MAXRC"  | "MAXRC_SAT"
                             | "MAXHC"  | "MAXHC_SAT"
                             | "MAXXC"  | "MAXXC_SAT"
                             | "MIN"    | "MIN_SAT"
                             | "MINR"   | "MINR_SAT"
                             | "MINH"   | "MINH_SAT"
                             | "MINX"   | "MINX_SAT"
                             | "MINC"   | "MINC_SAT"
                             | "MINRC"  | "MINRC_SAT"
                             | "MINHC"  | "MINHC_SAT"
                             | "MINXC"  | "MINXC_SAT"
                             | "MUL"    | "MUL_SAT"
                             | "MULR"   | "MULR_SAT"
                             | "MULH"   | "MULH_SAT"
                             | "MULX"   | "MULX_SAT"
                             | "MULC"   | "MULC_SAT"
                             | "MULRC"  | "MULRC_SAT"
                             | "MULHC"  | "MULHC_SAT"
                             | "MULXC"  | "MULXC_SAT"
                             | "RFL"    | "RFL_SAT"
                             | "RFLR"   | "RFLR_SAT"
```

```
                               | "RFLH"  | "RFLH_SAT"
                               | "RFLC"  | "RFLC_SAT"
                               | "RFLRC" | "RFLRC_SAT"
                               | "RFLHC" | "RFLHC_SAT"
                               | "SEQ"   | "SEQ_SAT"
                               | "SEQR"  | "SEQR_SAT"
                               | "SEQH"  | "SEQH_SAT"
                               | "SEQX"  | "SEQX_SAT"
                               | "SEQC"  | "SEQC_SAT"
                               | "SEQRC" | "SEQRC_SAT"
                               | "SEQHC" | "SEQHC_SAT"
                               | "SEQXC" | "SEQXC_SAT"
                               | "SFL"   | "SFL_SAT"
                               | "SFLR"  | "SFLR_SAT"
                               | "SFLH"  | "SFLH_SAT"
                               | "SFLX"  | "SFLX_SAT"
                               | "SFLC"  | "SFLC_SAT"
                               | "SFLRC" | "SFLRC_SAT"
                               | "SFLHC" | "SFLHC_SAT"
                               | "SFLXC" | "SFLXC_SAT"
                               | "SGE"   | "SGE_SAT"
                               | "SGER"  | "SGER_SAT"
                               | "SGEH"  | "SGEH_SAT"
                               | "SGEX"  | "SGEX_SAT"
                               | "SGEC"  | "SGEC_SAT"
                               | "SGERC" | "SGERC_SAT"
                               | "SGEHC" | "SGEHC_SAT"
                               | "SGEXC" | "SGEXC_SAT"
                               | "SGT"   | "SGT_SAT"
                               | "SGTR"  | "SGTR_SAT"
                               | "SGTH"  | "SGTH_SAT"
                               | "SGTX"  | "SGTX_SAT"
                               | "SGTC"  | "SGTC_SAT"
                               | "SGTRC" | "SGTRC_SAT"
                               | "SGTHC" | "SGTHC_SAT"
                               | "SGTXC" | "SGTXC_SAT"
                               | "SLE"   | "SLE_SAT"
                               | "SLER"  | "SLER_SAT"
                               | "SLEH"  | "SLEH_SAT"
                               | "SLEX"  | "SLEX_SAT"
                               | "SLEC"  | "SLEC_SAT"
                               | "SLERC" | "SLERC_SAT"
                               | "SLEHC" | "SLEHC_SAT"
                               | "SLEXC" | "SLEXC_SAT"
                               | "SLT"   | "SLT_SAT"
                               | "SLTR"  | "SLTR_SAT"
                               | "SLTH"  | "SLTH_SAT"
                               | "SLTX"  | "SLTX_SAT"
                               | "SLTC"  | "SLTC_SAT"
                               | "SLTRC" | "SLTRC_SAT"
                               | "SLTHC" | "SLTHC_SAT"
                               | "SLTXC" | "SLTXC_SAT"
                               | "SNE"   | "SNE_SAT"
                               | "SNER"  | "SNER_SAT"
                               | "SNEH"  | "SNEH_SAT"
                               | "SNEX"  | "SNEX_SAT"
                               | "SNEC"  | "SNEC_SAT"
```

```
                                | "SNERC" | "SNERC_SAT"
                                | "SNEHC" | "SNEHC_SAT"
                                | "SNEXC" | "SNEXC_SAT"
                                | "STR"   | "STR_SAT"
                                | "STRR"  | "STRR_SAT"
                                | "STRH"  | "STRH_SAT"
                                | "STRX"  | "STRX_SAT"
                                | "STRC"  | "STRC_SAT"
                                | "STRRC" | "STRRC_SAT"
                                | "STRHC" | "STRHC_SAT"
                                | "STRXC" | "STRXC_SAT"
                                | "SUB"   | "SUB_SAT"
                                | "SUBR"  | "SUBR_SAT"
                                | "SUBH"  | "SUBH_SAT"
                                | "SUBX"  | "SUBX_SAT"
                                | "SUBC"  | "SUBC_SAT"
                                | "SUBRC" | "SUBRC_SAT"
                                | "SUBHC" | "SUBHC_SAT"
                                | "SUBXC" | "SUBXC_SAT"

    <TRIop-instruction>    ::= <TRIop> <maskedDstReg> ","
                               <vectorSrc> "," <vectorSrc> ","
                               <vectorSrc>

    <TRIop>                ::= "MAD"   | "MAD_SAT"
                                | "MADR"  | "MADR_SAT"
                                | "MADH"  | "MADH_SAT"
                                | "MADX"  | "MADX_SAT"
                                | "MADC"  | "MADC_SAT"
                                | "MADRC" | "MADRC_SAT"
                                | "MADHC" | "MADHC_SAT"
                                | "MADXC" | "MADXC_SAT"
                                | "LRP"   | "LRP_SAT"
                                | "LRPR"  | "LRPR_SAT"
                                | "LRPH"  | "LRPH_SAT"
                                | "LRPX"  | "LRPX_SAT"
                                | "LRPC"  | "LRPC_SAT"
                                | "LRPRC" | "LRPRC_SAT"
                                | "LRPHC" | "LRPHC_SAT"
                                | "LRPXC" | "LRPXC_SAT"
                                | "X2D"   | "X2D_SAT"
                                | "X2DR"  | "X2DR_SAT"
                                | "X2DH"  | "X2DH_SAT"
                                | "X2DC"  | "X2DC_SAT"
                                | "X2DRC" | "X2DRC_SAT"
                                | "X2DHC" | "X2DHC_SAT"

    <KILop-instruction>    ::= <KILop> <ccMask>

    <KILop>                ::= "KIL"

    <TEXop-instruction>    ::= <TEXop> <maskedDstReg> ","
                               <vectorSrc> "," <texImageId>
    <TEXop>                 ::= "TEX"  | "TEX_SAT"
                              | "TEXC" | "TEXC_SAT"
                              | "TXP"  | "TXP_SAT"
                              | "TXPC" | "TXPC_SAT"

    <TXDop-instruction>     ::= <TXDop> <maskedDstReg> ","
                                <vectorSrc> "," <vectorSrc> ","
                                <vectorSrc> "," <texImageId>

    <TXDop>                 ::= "TXD"  | "TXD_SAT"
                              | "TXDC" | "TXDC_SAT"

    <scalarSrc>             ::= <absScalarSrc>
                              | <baseScalarSrc>

    <absScalarSrc>          ::= <negate> "|" <baseScalarSrc> "|"

    <baseScalarSrc>         ::= <signedScalarConstant>
                              | <negate> <namedScalarConstant>
                              | <negate> <vectorConstant> <scalarSuffix>
                              | <negate> <namedLocalParameter> <scalarSuffix>
                              | <negate> <numberedLocal> <scalarSuffix>
                              | <negate> <srcRegister> <scalarSuffix>

    <vectorSrc>             ::= <absVectorSrc>
                              | <baseVectorSrc>

    <absVectorSrc>          ::= <negate> "|" <baseVectorSrc> "|"

    <baseVectorSrc>         ::= <signedScalarConstant>
                              | <negate> <namedScalarConstant>
                              | <negate> <vectorConstant> <scalarSuffix>
                              | <negate> <vectorConstant> <swizzleSuffix>
                              | <negate> <namedLocalParameter> <scalarSuffix>
                              | <negate> <namedLocalParameter> <swizzleSuffix>
                              | <negate> <numberedLocal> <scalarSuffix>
                              | <negate> <numberedLocal> <swizzleSuffix>
                              | <negate> <srcRegister> <scalarSuffix>
                              | <negate> <srcRegister> <swizzleSuffix>

    <maskedDstReg>          ::= <dstRegister> <optionalWriteMask>
                                <optionalCCMask>

    <dstRegister>           ::= <fragTempReg>
                              | <fragOutputReg>
                              | "RC"
                              | "HC"

    <optionalCCMask>        ::= "(" <ccMask> ")"
                              | ""

    <ccMask>                ::= <ccMaskRule> <swizzleSuffix>
                              | <ccMaskRule> <scalarSuffix>

    <ccMaskRule>            ::= "EQ" | "GE" | "GT" | "LE" | "LT" | "NE" |
                                "TR" | "FL"
    <optionalWriteMask>    ::= ""
                             | "." "x"
                             | "."     "y"
                             | "." "x" "y"
                             | "."         "z"
                             | "." "x"     "z"
                             | "."     "y" "z"
                             | "." "x" "y" "z"
                             | "."             "w"
                             | "." "x"         "w"
                             | "."     "y"     "w"
                             | "." "x" "y"     "w"
                             | "."         "z" "w"
                             | "." "x"     "z" "w"
                             | "."     "y" "z" "w"
                             | "." "x" "y" "z" "w"

    <srcRegister>          ::= <fragAttribReg>
                             | <fragTempReg>

    <fragAttribReg>        ::= "f" "[" <fragAttribRegId> "]"

    <fragAttribRegId>      ::= "WPOS" | "COL0" | "COL1" | "FOGC" | "TEX0"
                             | "TEX1" | "TEX2" | "TEX3" | "TEX4" | "TEX5"
                             | "TEX6" | "TEX7"

    <fragTempReg>          ::= <fragF32Reg>
                             | <fragF16Reg>

    <fragF32Reg>           ::= "R0"  | "R1"  | "R2"  | "R3"
                             | "R4"  | "R5"  | "R6"  | "R7"
                             | "R8"  | "R9"  | "R10" | "R11"
                             | "R12" | "R13" | "R14" | "R15"
                             | "R16" | "R17" | "R18" | "R19"
                             | "R20" | "R21" | "R22" | "R23"
                             | "R24" | "R25" | "R26" | "R27"
                             | "R28" | "R29" | "R30" | "R31"

    <fragF16Reg>           ::= "H0"  | "H1"  | "H2"  | "H3"
                             | "H4"  | "H5"  | "H6"  | "H7"
                             | "H8"  | "H9"  | "H10" | "H11"
                             | "H12" | "H13" | "H14" | "H15"
                             | "H16" | "H17" | "H18" | "H19"
                             | "H20" | "H21" | "H22" | "H23"
                             | "H24" | "H25" | "H26" | "H27"
                             | "H28" | "H29" | "H30" | "H31"
                             | "H32" | "H33" | "H34" | "H35"
                             | "H36" | "H37" | "H38" | "H39"
                             | "H40" | "H41" | "H42" | "H43"
                             | "H44" | "H45" | "H46" | "H47"
                             | "H48" | "H49" | "H50" | "H51"
                             | "H52" | "H53" | "H54" | "H55"
                             | "H56" | "H57" | "H58" | "H59"
                             | "H60" | "H61" | "H62" | "H63"

    <fragOutputReg>        ::= "o" "[" <fragOutputRegName> "]"
    <fragOutputRegName>    ::= "COLR" | "COLH" | "DEPR" | "TEX0" | "TEX1"
                             | "TEX2" | "TEX3"

    <numberedLocal>        ::= "p" "[" <localNumber> "]"

    <localNumber>          ::= <integer> from 0 to
                               MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV - 1

    <scalarSuffix>         ::= "." <component>

    <swizzleSuffix>        ::= ""
                             | "." <component> <component>
                                   <component> <component>

    <component>            ::= "x" | "y" | "z" | "w"

    <texImageId>           ::= <texImageUnit> "," <texImageTarget>

    <texImageUnit>         ::= "TEX0"  | "TEX1"  | "TEX2"  | "TEX3"
                             | "TEX4"  | "TEX5"  | "TEX6"  | "TEX7"
                             | "TEX8"  | "TEX9"  | "TEX10" | "TEX11"
                             | "TEX12" | "TEX13" | "TEX14" | "TEX15"

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

    <constantDefinition>   ::= "DEFINE" <namedVectorConstant> "="
                               <vectorConstant>
                             | "DEFINE" <namedScalarConstant> "="
                               <scalarConstant>

    <localDeclaration>     ::= "DECLARE" <namedLocalParameter>
                               <optionalLocalValue>

    <optionalLocalValue>   ::= ""
                             | "=" <vectorConstant>
                             | "=" <scalarConstant>

    <vectorConstant>       ::= {" <vectorConstantList> "}"
                             | <namedVectorConstant>

    <vectorConstantList>   ::= <scalarConstant>
                             | <scalarConstant> "," <scalarConstant>
                             | <scalarConstant> "," <scalarConstant> ","
                               <scalarConstant>
                             | <scalarConstant> "," <scalarConstant> ","
                               <scalarConstant> "," <scalarConstant>

    <scalarConstant>       ::= <signedScalarConstant>
                             | <namedScalarConstant>

    <signedScalarConstant> ::= <optionalSign> <floatConstant>

    <namedScalarConstant>  ::= <identifier>    ((name of a scalar constant
                                                 in a DEFINE instruction))

    <namedVectorConstant>  ::= <identifier>    ((name of a vector constant
                                                 in a DEFINE instruction))
```

```
<namedLocalParameter>  ::= <identifier>    ((name of a local parameter
                                              in a DECLARE instruction))

<negate>               ::= "-" | "+" | ""

<optionalSign>         ::= "-" | "+" | ""

<identifier>           ::= see text below

<floatConstant>        ::= see text below
```

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z", "_", and "$") and digits ("0" through "9"); the first character must be a letter. The underscore ("_") and dollar sign ("$") count as a letters. Upper and lower case letters are different (names are case-sensitive).

The <floatConstant> rule matches a floating-point constant consisting of an integer part, a decimal point, a fraction part, an "e" or "E", and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of on or more digits ("0" through "9"). Either the integer part or the fraction parts (not both) may be missing; either the decimal point or the "e" (or "E") and the exponent (not both) may be missing.

A fragment program fails to load if it contains more than 1024 executable instructions. Executable instructions are those matching the <instruction> rule in the grammar, and do not include DEFINE or DECLARE instructions.

A fragment program fails to load if its total temporary and output register count exceeds 64. Each fp32 temporary or output register used by the program (R0-R31, o[COLR], and o[DEPR]) counts as two registers; each fp16 temporary or output register used by the program (H0-H63 and o[COLH]) count as a single register. For combiner programs, o[TEX0], o[TEX1], o[TEX2], and o[TEX3] are counted as one register each, whether or not they are used by the program.

A fragment program fails to load if any instruction sources more than one unique fragment attribute register. Instructions sourcing the same attribute register multiple times are acceptable.

A fragment program fails to load if any instruction sources more than one unique program parameter register. Instructions sourcing the same program parameter multiple times are acceptable.

A fragment program fails to load if multiple texture lookup instructions reference different targets for the same texture image unit.

A color fragment program (indicated by the "!!FP1.0" prefix) fails to load if it writes to any of the o[TEX0], o[TEX1], o[TEX2], or o[TEX3] output registers, or if it writes to both the o[COLR] and o[COLH] output registers.

A combiner fragment program (indicated by the "!!FCP1.0" prefix) fails to load if it fails to write to any of the o[TEX0], o[TEX1], o[TEX2], or o[TEX3] output registers, or if it writes to either the o[COLR] or the o[COLH] output register.

The error INVALID_OPERATION is generated by LoadProgramNV if a fragment program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated by LoadProgramNV if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded fragment program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in Sections 3.11.4 and 3.11.5.

**Section 3.11.4, Fragment Program Operation**

There are forty-five fragment program instructions. Fragment program instructions may have up to eight variants, including a suffix of "R", "H", or "X" to specify arithmetic precision (section 3.11.4.2), a suffix of "C" to allow an update of the condition code register (section 3.11.4.4), and a suffix of "_SAT" to clamp the result vector components to the range [0,1] (section 3.11.4.4). For example, the sixteen forms of the "ADD" instruction are "ADD", "ADDR", "ADDH", "ADDX", "ADDC", "ADDRC", "ADDHC", "ADDXC", "ADD_SAT", "ADDR_SAT", "ADDH_SAT", "ADDX_SAT", "ADDC_SAT", "ADDRC_SAT", "ADDHC_SAT", and "ADDXC_SAT".

Some mathematical instructions that support precision suffixes, typically those that involve complicated floating-point computations, do not support the "X" precision suffix.

The fragment program instructions and their respective input and output parameters are summarized in Table X.4.

| Instruction | Inputs | Output | Description |
|---|---|---|---|
| ADD[RHX][C][_SAT] | v,v | v | add |
| COS[RH ][C][_SAT] | s | ssss | cosine |
| DDX[RH ][C][_SAT] | v | v | derivative relative to x |
| DDY[RH ][C][_SAT] | v | v | derivative relative to y |
| DP3[RHX][C][_SAT] | v,v | ssss | 3-component dot product |
| DP4[RHX][C][_SAT] | v,v | ssss | 4-component dot product |
| DST[RH ][C][_SAT] | v,v | v | distance vector |
| EX2[RH ][C][_SAT] | s | ssss | exponential base 2 |
| FLR[RHX][C][_SAT] | v | v | floor |
| FRC[RHX][C][_SAT] | v | v | fraction |
| KIL | none | none | conditionally discard fragment |
| LG2[RH ][C][_SAT] | s | ssss | logarithm base 2 |
| LIT[RH ][C][_SAT] | v | v | compute light coefficients |
| LRP[RHX][C][_SAT] | v,v,v | v | linear interpolation |
| MAD[RHX][C][_SAT] | v,v,v | v | multiply and add |
| MAX[RHX][C][_SAT] | v,v | v | maximum |
| MIN[RHX][C][_SAT] | v,v | v | minimum |
| MOV[RHX][C][_SAT] | v | v | move |
| MUL[RHX][C][_SAT] | v,v | v | multiply |
| 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 |
| POW[RH ][C][_SAT] | s,s | ssss | exponentiation (x^y) |
| RCP[RH ][C][_SAT] | s | ssss | reciprocal |
| RFL[RH ][C][_SAT] | v,v | v | reflection vector |
| RSQ[RH ][C][_SAT] | s | ssss | reciprocal square root |
| SEQ[RHX][C][_SAT] | v,v | v | set on equal |
| SFL[RHX][C][_SAT] | v,v | v | set on false |
| SGE[RHX][C][_SAT] | v,v | v | set on greater than or equal |
| SGT[RHX][C][_SAT] | v,v | v | set on greater than |
| SIN[RH ][C][_SAT] | s | ssss | sine |
| SLE[RHX][C][_SAT] | v,v | v | set on less than or equal |
| SLT[RHX][C][_SAT] | v,v | v | set on less than |
| SNE[RHX][C][_SAT] | v,v | v | set on not equal |
| STR[RHX][C][_SAT] | v,v | v | set on true |
| SUB[RHX][C][_SAT] | v,v | v | subtract |
| TEX[C][_SAT] | v | v | texture lookup |
| TXD[C][_SAT] | v,v,v | v | texture lookup w/partials |
| TXP[C][_SAT] | v | v | projective texture lookup |
| UP2H[C][_SAT] | s | v | unpack two 16-bit floats |
| UP2US[C][_SAT] | s | v | unpack two unsigned 16-bit scalars |
| UP4B[C][_SAT] | s | v | unpack four signed 8-bit scalars |
| UP4UB[C][_SAT] | s | v | unpack four unsigned 8-bit scalars |
| X2D[RH ][C][_SAT] | v,v,v | v | 2D coordinate transformation |

**Table X.4: Summary of fragment program instructions. "[RHX]" indicates an optional arithmetic precision suffix. "[C]" indicates an optional condition code update suffix. "[_SAT]" indicates an optional clamp of result vector components to [0,1]. "v" indicates a 4-component vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.**

**Section 3.11.4.1: Fragment Program Storage Precision**

Registers in fragment program are stored in two different representations: 16-bit floating-point (fp16) and 32-bit floating-point (fp32). There is an additional 12-bit fixed-point representation (fx12) used only as an internal representation for instructions with the "X" precision qualifier.

In the 32-bit float (fp32) representation, each component is represented in floating-point with eight exponent and twenty-three mantissa bits, as in the standard IEEE single-precision format. If S represents the sign (0 or 1), E represents the exponent in the range [0,255], and M represents the mantissa in the range [0,2^23-1], then an fp32 float is decoded as:

```
(-1)^S * 0.0,                          if E == 0,
(-1)^S * 2^(E-127) * (1 + M/2^23),     if 0 < E < 255,
(-1)^S * INF,                          if E == 255 and M == 0,
NaN,                                   if E == 255 and M != 0.
```

INF (Infinity) is a special representation indicating numerical overflow. NaN (Not a Number) is a special representation indicating the result of illegal arithmetic operations, such as division by zero. Note that all normal fp32 values, zero, and INF have an associated sign. -0.0 and +0.0 are considered equivalent for the purposes of comparisons.

This representation is identical to the IEEE single-precision floating-point standard, except that no special representation is provided for denorms -- numbers in the range (-2^-126, +2^-126). All such numbers are flushed to zero.

In a 16-bit float (fp16) register, each component is represented similarly, except with only five exponent and ten mantissa bits. If S represents the sign (0 or 1), E represents the exponent in the range [0,31], and M represents the mantissa in the range [0,2^10-1], then an fp32 float is decoded as:

```
(-1)^S * 0.0,                          if E == 0 and M == 0,
(-1)^S * 2^-14 * M/2^10                if E == 0 and M != 0,
(-1)^S * 2^(E-15) * (1 + M/2^10),      if 0 < E < 31,
(-1)^S * INF,                          if E == 31 and M == 0, or
NaN,                                   if E == 31 and M != 0.
```

One important difference is that the fp16 representation, unlike fp32, supports denorms to maximize the limited precision of the 16-bit floating point encodings.

In the 12-bit fixed-point (fx12) format, numbers are represented as signed 12-bit two's complement integers with 10 fraction bits. The range of representable values is [-2048/1024, +2047/1024].

**Section 3.11.4.2: Fragment Program Operation Precision**

Fragment program instructions frequently perform mathematical operations. Such operations may be performed at one of three different precisions. Fragment programs can specify the precision of each instruction by using the precision suffix. If an instruction has a suffix of "R", calculations are carried out with 32-bit floating point operands and results. If an instruction has a suffix of "H", calculations are carried out using 16-bit floating point operands and results.  If an instruction has a suffix of "X", calculations are carried out using 12-bit fixed point operands and results.  For example, the instruction "MULR" performs a 32-bit floating-point multiply, "MULH" performs a 16-bit floating-point multiply, and "MULX" performs a 12-bit fixed-point multiply.  If no precision suffix is specified, calculations are carried out using the precision of the temporary register receiving the result.

Fragment program instructions may source registers or constants whose precisions differ from the precision specified with the instruction. Instructions may also generate intermediate results with a different precision than that of the destination register.  In these cases, the values sourced are converted to the precision specified by the instruction.

When converting to fx12 format, -INF and any values less than -2048/1024 become -2048/1024.  +INF, and any values greater than +2047/1024 become +2047/1024.  NaN becomes 0.

When converting to fp16 format, any values less than or equal to -2^16 are converted to -INF.  Any values greater than or equal to +2^16 are converted to +INF.  -INF, +INF, NaN, -0.0, and +0.0 are unchanged.  Any other values that are not exactly representable in fp16 format are converted to one of the two nearest representable values.

When converting to fp32 format, any values less than or equal to -2^128 are converted to -INF.  Any values greater than or equal to +2^128 are converted to +INF.  -INF, +INF, NaN, -0.0, and +0.0 are unchanged.  Any other values that are not exactly representable in fp32 format are converted to one of the two nearest representable values.

Fragment program instructions using the fragment attribute registers f[FOGC] or f[TEX0] through f[TEX7] will be carried out at full fp32 precision, regardless of the precision specified by the instruction.

**Section 3.11.4.3:  Fragment Program Operands**

Except for KIL, fragment program instructions operate on either vector or scalar operands, indicated in the grammar (see section 3.11.3) by the rules <vectorSrc> and <scalarSrc> respectively.

The basic set of scalar operands is defined by the grammar rule <baseScalarSrc>.  Scalar operands can be scalar constants (embedded or named), or single components of vector constants, local parameters, or registers allowed by the <srcRegister> rule.  A vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w" select the x, y, z, and w components, respectively, of the vector.

The basic set of vector operands is defined by the grammar rule <baseVectorSrc>.  Vector operands can include vector constants, local parameters, or registers allowed by the <srcRegister> rule.

Basic vector operands can be swizzled according to the <swizzleSuffix> rule.  In its most general form, the <swizzleSuffix> rule matches the pattern ".????" where each question mark is one of "x", "y", "z", or "w". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}. If the <swizzleSuffix> rule matches "", it is treated as though it were ".xyzw".

Operands can optionally be negated according to the <negate> rule in <baseScalarSrc> or <baseVectorSrc>. If the <negate> matches "-", each value is negated.

The absolute value of operands can be taken if the <vectorSrc> or <scalarSrc> rules match <absScalarSrc> or <absVectorSrc>. In this case, the absolute value of each component is taken. In addition, if the <negate> rule in <absScalarSrc> or <absVectorSrc> matches "-", the result is then negated.

Instructions requiring vector operands can also use scalar operands in the case where the <vectorSrc> rule matches <scalarSrc>. In such cases, a 4-component vector is produced by replicating the scalar.

After operands are loaded, they are converted to a data type corresponding to the operation precision specified in the fragment program instruction.

The following pseudo-code spells out the operand generation process. "SrcT" and "InstT" refer to the data types of the specified register or constant and the instruction, respectively. "VecSrcT" and "VecInstT" refer to 4-component vectors of the corresponding type. "absolute" is TRUE if the operand matches the <absScalarSrc> or <absVectorSrc> rules, and FALSE otherwise. "negateBase" is TRUE if the <negate> rule in <baseScalarSrc> or <baseVectorSrc> matches "-" and FALSE otherwise. "negateAbs" is TRUE if the <negate> rule in <absScalarSrc> or <absVectorSrc> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation. TypeConvert() is assumed to convert a scalar of type SrcT to a scalar of type InstT using the type conversion process specified above.

```
      VecInstT VectorLoad(VecSrcT source)
      {
          VecSrcT srcVal;
          VecInstT convertedVal;

          srcVal.x = source.c***;
          srcVal.y = source.*c**;
          srcVal.z = source.**c*;
          srcVal.w = source.***c;
          if (negateBase) {
             srcVal.x = -srcVal.x;
             srcVal.y = -srcVal.y;
             srcVal.z = -srcVal.z;
             srcVal.w = -srcVal.w;
          }
          if (absolute) {
             srcVal.x = abs(srcVal.x);
             srcVal.y = abs(srcVal.y);
             srcVal.z = abs(srcVal.z);
             srcVal.w = abs(srcVal.w);
          }
          if (negateAbs) {
             srcVal.x = -srcVal.x;
             srcVal.y = -srcVal.y;
             srcVal.z = -srcVal.z;
             srcVal.w = -srcVal.w;
          }

          convertedVal.x = TypeConvert(srcVal.x);
          convertedVal.y = TypeConvert(srcVal.y);
          convertedVal.z = TypeConvert(srcVal.z);
          convertedVal.w = TypeConvert(srcVal.w);
          return convertedVal;
      }

      InstT ScalarLoad(VecSrcT source)
      {
          SrcT srcVal;
          InstT convertedVal;

          srcVal = source.c***;
          if (negateBase) {
            srcVal = -srcVal;
          }
          if (absolute) {
             srcVal = abs(srcVal);
          }
          if (negateAbs) {
            srcVal = -srcVal;
          }

          convertedVal = TypeConvert(srcVal);
          return convertedVal;
      }
```

**Section 3.11.4.4, Fragment Program Destination Register Update**

Each fragment program instruction, except for KIL, writes a 4-component result vector to a single temporary or output register.

The four components of the result vector are first optionally clamped to the range [0,1]. The components will be clamped if and only if the result clamp suffix "_SAT" is present in the instruction name. The instruction "ADD_SAT" will clamp the results to [0,1]; the otherwise equivalent instruction "ADD" will not.

Since the instruction may be carried out at a different precision than the destination register, the components of the results vector are then converted to the data type corresponding to destination register.

Writes to individual components of the temporary register are controlled by two sets of enables: individual component write masks specified as part of the instruction and the optional condition code mask.

The component write mask is specified by the <optionalWriteMask> 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 optional condition code mask is specified by the <optionalCCMask> rule found in the <maskedDstReg> rule. If <optionalCCMask> matches "", all components are enabled. Otherwise, the condition code register is loaded and swizzled according to the swizzling 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.

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

A fragment 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 name. 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.  Note that 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
    unchanged.

    In the following example code,

        # R1=(-2, 0, 2, NaN)
        MOVC R0, R1;
        MOVC R0.xyz, R1.yzwx;
        MOVC R0 (NE), R1.zywx;

    the first instruction writes (-2,0,2,NaN) to R0 and updates the condition
    code to (LT,EQ,GT,UN).  The second instruction, writes to the "w"
    component of R0 and the condition code are disabled, 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,NaN,-2,0) and the condition code ends up with (EQ,UN,LT,EQ).

    The following pseudocode illustrates the process of writing a result
    vector to the destination register.  In the example, "ccMaskRule" refers
    to the condition code mask rule given by <ccMaskRule> (or "" if no rule is
    specified), "instrmask" refers to the component write mask given by the
    <optionalWriteMask> rule, "updatecc" is TRUE if condition code updates are
    enabled, and "clamp01" is TRUE if [0,1] result clamping is enabled.
    "destination" and "cc" refer to the register selected by <dstRegister> and
    the condition code, respectively.

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

```
  void UpdateDestination(VecDstT destination, VecInstT result)
  {
      // Load the original destination register and condition code.
      VecDstT resultDst;
      VecDstT merged;
      VecCC   mergedCC;

      // Clamp the result vector components to [0,1], if requested.
      if (clamp01) {
          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;
      }

      // Convert the result to the type of the destination register.
      resultDst.x = TypeConvert(result.x);
      resultDst.y = TypeConvert(result.y);
      resultDst.z = TypeConvert(result.z);
      resultDst.w = TypeConvert(result.w);

      // 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 (updatecc) mergedCC.x = GenerateCC(result.x);
      }
      if (instrMask.y && TestCC(cc.*c**)) {
          merged.y = result.y;
          if (updatecc) mergedCC.y = GenerateCC(result.y);
      }
      if (instrMask.z && TestCC(cc.**c*)) {
          merged.z = result.z;
          if (updatecc) mergedCC.z = GenerateCC(result.z);
      }
      if (instrMask.w && TestCC(cc.***c)) {
          merged.w = result.w;
          if (updatecc) mergedCC.w = GenerateCC(result.w);
      }

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

**Section 3.11.5, Fragment Program Instruction Set**

The following sections describe the instruction set available to fragment programs.

**Section 3.11.5.1, ADD: Add**

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x + tmp1.x;
  result.y = tmp0.y + tmp1.y;
  result.z = tmp0.z + tmp1.z;
  result.w = tmp0.w + tmp1.w;
```

The following special-case rules apply to addition:

```
  1. "A+B" is always equivalent to "B+A".
  2. NaN + <x> = NaN, for all <x>.
  3. +INF + <x> = +INF, for all <x> except NaN and -INF.
  4. -INF + <x> = -INF, for all <x> except NaN and +INF.
  5. +INF + -INF = NaN.
  6. -0.0 + <x> = <x>, for all <x>.
  7. +0.0 + <x> = <x>, for all <x> except -0.0.
```

**Section 3.11.5.2, COS: Cosine**

The COS instruction approximates the cosine of the angle specified by the scalar operand and replicates the approximation to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

The approximation function ApproxCosine is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxCosine(x) - cos(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

```
  1. ApproxCosine(NaN) = NaN.
  2. ApproxCosine(+/-INF) = NaN.
  3. ApproxCosine(+/-0.0) = +1.0.
```

**Section 3.11.5.3, DDX: Derivative Relative to X**

The DDX instruction computes approximate partial derivatives of the four components of the single operand with respect to the X window coordinate to yield a result vector. The partial derivative is evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialX(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

For components with partial derivatives that overflow (including +/-INF inputs), the resulting partials may be encoded as large floating-point numbers instead of +/-INF.

**Section 3.11.5.4, DDY: Derivative Relative to Y**

The DDY instruction computes approximate partial derivatives of the four components of the single operand with respect to the Y window coordinate to yield a result vector. The partial derivative is evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialY(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

For components with partial derivatives that overflow (including +/-INF inputs), the resulting partials may be encoded as large floating-point numbers instead of +/-INF.

**Section 3.11.5.5, DP3: 3-Component Dot Product**

The DP3 instruction computes a three component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z);
```

**Section 3.11.5.6, DP4: 4-Component Dot Product**

The DP4 instruction computes a four component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp2.z) + (tmp0.w * tmp1.w);
```

**Section 3.11.5.7, DST: Distance Vector**

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = 1.0;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z;
  result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DOT3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-fragment light attenuation calculations: a DOT3 operation involving the distance vector and an attenuation constants vector will yield the attenuation factor.

**Section 3.11.5.8, EX2: Exponential Base 2**

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = Approx2ToX(tmp);
  result.y = Approx2ToX(tmp);
  result.z = Approx2ToX(tmp);
  result.w = Approx2ToX(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | Approx2ToX(x) - 2^x | < 1.0 / 2^22, if 0.0 <= x < 1.0,
```

and, in general,

```
  | Approx2ToX(x) - 2^x | < (1.0 / 2^22) * (2^floor(x)).
```

The following special-case rules apply to exponential approximation:

1. Approx2ToX(NaN) = NaN.
2. Approx2ToX(-INF) = +0.0.
3. Approx2ToX(+INF) = +INF.
4. Approx2ToX(+/-0.0) = +1.0.

**Section 3.11.5.9, FLR: Floor**

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

The following special-case rules apply to floor computation:

1. floor(NaN) = NaN.
2. floor(<x>) = <x>, for -0.0, +0.0, -INF, and +INF. In all cases, the sign of the result is equal to the sign of the operand.

**Section 3.11.5.10, FRC: Fraction**

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.00, 1.00).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = tmp.x - floor(tmp.x);
  result.y = tmp.y - floor(tmp.y);
  result.z = tmp.z - floor(tmp.z);
  result.w = tmp.w - floor(tmp.w);
```

The following special-case rules, which can be derived from the rules for FLR and ADD apply to fraction computation:

1. fraction(NaN) = NaN.
2. fraction(+/-INF) = NaN.
3. fraction(+/-0.0) = +0.0.

**Section 3.11.5.11, KIL: Conditionally Discard Fragment**

The KIL instruction is unlike any other instruction in the instruction set. This instruction evaluates components of a swizzled condition code using a test expression identical to that used to evaluate condition code write masks (Section 3.11.4.4). If any condition code component evaluates to TRUE, the fragment is discarded. Otherwise, the instruction has no effect. The condition code components are specified, swizzled, and evaluated in the same manner as the condition code write mask.

```
  if (TestCC(rc.c***) || TestCC(rc.*c**) ||
      TestCC(rc.**c*) || TestCC(rc.***c)) {
     // Discard the fragment.
  } else {
    // Do nothing.
  }
```

If the fragment is discarded, it is treated as though it were not produced by rasterization. In particular, none of the per-fragment operations (such as stencil tests, blends, stencil, depth, or color buffer writes) are performed on the fragment.

**Section 3.11.5.12, LG2: Logarithm Base 2**

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxLog2(x) - log_2(x) | < 1.0 / 2^22.
```

The following special-case rules apply to logarithm approximation:

```
  1. ApproxLog2(NaN) = NaN.
  2. ApproxLog2(+INF) = +INF.
  3. ApproxLog2(+/-0.0) = -INF.
  4. ApproxLog2(x) = NaN, -INF < x < -0.0.
  5. ApproxLog2(-INF) = NaN.
```

**Section 3.11.5.13, LIT: Compute Light Coefficients**

The LIT instruction accelerates per-fragment lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm).

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (t.x < 0) t.x = 0;
  if (t.y < 0) t.y = 0;
  result.x = 1.0;
  result.y = t.x;
  result.z = (t.x > 0) ? ApproxPower(t.y, t.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation used to compute result.z are identical to that used in the POW instruction, including errors and the processing of any special cases.

**Section 3.11.5.14, LRP: Linear Interpolation**

The LRP instruction performs a component-wise linear interpolation to yield a result vector. It interpolates between the components of the second and third operands, using the first operand as a weight.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

**Section 3.11.5.15, MAD: Multiply and Add**

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = tmp0.x * tmp1.x + tmp2.x;
result.y = tmp0.y * tmp1.y + tmp2.y;
result.z = tmp0.z * tmp1.z + tmp2.z;
result.w = tmp0.w * tmp1.w + tmp2.w;
```

**Section 3.11.5.16, MAX: maximum**

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = max(tmp0.x, tmp1.x);
result.y = max(tmp0.y, tmp1.y);
result.z = max(tmp0.z, tmp1.z);
result.w = max(tmp0.w, tmp1.w);
```

The following special cases apply to the maximum operation:

1. max(A,B) is always equivalent to max(B,A).
2. max(NaN, <x>) == NaN, for all <x>.

**Section 3.11.5.17, MIN: minimum**

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = min(tmp0.x, tmp1.x);
result.y = min(tmp0.y, tmp1.y);
result.z = min(tmp0.z, tmp1.z);
result.w = min(tmp0.w, tmp1.w);
```

The following special cases apply to the minimum operation:

1. min(A,B) is always equivalent to min(B,A).
2. min(NaN, <x>) == NaN, for all <x>.

**Section 3.11.5.18, MOV: Move**

The MOV instruction copies the value of the operand to yield a result vector.

```
result = VectorLoad(op0);
```

**Section 3.11.5.19, MUL: Multiply**

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x * tmp1.x;
result.y = tmp0.y * tmp1.y;
result.z = tmp0.z * tmp1.z;
result.w = tmp0.w * tmp1.w;
```

The following special-case rules apply to multiplication:

1. "A*B" is always equivalent to "B*A".
2. NaN * <x> = NaN, for all <x>.
3. +/-0.0 * +/-INF = NaN.
4. +/-0.0 * <x> = +/-0.0, for all <x> except -INF, +INF, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
5. +/-INF * <x> = +/-INF, for all <x> except -0.0, +0.0, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
6. +1.0 * <x> = <x>, for all <x>.

**Section 3.11.5.20, PK2H: Pack Two 16-bit Floats**

The PK2H instruction converts the "x" and "y" components of the single operand into 16-bit floating-point format, packs the bit representation of these two floats into a 32-bit value, and replicates that value to all four components of the result vector. The PK2H instruction can be reversed by the UP2H instruction below.

```
  tmp0 = VectorLoad(op0);
  /* result obtained by combining raw bits of tmp0.x, tmp0.y */
  result.x = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.y = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.z = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.w = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

**Section 3.11.5.21, PK2US: Pack Two Unsigned 16-bit Scalars**

The PK2US instruction converts the "x" and "y" components of the single operand into a packed pair of 16-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the two converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK2US instruction can be reversed by the UP2US instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  us.x = round(65535.0 * tmp0.x);  /* us is a ushort vector */
  us.y = round(65535.0 * tmp0.y);
  /* result obtained by combining raw bits of us. */
  result.x = ((us.x) | (us.y << 16));
  result.y = ((us.x) | (us.y << 16));
  result.z = ((us.x) | (us.y << 16));
  result.w = ((us.x) | (us.y << 16));
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

**Section 3.11.5.22, PK4B: Pack Four Signed 8-bit Scalars**

The PK4B instruction converts the four components of the single operand into 8-bit signed quantities. The signed quantities are represented in a bit pattern where all '0' bits corresponds to -128/127 and all '1' bits corresponds to +127/127. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4B instruction can be reversed by the UP4B instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < -128/127) tmp0.x = -128/127;
  if (tmp0.y < -128/127) tmp0.y = -128/127;
  if (tmp0.z < -128/127) tmp0.z = -128/127;
  if (tmp0.w < -128/127) tmp0.w = -128/127;
  if (tmp0.x > +127/127) tmp0.x = +127/127;
  if (tmp0.y > +127/127) tmp0.y = +127/127;
  if (tmp0.z > +127/127) tmp0.z = +127/127;
  if (tmp0.w > +127/127) tmp0.w = +127/127;
  ub.x = round(127.0 * tmp0.x + 128.0);  /* ub is a ubyte vector */
  ub.y = round(127.0 * tmp0.y + 128.0);
  ub.z = round(127.0 * tmp0.z + 128.0);
  ub.w = round(127.0 * tmp0.w + 128.0);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

**Section 3.11.5.23, PK4UB: Pack Four Unsigned 8-bit Scalars**

The PK4UB instruction converts the four components of the single operand into a packed grouping of 8-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4UB instruction can be reversed by the UP4UB instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  if (tmp0.z < 0.0) tmp0.z = 0.0;
  if (tmp0.z > 1.0) tmp0.z = 1.0;
  if (tmp0.w < 0.0) tmp0.w = 0.0;
  if (tmp0.w > 1.0) tmp0.w = 1.0;
  ub.x = round(255.0 * tmp0.x);  /* ub is a ubyte vector */
  ub.y = round(255.0 * tmp0.y);
  ub.z = round(255.0 * tmp0.z);
  ub.w = round(255.0 * tmp0.w);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

The result must be written to a register with 32-bit components (an "R" register, o[COLR], or o[DEPR]). A fragment program will fail to load if any other register type is specified.

**Section 3.11.5.24, POW: Exponentiation**

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

```
  tmp0 = ScalarLoad(op0);
  tmp1 = ScalarLoad(op1);
  result.x = ApproxPower(tmp0, tmp1);
  result.y = ApproxPower(tmp0, tmp1);
  result.z = ApproxPower(tmp0, tmp1);
  result.w = ApproxPower(tmp0, tmp1);
```

The exponentiation approximation function is defined in terms of the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions, including errors and the processing of any special cases. In particular,

```
  ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a)).
```

The following special-case rules, which can be derived from the rules in the LG2, MUL, and EX2 instructions, apply to exponentiation:

```
  1. ApproxPower(<x>, <y>) = NaN, if x < -0.0,
  2. ApproxPower(<x>, <y>) = NaN, if x or y is NaN.
  3. ApproxPower(+/-0.0, +/-0.0) = NaN.
  4. ApproxPower(+INF, +/-0.0) = NaN.
  5. ApproxPower(+1.0, +/-INF) = NaN.
  6. ApproxPower(+/-0.0, <x>) = +0.0, if x > +0.0.
  7. ApproxPower(+/-0.0, <x>) = +INF, if x < -0.0.
  8. ApproxPower(+1.0, <x>)   = +1.0, if -INF < x < +INF.
  9. ApproxPower(+INF, <x>) = +INF, if x > +0.0.
  10. ApproxPower(+INF, <x>) = +INF, if x < -0.0.
  11. ApproxPower(<x>, +/-0.0) = +1.0, if +0.0 < x < +INF.
  12. ApproxPower(<x>, +1.0) ~= <x>, if x >= +0.0.
  13. ApproxPower(<x>, +INF) = +0.0, if -0.0 <= x < +1.0,
                               +INF, if x > +1.0,
  14. ApproxPower(<x>, -INF) = +INF, if -0.0 <= x < +1.0,
                               +0.0, if x > +1.0,
```

Note that 0^0 is defined here as NaN, since ApproxLog2(0) = -INF, and 0*(-INF) = NaN. In many other applications, including the standard C pow() function, 0^0 is defined as 1.0. This behavior can be emulated using additional instructions in much that same way that the pow() function is implemented on many CPUs.

Note that a logarithm is involved even if the exponent is an integer. This means that any exponentiating with a negative base will produce NaN. In constrast, it is possible in a "normal" mathematical formulation to raise negative numbers to integral powers (e.g., (-3)^2== 9, and (-0.5)^-2==4).

**Section 3.11.5.25, RCP: Reciprocal**

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxReciprocal(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 2.0.
```

The following special-case rules apply to reciprocation:

```
  1. ApproxReciprocal(NaN) = NaN.
  2. ApproxReciprocal(+INF) = +0.0.
  3. ApproxReciprocal(-INF) = -0.0.
  4. ApproxReciprocal(+0.0) = +INF.
  5. ApproxReciprocal(-0.0) = -INF.
```

**Section 3.11.5.26, RFL: Reflection Vector**

The RFL instruction computes the reflection of the second vector operand (the "direction" vector) about the vector specified by the first vector operand (the "axis" vector). Both operands are treated as 3D vectors (the w components are ignored). The result vector is another 3D vector (the "reflected direction" vector). The length of the result vector, ignoring rounding errors, should equal that of the second operand.

```
  axis = VectorLoad(op0);
  direction = VectorLoad(op1);
  tmp.w = (axis.x * axis.x + axis.y * axis.y +
           axis.z * axis.z);
  tmp.x = (axis.x * direction.x + axis.y * direction.y +
           axis.z * direction.z);
  tmp.x = 2.0 * tmp.x;
  tmp.x = tmp.x / tmp.w;
  result.x = tmp.x * axis.x - direction.x;
  result.y = tmp.x * axis.y - direction.y;
  result.z = tmp.x * axis.z - direction.z;
```

A fragment program will fail to load if the w component of the result is enabled in the component write mask (see the <optionalWriteMask> rule in the grammar).

**Section 3.11.5.27, RSQ: Reciprocal Square Root**

The RSQ instruction approximates the reciprocal of the square root of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxRSQRT(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 4.0.
```

The following special-case rules apply to reciprocal square roots:

```
  1. ApproxRSQRT(NaN) = NaN.
  2. ApproxRSQRT(+INF) = +0.0.
  3. ApproxRSQRT(-INF) = NaN.
  4. ApproxRSQRT(+0.0) = +INF.
  5. ApproxRSQRT(-0.0) = -INF.
  6. ApproxRSQRT(x) = NaN, if -INF < x < -0.0.
```

**Section 3.11.5.28, SEQ: Set on Equal To**

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SEQ:

```
  1. (<x> == <y>) and (<y> == <x>) always produce the same result.
  1. (NaN == <x>) is FALSE for all <x>, including NaN.
  2. (+INF == +INF) and (-INF == -INF) are TRUE.
  3. (-0.0 == +0.0) and (+0.0 == -0.0) are TRUE.
```

**Section 3.11.5.29, SFL: Set on False**

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

**Section 3.11.5.30, SGE: Set on Greater Than or Equal**

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SGE:

1. (NaN >= <x>) and (<x> >= NaN) are FALSE for all <x>.
2. (+INF >= +INF) and (-INF >= -INF) are TRUE.
3. (-0.0 >= +0.0) and (+0.0 >= -0.0) are TRUE.

**Section 3.11.5.31, SGT: Set on Greater Than**

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SGT:

1. (NaN > <x>) and (<x> > NaN) are FALSE for all <x>.
2. (-0.0 > +0.0) and (+0.0 > -0.0) are FALSE.

**Section 3.11.5.32, SIN: Sine**

The SIN instruction approximates the sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

The approximation function is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxSine(x) - sin(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxSine(NaN) = NaN.
2. ApproxSine(+/-INF) = NaN.
3. ApproxSine(+/-0.0) = +/-0.0. The sign of the result is equal to the sign of the single operand.

**Section 3.11.5.33, SLE: Set on Less Than or Equal**

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SLE:

1. (NaN <= <x>) and (<x> <= NaN) are FALSE for all <x>.
2. (+INF <= +INF) and (-INF <= -INF) are TRUE.
3. (-0.0 <= +0.0) and (+0.0 <= -0.0) are TRUE.

**Section 3.11.5.34, SLT: Set on Less Than**

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SLT:

1. (NaN < <x>) and (<x> < NaN) are FALSE for all <x>.
2. (-0.0 < +0.0) and (+0.0 < -0.0) are FALSE.

**Section 3.11.5.35, SNE: Set on Not Equal**

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
```

The following special-case rules apply to SNE:

1. (<x> != <y>) and (<y> != <x>) always produce the same result.
2. (NaN != <x>) is TRUE for all <x>, including NaN.
3. (+INF != +INF) and (-INF != -INF) are FALSE.
4. (-0.0 != +0.0) and (+0.0 != -0.0) are TRUE.

**Section 3.11.5.36, STR: Set on True**

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

**Section 3.11.5.37, SUB: Subtract**

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

The SUB instruction is completely equivalent to an identical ADD instruction in which the negate operator on the second operand is reversed:

1. "SUB R0, R1, R2" is equivalent to "ADD R0, R1, -R2".
2. "SUB R0, R1, -R2" is equivalent to "ADD R0, R1, R2".
3. "SUB R0, R1, |R2|" is equivalent to "ADD R0, R1, -|R2|".
4. "SUB R0, R1, -|R2|" is equivalent to "ADD R0, R1, |R2|".

**Section 3.11.5.38, TEX: Texture Lookup**

The TEX instruction performs a filtered texture lookup using the texture target given by <texImageTarget> belonging to the texture image unit given by <texImageUnit>. <texImageTarget> values of "1D", "2D", "3D", "CUBE", and "RECT" correspond to the texture targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, and TEXTURE_RECTANGLE_NV, respectively.

The (s,t,r) texture coordinates used for the lookup are the x, y, and z components of the single operand.

The texture lookup is performed as specified in Section 3.8. The LOD calculations in Section 3.8.5 are performed using an implementation dependent method to derive ds/dx, ds/dy, dt/dx, dt/dy, dr/dx, and dr/dy. The mapping of filtered texture components to the components of the result vector is dependent on the base internal format of the texture and is specified in Table X.5.

| Base Internal Format | Result Vector Components X | Y | Z | W |
|---|---|---|---|---|
| ALPHA | 0.0 | 0.0 | 0.0 | At |
| LUMINANCE | Lt | Lt | Lt | 1.0 |
| LUMINANCE_ALPHA | Lt | Lt | Lt | At |
| INTENSITY | It | It | It | It |
| RGB | Rt | Gt | Bt | 1.0 |
| RGBA | Rt | Gt | Bt | At |
| HILO_NV (signed) | HIt | LOt | HEMI | 1.0 |
| HILO_NV (unsigned) | HIt | LOt | 1.0 | 1.0 |
| DSDT_NV | DSt | DTt | 0.0 | 1.0 |
| DSDT_MAG_NV | DSt | DTt | MAGt | 1.0 |
| DSDT_MAG_INTENSITY_NV | DSt | DTt | MAGt | It |
| FLOAT_R_NV | Rt | 0.0 | 0.0 | 1.0 |
| FLOAT_RG_NV | Rt | Gt | 0.0 | 1.0 |
| FLOAT_RGB_NV | Rt | Gt | Bt | 1.0 |
| FLOAT_RGBA_NV | Rt | Gt | Bt | At |

**Table X.5: Mapping of filtered texel components to result vector components for the TEX instruction. 0.0 and 1.0 indicate that the corresponding constant value is written to the result vector. DEPTH_COMPONENT textures are treated as ALPHA, LUMINANCE, or INTENSITY, as specified in the texture's depth texture mode.**

**For HILO_NV textures with signed components, "HEMI" is defined as sqrt(MAX(0, 1-(HIt^2+LOt^2))).**

This instruction specifies a particular texture target, ignoring the standard hierarchy of texture enables (TEXTURE_CUBE_MAP_ARB, TEXTURE_3D, TEXTURE_2D, TEXTURE_1D) used to select a texture target in unextended OpenGL. If the specified texture target has a consistent set of images, a lookup is performed. Otherwise, the result of the instruction is the vector (0,0,0,0).

Although this instruction allows the selection of any texture target, a fragment program can not use more than one texture target for any given texture image unit.

**Section 3.11.5.39, TXD: Texture Lookup with Derivatives**

The TXD instruction performs a filtered texture lookup using the texture target given by <texImageTarget> belonging to the texture image unit given by <texImageUnit>. <texImageTarget> values of "1D", "2D", "3D", "CUBE", and "RECT" correspond to the texture targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, and TEXTURE_RECTANGLE_NV, respectively.

The (s,t,r) texture coordinates used for the lookup are the x, y, and z components of the first operand. The partial derivatives in the X direction (ds/dx, dt/dx, dr/dx) are specified by the x, y, and z components of the second operand. The partial derivatives in the Y direction (ds/dy, dt/dy, dr/dy) are specified by the x, y, and z components of the third operand.

The texture lookup is performed as specified in Section 3.8. The LOD calculations in Section 3.8.5 are performed using the specified partial derivatives. The mapping of filtered texture components to the components of the result vector is dependent on the base internal format of the texture and is specified in Table X.5.

This instruction specifies a particular texture target, ignoring the standard hierarchy of texture enables (TEXTURE_CUBE_MAP_ARB, TEXTURE_3D, TEXTURE_2D, TEXTURE_1D) used to select a texture target in unextended OpenGL. If the specified texture target has a consistent set of images, a lookup is performed. Otherwise, the result of the instruction is the vector (0,0,0,0).

Although this instruction allows the selection of any texture target, a fragment program can not use more than one texture target for any given texture image unit.

## Section 3.11.5.40, TXP: Projective Texture Lookup

The TXP instruction performs a filtered texture lookup using the texture target given by <texImageTarget> belonging to the texture image unit given by <texImageUnit>. <texImageTarget> values of "1D", "2D", "3D", "CUBE", and "RECT" correspond to the texture targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_ARB, and TEXTURE_RECTANGLE_NV, respectively.

For cube map textures, the (s,t,r) texture coordinates used for the lookup are given by x, y, and z, respectively. For all other textures, the (s,t,r) texture coordinates used for the lookup are given by x/w, y/w, and z/w, respectively, where x, y, z, and w are the corresponding components of the operand.

The texture lookup is performed as specified in Section 3.8. The LOD calculations in Section 3.8.5 are performed using an implementation dependent method to derive ds/dx, ds/dy, dt/dx, dt/dy, dr/dx, and dr/dy. The mapping of filtered texture components to the components of the result vector is dependent on the base internal format of the texture and is specified in Table X.5.

This instruction specifies a particular texture target, ignoring the standard hierarchy of texture enables (TEXTURE_CUBE_MAP_ARB, TEXTURE_3D, TEXTURE_2D, TEXTURE_1D) used to select a texture target in unextended OpenGL. If the specified texture target has a consistent set of images, a lookup is performed. Otherwise, the result of the instruction is the vector (0,0,0,0).

Although this instruction allows the selection of any texture target, a fragment program can not use more than one texture target for any given texture image unit.

**Section 3.11.5.41, UP2H: Unpack Two 16-Bit Floats**

The UP2H instruction unpacks two 16-bit floats stored together in a 32-bit scalar operand. The first 16-bit float (stored in the 16 least significant bits) is written into the "x" and "z" components of the result vector; the second is written into the "y" and "w" components of the result vector.

This operation undoes the type conversion and packing performed by the PK2H instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (fp16) (RawBits(tmp) & 0xFFFF);
  result.y = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
  result.z = (fp16) (RawBits(tmp) & 0xFFFF);
  result.w = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

**Section 3.11.5.42, UP2US: Unpack Two Unsigned 16-Bit Scalars**

The UP2US instruction unpacks two 16-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" and "z" components of the result vector are obtained from the 16 least significant bits of the operand; the "y" and "w" components are obtained from the 16 most significant bits.

This operation undoes the type conversion and packing performed by the PK2US instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.y = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
  result.z = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.w = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

**Section 3.11.5.43, UP4B: Unpack Four Signed 8-Bit Values**

The UP4B instruction unpacks four 8-bit signed values packed together in a 32-bit scalar operand. The signed quantities are encoded where a bit pattern of all '0' bits corresponds to -128/127 and a pattern of all '1' bits corresponds to +127/127. The "x" component of the result vector is the converted value corresponding to the 8 least significant bits of the operand; the "w" component corresponds to the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4B instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (((RawBits(tmp) >> 0) & 0xFF) - 128) / 127.0;
  result.y = (((RawBits(tmp) >> 8) & 0xFF) - 128) / 127.0;
  result.z = (((RawBits(tmp) >> 16) & 0xFF) - 128) / 127.0;
  result.w = (((RawBits(tmp) >> 24) & 0xFF) - 128) / 127.0;
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

**Section 3.11.5.44, UP4UB: Unpack Four Unsigned 8-Bit Scalars**

The UP4UB instruction unpacks four 8-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" component of the result vector is obtained from the 8 least significant bits of the operand; the "w" component is obtained from the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4UB instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFF) / 255.0;
  result.y = ((RawBits(tmp) >> 8)  & 0xFF) / 255.0;
  result.z = ((RawBits(tmp) >> 16) & 0xFF) / 255.0;
  result.w = ((RawBits(tmp) >> 24) & 0xFF) / 255.0;
```

Since the source operand must be a 32-bit scalar, a fragment program will fail to load if the operand is not obtained from a register with 32-bit components or from a program parameter.

**Section 3.11.5.45, X2D: 2D Coordinate Transformation**

The X2D instruction multiplies the 2D offset vector specified by the "x" and "y" components of the second vector operand by the 2x2 matrix specified by the four components of the third vector operand, and adds the transformed offset vector to the 2D vector specified by the "x" and "y" components of the first vector operand. The first component of the sum is written to the "x" and "z" components of the result; the second component is written to the "y" and "w" components of the result.

The X2D instruction can be used to displace texture coordinates in the same manner as the OFFSET_TEXTURE_2D_NV mode in the GL_NV_texture_shader extension.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.y = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
  result.z = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.w = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
```

**Section 3.11.6, Fragment Program Outputs**

Upon completion of fragment program execution, the output registers are used to replace the fragment's associated data.

For color fragment programs, the RGBA color of the fragment is taken from the output register (COLR or COLH). The R, G, B, and A color components are extracted from the "x", "y", "z", and "w" components, respectively, of the output register and are clamped to the range [0,1].

For combiner fragment programs, register combiner operations (as described in the NV_register_combiners specification) are then performed, regardless of the state of the REGISTER_COMBINERS_NV enable. The RGBA texture colors corresponding the TEXTURE0_ARB, TEXTURE1_ARB, TEXTURE2_ARB, and TEXTURE3_ARB combiner registers are taken from the TEX0, TEX1, TEX2, and TEX3 output registers, respectively. Any components of the TEX0, TEX1, TEX2, or TEX3 output registers that are not written to by the fragment program are undefined. The R, G, B, and A texture color components are extracted from the "x", "y", "z", and "w" output register components, respectively, and are clamped to the range [-1,1].

If the DEPR output register is written by the fragment program, the depth value of the fragment is taken from the z component of the DEPR output register. If depth clamping is enabled, the depth value is clamped to the range [min(n,f), max(n,f)], where n and f are the near and far depth range values. If depth clamping is disabled, the fragment is discarded if its depth value is outside the range [min(n,f), max(n,f)].

**Section 3.11.7, Required Fragment Program State**

The state required for managing fragment programs consists of:

a bit indicating whether or not fragment program mode is enabled;

an unsigned integer naming the currently bound fragment program

and the state that must be maintained to indicate which integers are currently in use as fragment program names.

Fragment program mode is initially disabled. The initial state of all 128 fragment program parameter registers is (0,0,0,0). The initial currently bound fragment program is zero.

Each fragment program object consists of:

an enumerant given the program target (FRAGMENT_PROGRAM_NV);

a boolean indicating whether the program is resident;

an array of type ubyte containing the program string;

an integer representing the length of the program string array;

one four-component floating-point vector for each named local parameter in the program;

and a set of MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV four-component floating-point vectors to hold numbered local parameters, each initially set to (0,0,0,0).

Initially, no program objects exist.

Additionally, the state required during the execution of a fragment program consists of: twelve 4-component floating-point fragment attribute registers, thirty-two 128-bit physical temporary registers, and a single 4-component condition code, whose components have one of four values (LT, EQ, GT, or UN).

Each time a fragment program is executed, the fragment attribute registers are initialized with the fragment's location and associated data, all temporary register components are initialized to zero, and all condition code components are initialized to EQ.

*Renumber Section 3.11 to Section 3.12, Antialiasing Application (p.140). No changes to the text of the section.*

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

None

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

### Add new section 5.7, Programs (after "Flush and Finish")

Programs are specified as an array of ubytes used to control the operation of portions of the GL. The array is a string of ASCII characters encoding the program.

The command

```
  LoadProgramNV(enum target, uint id, sizei len, const ubyte *program);
```

loads a program. The target parameter specifies the type of program loaded and can be VERTEX_PROGRAM_NV, VERTEX_STATE_PROGRAM_NV, or FRAGMENT_PROGRAM_NV. VERTEX_PROGRAM_NV specifies a program to be executed in vertex program mode as each vertex is specified. VERTEX_STATE_PROGRAM specifies a program to be run manually to update vertex state. FRAGMENT_PROGRAM specifies a program to be executed in fragment program mode as each fragment is rasterized.

Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the set of positive integers (zero is reserved). The error INVALID_VALUE is generated by LoadProgramNV if a program is loaded with an id of zero. The error INVALID_OPERATION is generated by LoadProgramNV or if a program is loaded for an id that is currently loaded with a program of a different program target. program is a pointer to an array of ubytes that represents the program being loaded. The length of the array in ubytes is indicated by len.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array. Tokens are processed in a case-sensitive manner: upper and lower-case letters are not considered equivalent.

Each program target has a corresponding Backus-Naur Form (BNF) grammar specifying the syntactically valid sequences for programs of the specified type. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

The error INVALID_OPERATION is generated by LoadProgramNV if a program fails to load because it is not syntactically correct or fails to satisfy all of the semantic restrictions corresponding to the program target.

A successfully loaded program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions is specific to the program target and is defined elsewhere.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the program string most recently passed to LoadProgramNV indicating the position of the first error, if any, in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

For targets whose programs are executed automatically (e.g., vertex and fragment programs), there must be a current program. The current vertex program is executed automatically in vertex program mode as vertices are specified. The current fragment program is executed automatically in fragment program mode as fragments are generated by rasterization. Current programs for a program target are updated by

```
  BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV or FRAGMENT_PROGRAM_NV. The error INVALID_OPERATION is generated by BindProgramNV if id names a program that has a type different than target (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error. However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

The INVALID_OPERATION error is generated when both fragment program mode is enabled and Begin, another GL command that performs an implicit Begin, or any other GL command that generates fragments is called, if the current fragment program is nonexistent or not valid. A fragment program may be invalid for reasons explained in Section 3.11.3.

Programs are deleted by calling

```
  void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
  void GenProgramsNV(sizei n, uint *ids);
```

returns n currently unused program names in ids.  These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV.

An implementation may choose to establish a working set of programs on which binding and/or manual execution are performed with higher performance.  A program that is currently part of this working set is said to be resident.

The command

```
  boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set.  If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences.  Otherwise the contents of residences are not changed.  If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate.  The residence status of a single named program can also be queried by calling GetProgramivNV (Section 6.1.13) with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident.  An implementation may choose to make a program resident only on first use, for example.  The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
  void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible.  As a result of making the requested programs resident, program names not among the requested programs may become non-resident.  Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero.  AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

The commands

```
  void ProgramNamedParameter4fNV(uint id, sizei len, const ubyte *name,
                                 float x, float y, float z, float w);
  void ProgramNamedParameter4dNV(uint id, sizei len, const ubyte *name,
                                 double x, double y, double z, double w);
  void ProgramNamedParameter4fvNV(uint id, sizei len, const ubyte *name,
                                  const float v[]);
  void ProgramNamedParameter4dvNV(uint id, sizei len, const ubyte *name,
                                  const double v[]);
```

specify a new value for the named program local parameter <name> belonging to the fragment program specified by <id>. <name> is a pointer to an array of ubytes holding the parameter name. <len> specifies the number of ubytes in the array given by <name>. The new x, y, z, and w components of the named local parameter are given by x, y, z, and w, respectively, for ProgramNamedParameter4fNV and ProgramNamedParameter4dNV, and by v[0], v[1], v[2], and v[3], respectively, for ProgramNamedParameter4fvNV and ProgramNamedParameter4dvNV. The error INVALID_OPERATION is generated if <id> specifies a nonexistent program or a program whose type does not suport named local parameters. The error INVALID_VALUE error is generated if <name> does not specify the name of a local parameter in the program corresponding to <id>. The error INVALID_VALUE is also generated if <len> is zero.

The commands

```
  void ProgramLocalParameter4fARB(enum target, uint index,
                                  float x, float y, float z, float w);
  void ProgramLocalParameter4fvARB(enum target, uint index,
                                   const float *params);
  void ProgramLocalParameter4dARB(enum target, uint index,
                                  double x, double y, double z, double w);
  void ProgramLocalParameter4dvARB(enum target, uint index,
                                   const double *params);
```

update the values of the numbered program local parameter <index> belonging to the program object currently bound to <target>. For ProgramLocalParameter4fARB and ProgramLocalParameter4dARB, the four components of the parameter are updated with the values of <x>, <y>, <z>, and <w>, respectively. For ProgramLocalParameter4fvARB and ProgramLocalParameter4dvARB, the four components of the parameter are updated with the array of four values pointed to by <params>. The error INVALID_VALUE is generated if <index> is greater than or equal to the number of numbered program local parameters supported by <target>.

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

### Modify Section 6.1.11, Pointer and String Queries (p. 206)

(modify last paragraph, p. 206) ... The possible values for <name> are VENDOR, RENDERER, VERSION, EXTENSIONS, and PROGRAM_ERROR_STRING_NV.

(add after last paragraph of section, p. 207) Queries of PROGRAM_ERROR_STRING_NV return a pointer to an implementation-dependent program load error string. If the last call to LoadProgramNV failed to load a program, the returned string describes a reason that the program failed to load. Otherwise, a pointer to an empty string (containing only a terminator) is returned.

Rename and modify Section 6.1.13, Vertex and Fragment Program Queries (from GL_NV_fragment_program). Portions of this section pertaining to fragment programs are copied verbatim.

(insert after discussion of GetProgramParameter[fd]vNV)

The commands

```
  void GetProgramNamedParameterfvNV(uint id, sizei len,
                                    const ubyte *name, float *params);
  void GetProgramNamedParameterdvNV(uint id, sizei len,
                                    const ubyte *name, double *params);
```

obtain the current program named local parameter value for the parameter named <name> belonging to the program given by <id>. <name> is a pointer to an array of ubytes holding the parameter name. <len> specifies the number of ubytes in the array given by <name>. The error INVALID_OPERATION is generated if <id> specifies a nonexistent program or a program whose type does not suport named local parameters. The error INVALID_VALUE is generated if <name> does not specify the name of a local parameter in the program corresponding to <id>. The error INVALID_VALUE is also generated if <len> is zero. Each named program local parameter is an array of four values.

The commands

```
  void GetProgramLocalParameterdvARB(enum target, uint index,
                                     double *params);
  void GetProgramLocalParameterfvARB(enum target, uint index,
                                     float *params);
```

obtain the current value for the numbered program local parameter <index> belonging to the program object currently bound to <target>, and places the information in the array <params>. The error INVALID_ENUM is generated if <target> specifies a nonexistent program target or a program target that does not support numbered program local parameters. The error INVALID_VALUE is generated if <index> is greater than or equal to the implementation-dependent number of supported numbered program local parameters for the program target.

When the program target type is FRAGMENT_PROGRAM_NV, each numbered program local parameter returned is an array of four values. ...

The command

```
  void GetProgramivNV(uint id, enum pname, int *params);
```

obtains program state named by pname for the program named id in the array params. pname must be one of PROGRAM_TARGET_NV, PROGRAM_LENGTH_NV, or PROGRAM_RESIDENT_NV. The error INVALID_OPERATION is generated if the program named id does not exist.

The command

```
  void GetProgramStringNV(uint id, enum pname,
                          ubyte *program);
```

obtains the program string for program id. pname must be PROGRAM_STRING_NV. n ubytes are returned into the array program where n is the length of the program in ubytes. GetProgramivNV with PROGRAM_LENGTH_NV can be used to query the length of a program's string. The INVALID_OPERATION error is generated if the program named id does not exist.

...

The command

```
  boolean IsProgramNV(uint id);
```

returns TRUE if program is the name of a program object. If program is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramNV returns FALSE. A name returned by GenProgramsNV but not yet loaded with a program is not the name of a program object."

**Additions to Appendix F of the OpenGL 1.2.1 Specification (ARB Extensions)**

**Modify Section F.2.3 (Changes to Section 2.6), p.240**

(modify last paragraph on p.240) ... Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture coordinate sets supported is implementation dependent, but must be at least 1. The number of texture coordinate sets supported may be queried with the state MAX_TEXTURE_COORDS_NV.

**Modify Section F.2.4 (Changes to Section 2.7), p.241**

(modify the last paragraph on p.241, carrying over to p.243) Implementations may support more than one set of texture coordinates. The commands

```
    void MultiTexCoord{1234}{sifd}ARB(enum texture, T coords)
    void MultiTexCoord{1234}{sifd}vARB(enum texture, T coords)
```

take the coordinate set to be modified as the <texture> parameter. <texture> is a symbolic constant of the form TEXTUREi_ARB, indicating that texture coordinate set i is to be modified. The constants obey TEXTUREi_ARB = TEXTURE0_ARB + i (i is in the range 0 to k-1, where k is the implementation dependent number of texture units defined by MAX_TEXTURE_COORDS_NV).

**Modify Section F.2.5 (Changes to Section 2.8), p.243**

(modify first and second paragraphs of section) ... The client may specify up to 5 plus the value of MAX_TEXTURE_COORDS_NV arrays; one each to store vertex coordinates...

In implementations which support more than one texture coordinate set, the command

```
    void ClientActiveTextureARB(enum texture)
```

is used to select the vertex array client state parameters to be modified by the TexCoordPointer command and the array affected by EnableClientState and DisableClientState with the parameter TEXTURE_COORD_ARRAY. This command sets the state variable CLIENT_ACTIVE_TEXTURE_ARB. Each texture coordinate set has a client state vector which is selected when this command is invoked. This state vector also includes the vertex array state. This command also selects the texture coordinate set state used for queries of client state.

(modify first paragraph on p.244) If the number of supported texture coordinate sets (the value of MAX_TEXTURE_COORDS_NV) is k, ...

**Modify Section F.2.6 (Changes to Section 2.10.2), p.244**

(modify first paragraph) For each texture coordinate set, a 4x4 matrix is applied to the corresponding texture coordinates...

(replace second and third paragraphs) The command

```
  void ActiveTextureARB(enum texture);
```

specifies the active texture unit selector, ACTIVE_TEXTURE_ARB. Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit (consisting of a texture matrix stack and texture coordinate generation state) and a texture image unit (consisting of all the texture state defined in Section 3.8). In implementations with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector specifies the texture unit accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX_MODE is TEXTURE), TexGen (Section 2.10.4), Enable/Disable (if any texture coordinate generation enum is selected), as well as queries of the current texture coordinates and current raster texture coordinates. If the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_COORD_SETS_NV, the error INVALID_OPERATION is generated by any such command.

The active texture unit selector also selects the texture unit accessed by commands involving texture image processing (Section 3.8). Such commands include all variants of TexEnv, TexParameter, and TexImage commands, BindTexture, Enable/Disable for any texture target (e.g., TEXTURE_2D), and queries of all such state. If the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_IMAGE_UNITS_NV, the error INVALID_OPERATION is generated by any such command.

ActiveTextureARB generates the error INVALID_ENUM if an invalid <texture> is specified. <texture> is a symbolic constant of the form TEXTUREi_ARB, indicating that texture unit i is to be modified. The constants obey

TEXTUREi_ARB = TEXTURE0_ARB + i (i is in the range 0 to k-1, where k is the larger of the MAX_TEXTURE_COORDS_NV and MAX_TEXTURE_IMAGE_UNITS_NV). For compatibility with old OpenGL specifications, the implementation dependent constant MAX_TEXTURE_UNITS_ARB specifies the number of conventional texture units supported by the implementation. Its value must be no larger than the minimum of MAX_TEXTURE_COORDS_NV and MAX_TEXTURE_IMAGE_UNITS_NV.

**Modify Section F.2.12 (Changes to Section 3.8.10), p.249**

(modify next-to-last paragraph) Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by MAX_TEXTURE_UNITS_ARB may be incomplete and are always treated as disabled.

**Modify Section F.2.15 (Changes to Section 6.1.2), p.251**

(add to end of paragraph) Queries of texture state variables corresponding to texture coordinate processing unit (namely, TexGen state and enables, and matrices) will produce an INVALID_OPERATION error if the value of ACTIVE_TEXTURE_ARB is greater than or equal to MAX_TEXTURE_COORDS_NV. All other texture state queries will result in an INVALID_OPERATION error if the value of ACTIVE_TEXTURE_ARB is greater than or equal to MAX_TEXTURE_IMAGE_UNITS_NV.

## Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

## Dependencies on GL_NV_vertex_program

If NV_vertex_program is supported, the description of LoadProgramNV in Section 2.14.1.7 (up to the BNF description of vertex programs) is deleted, as it is replaced by the contents of Section 5.7 in this specification. The general error descriptions in Section 2.14.1.7 common to Section 5.7 (like INVALID_OPERATION if the program fails to compile) should also be deleted. Section 2.14.1.8 should also be deleted. Section 6.1.13 is modified by this specification as described above.

## Dependencies on NV_register_combiners

If NV_register_combiners is not supported, combiner programs are not supported, the TEX0, TEX1, TEX2, and TEX3 output registers are eliminated, and all references to both in this extension are deleted.

## Dependencies on NV_texture_shader

If NV_texture_shader is not supported, the comment about texture shaders being disabled in fragment program mode is not applicable.

**Dependencies on NV_texture_rectangle**

If NV_texture_rectangle is not supported, the references to "RECT" in the <texImageTarget> grammar rule and TEXTURE_RECTANGLE_NV are not applicable.

**Dependencies on ARB_texture_cube_map**

If NV_texture_rectangle is not supported, the references to "CUBE" in the <texImageTarget> grammar rule and TEXTURE_CUBE_MAP_ARB are not applicable.

**Dependencies on EXT_fog_coord**

If EXT_fog_coord is not supported, references to "fog coordinate" in the definition of the "FOGC" fragment attribute register should be removed.

**Dependencies on NV_depth_clamp**

If NV_depth_clamp is not supported, section 3.11.6 is modified to remove discussion of the depth clamp enable and instead indicate that fragments with depth values outside [min(n,f), max(n,f)] are always discarded.

**Dependencies on ARB_depth_texture and SGIX_depth_texture**

If ARB_depth_texture is not supported, but SGIX_depth_texture is supported, the discussion of Table X.5 is modified to indicate that DEPTH_COMPONENT textures are treated as LUMINANCE.

If neither extension is supported, the discussion of DEPTH_COMPONENT textures in Table X.5 should be removed.

**Dependencies on NV_float_buffer**

If NV_float_buffer is not supported, references to FLOAT_R_NV, FLOAT_RG_NV, FLOAT_RGB_NV, and FLOAT_RGBA_NV internal texture formats in Table X.5 should be removed.

**Dependencies on ARB_vertex_program**

This extension does not have any explicit dependencies, but the APIs for setting and querying numbered local parameters (ProgramLocalParameter*ARB and GetProgramLocalParameter*ARB) were taken directly from this extension,

**GLX Protocol**

Most of the GLX protocol needed to implement this extension is described in the GL_NV_vertex_program extension specification and will not be repeated here.

```
    The following two rendering commands are potentially large, and hence can
    be sent in a glXRender or glXRenderLarge request.

        ProgramNamedParameter4fvNV
            2           28+len+p        rendering command length
            2           4218            rendering command opcode
            4           CARD32          id
            4           CARD32          len
            4           FLOAT32         params[0]
            4           FLOAT32         params[1]
            4           FLOAT32         params[2]
            4           FLOAT32         params[3]
            len         LISTofCARD8     name
            p                           unused, p=pad(len)

         If the command is encoded in a glxRenderLarge request, the command
         opcode and command length fields above are expanded to 4 bytes each:

            4           32+len+p        rendering command length
            4           4218            rendering command opcode

        ProgramNamedParameter4dvNV
            2           44+len+p        rendering command length
            2           4219            rendering command opcode
            4           CARD32          id
            4           CARD32          len
            8           FLOAT64         params[0]
            8           FLOAT64         params[1]
            8           FLOAT64         params[2]
            8           FLOAT64         params[3]
            len         LISTofCARD8     name
            p                           unused, p=pad(len)

         If the command is encoded in a glxRenderLarge request, the command
         opcode and command length fields above are expanded to 4 bytes each:

            4           48+len+p        rendering command length
            4           4219            rendering command opcode
```

```
The remaining two commands are non-rendering commands.  These commands are
sent separately (i.e., not as part of a glXRender or glXRenderLarge
request), using the glXVendorPrivateWithReply request:
```

**GetProgramNamedParameter4fvNV**

```
      1           CARD8           opcode (X assigned)
      1           17              GLX opcode (glXVendorPrivateWithReply)
      2           4+(len+p)/4     request length
      4           1310            vendor specific opcode
      4           GLX_CONTEXT_TAG context tag
      4           INT32           len
      len         LISTofCARD8     name
      p                           unused, p=pad(len)
    =>

    If the command succeeds, 4 floats are sent in the reply:

      1           1               reply
      1                           unused
      2           CARD16          sequence number
      4           4               reply length
      24                          unused
      16          LISTofFLOAT32   params

    Otherwise, an empty reply is sent, indicating that a GL error
    occured:

      1           1               reply
      1                           unused
      2           CARD16          sequence number
      4           0               reply length
      24                          unused
```

```
        GetProgramNamedParameter4dvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4+(len+p)/4     request length
            4           1311            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           len
            len         LISTofCARD8     name
            p                           unused, p=pad(len)
          =>

          If the command succeeds, 4 doubles are sent in the reply:

            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           8               reply length
            24                          unused
            32          LISTofFLOAT64   params

          Otherwise, an empty reply is sent, indicating that a GL error
          occured:

            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            24                          unused
```

**Errors**

INVALID_OPERATION is generated by Begin, DrawPixels, Bitmap, CopyPixels, or a command that performs an explicit Begin if FRAGMENT_PROGRAM_NV is enabled and the currently bound fragment program does not exist.

INVALID_OPERATION is generated by ProgramNamedParameter4fNV, ProgramNamedParameter4dNV, ProgramNamedParameter4fvNV, ProgramNamedParameter4dvNV, GetProgramNamedParameterfvNV, or GetProgramNamedParameterdvNV if <id> specifies a nonexistent program or a program whose type does not suport local parameters.

INVALID_VALUE is generated by ProgramNamedParameter4fNV, ProgramNamedParameter4dNV, ProgramNamedParameter4fvNV, ProgramNamedParameter4dvNV, GetProgramNamedParameterfvNV, or GetProgramNamedParameterdvNV if <len> is zero.

INVALID_VALUE is generated by ProgramNamedParameter4fNV, ProgramNamedParameter4dNV, ProgramNamedParameter4fvNV, ProgramNamedParameter4dvNV, GetProgramNamedParameterfvNV, or GetProgramNamedParameterdvNV if <name> does not specify the name of a local parameter in the program corresponding to <id>.

INVALID_OPERATION is generated by any command accessing texture coordinate processing state if the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_COORD_SETS_NV.

INVALID_OPERATION is generated by any command accessing texture image processing state if the texture unit number corresponding to the current value of ACTIVE_TEXTURE_ARB is greater than or equal to the implementation dependent constant MAX_TEXTURE_IMAGE_UNITS_NV.

*(The following are error descriptions copied from GL_NV_vertex_program that apply to this extension as well. These modifications do not affect the behavior of that extension.)*

INVALID_VALUE is generated by LoadProgramNV if id is zero.

INVALID_OPERATION is generated by LoadProgramNV if the program corresponding to id is currently loaded but has a program type different from that given by target.

INVALID_OPERATION is generated by LoadProgramNV if the program specified is syntactically incorrect for the program type specified by target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

INVALID_OPERATION is generated by LoadProgramNV if the problem specified fails to conform to any of the semantic restrictions imposed on programs of the type specified by target. The value of PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

INVALID_OPERATION is generated by BindProgramNV if target does not match the type of the program named by id.

INVALID_VALUE is generated by AreProgramsResidentNV if any of the queried programs are zero or do not exist.

INVALID_OPERATION is generated by GetProgramivNV or GetProgramStringNV if the program named id does not exist.

**New State**

| Get Value | Type | Get Command | Initial Value | Description | Section | Attribute |
|---|---|---|---|---|---|---|
| FRAGMENT_PROGRAM_NV | B | IsEnabled | FALSE | fragment program mode enable | 3.11 | enable |
| FRAGMENT_PROGRAM_BINDING_NV | Z+ | GetIntegerv | 0 | bound fragment program | 5.7 | - |

**Table X.6. New State Introduced by NV_fragment_program.**

| Get Value | Type | Get Command | Initial Value | Description | Section | Attribute |
|---|---|---|---|---|---|---|
| PROGRAM_ERROR_POSITION_NV | Z | GetIntegerv | -1 | program error position | 5.7 | - |
| PROGRAM_TARGET_NV | Z2 | GetProgramivNV | 0 | program target | 6.1.13 | - |
| PROGRAM_LENGTH_NV | Z+ | GetProgramivNV | 0 | program length | 6.1.13 | - |
| PROGRAM_RESIDENT_NV | Z2 | GetProgramivNV | False | program residency | 6.1.13 | - |
| PROGRAM_STRING_NV | ubxn | GetProgramStringNV | "" | program string | 6.1.13 | - |
| - | nxR4 | GetProgramNamed-ParameterNV | (0,0,0,0) | named program local parameter value | 5.7 | - |
| - | 64+xR4 | GetProgramLocal-ParameterARB | (0,0,0,0) | numbered program local parameter | 5.7 | - |

**Table X.7. Program Object State common to NV_vertex_program and NV_fragment_program.**

| Get Value | Type | Get Command | Initial Value | Description | Section | Attribute |
|---|---|---|---|---|---|---|
| - | 12xR4 | - | fragment data | fragment attribute registers | 3.11.1.1 | - |
| - | 16xR4 | - | (0,0,0,0) | fp32 temporary registers | 3.11.1.2 | - |
| - | 32xR4 | - | (0,0,0,0) | fp16 temporary registers | 3.11.1.2 | - |
| | (Z_4)4 | - | (EQ,EQ,EQ,EQ) | condition code register address register | 3.11.1.4 | - |

**Table X.8. Fragment Program Per-Fragment Execution State.**

## New Implementation Dependent State

| Get Value | Type | Get Command | Minimum Value | Description | Section | Attribute |
|---|---|---|---|---|---|---|
| MAX_TEXTURE_COORDS_NV | Z+ | GetIntegerv | 2 | number of texture coordinate sets supported | 2.6 | - |
| MAX_TEXTURE_IMAGE_UNITS_NV | Z+ | GetIntegerv | 2 | number of texture image units supported | 2.10.2 | - |
| MAX_FRAGMENT_PROGRAM_ LOCAL_PARAMETERS_NV | Z+ | GetIntegerv | 64 | number of numbered local parameters supported | 3.11.7 | - |

**Name**

    NV_fragment_program_option

**Name Strings**

    GL_NV_fragment_program_option

**Status**

    Shipping.

**Version**

```
    Last Modified:      05/16/2004
    NVIDIA Revision:    2
```

**Number**

    Unassigned

**Dependencies**

    ARB_fragment_program is required.

    NV_fragment_program is required.

**Overview**

```
    This 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_program;

    to the beginning of their fragment programs.

    The functionality provided by this extension, which is roughly
    equivalent to that provided by the NV_fragment_program extension,
    includes:

      * increased control over precision in arithmetic computations and
        storage,

      * data-dependent conditional writemasks,

      * an absolute value operator on scalar and swizzled operand loads,

      * instructions to compute partial derivatives, and perform texture
        lookups using specified partial derivatives,

      * fully orthogonal "set on" instructions,

      * instructions to compute reflection vector and perform a 2D
        coordinate transform, and
```

* instructions to pack and unpack multiple quantities into a single
  component.

**Issues**

*Why is this a separate extension, rather than just an additional feature of NV_fragment_program?*

RESOLVED: The NV_fragment_program specification was complete (with a published implementation) prior to the completion of ARB_fragment_program. Future NVIDIA fragment program extensions should contain extensions to the ARB_fragment_program execution environment as a standard feature.

*Should a similar option be provided to expose ARB_fragment_program features not found in NV_fragment_program (e.g., state bindings, certain "macro" instructions) under the NV_fragment_program interface?*

RESOLVED: No. Why not just write an ARB program?

*The ARB_fragment_program spec has a minor grammar bug that requires that inline scalar constants used as scalar operands include a component selector. In other words, you have to say "11.0.x" to use the constant "11.0". What should we do here?*

RESOLVED: The NV_fragment_program_option grammar will correct this problem, which should be fixed in future revisions to the ARB language.

**New Procedures and Functions**

None.

**New Tokens**

None.

**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 3.11.2 of ARB_fragment_program (Fragment Program Grammar and Restrictions):**

(mostly add to existing grammar rules, modify a few existing grammar rules -- changes marked with "***")

```
<optionName>            ::= "NV_fragment_program"

<TexInstruction>        ::= <TXDop_instruction>
```

```
    <VECTORop>              ::= "DDX"
                              | "DDY"
                              | "PK2H"
                              | "PK2US"
                              | "PK4B"
                              | "PK4UB"

    <SCALARop>              ::= "UP2H"
                              | "UP2US"
                              | "UP4B"
                              | "UP4UB"

    <BINop>                 ::= "RFL"
                              | "SEQ"
                              | "SFL"
                              | "SGT"
                              | "SLE"
                              | "SNE"
                              | "STR"

    <TRIop>                 ::= "X2D"

    <TXDop_instruction>     ::= <TXDop> <instResult> "," <instOperandV> ","
                                <instOperandV> "," <instOperandV> ","
                                <texTarget>

    <TXDop>                 ::= "TXD"

    <killCond>              ::= <ccTest>

    <instOperandV>          ::= <instOperandAbsV>

    <instOperandAbsV>       ::= <optSign> "|" <instOperandBaseV> "|"

    <instOperandS>          ::= <instOperandAbsS>

    <instOperandAbsS>       ::= <optSign> "|" <instOperandBaseS> "|"

    <instResult>            ::= <instResultCC>

    <instResultCC>          ::= <instResultBase> <ccMask>

    <TEMP_statement>        ::= <varSize> "TEMP" <varNameList>

    <OUTPUT_statement>      ::= <varSize> "OUTPUT" <establishName> "="
                                  <resultUseD>

    <varSize>               ::= "SHORT"
                              | "LONG"

    <paramUseV>             ::= <constantScalar>
                                  (*** instead of <constantScalar>
                                       <swizzleSuffix>)

    <paramUseS>             ::= <constantScalar>
                                  (*** instead of <constantScalar>
                                       <scalarSuffix>)
```

```
<ccMask>                ::= "(" <ccTest> ")"

<ccTest>                ::= <ccMaskRule> <swizzleSuffix>

<ccMaskRule>            ::= "EQ"
                          | "GE"
                          | "GT"
                          | "LE"
                          | "LT"
                          | "NE"
                          | "TR"
                          | "FL"
```

(modify language describing reserved keywords) The following strings are reserved keywords and may not be used as identifiers:

    ALIAS, ATTRIB, END, OPTION, OUTPUT, PARAM, TEMP, fragment,
    program, result, state, and texture.

Additionally, all the instruction names (and variants) listed in Table X.5 are reserved.

**Modify Section 3.11.3.3, Fragment Program Temporaries**

(replace second paragraph) Fragment program temporary variables can be declared explicitly using the <TEMP_statement> grammar rule. Each such statement can declare one or more temporaries. Temporary declaration can optionally specify a variable size, using the <varSize> grammar rule. Variables declared as "SHORT" will represented with at least 16 bits per component (5 bits of exponent, 10 bits of mantissa). Variables declared as "LONG" will be represented with at least 32 bits per component (8 bits of exponent, 23 bits of mantissa). Fragment program temporary variables can not be declared implicitly.

**Modify Section 3.11.3.4, Fragment Program Results**

(replace second paragraph) Fragment program result variables can be declared explicitly using the <OUTPUT_statement> grammar rule, or implicitly using the <resultBinding> grammar rule in an executable instruction. Explicit result variable declaration can optionally specify a variable size, using the <varSize> grammar rule. Variables declared as "SHORT" will represented with at least 16 bits per component (5 bits of exponent, 10 bits of mantissa). Variables declared as "LONG" will be represented with at least 32 bits per component (8 bits of exponent, 23 bits of mantissa). Each fragment program result variable is bound to a fragment attribute used in subsequent back-end processing. The set of fragment program result variable bindings is given in Table X.3.

(add to the end of a section) A fragment program will fail to load if contains instructions writing to variables bound to the same result, but declared with different sizes.

**Add New Section 3.11.3.X, Condition Code Register (insert after Section 3.11.3.4, Fragment Program Results)**

The fragment program condition code register is a single four-component vector. 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 fragment program instructions can optionally update the condition code register. When a fragment program instruction updates the 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 register is initialized to a vector of EQ values each time a fragment program executes.

**Modify Section 3.11.4, Fragment Program Execution Environment**

(modify instruction table) There are fifty-two fragment program instructions. Fragment program instructions may have up to sixteen variants, including a suffix of "R", "H", or "X" to specify arithmetic precision (section 3.11.4.X), a suffix of "C" to allow an update of the condition code register (section 3.11.3.X), and a suffix of "_SAT" to clamp the result vector components to the range [0,1] (section 3.11.4.3). For example, the sixteen forms of the "ADD" instruction are "ADD", "ADDR", "ADDH", "ADDX", "ADDC", "ADDRC", "ADDHC", "ADDXC", "ADD_SAT", "ADDR_SAT", "ADDH_SAT", "ADDX_SAT", "ADDC_SAT", "ADDRC_SAT", "ADDHC_SAT", and "ADDXC_SAT".The instructions and their respective input and output parameters are summarized in Table X.5.

| | Modifiers | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Instr.** | **R** | **H** | **X** | **C** | **S** | **Inputs** | **Output** | **Description** |
| ABS | X | X | X | X | X | v | v | absolute value |
| ADD | X | X | X | X | X | v,v | v | add |
| CMP | - | - | - | - | 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 |
| 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 |
| 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 |
| 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 |
| 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 |

| Instr. | Modifiers R | H | X | C | S | Inputs | Output | Description |
|---|---|---|---|---|---|---|---|---|
| MOV | X | X | X | X | X | v | v | move |
| MUL | X | X | X | X | X | v,v | v | multiply |
| 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 |
| POW | X | X | - | X | X | s,s | ssss | exponentiate |
| RCP | X | X | - | X | X | s | ssss | reciprocal |
| RFL | X | X | - | X | X | v | v | reflection vector |
| RSQ | X | X | - | X | X | s | ssss | reciprocal square root |
| SCS | - | - | - | - | 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 | 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/partials |
| 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 | 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" saturation modifier, 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 3.11.4.1, Fragment Program Operands**

(add prior to the discussion of negation) A component-wise absolute value operation can optionally performed on the operand if the operand is surrounded with two "|" characters. For example, "|src|" indicates that a component-wise absolute value operation should be performed on the variable named "src". In terms of the grammar, this operation is performed if the <instOperandV> or <instOperandS> grammar rules match <instOperandAbsV> or <instOperandAbsS>, respectively.

(modify operand load pseudo-code) The following pseudo-code spells out the operand generation process. In the example, "float" is a floating-point scalar type, while "floatVec" is a four-component vector. "source" refers to the register used for the operand, matching the <srcReg> rule. "abs" is TRUE if an absolute value operation should be performed on the operand (<instOperandAbsV> or <instOperandAbsS> rules) "negate" is TRUE if the <optionalSign> rule in <scalarSrcReg> or <swizzleSrcReg> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation; the ".c" modifier refers to the single component selected for a scalar load.

```
floatVec VectorLoad(floatVec source)
{
    floatVec operand;

    operand.x = source.c***;
    operand.y = source.*c**;
    operand.z = source.**c*;
    operand.w = source.***c;
    if (abs) {
       operand.x = abs(operand.x);
       operand.y = abs(operand.y);
       operand.z = abs(operand.z);
       operand.w = abs(operand.w);
    }
    if (negate) {
       operand.x = -operand.x;
       operand.y = -operand.y;
       operand.z = -operand.z;
       operand.w = -operand.w;
    }

    return operand;
}

float ScalarLoad(floatVec source)
{
    float operand;

    operand = source.c;
    if (abs) {
      operand = abs(operand);
    if (negate) {
      operand = -operand;
    }

    return operand;
}
```

**Add New Section 3.11.4.X, Fragment Program Operation Precision (insert after Section 3.11.4,2, Fragment Program Parameter Arrays)**

Fragment program implementations may be able to perform instructions with different levels of arithmetic precision. The "R", "H", and "X" opcode precision modifiers (Section 3.11.4) specify the minimum precision used to perform arithmetic operations. Instructions with an "R" precision modifiers will be carried out at no less than IEEE single-precision floating-point (8 bits of exponent, 23 bits of mantissa). Instructions with an "H" precision modifier will be carried out at no less than 16-bit floating-point precision (5 bits of exponent, 10 bits of mantissa). Instructions with an "X" precision modifier will be carried out at no less than signed 12-bit fixed-point precision (two's complement with 10 fraction bits).

If the result of a computation overflows the range of numbers supported by the instruction precision, the result will be +/-INF (infinity) for "R" and "H" precision, or -2048/1024 or +2047/1024 for "X" precision.

If no precision modifier is specified, the instruction will be carried out with at least as much precision as the destination variable.

Rewrite Section 3.11.4.3, Fragment Program Destination Register Update

Most fragment program instructions write a 4-component result vector to a single temporary or fragment 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> rule. The 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 fragment 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 fragment 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, "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;

        // 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 (updatecc) mergedCC.x = GenerateCC(result.x);
        }
        if (instrMask.y && TestCC(cc.*c**)) {
            merged.y = result.y;
            if (updatecc) mergedCC.y = GenerateCC(result.y);
        }
        if (instrMask.z && TestCC(cc.**c*)) {
            merged.z = result.z;
            if (updatecc) mergedCC.z = GenerateCC(result.z);
        }
        if (instrMask.w && TestCC(cc.***c)) {
            merged.w = result.w;
            if (updatecc) mergedCC.w = GenerateCC(result.w);
        }

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

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

**Section 3.11.4.5.3, NV_fragment_program Option**

If a fragment program specifies the "NV_fragment_program" option, the grammar will be extended to support the features found in the NV_fragment_program extension not present in the ARB_fragment_program extension, including:

* the availability of the following instructions:

    - DDX (partial derivative relative to X),
    - DDY (partial derivative relative to Y),
    - PK2H (pack as two half floats),
    - PK2US (pack as two unsigned shorts),
    - PK4B (pack as four signed bytes),
    - PK4UB (pack as four unsigned bytes),
    - RFL (reflection vector),
    - SEQ (set on equal to),
    - SFL (set on false),
    - SGT (set on greater than),
    - SLE (set on less than or equal to),
    - SNE (set on not equal to),
    - STR (set on true),
    - TXD (texture lookup with computed partial derivatives),
    - UP2H (unpack two half floats),
    - UP2US (unpack two unsigned shorts),
    - UP4B (unpack four signed bytes),
    - UP4UB (unpack four unsigned bytes), and
    - X2D (2D coordinate transformation),

* opcode precision suffixes "R", "H", and "X", to specify the precision of arithmetic operations ("R" specifies 32-bit floating-point computations, "H" specifies 16-bit floating-point computations, and "X" specifies 12-bit signed fixed-point computations with 10 fraction bits),

* the availability of the "SHORT" and "LONG" variable precision keywords to control the size of a variable's components,

* a four-component condition code register to hold the sign of result vector components (useful for comparisons),

* a condition code update opcode suffix "C", where the results of the instruction are used to update the condition code register,

* a condition code write mask operator, where the condition code register is swizzled and tested, and the test results are used to mask register writes,

* an absolute value operator on scalar and swizzled source inputs

The added functionality is identical to that provided by the NV_fragment_program extension specification.

**Modify Section 3.11.5, Fragment Program ALU Instruction Set**

**Section 3.11.5.30, DDX: Derivative Relative to X**

The DDX instruction computes approximate partial derivatives of the four components of the single operand with respect to the X window coordinate to yield a result vector. The partial derivatives are evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialX(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

**Section 3.11.5.31, DDY: Derivative Relative to Y**

The DDY instruction computes approximate partial derivatives of the four components of the single operand with respect to the Y window coordinate to yield a result vector. The partial derivatives are evaluated at the center of the pixel.

```
  f = VectorLoad(op0);
  result = ComputePartialY(f);
```

Note that the partial derivates obtained by this instruction are approximate, and derivative-of-derivate instruction sequences may not yield accurate second derivatives.

**Section 3.11.5.32, PK2H: Pack Two 16-bit Floats**

The PK2H instruction converts the "x" and "y" components of the single operand into 16-bit floating-point format, packs the bit representation of these two floats into a 32-bit value, and replicates that value to all four components of the result vector. The PK2H instruction can be reversed by the UP2H instruction below.

```
  tmp0 = VectorLoad(op0);
  /* result obtained by combining raw bits of tmp0.x, tmp0.y */
  result.x = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.y = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.z = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
  result.w = RawBits(tmp0.x) | (RawBits(tmp0.y) << 16);
```

A fragment program will fail to load if it contains a PK2H instruction that writes its results to a variable declared as "SHORT".

**Section 3.11.5.33, PK2US: Pack Two Unsigned 16-bit Scalars**

The PK2US instruction converts the "x" and "y" components of the single operand into a packed pair of 16-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to 0.0 and all '1' bits corresponds to 1.0. The bit representations of the two converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK2US instruction can be reversed by the UP2US instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  us.x = round(65535.0 * tmp0.x);  /* us is a ushort vector */
  us.y = round(65535.0 * tmp0.y);
  /* result obtained by combining raw bits of us. */
  result.x = ((us.x) | (us.y << 16));
  result.y = ((us.x) | (us.y << 16));
  result.z = ((us.x) | (us.y << 16));
  result.w = ((us.x) | (us.y << 16));
```

A fragment program will fail to load if it contains a PK2S instruction that writes its results to a variable declared as "SHORT".

**Section 3.11.5.34, PK4B: Pack Four Signed 8-bit Scalars**

The PK4B instruction converts the four components of the single operand into 8-bit signed quantities. The signed quantities are represented in a bit pattern where all '0' bits corresponds to -128/127 and all '1' bits corresponds to +127/127. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4B instruction can be reversed by the UP4B instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < -128/127) tmp0.x = -128/127;
  if (tmp0.y < -128/127) tmp0.y = -128/127;
  if (tmp0.z < -128/127) tmp0.z = -128/127;
  if (tmp0.w < -128/127) tmp0.w = -128/127;
  if (tmp0.x > +127/127) tmp0.x = +127/127;
  if (tmp0.y > +127/127) tmp0.y = +127/127;
  if (tmp0.z > +127/127) tmp0.z = +127/127;
  if (tmp0.w > +127/127) tmp0.w = +127/127;
  ub.x = round(127.0 * tmp0.x + 128.0);  /* ub is a ubyte vector */
  ub.y = round(127.0 * tmp0.y + 128.0);
  ub.z = round(127.0 * tmp0.z + 128.0);
  ub.w = round(127.0 * tmp0.w + 128.0);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

A fragment program will fail to load if it contains a PK4B instruction that writes its results to a variable declared as "SHORT".

**Section 3.11.5.35, PK4UB: Pack Four Unsigned 8-bit Scalars**

The PK4UB instruction converts the four components of the single operand into a packed grouping of 8-bit unsigned scalars. The scalars are represented in a bit pattern where all '0' bits corresponds to

0.0 and all '1' bits corresponds to 1.0. The bit representations of the four converted components are packed into a 32-bit value, and that value is replicated to all four components of the result vector. The PK4UB instruction can be reversed by the UP4UB instruction below.

```
  tmp0 = VectorLoad(op0);
  if (tmp0.x < 0.0) tmp0.x = 0.0;
  if (tmp0.x > 1.0) tmp0.x = 1.0;
  if (tmp0.y < 0.0) tmp0.y = 0.0;
  if (tmp0.y > 1.0) tmp0.y = 1.0;
  if (tmp0.z < 0.0) tmp0.z = 0.0;
  if (tmp0.z > 1.0) tmp0.z = 1.0;
  if (tmp0.w < 0.0) tmp0.w = 0.0;
  if (tmp0.w > 1.0) tmp0.w = 1.0;
  ub.x = round(255.0 * tmp0.x);  /* ub is a ubyte vector */
  ub.y = round(255.0 * tmp0.y);
  ub.z = round(255.0 * tmp0.z);
  ub.w = round(255.0 * tmp0.w);
  /* result obtained by combining raw bits of ub. */
  result.x = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.y = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.z = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
  result.w = ((ub.x) | (ub.y << 8) | (ub.z << 16) | (ub.w << 24));
```

A fragment program will fail to load if it contains a PK4UB instruction that writes its results to a variable declared as "SHORT".

**Section 3.11.5.36, RFL: Reflection Vector**

The RFL instruction computes the reflection of the second vector operand (the "direction" vector) about the vector specified by the first vector operand (the "axis" vector). Both operands are treated as 3D vectors (the w components are ignored). The result vector is another 3D vector (the "reflected direction" vector). The length of the result vector, ignoring rounding errors, should equal that of the second operand.

```
  axis = VectorLoad(op0);
  direction = VectorLoad(op1);
  tmp.w = (axis.x * axis.x + axis.y * axis.y +
           axis.z * axis.z);
  tmp.x = (axis.x * direction.x + axis.y * direction.y +
           axis.z * direction.z);
  tmp.x = 2.0 * tmp.x;
  tmp.x = tmp.x / tmp.w;
  result.x = tmp.x * axis.x - direction.x;
  result.y = tmp.x * axis.y - direction.y;
  result.z = tmp.x * axis.z - direction.z;
```

A fragment program will fail to load if the w component of the result is enabled in the component write mask.

**Section 3.11.5.37, SEQ: Set on Equal**

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
```

**Section 3.11.5.38, SFL: Set on False**

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

**Section 3.11.5.39, SGT: Set on Greater Than**

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
```

**Section 3.11.5.40, SLE: Set on Less Than or Equal**

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
```

**Section 3.11.5.41, SNE: Set on Not Equal**

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
```

**Section 3.11.5.42, STR: Set on True**

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

**Section 3.11.5.43, UP2H: Unpack Two 16-Bit Floats**

The UP2H instruction unpacks two 16-bit floats stored together in a 32-bit scalar operand. The first 16-bit float (stored in the 16 least significant bits) is written into the "x" and "z" components of the result vector; the second is written into the "y" and "w" components of the result vector.

This operation undoes the type conversion and packing performed by the PK2H instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (fp16) (RawBits(tmp) & 0xFFFF);
  result.y = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
  result.z = (fp16) (RawBits(tmp) & 0xFFFF);
  result.w = (fp16) ((RawBits(tmp) >> 16) & 0xFFFF);
```

A fragment program will fail to load if it contains a UP2H instruction whose operand is a variable declared as "SHORT".

**Section 3.11.5.44, UP2US: Unpack Two Unsigned 16-Bit Scalars**

The UP2US instruction unpacks two 16-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" and "z" components of the result vector are obtained from the 16 least significant bits of the operand; the "y" and "w" components are obtained from the 16 most significant bits.

This operation undoes the type conversion and packing performed by the PK2US instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.y = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
  result.z = ((RawBits(tmp) >> 0)  & 0xFFFF) / 65535.0;
  result.w = ((RawBits(tmp) >> 16) & 0xFFFF) / 65535.0;
```

A fragment program will fail to load if it contains a UP2S instruction whose operand is a variable declared as "SHORT".

**Section 3.11.5.45, UP4B: Unpack Four Signed 8-Bit Values**

The UP4B instruction unpacks four 8-bit signed values packed together in a 32-bit scalar operand. The signed quantities are encoded where a bit pattern of all '0' bits corresponds to -128/127 and a pattern of all '1' bits corresponds to +127/127. The "x" component of the result vector is the converted value corresponding to the 8 least significant bits of the operand; the "w" component corresponds to the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4B instruction.

```
  tmp = ScalarLoad(op0);
  result.x = (((RawBits(tmp) >> 0) & 0xFF) - 128) / 127.0;
  result.y = (((RawBits(tmp) >> 8) & 0xFF) - 128) / 127.0;
  result.z = (((RawBits(tmp) >> 16) & 0xFF) - 128) / 127.0;
  result.w = (((RawBits(tmp) >> 24) & 0xFF) - 128) / 127.0;
```

A fragment program will fail to load if it contains a UP4B instruction whose operand is a variable declared as "SHORT".

**Section 3.11.5.46, UP4UB: Unpack Four Unsigned 8-Bit Scalars**

The UP4UB instruction unpacks four 8-bit unsigned values packed together in a 32-bit scalar operand. The unsigned quantities are encoded where a bit pattern of all '0' bits corresponds to 0.0 and a pattern of all '1' bits corresponds to 1.0. The "x" component of the result vector is obtained from the 8 least significant bits of the operand; the "w" component is obtained from the 8 most significant bits.

This operation undoes the type conversion and packing performed by the PK4UB instruction.

```
  tmp = ScalarLoad(op0);
  result.x = ((RawBits(tmp) >> 0)  & 0xFF) / 255.0;
  result.y = ((RawBits(tmp) >> 8)  & 0xFF) / 255.0;
  result.z = ((RawBits(tmp) >> 16) & 0xFF) / 255.0;
  result.w = ((RawBits(tmp) >> 24) & 0xFF) / 255.0;
```

A fragment program will fail to load if it contains a UP4UB instruction whose operand is a variable declared as "SHORT".

**Section 3.11.5.47, X2D: 2D Coordinate Transformation**

The X2D instruction multiplies the 2D offset vector specified by the "x" and "y" components of the second vector operand by the 2x2 matrix specified by the four components of the third vector operand, and adds the transformed offset vector to the 2D vector specified by the "x" and "y" components of the first vector operand. The first component of the sum is written to the "x" and "z" components of the result; the second component is written to the "y" and "w" components of the result.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.y = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
  result.z = tmp0.x + tmp1.x * tmp2.x + tmp1.y * tmp2.y;
  result.w = tmp0.y + tmp1.x * tmp2.z + tmp1.y * tmp2.w;
```

**Modify Section, 3.11.6.4 KIL: Kill fragment**

Rather than mapping a coordinate set to a color, this function prevents a fragment from receiving any future processing. If any component of its source vector is negative, the processing of this fragment will be discontinued and no further outputs to this fragment will occur. Subsequent stages of the GL pipeline will be skipped for this fragment.

A KIL instruction may be specified using either a vector operand or a condition code test. If a vector operand is specified, the following is performed:

```
  tmp = VectorLoad(op0);
  if ((tmp.x < 0) || (tmp.y < 0) ||
      (tmp.z < 0) || (tmp.w < 0))
  {
      exit;
  }
```

If a condition code is specified, the following is performed:

```
  if (TestCC(rc.c***) || TestCC(rc.*c**) ||
      TestCC(rc.**c*) || TestCC(rc.***c))
  {
     exit;
  }
```

**Add Section 3.11.6.5, TXD: Texture Lookup with Derivatives**

The TXD instruction takes the first three components of its first vector operand and maps them to s, t, and r. 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. In this calculation, ds/dx, dt/dx, and dr/dx are given by the x, y, and z components, respectively, of the second vector operand. ds/dy, dt/dy, and dr/dy are given by the x, y, and z components of the third 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);
```

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

None.

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

None.

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

None.

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

None.

**Additions to the AGL/GLX/WGL Specifications**

None.

**Dependencies on ARB_fragment_program**

This specification is based on a modified version of the grammar published in the ARB_fragment_program specification. This modified grammar (see below) 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.

```
<program>               ::= <optionSequence> <statementSequence> "END"

<optionSequence>        ::= <optionSequence> <option>
                          | /* empty */

<option>                ::= "OPTION" <optionName> ";"

<optionName>            ::= "ARB_fog_exp"
                          | "ARB_fog_exp2"
                          | "ARB_fog_linear"
                          | "ARB_precision_hint_fastest"
                          | "ARB_precision_hint_nicest"

<statementSequence>     ::= <statement> <statementSequence>
                          | /* empty */

<statement>             ::= <instruction> ";"
                          | <namingStatement> ";"

<instruction>           ::= <ALUInstruction>
                          | <TexInstruction>

<ALUInstruction>        ::= <VECTORop_instruction>
                          | <SCALARop_instruction>
                          | <BINSCop_instruction>
                          | <BINop_instruction>
                          | <TRIop_instruction>
                          | <SWZop_instruction>

<TexInstruction>        ::= <TEXop_instruction>
                          | <KILop_instruction>

<VECTORop_instruction>  ::= <VECTORop> <instResult> "," <instOperandV>
```

```
    <VECTORop>                ::= "ABS"
                                | "FLR"
                                | "FRC"
                                | "LIT"
                                | "MOV"

    <SCALARop_instruction>  ::= <SCALARop> <instResult> "," <instOperandS>

    <SCALARop>                ::= "COS"
                                | "EX2"
                                | "LG2"
                                | "RCP"
                                | "RSQ"
                                | "SCS"
                                | "SIN"

    <BINSCop_instruction>   ::= <BINSCop> <instResult> "," <instOperandS> ","
                                <instOperandS>

    <BINSCop>                 ::= "POW"

    <BINop_instruction>     ::= <BINop> <instResult> "," <instOperandV> ","
                                <instOperandV>

    <BINop>                   ::= "ADD"
                                | "DP3"
                                | "DP4"
                                | "DPH"
                                | "DST"
                                | "MAX"
                                | "MIN"
                                | "MUL"
                                | "SGE"
                                | "SLT"
                                | "SUB"
                                | "XPD"

    <TRIop_instruction>     ::= <TRIop> <instResult> "," <instOperandV> ","
                                <instOperandV> "," <instOperandV>

    <TRIop>                   ::= "CMP"
                                | "MAD"
                                | "LRP"

    <SWZop_instruction>     ::= <SWZop> <instResult> "," <instOperandVNS> ","
                                <extendedSwizzle>

    <SWZop>                   ::= "SWZ"

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

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

    <KILop_instruction>     ::= <KILop> <killCond>
    <KILop>                 ::= "KIL"

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

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

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

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

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

    <killCond>              ::= <instOperandV>

    <instOperandV>          ::= <instOperandBaseV>

    <instOperandBaseV>      ::= <optSign> <attribUseV>
                              | <optSign> <tempUseV>
                              | <optSign> <paramUseV>

    <instOperandS>          ::= <instOperandBaseS>

    <instOperandBaseS>      ::= <optSign> <attribUseS>
                              | <optSign> <tempUseS>
                              | <optSign> <paramUseS>

    <instOperandVNS>        ::= <attribUseVNS>
                              | <tempUseVNS>
                              | <paramUseVNS>

    <instResult>            ::= <instResultBase>

    <instResultBase>        ::= <tempUseW>
                              | <resultUseW>

    <namingStatement>       ::= <ATTRIB_statement>
                              | <PARAM_statement>
                              | <TEMP_statement>
                              | <OUTPUT_statement>
                              | <ALIAS_statement>

    <ATTRIB_statement>      ::= "ATTRIB" <establishName> "=" <attribUseD>

    <PARAM_statement>       ::= <PARAM_singleStmt>
                              | <PARAM_multipleStmt>

    <PARAM_singleStmt>      ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>    ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                <paramMultipleInit>
    <optArraySize>          ::= /* empty */
                              | <integer> /* [1,MAX_PROGRAM_PARAMETERS_ARB]*/

    <paramSingleInit>       ::= "=" <paramUseDB>

    <paramMultipleInit>     ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>     ::= <paramUseDM>
                              | <paramUseDM> "," <paramMultInitList>

    <TEMP_statement>        ::= "TEMP" <varNameList>

    <OUTPUT_statement>      ::= "OUTPUT" <establishName> "=" <resultUseD>

    <ALIAS_statement>       ::= "ALIAS" <establishName> "=" <establishedName>

    <establishedName>       ::= <tempVarName>
                              | <addrVarName>
                              | <attribVarName>
                              | <paramArrayVarName>
                              | <paramSingleVarName>
                              | <resultVarName>

    <varNameList>           ::= <establishName>
                              | <establishName> "," <varNameList>

    <establishName>         ::= <identifier>

    <attribUseV>            ::= <attribBasic> <swizzleSuffix>
                              | <attribVarName> <swizzleSuffix>
                              | <attribColor> <swizzleSuffix>
                              | <attribColor> "." <colorType> <swizzleSuffix>

    <attribUseS>            ::= <attribBasic> <scalarSuffix>
                              | <attribVarName> <scalarSuffix>
                              | <attribColor> <scalarSuffix>
                              | <attribColor> "." <colorType> <scalarSuffix>

    <attribUseVNS>          ::= <attribBasic>
                              | <attribVarName>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribUseD>            ::= <attribBasic>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribBasic>           ::= "fragment" "." <attribFragBasic>

    <attribFragBasic>       ::= "texcoord" <optTexCoordNum>
                              | "fogcoord"
                              | "position"

    <attribColor>           ::= "fragment" "." "color"
    <paramUseV>             ::= <paramSingleVarName> <swizzleSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <swizzleSuffix>
                              | <stateSingleItem> <swizzleSuffix>
                              | <programSingleItem> <swizzleSuffix>
                              | <constantVector> <swizzleSuffix>
                              | <constantScalar> <swizzleSuffix>

    <paramUseS>             ::= <paramSingleVarName> <scalarSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <scalarSuffix>
                              | <stateSingleItem> <scalarSuffix>
                              | <programSingleItem> <scalarSuffix>
                              | <constantVector> <scalarSuffix>
                              | <constantScalar> <scalarSuffix>

    <paramUseVNS>           ::= <paramSingleVarName>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                              | <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <constantScalar>

    <paramUseDB>            ::= <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <paramUseDM>            ::= <stateMultipleItem>
                              | <programMultipleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <stateMultipleItem>     ::= <stateSingleItem>
                              | "state" "." <stateMatrixRows>

    <stateSingleItem>       ::= "state" "." <stateMaterialItem>
                              | "state" "." <stateLightItem>
                              | "state" "." <stateLightModelItem>
                              | "state" "." <stateLightProdItem>
                              | "state" "." <stateFogItem>
                              | "state" "." <stateMatrixRow>
                              | "state" "." <stateTexEnvItem>
                              | "state" "." <stateDepthItem>

    <stateMaterialItem>     ::= "material" "." <stateMatProperty>
                              | "material" "." <faceType> "."
                                <stateMatProperty>

    <stateMatProperty>      ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "emission"
                              | "shininess"

    <stateLightItem>        ::= "light" "[" <stateLightNumber> "]" "."
                                <stateLightProperty>
    <stateLightProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "position"
                              | "attenuation"
                              | "spot" "." <stateSpotProperty>
                              | "half"

    <stateSpotProperty>     ::= "direction"

    <stateLightModelItem>   ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>     ::= "." "ambient"
                              | "." "scenecolor"
                              | "." <faceType> "." "scenecolor"

    <stateLightProdItem>    ::= "lightprod" "[" <stateLightNumber> "]" "."
                                <stateLProdProperty>
                              | "lightprod" "[" <stateLightNumber> "]" "."
                                <faceType> "." <stateLProdProperty>

    <stateLProdProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"

    <stateLightNumber>      ::= <integer> /* [0,MAX_LIGHTS-1] */

    <stateFogItem>          ::= "fog" "." <stateFogProperty>

    <stateFogProperty>      ::= "color"
                              | "params"

    <stateMatrixRows>       ::= <stateMatrixItem>
                              | <stateMatrixItem> "." <stateMatModifier>
                              | <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> ".." <stateMatrixRowNum>
                                "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> ".."
                                <stateMatrixRowNum> "]"

    <stateMatrixRow>        ::= <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> "]"

    <stateMatrixItem>       ::= "matrix" "." <stateMatrixName>

    <stateMatModifier>      ::= "inverse"
                              | "transpose"
                              | "invtrans"
    <stateMatrixName>        ::= "modelview" <stateOptModMatNum>
                               | "projection"
                               | "mvp"
                               | "texture" <optTexCoordNum>
                               | "palette" "[" <statePaletteMatNum> "]"
                               | "program" "[" <stateProgramMatNum> "]"

    <stateMatrixRowNum>      ::= <integer> /* [0,3] */

    <stateOptModMatNum>      ::= /* empty */
                               | "[" <stateModMatNum> "]"

    <stateModMatNum>         ::= <integer> /*[0,MAX_VERTEX_UNITS_ARB-1]*/

    <statePaletteMatNum>     ::= <integer> /*[0,MAX_PALETTE_MATRICES_ARB-1]*/

    <stateProgramMatNum>     ::= <integer> /*[0,MAX_PROGRAM_MATRICES_ARB-1]*/

    <stateTexEnvItem>        ::= "texenv" <optLegacyTexUnitNum> "."
                                 <stateTexEnvProperty>

    <stateTexEnvProperty>    ::= "color"

    <stateDepthItem>         ::= "depth" "." <stateDepthProperty>

    <stateDepthProperty>     ::= "range"

    <programSingleItem>      ::= <progEnvParam>
                               | <progLocalParam>

    <programMultipleItem>    ::= <progEnvParams>
                               | <progLocalParams>

    <progEnvParams>          ::= "program" "." "env" "[" <progEnvParamNums> "]"

    <progEnvParamNums>       ::= <progEnvParamNum>
                               | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>           ::= "program" "." "env" "[" <progEnvParamNum> "]"

    <progLocalParams>        ::= "program" "." "local" "[" <progLocalParamNums>
                                 "]"

    <progLocalParamNums>     ::= <progLocalParamNum>
                               | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>         ::= "program" "." "local" "[" <progLocalParamNum>
                                 "]"

    <progEnvParamNum>        ::= <integer>
                                 /*[0,MAX_PROGRAM_ENV_PARAMETERS_ARB-1]*/

    <progLocalParamNum>      ::= <integer>
                                 /*[0,MAX_PROGRAM_LOCAL_PARAMETERS_ARB-1]*/

    <constantVector>         ::= "{" <constantVectorList> "}"
    <constantVectorList>    ::= <signedConstantScalar>
                               | <signedConstantScalar> ","
                                 <signedConstantScalar>
                               | <signedConstantScalar> ","
                                 <signedConstantScalar> ","
                                 <signedConstantScalar>
                               | <signedConstantScalar> ","
                                 <signedConstantScalar> ","
                                 <signedConstantScalar> ","
                                 <signedConstantScalar>

    <signedConstantScalar>  ::= <optSign> <constantScalar>

    <constantScalar>        ::= <floatConstant>

    <floatConstant>         ::= <float>

    <tempUseV>              ::= <tempVarName> <swizzleSuffix>

    <tempUseS>              ::= <tempVarName> <scalarSuffix>

    <tempUseVNS>            ::= <tempVarName>

    <tempUseW>              ::= <tempVarName> <optWriteMask>

    <resultUseW>            ::= <resultBasic> <optWriteMask>
                               | <resultVarName> <optWriteMask>

    <resultUseD>            ::= <resultBasic>

    <resultBasic>           ::= "result" "." <resultFragBasic>

    <resultFragBasic>       ::= "color" <resultOptColorNum>
                               | "depth"

    <resultOptColorNum>     ::= /* empty */

    <arrayMem>              ::= <arrayMemAbs>

    <arrayMemAbs>           ::= <integer>

    <optWriteMask>          ::= /* empty */
                               | <xyzwMask>
                               | <rgbaMask>
    <xyzwMask>              ::= "." "x"
                               | "." "y"
                               | "." "xy"
                               | "." "z"
                               | "." "xz"
                               | "." "yz"
                               | "." "xyz"
                               | "." "w"
                               | "." "xw"
                               | "." "yw"
                               | "." "xyw"
                               | "." "zw"
                               | "." "xzw"
                               | "." "yzw"
                               | "." "xyzw"

    <rgbaMask>              ::= "." "r"
                               | "." "g"
                               | "." "rg"
                               | "." "b"
                               | "." "rb"
                               | "." "gb"
                               | "." "rgb"
                               | "." "a"
                               | "." "ra"
                               | "." "ga"
                               | "." "rga"
                               | "." "ba"
                               | "." "rba"
                               | "." "gba"
                               | "." "rgba"

    <swizzleSuffix>         ::= /* empty */
                               | "." <component>
                               | "." <xyzwComponent> <xyzwComponent>
                                 <xyzwComponent> <xyzwComponent>
                               | "." <rgbaComponent> <rgbaComponent>
                                 <rgbaComponent> <rgbaComponent>

    <extendedSwizzle>       ::= <extSwizComp> "," <extSwizComp> ","
                                 <extSwizComp> "," <extSwizComp>

    <extSwizComp>           ::= <optSign> <xyzwExtSwizSel>
                               | <optSign> <rgbaExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                               | "1"
                               | <xyzwComponent>

    <rgbaExtSwizSel>        ::= <rgbaComponent>

    <scalarSuffix>          ::= "." <component>

    <component>             ::= <xyzwComponent>
                               | <rgbaComponent>
    <xyzwComponent>         ::= "x"
                               | "y"
                               | "z"
                               | "w"

    <rgbaComponent>         ::= "r"
                               | "g"
                               | "b"
                               | "a"

    <optSign>               ::= /* empty */
                               | "-"
                               | "+"

    <faceType>              ::= "front"
                               | "back"

    <colorType>             ::= "primary"
                               | "secondary"

    <optTexCoordNum>        ::= /* empty */
                               | "[" <texCoordNum> "]"

    <texCoordNum>           ::= <integer> /*[0,MAX_TEXTURE_COORDS_ARB-1]*/

    <optLegacyTexUnitNum>   ::= /* empty */
                               | "[" <legacyTexUnitNum> "]"

    <legacyTexUnitNum>      ::= <integer> /*[0,MAX_TEXTURE_UNITS-1]*/
```

The <integer>, <float>, and <identifier> grammar rules match integer constants, floating point constants, and identifier names as described in the ARB_vertex_program specification. The <float> grammar rule here is identical to the <floatConstant> grammar rule in ARB_vertex_program.

The grammar rules <tempVarName>, <addrVarName>, <attribVarName>, <paramArrayVarName>, <paramSingleVarName>, <resultVarName> refer to the names of temporary, address register, attribute, program parameter array, program parameter, and result variables declared in the program text.

**GLX Protocol**

None.

**Errors**

None.

**New State**

None.

**Revision History**

```
    Rev.  Date      Author   Changes
    ----  --------  -------  ---------------------------------------------
    2     05/16/04  pbrown   Documented terminals in modified fragment
                             program grammar.

    1     --------  pbrown   Internal pre-release revisions.
```

**Name**

    NV_half_float

**Name Strings**

    GL_NV_half_float

**Notice**

**IP Status**

    NVIDIA Proprietary.

**Status**

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

**Version**

```
    Last Modified Date:         02/25/2004
    NVIDIA Revision:            9
```

**Number**

    283

**Dependencies**

    Written based on the wording of the OpenGL 1.3 specification.

    OpenGL 1.1 is required.

    NV_float_buffer affects the definition of this extension.

    EXT_fog_coord affects the definition of this extension.

    EXT_secondary_color affects the definition of this extension.

    EXT_vertex_weighting affects the definition of this extension.

    NV_vertex_program affects the definition of this extension.

**Overview**

    This extension introduces a new storage format and data type for
    half-precision (16-bit) floating-point quantities.  The floating-point
    format is very similar to the IEEE single-precision floating-point
    standard, except that it has only 5 exponent bits and 10 mantissa bits.
    Half-precision floats are smaller than full precision floats and provide a
    larger dynamic range than similarly-sized normalized scalar data types.

    This extension allows applications to use half-precision floating point
    data when specifying vertices or pixel data.  It adds new commands to

specify vertex attributes using the new data type, and extends the existing vertex array and image specification commands to accept the new data type.

This storage format is also used to represent 16-bit components in the floating-point frame buffers, as defined in the NV_float_buffer extension.

**Issues**

*What should the new data type be called? "half"? "hfloat"? In addition, what should the immediate mode function suffix be? "h"? "hf"?*

RESOLVED: half and "h". This convention builds on the convention of using the type "double" to describe double-precision floating-point numbers. Here, "half" will refer to half-precision floating-point numbers.

Even though the 16-bit float data type is a first-class data type, it is still more problematic than the other types in the sense that no native programming languages support the data type. "hfloat/hf" would have reflected a second-class status better than "half/h".

Both names are not without conflicting precedents. The name "half" is used to connote 16-bit scalar values on some 32-bit CPU architectures (e.g., PowerPC). The name "hfloat" has been used to describe 128-bit floating-point data on VAX systems.

*Should we provide immediate-mode entry points for half-precision floating-point data types?*

RESOLVED: Yes, for orthogonality. Also useful as a fallback for the "general" case for ArrayElement.

*Should we support half-precision floating-point color index data?*

RESOLVED: No.

*Should half-precision data be accepted by all commands that accept pixel data or only a subset?*

RESOLVED: All functions. Note that some textures or frame buffers may store the half-precision floating-point data natively.

Since half float data would be accepted in some cases, it will be necessary for drivers to provide some data conversion code. This code can be reused to handle the less common commands.

**New Procedures and Functions**

```
    void Vertex2hNV(half x, half y);
    void Vertex2hvNV(const half *v);
    void Vertex3hNV(half x, half y, half z);
    void Vertex3hvNV(const half *v);
    void Vertex4hNV(half x, half y, half z, half w);
    void Vertex4hvNV(const half *v);
    void Normal3hNV(half nx, half ny, half nz);
    void Normal3hvNV(const half *v);
    void Color3hNV(half red, half green, half blue);
    void Color3hvNV(const half *v);
    void Color4hNV(half red, half green, half blue, half alpha);
    void Color4hvNV(const half *v);
    void TexCoord1hNV(half s);
    void TexCoord1hvNV(const half *v);
    void TexCoord2hNV(half s, half t);
    void TexCoord2hvNV(const half *v);
    void TexCoord3hNV(half s, half t, half r);
    void TexCoord3hvNV(const half *v);
    void TexCoord4hNV(half s, half t, half r, half q);
    void TexCoord4hvNV(const half *v);
    void MultiTexCoord1hNV(enum target, half s);
    void MultiTexCoord1hvNV(enum target, const half *v);
    void MultiTexCoord2hNV(enum target, half s, half t);
    void MultiTexCoord2hvNV(enum target, const half *v);
    void MultiTexCoord3hNV(enum target, half s, half t, half r);
    void MultiTexCoord3hvNV(enum target, const half *v);
    void MultiTexCoord4hNV(enum target, half s, half t, half r, half q);
    void MultiTexCoord4hvNV(enum target, const half *v);
    void FogCoordhNV(half fog);
    void FogCoordhvNV(const half *fog);
    void SecondaryColor3hNV(half red, half green, half blue);
    void SecondaryColor3hvNV(const half *v);
    void VertexWeighthNV(half weight);
    void VertexWeighthvNV(const half *weight);
    void VertexAttrib1hNV(uint index, half x);
    void VertexAttrib1hvNV(uint index, const half *v);
    void VertexAttrib2hNV(uint index, half x, half y);
    void VertexAttrib2hvNV(uint index, const half *v);
    void VertexAttrib3hNV(uint index, half x, half y, half z);
    void VertexAttrib3hvNV(uint index, const half *v);
    void VertexAttrib4hNV(uint index, half x, half y, half z, half w);
    void VertexAttrib4hvNV(uint index, const half *v);
    void VertexAttribs1hvNV(uint index, sizei n, const half *v);
    void VertexAttribs2hvNV(uint index, sizei n, const half *v);
    void VertexAttribs3hvNV(uint index, sizei n, const half *v);
    void VertexAttribs4hvNV(uint index, sizei n, const half *v);
```

**New Tokens**

    Accepted by the <type> argument of VertexPointer, NormalPointer,
    ColorPointer, TexCoordPointer, FogCoordPointerEXT,
    SecondaryColorPointerEXT, VertexWeightPointerEXT, VertexAttribPointerNV,
    DrawPixels, ReadPixels, TexImage1D, TexImage2D, TexImage3D, TexSubImage1D,
    TexSubImage2D, TexSubImage3D, and GetTexImage:

        HALF_FLOAT_NV                                   0x140B

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

**Modify Section 2.3, GL Command Syntax (p. 7)**

(Modify the last paragraph, p. 7. In the text below, "e*" represents the epsilon character used to indicate no character.)

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form

```
    rtype Name{e*1234}{e* b s i h f d ub us ui}{e*v}
      ( [args ,] T arg1, ... , T argN [, args]);
```

(Modify Table 2.1, p. 8 -- add new row)

| Letter | Corresponding GL Type |
|---|---|
| h | half |

(add after last paragraph, p. 8) The half data type is a floating-point data type encoded in an unsigned scalar data type. If the unsigned scalar holding a half has a value of N, the corresponding floating point number is

```
    (-1)^S * 0.0,                         if E == 0 and M == 0,
    (-1)^S * 2^-14 * (M / 2^10),          if E == 0 and M != 0,
    (-1)^S * 2^(E-15) * (1 + M/2^10),     if 0 < E < 31,
    (-1)^S * INF,                         if E == 31 and M == 0, or
    NaN,                                  if E == 31 and M != 0,
```

where

```
    S = floor((N mod 65536) / 32768),
    E = floor((N mod 32768) / 1024), and
    M = N mod 1024.
```

INF (Infinity) is a special representation indicating numerical overflow. NaN (Not a Number) is a special representation indicating the result of illegal arithmetic operations, such as computing the square root or logarithm of a negative number. Note that all normal values, zero, and INF have an associated sign. -0.0 and +0.0 are considered equivalent for the purposes of comparisons. Note also that half is not a native type in most CPUs, so some special processing may be required to generate or interpret half data.

(Modify Table 2.2, p. 9 -- add new row)

| GL Type | Minimum Bit Width | Description |
|---|---|---|
| half | 16 | half-precision floating-point value encoded in an unsigned scalar |

**Modify Section 2.7, Vertex Specification, p. 19**

(Modify the descriptions of the immediate mode functions in this section, including those introduced by extensions.)

```
    void Vertex[234][sihfd]( T coords );
    void Vertex[234][sihfd]v( T coords );
...
    void TexCoord[1234][sihfd]( T coords );
    void TexCoord[1234][sihfd]v( T coords );
...
    void MultiTexCoord[1234][sihfd](enum texture, T coords);
    void MultiTexCoord[1234][sihfd]v(enum texture, T coords);
...
    void Normal3[bsihfd][ T coords );
    void Normal3[bsihfd]v( T coords );
...
    void Color[34][bsihfd ubusui]( T components );
    void Color[34][bsihfd ubusui]v( T components );
...
    void FogCoord[fd]EXT(T fog);
    void FogCoordhNV(T fog);
    void FogCoord[fd]vEXT(T fog);
    void FogCoordhvNV(T fog);
...
    void SecondaryColor3[bsihfd ubusui]( T components );
    void SecondaryColor3hNV( T components );
    void SecondaryColor3[bsihfd ubusui]v( T components );
    void SecondaryColor3hvNV( T components );
...
    void VertexWeightfEXT(T weight);
    void VertexWeighthNV(T weight);
    void VertexWeightfvEXT(T weight);
    void VertexWeighthvNV(T weight);
...
    void VertexAttrib[1234][shfd]NV(uint index, T components);
    void VertexAttrib4ubNV(uint index, T components);
    void VertexAttrib[1234][shfd]vNV(uint index, T components);
    void VertexAttrib4ubvNV(uint index, T components);
    void VertexAttribs[1234][shfd]vNV(uint index, sizei n, T components);
    void VertexAttribs4ubvNV(uint index, sizei n, T components);
....
```

**Modify Section 2.8, Vertex Arrays, p. 21**

(Modify 1st paragraph on p. 22) ... For <type>, the values BYTE, SHORT, INT, FLOAT, HALF_FLOAT_NV, and DOUBLE indicate types byte, short, int, float, half, and double, respectively. ...

(Modify Table 2.4, p. 23)

| Command | Sizes | Types |
|---|---|---|
| VertexPointer | 2,3,4 | short, int, float, half, double |
| NormalPointer | 3 | byte, short, int, float, half, double |
| ColorPointer | 3,4 | byte, ubyte, short, ushort, int, uint, float, half, double |
| IndexPointer | 1 | ubyte, short, int, float, double |
| TexCoordPointer | 1,2,3,4 | short, int, float, half, double |
| EdgeFlagPointer | 1 | boolean |
| FogCoordPointerEXT | 1 | float, half, double |
| SecondaryColorPointerEXT | 3 | byte, ubyte, short, ushort, int, uint, float, half, double |
| VertexWeightPointerEXT | 1 | float, half |

Table 2.4: Vertex array sizes (values per vertex) and data types.

**Modify Section 2.13, Colors and Coloring, p.44**

(Modify Table 2.6, p. 45) Add new row to the table:

| GL Type | Conversion |
|---|---|
| half | c |

Modify NV_vertex_program_spec, Section 2.14.3, Vertex Arrays for Vertex Attributes.

(modify paragraph describing VertexAttribPointer) ... type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, HALF_FLOAT_NV, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, half, double, and ubyte respectively. ...

(add to end of paragraph describing mapping of vertex arrays to immediate-mode functions) ... For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,h,d,ub], corresponding to the array types short, int, float, half, double, and ubyte respectively.

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

**Modify Section 3.6.4, Rasterization of Pixel Rectangles (p. 91)**

(Modify Table 3.5, p. 94 -- add new row)

| type Parameter Token Name | Corresponding GL Data Type | Special Interpretation |
|---|---|---|
| HALF_FLOAT_NV | half | No |

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

Modify Section 4.3.2, Reading Pixels (p. 173)

(modify Final Conversion, p. 177) For an index, if the type is not FLOAT or HALF_FLOAT_NV, final conversion consists of masking the index with the value given in Table 4.6; if the type is FLOAT or HALF_FLOAT_NV, then the integer index is converted to a GL float or half data value. For an RGBA color, components are clamped depending on the data type of the buffer being read. For fixed-point buffers, each component is clamped to [0.1]. For floating-point buffers, if <type> is not FLOAT or HALF_FLOAT_NV, each component is clamped to [0,1] if <type> is unsigned or [-1,1] if <type> is signed and then converted according to Table 4.7.

(Modify Table 4.7, p. 178 -- add new row)

| type Parameter | GL Data Type | Component Conversion Formula |
|---|---|---|
| HALF_FLOAT_NV | half | c = f |

**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 Appendix A of the OpenGL 1.3 Specification (Invariance)**

None.

**Additions to the AGL/GLX/WGL Specifications**

None.

**GLX Protocol (Modification to the GLX 1.3 Protocol Encoding Specification)**

Add to Section 1.4 (p.2), Common Types

| | |
|---|---|
| FLOAT16 | A 16-bit floating-point value in the format specified in the NV_half_float extension specification. |

Modify Section 2.3.3 (p. 79), GL Rendering Commands

The following rendering commands are sent to the server as part of a glXRender request:

**Vertex2hvNV**

```
   2          8               rendering command length
   2          4240            rendering command opcode
   2          FLOAT16         v[0]
   2          FLOAT16         v[1]
```

```
Vertex3hvNV
    2           12              rendering command length
    2           4241            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2                           unused

Vertex4hvNV
    2           12              rendering command length
    2           4242            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2           FLOAT16         v[3]

Normal3hvNV
    2           12              rendering command length
    2           4243            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2                           unused

Color3hvNV
    2           12              rendering command length
    2           4244            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2                           unused

Color4hvNV
    2           12              rendering command length
    2           4245            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
    2           FLOAT16         v[2]
    2           FLOAT16         v[3]

TexCoord1hvNV
    2           8               rendering command length
    2           4246            rendering command opcode
    2           FLOAT16         v[0]
    2                           unused

TexCoord2hvNV
    2           8               rendering command length
    2           4247            rendering command opcode
    2           FLOAT16         v[0]
    2           FLOAT16         v[1]
```

```
        TexCoord3hvNV
            2           12              rendering command length
            2           4248            rendering command opcode
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        TexCoord4hvNV
            2           12              rendering command length
            2           4249            rendering command opcode
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2           FLOAT16         v[3]

        MultiTexCoord1hvNV
            2           12              rendering command length
            2           4250            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2                           unused

        MultiTexCoord2hvNV
            2           12              rendering command length
            2           4251            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]

        MultiTexCoord3hvNV
            2           16              rendering command length
            2           4252            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        MultiTexCoord4hvNV
            2           16              rendering command length
            2           4253            rendering command opcode
            4           ENUM            target
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2           FLOAT16         v[3]

        FogCoordhvNV
            2           8               rendering command length
            2           4254            rendering command opcode
            2           FLOAT16         v[0]
            2                           unused
        SecondaryColor3hvNV
            2           12              rendering command length
            2           4255            rendering command opcode
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        VertexWeighthvNV
            2           8               rendering command length
            2           4256            rendering command opcode
            2           FLOAT16         v[0]
            2                           unused

        VertexAttrib1hvNV
            2           12              rendering command length
            2           4257            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2                           unused

        VertexAttrib2hvNV
            2           12              rendering command length
            2           4258            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]

        VertexAttrib3hvNV
            2           16              rendering command length
            2           4259            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2                           unused

        VertexAttrib4hvNV
            2           16              rendering command length
            2           4260            rendering command opcode
            4           CARD32          index
            2           FLOAT16         v[0]
            2           FLOAT16         v[1]
            2           FLOAT16         v[2]
            2           FLOAT16         v[3]

        VertexAttribs1hvNV
            2           12+2*n+p        rendering command length
            2           4261            rendering command opcode
            4           CARD32          index
            4           CARD32          n
            2*n         LISTofFLOAT16   v
            p                           unused, p=pad(2*n)
        VertexAttribs2hvNV
           2          12+4*n          rendering command length
           2          4262            rendering command opcode
           4          CARD32          index
           4          CARD32          n
           4*n        LISTofFLOAT16   v

        VertexAttribs3hvNV
           2          12+6*n+p        rendering command length
           2          4263            rendering command opcode
           4          CARD32          index
           4          CARD32          n
           6*n        LISTofFLOAT16   v
           p                          unused, p=pad(6*n)

        VertexAttribs4hvNV
           2          12+8*n          rendering command length
           2          4264            rendering command opcode
           4          CARD32          index
           4          CARD32          n
           8*n        LISTofFLOAT16   v
```

**Modify Section 2.3.4, GL Rendering Commands That May Be Large (p. 127)**

(Modify the ARRAY_INFO portion of the DrawArrays encoding (p.129) to reflect the new data type supported by vertex arrays.)

```
        ARRAY_INFO

           4        enum                  data type
                    0x1400   i=1          BYTE
                    0x1401   i=1          UNSIGNED_BYTE
                    0x1402   i=2          SHORT
                    ...
                    0x140B   i=2          HALF_FLOAT_NV
           4        INT32                 j
           4        ENUM                  array type
               ...
```

**Modify Appendix A, Pixel Data (p. 148)**

(Modify Table A.1, p. 149 -- add new row for HALF_FLOAT_NV data)

| type | Encoding | Protocol Type | nbytes |
|---|---|---|---|
| HALF_FLOAT_NV | 0x140B | CARD16 | 2 |

**Dependencies on NV_float_buffer**

If NV_float_buffer is not supported, the fixed and floating-point color buffer language in ReadPixels "Final Conversion" should be removed.

**Dependencies on EXT_fog_coord, EXT_secondary_color, and EXT_vertex_weighting**

If EXT_fog_coord, EXT_secondary_color, or EXT_vertex_weighting are not supported, references to FogCoordPointerEXT, SecondaryColorPointerEXT, and VertexWeightEXT, respectively, should be removed.

**Dependencies on NV_vertex_program**

If NV_vertex_program is not supported, references to VertexAttribPointerNV should be removed, as should references to VertexAttrib*h[v] commands.

**Errors**

None.

**New State**

None.

**New Implementation Dependent State**

```
    Rev.    Date    Author   Changes
    ----  -------- --------  --------------------------------------------
      9   02/25/04  pbrown   Fixed incorrect language using division by zero
                             as an example of something producing a NaN.
```

## Name

    NV_primitive_restart

## Name Strings

    GL_NV_primitive_restart

## Notice

## IP Status

NVIDIA Proprietary.

## Status

Implemented in CineFX (NV30) Emulation driver, August 2002.
Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

## Version

NVIDIA Date: August 29, 2002 (version 0.1)

## Number

285

## Dependencies

Written based on the wording of the OpenGL 1.3 specification.

## Overview

This extension allows applications to easily and inexpensively restart a primitive in its middle. A "primitive restart" is simply the same as an End command, followed by another Begin command with the same mode as the original. The typical expected use of this feature is to draw a mesh with many triangle strips, though primitive restarts are legal for all primitive types, even for points (where they are not useful).

Although the EXT_multi_draw_arrays extension did reduce the overhead of such drawing techniques, they still remain more expensive than one would like.

This extension provides an extremely lightweight primitive restart, which is accomplished by allowing the application to choose a special index number that signals that a primitive restart should occur, rather than a vertex being provoked. This index can be an arbitrary 32-bit integer for maximum application convenience.

In addition, for full orthogonality, a special OpenGL command is provided to restart primitives when in immediate mode. This command is not likely to increase performance in any significant fashion, but providing it greatly simplifies the specification and implementation of display list compilation and indirect rendering.

**Issues**

* *What should the default primitive restart index be?*

RESOLVED: Zero. It's tough to pick another number that is meaningful for all three element data types. In practice, apps are likely to set it to 0xFFFF or 0xFFFFFFFF.

* *Are primitives other than triangle strips supported?*

RESOLVED: Yes. One example of how this can be useful is for rendering a heightfield. The "standard" way to render a heightfield uses a number of triangle strips, one for each row of the grid. Another method, which can produce higher-quality meshes, is to render a number of 8-triangle triangle fans. This has the effect of alternating the direction of tessellation, as shown in the diagram below. Primitive restarts enhance the performance of both techniques.

```
-------------------------        -------------------------
| /| /| /| /| /| /| /| /|        |\ | /|\ | /|\ | /|\ | /|
|/ |/ |/ |/ |/ |/ |/ |/ |        | \|/ | \|/ | \|/ | \|/ |
-------------------------        ---*-----*-----*-----*---
| /| /| /| /| /| /| /| /|        | /|\ | /|\ | /|\ | /|\ |
|/ |/ |/ |/ |/ |/ |/ |/ |        |/ | \|/ | \|/ | \|/ | \|
-------------------------        -------------------------

        Two strips             Four fans (centers marked '*')
```

* *How is this feature turned on and off?*

RESOLVED: Via a glEnable/DisableClientState setting. It is not possible to select a restart index that is guaranteed to be unused.

* *Is the immediate mode PrimitiveRestartNV needed?*

RESOLVED: Yes. It is difficult to make indirect rendering to work without it, and it is near impossible to make display lists work without it. It is a very clean way to resolve these issues.

* *How is indirect rendering handled?*

RESOLVED: Because of PrimitiveRestartNV, it works very easily. PrimitiveRestartNV has a wire protocol and therefore it can easily be inserted as needed. The server tracks the current Begin mode, relieving the client of this burden.

Note that in practice, we expect that this feature is essentially useless for indirect rendering.

* *How does this extension interact with NV_element_array and NV_vertex_array_range?*

    RESOLVED: It doesn't, not even for performance.  It should be
    fast on hardware that supports the feature with or without the
    use of element arrays, with or without vertex array range.

*   *Does this extension affect ArrayElement and DrawArrays, or just DrawElements?*

    RESOLVED: All of them.  It applies to ArrayElement and to the
    rest as a consequence.  It is likely not useful with any other
    than DrawElements, but nevertheless not prohibited.

*   *In the case of ArrayElement, what happens if the restart index is used outside Begin/End?*

    RESOLVED: Since this is defined as being equivalent to a call to
    PrimitiveRestartNV, and PrimitiveRestartNV is an
    INVALID_OPERATION when not inside Begin/End, this is just an
    error.

*   *For DrawRangeElements/LockArrays purposes, must the restart index lie within the start/end range?*

    RESOLVED: No, this would to some extent defeat the point if the
    restart index was, e.g., 0xFFFFFFFF.  I don't believe any spec
    language is required here, since hitting this index does not
    cause a vertex to be dereferenced.

*   *Should this state push/pop?*

    RESOLVED: Yes, as vertex array client state.

**New Procedures and Functions**

```
void PrimitiveRestartNV(void);
void PrimitiveRestartIndexNV(uint index);
```

**New Tokens**

    Accepted by the <array> parameter of EnableClientState and
    DisableClientState, by the <cap> parameter of IsEnabled, and by
    the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
    GetDoublev:

```
    PRIMITIVE_RESTART_NV                          0x8558
```

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

```
    PRIMITIVE_RESTART_INDEX_NV                    0x8559
```

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

Add a section 2.6.X "Primitive Restarts", immediately after section 2.6.2 "Polygon Edges" (page 19):

**"2.6.X Primitive Restarts**

An OpenGL primitive may be restarted with the command

```
  void PrimitiveRestartNV(void)
```

Between the execution of a Begin and its corresponding End, this command is equivalent to a call to End, followed by a call to Begin where the mode argument is the same mode as that used by the previous Begin. Outside the execution of a Begin and its corresponding End, this command generates the error INVALID_OPERATION."

Add PrimitiveRestartNV to the list of commands that are allowed between Begin and End in section 2.6.3 "GL Commands within Begin/End" (page 19).

**Add to section 2.8 "Vertex Arrays", after the description of ArrayElement (page 24):**

"Primitive restarting is enabled or disabled by calling EnableClientState or DisableClientState with parameter PRIMITIVE_RESTART_NV. The command

```
  void PrimitiveRestartIndexNV(uint index)
```

specifies the index of a vertex array element that is treated specially when primitive restarting is enabled. When ArrayElement is called between an execution of Begin and the corresponding execution of End, if i is equal to PRIMITIVE_RESTART_INDEX_NV, then no vertex data is derefererenced, and no current vertex state is modified. Instead, it is as if PrimitiveRestartNV had been called."

**Replace the last paragraph of section 2.8 "Vertex Arrays" (page 28) with the following:**

"If the number of supported texture units (the value of MAX_TEXTURE_UNITS) is k, then the client state required to implement vertex arrays consists of 7+k boolean values, 5+k memory pointers, 5+k integer stride values, 4+k symbolic constants representing array types, 3+k integers representing values per element, and an unsigned integer representing the restart index. In the initial state, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, the integers representing values per element are each four, and the restart index is zero."

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

None.

**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)**

**Add to the end of Section 5.4 "Display Lists":**

"PrimitiveRestartIndexNV is not compiled into display lists, but is executed immediately."

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

None.

**GLX Protocol**

One new GL command is added.

The following rendering command is sent to the server as part of a glXRender request:

```
    PrimitiveRestartNV
        2           4               rendering command length
        2           ????            rendering command opcode
```

**Errors**

The error INVALID_OPERATION is generated if PrimitiveRestartNV is called outside the execution of Begin and the corresponding execution of End.

The error INVALID_OPERATION is generated if PrimitiveRestartIndexNV is called between the execution of Begin and the corresponding execution of End.

**New State**

| Get Value | Get Command | Type | Initial Value | Sec | Attrib |
|---|---|---|---|---|---|
| PRIMITIVE_RESTART_NV | IsEnabled | B | FALSE | 2.8 | vertex-array |
| PRIMITIVE_RESTART_INDEX_NV | GetIntegerv | Z+ | 0 | 2.8 | vertex-array |

**Name**

    NV_texture_expand_normal

**Name Strings**

    GL_NV_texture_expand_normal

**Notice**

**IP Status**

    NVIDIA Proprietary.

**Status**

    Implemented, November 2002

**Version**

```
Last Modified:      $Date: 2002/11/15 $
NVIDIA Revision:    3
```

**Number**

    Unassigned

**Dependencies**

    OpenGL 1.1 is required.

**Overview**

This extension provides a remapping mode where unsigned texture components (in the range [0,1]) can be treated as though they contained signed data (in the range [-1,+1]). This allows applications to easily encode signed data into unsigned texture formats.

The functionality of this extension is nearly identical to the EXPAND_NORMAL_NV remapping mode provided in the NV_register_combiners extension, although it applies even if register combiners are used.

**Issues**

*(1) When is the remapping applied?*

RESOLVED: It would be possible to remap after loading each texel, remap after all filtering is done, or something in between. Ignoring implementation-dependent rounding errors, it really doesn't matter.

The spec language says that the remapping is applied after filtering texel values within each level. For LINEAR_MIPMAP_LINEAR, this means that the remapping is "done" twice. This approach was chosen solely to simplify the spec language, and does not necessarily reflect NVIDIA's implementation.

*(2) Should the remapping mode apply to textures with signed components?*

RESOLVED:  No -- the EXPAND_NORMAL_NV mapping is ignored for such textures.

*(3) NV_texture_shader provides several internal formats with a mix of signed and unsigned components.  For example, the base formats DSDT_MAG_NV, and DSDT_MAG_INTENSITY_NV have this property, and there is a variant of RGBA where the RGB components are signed, but the A component is unsigned.  What should happen in this case?*

RESOLVED:  The unsigned components are remapped; the signed components are unmodified.

*(4) What should be said about signed fixed-point precision and range of actual implementations?*

RESOLVED:  The fundamental problem is that it is not possible to derive a linear mapping taking unsigned values that exactly represents -1.0, 0.0, and +1.0.

The mapping chosen for current NVIDIA implementations does not exactly represent +1.0.  For an n-bit fixed-point component, 0 maps to -1.0, 2^(n-1) maps to 0.0, and 2^n-1 (maximum value) maps to 1.0 - 1/(2^(n-1)).  This same conversion is applied to stored textures using the signed texture types in NV_texture_shader.

This specification is written using the conventional OpenGL mapping where -1.0 and +1.0 can be represented exactly, but 0.0 can not. The specification is simpler and avoids precision-dependent language describing the mapping.  We expect some leeway in how the remapping is applied.

This issue is discussed in more detail in the issues section of the NV_texture_shader specification (the question is phrased identically).

*(5) Are texture border color components remapped?*

RESOLVED:  Yes -- if the border values are used for filtering, border color components are remapped identically to normal texel components.

**New Procedures and Functions**

None.

**New Tokens**

*Accepted by the <pname> parameters of TexParameteri, TexParameteriv, TexParameterf, TexParameterfv, GetTexParameteri, and GetTexParameteriv:*

```
    TEXTURE_UNSIGNED_REMAP_MODE_NV              0x888F
```

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

None.

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

**Modify Section 3.8.4, Texture Parameters, p.135**

(modify Table 3.19, p. 137)

| Name | Type | Legal Values |
|---|---|---|
| TEXTURE_UNSIGNED_ REMAP_MODE_NV | enum | EXPAND_NORMAL_NV, NONE |

**Modify Section 3.8.8,  Texture Minification, p.140**

(add after the last paragraph before the "Mipmapping" subsection, p. 144)

After the texture filter is applied, the filtered texture values are optionally rescaled, converting unsigned texture components encoded in the range [0,1] to signed values in the range [-1,+1].  If the texture parameter TEXTURE_UNSIGNED_REMAP_MODE_NV is EXPAND_NORMAL_NV, the filtered values for each unsigned component of the texture is transformed by

```
    tau = 2 * tau - 1.
```

For components

**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.

**GLX Protocol**

None.

**Errors**

None.

**New State**

(add to table 6.15, p. 230)

| Get Value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| TEXTURE_UNSIGNED_REMAP_MODE_NV | nxZ2 | GetTexParameteriv | NONE | unsigned component remapping | 3.8.8 | texture |

**Name**

    NV_vertex_program2

**Name Strings**

    GL_NV_vertex_program2

**Notice**

**IP Status**

    NVIDIA Proprietary.

**Status**

    Implemented in CineFX (NV30) Emulation driver, August 2002.
    Shipping in Release 40 NVIDIA driver for CineFX hardware, January 2003.

**Version**

    Last Modified Date:  $Date: 2003/05/12 $
    NVIDIA Revision: Revision: #30

**Number**

    287

**Dependencies**

Written based on the wording of the OpenGL 1.3 Specification and requires OpenGL 1.3.

Written based on the wording of the NV_vertex_program extension specification, version 1.0.

NV_vertex_program is required.

**Overview**

This extension further enhances the concept of vertex programmability introduced by the NV_vertex_program extension, and extended by NV_vertex_program1_1.  These extensions create a separate vertex program mode where the configurable vertex transformation operations in unextended OpenGL are replaced by a user-defined program.

This extension introduces the VP2 execution environment, which extends the VP1 execution environment introduced in NV_vertex_program.  The VP2 environment provides several language features not present in previous vertex programming execution environments:

* Branch instructions allow a program to jump to another instruction specified in the program.

```
      * Branching support allows for up to four levels of subroutine
        calls/returns.

      * A four-component condition code register allows an application to
        compute a component-wise write mask at run time and apply that mask to
        register writes.

      * Conditional branches are supported, where the condition code register
        is used to determine if a branch should be taken.

      * Programmable user clipping is supported support (via the CLP0-CLP5
        clip distance registers).  Primitives are clipped to the area where
        the interpolated clip distances are greater than or equal to zero.

      * Instructions can perform a component-wise absolute value operation on
        any operand load.

    The VP2 execution environment provides a number of new instructions, and
    extends the semantics of several instructions already defined in
    NV_vertex_program.

      * ARR:  Operates like ARL, except that float-to-int conversion is done
        by rounding.  Equivalent results could be achieved (less efficiently)
        in NV_vertex program using an ADD/ARL sequence and a program parameter
        holding the value 0.5.

      * BRA, CAL, RET:  Branch, subroutine call, and subroutine return
        instructions.

      * COS, SIN:  Adds support for high-precision sine and cosine
        computations.

      * FLR, FRC:  Adds support for computing the floor and fractional portion
        of floating-point vector components.  Equivalent results could be
        achieved (less efficiently) in NV_vertex_program using the EXP
        instruction to compute the fractional portion of one component at a
        time.

      * EX2, LG2:  Adds support for high-precision exponentiation and
        logarithm computations.

      * ARA:  Adds pairs of components of an address register; useful for
        looping and other operations.

      * SEQ, SFL, SGT, SLE, SNE, STR:  Add six new "set on" instructions,
        similar to the SLT and SGE instructions defined in NV_vertex_program.
        Equivalent results could be achieved (less efficiently) in
        NV_vertex_program with multiple SLT, SGE, and arithmetic instructions.

      * SSG:  Adds a new "set sign" operation, which produces a vector holding
        negative one for negative components, zero for components with a value
        of zero, and positive one for positive components.  Equivalent results
        could be achieved (less efficiently) in NV_vertex_program with
        multiple SLT, SGE, and arithmetic instructions.

      * The ARL instruction is extended to operate on four components instead
        of a single component.
```

* All instructions that produce integer or floating-point result vectors have variants that update the condition code register based on the result vector.

This extension also raises some of the resource limitations in the NV_vertex_program extension.

* 256 program parameter registers (versus 96 in NV_vertex_program).

* 16 temporary registers (versus 12 in NV_vertex_program).

* Two four-component integer address registers (versus one single-component register in NV_vertex_program).

* 256 total vertex program instructions (versus 128 in NV_vertex_program).

* Including loops, programs can execute up to 64K instructions.

**Issues**

*This extension builds upon the NV_vertex_program extension. Should this specification contain selected edits to the NV_vertex_program specification or should the specs be unified?*

RESOLVED: Since NV_vertex_program and NV_vertex_program2 programs share many features, the main section of this specification is unified and describes both types of programs. Other sections containing NV_vertex_program features that are unchanged by this extension will not be edited.

*How can a program use condition codes to avoid extra computations?*

Consider the example of evaluating the OpenGL lighting model for a given light. If the diffuse dot product is negative (roughly 1/2 the time for random geometry), the only contribution to the light is ambient. In this case, condition codes and branching can skip over a number of unneeded instructions.

```
        # R0 holds accumulated light color
        # R2 holds normal
        # R3 holds computed light vector
        # R4 holds computed half vector
        # c[0] holds ambient light/material product
        # c[1] holds diffuse light/material product
        # c[2].xyz holds specular light/material product
        # c[2].w   holds specular exponent
        DP3C R1.x, R2, R3;            # diffuse dot product
        ADD  R0, R0, c[0];            # accumulate ambient
        BRA  pointsAway (LT.x)        # skip rest if diffuse dot < 0
        MOV  R1.w, c[2].w;
        DP3  R1.y, R2, R4;            # specular dot product
        LIT  R1, R1;                  # compute expontiated specular
        MAD  R4, c[1], R0.y;          # accumulate diffuse
        MAD  R4, c[2], R0.z;          # accumulate specular
      pointsAway:
        ...                           # continue execution
```

*How can a program use subroutines and branch tables?*

With subroutines, a program can encapsulate a small piece of functionality into a subroutine and call it multiple times, as in CPU code.  Applications will need to identify the registers used to pass data to and from the subroutine.

Subroutines could be used for applications like evaluating lighting equations for a single light.  With conditional branching and subroutines, a variable number of lights (which could even vary per-vertex) can be easily supported.

```
      accumulate:
        # R0 holds the accumulated result
        # R1 holds the value to add
        ADD R0, R1;
        RET;

        # Compute floor(A)*B by repeated addition using a subroutine.  Yes,
        # this is a stupid example.
        #
        # c[0] holds (A,B,0,1).
        # R0 holds the accumulated result
        # R1 holds B, the value to accumulate.
        # R2 holds the number of iterations remaining.
        MOV R0, c[0].z;               # start with zero
        MOV R1, c[0].y;
        FLRC R2.x, c[0].x;
        BRA done (LE.x);
      top:
        CAL accumulate;
        ADDC R2.x, R2.x, -c[0].w;     # decrement count
        BRA top (GT.x);
      done:
        ...
```

*How can conventional OpenGL clip planes be supported in vertex programs?*

The clip distance in the OpenGL specification can be evaluated with a simple DP4 instruction that writes to one of the six clip distance registers. Primitives will automatically be clipped to the half-space where o[CLPx] >= 0, which matches the definition in the spec.

```
    # R0 holds eye coordinates
    # c[0] holds eye-space clip plane coefficients
    DP4 o[CLP0].x, R0, c[0];
```

Note that the clip plane or clip distance volume corresponding to the o[CLPn] register used must be enabled, or no clipping will be performed.

The clip distance registers allow for clip distance volumes to be computed more-or-less arbitrarily. To approximate clipping to a sphere of radius <n>, the following code can be used.

```
    # R0 holds eye coordinates
    # c[0].xyz holds sphere center
    # c[0].w holds the square of the sphere radius
    SUB R1.xyz, R0, c[0];           # distance vector
    DP3 R1.w, R1, R1;               # compute distance squared
    SUB o[CLP0].x, c[0].w, R1.w;    # compute r^2 - d^2
```

Since the clip distance is interpolated linearly over a primitive, the clip distance evaluated at a point will represent a piecewise-linear approximation of the true distance. The approximation will become increasingly more accurate as the primitive is tesselated more finely.

*How can looping be achieved in vertex programs?*

Simple loops can be achieved using a general purpose floating-point register component as a counter. The following code calls a function named "function" <n> times, where <n> is specified in a program parameter register component.

```
    # c[0].x holds the number of iterations to execute.
    # c[1].x holds the constant 1.0.
    MOVC R15.x, c[0].x;
  startLoop:
    CAL  function (GT.x);            # if (counter > 0) function();
    SUBC R15.x, R15.x, c[1].x;       # counter = counter - 1;
    BRA  startLoop (GT.x);           # if (counter > 0) goto start;
  endLoop:
    ...
```

More complex loops (where a separate index may be needed for indexed addressing into the program parameter array) can be achieved using the ARA instruction, which will add the x/z and y/w components of an address register.

```
      # c[0].x holds the number of iterations to execute
      # c[0].y holds the initial index value
      # c[0].z holds the constant -1.0 (used for the iteration count)
      # c[0].w holds the index step value
      ARLC A1, c[0];
    startLoop:
      CAL  function (GT.x);            # if (counter > 0) function();
                                       # Note: A1.y can be used for
                                       # indexing in function().
      ARAC A1.xy, A1;                  # counter = counter - 1;
                                       # index += loopStep;
      BRA  startLoop (GT.x);           # if (counter > 0) goto start;
    endLoop:
      ...
```

*Should this specification add support for vertex state programs beyond the VP1 execution environment?*

No. Vertex state programs are a little-used feature of NV_vertex_program and don't perform particularly well. They are still supported for compatibility with the original NV_vertex_program spec, but they will not be extended to support new features.

*How are NaN's be handled in the "set on" instructions (SEQ, SGE, SGT, SLE, SLT, SNE)? What about MIN, MAX? SSG? When doing condition code tests?*

Any of these instructions involving a NaN operand will produce a NaN result. This behavior differs from the NV_fragment_program extension. There, SEQ, SGE, SGT, SLE, and SLT will produce 0.0 if either operand is a NaN, and SNE will produce 1.0 if either operand is a NaN.

For condition code updates, NaN values will result in "UN" condition codes. All conditionals using a "UN" condition code, except "TR" and "NE" will evaluate to false. This behavior is identical to the functionality in NV_fragment_program.

*How can the various features of this extension be used to provide skinning functionality similar to that in ARB_vertex_blend and ARB_matrix_palette? And how can that functionality be extended?*

Assume an implementation that allows application of up to 8 matrices at once. Further assume that v[12].xyzw and v[13].xyzw hold the set of 8 weights, and v[14].xyzw and v[15].xyzw hold the set of 8 matrix indices. Furthermore, assume that the palette of matrices are stored/tracked at c[0], c[4], c[8], and so on. As an additional optimization, an application can specify that fewer than 8 matrices should be applied by storing a negative palette index immediately after the last index is applied.

Skinning support in this example can be provided by the following code:

```
        ARLC A0, v[14];                 # load 4 palette indices at once
        DP4 R1.x, c[A0.x+0], v[0];      # 1st matrix transform
        DP4 R1.y, c[A0.x+1], v[0];
        DP4 R1.z, c[A0.x+2], v[0];
        DP4 R1.w, c[A0.x+3], v[0];
        MUL R0, R1, v[12].x;            # accumulate weighted sum in R0
        BRA end (LT.y);                 # stop on a negative matrix index
        DP4 R1.x, c[A0.y+0], v[0];      # 2nd matrix transform
        DP4 R1.y, c[A0.y+1], v[0];
        DP4 R1.z, c[A0.y+2], v[0];
        DP4 R1.w, c[A0.y+3], v[0];
        MAD R0, R1, v[12].y, R0;        # accumulate weighted sum in R0
        BRA end (LT.z);                 # stop on a negative matrix index

        ...                             # 3rd and 4th matrix transform

        ARLC A0, v[15];                 # load next four palette indices
        BRA end (LT.x);
        DP4 R1.x, c[A0.x+0], v[0];      # 5th matrix transform
        DP4 R1.y, c[A0.x+1], v[0];
        DP4 R1.z, c[A0.x+2], v[0];
        DP4 R1.w, c[A0.x+3], v[0];
        MAD R0, R1, v[13].x, R0;        # accumulate weighted sum in R0
        BRA end (LT.y);                 # stop on a negative matrix index

        ...                             # 6th, 7th, and 8th matrix transform

      end:
        ...                             # any additional instructions
```

The amount of code used by this example could further be reduced using a subroutine performing four transformations at a time:

```
        ARLC A0, v[14];  # load first four indices
        CAL  skin4;      # do first four transformations
        BRA  end (LT);   # end if any of the first 4 indices was < 0
        ARLC A0, v[15];  # load second four indices
        CAL  skin4;      # do second four transformations
      end:
        ...              # any additional instructions
```

*Why does the RCC instruction exist?*

RESOLVED:  To perform numeric operations that will avoid overflow and underflow issues.

*Should the specification provide more examples?*

RESOLVED:  It would be nice.

## New Procedures and Functions

None.

## New Tokens

None.

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

**Modify Section 2.11, Clipping (p. 39)**

(modify last paragraph, p. 39) When the GL is not in vertex program mode

(section 2.14), this view volume may be further restricted by as many as n client-defined clip planes to generate the clip volume. ...

(add before next-to-last paragraph, p. 40) When the GL is in vertex program mode, the view volume may be restricted to the individual clip distance volumes derived from the per-vertex clip distances (o[CLP0] - o[CLP5]). Clip distance volumes are applied if and only if per-vertex clip distances are not supported in the vertex program execution environment. A point P belonging to the primitive under consideration is in the clip distance volume numbered n if and only if

```
  c_n(P) >= 0,
```

where c_n(P) is the interpolated value of the clip distance CLPn at the point P. For point primitives, c_n(P) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 3.4 and 3.5.

(modify next-to-last paragraph, p.40) Client-defined clip planes or clip distance volumes are enabled with the generic Enable command and disabled with the Disable command. The value of the argument to either command is CLIP PLANEi where i is an integer between 0 and n; specifying a value of i enables or disables the plane equation with index i. The constants obey CLIP PLANEi = CLIP PLANE0 + i.

**Add Section 2.14, Vertex Programs (p. 57).** This section supersedes the similar section added in the NV_vertex_program extension and extended in the NV_vertex_program1_1 extension.

The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable, but essentially hard-wired, sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters.

The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

**Section 2.14.1, Vertex Program Execution Environment**

The vertex program execution environment is an operational model that defines how a program is executed. The execution environment includes a set of instructions, a set of registers, and semantic rules defining how operations are performed. There are three vertex program execution environments, VP1, VP1.1, and VP2. The environment names are taken from the mandatory program prefix strings found at the beginning of all vertex programs. The VP1.1 execution environment is a minor addition to the VP1 execution environment, so references to the VP1 execution environment below apply to both VP1 and VP1.1 execution environments except where otherwise noted.

The vertex program instruction set consists primarily of floating-point 4-component vector operations operating on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from the processing of other vertices. Vertex programs execute without data hazards so results computed in one operation can be used immediately afterwards. Vertex programs produce a set of vertex result vectors that becomes the set of transformed vertex parameters used by primitive assembly.

In the VP1 environment, vertex programs execute a finite fixed sequence of instructions with no branching or looping. In the VP2 environment, vertex programs support conditional and unconditional branches and four levels of subroutine calls.

The vertex program register set consists of six types of registers described in the following sections.

**Section 2.14.1.1, Vertex Attribute Registers**

The Vertex Attribute Registers are sixteen 4-component vector floating-point registers containing the current vertex's per-vertex attributes. These registers are numbered 0 through 15. These registers are private to each vertex program invocation and are initialized at each vertex program invocation by the current vertex attribute state specified with VertexAttribNV commands. These registers are read-only during vertex program execution. The VertexAttribNV commands used to update the vertex attribute registers can be issued both outside and inside of Begin/End pairs. Vertex program execution is provoked by updating vertex attribute zero. Updating vertex attribute zero outside of a Begin/End pair is ignored without generating any error (identical to the Vertex command operation).

The commands

```
  void VertexAttrib{1234}{sfd}NV(uint index, T coords);
  void VertexAttrib{1234}{sfd}vNV(uint index, T coords);
  void VertexAttrib4ubNV(uint index, T coords);
  void VertexAttrib4ubvNV(uint index, T coords);
```

specify the particular current vertex attribute indicated by index. The coordinates for each vertex attribute are named x, y, z, and w. The VertexAttrib1NV family of commands sets the x coordinate to the provided single argument while setting y and z to 0 and w to 1. Similarly, VertexAttrib2NV sets x and y to the specified values, z to 0 and w to 1; VertexAttrib3NV sets x, y, and z, with w set to 1, and VertexAttrib4NV sets all four coordinates. The error INVALID_VALUE is generated if index is greater than 15.

No conversions are applied to the vertex attributes specified as type short, float, or double. However, vertex attributes specified as type ubyte are converted as described by Table 2.6.

The commands

```
  void VertexAttribs{1234}{sfd}vNV(uint index, sizei n, T coords[]);
  void VertexAttribs4ubvNV(uint index, sizei n, GLubyte coords[]);
```

specify a contiguous set of n vertex attributes. The effect of

```
  VertexAttribs{1234}{sfd}vNV(index, n, coords)
```

is the same (assuming no errors) as the command sequence

```
  #define NUM k  /* where k is 1, 2, 3, or 4 components */
  int i;
  for (i=n-1; i>=0; i--) {
    VertexAttrib{NUM}{sfd}vNV(i+index, &coords[i*NUM]);
  }
```

VertexAttribs4ubvNV behaves similarly.

The VertexAttribNV calls equivalent to VertexAttribsNV are issued in reverse order so that vertex program execution is provoked when index is zero only after all the other vertex attributes have first been specified.

The set and operation of vertex attribute registers are identical for both VP1 and VP2 execution environment.

**Section 2.14.1.2, Program Parameter Registers**

The Program Parameter Registers are a set of 4-component floating-point vector registers containing the vertex program parameters. In the VP1 execution environment, there are 96 registers, numbered 0 through 95. In the VP2 execution environment, there are 256 registers, numbered 0 through 255. This relatively large set of registers is intended to hold parameters such as matrices, lighting parameters, and constants required by vertex programs. Vertex program parameter registers can be updated in one of two ways: by the ProgramParameterNV commands outside of a Begin/End pair or by a vertex state program executed outside of a Begin/End pair (vertex state programs are discussed in section 2.14.3).

The commands

```
  void ProgramParameter4fNV(enum target, uint index,
                            float x, float y, float z, float w)
  void ProgramParameter4dNV(enum target, uint index,
                            double x, double y, double z, double w)
```

specify the particular program parameter indicated by index. The coordinates values x, y, z, and w are assigned to the respective components of the particular program parameter. target must be VERTEX_PROGRAM_NV.

The commands

```
  void ProgramParameter4dvNV(enum target, uint index, double *params);
  void ProgramParameter4fvNV(enum target, uint index, float *params);
```

operate identically to ProgramParameter4fNV and ProgramParameter4dNV respectively except that the program parameters are passed as an array of four components.

The error INVALID_VALUE is generated if the specified index is greater than or equal to the number of program parameters in the execution environment (96 for VP1, 256 for VP2).

The commands

```
  void ProgramParameters4dvNV(enum target, uint index,
                              uint num, double *params);
  void ProgramParameters4fvNV(enum target, uint index,
                              uint num, float *params);
```

specify a contiguous set of num program parameters.  The effect is the same (assuming no errors) as

```
  for (i=index; i<index+num; i++) {
    ProgramParameter4{fd}vNV(target, i, &params[i*4]);
  }
```

The error INVALID_VALUE is generated if sum of <index> and <num> is greater than the number of program parameters in the execution environment (96 for VP1, 256 for VP2).

The program parameter registers are shared to all vertex program invocations within a rendering context.  ProgramParameterNV command updates and vertex state program executions are serialized with respect to vertex program invocations and other vertex state program executions.

Writes to the program parameter registers during vertex state program execution can be maskable on a per-component basis.

The initial value of all 96 (VP1) or 256 (VP2) program parameter registers is (0,0,0,0).

**Section 2.14.1.3, Address Registers**

The Address Registers are 4-component vector registers with signed 10-bit integer components.  In the VP1 execution environment, there is only a single address register (A0) and only the x component of the register is accessible.  In the VP2 execution environment, there are two address registers (A0 and A1), of which all four components are accessible.  The address registers are private to each vertex program invocation and are initialized to (0,0,0,0) at every vertex program invocation.  These registers can be written during vertex program execution (but not read) and their values can be used for as a relative offset for reading vertex program parameter registers.  Only the vertex program parameter registers can be read using relative addressing (writes using relative addressing are not supported).

See the discussion of relative addressing of program parameters in section 2.14.2.1 and the discussion of the ARL instruction in section 2.14.3.4.

**Section 2.14.1.4, Temporary Registers**

The Temporary Registers are 4-component floating-point vector registers used to hold temporary results during vertex program execution. In the VP1 execution environment, there are 12 temporary registers, numbered 0 through 11. In the VP2 execution environment, there are 16 temporary registers, numbered 0 through 15. These registers are private to each vertex program invocation and initialized to (0,0,0,0) at every vertex program invocation. These registers can be read and written during vertex program execution. Writes to these registers can be maskable on a per-component basis.

In the VP2 execution environment, there is one additional temporary pseudo-register, "CC". CC is treated as unnumbered, write-only temporary register, whose sole purpose is to allow instructions to modify the condition code register (section 2.14.1.6) without overwriting the contents of any temporary register.

**Section 2.14.1.5, Vertex Result Registers**

The Vertex Result Registers are 4-component floating-point vector registers used to write the results of a vertex program. There are 15 result registers in the VP1 execution environment, and 21 in the VP2 execution environment. Each register value is initialized to (0,0,0,1) at the invocation of each vertex program. Writes to the vertex result registers can be maskable on a per-component basis. These registers are named in Table X.1 and further discussed below.

| Vertex Result Register Name | Description | Component Interpretation |
|---|---|---|
| HPOS | Homogeneous clip space position | (x,y,z,w) |
| COL0 | Primary color (front-facing) | (r,g,b,a) |
| COL1 | Secondary color (front-facing) | (r,g,b,a) |
| BFC0 | Back-facing primary color | (r,g,b,a) |
| BFC1 | Back-facing secondary color | (r,g,b,a) |
| FOGC | Fog coordinate | (f,*,*,*) |
| PSIZ | Point size | (p,*,*,*) |
| TEX0 | Texture coordinate set 0 | (s,t,r,q) |
| TEX1 | Texture coordinate set 1 | (s,t,r,q) |
| TEX2 | Texture coordinate set 2 | (s,t,r,q) |
| TEX3 | Texture coordinate set 3 | (s,t,r,q) |
| TEX4 | Texture coordinate set 4 | (s,t,r,q) |
| TEX5 | Texture coordinate set 5 | (s,t,r,q) |
| TEX6 | Texture coordinate set 6 | (s,t,r,q) |
| TEX7 | Texture coordinate set 7 | (s,t,r,q) |
| CLP0(*) | Clip distance 0 | (d,*,*,*) |
| CLP1(*) | Clip distance 1 | (d,*,*,*) |
| CLP2(*) | Clip distance 2 | (d,*,*,*) |
| CLP3(*) | Clip distance 3 | (d,*,*,*) |
| CLP4(*) | Clip distance 4 | (d,*,*,*) |
| CLP5(*) | Clip distance 5 | (d,*,*,*) |

**Table X.1: Vertex Result Registers. (*) Registers CLP0 through CLP5, are available only in the VP2 execution environment.**

HPOS is the transformed vertex's homogeneous clip space position. The vertex's homogeneous clip space position is converted to normalized device coordinates and transformed to window coordinates as described at the end of section 2.10 and in section 2.11. Further processing (subsequent to vertex program termination) is responsible for clipping primitives assembled from vertex program-generated vertices as described in section 2.10 but all client-defined clip planes are treated as if they are disabled when vertex program mode is enabled.

Four distinct color results can be generated for each vertex. COL0 is the transformed vertex's front-facing primary color. COL1 is the transformed vertex's front-facing secondary color. BFC0 is the transformed vertex's back-facing primary color. BFC1 is the transformed vertex's back-facing secondary color.

Primitive coloring may operate in two-sided color mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_NV. The selection between the back-facing colors and the front-facing colors depends on the primitive of which the vertex is a part. If the primitive is a point or a line segment, the front-facing colors are always selected. If the primitive is a polygon and two-sided color mode is disabled, the front-facing colors are selected. If it is a polygon and two-sided color mode is enabled, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. This facingness determination is identical to the two-sided lighting facingness determination described in section 2.13.1.

The selected primary and secondary colors for each primitive are clamped to the range [0,1] and then interpolated across the assembled primitive during rasterization with at least 8-bit accuracy for each color component.

FOGC is the transformed vertex's fog coordinate. The register's first floating-point component is interpolated across the assembled primitive during rasterization and used as the fog distance to compute per-fragment the fog factor when fog is enabled. However, if both fog and vertex program mode are enabled, but the FOGC vertex result register is not written, the fog factor is overridden to 1.0. The register's other three components are ignored.

Point size determination may operate in program-specified point size mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_NV. If the vertex is for a point primitive and the mode is enabled and the PSIZ vertex result is written, the point primitive's size is determined by the clamped x component of the PSIZ register. Otherwise (because vertex program mode is disabled, program-specified point size mode is disabled, or because the vertex program did not write PSIZ), the point primitive's size is determined by the point size state (the state specified using the PointSize command).

The PSIZ register's x component is clamped to the range zero through either the hi value of ALIASED_POINT_SIZE_RANGE if point smoothing is disabled or the hi value of the SMOOTH_POINT_SIZE_RANGE if point smoothing is enabled. The register's other three components are ignored.

If the vertex is not for a point primitive, the value of the PSIZ vertex result register is ignored.

TEX0 through TEX7 are the transformed vertex's texture coordinate sets for texture units 0 through 7. These floating-point coordinates are interpolated across the assembled primitive during rasterization and used for accessing textures. If the number of texture units supported is less than eight, the values of vertex result registers that do not correspond to existent texture units are ignored.

CLP0 through CLP5, available only in the VP2 execution environment, are the transformed vertex's clip distances. These floating-point coordinates are used by post-vertex program clipping process (see section 2.11).

**Section 2.14.1.6, The Condition Code Register**

The VP2 execution environment provides a single four-component vector called the condition code register. 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 the condition code register. When a vertex program instruction updates the 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 register is initialized to a vector of EQ values each time a vertex program executes.

There is no condition code register available in the VP1 execution environment.

**Section 2.14.1.7, Semantic Meaning for Vertex Attributes and Program Parameters**

One important distinction between the conventional GL vertex transformation mode and the vertex program mode is that per-vertex parameters and other state parameters in vertex program mode do not have dedicated semantic interpretations the way that they do with the conventional GL vertex transformation mode.

For example, in the conventional GL vertex transformation mode, the Normal command specifies a per-vertex normal. The semantic that the Normal command supplies a normal for lighting is established because that is how the per-vertex attribute supplied by the Normal command is used by the conventional GL vertex transformation mode. Similarly, other state parameters such as a light source position have semantic interpretations based on how the conventional GL vertex transformation model uses each particular parameter.

In contrast, vertex attributes and program parameters for vertex programs have no pre-defined semantic meanings. The meaning of a vertex attribute or program parameter in vertex program mode is defined by how the vertex attribute or program parameter is used by the current vertex program to compute and write values to the Vertex Result Registers. This is the reason that per-vertex attributes and program parameters for vertex programs are numbered instead of named.

For convenience however, the existing per-vertex parameters for the conventional GL vertex transformation mode (vertices, normals, colors, fog coordinates, vertex weights, and texture coordinates) are aliased to numbered vertex attributes. This aliasing is specified in Table X.2. The table includes how the various conventional components map to the 4-component vertex attribute components.

| Vertex Attribute Register Number | Conventional Per-vertex Parameter | Conventional Per-vertex Parameter Command | Conventional Component Mapping |
|---|---|---|---|
| 0 | vertex position | Vertex | x,y,z,w |
| 1 | vertex weights | VertexWeightEXT | w,0,0,1 |
| 2 | normal | Normal | x,y,z,1 |
| 3 | primary color | Color | r,g,b,a |
| 4 | secondary color | SecondaryColorEXT | r,g,b,1 |
| 5 | fog coordinate | FogCoordEXT | fc,0,0,1 |
| 6 | - | - | - |
| 7 | - | - | - |
| 8 | texture coord 0 | MultiTexCoord(GL_TEXTURE0_ARB, ...) | s,t,r,q |
| 9 | texture coord 1 | MultiTexCoord(GL_TEXTURE1_ARB, ...) | s,t,r,q |
| 10 | texture coord 2 | MultiTexCoord(GL_TEXTURE2_ARB, ...) | s,t,r,q |
| 11 | texture coord 3 | MultiTexCoord(GL_TEXTURE3_ARB, ...) | s,t,r,q |
| 12 | texture coord 4 | MultiTexCoord(GL_TEXTURE4_ARB, ...) | s,t,r,q |
| 13 | texture coord 5 | MultiTexCoord(GL_TEXTURE5_ARB, ...) | s,t,r,q |
| 14 | texture coord 6 | MultiTexCoord(GL_TEXTURE6_ARB, ...) | s,t,r,q |
| 15 | texture coord 7 | MultiTexCoord(GL_TEXTURE7_ARB, ...) | s,t,r,q |

**Table X.2: Aliasing of vertex attributes with conventional per-vertex parameters.**

Only vertex attribute zero is treated specially because it is the attribute that provokes the execution of the vertex program; this is the attribute that aliases to the Vertex command's vertex coordinates.

The result of a vertex program is the set of post-transformation vertex parameters written to the Vertex Result Registers. All vertex programs must write a homogeneous clip space position, but the other Vertex Result Registers can be optionally written.

Clipping and culling are not the responsibility of vertex programs because these operations assume the assembly of multiple vertices into a primitive. View frustum clipping is performed subsequent to vertex program execution. Clip planes are not supported in the VP1 execution environment. Clip planes are supported indirectly via the clip distance (o[CLPx]) registers in the VP2 execution environment.

**Section 2.14.1.8, Vertex Program Specification**

Vertex programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program.

The command

```
  LoadProgramNV(enum target, uint id, sizei len,
                const ubyte *program);
```

loads a vertex program when the target parameter is VERTEX_PROGRAM_NV. Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the positive integers (zero is reserved). The error INVALID_VALUE occurs if a program is loaded with an id of zero. The error INVALID_OPERATION is generated if a program is loaded for an id that is currently loaded with a program of a different program target. Managing the program name space and binding to vertex programs is discussed later in section 2.14.1.8.

program is a pointer to an array of ubytes that represents the program being loaded. The length of the array is indicated by len.

A second program target type known as vertex state programs is discussed in 2.14.4.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for several types of vertex programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

The grammar provides for three different vertex program types, corresponding to the three vertex program execution environments. VP1, VP1.1, and VP2 programs match the grammar rules <vp1-program>, <vp11-program>, and <vp2-program>, respectively. Some grammar rules correspond to features or instruction forms available only in certain execution environments. Rules beginning with the prefix "vp1-" are available only to VP1 and VP1.1 programs. Rules beginning with the prefixes "vp11-" and "vp2-" are available only to VP1.1 and VP2 programs, respectively.

```
<program>              ::= <vp1-program>
                         | <vp11-program>
                         | <vp2-program>

<vp1-program>          ::= "!!VP1.0" <programBody> "END"

<vp11-program>         ::= "!!VP1.1" <programBody> "END"

<vp2-program>          ::= "!!VP2.0" <programBody> "END"

<programBody>          ::= <optionSequence> <programText>

<optionSequence>       ::= <option> <optionSequence>
                         | ""

<option>               ::= "OPTION" <vp11-option> ";"
                         | "OPTION" <vp2-option> ";"

<vp11-option>          ::= "NV_position_invariant"

<vp2-option>           ::= "NV_position_invariant"

<programText>          ::= <programTextItem> <programText>
                         | ""
```

```
    <programTextItem>      ::= <instruction> ";"
                             | <vp2-instructionLabel>

    <instruction>          ::= <ARL-instruction>
                             | <VECTORop-instruction>
                             | <SCALARop-instruction>
                             | <BINop-instruction>
                             | <TRIop-instruction>
                             | <vp2-BRA-instruction>
                             | <vp2-RET-instruction>
                             | <vp2-ARA-instruction>

    <ARL-instruction>      ::= <vp1-ARL-instruction>
                             | <vp2-ARL-instruction>

    <vp1-ARL-instruction>  ::= "ARL" <maskedAddrReg> "," <scalarSrc>

    <vp2-ARL-instruction>  ::= <vp2-ARLop> <maskedAddrReg> "," <vectorSrc>

    <vp2-ARLop>            ::= "ARL" | "ARLC"
                             | "ARR" | "ARRC"

    <VECTORop-instruction> ::= <VECTORop> <maskedDstReg> "," <vectorSrc>

    <VECTORop>             ::= "LIT"
                             | "MOV"
                             | <vp11-VECTORop>
                             | <vp2-VECTORop>

    <vp11-VECTORop>        ::= "ABS"

    <vp2-VECTORop>         ::=         "ABSC"
                             | "FLR" | "FLRC"
                             | "FRC" | "FRCC"
                             |         "LITC"
                             |         "MOVC"
                             | "SSG" | "SSGC"

    <SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrc>

    <SCALARop>             ::= "EXP"
                             | "LOG"
                             | "RCP"
                             | "RSQ"
                             | <vp2-SCALARop>

    <vp2-SCALARop>         ::= "COS"  | "COSC"
                             | "EX2"  | "EX2C"
                             | "LG2"  | "LG2C"
                             |          "EXPC"
                             |          "LOGC"
                             |          "RCPC"
                             |          "RSQC"


    <BINop-instruction>    ::= <BINop> <maskedDstReg> "," <vectorSrc> ","
                               <vectorSrc>
    <BINop>                 ::= "ADD"
                             | "DP3"
                             | "DP4"
                             | "DST"
                             | "MAX"
                             | "MIN"
                             | "MUL"
                             | "SGE"
                             | "SLT"
                             | <vp11-BINop>
                             | <vp2-BINop>

    <vp11-BINop>            ::= "DPH"

    <vp2-BINop>             ::=         "ADDC"
                             |         "DP3C"
                             |         "DP4C"
                             |         "DPHC"
                             |         "DSTC"
                             |         "MAXC"
                             |         "MINC"
                             |         "MULC"
                             | "SEQ" | "SEQC"
                             | "SFL" | "SFLC"
                             |         "SGEC"
                             | "SGT" | "SGTC"
                             |         "SLTC"
                             | "SLE" | "SLEC"
                             | "SNE" | "SNEC"
                             | "STR" | "STRC"

    <TRIop-instruction>     ::= <TRIop> <maskedDstReg> "," <vectorSrc> ","
                               <vectorSrc> "," <vectorSrc>

    <TRIop>                 ::= "MAD"
                             | <vp2-TRIop>

    <vp2-TRIop>             ::= "MADC"

    <vp2-BRA-instruction>   ::= <vp2-BRANCHop> <vp2-branchLabel>
                                 <vp2-branchCondition>

    <vp2-BRANCHop>          ::= "BRA"
                             | "CAL"

    <vp2-RET-instruction>   ::= "RET" <vp2-branchCondition>

    <vp2-ARA-instruction>   ::= <vp2-ARAop> <maskedAddrReg> "," <addrRegister>

    <vp2-ARAop>             ::= "ARA" | "ARAC"

    <scalarSrc>             ::= <baseScalarSrc>
                             | <vp2-absScalarSrc>

    <vp2-absScalarSrc>      ::= <optionalSign> "|" <baseScalarSrc> "|"
    <baseScalarSrc>        ::= <optionalSign> <srcRegister> <scalarSuffix>

    <vectorSrc>            ::= <baseVectorSrc>
                             | <vp2-absVectorSrc>

    <vp2-absVectorSrc>     ::= <optionalSign> "|" <baseVectorSrc> "|"

    <baseVectorSrc>        ::= <optionalSign> <srcRegister> <swizzleSuffix>

    <srcRegister>          ::= <vtxAttribRegister>
                             | <progParamRegister>
                             | <tempRegister>

    <maskedDstReg>         ::= <dstRegister> <optionalWriteMask>
                                   <optionalCCMask>

    <dstRegister>          ::= <vtxResultRegister>
                             | <tempRegister>
                             | <vp2-nullRegister>

    <vp2-nullRegister>     ::= "CC"

    <vp2-branchCondition>  ::= <optionalCCMask>

    <vtxAttribRegister>    ::= "v" "[" vtxAttribRegNum "]"

    <vtxAttribRegNum>      ::= decimal integer from 0 to 15 inclusive
                             | "OPOS"
                             | "WGHT"
                             | "NRML"
                             | "COL0"
                             | "COL1"
                             | "FOGC"
                             | "TEX0"
                             | "TEX1"
                             | "TEX2"
                             | "TEX3"
                             | "TEX4"
                             | "TEX5"
                             | "TEX6"
                             | "TEX7"

    <progParamRegister>    ::= <absProgParamReg>
                             | <relProgParamReg>

    <absProgParamReg>      ::= "c" "[" <progParamRegNum> "]"

    <progParamRegNum>      ::= <vp1-progParamRegNum>
                             | <vp2-progParamRegNum>

    <vp1-progParamRegNum>  ::= decimal integer from 0 to 95 inclusive

    <vp2-progParamRegNum>  ::= decimal integer from 0 to 255 inclusive

    <relProgParamReg>      ::= "c" "[" <scalarAddr> <relProgParamOffset> "]"
    <relProgParamOffset>   ::= ""
                             | "+" <progParamPosOffset>
                             | "-" <progParamNegOffset>

    <progParamPosOffset>   ::= <vp1-progParamPosOff>
                             | <vp2-progParamPosOff>

    <vp1-progParamPosOff>  ::= decimal integer from 0 to 63 inclusive

    <vp2-progParamPosOff>  ::= decimal integer from 0 to 255 inclusive

    <progParamNegOffset>   ::= <vp1-progParamNegOff>
                             | <vp2-progParamNegOff>

    <vp1-progParamNegOff>  ::= decimal integer from 0 to 64 inclusive

    <vp2-progParamNegOff>  ::= decimal integer from 0 to 256 inclusive

    <tempRegister>         ::= "R0"  | "R1"  | "R2"  | "R3"
                             | "R4"  | "R5"  | "R6"  | "R7"
                             | "R8"  | "R9"  | "R10" | "R11"

    <vp2-tempRegister>     ::= "R12" | "R13" | "R14" | "R15"

    <vtxResultRegister>    ::= "o" "[" <vtxResultRegName> "]"

    <vtxResultRegName>     ::= "HPOS"
                             | "COL0"
                             | "COL1"
                             | "BFC0"
                             | "BFC1"
                             | "FOGC"
                             | "PSIZ"
                             | "TEX0"
                             | "TEX1"
                             | "TEX2"
                             | "TEX3"
                             | "TEX4"
                             | "TEX5"
                             | "TEX6"
                             | "TEX7"
                             | <vp2-resultRegName>

    <vp2-resultRegName>    ::= "CLP0"
                             | "CLP1"
                             | "CLP2"
                             | "CLP3"
                             | "CLP4"
                             | "CLP5"

    <scalarAddr>           ::= <addrRegister> "." <addrRegisterComp>

    <maskedAddrReg>        ::= <addrRegister> <addrWriteMask>

    <addrRegister>         ::= "A0"
                             | <vp2-addrRegister>
    <vp2-addrRegister>     ::= "A1"

    <addrRegisterComp>     ::= "x"
                             | <vp2-addrRegisterComp>

    <vp2-addrRegisterComp> ::= "y"
                             | "z"
                             | "w"

    <addrWriteMask>        ::= "." "x"
                             | <vp2-addrWriteMask>

    <vp2-addrWriteMask>     ::= ""
                             | "."     "y"
                             | "." "x" "y"
                             | "."         "z"
                             | "." "x"     "z"
                             | "."     "y" "z"
                             | "." "x" "y" "z"
                             | "."             "w"
                             | "." "x"         "w"
                             | "."     "y"     "w"
                             | "." "x" "y"     "w"
                             | "."         "z" "w"
                             | "." "x"     "z" "w"
                             | "."     "y" "z" "w"
                             | "." "x" "y" "z" "w"


    <optionalSign>         ::= ""
                             | "-"
                             | <vp2-optionalSign>

    <vp2-optionalSign>     ::= "+"

    <vp2-instructionLabel> ::= <vp2-branchLabel> ":"

    <vp2-branchLabel>      ::= <identifier>

    <optionalWriteMask>    ::= ""
                             | "." "x"
                             | "."     "y"
                             | "." "x" "y"
                             | "."         "z"
                             | "." "x"     "z"
                             | "."     "y" "z"
                             | "." "x" "y" "z"
                             | "."             "w"
                             | "." "x"         "w"
                             | "."     "y"     "w"
                             | "." "x" "y"     "w"
                             | "."         "z" "w"
                             | "." "x"     "z" "w"
                             | "."     "y" "z" "w"
                             | "." "x" "y" "z" "w"
```

```
<optionalCCMask>       ::= ""
                         | <vp2-ccMask>

<vp2-ccMask>           ::= "(" <vp2-ccMaskRule> <swizzleSuffix> ")"

<vp2-ccMaskRule>       ::= "EQ" | "GE" | "GT" | "LE" | "LT" | "NE"
                         | "TR" | "FL"

<scalarSuffix>         ::= "." <component>

<swizzleSuffix>        ::= ""
                         | "." <component>
                         | "." <component> <component>
                               <component> <component>

<component>            ::= "x"
                         | "y"
                         | "z"
                         | "w"
```

The <identifier> rule matches a sequence of one or more letters ("A" through "Z", "a" through "z", and "_") and digits ("0" through "9"); the first character must be a letter. The underscore ("_") counts as a letter. Upper and lower case letters are different (names are case-sensitive).

The <vertexAttribRegNum> rule matches both register numbers 0 through 15 and a set of mnemonics that abbreviate the aliasing of conventional per-vertex parameters to vertex attribute register numbers. Table X.3 shows the mapping from mnemonic to vertex attribute register number and what the mnemonic abbreviates.

| Mnemonic | Vertex Attribute Register Number | Meaning |
|---|---|---|
| "OPOS" | 0 | object position |
| "WGHT" | 1 | vertex weight |
| "NRML" | 2 | normal |
| "COL0" | 3 | primary color |
| "COL1" | 4 | secondary color |
| "FOGC" | 5 | fog coordinate |
| "TEX0" | 8 | texture coordinate 0 |
| "TEX1" | 9 | texture coordinate 1 |
| "TEX2" | 10 | texture coordinate 2 |
| "TEX3" | 11 | texture coordinate 3 |
| "TEX4" | 12 | texture coordinate 4 |
| "TEX5" | 13 | texture coordinate 5 |
| "TEX6" | 14 | texture coordinate 6 |
| "TEX7" | 15 | texture coordinate 7 |

**Table X.3: The mapping between vertex attribute register numbers, mnemonics, and meanings.**

A vertex program fails to load if it does not write at least one component of the HPOS register.

A vertex program fails to load in the VP1 execution environment if it contains more than 128 instructions. A vertex program fails to load in the VP2 execution environment if it contains more than 256 instructions. Each block of text matching the <instruction> rule counts as an instruction.

A vertex program fails to load if any instruction sources more than one unique program parameter register. An instruction can match the <progParamRegister> rule more than once only if all such matches are identical.

A vertex program fails to load if any instruction sources more than one unique vertex attribute register. An instruction can match the <vtxAttribRegister> rule more than once only if all such matches refer to the same register.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the last loaded program string indicating where the first error in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

**Section 2.14.1.9, Vertex Program Binding and Program Management**

The current vertex program is invoked whenever vertex attribute zero is updated (whether by a VertexAttributeNV or Vertex command). The current vertex program is updated by

```
  BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV. This binds the vertex program named by id as the current vertex program. The error INVALID_OPERATION is generated if id names a program that is not a vertex program (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error.
In particular, binding to program id zero does not generate an error.

However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

Programs are deleted by calling

```
  void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
  void GenProgramsNV(sizei n, uint *ids);
```

returns n previously unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV. The error INVALID_VALUE is generated if n is negative.

An implementation may choose to establish a working set of programs on which binding and ExecuteProgramNV operations (execute programs are explained in section 2.14.4) are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
  boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
  void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

**Section 2.14.2, Vertex Program Operation**

In the VP1 execution environment, there are twenty-one vertex program instructions. Four instructions (ABS, DPH, RCC, and SUB) are available only in the VP1.1 execution environment. The instructions and their respective input and output parameters are summarized in Table X.4.

| Instruction | Inputs | Output | Description |
|---|---|---|---|
| ABS(*) | v | v | absolute value |
| ADD | v,v | v | add |
| ARL | v | as | address register load |
| DP3 | v,v | ssss | 3-component dot product |
| DP4 | v,v | ssss | 4-component dot product |
| DPH(*) | v,v | ssss | homogeneous dot product |
| DST | v,v | v | distance vector |
| EXP | s | v | exponential base 2 (approximate) |
| LIT | v | v | compute light coefficients |
| LOG | s | v | logarithm base 2 (approximate) |
| MAD | v,v,v | v | multiply and add |
| MAX | v,v | v | maximum |
| MIN | v,v | v | minimum |
| MOV | v | v | move |
| MUL | v,v | v | multiply |
| RCC(*) | s | ssss | reciprocal (clamped) |
| RCP | s | ssss | reciprocal |
| RSQ | s | ssss | reciprocal square root |
| SGE | v,v | v | set on greater than or equal |
| SLT | v,v | v | set on less than |
| SUB(*) | v,v | v | subtract |

**Table X.4: Summary of vertex program instructions in the VP1 execution environment. "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 vector, "as" indicates a single component of an address register.**

In the VP2 execution environment, are thirty-nine vertex program instructions. Vertex program instructions may have an optional suffix of "C" to allow an update of the condition code register (section 2.14.1.6). For example, there are two instructions to perform vector addition, "ADD" and "ADDC". The vertex program instructions available in the VP2 execution environment and their respective input and output parameters are summarized in Table X.5.

| Instruction | Inputs | Output | Description |
|---|---|---|---|
| ABS[C] | v | v | absolute value |
| ADD[C] | v,v | v | add |
| ARA[C] | av | av | address register add |
| ARL[C] | v | av | address register load |
| ARR[C] | v | av | address register load (with round) |
| BRA | as | none | branch |
| CAL | as | none | subroutine call |
| COS[C] | s | ssss | cosine |
| DP3[C] | v,v | ssss | 3-component dot product |
| DP4[C] | v,v | ssss | 4-component dot product |
| DPH[C] | v,v | ssss | homogeneous dot product |
| DST[C] | v,v | v | distance vector |
| EX2[C] | s | ssss | exponential base 2 |
| EXP[C] | s | v | exponential base 2 (approximate) |
| FLR[C] | v | v | floor |
| FRC[C] | v | v | fraction |
| LG2[C] | s | ssss | logarithm base 2 |
| LIT[C] | v | v | compute light coefficients |
| LOG[C] | s | v | logarithm base 2 (approximate) |
| MAD[C] | v,v,v | v | multiply and add |
| MAX[C] | v,v | v | maximum |
| MIN[C] | v,v | v | minimum |
| MOV[C] | v | v | move |
| MUL[C] | v,v | v | multiply |
| RCC[C] | s | ssss | reciprocal (clamped) |
| RCP[C] | s | ssss | reciprocal |
| RET | none | none | subroutine call return |
| RSQ[C] | s | ssss | reciprocal square root |
| SEQ[C] | v,v | v | set on equal |
| SFL[C] | v,v | v | set on false |
| SGE[C] | v,v | v | set on greater than or equal |
| SGT[C] | v,v | v | set on greater than |
| SIN[C] | s | ssss | sine |
| SLE[C] | v,v | v | set on less than or equal |
| SLT[C] | v,v | v | set on less than |
| SNE[C] | v,v | v | set on not equal |
| SSG[C] | v | v | set sign |
| STR[C] | v,v | v | set on true |
| SUB[C] | v,v | v | subtract |

**Table X.5: Summary of vertex program instructions in the VP2 execution environment. "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 vector, "av" indicates a full address register, "as" indicates a single component of an address register.**

**Section 2.14.2.1, Vertex Program Operands**

Most vertex program instructions operate on floating-point vectors, floating-point scalars, or integer scalars as, indicated in the grammar (see section 2.14.1.8) by the rules <vectorSrc>, <scalarSrc>, and <scalarAddr>, respectively.

The basic set of floating-point scalar operands is defined by the grammar rule <baseScalarSrc>. Scalar operands are single components of vertex attribute, program parameter, or temporary registers, as allowed by the <srcRegister> rule. A vector component is selected by the <scalarSuffix> rule, where the characters "x", "y", "z", and "w" select the x, y, z, and w components, respectively, of the vector.

The basic set of floating-point vector operands is defined by the grammar rule <baseVectorSrc>. Vector operands can be obtained from vertex attribute, program parameter, or temporary registers as allowed by the <srcRegister> rule.

Basic vector operands can be swizzled according to the <swizzleSuffix> rule. In its most general form, the <swizzleSuffix> rule matches the pattern ".????" where each question mark is replaced with one of "x", "y", "z", or "w". For such patterns, the x, y, z, and w components of the operand are taken from the vector components named by the first, second, third, and fourth character of the pattern, respectively. For example, if the swizzle suffix is ".yzzx" and the specified source contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}.

If the <swizzleSuffix> rule matches "", it is treated as though it were ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

Floating-point scalar or vector operands can optionally be negated according to the <negate> rules in <baseScalarSrc> and <baseVectorSrc>. If the <negate> matches "-", each operand or operand component is negated.

In the VP2 execution environment, a component-wise absolute value operation is performed on an operand if the <scalarSrc> or <vectorSrc> rules match <vp2-absScalarSrc> or <vp2-absVectorSrc>. In this case, the absolute value of each component of the operand is taken. In addition, if the <negate> rule in <vp2-absScalarSrc> or <vp2-absVectorSrc> matches "-", each component is subsequently negated.

Integer scalar operands are single components of one of the address register vectors, as identified by the <addrRegister> rule. A vector component is selected by the <scalarSuffix> rule in the same manner as floating-point scalar operands. Negation and absolute value operations are not available for integer scalar operands.

The following pseudo-code spells out the operand generation process. In the pseudo-code, "float" and "int" are floating-point and integer scalar types, while "floatVec" and "intVec" are four-component vectors. "source" is the register used for the operand, matching the <srcRegister> or <addrRegister> rules. "absolute" is TRUE if the operand matches the <vp2-absScalarSrc> or <vp2-absVectorSrc> rules, and FALSE otherwise. "negateBase" is TRUE if the <negate> rule in <baseScalarSrc> or

    <baseVectorSrc> matches "-" and FALSE otherwise.  "negateAbs" is TRUE if
    the <negate> rule in <vp2-absScalarSrc> or <vp2-absVectorSrc> matches "-"
    and FALSE otherwise.  The ".c***", ".*c**", ".**c*", ".***c" modifiers
    refer to the x, y, z, and w components obtained by the swizzle operation.

```
      floatVec VectorLoad(floatVec source)
      {
          floatVec operand;

          operand.x = source.c***;
          operand.y = source.*c**;
          operand.z = source.**c*;
          operand.w = source.***c;
          if (negateBase) {
             operand.x = -operand.x;
             operand.y = -operand.y;
             operand.z = -operand.z;
             operand.w = -operand.w;
          }
          if (absolute) {
             operand.x = abs(operand.x);
             operand.y = abs(operand.y);
             operand.z = abs(operand.z);
             operand.w = abs(operand.w);
          }
          if (negateAbs) {
             operand.x = -operand.x;
             operand.y = -operand.y;
             operand.z = -operand.z;
             operand.w = -operand.w;
          }

          return operand;
      }

      float ScalarLoad(floatVec source)
      {
          float operand;

          operand = source.c***;
          if (negateBase) {
            operand = -operand;
          }
          if (absolute) {
             operand = abs(operand);
          }
          if (negateAbs) {
            operand = -operand;
          }

          return operand;
      }
```

```
  intVec AddrVectorLoad(intVec addrReg)
  {
      intVec operand;

      operand.x = source.c***;
      operand.y = source.*c**;
      operand.z = source.**c*;
      operand.w = source.***c;

      return operand;
  }

  int AddrScalarLoad(intVec addrReg)
  {
      return source.c***;
  }
```

If an operand is obtained from a program parameter register, by matching the <progParamRegister> rule, the register number can be obtained by absolute or relative addressing.

When absolute addressing is used, by matching the <absProgParamReg> rule, the program parameter register number is the number matching the <progParamRegNum>.

When relative addressing is used, by matching the <relProgParamReg> rule, the program parameter register number is computed during program execution. An index is computed by adding the integer scalar operand specified by the <scalarAddr> rule to the positive or negative offset specified by the <progParamOffset> rule. If <progParamOffset> matches "", an offset of zero is used.

The following pseudo-code spells out the process of loading a program parameter. "addrReg" refers to the address register used for relative addressing, "absolute" is TRUE if the operand uses absolute addressing and FALSE otherwise. "paramNumber" is the program parameter number for absolute addressing; "paramOffset" is the program parameter offset for relative addressing. "paramRegiser" is an array holding the complete set of program parameter registers.

```
  floatVec ProgramParameterLoad(intVec addrReg)
  {
    int index;

    if (absolute) {
      index = paramNumber;
    } else {
      index = AddrScalarLoad(addrReg) + paramOffset
    }

    return paramRegister[index];
  }
```

**Section 2.14.2.2, Vertex Program Destination Register Update**

Most vertex program instructions write a 4-component result vector to a single temporary, vertex result, or address register. Writes to individual components of the destination register are controlled by individual component write masks specified as part of the instruction. In the VP2 execution environment, writes are additionally controlled by the a condition code write mask, which is computed at run time.

The component write mask is specified by the <optionalWriteMask> rule found in the <maskedDstReg> or <maskedAddrReg> 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.

In the VP2 execution environment, the condition code write mask is specified by the <optionalCCMask> rule found in the <maskedDstReg> and <maskedAddrReg> rules. If the condition code mask matches "", all components are enabled. Otherwise, the 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.

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 the component write mask, and the optional condition code mask (if applicable). Otherwise, the component of the destination register remains unchanged.

In the VP2 execution environment, 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. 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, "instrmask" refers to the component write mask given by the <optionalWriteMask> rule. In the VP1 execution environment, "ccMaskRule" is always "" and "updatecc" is always FALSE. In the VP2 execution environment, "ccMaskRule" refers to the condition code mask rule given by <vp2-optionalCCMask> 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;

      // 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 (updatecc) mergedCC.x = GenerateCC(result.x);
      }
      if (instrMask.y && TestCC(cc.*c**)) {
          merged.y = result.y;
          if (updatecc) mergedCC.y = GenerateCC(result.y);
      }
      if (instrMask.z && TestCC(cc.**c*)) {
          merged.z = result.z;
          if (updatecc) mergedCC.z = GenerateCC(result.z);
      }
      if (instrMask.w && TestCC(cc.***c)) {
          merged.w = result.w;
          if (updatecc) mergedCC.w = GenerateCC(result.w);
      }

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

**Section 2.14.2.3, Vertex Program Execution**

In the VP1 execution environment, vertex programs consist of a sequence of instructions without no support for branching. Vertex programs begin by executing the first instruction in the program, and execute instructions in the order specified in the program until the last instruction is reached.

VP2 vertex programs can contain one or more instruction labels, matching the grammar rule <vp2-instructionLabel>. An instruction label can be referred to explicitly in branch (BRA) or subroutine call (CAL) instructions. Instruction labels can be defined or used at any point in the body of a program, and can be used in instructions before being defined in the program string.

VP2 vertex program branching instructions can be conditional. The branch condition is specified by the <vp2-conditionMask> 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 vertex 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)
    BRA label1 (LT.xyzw);
    MOV R0,R1;             # not executed
  label1:
    BRA label2 (LT.wyzw);
    MOV R0,R2;             # executed
  label2:
```

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

VP2 vertex programs can 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 vertex program will terminate if a CAL instruction is executed with four entries already in the call stack or if a RET instruction is executed with an empty call stack.

If a VP2 vertex 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 vertex program will terminate after executing a RET instruction with an empty call stack. A vertex program will also terminate after executing the last instruction in the program, unless that instruction was a taken branch.

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

A vertex program will terminate abnormally if a subroutine call instruction produces a call stack overflow. Additionally, a vertex program will terminate abnormally after executing 65536 instructions to prevent hangs caused by infinite loops in the program.

When a vertex program terminates, normally or abnormally, it will emit a vertex whose attributes are taken from the final values of the vertex result registers (section 2.14.1.5).

**Section 2.14.3, Vertex Program Instruction Set**

The following sections describe the set of supported vertex program instructions. Instructions available only in the VP1.1 or VP2 execution environment will be noted in the instruction description.

Each section will contain pseudocode describing the instruction. Instructions will have up to three operands, referred to as "op0", "op1", and "op2". The operands are loaded using the mechanisms specified in section 2.14.2.1. Most instructions will generate a result vector called "result". The result vector is then written to the destination register specified in the instruction using the mechanisms specified in section 2.14.2.2.

Operands and results are represented as 32-bit single-precision floating-point numbers according to the IEEE 754 floating-point specification. IEEE denorm encodings, used to represent numbers smaller than 2^-126, are not supported. All such numbers are flushed to zero. There are three special encodings referred to in this section: +INF means "positive infinity", -INF means "negative infinity", and NaN refers to "not a number".

Arithmetic operations are typically carried out in single precision according to the rules specified in the IEEE 754 specification. Any exceptions and special cases will be noted in the instruction description.

**Section 2.14.3.1, ABS: Absolute Value**

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

```
  tmp = VectorLoad(op0);
  result.x = abs(tmp.x);
  result.y = abs(tmp.y);
  result.z = abs(tmp.z);
  result.w = abs(tmp.w);
```

The following special-case rules apply to absolute value operation:

1. abs(NaN) = NaN.
2. abs(-INF) = abs(+INF) = +INF.
3. abs(-0.0) = abs(+0.0) = +0.0.

The ABS instruction is available only in the VP1.1 and VP2 execution environments.

In the VP1.0 execution environment, the same functionality can be achieved with "MAX result, src, -src".

In the VP2 execution environment, the ABS instruction is effectively obsolete, since instructions can take the absolute value of each operand at no cost.

**Section 2.14.3.2, ADD: Add**

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x + tmp1.x;
  result.y = tmp0.y + tmp1.y;
  result.z = tmp0.z + tmp1.z;
  result.w = tmp0.w + tmp1.w;
```

The following special-case rules apply to addition:

1. "A+B" is always equivalent to "B+A".
2. NaN + <x> = NaN, for all <x>.
3. +INF + <x> = +INF, for all <x> except NaN and -INF.
4. -INF + <x> = -INF, for all <x> except NaN and +INF.
5. +INF + -INF = NaN.
6. -0.0 + <x> = <x>, for all <x>.
7. +0.0 + <x> = <x>, for all <x> except -0.0.

**Section 2.14.3.3, ARA: Address Register Add**

The ARA instruction adds two pairs of components of a vector address register operand to produce an integer result vector. The "x" and "z" components of the result vector contain the sum of the "x" and "z" components of the operand; the "y" and "w" components of the result vector contain the sum of the "y" and "w" components of the operand. Each component of the result vector is clamped to [-512, +511], the range of representable address register components.

```
  itmp = AddrVectorLoad(op0);
  iresult.x = itmp.x + itmp.z;
  iresult.y = itmp.y + itmp.w;
  iresult.z = itmp.x + itmp.z;
  iresult.w = itmp.y + itmp.w;
  if (iresult.x < -512) iresult.x = -512;
  if (iresult.x > 511)  iresult.x = 511;
  if (iresult.y < -512) iresult.y = -512;
  if (iresult.y > 511)  iresult.y = 511;
  if (iresult.z < -512) iresult.z = -512;
  if (iresult.z > 511)  iresult.z = 511;
  if (iresult.w < -512) iresult.w = -512;
  if (iresult.w > 511)  iresult.w = 511;
```

Component swizzling is not supported when the operand is loaded.

The ARA instruction is available only in the VP2 execution environment.

**Section 2.14.3.4, ARL: Address Register Load**

In the VP1 execution environment, the ARL instruction loads a single scalar operand and performs a floor operation to generate an integer scalar to be written to the address register.

```
  tmp = ScalarLoad(op0);
  iresult.x = floor(tmp);
```

In the VP2 execution environment, the ARL instruction loads a single vector operand and performs a component-wise floor operation to generate an integer result vector.  Each component of the result vector is clamped to [-512, +511], the range of representable address register components. The ARL instruction applies all masking operations to address register writes as are described in section 2.14.2.2.

```
  tmp = VectorLoad(op0);
  iresult.x = floor(tmp.x);
  iresult.y = floor(tmp.y);
  iresult.z = floor(tmp.z);
  iresult.w = floor(tmp.w);
  if (iresult.x < -512) iresult.x = -512;
  if (iresult.x > 511)  iresult.x = 511;
  if (iresult.y < -512) iresult.y = -512;
  if (iresult.y > 511)  iresult.y = 511;
  if (iresult.z < -512) iresult.z = -512;
  if (iresult.z > 511)  iresult.z = 511;
  if (iresult.w < -512) iresult.w = -512;
  if (iresult.w > 511)  iresult.w = 511;
```

The following special-case rules apply to floor computation:

1. floor(NaN) = NaN.
2. floor(<x>) = <x>, for -0.0, +0.0, -INF, and +INF.  In all cases, the sign of the result is equal to the sign of the operand.

**Section 2.14.3.5, ARR: Address Register Load (with round)**

The ARR instruction loads a single vector operand and performs a component-wise round operation to generate an integer result vector. Each component of the result vector is clamped to [-512, +511], the range of representable address register components. The ARR instruction applies all masking operations to address register writes as described in section 2.14.2.2.

```
  tmp = VectorLoad(op0);
  iresult.x = round(tmp.x);
  iresult.y = round(tmp.y);
  iresult.z = round(tmp.z);
  iresult.w = round(tmp.w);
  if (iresult.x < -512) iresult.x = -512;
  if (iresult.x > 511)  iresult.x = 511;
  if (iresult.y < -512) iresult.y = -512;
  if (iresult.y > 511)  iresult.y = 511;
  if (iresult.z < -512) iresult.z = -512;
  if (iresult.z > 511)  iresult.z = 511;
  if (iresult.w < -512) iresult.w = -512;
  if (iresult.w > 511)  iresult.w = 511;
```

The rounding function, round(x), returns the nearest integer to <x>. If the fractional portion of <x> is 0.5, round(x) selects the nearest even integer.

The ARR instruction is available only in the VP2 execution environment.

**Section 2.14.3.6, BRA: Branch**

The BRA instruction conditionally transfers control to the instruction following the label specified in the instruction. 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 <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction matching the <vp2-branchLabel> grammar rule.

The BRA instruction is available only in the VP2 execution environment.

**Section 2.14.3.7, CAL: Subroutine Call**

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. It also pushes a reference to the instruction immediately following the CAL instruction onto the call stack, where execution will continue after executing the matching RET instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth >= 4) {
      // terminate vertex program
    } else {
      callStack[callStackDepth] = nextInstruction;
      callStackDepth++;
    }
    // continue execution at instruction following <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction matching the <vp2-branchLabel> grammar rule, <callStackDepth> is the current depth of the call stack, <callStack> is an array holding the call stack, and <nextInstruction> is a reference to the instruction immediately following the present one in the program string.

The CAL instruction is available only in the VP2 execution environment.

**Section 2.14.3.8, COS: Cosine**

The COS instruction approximates the cosine of the angle specified by the scalar operand and replicates the approximation to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

The approximation function ApproxCosine is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxCosine(x) - cos(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxCosine(NaN) = NaN.
2. ApproxCosine(+/-INF) = NaN.
3. ApproxCosine(+/-0.0) = +1.0.

The COS instruction is available only in the VP2 execution environment.

**Section 2.14.3.9, DP3: 3-component Dot Product**

The DP3 instruction computes a three component dot product of the two operands (using the x, y, and z components) and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z);
```

**Section 2.14.3.10, DP4: 4-component Dot Product**

The DP4 instruction computes a four component dot product of the two operands and replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + (tmp0.w * tmp1.w);
```

**Section 2.14.3.11, DPH: Homogeneous Dot Product**

The DPH instruction computes a four-component dot product of the two operands, except that the W component of the first operand is assumed to be 1.0. The instruction replicates the dot product to all four components of the result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1):
  result.x = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
  result.y = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
  result.z = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
  result.w = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) +
             (tmp0.z * tmp1.z) + tmp1.w;
```

The DPH instruction is available only in the VP1.1 and VP2 execution environments.

**Section 2.14.3.12, DST: Distance Vector**

The DST instruction computes a distance vector from two specially-formatted operands. The first operand should be of the form [NA, d^2, d^2, NA] and the second operand should be of the form [NA, 1/d, NA, 1/d], where NA values are not relevant to the calculation and d is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, d, d^2, 1/d].

The exact behavior is specified in the following pseudo-code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = 1.0;
result.y = tmp0.y * tmp1.y;
result.z = tmp0.z;
result.w = tmp1.w;
```

Given an arbitrary vector, d^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/d can be obtained from d^2 using the RSQ instruction.

This distance vector is useful for per-vertex light attenuation calculations: a DP3 operation using the distance vector and an attenuation constants vector as operands will yield the attenuation factor.

**Section 2.14.3.13, EX2: Exponential Base 2**

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates it to all four components of the result vector.

```
tmp = ScalarLoad(op0);
result.x = Approx2ToX(tmp);
result.y = Approx2ToX(tmp);
result.z = Approx2ToX(tmp);
result.w = Approx2ToX(tmp);
```

The approximation function is accurate to at least 22 bits:

```
| Approx2ToX(x) - 2^x | < 1.0 / 2^22, if 0.0 <= x < 1.0,
```

and, in general,

```
| Approx2ToX(x) - 2^x | < (1.0 / 2^22) * (2^floor(x)).
```

The following special-case rules apply to exponential approximation:

```
1. Approx2ToX(NaN) = NaN.
2. Approx2ToX(-INF) = +0.0.
3. Approx2ToX(+INF) = +INF.
4. Approx2ToX(+/-0.0) = +1.0.
```

The EX2 instruction is available only in the VP2 execution environment.

**Section 2.14.3.14, EXP: Exponential Base 2 (approximate)**

The EXP instruction computes a rough approximation of 2 raised to the power of the scalar operand. The approximation is returned in the "z" component of the result vector. A vertex program can also use the "x" and "y" components of the result vector to generate a more accurate approximation by evaluating

```
    result.x * f(result.y),
```

where f(x) is a user-defined function that approximates 2^x over the domain [0.0, 1.0). The "w" component of the result vector is always 1.0.

The exact behavior is specified in the following pseudo-code:

```
  tmp = ScalarLoad(op0);
  result.x = 2^floor(tmp);
  result.y = tmp - floor(tmp);
  result.z = RoughApprox2ToX(tmp);
  result.w = 1.0;
```

The approximation function is accurate to at least 11 bits:

```
  | RoughApprox2ToX(x) - 2^x | < 1.0 / 2^11, if 0.0 <= x < 1.0,
```

and, in general,

```
  | RoughApprox2ToX(x) - 2^x | < (1.0 / 2^11) * (2^floor(x)).
```

The following special cases apply to the EXP instruction:

1. RoughApprox2ToX(NaN) = NaN.
2. RoughApprox2ToX(-INF) = +0.0.
3. RoughApprox2ToX(+INF) = +INF.
4. RoughApprox2ToX(+/-0.0) = +1.0.

The EXP instruction is present for compatibility with the original NV_vertex_program instruction set; it is recommended that applications using NV_vertex_program2 use the EX2 instruction instead.

**Section 2.14.3.15, FLR: Floor**

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of -3.6 is -4.0.

```
  tmp = VectorLoad(op0);
  result.x = floor(tmp.x);
  result.y = floor(tmp.y);
  result.z = floor(tmp.z);
  result.w = floor(tmp.w);
```

The following special-case rules apply to floor computation:

1. floor(NaN) = NaN.
2. floor(<x>) = <x>, for -0.0, +0.0, -INF, and +INF. In all cases, the sign of the result is equal to the sign of the operand.

The FLR instruction is available only in the VP2 execution environment.

**Section 2.14.3.16, FRC: Fraction**

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see FLR), and is always in the range [0.00, 1.00).

For negative values, the fractional portion is NOT the number written to the right of the decimal point -- the fractional portion of -1.7 is not 0.7 -- it is 0.3. 0.3 is produced by subtracting the floor of -1.7 (-2.0) from -1.7.

```
  tmp = VectorLoad(op0);
  result.x = tmp.x - floor(tmp.x);
  result.y = tmp.y - floor(tmp.y);
  result.z = tmp.z - floor(tmp.z);
  result.w = tmp.w - floor(tmp.w);
```

The following special-case rules, which can be derived from the rules for FLR and ADD apply to fraction computation:

1. fraction(NaN) = NaN.
2. fraction(+/-INF) = NaN.
3. fraction(+/-0.0) = +0.0.

The FRC instruction is available only in the VP2 execution environment.

**Section 2.14.3.17, LG2: Logarithm Base 2**

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxLog2(tmp);
  result.y = ApproxLog2(tmp);
  result.z = ApproxLog2(tmp);
  result.w = ApproxLog2(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxLog2(x) - log_2(x) | < 1.0 / 2^22.
```

The following special-case rules apply to logarithm approximation:

```
  1. ApproxLog2(NaN) = NaN.
  2. ApproxLog2(+INF) = +INF.
  3. ApproxLog2(+/-0.0) = -INF.
  4. ApproxLog2(x) = NaN, -INF < x < -0.0.
  5. ApproxLog2(-INF) = NaN.
```

The LG2 instruction is available only in the VP2 execution environment.

**Section 2.14.3.18, LIT: Compute Light Coefficients**

The LIT instruction accelerates per-vertex lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The "x" component of the operand is assumed to hold a diffuse dot product (n dot VP_pli, as in the vertex lighting equations in Section 2.13.1). The "y" component of the operand is assumed to hold a specular dot product (n dot h_i). The "w" component of the operand is assumed to hold the specular exponent of the material (s_rm), and is clamped to the range (-128, +128) exclusive.

The "x" component of the result vector receives the value that should be multiplied by the ambient light/material product (always 1.0). The "y" component of the result vector receives the value that should be multiplied by the diffuse light/material product (n dot VP_pli). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product (f_i * (n dot h_i) ^ s_rm). The "w" component of the result is the constant 1.0.

Negative diffuse and specular dot products are clamped to 0.0, as is done in the standard per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

```
  tmp = VectorLoad(op0);
  if (t.x < 0) t.x = 0;
  if (t.y < 0) t.y = 0;
  if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
  else if (t.w > 128-epsilon) t.w = 128-epsilon;
  result.x = 1.0;
  result.y = t.x;
  result.z = (t.x > 0) ? RoughApproxPower(t.y, t.w) : 0.0;
  result.w = 1.0;
```

The exponentiation approximation function is defined in terms of the base 2 exponentiation and logarithm approximation operations in the EXP and LOG instructions, including errors and the processing of any special cases. In particular,

```
  RoughApproxPower(a,b) = RoughApproxExp2(b * RoughApproxLog2(a)).
```

The following special-case rules, which can be derived from the rules in the LOG, MUL, and EXP instructions, apply to exponentiation:

```
  1. RoughApproxPower(NaN, <x>) = NaN,
  2. RoughApproxPower(<x>, <y>) = NaN, if x <= -0.0,
  3. RoughApproxPower(+/-0.0, <x>) = +0.0, if x > +0.0, or
                                     +INF, if x < -0.0,
  4. RoughApproxPower(+1.0, <x>) = +1.0, if x is not NaN,
  5. RoughApproxPower(+INF, <x>) = +INF, if x > +0.0, or
                                   +0.0, if x < -0.0,
  6. RoughApproxPower(<x>, +/-0.0) = +1.0, if x >= -0.0
  7. RoughApproxPower(<x>, +INF) = +0.0, if -0.0 <= x < +1.0,
                                   +INF, if x > +1.0,
  8. RoughApproxPower(<x>, +INF) = +INF, if -0.0 <= x < +1.0,
                                   +0.0, if x > +1.0,
  9. RoughApproxPower(<x>, +1.0) = <x>, if x >= +0.0, and
  10. RoughApproxPower(<x>, NaN) = NaN.
```

**Section 2.14.3.19, LOG: Logarithm Base 2 (Approximate)**

The LOG instruction computes a rough approximation of the base 2 logarithm of the absolute value of the scalar operand. The approximation is returned in the "z" component of the result vector. A vertex program can also use the "x" and "y" components of the result vector to generate a more accurate approximation by evaluating

```
    result.x + f(result.y),
```

where f(x) is a user-defined function that approximates 2^x over the domain [1.0, 2.0). The "w" component of the result vector is always 1.0.

The exact behavior is specified in the following pseudo-code:

```
  tmp = fabs(ScalarLoad(op0));
  result.x = floor(log2(tmp));
  result.y = tmp / (2^floor(log2(tmp)));
  result.z = RoughApproxLog2(tmp);
  result.w = 1.0;
```

The approximation function is accurate to at least 11 bits:

```
  | RoughApproxLog2(x) - log_2(x) | < 1.0 / 2^11.
```

The following special-case rules apply to the LOG instruction:

1. RoughApproxLog2(NaN) = NaN.
2. RoughApproxLog2(+INF) = +INF.
3. RoughApproxLog2(+0.0) = -INF.

The LOG instruction is present for compatibility with the original NV_vertex_program instruction set; it is recommended that applications using NV_vertex_program2 use the LG2 instruction instead.

**Section 2.14.3.20, MAD: Multiply And Add**

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  tmp2 = VectorLoad(op2);
  result.x = tmp0.x * tmp1.x + tmp2.x;
  result.y = tmp0.y * tmp1.y + tmp2.y;
  result.z = tmp0.z * tmp1.z + tmp2.z;
  result.w = tmp0.w * tmp1.w + tmp2.w;
```

All special case rules applicable to the ADD and MUL instructions apply to the individual components of the MAD operation as well.

**Section 2.14.3.21, MAX: Maximum**

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = max(tmp0.x, tmp1.x);
result.y = max(tmp0.y, tmp1.y);
result.z = max(tmp0.z, tmp1.z);
result.w = max(tmp0.w, tmp1.w);
```

The following special cases apply to the maximum operation:

1. max(A,B) is always equivalent to max(B,A).
2. max(NaN, <x>) == NaN, for all <x>.

**Section 2.14.3.22, MIN: Minimum**

The MIN instruction computes component-wise minimums of the values in the two operands to yield a result vector.

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = min(tmp0.x, tmp1.x);
result.y = min(tmp0.y, tmp1.y);
result.z = min(tmp0.z, tmp1.z);
result.w = min(tmp0.w, tmp1.w);
```

The following special cases apply to the minimum operation:

1. min(A,B) is always equivalent to min(B,A).
2. min(NaN, <x>) == NaN, for all <x>.

**Section 2.14.3.23, MOV: Move**

The MOV instruction copies the value of the operand to yield a result vector.

```
result = VectorLoad(op0);
```

**Section 2.14.3.24, MUL: Multiply**

The MUL instruction performs a component-wise multiply of the two operands to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x * tmp1.x;
  result.y = tmp0.y * tmp1.y;
  result.z = tmp0.z * tmp1.z;
  result.w = tmp0.w * tmp1.w;
```

The following special-case rules apply to multiplication:

1. "A*B" is always equivalent to "B*A".
2. NaN * <x> = NaN, for all <x>.
3. +/-0.0 * +/-INF = NaN.
4. +/-0.0 * <x> = +/-0.0, for all <x> except -INF, +INF, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
5. +/-INF * <x> = +/-INF, for all <x> except -0.0, +0.0, and NaN. The sign of the result is positive if the signs of the two operands match and negative otherwise.
6. +1.0 * <x> = <x>, for all <x>.

**Section 2.14.3.25, RCC: Reciprocal (Clamped)**

The RCC instruction approximates the reciprocal of the scalar operand, clamps the result to one of two ranges, and replicates the clamped result to all four components of the result vector.

If the approximate reciprocal is greater than 0.0, the result is clamped to the range [2^-64, 2^+64]. If the approximate reciprocal is not greater than zero, the result is clamped to the range [-2^+64, -2^-64].

```
  tmp = ScalarLoad(op0);
  result.x = ClampApproxReciprocal(tmp);
  result.y = ClampApproxReciprocal(tmp);
  result.z = ClampApproxReciprocal(tmp);
  result.w = ClampApproxReciprocal(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ClampApproxReciprocal(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 2.0.
```

The following special-case rules apply to reciprocation:

```
  1. ClampApproxReciprocal(NaN) = NaN.
  2. ClampApproxReciprocal(+INF) = +2^-64.
  3. ClampApproxReciprocal(-INF) = -2^-64.
  4. ClampApproxReciprocal(+0.0) = +2^64.
  5. ClampApproxReciprocal(-0.0) = -2^64.
  6. ClampApproxReciprocal(x) = +2^-64, if -2^64 < x < +INF.
  7. ClampApproxReciprocal(x) = -2^-64, if -INF < x < -2^-64.
  8. ClampApproxReciprocal(x) = +2^64, if +0.0 < x < +2^-64.
  9. ClampApproxReciprocal(x) = -2^64, if -2^-64 < x < -0.0.
```

The RCC instruction is available only in the VP1.1 and VP2 execution environments.

**Section 2.14.3.26, RCP: Reciprocal**

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxReciprocal(tmp);
  result.y = ApproxReciprocal(tmp);
  result.z = ApproxReciprocal(tmp);
  result.w = ApproxReciprocal(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxReciprocal(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 2.0.
```

The following special-case rules apply to reciprocation:

```
  1. ApproxReciprocal(NaN) = NaN.
  2. ApproxReciprocal(+INF) = +0.0.
  3. ApproxReciprocal(-INF) = -0.0.
  4. ApproxReciprocal(+0.0) = +INF.
  5. ApproxReciprocal(-0.0) = -INF.
```

**Section 2.14.3.27, RET: Subroutine Call Return**

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction by popping an instruction reference off the top of the call stack and transferring control to the referenced instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate vertex program
    } else {
      callStackDepth--;
      instruction = callStack[callStackDepth];
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }
```

In the pseudocode, <callStackDepth> is the depth of the call stack, <callStack> is an array holding the call stack, and <instruction> is a reference to an instruction previously pushed onto the call stack.

The RET instruction is available only in the VP2 execution environment.

**Section 2.14.3.28, RSQ: Reciprocal Square Root**

The RSQ instruction approximates the reciprocal of the square root of the scalar operand and replicates it to all four components of the result vector.

```
  tmp = ScalarLoad(op0);
  result.x = ApproxRSQRT(tmp);
  result.y = ApproxRSQRT(tmp);
  result.z = ApproxRSQRT(tmp);
  result.w = ApproxRSQRT(tmp);
```

The approximation function is accurate to at least 22 bits:

```
  | ApproxRSQRT(x) - (1/x) | < 1.0 / 2^22, if 1.0 <= x < 4.0.
```

The following special-case rules apply to reciprocal square roots:

```
  1. ApproxRSQRT(NaN) = NaN.
  2. ApproxRSQRT(+INF) = +0.0.
  3. ApproxRSQRT(-INF) = NaN.
  4. ApproxRSQRT(+0.0) = +INF.
  5. ApproxRSQRT(-0.0) = -INF.
  6. ApproxRSQRT(x) = NaN, if -INF < x < -0.0.
```

**Section 2.14.3.29, SEQ: Set on Equal**

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SEQ:

```
  1. (<x> == <y>) and (<y> == <x>) always produce the same result.
  1. (NaN == <x>) is FALSE for all <x>, including NaN.
  2. (+INF == +INF) and (-INF == -INF) are TRUE.
  3. (-0.0 == +0.0) and (+0.0 == -0.0) are TRUE.
```

The SEQ instruction is available only in the VP2 execution environment.

**Section 2.14.3.30, SFL: Set on False**

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

The SFL instruction is available only in the VP2 execution environment.

**Section 2.14.3.31, SGE: Set on Greater Than or Equal**

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than or equal that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SGE:

1. (NaN >= <x>) and (<x> >= NaN) are FALSE for all <x>.
2. (+INF >= +INF) and (-INF >= -INF) are TRUE.
3. (-0.0 >= +0.0) and (+0.0 >= -0.0) are TRUE.

**Section 2.14.3.32, SGT: Set on Greater Than**

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SGT:

1. (NaN > <x>) and (<x> > NaN) are FALSE for all <x>.
2. (-0.0 > +0.0) and (+0.0 > -0.0) are FALSE.

The SGT instruction is available only in the VP2 execution environment.

**Section 2.14.3.33, SIN: Sine**

The SIN instruction approximates the sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

The approximation function is accurate to at least 22 bits with an angle in the range [0,2*PI].

```
  | ApproxSine(x) - sin(x) | < 1.0 / 2^22, if 0.0 <= x < 2.0 * PI.
```

The error in the approximation will typically increase with the absolute value of the angle when the angle falls outside the range [0,2*PI].

The following special-case rules apply to cosine approximation:

1. ApproxSine(NaN) = NaN.
2. ApproxSine(+/-INF) = NaN.
3. ApproxSine(+/-0.0) = +/-0.0. The sign of the result is equal to the sign of the single operand.

The SIN instruction is available only in the VP2 execution environment.

**Section 2.14.3.34, SLE: Set on Less Than or Equal**

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SLE:

1. (NaN <= <x>) and (<x> <= NaN) are FALSE for all <x>.
2. (+INF <= +INF) and (-INF <= -INF) are TRUE.
3. (-0.0 <= +0.0) and (+0.0 <= -0.0) are TRUE.

The SLE instruction is available only in the VP2 execution environment.

**Section 2.14.3.35, SLT: Set on Less Than**

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SLT:

```
  1. (NaN < <x>) and (<x> < NaN) are FALSE for all <x>.
  2. (-0.0 < +0.0) and (+0.0 < -0.0) are FALSE.
```

**Section 2.14.3.36, SNE: Set on Not Equal**

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
  if (tmp0.x is NaN or tmp1.x is NaN) result.x = NaN;
  if (tmp0.y is NaN or tmp1.y is NaN) result.y = NaN;
  if (tmp0.z is NaN or tmp1.z is NaN) result.z = NaN;
  if (tmp0.w is NaN or tmp1.w is NaN) result.w = NaN;
```

The following special-case rules apply to SNE:

```
  1. (<x> != <y>) and (<y> != <x>) always produce the same result.
  2. (NaN != <x>) is TRUE for all <x>, including NaN.
  3. (+INF != +INF) and (-INF != -INF) are FALSE.
  4. (-0.0 != +0.0) and (+0.0 != -0.0) are TRUE.
```

The SNE instruction is available only in the VP2 execution environment.

**Section 2.14.3.37, SSG: Set Sign**

The SSG instruction generates a result vector containing the signs of each component of the single operand. Each component of the result vector is 1.0 if the corresponding component of the operand is greater than zero, 0.0 if the corresponding component of the operand is equal to zero, and -1.0 if the corresponding component of the operand is less than zero.

```
  tmp = VectorLoad(op0);
  result.x = SetSign(tmp.x);
  result.y = SetSign(tmp.y);
  result.z = SetSign(tmp.z);
  result.w = SetSign(tmp.w);
```

The following special-case rules apply to SSG:

1. SetSign(NaN) = NaN.
2. SetSign(-0.0) = SetSign(+0.0) = 0.0.
3. SetSign(-INF) = -1.0.
4. SetSign(+INF) = +1.0.
5. SetSign(x) = -1.0, if -INF < x < -0.0.
6. SetSign(x) = +1.0, if +0.0 < x < +INF.

The SSG instruction is available only in the VP2 execution environment.

**Section 2.14.3.38, STR: Set on True**

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

The STR instruction is available only in the VP2 execution environment.

**Section 2.14.3.39,  SUB:  Subtract**

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = tmp0.x - tmp1.x;
  result.y = tmp0.y - tmp1.y;
  result.z = tmp0.z - tmp1.z;
  result.w = tmp0.w - tmp1.w;
```

The SUB instruction is completely equivalent to an identical ADD instruction in which the negate operator on the second operand is reversed:

1. "SUB R0, R1, R2" is equivalent to "ADD R0, R1, -R2".
2. "SUB R0, R1, -R2" is equivalent to "ADD R0, R1, R2".
3. "SUB R0, R1, |R2|" is equivalent to "ADD R0, R1, -|R2|".
4. "SUB R0, R1, -|R2|" is equivalent to "ADD R0, R1, |R2|".

The SUB instruction is available only in the VP1.1 and VP2 execution environments.

**2.14.4  Vertex Arrays for Vertex Attributes**

Data for vertex attributes in vertex program mode may be specified using vertex array commands.  The client may specify and enable any of sixteen vertex attribute arrays.

The vertex attribute arrays are ignored when vertex program mode is disabled.  When vertex program mode is enabled, vertex attribute arrays are used.

The command

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

describes the locations and organizations of the sixteen vertex attribute arrays.  index specifies the particular vertex attribute to be described.  size indicates the number of values per vertex that are stored in the array; size must be one of 1, 2, 3, or 4. type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, double, and ubyte respectively.  The INVALID_OPERATION error is generated if type is UNSIGNED_BYTE and size is not 4.  The INVALID_VALUE error is generated if index is greater than 15.  The INVALID_VALUE error is generated if stride is negative.

The one, two, three, or four values in an array that correspond to a single vertex attribute comprise an array element.  The values within each array element at stored sequentially in memory.  If the stride is specified as zero, then array elements are stored sequentially as well.  Otherwise points to the ith and (i+1)st elements of an array differ by stride basic machine units (typically unsigned bytes), the pointer to the (i+1)st element being greater. pointer specifies the location in memory of the first value of the first element of the array being specified.

Vertex attribute arrays are enabled with the EnableClientState command and disabled with the DisableClientState command. The value of the argument to either command is VERTEX_ATTRIB_ARRAYi_NV where i is an integer between 0 and 15; specifying a value of i enables or disables the vertex attribute array with index i. The constants obey VERTEX_ATTRIB_ARRAYi_NV = VERTEX_ATTRIB_ARRAY0_NV + i.

When vertex program mode is enabled, the ArrayElement command operates as described in this section in contrast to the behavior described in section 2.8. Likewise, any vertex array transfer commands that are defined in terms of ArrayElement (DrawArrays, DrawElements, and DrawRangeElements) assume the operation of ArrayElement described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the ArrayElement command transfers the ith element of particular enabled vertex arrays as described below. For each enabled vertex attribute array, it is as though the corresponding command from section 2.14.1.1 were called with a pointer to element i. For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,d,ub], corresponding to the array types short, int, float, double, and ubyte respectively.

However, if a given vertex attribute array is disabled, but its corresponding aliased conventional per-vertex parameter's vertex array (as described in section 2.14.1.6) is enabled, then it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. In this case, the corresponding command is determined as described in section 2.8's description of ArrayElement.

If the vertex attribute array 0 is enabled, it is as though VertexAttrib[size][type]v(0, ...) is executed last, after the executions of other corresponding commands. If the vertex attribute array 0 is disabled but the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of other corresponding commands.

### 2.14.5 Vertex State Programs

Vertex state programs share the same instruction set as and a similar execution model to vertex programs. While vertex programs are executed implicitly when a vertex transformation is provoked, vertex state programs are executed explicitly, independently of any vertices. Vertex state programs can write program parameter registers, but may not write vertex result registers. Vertex state programs have not been extended beyond the the VP1.0 execution environment, and are offered solely for compatibility with that execution environment.

The purpose of a vertex state program is to update program parameter registers by means of an application-defined program. Typically, an application will load a set of program parameters and then execute a vertex state program that reads and updates the program parameter registers. For example, a vertex state program might normalize a set of unnormalized vectors previously loaded as program parameters. The expectation is that subsequently executed vertex programs would use the normalized program parameters.

Vertex state programs are loaded with the same LoadProgramNV command (see section 2.14.1.8) used to load vertex programs except that the target must be VERTEX_STATE_PROGRAM_NV when loading a vertex state program.

Vertex state programs must conform to a more limited grammar than the grammar for vertex programs. The vertex state program grammar for syntactically valid sequences is the same as the grammar defined in section 2.14.1.8 with the following modified rules:

```
<program>              ::= <vp1-program>

<vp1-program>          ::= "!!VSP1.0" <programBody> "END"

<dstReg>               ::= <absProgParamReg>
                         | <temporaryReg>

<vertexAttribReg>      ::= "v" "[" "0" "]"
```

A vertex state program fails to load if it does not write at least one program parameter register.

A vertex state program fails to load if it contains more than 128 instructions.

A vertex state program fails to load if any instruction sources more than one unique program parameter register.

A vertex state program fails to load if any instruction sources more than one unique vertex attribute register (this is necessarily true because only vertex attribute 0 is available in vertex state programs).

The error INVALID_OPERATION is generated if a vertex state program fails to load because it is not syntactically correct or for one of the other reasons listed above.

A successfully loaded vertex state program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

Executing vertex state programs is legal only outside a Begin/End pair. A vertex state program may not read any vertex attribute register other than register zero. A vertex state program may not write any vertex result register.

The command

```
  ExecuteProgramNV(enum target, uint id, const float *params);
```

executes the vertex state program named by id. The target must be VERTEX_STATE_PROGRAM_NV and the id must be the name of program loaded with a target type of VERTEX_STATE_PROGRAM_NV. params points to an array of four floating-point values that are loaded into vertex attribute register zero (the only vertex attribute readable from a vertex state program).

The INVALID_OPERATION error is generated if the named program is nonexistent, is invalid, or the program is not a vertex state program. A vertex state program may not be valid for reasons explained in section 2.14.5.

### 2.14.6, Program Options

In the VP1.1 and VP2.0 execution environment, vertex programs may specify one or more program options that modify the execution environment, according to the <option> grammar rule. The set of options available to the program is described below.

#### Section 2.14.6.1, Position-Invariant Vertex Program Option

If <vp11-option> or <vp2-option> matches "NV_position_invariant", the vertex program is presumed to be position-invariant. By default, vertex programs are not position-invariant. Even if programs emulate the conventional OpenGL transformation model, they may still not produce the exact same transform results, due to rounding errors or different operation orders. Such programs may not work well for multi-pass rendering algorithms where the second and subsequent passes use an EQUAL depth test.

Position-invariant vertex programs do not compute a final vertex position; instead, the GL computes vertex coordinates as described in section 2.10. This computation should produce exactly the same results as the conventional OpenGL transformation model, assuming vertex weighting and vertex blending are disabled.

A vertex program that specifies the position-invariant option will fail to load if it writes to the HPOS result register.

Additionally, in the VP1.1 execution environment, position-invariant programs can not use relative addressing for program parameters. Any position-invariant VP1.1 program matches the grammar rule <relProgParamReg>, will fail to load. No such restriction exists for VP2.0 programs.

For position-invariant programs, the limit on the number of instructions allowed in a program is reduced by four: position-invariant VP1.1 and VP2.0 programs may have no more than 124 or 252 instructions, respectively.

### 2.14.7 Tracking Matrices

As a convenience to applications, standard GL matrix state can be tracked into program parameter vectors. This permits vertex programs to access matrices specified through GL matrix commands.

In addition to GL's conventional matrices, several additional matrices are available for tracking. These matrices have names of the form MATRIXi_NV where i is between zero and n-1 where n is the value of the MAX_TRACK_MATRICES_NV implementation dependent constant. The MATRIXi_NV constants obey MATRIXi_NV = MATRIX0_NV + i. The value of MAX_TRACK_MATRICES_NV must be at least eight. The maximum stack depth for tracking matrices is defined by the MAX_TRACK_MATRIX_STACK_DEPTH_NV and must be at least 1.

The command

```
  TrackMatrixNV(enum target, uint address, enum matrix, enum transform);
```

tracks a given transformed version of a particular matrix into a contiguous sequence of four vertex program parameter registers beginning at address. target must be VERTEX_PROGRAM_NV (though tracked matrices apply to vertex state programs as well because both vertex state programs and vertex programs shared the same program parameter registers). matrix must be one of NONE, MODELVIEW, PROJECTION, TEXTURE, TEXTUREi_ARB (where i is between 0 and n-1 where n is the number of texture units supported), COLOR (if the ARB_imaging subset is supported), MODELVIEW_PROJECTION_NV, or MATRIXi_NV. transform must be one of IDENTITY_NV, INVERSE_NV, TRANSPOSE_NV, or INVERSE_TRANSPOSE_NV. The INVALID_VALUE error is generated if address is not a multiple of four.

The MODELVIEW_PROJECTION_NV matrix represents the concatenation of the current modelview and projection matrices. If M is the current modelview matrix and P is the current projection matrix, then the MODELVIEW_PROJECTION_NV matrix is C and computed as

```
    C = P M
```

Matrix tracking for the specified program parameter register and the next consecutive three registers is disabled when NONE is supplied for matrix. When tracking is disabled the previously tracked program parameter registers retain the state of their last tracked values. Otherwise, the specified transformed version of matrix is tracked into the specified program parameter register and the next three registers. Whenever the matrix changes, the transformed version of the matrix is updated in the specified range of program parameter registers. If TEXTURE is specified for matrix, the texture matrix for the current active texture unit is tracked. If TEXTUREi_ARB is specified for matrix, the <i>th texture matrix is tracked.

Matrices are tracked row-wise meaning that the top row of the transformed matrix is loaded into the program parameter address, the second from the top row of the transformed matrix is loaded into the program parameter address+1, the third from the top row of the transformed matrix is loaded into the program parameter address+2, and the bottom row of the transformed matrix is loaded into the program parameter address+3. The transformed matrix may be identical to the specified matrix, the inverse of the specified matrix, the transpose of the specified matrix, or the inverse transpose of the specified matrix, depending on the value of transform.

When matrix tracking is enabled for a particular program parameter register sequence, updates to the program parameter using ProgramParameterNV commands, a vertex program, or a vertex state program are not possible. The INVALID_OPERATION error is generated if a ProgramParameterNV command is used to update a program parameter register currently tracking a matrix.

The INVALID_OPERATION error is generated by ExecuteProgramNV when the vertex state program requested for execution writes to a program parameter register that is currently tracking a matrix because the program is considered invalid.

### 2.14.8 Required Vertex Program State

The state required for vertex programs consists of:

a bit indicating whether or not program mode is enabled;

a bit indicating whether or not two-sided color mode is enabled;

a bit indicating whether or not program-specified point size mode is enabled;

256 4-component floating-point program parameter registers;

16 4-component vertex attribute registers (though this state is aliased with the current normal, primary color, secondary color, fog coordinate, weights, and texture coordinate sets);

24 sets of matrix tracking state for each set of four sequential program parameter registers, consisting of a n-valued integer indicated the tracked matrix or GL_NONE (where n is 5 + the number of texture units supported + the number of tracking matrices supported) and a four-valued integer indicating the transformation of the tracked matrix;

an unsigned integer naming the currently bound vertex program

and the state must be maintained to indicate which integers are currently in use as program names.

Each existent program object consists of a target, a boolean indicating whether the program is resident, an array of type ubyte containing the program string, and the length of the program string array.  Initially, no program objects exist.

Program mode, two-sided color mode, and program-specified point size mode are all initially disabled.

The initial state of all 256 program parameter registers is (0,0,0,0).

The initial state of the 16 vertex attribute registers is (0,0,0,1) except in cases where a vertex attribute register aliases to a conventional GL transform mode vertex parameter in which case the initial state is the initial state of the respective aliased conventional vertex parameter.

The initial state of the 24 sets of matrix tracking state is NONE for the tracked matrix and IDENTITY_NV for the transformation of the tracked matrix.

The initial currently bound program is zero.

The client state required to implement the 16 vertex attribute arrays consists of 16 boolean values, 16 memory pointers, 16 integer stride values, 16 symbolic constants representing array types, and 16 integers representing values per element. Initially, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four."

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

None.

**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.

**Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)**

None.

**Additions to the AGL/GLX/WGL Specifications**

None.

**GLX Protocol**

All relevant protocol is defined in the NV_vertex_program extension.

**Errors**

This list includes the errors specified in the NV_vertex_program extension, modified as appropriate.

The error INVALID_VALUE is generated if VertexAttribNV is called where index is greater than 15.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 255 (was 95 in NV_vertex_program).

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where index is greater than 15.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where size is not one of 1, 2, 3, or 4.

```
    The error INVALID_VALUE is generated if VertexAttribPointerNV is called
    where stride is negative.

    The error INVALID_OPERATION is generated if VertexAttribPointerNV is
    called where type is UNSIGNED_BYTE and size is not 4.

    The error INVALID_VALUE is generated if LoadProgramNV is used to load a
    program with an id of zero.

    The error INVALID_OPERATION is generated if LoadProgramNV is used to load
    an id that is currently loaded with a program of a different program
    target.

    The error INVALID_OPERATION is generated if the program passed to
    LoadProgramNV fails to load because it is not syntactically correct based
    on the specified target.  The value of PROGRAM_ERROR_POSITION_NV is still
    updated when this error is generated.

    The error INVALID_OPERATION is generated if LoadProgramNV has a target of
    VERTEX_PROGRAM_NV and the specified program fails to load because it does
    not write the HPOS register at least once.  The value of
    PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

    The error INVALID_OPERATION is generated if LoadProgramNV has a target of
    VERTEX_STATE_PROGRAM_NV and the specified program fails to load because it
    does not write at least one program parameter register.  The value of
    PROGRAM_ERROR_POSITION_NV is still updated when this error is generated.

    The error INVALID_OPERATION is generated if the vertex program or vertex
    state program passed to LoadProgramNV fails to load because it contains
    more than 128 instructions (VP1 programs) or 256 instructions (VP2
    programs).  The value of PROGRAM_ERROR_POSITION_NV is still updated when
    this error is generated.

    The error INVALID_OPERATION is generated if a program is loaded with
    LoadProgramNV for id when id is currently loaded with a program of a
    different target.

    The error INVALID_OPERATION is generated if BindProgramNV attempts to bind
    to a program name that is not a vertex program (for example, if the
    program is a vertex state program).

    The error INVALID_VALUE is generated if GenProgramsNV is called where n is
    negative.

    The error INVALID_VALUE is generated if AreProgramsResidentNV is called
    and any of the queried programs are zero or do not exist.

    The error INVALID_OPERATION is generated if ExecuteProgramNV executes a
    program that does not exist.

    The error INVALID_OPERATION is generated if ExecuteProgramNV executes a
    program that is not a vertex state program.
```

The error INVALID_OPERATION is generated if Begin, RasterPos, or a command that performs an explicit Begin is called when vertex program mode is enabled and the currently bound vertex program writes program parameters that are currently being tracked.

The error INVALID_OPERATION is generated if ExecuteProgramNV is called and the vertex state program to execute writes program parameters that are currently being tracked.

The error INVALID_VALUE is generated if TrackMatrixNV has a target of VERTEX_PROGRAM_NV and attempts to track an address is not a multiple of four.

The error INVALID_VALUE is generated if GetProgramParameterNV is called to query an index greater than 255 (was 95 in NV_vertex_program).

The error INVALID_VALUE is generated if GetVertexAttribNV is called to query an <index> greater than 15, or if <index> is zero and <pname> is CURRENT_ATTRIB_NV.

The error INVALID_VALUE is generated if GetVertexAttribPointervNV is called to query an index greater than 15.

The error INVALID_OPERATION is generated if GetProgramivNV is called and the program named id does not exist.

The error INVALID_OPERATION is generated if GetProgramStringNV is called and the program named <program> does not exist.

The error INVALID_VALUE is generated if GetTrackMatrixivNV is called with an <address> that is not divisible by four or greater than or equal to 256 (was 96 in NV_vertex_program).

The error INVALID_VALUE is generated if AreProgramsResidentNV, DeleteProgramsNV, GenProgramsNV, or RequestResidentProgramsNV are called where <n> is negative.

The error INVALID_VALUE is generated if LoadProgramNV is called where <len> is negative.

The error INVALID_VALUE is generated if ProgramParameters4dvNV or ProgramParameters4fvNV are called where <count> is negative.

The error INVALID_VALUE is generated if VertexAttribs{1,2,3,4}{d,f,s}vNV is called where <count> is negative.

The error INVALID_ENUM is generated if BindProgramNV, GetProgramParameterfvNV, GetProgramParameterdvNV, GetTrackMatrixivNV, ProgramParameter4fNV, ProgramParameter4dNV, ProgramParameter4fvNV, ProgramParameter4dvNV, ProgramParameters4fvNV, ProgramParameters4dvNV, or TrackMatrixNV are called where <target> is not VERTEX_PROGRAM_NV.

The error INVALID_ENUM is generated if LoadProgramNV or ExecuteProgramNV are called where <target> is not either VERTEX_PROGRAM_NV or VERTEX_STATE_PROGRAM_NV.

**New State**

**(Modify Table X.5, New State Introduced by NV_vertex_program from the NV_vertex_program specification.)**

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| PROGRAM_PARAMETER_NV | 256xR4 | GetProgramParameterNV | (0,0,0,0) | program parameters | 2.14.1.2 | - |

**(Modify Table X.7. Vertex Program Per-vertex Execution State. "VP1" and "VP2" refer to the VP1 and VP2 execution environments, respectively.)**

| Get Value | Type | Get Command | Initial Value | Description | Sec | Attribute |
|---|---|---|---|---|---|---|
| - | 12xR4 | - | (0,0,0,0) | VP1 temporary registers | 2.14.1.4 | - |
| - | 16xR4 | - | (0,0,0,0) | VP2 temporary registers | 2.14.1.4 | - |
| - | 15xR4 | - | (0,0,0,1) | vertex result registers | 2.14.1.4 | - |
|  | Z4 | - | (0,0,0,0) | VP1 address register | 2.14.1.3 | - |
|  | 2xZ4 | - | (0,0,0,0) | VP2 address registers | 2.14.1.3 | - |

**Name**

    NV_vertex_program2_option

**Name Strings**

    GL_NV_vertex_program2_option

**Status**

    Shipping.

**Version**

```
    Last Modified:      05/16/2004
    NVIDIA Revision:    2
```

**Number**

    Unassigned

**Dependencies**

    ARB_vertex_program is required.

**Overview**

This 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_program2;
```

to the beginning of their vertex programs.

The functionality provided by this extension, which is roughly equivalent to that provided by the NV_vertex_program2 extension, includes:

* general purpose dynamic branching,

* subroutine calls,

* data-dependent conditional write masks,

* programmable user clip distances,

* address registers with four components (instead of just one),

* absolute value operator on scalar and swizzled operand loads,

* rudimentary address register math,

* SIN and COS trigonometry instructions, and

* fully orthogonal "set on" instructions, including a "set sign"

instruction.

**Issues**

*Why is this a separate extension, rather than just an additional feature of NV_vertex_program2?*

RESOLVED: The NV_vertex_program2 specification was completed (with a published implementation) prior to the completion of ARB_vertex_program. Future NVIDIA vertex program extensions should contain extensions to the ARB_vertex_program execution environment as a standard feature.

*NV_vertex_program1_1 contains one feature not found in ARB_vertex_program: the "RCC" (reciprocal clamped) instruction. Should a "NV_vertex_program1_1" program option be provided to expose this small amount of missing functionality?*

RESOLVED: No. By itself, that functionality is not all that interesting.

*Should this extension provide a mechanism to specify an "ARB" version of NV_vertex_program state programs (!!VSP1.0)?*

RESOLVED: No.

*Should a similar option be provided to expose ARB_vertex_program features not found in NV_vertex_program (e.g., local parameters, state bindings, certain "macro" instructions) under the NV_vertex_program interface?*

RESOLVED: No. Why not just write an ARB program in that case?

*The ARB_vertex_program spec has a minor grammar bug that requires that inline scalar constants used as scalar operands include a component selector. In other words, you have to say "11.0.x" to use the constant "11.0". What should we do here?*

RESOLVED: The NV_vertex_program2_option grammar will correct this problem, which should be fixed in future revisions to the ARB language.

**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 |

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

**Modify Section 2.11, Clipping (p. 42)**

(insert before the second paragraph, p. 43) In vertex program mode, conventional user clipping is performed if the vertex program is position-invariant (section 2.14.4.5.1). When the vertex program is not position-invariant, it can write a single floating-point clip distance for each supported clip plane. The half-space corresponding to clip plane <n> is given by the set of points that satisfy the inequality

```
  c_n(P) >=0,
```

where c_n(P) is the value of clip distance <n> at point P. For point primitives, c_n(P) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 3.4 and 3.5.

**Modify Section 2.14.2, Vertex Program Grammar and Restrictions**

(mostly add to existing grammar rules, modify a few existing grammar rules -- changes marked with "***")

```
<optionName>            ::= "NV_vertex_program2"

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

<instruction>           ::= <FlowInstruction>

<ALUInstruction>        ::= <ARAop_instruction>

<FlowInstruction>       ::= <BRAop_instruction>
                          | <FLOWCCop_instruction>

<VECTORop>              ::= "SSG"

<SCALARop>              ::= "COS"
                          | "RCC"
                          | "SIN"

<BINop>                 ::= "SEQ"
                          | "SFL"
                          | "SGT"
                          | "SLE"
                          | "SNE"
                          | "STR"

<ARLop>                 ::= "ARR"

<ARLop_src>             ::= <instOperandV>
                               (*** instead of <instOperandS>)

<ARAop_instruction>     ::= <ARAop> <instResultAddr> ","
                            <instOperandAddrVNS>

<ARAop>                 ::= "ARA"

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

```
    <BRAop>                 ::= "BRA"
                              | "CAL"

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

    <FLOWCCop>              ::= "RET"

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

    <instOperandV>          ::= <instOperandAbsV>

    <instOperandAbsV>       ::= <optSign> "|" <instOperandBaseV> "|"

    <instOperandS>          ::= <instOperandAbsS>

    <instOperandAbsS>       ::= <optSign> "|" <instOperandBaseS> "|"


    <instOperandAddrVNS>    ::= <addrUseVNS>

    <instResult>            ::= <instResultCC>

    <instResultCC>          ::= <instResultBase> <ccMask>

    <instResultAddr>        ::= <instResultAddrCC>

    <instResultAddrCC>      ::= <instResultAddrBase> <ccMask>

    <branchLabel>           ::= <identifier>

    <paramUseV>             ::= <constantScalar>
                                  (*** instead of <constantScalar>
                                       <swizzleSuffix>)

    <paramUseS>             ::= <constantScalar>
                                  (*** instead of <constantScalar>
                                       <scalarSuffix>)

    <resultVtxBasic>        ::= "clip" "[" <clipPlaneNum> "]"

    <addrUseVNS>            ::= <addrVarName>

    <addrUseW>              ::= <addrVarName> <optAddrWriteMask>
                                  (*** instead of <addrVarName>
                                       <addrWriteMask>)

    <ccMask>                ::= "(" <ccTest> ")"

    <ccTest>                ::= <ccMaskRule> <swizzleSuffix>

    <ccMaskRule>            ::= "EQ"
                              | "GE"
                              | "GT"
                              | "LE"
                              | "LT"
                              | "NE"
```

```
                             | "TR"
                             | "FL"

    <optAddrWriteMask>      ::= <optWriteMask>
                                  (*** instead of "." "x")

    <addrComponent>         ::= <xyzwComponent>
                                  (*** instead of "x")
```

(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, POW, RCC, RCP, RET,
    RSQ, SEQ, SFL, SGE, SGT, SIN, SLE, SLT, SNE, SUB, SSG, STR, SWZ,
    TEMP, XPD, program, result, state, and vertex.
```

**Add to Section 2.14.3.4, Vertex Program Results**

(add to binding table)

| Binding | Components | Description |
|---|---|---|
| result.clip[n] | (d,*,*,*) | clip plane distance |

(add a paragraph before the last one) If a result variable binding matches "result.clip[n]", updates to the "x" component of the result variable set the clip distance for clip plane <n>.

(modify last paragraph) When in vertex program mode, all attributes of a transformed vertex, except for clip distances, are undefined at each vertex program invocation. Any results, or even individual components of results, that are not written to during vertex program execution remain undefined. All clip distances are initially zero, and remain zero if not written by the vertex program.

**Modify Section 2.14.3.5, Vertex Program Address Registers**

(modify first paragraph) Vertex program address register variables are a set of four-component signed integer vectors. Address registers are used as indices when performing relative addressing in program parameter arrays (section 2.14.4.2).

(modify third paragraph) Vertex program address register variables are undefined at each vertex program invocation. Address registers can be written by the ARA, ARL, and ARL instructions (section 2.14.5), and will be read by the ARA instruction and when a program uses relative addressing in program parameter arrays.

**Add New Section 2.14.3.X, Condition Code Register (insert after Section 2.14.3.5, Vertex Program Address Registers)**

The vertex program condition code register is a single four-component vector. 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 the condition code register. When a vertex program instruction updates the 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 register is initialized to a vector of EQ values each time a vertex program executes.

**Modify Section 2.14.4, Vertex Program Execution Environment**

(modify 3rd paragraph) Vertex programs execute a sequence of instructions, with support for conditional and unconditional branches, subroutine calls, and returns. Vertex programs begin by executing the instruction following the label "main". If no label "main" is defined, execution begins at the first instruction in the program. Instructions are executed in the order specified in the program, jumping when specified in branch instructions, until the end of the program is reached.

(modify instruction table) There are forty-two vertex program instructions. Vertex program instructions may have an optional suffix of "C" to allow an update of the condition code register (section 2.14.3.X). For example, there are two instructions to perform vector addition, "ADD" and "ADDC". The instructions and their respective input and output parameters are summarized in Table X.5.

| Instruction | Inputs | Output | Description |
|---|---|---|---|
| ABS[C] | v | v | absolute value |
| ADD[C] | v,v | v | add |
| ARA[C] | a | a | address register add |
| ARL[C] | s | a | address register load |
| ARR[C] | v | a | address register load (round) |
| BRA | c | - | branch |
| CAL | c | - | subroutine call |
| COS[C] | s | ssss | cosine |
| DP3[C] | v,v | ssss | 3-component dot product |
| DP4[C] | v,v | ssss | 4-component dot product |
| DPH[C] | v,v | ssss | homogeneous dot product |
| DST[C] | v,v | v | distance vector |
| EX2[C] | s | ssss | exponential base 2 |
| EXP[C] | s | v | exponential base 2 (approximate) |
| FLR[C] | v | v | floor |
| FRC[C] | v | v | fraction |
| LG2[C] | s | ssss | logarithm base 2 |
| LIT[C] | v | v | compute light coefficients |
| LOG[C] | s | v | logarithm base 2 (approximate) |
| MAD[C] | v,v,v | v | multiply and add |
| MAX[C] | v,v | v | maximum |
| MIN[C] | v,v | v | minimum |
| MOV[C] | v | v | move |

| | | | |
|---|---|---|---|
| MUL[C] | v,v | v | multiply |
| POW[C] | s,s | ssss | exponentiate |
| RCC[C] | s | ssss | reciprocal (clamped) |
| RCP[C] | s | ssss | reciprocal |
| RET | c | - | subroutine return |
| RSQ[C] | s | ssss | reciprocal square root |
| SEQ[C] | v,v | v | set on equal |
| SFL[C] | v,v | v | set on false |
| SGE[C] | v,v | v | set on greater than or equal |
| SGT[C] | v,v | v | set on greater than |
| SIN[C] | s | ssss | sine |
| SLE[C] | v,v | v | set on less than or equal |
| SLT[C] | v,v | v | set on less than |
| SNE[C] | v,v | v | set on not equal |
| SSG[C] | v | v | set sign |
| STR[C] | v,v | v | set on true |
| SUB[C] | v,v | v | subtract |
| SWZ[C] | v | v | extended swizzle |
| XPD[C] | v,v | v | cross product |

Table X.5: Summary of vertex program instructions. "[C]" indicates that the opcode supports the condition code update modifier. "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.

**Modify Section 2.14.4.1, Vertex Program Operands**

(add prior to the discussion of negation) A component-wise absolute value operation can optionally performed on the operand if the operand is surrounded with two "|" characters. For example, "|src|" indicates that a component-wise absolute value operation should be performed on the variable named "src". In terms of the grammar, this operation is performed if the <instOperandV> or <instOperandS> grammar rules match <instOperandAbsV> or <instOperandAbsS>, respectively.

(modify operand load pseudo-code) The following pseudo-code spells out the operand generation process. In the example, "float" is a floating-point scalar type, while "floatVec" is a four-component vector. "source" refers to the register used for the operand, matching the <srcReg> rule. "abs" is TRUE if an absolute value operation should be performed on the operand (<instOperandAbsV> or <instOperandAbsS> rules) "negate" is TRUE if the <optionalSign> rule in <scalarSrcReg> or <swizzleSrcReg> matches "-" and FALSE otherwise. The ".c***", ".*c**", ".**c*", ".***c" modifiers refer to the x, y, z, and w components obtained by the swizzle operation; the ".c" modifier refers to the single component selected for a scalar load.

```
floatVec VectorLoad(floatVec source)
{
    floatVec operand;

    operand.x = source.c***;
    operand.y = source.*c**;
    operand.z = source.**c*;
    operand.w = source.***c;
```

```
        if (abs) {
           operand.x = abs(operand.x);
           operand.y = abs(operand.y);
           operand.z = abs(operand.z);
           operand.w = abs(operand.w);
        }
        if (negate) {
           operand.x = -operand.x;
           operand.y = -operand.y;
           operand.z = -operand.z;
           operand.w = -operand.w;
        }

        return operand;
    }

    float ScalarLoad(floatVec source)
    {
        float operand;

        operand = source.c;
        if (abs) {
          operand = abs(operand);
        if (negate) {
          operand = -operand;
        }

        return operand;
    }
```

**Rewrite Section 2.14.4.3, Vertex Program Destination Register Update**

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. The 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, "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;

          // 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 (updatecc) mergedCC.x = GenerateCC(result.x);
          }
          if (instrMask.y && TestCC(cc.*c**)) {
              merged.y = result.y;
              if (updatecc) mergedCC.y = GenerateCC(result.y);
          }
          if (instrMask.z && TestCC(cc.**c*)) {
              merged.z = result.z;
              if (updatecc) mergedCC.z = GenerateCC(result.z);
          }
          if (instrMask.w && TestCC(cc.***c)) {
              merged.w = result.w;
              if (updatecc) 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 Section 2.14.4.X, Vertex Program Branching (before Section 2.14.4.4, Vertex Program Result Processing)**

Vertex programs can contain one or more instruction labels, matching the grammar rule <branchLabel>. An instruction label can be referred to explicitly in branch (BRA) or subroutine call (CAL) instructions. Instruction labels can be defined or used at any point in the body of a program, and can be used in instructions before being defined in the program string.

Branching instructions can be conditional. The branch condition is specified by the <optBranchCond> 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 vertex 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)
    BRA label1 (LT.xyzw);
    MOV R0,R1;             # not executed
  label1:
    BRA label2 (LT.wyzw);
    MOV R0,R2;             # executed
  label2:
```

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

Vertex programs can 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 vertex 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.

If a vertex 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 vertex program will terminate after executing a RET instruction with an empty call stack. A vertex program will also terminate after executing the last instruction in the program, unless that instruction was a taken branch.

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

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

When a vertex program terminates, normally or abnormally, it will emit a vertex whose attributes are taken from the final values of the vertex result registers (section 2.14.1.5).

**Modify Section 2.14.4.4, Vertex Program Result Processing**

(modify 3rd paragraph) Transformed vertices are then assembled into primitives and clipped as described in section 2.11. Clip distance results are used to control user clip planes.

**Add to Section 2.14.4.5, Vertex Program Options:**

**Section 2.14.4.5.2, NV_vertex_program2 Option**

If a vertex program specifies the "NV_vertex_program2" program option, the grammar will be extended to support the features found in the NV_vertex_program2 extension not present in the ARB_vertex_program extension, including:

* the availability of the following instructions:

    - ARA (address register add, useful for looping),
    - ARR (address register load with round),
    - BRA (branch),
    - CAL (subroutine call),
    - COS (cosine),
    - RET (subroutine return),
    - SEQ (set on equal),
    - SFL (set on false),
    - SGT (set on greater than),
    - SIN (sine),
    - SLE (set on less than or equal),
    - SNE (set on not equal),
    - SSG (set sign), and
    - STR (set on true).

* up to MAX_CALL_DEPTH_NV levels of subroutine calls/returns,

* a four-component condition code register to hold the sign of result vector components (useful for comparisons),

* a condition code update opcode suffix "C", where the results of the instruction are used to update the condition code register,

* a condition code write mask operator, where the condition code register is swizzled and tested, and the test results are used to mask register writes,

* six clip distance result bindings that can be used to perform

more complicated user clipping operations than those provided with the position invariant program option,

* four-component address registers (instead of one-component registers in ARB_vertex_program), with the "ARL" instruction extended to produce a vector result,

* an absolute value operator on scalar and swizzled operands.

The added functionality is identical to that provided by NV_vertex_program2 extension specification.

**Modify Section 2.14.5.3, ARL: Address Register Load**

The ARL instruction loads a single vector operand and performs a component-wise floor operation to generate a signed integer result vector.

```
  tmp = VectorLoad(op0);
  iresult.x = floor(tmp.x);
  iresult.y = floor(tmp.y);
  iresult.z = floor(tmp.z);
  iresult.w = floor(tmp.w);
```

The floor operation returns the largest integer less than or equal to the operand. For example floor(-1.7) = -2.0, floor(+1.0) = +1.0, and floor(+3.7) = +3.0.

Note that in the unextended ARB_vertex_program specification, the ARL instruction loads a scalar operand and generates a scalar result.

**Add to Section 2.14.5, Vertex Program Instruction Set**

**Section 2.14.5.28, ARA: Address Register Add**

The ARA instruction adds two pairs of components of a vector address register operand to produce an integer result vector. The "x" and "z" components of the result vector contain the sum of the "x" and "z" components of the operand; the "y" and "w" components of the result vector contain the sum of the "y" and "w" components of the operand.

```
  itmp = AddrVectorLoad(op0);
  iresult.x = itmp.x + itmp.z;
  iresult.y = itmp.y + itmp.w;
  iresult.z = itmp.x + itmp.z;
  iresult.w = itmp.y + itmp.w;
```

Component swizzling is not supported when the operand is loaded.

**Section 2.14.5.29, ARR: Address Register Load (with round)**

The ARR instruction loads a single vector operand and performs a component-wise round operation to generate a signed integer result vector.

```
  tmp = VectorLoad(op0);
  iresult.x = floor(tmp.x);
  iresult.y = floor(tmp.y);
  iresult.z = floor(tmp.z);
  iresult.w = floor(tmp.w);
```

The round operation returns the nearest integer to the operand. For example round(-1.7) = -2.0, round(+1.0) = +1.0, and round(+3.7) = +4.0.

**Section 2.14.5.30, BRA: Branch**

The BRA instruction conditionally transfers control to the instruction following the label specified in the instruction. 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 <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction according to the <branchLabel> grammar rule.

**Section 2.14.5.31, CAL: Subroutine Call**

The CAL instruction conditionally transfers control to the instruction following the label specified in the instruction. It also pushes a reference to the instruction immediately following the CAL instruction onto the call stack, where execution will continue after executing the matching RET instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth >= MAX_PROGRAM_CALL_DEPTH_NV) {
      // terminate vertex program
    } else {
      callStack[callStackDepth] = nextInstruction;
      callStackDepth++;
    }
    // continue execution at instruction following <branchLabel>
  } else {
    // do nothing
  }
```

In the pseudocode, <branchLabel> is the label specified in the instruction matching the <branchLabel> grammar rule, <callStackDepth> is the current depth of the call stack, <callStack> is an array holding the call stack, and <nextInstruction> is a reference to the instruction immediately following the present one in the program string.

**Section 2.14.5.32, COS: Cosine**

The COS instruction approximates the cosine of the angle specified by the scalar operand and replicates the approximation to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxCosine(tmp);
  result.y = ApproxCosine(tmp);
  result.z = ApproxCosine(tmp);
  result.w = ApproxCosine(tmp);
```

**Section 2.14.5.33, RCC: Reciprocal (Clamped)**

The RCC instruction approximates the reciprocal of the scalar operand, clamps the result to one of two ranges, and replicates the clamped result to all four components of the result vector.

If the approximated reciprocal is greater than 0.0, the result is clamped to the range [2^-64, 2^+64]. If the approximate reciprocal is not greater than zero, the result is clamped to the range [-2^+64, -2^-64].

```
  tmp = ScalarLoad(op0);
  result.x = ClampApproxReciprocal(tmp);
  result.y = ClampApproxReciprocal(tmp);
  result.z = ClampApproxReciprocal(tmp);
  result.w = ClampApproxReciprocal(tmp);
```

The following rule applies to reciprocation:

  1. ApproxReciprocal(+1.0) = +1.0.

**Section 2.14.5.34, RET: Subroutine Call Return**

The RET instruction conditionally returns from a subroutine initiated by a CAL instruction by popping an instruction reference off the top of the call stack and transferring control to the referenced instruction. The following pseudocode describes the operation of the instruction:

```
  if (TestCC(cc.c***) || TestCC(cc.*c**) ||
      TestCC(cc.**c*) || TestCC(cc.***c)) {
    if (callStackDepth <= 0) {
      // terminate vertex program
    } else {
      callStackDepth--;
      instruction = callStack[callStackDepth];
    }

    // continue execution at <instruction>
  } else {
    // do nothing
  }
```

In the pseudocode, <callStackDepth> is the depth of the call stack, <callStack> is an array holding the call stack, and <instruction> is a reference to an instruction previously pushed onto the call stack.

**Section 2.14.5.35, SEQ: Set on Equal**

The SEQ instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x == tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y == tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z == tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w == tmp1.w) ? 1.0 : 0.0;
```

**Section 2.14.5.36, SFL: Set on False**

The SFL instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 0.0.

```
  result.x = 0.0;
  result.y = 0.0;
  result.z = 0.0;
  result.w = 0.0;
```

**Section 2.14.5.37, SGT: Set on Greater Than**

The SGT instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operands is greater than that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x > tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y > tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z > tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w > tmp1.w) ? 1.0 : 0.0;
```

**Section 2.14.5.38, SIN: Sine**

The SIN instruction approximates the sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [0,2*PI].

```
  tmp = ScalarLoad(op0);
  result.x = ApproxSine(tmp);
  result.y = ApproxSine(tmp);
  result.z = ApproxSine(tmp);
  result.w = ApproxSine(tmp);
```

**Section 2.14.5.39, SLE: Set on Less Than or Equal**

The SLE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is less than or equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x <= tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y <= tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z <= tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w <= tmp1.w) ? 1.0 : 0.0;
```

### Section 2.14.5.40, SNE: Set on Not Equal

The SNE instruction performs a component-wise comparison of the two operands. Each component of the result vector is 1.0 if the corresponding component of the first operand is not equal to that of the second, and 0.0 otherwise.

```
  tmp0 = VectorLoad(op0);
  tmp1 = VectorLoad(op1);
  result.x = (tmp0.x != tmp1.x) ? 1.0 : 0.0;
  result.y = (tmp0.y != tmp1.y) ? 1.0 : 0.0;
  result.z = (tmp0.z != tmp1.z) ? 1.0 : 0.0;
  result.w = (tmp0.w != tmp1.w) ? 1.0 : 0.0;
```

### Section 2.14.5.41, SSG: Set Sign

The SSG instruction generates a result vector containing the signs of each component of the single vector operand. Each component of the result vector is 1.0 if the corresponding component of the operand is greater than zero, 0.0 if the corresponding component of the operand is equal to zero, and -1.0 if the corresponding component of the operand is less than zero.

```
  tmp = VectorLoad(op0);
  result.x = SetSign(tmp.x);
  result.y = SetSign(tmp.y);
  result.z = SetSign(tmp.z);
  result.w = SetSign(tmp.w);
```

### Section 2.14.5.42, STR: Set on True

The STR instruction is a degenerate case of the other "Set on" instructions that sets all components of the result vector to 1.0.

```
  result.x = 1.0;
  result.y = 1.0;
  result.z = 1.0;
  result.w = 1.0;
```

## 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**

This specification is based on a modified version of the grammar published in the ARB_vertex_program specification. This modified grammar (see below) 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.

```
<program>               ::= <optionSequence> <statementSequence> "END"

<optionSequence>        ::= <optionSequence> <option>
                          | /* empty */

<option>                ::= "OPTION" <optionName> ";"

<optionName>            ::= "ARB_position_invariant"

<statementSequence>     ::= <statement> <statementSequence>
                          | /* empty */

<statement>             ::= <instruction> ";"
                          | <namingStatement> ";"

<instruction>           ::= <ALUInstruction>

<ALUInstruction>        ::= <VECTORop_instruction>
                          | <SCALARop_instruction>
                          | <BINSCop_instruction>
                          | <BINop_instruction>
                          | <TRIop_instruction>
                          | <SWZop_instruction>
                          | <ARLop_instruction>

<VECTORop_instruction>  ::= <VECTORop> <instResult> "," <instOperandV>

<VECTORop>              ::= "ABS"
                          | "FLR"
                          | "FRC"
                          | "LIT"
                          | "MOV"

<SCALARop_instruction>  ::= <SCALARop> <instResult> "," <instOperandS>
```

```
    <SCALARop>               ::= "EX2"
                               | "EXP"
                               | "LG2"
                               | "LOG"
                               | "RCP"
                               | "RSQ"

    <BINSCop_instruction>    ::= <BINSCop> <instResult> "," <instOperandS> ","
                                 <instOperandS>

    <BINSCop>                ::= "POW"

    <BINop_instruction>      ::= <BINop> <instResult> "," <instOperandV> ","
                                 <instOperandV>

    <BINop>                  ::= "ADD"
                               | "DP3"
                               | "DP4"
                               | "DPH"
                               | "DST"
                               | "MAX"
                               | "MIN"
                               | "MUL"
                               | "SGE"
                               | "SLT"
                               | "SUB"
                               | "XPD"

    <TRIop_instruction>      ::= <TRIop> <instResult> "," <instOperandV> ","
                                 <instOperandV> "," <instOperandV>

    <TRIop>                  ::= "MAD"

    <SWZop_instruction>      ::= <SWZop> <instResult> "," <instOperandVNS> ","
                                 <extendedSwizzle>

    <SWZop>                  ::= "SWZ"

    <ARLop_instruction>      ::= <ARLop> <instResultAddr> "," <ARLop_src>

    <ARLop>                  ::= "ARL"

    <ARLop_src>              ::= <instOperandS>

    <instOperandV>           ::= <instOperandBaseV>

    <instOperandBaseV>       ::= <optSign> <attribUseV>
                               | <optSign> <tempUseV>
                               | <optSign> <paramUseV>

    <instOperandS>           ::= <instOperandBaseS>

    <instOperandBaseS>       ::= <optSign> <attribUseS>
                               | <optSign> <tempUseS>
                               | <optSign> <paramUseS>
    <instOperandVNS>        ::= <attribUseVNS>
                              | <tempUseVNS>
                              | <paramUseVNS>

    <instResult>            ::= <instResultBase>

    <instResultBase>        ::= <tempUseW>
                              | <resultUseW>

    <instResultAddr>        ::= <instResultAddrBase>

    <instResultAddrBase>    ::= <addrUseW>

    <namingStatement>       ::= <ATTRIB_statement>
                              | <PARAM_statement>
                              | <TEMP_statement>
                              | <OUTPUT_statement>
                              | <ALIAS_statement>
                              | <ADDRESS_statement>

    <ATTRIB_statement>      ::= "ATTRIB" <establishName> "=" <attribUseD>

    <PARAM_statement>       ::= <PARAM_singleStmt>
                              | <PARAM_multipleStmt>

    <PARAM_singleStmt>      ::= "PARAM" <establishName> <paramSingleInit>

    <PARAM_multipleStmt>    ::= "PARAM" <establishName> "[" <optArraySize> "]"
                                <paramMultipleInit>

    <optArraySize>          ::= /* empty */
                              | <integer> /* [1,MAX_PROGRAM_PARAMETERS_ARB]*/

    <paramSingleInit>       ::= "=" <paramUseDB>

    <paramMultipleInit>     ::= "=" "{" <paramMultInitList> "}"

    <paramMultInitList>     ::= <paramUseDM>
                              | <paramUseDM> "," <paramMultInitList>

    <TEMP_statement>        ::= "TEMP" <varNameList>

    <OUTPUT_statement>      ::= "OUTPUT" <establishName> "=" <resultUseD>

    <ALIAS_statement>       ::= "ALIAS" <establishName> "=" <establishedName>

    <establishedName>       ::= <tempVarName>
                              | <addrVarName>
                              | <attribVarName>
                              | <paramArrayVarName>
                              | <paramSingleVarName>
                              | <resultVarName>

    <ADDRESS_statement>     ::= "ADDRESS" <varNameList>

    <varNameList>           ::= <establishName>
                              | <establishName> "," <varNameList>
    <establishName>         ::= <identifier>

    <attribUseV>            ::= <attribBasic> <swizzleSuffix>
                              | <attribVarName> <swizzleSuffix>
                              | <attribColor> <swizzleSuffix>
                              | <attribColor> "." <colorType> <swizzleSuffix>

    <attribUseS>            ::= <attribBasic> <scalarSuffix>
                              | <attribVarName> <scalarSuffix>
                              | <attribColor> <scalarSuffix>
                              | <attribColor> "." <colorType> <scalarSuffix>

    <attribUseVNS>          ::= <attribBasic>
                              | <attribVarName>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribUseD>            ::= <attribBasic>
                              | <attribColor>
                              | <attribColor> "." <colorType>

    <attribBasic>           ::= "vertex" "." <attribVtxBasic>

    <attribVtxBasic>        ::= "position"
                              | "weight" <vtxOptWeightNum>
                              | "normal"
                              | "fogcoord"
                              | "texcoord" <optTexCoordNum>
                              | "matrixindex" "[" <vtxWeightNum> "]"
                              | "attrib" "[" <vtxAttribNum> "]"

    <attribColor>           ::= "vertex" "." "color"

    <paramUseV>             ::= <paramSingleVarName> <swizzleSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <swizzleSuffix>
                              | <stateSingleItem> <swizzleSuffix>
                              | <programSingleItem> <swizzleSuffix>
                              | <constantVector> <swizzleSuffix>
                              | <constantScalar> <swizzleSuffix>

    <paramUseS>             ::= <paramSingleVarName> <scalarSuffix>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                                <scalarSuffix>
                              | <stateSingleItem> <scalarSuffix>
                              | <programSingleItem> <scalarSuffix>
                              | <constantVector> <scalarSuffix>
                              | <constantScalar> <scalarSuffix>

    <paramUseVNS>           ::= <paramSingleVarName>
                              | <paramArrayVarName> "[" <arrayMem> "]"
                              | <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <constantScalar>
    <paramUseDB>             ::= <stateSingleItem>
                              | <programSingleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <paramUseDM>             ::= <stateMultipleItem>
                              | <programMultipleItem>
                              | <constantVector>
                              | <signedConstantScalar>

    <stateMultipleItem>      ::= <stateSingleItem>
                              | "state" "." <stateMatrixRows>

    <stateSingleItem>        ::= "state" "." <stateMaterialItem>
                              | "state" "." <stateLightItem>
                              | "state" "." <stateLightModelItem>
                              | "state" "." <stateLightProdItem>
                              | "state" "." <stateFogItem>
                              | "state" "." <stateMatrixRow>
                              | "state" "." <stateTexGenItem>
                              | "state" "." <stateClipPlaneItem>
                              | "state" "." <statePointItem>

    <stateMaterialItem>      ::= "material" "." <stateMatProperty>
                              | "material" "." <faceType> "."
                                <stateMatProperty>

    <stateMatProperty>       ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "emission"
                              | "shininess"

    <stateLightItem>         ::= "light" "[" <stateLightNumber> "]" "."
                                <stateLightProperty>

    <stateLightProperty>     ::= "ambient"
                              | "diffuse"
                              | "specular"
                              | "position"
                              | "attenuation"
                              | "spot" "." <stateSpotProperty>
                              | "half"

    <stateSpotProperty>      ::= "direction"

    <stateLightModelItem>    ::= "lightmodel" <stateLModProperty>

    <stateLModProperty>      ::= "." "ambient"
                              | "." "scenecolor"
                              | "." <faceType> "." "scenecolor"

    <stateLightProdItem>     ::= "lightprod" "[" <stateLightNumber> "]" "."
                                <stateLProdProperty>
                              | "lightprod" "[" <stateLightNumber> "]" "."
                                <faceType> "." <stateLProdProperty>
    <stateLProdProperty>    ::= "ambient"
                              | "diffuse"
                              | "specular"

    <stateLightNumber>      ::= <integer> /* [0,MAX_LIGHTS-1] */

    <stateFogItem>          ::= "fog" "." <stateFogProperty>

    <stateFogProperty>      ::= "color"
                              | "params"

    <stateMatrixRows>       ::= <stateMatrixItem>
                              | <stateMatrixItem> "." <stateMatModifier>
                              | <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> ".." <stateMatrixRowNum>
                                "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> ".."
                                <stateMatrixRowNum> "]"

    <stateMatrixRow>        ::= <stateMatrixItem> "." "row" "["
                                <stateMatrixRowNum> "]"
                              | <stateMatrixItem> "." <stateMatModifier> "."
                                "row" "[" <stateMatrixRowNum> "]"

    <stateMatrixItem>       ::= "matrix" "." <stateMatrixName>

    <stateMatModifier>      ::= "inverse"
                              | "transpose"
                              | "invtrans"

    <stateMatrixName>       ::= "modelview" <stateOptModMatNum>
                              | "projection"
                              | "mvp"
                              | "texture" <optTexCoordNum>
                              | "palette" "[" <statePaletteMatNum> "]"
                              | "program" "[" <stateProgramMatNum> "]"

    <stateMatrixRowNum>     ::= <integer> /* [0,3] */

    <stateOptModMatNum>     ::= /* empty */
                              | "[" <stateModMatNum> "]"

    <stateModMatNum>        ::= <integer> /*[0,MAX_VERTEX_UNITS_ARB-1]*/

    <statePaletteMatNum>    ::= <integer> /*[0,MAX_PALETTE_MATRICES_ARB-1]*/

    <stateProgramMatNum>    ::= <integer> /*[0,MAX_PROGRAM_MATRICES_ARB-1]*/

    <stateTexGenItem>       ::= "texgen" <optTexCoordNum> "."
                                <stateTexGenType> "." <stateTexGenCoord>

    <stateTexGenType>       ::= "eye"
                              | "object"

    <stateTexGenCoord>      ::= "s"
                              | "t"
```

```
                           | "r"
                           | "q"

    <stateClipPlaneItem>    ::= "clip" "[" <clipPlaneNum> "]" "." "plane"

    <statePointItem>        ::= "point" "." <statePointProperty>

    <statePointProperty>    ::= "size"
                              | "attenuation"

    <programSingleItem>     ::= <progEnvParam>
                              | <progLocalParam>

    <programMultipleItem>   ::= <progEnvParams>
                              | <progLocalParams>

    <progEnvParams>         ::= "program" "." "env" "[" <progEnvParamNums> "]"

    <progEnvParamNums>      ::= <progEnvParamNum>
                              | <progEnvParamNum> ".." <progEnvParamNum>

    <progEnvParam>          ::= "program" "." "env" "[" <progEnvParamNum> "]"

    <progLocalParams>       ::= "program" "." "local" "[" <progLocalParamNums>
                                "]"

    <progLocalParamNums>    ::= <progLocalParamNum>
                              | <progLocalParamNum> ".." <progLocalParamNum>

    <progLocalParam>        ::= "program" "." "local" "[" <progLocalParamNum>
                                "]"

    <progEnvParamNum>       ::= <integer>
                                /*[0,MAX_PROGRAM_ENV_PARAMETERS_ARB-1]*/

    <progLocalParamNum>     ::= <integer>
                                /*[0,MAX_PROGRAM_LOCAL_PARAMETERS_ARB-1]*/

    <constantVector>        ::= "{" <constantVectorList> "}"

    <constantVectorList>    ::= <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar>
                              | <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar> ","
                                <signedConstantScalar>

    <signedConstantScalar>  ::= <optSign> <constantScalar>

    <constantScalar>        ::= <floatConstant>

    <floatConstant>         ::= <float>
    <tempUseV>              ::= <tempVarName> <swizzleSuffix>

    <tempUseS>              ::= <tempVarName> <scalarSuffix>

    <tempUseVNS>            ::= <tempVarName>

    <tempUseW>              ::= <tempVarName> <optWriteMask>

    <resultUseW>            ::= <resultBasic> <optWriteMask>
                              | <resultVarName> <optWriteMask>
                              | <resultVtxColor> <optWriteMask>
                              | <resultVtxColor> "." <colorType>
                                <optWriteMask>
                              | <resultVtxColor> "." <faceType> <optWriteMask>
                              | <resultVtxColor> "." <faceType> "."
                                <colorType> "." <optWriteMask>

    <resultUseD>            ::= <resultBasic>
                              | <resultVtxColor>
                              | <resultVtxColor> "." <colorType>
                              | <resultVtxColor> "." <faceType>
                              | <resultVtxColor> "." <faceType> "."
                                <colorType>

    <resultBasic>           ::= "result" "." <resultVtxBasic>

    <resultVtxBasic>        ::= "position"
                              | "fogcoord"
                              | "pointsize"
                              | "texcoord" <optTexCoordNum>

    <resultVtxColor>        ::= "result" "." "color"

    <arrayMem>              ::= <arrayMemAbs>
                              | <arrayMemRel>

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

    <arrayMemAbs>           ::= <integer>

    <arrayMemRelOffset>     ::= /* empty */
                              | "+" <integer>
                              | "-" <integer>

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

    <addrUseW>              ::= <addrVarName> <addrWriteMask>

    <addrWriteMask>         ::= "." "x"

    <optWriteMask>          ::= /* empty */
                              | <xyzwMask>

    <xyzwMask>              ::= "." "x"
                              | "." "y"
                              | "." "xy"
                              | "." "z"
                              | "." "xz"
                              | "." "yz"
                              | "." "xyz"
                              | "." "w"
                              | "." "xw"
                              | "." "yw"
                              | "." "xyw"
                              | "." "zw"
                              | "." "xzw"
                              | "." "yzw"
                              | "." "xyzw"

    <swizzleSuffix>         ::= /* empty */
                              | "." <component>
                              | "." <xyzwComponent> <xyzwComponent>
                                <xyzwComponent> <xyzwComponent>

    <extendedSwizzle>       ::= <extSwizComp> "," <extSwizComp> ","
                                <extSwizComp> "," <extSwizComp>

    <extSwizComp>           ::= <optSign> <xyzwExtSwizSel>

    <xyzwExtSwizSel>        ::= "0"
                              | "1"
                              | <xyzwComponent>

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

    <addrComponent>         ::= "x"

    <scalarSuffix>          ::= "." <component>

    <component>             ::= <xyzwComponent>

    <xyzwComponent>         ::= "x"
                              | "y"
                              | "z"
                              | "w"

    <optSign>               ::= /* empty */
                              | "-"
                              | "+"

    <faceType>              ::= "front"
                              | "back"

    <colorType>             ::= "primary"
                              | "secondary"

    <vtxAttribNum>          ::= <integer> /*[0,MAX_VERTEX_ATTRIBS_ARB-1]*/

    <vtxOptWeightNum>       ::= /* empty */
                              | "[" <vtxWeightNum> "]"

    <vtxWeightNum>          ::= <integer> /*[0,MAX_VERTEX_UNITS_ARB-1] must be
                                divisible by four */
    <optTexCoordNum>        ::= /* empty */
                              | "[" <texCoordNum> "]"

    <texCoordNum>           ::= <integer> /*[0,MAX_TEXTURE_COORDS_ARB-1]*/

    <clipPlaneNum>          ::= <integer> /*[0,MAX_CLIP_PLANES-1]*/
```

The <integer>, <float>, and <identifier> grammar rules match integer constants, floating point constants, and identifier names as described in the ARB_vertex_program specification. The <float> grammar rule here is identical to the <floatConstant> grammar rule in ARB_vertex_program.

The grammar rules <tempVarName>, <addrVarName>, <attribVarName>, <paramArrayVarName>, <paramSingleVarName>, <resultVarName> refer to the names of temporary, address register, attribute, program parameter array, program parameter, and result variables declared in the program text.

**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 inst-ruction count | 2.14.4.4 | - |
| MAX_PROGRAM_CALL_DEPTH_NV | Z+ | GetProgramivARB | 4 | maximum program call stack depth | 2.14.4.4 | - |

(add to Table X.11. New Implementation-Dependent Values Introduced by ARB_vertex_program. Values queried by GetProgramivARB require a <pname> of VERTEX_PROGRAM_ARB.)

**Revision History**

| Rev. | Date | Author | Changes |
|---|---|---|---|
| 2 | 05/16/04 | pbrown | Documented terminals in modified vertex program grammar. |
| 1 | -------- | pbrown | Internal pre-release revisions. |