NVIDIA OpenGL Extension Specifications for the CineFX Architecture (NV30)

NVIDIA Corporation

Mark J. Kilgard, *editor*

*mjk@nvidia.com*

September 23, 2002

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

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


Other OpenGL extension specifications can be found at:

http://oss.sgi.com/projects/ogl-sample/registry/
# Table of Contents

<table>
<thead>
<tr>
<th>Table of NVIDIA OpenGL Extension Support</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT_blend_func_separate</td>
<td>7</td>
</tr>
<tr>
<td>EXT_stencil_two_side</td>
<td>10</td>
</tr>
<tr>
<td>NV_element_array</td>
<td>19</td>
</tr>
<tr>
<td>NV_float_buffer</td>
<td>25</td>
</tr>
<tr>
<td>NV_fragment_program</td>
<td>42</td>
</tr>
<tr>
<td>NV_half_float</td>
<td>136</td>
</tr>
<tr>
<td>NV_primitive_restart</td>
<td>148</td>
</tr>
<tr>
<td>NV_vertex_program2</td>
<td>153</td>
</tr>
</tbody>
</table>
## Table of NVIDIA OpenGL Extension Support

<table>
<thead>
<tr>
<th>Extension</th>
<th>RIVA TNT family</th>
<th>NV1x family</th>
<th>NV2x family</th>
<th>NV3x family</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARB_depth_texture</td>
<td>Em</td>
<td>Post-R25</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_imaging</td>
<td>R10</td>
<td>R10</td>
<td>X</td>
<td>X</td>
<td>1.2 imaging subset</td>
</tr>
<tr>
<td>ARB_multisample</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_multitexture</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_point_parameters</td>
<td>R35</td>
<td>R35</td>
<td>R35</td>
<td>X</td>
<td>1.4 functionality</td>
</tr>
<tr>
<td>ARB_shadow</td>
<td>Em</td>
<td>Post-R25</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_texture_border_clamp</td>
<td>Em</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_texture_compression</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_texture_cube_map</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_texture_env_add</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_texture_env_combine</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>ARB_texture_env_crossbar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See explanation</td>
</tr>
<tr>
<td>ARB_texture_env_dot3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
</tr>
<tr>
<td>ARB_transpose_matrix</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_abgr</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
</tr>
<tr>
<td>EXT_bgra</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
</tr>
<tr>
<td>EXT_blend_color</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_blend_func_separate</td>
<td>Em</td>
<td>Em</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_blend_minmax</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_blend_subtract</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_clip_volume_hint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R20+</td>
</tr>
<tr>
<td>EXT_compiled_vertex_array</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EXT_draw_range_elements</td>
<td>R20</td>
<td>R20</td>
<td>R20</td>
<td>X</td>
<td>1.2 functionality</td>
</tr>
<tr>
<td>EXT_fog_coord</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_multi_draw_arrays</td>
<td>R25</td>
<td>R25</td>
<td>R25</td>
<td>X</td>
<td>1.4 functionality</td>
</tr>
<tr>
<td>EXT_packed_pixels</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.2 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_packed_texture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_point_parameters</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
</tr>
<tr>
<td>EXT_rescale_normal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.2 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_secondary_color</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
</tr>
<tr>
<td>EXT_separate_specular_color</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.2 functionality</td>
</tr>
<tr>
<td>EXT_shadow_funcs</td>
<td>Em</td>
<td>Post-R25</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_shared_texture_palette</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_stencil_wrap</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_stencil_two_side</td>
<td>Em</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_texture3D</td>
<td>sw</td>
<td>sw</td>
<td>X</td>
<td>X</td>
<td>1.2 functionality</td>
</tr>
<tr>
<td>EXT_texture_compression_s3tc</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>use ARB version</td>
</tr>
<tr>
<td>EXT_texture_cube_map</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_texture_edge_clamp</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.2 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_texture_env_add</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_texture_env_dot3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_texture_env_combine</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.3 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_texture_filter_anisotropic</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_texture_lod_bias</td>
<td>R10</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
</tr>
<tr>
<td>EXT_texture_object</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.1 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_vertex_array</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.1 functionality</td>
<td></td>
</tr>
<tr>
<td>EXT_vertex_weighting</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KTX_buffer_region</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP_occlusion_test</td>
<td>Em</td>
<td>R25</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM_texture_mirrored_repeat</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
</tr>
<tr>
<td>NV_blend_square</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1.4 functionality</td>
<td></td>
</tr>
<tr>
<td>NV_copy_depth_to_color</td>
<td>Em</td>
<td>R20</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_depth_clamp</td>
<td>Em</td>
<td>Post-R25</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_evaluators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Discontinued</td>
</tr>
<tr>
<td>NV_element_array</td>
<td>Em</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_fence</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_float_buffer</td>
<td>Em</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_fog_distance</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NV_fragment_program</td>
<td>Em</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_half_float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>NV_light_max_exponent</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Extension</td>
<td>RIVA TNT family</td>
<td>NV1x family</td>
<td>NV2x family</td>
<td>NV3x family</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>NV_multisample_filter_hint</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_occlusion_query</td>
<td>Em</td>
<td>R25</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_packed_depth_stencil</td>
<td>R10+</td>
<td>R10+</td>
<td>R10+</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NV_pixel_data_range</td>
<td>R40</td>
<td>R40</td>
<td>R40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_point_sprite</td>
<td>R35+</td>
<td>R25</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NV_primitive_restart</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_register_combiners</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_register_combiners2</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texgen_emboss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texgen_reflection</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X use 1.3 functionality</td>
</tr>
<tr>
<td>NV_texture_compression_vtc</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texture_env_combine4</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texture_rectangle</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texture_shader</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texture_shader2</td>
<td>Em</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_texture_shader3</td>
<td>Em</td>
<td>R25</td>
<td></td>
<td></td>
<td>X only NV25 &amp; NV3x in HW</td>
</tr>
<tr>
<td>NV_vertex_array_range</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_vertex_array_range2</td>
<td></td>
<td>R10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_vertex_program</td>
<td>R10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_vertex_program1</td>
<td></td>
<td>R25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NV_vertex_program2</td>
<td>Em</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1_s3tc</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X no spec</td>
</tr>
<tr>
<td>SGIS_generate_mipmap</td>
<td></td>
<td>R10</td>
<td></td>
<td></td>
<td>X X 1.4 functionality</td>
</tr>
<tr>
<td>SGIS_multitexture</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X use 1.3 version</td>
</tr>
<tr>
<td>SGIS_texture_lod</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X 1.2 functionality</td>
</tr>
<tr>
<td>SGIX_depth_texture</td>
<td></td>
<td>Em</td>
<td></td>
<td></td>
<td>X X use 1.4 version</td>
</tr>
<tr>
<td>SGIX_shadow</td>
<td></td>
<td>Em</td>
<td></td>
<td></td>
<td>X X use 1.4 version</td>
</tr>
<tr>
<td>WGL_ARB_buffer_region</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_ARB_extensions_string</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_ARB_multisample</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X see ARB multisample</td>
</tr>
<tr>
<td>WGL_ARB_pixel_format</td>
<td></td>
<td>R10</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_ARB_pbuffer</td>
<td></td>
<td>R10</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_ARB_render_texture</td>
<td></td>
<td>R25</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_EXT_extensions_string</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_EXT_swap_control</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_NV_float_buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X see NV float buffer</td>
</tr>
<tr>
<td>WGL_NV_render_depth_texture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WGL_NV_render_texture_rectangle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X Win32</td>
</tr>
<tr>
<td>WIN_swap_hint</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X X Win32, no spec</td>
</tr>
</tbody>
</table>

Key for table entries:

- **X** = supported
- **sw** = supported by software rasterization (expect poor performance)
- **Em** = like sw, but only supported when appropriate emulation mode (NV20 Emulate, NV25 Emulate, NV30 Emulate) is enabled - emulated extensions may be extremely slow and not suitable for production use
- **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 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.

no spec = no suitable specification available

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

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>RGB Factors</th>
<th>Alpha Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>(0, 0, 0)</td>
<td>0</td>
</tr>
<tr>
<td>ONE</td>
<td>(1, 1, 1)</td>
<td>1</td>
</tr>
<tr>
<td>DST_COLOR</td>
<td>(Rd/Kr, Gd/Kg, Bd/Kb)</td>
<td>Ad/Ka</td>
</tr>
<tr>
<td>ONE_MINUS_DST_COLOR</td>
<td>(1-Rd/Kr, 1-Gd/Kg, 1-Bd/Kb)</td>
<td>1-Ad/Ka</td>
</tr>
<tr>
<td>SRC_ALPHA</td>
<td>(As/Ka, As/Ka, As/Ka)</td>
<td>As/Ka</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>(1-As/Ka, 1-As/Ka, 1-As/Ka)</td>
<td>1-As/Ka</td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td>(Ad/Ka, Ad/Ka, Ad/Ka)</td>
<td>Ad/Ka</td>
</tr>
<tr>
<td>ONE_MINUS_DST_ALPHA</td>
<td>(1-Ad/Ka, 1-Ad/Ka, 1-Ad/Ka)</td>
<td>1-Ad/Ka</td>
</tr>
<tr>
<td>CONSTANT_COLOR</td>
<td>(Rc, Gc, Bc)</td>
<td>Ac</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_COLOR</td>
<td>(1-Rc, 1-Gc, 1-Bc)</td>
<td>1-Ac</td>
</tr>
<tr>
<td>CONSTANT_ALPHA</td>
<td>(Ac, Ac, Ac)</td>
<td>Ac</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_ALPHA</td>
<td>(1-Ac, 1-Ac, 1-Ac)</td>
<td>1-Ac</td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>(f, f, f)</td>
<td>1</td>
</tr>
</tbody>
</table>

The commands that control blending are

```c
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.

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Get Command</th>
<th>Type</th>
<th>Value</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEND_SRC_RGB_EXT</td>
<td>GetFloatv</td>
<td>Z</td>
<td>ONE</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_DST_RGB_EXT</td>
<td>GetFloatv</td>
<td>Z</td>
<td>ZERO</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_SRC_ALPHA_EXT</td>
<td>GetFloatv</td>
<td>Z</td>
<td>ONE</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_DST_ALPHA_EXT</td>
<td>GetFloatv</td>
<td>Z</td>
<td>ZERO</td>
<td>color-buffer</td>
</tr>
</tbody>
</table>

New Implementation Dependent State

None
Name

EXT_stencil_two_side

Name Strings

GL_EXT_stencil_two_side

Notice

Copyright NVIDIA Corporation, 2001-2002.

Status


Version

NVIDIA Date: NVIDIA Date: August 29, 2002

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:

```c
glDepthMask(0);
glColorMask(0,0,0,0);
glEnable(GL_CULL_FACE);
glEnable(GL_STENCIL_TEST);

glStencilMask(~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);

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 buffer 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:

```c
    glStencilMask(~0);
    glStencilClear(0x1A);
    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 similar 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:

```c
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_TRIANGLES, 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

```c
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:

```c
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

```c
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:

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCIL_FUNC</td>
<td>2xZ8</td>
<td>GetIntegerv</td>
<td>ALWAYS</td>
<td>Stencil function</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
<tr>
<td>STENCIL_VALUE_MASK</td>
<td>2xZ+</td>
<td>GetIntegerv</td>
<td>1’s</td>
<td>Stencil mask</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
<tr>
<td>STENCIL_REF</td>
<td>2xZ+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stencil reference</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
<tr>
<td>STENCIL_FAIL</td>
<td>2xZ6</td>
<td>GetIntegerv</td>
<td>KEEP</td>
<td>Stencil fail action</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
<tr>
<td>STENCIL_PASS_DEPTH_FAIL</td>
<td>2xZ6</td>
<td>GetIntegerv</td>
<td>KEEP</td>
<td>Stencil depth buffer fail action</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
<tr>
<td>STENCIL_PASS_DEPTH_PASS</td>
<td>2xZ6</td>
<td>GetIntegerv</td>
<td>KEEP</td>
<td>Stencil depth buffer pass action</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
</tbody>
</table>

(Type field is amended with "2x" prefix.)

(table 6.15, page 205) add the following entries:

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCIL_TEST_TWO_SIDE_EXT</td>
<td>B</td>
<td>IsEnabled</td>
<td>False</td>
<td>Two-sided stencil test enable</td>
<td>4.1.4</td>
<td>stencil-buffer/enable</td>
</tr>
<tr>
<td>ACTIVE_STENCIL_FACE_EXT</td>
<td>Z2</td>
<td>GetIntegerv</td>
<td>FRONT</td>
<td>Active stencil face selector</td>
<td>4.1.4</td>
<td>stencil-buffer</td>
</tr>
</tbody>
</table>

(table 6.16, page 205) amend the following entry:

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCIL_WRITE_MASK</td>
<td>2xZ+</td>
<td>GetIntegerv</td>
<td>1’s</td>
<td>Stencil buffer writemask</td>
<td>4.2.2</td>
<td>stencil-buffer</td>
</tr>
</tbody>
</table>

(Type field is amended with "2x" prefix.)
Revision History

None
Name

NV_element_array

Name Strings

GL_NV_element_array

Notice

Copyright NVIDIA Corporation, 2002.

Status


Version

NVIDIA Date: August 29, 2002 (version 0.1)

Number

??

 Dependencies

Written based on the wording of the OpenGL 1.3 specification.

Assumes support for the NV_vertex_array_range extension.

Overview

This extension allows applications to provide indices for the DrawElements-style vertex array commands with higher performance. The indices can be placed inside a vertex array range, allowing hardware to pull them directly from an application-managed buffer, to achieve maximum performance.

Automatic pulling of indices reduces the amount of CPU work required to transfer large amounts of geometry to a minimal level, freeing up that CPU time for other tasks.

Though it is anticipated that this extension will be used almost exclusively in concert with NV_vertex_array_range, there is no fundamental reason that it must be used in that way.

Issues

* Is this extension necessary, or will ATI_element_array suffice?

RESOLVED: This extension is necessary. For one, it will avoid confusion, seeing as this extension interacts with NV_vertex_array_range, while ATI_element_array interacts with the ATI_vertex_array_object extension. In addition, that extension is missing a number of features that we desire to expose.
* Should this be a piece of state or be determined based on the pointer handed to DrawElements?

RESOLVED: A piece of state. Checking the argument adds overhead to a critical driver path.

* Should the array be called "index" or "element"?

RESOLVED: "Element" is preferable, because there is already an IndexPointer for color index.

* Should entry points be offered in the style of EXT_multi_draw_arrays?

RESOLVED: Yes. There are a total of four new drawing entry points.

* Which name should be used, "DrawElementArrayNV" or "DrawElementsArrayNV"?

RESOLVED: The former.

* Can element arrays be strided?

RESOLVED: No. I see no reason to allow this.

* Should there be a "first" argument to DrawElementArrayNV?

RESOLVED: Yes. It is more consistent with the DrawArrays model, and it is a useful feature.

* Should DrawRangeElementArray exist, or should "start" and "end" be properties of the element array specification?

RESOLVED: This entry point should be present, because the "start" and "end" might be different for different segments of the element array.

* Must the element array be enabled?

RESOLVED: No. There is no reason to require this, since its use is implied. This also does not impose any compatibility risk for the possibility of both extensions being implemented, because the "NV" entry points would simply ignore this enable.

It might conceivably make sense to force the user to enable this array if one were to support an index array for each vertex attribute in the future, but we see that as a dead-end feature.

So this extension provides no enable for the element array.

* Should the Gets use the same enumerants as the ATI spec?

RESOLVED: Yes. It's essentially the same state.
* Is there a limit on the maximum offset (i.e. first+count-1) usable with NV_element_array?

RESOLVED: No such limit is required.

* How should the state push/pop?

RESOLVED: Yes, as client state under GL_CLIENT_VERTEX_ARRAY_BIT. ATI_element_array uses "client" as the attrib type, but this does not make clear what bit is to be used. It would only make sense, however, for this to be vertex array state. The OpenGL 1.3 specification's state tables use the term "vertex-array" in this column; that convention has been maintained in this spec.

* How do vertex array range validity checks work with this extension?

RESOLVED: They don't interact.

New Procedures and Functions

void ElementPointerNV(enum type, const void *pointer);
void DrawElementArrayNV(enum mode, int first, sizei count);
void DrawRangeElementArrayNV(enum mode, uint start, uint end,
                              int first, sizei count);
void MultiDrawElementArrayNV(enum mode, const int *first,
                             const sizei *count, sizei primcount);
void MultiDrawRangeElementArrayNV(enum mode, uint start, uint end,
                                   const int *first,
                                   const sizei *count,
                                   sizei primcount);

New Tokens

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

    ELEMENT_ARRAY_TYPE_NV     0x8769

Accepted by the <pname> parameter of GetPointerv:

    ELEMENT_ARRAY_POINTER_NV  0x876A

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

Add ElementPointerNV to the list of commands that are disallowed inside a Begin and End, but may or may not generate an error, in section 2.6.3 "GL Commands within Begin/End" (page 19).

Add to section 2.8 "Vertex Arrays", following the description of DrawRangeElements and before InterleavedArrays (page 26):

"The command

    void ElementPointerNV(enum type, const void *pointer)

specifies the current element array. type must be one of
UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT, indicating that the values referenced by pointer are indices of GL type ubyte, ushort, or uint respectively.

The commands

```c
void DrawElementArrayNV(enum mode, int first, sizei count)
void DrawRangeElementArrayNV(enum mode, uint start, uint end, int first, sizei count)
```

operate similarly to DrawElements and DrawRangeElements, except that the type and indices arguments have implicitly been provided by the ElementPointerNV command. The indices argument is computed as:

```
pointer + first*size
```

where size is the size in basic machine units of the current element array type.

The command

```c
void MultiDrawElementArrayNV(enum mode, const int *first, const sizei *count, sizei primcount)
```

is equivalent to the following sequence of commands:

```c
for (i = 0; i < primcount; i++) {
    if ((first[i] >= 0) && (count[i] > 0)) {
        DrawElementArrayNV(mode, first[i], count[i]);
    }
}
```

The command

```c
void MultiDrawRangeElementArrayNV(enum mode, uint start, uint end, const int *first, const sizei *count, sizei primcount)
```

is equivalent to the following sequence of commands:

```c
for (i = 0; i < primcount; i++) {
    if ((first[i] >= 0) && (count[i] > 0)) {
        DrawRangeElementArrayNV(mode, start, end, first[i], count[i]);
    }
}
```

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 5+k boolean values, 6+k memory pointers, 5+k integer stride values, 5+k symbolic constants representing array types, and 3+k integers representing values per element. 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, except for the element array type, which is UNSIGNED_INT, and the integers representing values per element are each four.

Add to the section describing the operation of the vertex array range:

"When the vertex array range is valid, changes to element array data needed by the DrawElementArrayNV, DrawRangeElementArrayNV, MultiDrawElementArrayNV, and MultiDrawRangeElementArrayNV commands may behave in exactly the same non-sequential fashion as changes to vertex array data.

DrawElementArrayNV, DrawRangeElementArrayNV, MultiDrawElementArrayNV, and MultiDrawRangeElementArrayNV commands may generate undefined vertices when the vertex array range is enabled if any element required in the element array falls outside of the vertex array range."

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)

Replace the last sentence of the first paragraph of section 5.4 "Display Lists" (page 191) with the following:

"(Vertex array and element array pointers are dereferenced when the commands ArrayElement, DrawArrays, DrawElements, DrawRangeElements, DrawElementArrayNV, DrawRangeElementArrayNV, MultiDrawElementArrayNV, MultiDrawRangeElementArrayNV are accumulated into a display list.)"

Add to the end of Section 5.4 "Display Lists":

"ElementArrayNV is not compiled into display lists, but is executed immediately.

If a display list is compiled while VERTEX_ARRAY_RANGE_NV is enabled, the commands ArrayElement, DrawArrays, DrawElements, DrawRangeElements, DrawElementArrayNV, DrawRangeElementArrayNV, MultiDrawElementArrayNV, and MultiDrawRangeElementArrayNV are accumulated into a display list as if VERTEX_ARRAY_RANGE_NV is disabled."

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

Add ELEMENT_ARRAY_POINTER_NV to the list of possible values for the pname parameter of GetPointerv in section 6.1.11 "Pointer and String Queries" (page 206).
GLX Protocol

None

Errors

The error INVALID_OPERATION is generated if DrawElementArrayNV, DrawRangeElementArrayNV, MultiDrawElementArrayNV, or MultiDrawRangeElementArrayNV is called between the execution of Begin and the corresponding execution of End.

The error INVALID_ENUM is generated if ElementPointerNV is called where type is not one of UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT.

The error INVALID_VALUE is generated if DrawElementArrayNV or DrawRangeElementArrayNV is called where either first or count is negative.

The error INVALID_VALUE is generated if MultiDrawElementArrayNV or MultiDrawRangeElementArrayNV is called where primcount is negative.

The error INVALID_VALUE is generated if DrawRangeElementArrayNV or MultiDrawRangeElementArrayNV is called where start is greater than end.

The error INVALID_ENUM is generated if DrawElementArrayNV, DrawRangeElementArrayNV, MultiDrawElementArrayNV, or MultiDrawRangeElementArrayNV is called where mode is not one of the modes supported by the implementation.

New State

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Get Command</th>
<th>Type</th>
<th>Value</th>
<th>Sec</th>
<th>Attrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT_ARRAY_POINTER_NV</td>
<td>GetPointerv</td>
<td>Z+</td>
<td>0</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>ELEMENT_ARRAY_TYPE_NV</td>
<td>GetIntegerv</td>
<td>Z3</td>
<td>UNSIGNED_INT</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
</tbody>
</table>

NVIDIA Implementation Details

The NV30 implementation of this feature through VAR requires for hardware acceleration that:

- type be UNSIGNED_SHORT or UNSIGNED_INT
- ELEMENT_ARRAY_POINTER_NV be aligned to the size of one element, i.e. 2 bytes for a ushort, 4 bytes for a uint.

Failing to meet these requirements will lead to greatly subpar performance, especially if the indices are in uncached memory.
Name

NV_float_buffer

Name Strings

GL_NV_float_buffer
WGL_NV_float_buffer

Notice

Copyright NVIDIA Corporation, 2001-2002.

Status


Version

Last Modified:      $Date: 2002/08/29 $
NVIDIA Revision:    $Revision: #1 $

Number

Unassigned.

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 quantities 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 artificially 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 a 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_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.zxxz, H3.yyww;   # bilinear filter weights
TEX H1, R2.xyxy, TEX0, RECT; # lookup 1st sample
MULH H0, H1, H3.x;           # blend
TEX H1, R2.zzyy, TEX0, RECT; # lookup 2nd sample
MADH H0, H1, H3.y, H0;       # blend
TEX H0, R2.xwwx, TEX0, RECT; # lookup 3rd sample
MADH H0, H1, H3.z, H0;       # blend
TEX H1, R2.zwwy, 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 downsmaple filter, where floating-point component values are averaged. Components should not be clamped during any such mipmap generation.

New Procedures and Functions

None.

New Tokens

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

- `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> and <pfAttribFList> arrays of wglChoosePixelFormatARB:

- 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 paragraph, 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.)

<table>
<thead>
<tr>
<th>Base Internal Format</th>
<th>Pixel Group Type</th>
<th>Component Values</th>
<th>Internal Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>RGBA</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>LUMINANCE</td>
<td>RGBA</td>
<td>R</td>
<td>L</td>
</tr>
<tr>
<td>LUMINANCE_ALPHA</td>
<td>RGBA</td>
<td>R,A</td>
<td>L,A</td>
</tr>
<tr>
<td>INTENSITY</td>
<td>RGBA</td>
<td>R</td>
<td>I</td>
</tr>
<tr>
<td>RGB</td>
<td>RGBA</td>
<td>R,G,B</td>
<td>R,G,B</td>
</tr>
<tr>
<td>RGBA</td>
<td>RGBA</td>
<td>R,G,B,A</td>
<td>R,G,B,A</td>
</tr>
<tr>
<td>* COLOR_INDEX</td>
<td>CI</td>
<td>CI</td>
<td>CI</td>
</tr>
<tr>
<td>* DEPTH_COMPONENT</td>
<td>DEPTH</td>
<td>DEPTH</td>
<td>DEPTH</td>
</tr>
<tr>
<td>* HILO_NV</td>
<td>HILO</td>
<td>HI,LO</td>
<td>HI,LO</td>
</tr>
<tr>
<td>* DSDT_NV</td>
<td>TEXOFF</td>
<td>DS,DT</td>
<td>DS,DT</td>
</tr>
<tr>
<td>* DSDT_MAG_NV</td>
<td>TEXOFF</td>
<td>DS,DT,MAG</td>
<td>DS,DT,MAG</td>
</tr>
<tr>
<td>* DSDT_MAG_INTENSITY_NV</td>
<td>TEXOFF</td>
<td>or RGBA</td>
<td>DS,DT,MAG,VIB,DS,DT,MAG,I</td>
</tr>
<tr>
<td>FLOAT_R_NV</td>
<td>RGBA</td>
<td>R</td>
<td>R (float)</td>
</tr>
<tr>
<td>FLOAT_RG_NV</td>
<td>RGBA</td>
<td>R,G</td>
<td>R,G (float)</td>
</tr>
</tbody>
</table>

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.)
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)

(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)

<table>
<thead>
<tr>
<th>Base Internal Format</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT_R_NV</td>
<td>R</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FLOAT_RG_NV</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FLOAT_RGB_NV</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>FLOAT_RGBA_NV</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

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, auxiliary, 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_FLOAT_RGB_NV, or 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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Size</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>WGL_RED_BITS_ARB</td>
<td>RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA</td>
</tr>
<tr>
<td>G</td>
<td>WGL_GREEN_BITS_ARB</td>
<td>RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA</td>
</tr>
<tr>
<td>B</td>
<td>WGL_BLUE_BITS_ARB</td>
<td>RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA</td>
</tr>
<tr>
<td>A</td>
<td>WGL_ALPHA_BITS_ARB</td>
<td>RGB, RGBA, FLOAT_R, FLOAT_RG, FLOAT_RGB, FLOAT_RGBA</td>
</tr>
</tbody>
</table>

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.

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)

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_FLOAT_COMPONENTS_NV</td>
<td>n x B</td>
<td>GetTexLevel-</td>
<td>0</td>
<td>True if texture holds unclamped floating-point values</td>
<td>3.8</td>
<td>-</td>
</tr>
</tbody>
</table>

(Modify Table 6.19, Framebuffer Control, p. 227)

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR_COLOR_VALUE</td>
<td>C</td>
<td>GetFloatv</td>
<td>0,0,0,0</td>
<td>Color buffer clear value (RGBA mode), each value clamped to [0,1].</td>
<td>4.2.3</td>
<td>color-buffer</td>
</tr>
<tr>
<td>FLOAT_CLEAR_COLOR_VALUE_NV</td>
<td>4xR</td>
<td>GetFloatv</td>
<td>0,0,0,0</td>
<td>Color buffer clear value (RGBA mode), each value unclamped.</td>
<td>4.2.3</td>
<td>color-buffer</td>
</tr>
</tbody>
</table>

New Implementation Dependent State

(Modify Table 6.28, Implementation Dependent Values, p. 236)

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT_RGBA_MODE_NV</td>
<td>B</td>
<td>GetBooleanv</td>
<td>-</td>
<td>True if color buffers store floating-point data</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>
Name

NV_fragment_program

Name Strings

GL_NV_fragment_program

Notice

Copyright NVIDIA Corporation, 2001-2002.

IP Status

NVIDIA Proprietary.

Status


Version

Last Modified Date: $Date: 2002/08/29$
NVIDIA Revision: $Revision: #1$

Number

UNASSIGNED

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_register_combiners affects the definition of this extension.

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.

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 be no longer 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 perform 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 correspondence 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 explicitly 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 \(f[\text{COL0}]\) and \(f[\text{COL1}]\) 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, denoms are numbers smaller in absolute value than $2^{-126}$. For 16-bit floats used by this extension, denoms are numbers smaller in absolute value than $2^{-14}$.

**RESOLVED:** For 32-bit data types, hardware support for denoms was considered too expensive relative to the benefit provided. Computational results that would otherwise produce denoms are flushed to zero. For 16-bit data types, hardware denorm support will be
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:** NV_register_combiners is still available, even in fragment program mode, because it is commonly used and provides a powerful blending model. For example, it allows for four operands, fast 1-x operations, separate operations on color and alpha components, and more. These operations could be performed by fragment programs, but would require multiple instructions and program parameter constants.

Supporting both extensions simultaneously allows a programmer to write a program to obtain texture colors and then use the combiners to obtain a final fragment color.

As such, there are two different types of fragment programs: one "normal" and one for combiners. For combiner programs, the texture colors TEXTURE0 through TEXTURE3 are taken from output registers TEX0 through TEX3, respectively. The other combiner registers are not modified in fragment program mode.

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 distinction 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 by 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.
What is the clamping behavior on fragment program outputs feeding register combiners? Will the results be clamped to [0,1], [-1,1], or something else? Note that the "X" instructions clamp to [-2,2].

RESOLVED: Texture coordinate output registers will be clamped to the range [-1,1].

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 ":\((R0.x == 0) \| (R1.x == 0)\):

\[
\text{MOVC RC.x, R0.x;}
\text{MOVC RC.x (NE), R1.x;}
\]

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:

\[
\begin{align*}
\text{DEFINE halfTexel} & \equiv \{0.03125, 0\}; \quad \# 1/32 (1/2 a texel) \\
\text{ADD} & \quad \text{R0, f[TEX0], -halfTexel}; \quad \# \text{coords of left sample} \\
\text{ADD} & \quad \text{R1, f[TEX0], +halfTexel}; \quad \# \text{coords of right sample} \\
\text{TEX} & \quad \text{R0, R0, TEX0, 2D}; \quad \# \text{lookup left sample} \\
\text{TEX} & \quad \text{R1, R1, TEX0, 2D}; \quad \# \text{lookup right sample} \\
\text{MUL} & \quad \text{R2.x, R0.x, 16}; \quad \# \text{scale X coords to texels} \\
\text{FRC} & \quad \text{R2.x, R2.x}; \quad \# \text{get fraction, filter weight} \\
\text{LRP} & \quad \text{R0, R2, R1, R0}; \quad \# \text{blend samples based on weight}
\end{align*}
\]

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.

New Procedures and Functions

\[
\begin{align*}
\text{void ProgramNamedParameter4fNV} & \quad (\text{uint id, sizei len, const ubyte *name, float x, float y, float z, float w}); \\
\text{void ProgramNamedParameter4dNV} & \quad (\text{uint id, sizei len, const ubyte *name, double x, double y, double z, double w}); \\
\text{void ProgramNamedParameter4fvNV} & \quad (\text{uint id, sizei len, const ubyte *name, const float v[]}); \\
\text{void ProgramNamedParameter4dvNV} & \quad (\text{uint id, sizei len, const ubyte *name, const double v[]}); \\
\text{void GetProgramNamedParameterfvNV} & \quad (\text{uint id, sizei len, const ubyte *name, float *params}); \\
\text{void GetProgramNamedParameterdvNV} & \quad (\text{uint id, sizei len, const ubyte *name, double *params});
\end{align*}
\]
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, const float *params);
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

+-----------------+-----------------+
|                 |                 |
| Conventional    | Texture         |
| Texture Fetch   | Shaders         |
|                 | Fragment        |

+-----------------+-----------------+
| TEXTURE_        | SHADER_NV       |
| enable          |                  |

+-----------------+-----------------+
|                 |                 |
| Conventional    | Register        |
| TexEnv          | Combiners       |
| Color Sum       | Combiners       |
| Fog             |                 |

+-----------------+-----------------+
| REGISTER_       |                |
| COMBINERS_NV    |                  |
| NV enable       |                  |

+-----------------+-----------------+
|                 |                 |
| FRAGMENT_       | Program?        |
| PROGRAM_NV      |                  |
| NV enable       |                  |

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 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)

(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

\[ f = \frac{(1-t) * f_a / w_a + t * f_b / w_b}{(1-t) / w_a + t / w_b} \]

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

\[ f = (1-t) * f_a + t * f_b. \]  

... 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

\[ f = \frac{a * f_a / w_a + b * f_b / w_b + c * f_c / w_c}{a / w_a + b / w_b + c / w_c} \]

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

\[ f = a * f_a + b * f_b + c * f_c; \]  

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.

There are two types of fragment programs: A color fragment program is used to compute new color values to be associated with each fragment. A combiner fragment program is used to compute texture colors used by the NV_register_combiners extension. Either type of fragment program can be used to compute a new depth value to be associated with each fragment.

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 (NV_texture_shader extension) are disabled, regardless of the state of TEXTURE_SHADER_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.

<table>
<thead>
<tr>
<th>Fragment Attribute Register Name</th>
<th>Description</th>
<th>Component Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[WPOS]</td>
<td>Position of the fragment center.</td>
<td>(x,y,z,1/w)</td>
</tr>
<tr>
<td>f[COL0]</td>
<td>Interpolated primary color</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>f[COL1]</td>
<td>Interpolated secondary color</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>f[FOGC]</td>
<td>Interpolated fog distance/coord</td>
<td>(z,0,0,0)</td>
</tr>
<tr>
<td>f[TEX0]</td>
<td>Texture coordinate (unit 0)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX1]</td>
<td>Texture coordinate (unit 1)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX2]</td>
<td>Texture coordinate (unit 2)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX3]</td>
<td>Texture coordinate (unit 3)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX4]</td>
<td>Texture coordinate (unit 4)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX5]</td>
<td>Texture coordinate (unit 5)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX6]</td>
<td>Texture coordinate (unit 6)</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>f[TEX7]</td>
<td>Texture coordinate (unit 7)</td>
<td>(s,t,r,q)</td>
</tr>
</tbody>
</table>

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 \(f[\text{COL0}]\) and \(f[\text{COL1}]\) 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.

<table>
<thead>
<tr>
<th>Fragment Temporary Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R31</td>
<td>Four 32-bit (fp32) floating point values (s.e8.m23)</td>
</tr>
<tr>
<td>H0-H63</td>
<td>Four 16-bit (fp16) floating point values (s.e5.m10)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Output Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(o[\text{COLR}])</td>
<td>Final RGBA fragment color, fp32 format (color programs)</td>
</tr>
<tr>
<td>(o[\text{COLH}])</td>
<td>Final RGBA fragment color, fp16 format (color programs)</td>
</tr>
<tr>
<td>(o[\text{TEX0}])</td>
<td>TEXTURE0 output, fp16 format (combiner programs)</td>
</tr>
<tr>
<td>(o[\text{TEX1}])</td>
<td>TEXTURE1 output, fp16 format (combiner programs)</td>
</tr>
<tr>
<td>(o[\text{TEX2}])</td>
<td>TEXTURE2 output, fp16 format (combiner programs)</td>
</tr>
<tr>
<td>(o[\text{TEX3}])</td>
<td>TEXTURE3 output, fp16 format (combiner programs)</td>
</tr>
<tr>
<td>(o[\text{DEPR}])</td>
<td>Final fragment depth value, fp32 format</td>
</tr>
</tbody>
</table>

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 invalid 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAD R0, R1, 2, -1;</td>
<td>MAD R0, R1, {2,-1,0,0}.x, {2,-1,0,0}.y;</td>
</tr>
<tr>
<td>ADD R0, {1,2,3,4}, 4;</td>
<td>ADD R0, {1,2,3,4}.xyzw, {1,2,3,4}.w;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R0, {1,2,3,4}, -4;</td>
</tr>
<tr>
<td>ADD R0, {1,2,3,4},</td>
</tr>
<tr>
<td>ADD R0, {1,2,3,4}, -{-1,-2,-3,-4};</td>
</tr>
<tr>
<td>ADD R0, {1,2,3,4}, {4,5,6,7}.x;</td>
</tr>
</tbody>
</table>
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"
| "FRCX" | "FRCX_SAT"
| "FRCX" | "FRCX_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" | "MADR" | "MADH" | "MADX" | "MADC" | "MADRC" | "MADHC" | "MADX" | "LRP" | "LRPR" | "LRPH" | "LRPX" | "LRPC" | "LRPRC" | "LRPHC" | "LRPX" | "X2D" | "X2DR" | "X2DH" | "X2DC" | "X2DRC" | "X2DHC"**

**<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> <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" | "z" | "x", "y" | "x", "z" | "y", "z" | "x", "y", "z" | "w" | "x", "w" | "y", "w" | "z", "w" | "x", "y", "w" | "x", "z", "w" | "y", "z", "w" | "x", "y", "z", "w"

<srcRegister> ::= <fragAttribReg> | <fragTempReg>

<fragAttribReg> ::= "f" 
<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> ::= "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> ::= <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 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 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.

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", "ADDR", "ADDHC", "ADDX", "ADD_SAT", "ADDR_SAT", "ADDH_SAT", "ADDX_SAT", "ADDC_SAT", "ADDR_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.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inputs</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD[RHX][C]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>add</td>
</tr>
<tr>
<td>COS[RH][C]_[SAT]</td>
<td>s</td>
<td>ssss</td>
<td>cosine</td>
</tr>
<tr>
<td>DDX[RH][C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>derivative relative to x</td>
</tr>
<tr>
<td>DDY[RH][C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>derivative relative to y</td>
</tr>
<tr>
<td>DP3[RHX]_[SAT]</td>
<td>v,v</td>
<td>ssss</td>
<td>3-component dot product</td>
</tr>
<tr>
<td>DP4[RHX]_[SAT]</td>
<td>v,v</td>
<td>ssss</td>
<td>4-component dot product</td>
</tr>
<tr>
<td>DST[RH][C]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>distance vector</td>
</tr>
<tr>
<td>EX2[RH][C]_[SAT]</td>
<td>s</td>
<td>ssss</td>
<td>exponential base 2</td>
</tr>
<tr>
<td>FLO[RHX][C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>floor</td>
</tr>
<tr>
<td>FRC[RHX][C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>fraction</td>
</tr>
<tr>
<td>KIL</td>
<td>none</td>
<td>none</td>
<td>conditionally discard fragment</td>
</tr>
<tr>
<td>LG2[RH][C]_[SAT]</td>
<td>s</td>
<td>ssss</td>
<td>logarithm base 2</td>
</tr>
<tr>
<td>LIT[RH][C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>compute light coefficients</td>
</tr>
<tr>
<td>LRP[RHX]_[SAT]</td>
<td>v,v,v</td>
<td>v</td>
<td>linear interpolation</td>
</tr>
<tr>
<td>MAD[RHX]_[SAT]</td>
<td>v,v,v</td>
<td>v</td>
<td>multiply and add</td>
</tr>
<tr>
<td>MAX[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>maximum</td>
</tr>
<tr>
<td>MIN[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>minimum</td>
</tr>
<tr>
<td>MOV[RHX]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>move</td>
</tr>
<tr>
<td>MUL[RHX]_[SAT]</td>
<td>v,v,v</td>
<td>v</td>
<td>multiply</td>
</tr>
<tr>
<td>PK2H[RHX]_[SAT]</td>
<td>s</td>
<td>v</td>
<td>pack two 16-bit floats</td>
</tr>
<tr>
<td>PK2US[RHX]_[SAT]</td>
<td>s</td>
<td>v</td>
<td>pack two unsigned 16-bit scalars</td>
</tr>
<tr>
<td>PK4B[RHX]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>pack four signed 8-bit scalars</td>
</tr>
<tr>
<td>PK4UB[RHX]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>pack four unsigned 8-bit scalars</td>
</tr>
<tr>
<td>POW[RH][C]_[SAT]</td>
<td>s,s</td>
<td>ssss</td>
<td>exponentiation ((x^y))</td>
</tr>
<tr>
<td>RCP[RH][C]_[SAT]</td>
<td>s</td>
<td>ssss</td>
<td>reciprocal</td>
</tr>
<tr>
<td>RFL[RH][C]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>reflection vector</td>
</tr>
<tr>
<td>RSQ[RH][C]_[SAT]</td>
<td>s</td>
<td>ssss</td>
<td>reciprocal square root</td>
</tr>
<tr>
<td>SEQ[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on equal</td>
</tr>
<tr>
<td>SPL[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on false</td>
</tr>
<tr>
<td>SGE[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on greater than or equal</td>
</tr>
<tr>
<td>SGT[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on greater than</td>
</tr>
<tr>
<td>S1N[RH][C]_[SAT]</td>
<td>s</td>
<td>ssss</td>
<td>sine</td>
</tr>
<tr>
<td>SLE[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on less than or equal</td>
</tr>
<tr>
<td>SLT[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on less than</td>
</tr>
<tr>
<td>SNE[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on not equal</td>
</tr>
<tr>
<td>STR[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>set on true</td>
</tr>
<tr>
<td>SUB[RHX]_[SAT]</td>
<td>v,v</td>
<td>v</td>
<td>subtract</td>
</tr>
<tr>
<td>TEX[C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>texture lookup</td>
</tr>
<tr>
<td>TXD[C]_[SAT]</td>
<td>v,v,v</td>
<td>v</td>
<td>texture lookup w/partials</td>
</tr>
<tr>
<td>TXP[C]_[SAT]</td>
<td>v</td>
<td>v</td>
<td>projective texture lookup</td>
</tr>
<tr>
<td>UP2H[C]_[SAT]</td>
<td>s</td>
<td>v</td>
<td>unpack two 16-bit floats</td>
</tr>
<tr>
<td>UP2US[C]_[SAT]</td>
<td>s</td>
<td>v</td>
<td>unpack two unsigned 16-bit scalars</td>
</tr>
<tr>
<td>UP4B[C]_[SAT]</td>
<td>s</td>
<td>v</td>
<td>unpack four signed 8-bit scalars</td>
</tr>
<tr>
<td>UP4UB[C]_[SAT]</td>
<td>s</td>
<td>v</td>
<td>unpack four unsigned 8-bit scalars</td>
</tr>
<tr>
<td>X2D[RH][C]_[SAT]</td>
<td>v,v,v</td>
<td>v</td>
<td>2D coordinate transformation</td>
</tr>
</tbody>
</table>

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
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 <absScalarSrc> or <absVectorSrc> 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,GT).

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.

```c
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.

\[
\begin{align*}
\text{tmp0} &= \text{VectorLoad}(\text{op0}); \\
\text{tmp1} &= \text{VectorLoad}(\text{op1}); \\
\text{result.x} &= \text{tmp0}.x + \text{tmp1}.x; \\
\text{result.y} &= \text{tmp0}.y + \text{tmp1}.y; \\
\text{result.z} &= \text{tmp0}.z + \text{tmp1}.z; \\
\text{result.w} &= \text{tmp0}.w + \text{tmp1}.w;
\end{align*}
\]

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].

\[
\begin{align*}
\text{tmp} &= \text{ScalarLoad}(\text{op0}); \\
\text{result.x} &= \text{ApproxCosine}(\text{tmp}); \\
\text{result.y} &= \text{ApproxCosine}(\text{tmp}); \\
\text{result.z} &= \text{ApproxCosine}(\text{tmp}); \\
\text{result.w} &= \text{ApproxCosine}(\text{tmp});
\end{align*}
\]

The approximation function ApproxCosine is accurate to at least 22 bits with an angle in the range [0,2*PI].

\[|\text{ApproxCosine}(x) - \cos(x)| < 1.0 / 2^{22}, \text{ if } 0.0 \leq x < 2.0 \times \text{PI}.\]

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. \(\text{ApproxCosine}(\text{NaN}) = \text{NaN}\).
2. \(\text{ApproxCosine}(+/-\text{INF}) = \text{NaN}\).
3. \(\text{ApproxCosine}(+/-0.0) = +1.0\).
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.

\[ f = \text{VectorLoad}(\text{op0}); \]
\[ \text{result} = \text{ComputePartialX}(f); \]

Note that the partial derivatives 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.

\[ f = \text{VectorLoad}(\text{op0}); \]
\[ \text{result} = \text{ComputePartialY}(f); \]

Note that the partial derivatives 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.

\[ \text{tmp0} = \text{VectorLoad}(\text{op0}); \]
\[ \text{tmp1} = \text{VectorLoad}(\text{op1}); \]
\[ \text{result.x} = (\text{tmp0.x} \times \text{tmp1.x}) + (\text{tmp0.y} \times \text{tmp1.y}) + (\text{tmp0.z} \times \text{tmp2.z}); \]
\[ \text{result.y} = (\text{tmp0.x} \times \text{tmp1.x}) + (\text{tmp0.y} \times \text{tmp1.y}) + (\text{tmp0.z} \times \text{tmp2.z}); \]
\[ \text{result.z} = (\text{tmp0.x} \times \text{tmp1.x}) + (\text{tmp0.y} \times \text{tmp1.y}) + (\text{tmp0.z} \times \text{tmp2.z}); \]
\[ \text{result.w} = (\text{tmp0.x} \times \text{tmp1.x}) + (\text{tmp0.y} \times \text{tmp1.y}) + (\text{tmp0.z} \times \text{tmp2.z}); \]
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.

```c
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:

```c
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.

```c
    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:

\[
| \text{Approx2ToX}(x) - 2^x | < \frac{1.0}{2^{22}}, \text{ if } 0.0 \leq x < 1.0,
\]

and, in general,

\[
| \text{Approx2ToX}(x) - 2^x | < \left( \frac{1.0}{2^{22}} \right) \times (2^{\text{floor}(x))}.
\]

The following special-case rules apply to exponential approximation:

1. \text{Approx2ToX}(\text{NaN}) = \text{NaN}.
2. \text{Approx2ToX}(-\text{INF}) = +0.0.
3. \text{Approx2ToX}(+\text{INF}) = +\text{INF}.
4. \text{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.

```c
    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. \text{floor}(\text{NaN}) = \text{NaN}.
2. \text{floor}(<x>) = <x>, for -0.0, +0.0, -\text{INF}, and +\text{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.

\[
\text{tmp} = \text{VectorLoad}(\text{op0});
\]
\[
\text{result}.x = \text{tmp}.x - \text{floor}(\text{tmp}.x);
\]
\[
\text{result}.y = \text{tmp}.y - \text{floor}(\text{tmp}.y);
\]
\[
\text{result}.z = \text{tmp}.z - \text{floor}(\text{tmp}.z);
\]
\[
\text{result}.w = \text{tmp}.w - \text{floor}(\text{tmp}.w);
\]

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.

\[
\text{if (TestCC(rc.c*** || TestCC(rc.\*c**)) ||}
\]
\[
\text{TestCC(rc.\*c*) || TestCC(rc.***c)))} \{
\]
\[
\text{// Discard the fragment.}
\]
\[
\text{else} \{
\]
\[
\text{// Do nothing.}
\]
\[
\}
\]

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.

\[
\text{tmp} = \text{ScalarLoad}(\text{op0});
\text{result.x} = \text{ApproxLog2}(\text{tmp});
\text{result.y} = \text{ApproxLog2}(\text{tmp});
\text{result.z} = \text{ApproxLog2}(\text{tmp});
\text{result.w} = \text{ApproxLog2}(\text{tmp});
\]

The approximation function is accurate to at least 22 bits:

\[| \text{ApproxLog2}(x) - \log_2(x) | < 1.0 / 2^{22}.\]

The following special-case rules apply to logarithm approximation:

1. \text{ApproxLog2}(\text{NaN}) = \text{NaN}.
2. \text{ApproxLog2}(+\text{INF}) = +\text{INF}.
3. \text{ApproxLog2}(+/-0.0) = -\text{INF}.
4. \text{ApproxLog2}(x) = \text{NaN}, -\text{INF} < x < -0.0.
5. \text{ApproxLog2}(-\text{INF}) = \text{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.

\[
\text{tmp} = \text{VectorLoad}(\text{op0});
\text{if (t.x < 0) } t.x = 0;
\text{if (t.y < 0) } t.y = 0;
\text{result.x} = 1.0;
\text{result.y} = t.x;
\text{result.z} = (t.x > 0) \ ? \text{ApproxPower}(t.y, t.w) : 0.0;
\text{result.w} = 1.0;
\]

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.

\[
\begin{align*}
\text{tmp0} &= \text{VectorLoad}(\text{op0}); \\
\text{tmp1} &= \text{VectorLoad}(\text{op1}); \\
\text{tmp2} &= \text{VectorLoad}(\text{op2}); \\
\text{result.x} &= \text{tmp0.x} \times \text{tmp1.x} + (1 - \text{tmp0.x}) \times \text{tmp2.x}; \\
\text{result.y} &= \text{tmp0.y} \times \text{tmp1.y} + (1 - \text{tmp0.y}) \times \text{tmp2.y}; \\
\text{result.z} &= \text{tmp0.z} \times \text{tmp1.z} + (1 - \text{tmp0.z}) \times \text{tmp2.z}; \\
\text{result.w} &= \text{tmp0.w} \times \text{tmp1.w} + (1 - \text{tmp0.w}) \times \text{tmp2.w};
\end{align*}
\]

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.

\[
\begin{align*}
\text{tmp0} &= \text{VectorLoad}(\text{op0}); \\
\text{tmp1} &= \text{VectorLoad}(\text{op1}); \\
\text{tmp2} &= \text{VectorLoad}(\text{op2}); \\
\text{result.x} &= \text{tmp0.x} \times \text{tmp1.x} + \text{tmp2.x}; \\
\text{result.y} &= \text{tmp0.y} \times \text{tmp1.y} + \text{tmp2.y}; \\
\text{result.z} &= \text{tmp0.z} \times \text{tmp1.z} + \text{tmp2.z}; \\
\text{result.w} &= \text{tmp0.w} \times \text{tmp1.w} + \text{tmp2.w};
\end{align*}
\]

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.

\[
\begin{align*}
\text{tmp0} &= \text{VectorLoad}(\text{op0}); \\
\text{tmp1} &= \text{VectorLoad}(\text{op1}); \\
\text{result.x} &= \max(\text{tmp0.x}, \text{tmp1.x}); \\
\text{result.y} &= \max(\text{tmp0.y}, \text{tmp1.y}); \\
\text{result.z} &= \max(\text{tmp0.z}, \text{tmp1.z}); \\
\text{result.w} &= \max(\text{tmp0.w}, \text{tmp1.w});
\end{align*}
\]

The following special cases apply to the maximum operation:

1. \(\max(A,B)\) is always equivalent to \(\max(B,A)\).
2. \(\max(\text{NaN}, <x>) = \text{NaN}\), 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.

```c
    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. $\text{min}(A,B)$ is always equivalent to $\text{min}(B,A)$.
2. $\text{min}(\text{NaN}, <x>) == \text{NaN}$, for all $<x>$.

Section 3.11.5.18, MOV: Move

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

```c
    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.

```c
    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. $\text{NaN} * <x> = \text{NaN}$, for all $<x>$.
3. $+/-0.0 * +/-\text{INF} = \text{NaN}$.
4. $+/-0.0 * <x> = +/-0.0$, for all $<x>$ except -$\text{INF}$, $+\text{INF}$, and $\text{NaN}$. The sign of the result is positive if the signs of the two operands match and negative otherwise.
5. $+/\text{INF} * <x> = +/-\text{INF}$, for all $<x>$ except $-0.0$, $+0.0$, and $\text{NaN}$. 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.

```c
  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.

```c
  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.

\[
\text{tmp0} = \text{VectorLoad(op0)};
\]

\[
\begin{align*}
\text{if (tmp0.x < -128/127) tmp0.x} & = -128/127; \\
\text{if (tmp0.y < -128/127) tmp0.y} & = -128/127; \\
\text{if (tmp0.z < -128/127) tmp0.z} & = -128/127; \\
\text{if (tmp0.w < -128/127) tmp0.w} & = -128/127;
\end{align*}
\]

\[
\begin{align*}
\text{if (tmp0.x > +127/127) tmp0.x} & = +127/127; \\
\text{if (tmp0.y > +127/127) tmp0.y} & = +127/127; \\
\text{if (tmp0.z > +127/127) tmp0.z} & = +127/127; \\
\text{if (tmp0.w > +127/127) tmp0.w} & = +127/127;
\end{align*}
\]

\[
\text{ub.x} = \text{round}(127.0 \times \text{tmp0.x} + 128.0); \quad /* \text{ub is a ubyte vector} */
\]

\[
\begin{align*}
\text{ub.y} & = \text{round}(127.0 \times \text{tmp0.y} + 128.0); \\
\text{ub.z} & = \text{round}(127.0 \times \text{tmp0.z} + 128.0); \\
\text{ub.w} & = \text{round}(127.0 \times \text{tmp0.w} + 128.0);
\end{align*}
\]

\[
\begin{align*}
\text{result.x} & = (\text{ub.x}) \mid (\text{ub.y} \ll 8) \mid (\text{ub.z} \ll 16) \mid (\text{ub.w} \ll 24)); \\
\text{result.y} & = (\text{ub.x}) \mid (\text{ub.y} \ll 8) \mid (\text{ub.z} \ll 16) \mid (\text{ub.w} \ll 24)); \\
\text{result.z} & = (\text{ub.x}) \mid (\text{ub.y} \ll 8) \mid (\text{ub.z} \ll 16) \mid (\text{ub.w} \ll 24)); \\
\text{result.w} & = (\text{ub.x}) \mid (\text{ub.y} \ll 8) \mid (\text{ub.z} \ll 16) \mid (\text{ub.w} \ll 24));
\end{align*}
\]

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.

```c
    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.

```c
  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,

```c
  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. \( \text{ApproxPower}(\text{NaN}, <x>) = \text{NaN}, \)
2. \( \text{ApproxPower}(<x>, <y>) = \text{NaN}, \) if \( x < -0.0, \)
3. \( \text{ApproxPower}(+/0.0, <x>) = +0.0, \) if \( x > +0.0, \) or \( +\text{INF}, \) if \( x < -0.0, \)
4. \( \text{ApproxPower}(+1.0, <x>) = +1.0, \) if \( x \) is not NaN,
5. \( \text{ApproxPower}(+\text{INF}, <x>) = +\text{INF}, \) if \( x > +0.0, \) or \( +0.0, \) if \( x < -0.0, \)
6. \( \text{ApproxPower}(<x>, +/-0.0) = +1.0, \) if \( x >= -0.0 \)
7. \( \text{ApproxPower}(<x>, +\text{INF}) = +0.0, \) if \( -0.0 <= x < +1.0, \)
8. \( \text{ApproxPower}(<x>, +\text{INF}) = +\text{INF}, \) if \( -0.0 <= x < +1.0, \)
9. \( \text{ApproxPower}(<x>, +1.0) = <x>, \) if \( x >= +0.0, \) and
10. \( \text{ApproxPower}(<x>, \text{NaN}) = \text{NaN}. \)

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 contrast, 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.

```c
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.

```c
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.

```cpp
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:

\[ | \text{ApproxRSQRT}(x) - (1/x) | < 1.0 / 2^{22}, \text{ if } 1.0 \leq x < 4.0. \]

The following special-case rules apply to reciprocal square roots:

1. \( \text{ApproxRSQRT}(\text{NaN}) = \text{NaN}. \)
2. \( \text{ApproxRSQRT}(+\text{INF}) = +0.0. \)
3. \( \text{ApproxRSQRT}(-\text{INF}) = \text{NaN}. \)
4. \( \text{ApproxRSQRT}(+0.0) = +\text{INF}. \)
5. \( \text{ApproxRSQRT}(-0.0) = -\text{INF}. \)
6. \( \text{ApproxRSQRT}(x) = \text{NaN}, \text{ if } -\text{INF} < x < -0.0. \)

**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.

```cpp
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. \( (<x> == <y>) \text{ and } (<y> == <x>) \) always produce the same result.
2. \( (\text{NaN} == <x>) \) is FALSE for all \(<x>\), including NaN.
3. \( (+\text{INF} == +\text{INF}) \text{ and } (-\text{INF} == -\text{INF}) \) are TRUE.
4. \( (-0.0 == +0.0) \text{ and } (+0.0 == -0.0) \) 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.

```cpp
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 $[0,2\pi]$.

```c
    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 $[0,2\pi]$.

$$|\text{ApproxSine}(x) - \sin(x)| < 1.0 / 2^{22}, \text{ if } 0.0 \leq x < 2.0 \times \pi.$$  

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.

```c
    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.

```c
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.

```c
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.

```c
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.

\[
\begin{align*}
tmp0 &= \text{VectorLoad}(op0); \\
tmp1 &= \text{VectorLoad}(op1); \\
result.x &= tmp0.x - tmp1.x; \\
result.y &= tmp0.y - tmp1.y; \\
result.z &= tmp0.z - tmp1.z; \\
result.w &= tmp0.w - tmp1.w;
\end{align*}
\]

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 \(<\text{texImageTarget}>\) belonging to the texture image unit given by \(<\text{texImageUnit}>\). \(<\text{texImageTarget}>\) 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.
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, "HEMI" is defined as sqrt(MAX(0, 1-(HIIt^2 + LOt^2))) and will be the third component of a 3D unit vector whose first two components are HIIt and LOt.

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.

\[
\begin{align*}
\text{tmp} &= \text{ScalarLoad}(\text{op0}); \\
\text{result}.x &= (\text{fp16}) (\text{RawBits(tmp)} & \text{0xFFFF}); \\
\text{result}.y &= (\text{fp16}) ((\text{RawBits(tmp)} >> 16) & \text{0xFFFF}); \\
\text{result}.z &= (\text{fp16}) (\text{RawBits(tmp)} & \text{0xFFFF}); \\
\text{result}.w &= (\text{fp16}) ((\text{RawBits(tmp)} >> 16) & \text{0xFFFF}); \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{tmp} &= \text{ScalarLoad}(\text{op0}); \\
\text{result}.x &= ((\text{RawBits(tmp)} >> 0) & \text{0xFFFF}) / 65535.0; \\
\text{result}.y &= ((\text{RawBits(tmp)} >> 16) & \text{0xFFFF}) / 65535.0; \\
\text{result}.z &= ((\text{RawBits(tmp)} >> 0) & \text{0xFFFF}) / 65535.0; \\
\text{result}.w &= ((\text{RawBits(tmp)} >> 16) & \text{0xFFFF}) / 65535.0; \\
\end{align*}
\]

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.

\[
\begin{align*}
tmp0 &= \text{VectorLoad}(\text{op0}); \\
tmp1 &= \text{VectorLoad}(\text{op1}); \\
tmp2 &= \text{VectorLoad}(\text{op2}); \\
result.x &= tmp0.x + tmp1.x \times tmp2.x + tmp1.y \times tmp2.y; \\
result.y &= tmp0.y + tmp1.x \times tmp2.z + tmp1.y \times tmp2.w; \\
result.z &= tmp0.x + tmp1.x \times tmp2.x + tmp1.y \times tmp2.y; \\
result.w &= tmp0.y + tmp1.x \times tmp2.z + tmp1.y \times tmp2.w;
\end{align*}
\]

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

```c
void GenProgramsNV(sizei n, uint *ids);
```

returns \(n\) currently unused program names in \(ids\). These names are marked as used, for the purposes of \(\text{GenProgramsNV}\) only, but they become existent programs only when they are first loaded using \(\text{LoadProgramNV}\).

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

```c
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 \(\text{GetProgramivNV}\) (Section 6.1.13) with \(id\) set to the name of the program and \(pname\) set to PROGRAM_RESIDENT_NV.

\(\text{AreProgramsResidentNV}\) 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

```c
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. \(\text{RequestResidentProgramsNV}\) silently ignores attempts to make resident nonexistent program names or zero. \(\text{AreProgramsResidentNV}\) can be called after \(\text{RequestResidentProgramsNV}\) to determine which programs actually became resident.
The commands

```c
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 support 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

```c
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

```c
void GetProgramStringNV(uint id, enum pname,  
    ubyte *program);
```

obtains the program string for program id.  `pname` must be
`PROGRAM_STRING_NV`.  `n` bytes are returned into the array `program`
where `n` is the length of the program in bytes.  `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

```c
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

```c
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

```c
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

```c
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 occurred:

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 occurred:

1          1               reply
1                           unused
2          CARD16          sequence number
4          0               reply length
24                          unused

Errors

INVALID_OPERATION is generated by GetProgramNamedParameter4dvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4dvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4fvNV, or GetProgramNamedParameter4dvNV if <id> specifies a nonexistent program or a program whose type does not support local parameters.

INVALID_VALUE is generated by GetProgramNamedParameter4dvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4dvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4fvNV, or GetProgramNamedParameter4dvNV if <len> is zero.

INVALID_VALUE is generated by GetProgramNamedParameter4dvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4dvNV, GetProgramNamedParameter4fvNV, GetProgramNamedParameter4fvNV, or GetProgramNamedParameter4dvNV 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

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Section</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAGMENT_PROGRAM_NV</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>fragment program</td>
<td>3.11</td>
<td>enable</td>
</tr>
<tr>
<td>FRAGMENT_PROGRAM_BINDING_NV</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>bound fragment program</td>
<td>5.7</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table X.6. New State Introduced by NV_fragment_program.**
Table X.7. Program Object State common to NV_vertex_program and NV_fragment_program.

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Section</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM_ERROR_POSITION_NV</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>-1</td>
<td>program error</td>
<td>5.7</td>
<td>-</td>
</tr>
<tr>
<td>PROGRAM_TARGET_NV</td>
<td>Z2</td>
<td>GetProgramivNV</td>
<td>0</td>
<td>program target</td>
<td>6.1.13</td>
<td>-</td>
</tr>
<tr>
<td>PROGRAM_LENGTH_NV</td>
<td>Z+</td>
<td>GetProgramivNV</td>
<td>0</td>
<td>program length</td>
<td>6.1.13</td>
<td>-</td>
</tr>
<tr>
<td>PROGRAM_RESIDENT_NV</td>
<td>Z2</td>
<td>GetProgramivNV</td>
<td>False</td>
<td>program residency</td>
<td>6.1.13</td>
<td>-</td>
</tr>
<tr>
<td>PROGRAM_STRING_NV</td>
<td>ubxn</td>
<td>GetProgramStringNV</td>
<td>&quot;&quot;</td>
<td>program string</td>
<td>6.1.13</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>rdR4</td>
<td>GetProgramNamedParameterNV</td>
<td>(0,0,0,0)</td>
<td>named program local</td>
<td>5.7</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>64+xR4</td>
<td>GetProgramLocalParameterARB</td>
<td>(0,0,0,0)</td>
<td>numbered program</td>
<td>5.7</td>
<td>-</td>
</tr>
</tbody>
</table>

Table X.8. Fragment Program Per-Fragment Execution State.

New Implementation Dependent State

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Section</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_TEXTURE_COORDS_NV</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>2</td>
<td>number of texture coordinate sets supported</td>
<td>2.6</td>
<td>-</td>
</tr>
<tr>
<td>MAX_TEXTURE_IMAGE_UNITS_NV</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>2</td>
<td>number of texture image units supported</td>
<td>2.10.2</td>
<td>-</td>
</tr>
<tr>
<td>MAX_FRAGMENT_PROGRAM_LOCAL_PARAMETERS_NV</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>64</td>
<td>number of numbered local parameters supported</td>
<td>3.11.7</td>
<td>-</td>
</tr>
</tbody>
</table>
Name

NV_half_float

Name Strings

GL_NV_half_float

Notice

Copyright NVIDIA Corporation, 2001-2002.

IP Status

NVIDIA Proprietary.

Version

Last Modified Date: $Date: 2002/07/19 $
NVIDIA Revision: $Revision: #8 $

Number

UNASSIGNED

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

```c
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)

<table>
<thead>
<tr>
<th>Letter</th>
<th>Corresponding GL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>half</td>
</tr>
</tbody>
</table>

(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

\[
\begin{align*}
(-1)^{S} \times 0.0, & \quad \text{if } E = 0 \text{ and } M = 0, \\
(-1)^{S} \times 2^{-14} \times (M / 2^{10}), & \quad \text{if } E = 0 \text{ and } M \neq 0, \\
(-1)^{S} \times 2^{(E-15)} \times (1 + M/2^{10}), & \quad \text{if } 0 < E < 31, \\
(-1)^{S} \times \text{INF}, & \quad \text{if } E = 31 \text{ and } M = 0, \text{ or} \\
\text{NaN}, & \quad \text{if } E = 31 \text{ and } M \neq 0, \\
\end{align*}
\]

where

\[
\begin{align*}
S & = \text{floor}((N \text{ mod } 65536) / 32768), \\
E & = \text{floor}((N \text{ mod } 32768) / 1024), \text{ and} \\
M & = N \text{ mod } 1024.
\end{align*}
\]

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 values, zero, and INF have an associated sign. -0.0 and +0.0 should be 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)

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>16</td>
<td>half-precision floating-point value encoded in an unsigned scalar</td>
</tr>
</tbody>
</table>

Modify Section 2.7, Vertex Specification, p. 19

(Modify the descriptions of the immediate mode functions in this section, including those introduced by extensions.)

```c
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

<table>
<thead>
<tr>
<th>Command</th>
<th>Sizes</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexPointer</td>
<td>2,3,4</td>
<td>short, int, float, half, double</td>
</tr>
<tr>
<td>NormalPointer</td>
<td>3</td>
<td>byte, short, int, float, half, double</td>
</tr>
<tr>
<td>ColorPointer</td>
<td>3,4</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>IndexPointer</td>
<td>1</td>
<td>ubyte, short, int, float, double</td>
</tr>
<tr>
<td>TexCoordPointer</td>
<td>1,2,3,4</td>
<td>short, int, float, half, double</td>
</tr>
<tr>
<td>EdgeFlagPointer</td>
<td>1</td>
<td>boolean</td>
</tr>
<tr>
<td>FogCoordPointerEXT</td>
<td>1</td>
<td>float, half, double</td>
</tr>
<tr>
<td>SecondaryColorPointerEXT</td>
<td>3</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>VertexWeightPointerEXT</td>
<td>1</td>
<td>float, half</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>c</td>
</tr>
</tbody>
</table>


(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)

<table>
<thead>
<tr>
<th>type Parameter</th>
<th>Corresponding GL Data Type</th>
<th>Special Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF_FLOAT_NV</td>
<td>half</td>
<td>No</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>type</th>
<th>Parameter</th>
<th>GL Data Type</th>
<th>Component Conversion Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF_FLOAT_NV</td>
<td>half</td>
<td>c = f</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Vertex2hvNV</th>
<th>2</th>
<th>8</th>
<th>rendering command length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4240</td>
<td>rendering command opcode</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>FLOAT16</td>
<td>v[0]</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>FLOAT16</td>
<td>v[1]</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Command Name</th>
<th>Rendering Command Length</th>
<th>Rendering Command Opcode</th>
<th>Parameters</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecondaryColor3hvNV</td>
<td>2</td>
<td>4255</td>
<td>2 FLOAT16 v[0], 2 FLOAT16 v[1], 2 FLOAT16 v[2]</td>
<td></td>
</tr>
<tr>
<td>VertexWeighthvNV</td>
<td>2</td>
<td>4256</td>
<td>2 FLOAT16 v[0]</td>
<td>unused</td>
</tr>
<tr>
<td>VertexAttrib1hvNV</td>
<td>2</td>
<td>4257</td>
<td>4 CARD32 index, 2 FLOAT16 v[0]</td>
<td>unused</td>
</tr>
<tr>
<td>VertexAttrib2hvNV</td>
<td>2</td>
<td>4258</td>
<td>4 CARD32 index, 2 FLOAT16 v[0], 2 FLOAT16 v[1]</td>
<td></td>
</tr>
<tr>
<td>VertexAttrib3hvNV</td>
<td>2</td>
<td>4259</td>
<td>4 CARD32 index, 2 FLOAT16 v[0], 2 FLOAT16 v[1], 2 FLOAT16 v[2]</td>
<td>unused</td>
</tr>
<tr>
<td>VertexAttrib4hvNV</td>
<td>2</td>
<td>4260</td>
<td>4 CARD32 index, 2 FLOAT16 v[0], 2 FLOAT16 v[1], 2 FLOAT16 v[2], 2 FLOAT16 v[3]</td>
<td></td>
</tr>
<tr>
<td>VertexAttribs1hvNV</td>
<td>2 + (2n + p)</td>
<td>4261</td>
<td>4 CARD32 index, 2 CARD32 n, 2n LIST of FLOAT16 v, p = pad (2n)</td>
<td>unused</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>type</th>
<th>Encoding</th>
<th>Protocol Type</th>
<th>nbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF_FLOAT_NV</td>
<td>0x140B</td>
<td>CARD16</td>
<td>2</td>
</tr>
</tbody>
</table>

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

None.
Name

NV_half_float

Name Strings

GL_NV_half_float

Contact

Pat Brown, NVIDIA Corporation (pbrown 'at' nvidia.com)

Notice

Copyright NVIDIA Corporation, 2001-2002.

IP Status

NVIDIA Proprietary.

Status


Version

Last Modified Date: $Date: 2002/09/09 $
NVIDIA Revision: $Revision: #2 $

Number

UNASSIGNED

Dependencies

Written based on the wording of the OpenGL 1.1 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 VertexAttribS1hNV(uint index, sizei n, const half *v);
void VertexAttribS2hNV(uint index, sizei n, const half *v);
void VertexAttribS3hNV(uint index, sizei n, const half *v);
void VertexAttribS4hNV(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)

<table>
<thead>
<tr>
<th>Letter</th>
<th>Corresponding GL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>half</td>
</tr>
</tbody>
</table>

(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

\begin{align*}
    (-1)^S \times 0.0, & \quad \text{if } E = 0 \text{ and } M = 0, \\
    (-1)^S \times 2^{-14} \times (M / 2^{10}), & \quad \text{if } E = 0 \text{ and } M \neq 0, \\
    (-1)^S \times 2^{(E-15)} \times (1 + M/2^{10}), & \quad \text{if } 0 < E < 31, \\
    (-1)^S \times \text{INF}, & \quad \text{if } E = 31 \text{ and } M = 0, \text{ or } \\
    \text{NaN}, & \quad \text{if } E = 31 \text{ and } M \neq 0, \\
\end{align*}

where

\begin{align*}
    S &= \text{floor}((N \mod 65536) / 32768), \\
    E &= \text{floor}((N \mod 32768) / 1024), \text{ and} \\
    M &= N \mod 1024.
\end{align*}

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 values, zero, and INF have an associated sign. -0.0 and +0.0 should be 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)

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>16</td>
<td>half-precision floating-point value encoded in an unsigned scalar</td>
</tr>
</tbody>
</table>

Modify Section 2.7, Vertex Specification, p. 19

(Modify the descriptions of the immediate mode functions in this section, including those introduced by extensions.)

```c
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)

<table>
<thead>
<tr>
<th>Command</th>
<th>Sizes</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexPointer</td>
<td>2,3,4</td>
<td>short, int, float, half, double</td>
</tr>
<tr>
<td>NormalPointer</td>
<td>3</td>
<td>byte, short, int, float, half, double</td>
</tr>
<tr>
<td>ColorPointer</td>
<td>3,4</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>IndexPointer</td>
<td>1</td>
<td>ubyte, short, int, float, double</td>
</tr>
<tr>
<td>TexCoordPointer</td>
<td>1,2,3,4</td>
<td>short, int, float, half, double</td>
</tr>
<tr>
<td>EdgeFlagPointer</td>
<td>1</td>
<td>boolean</td>
</tr>
<tr>
<td>FogCoordPointerEXT</td>
<td>1</td>
<td>float, half, double</td>
</tr>
<tr>
<td>SecondaryColorPointerEXT</td>
<td>3</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>VertexWeightPointerEXT</td>
<td>1</td>
<td>float, half</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>c</td>
</tr>
</tbody>
</table>


(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)

<table>
<thead>
<tr>
<th>type Parameter</th>
<th>Corresponding GL Data Type</th>
<th>Special Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF_FLOAT_NV</td>
<td>half</td>
<td>No</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>type Parameter</th>
<th>GL Data Type</th>
<th>Component Conversion Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF_FLOAT_NV</td>
<td>half</td>
<td>c = f</td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>rendering command length</td>
</tr>
<tr>
<td>2</td>
<td>4240</td>
<td>rendering command opcode</td>
</tr>
<tr>
<td>2</td>
<td>FLOAT16</td>
<td>v[0]</td>
</tr>
<tr>
<td>2</td>
<td>FLOAT16</td>
<td>v[1]</td>
</tr>
</tbody>
</table>
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

VertexWeightvNV
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)
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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>enum</td>
<td>data type</td>
<td></td>
</tr>
<tr>
<td>0x1400</td>
<td>i=1</td>
<td>BYTE</td>
<td></td>
</tr>
<tr>
<td>0x1401</td>
<td>i=1</td>
<td>UNSIGNED_BYTE</td>
<td></td>
</tr>
<tr>
<td>0x1402</td>
<td>i=2</td>
<td>SHORT</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x140B</td>
<td>i=2</td>
<td>HALF_FLOAT_NV</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INT32</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ENUM</td>
<td>array type</td>
<td></td>
</tr>
</tbody>
</table>

Modify Appendix A, Pixel Data (p. 148)

(Modify Table A.1, p. 149 -- add new row for HALF_FLOAT_NV data)

<table>
<thead>
<tr>
<th>type</th>
<th>Encoding</th>
<th>Protocol Type</th>
<th>nbytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALF_FLOAT_NV</td>
<td>0x140B</td>
<td>CARD16</td>
<td>2</td>
</tr>
</tbody>
</table>

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

None.
Name

NV_primitive_restart

Name Strings

GL_NV_primitive_restart

Notice

Copyright NVIDIA Corporation, 2002.

IP Status

NVIDIA Proprietary.

Status


Version

NVIDIA Date: August 29, 2002 (version 0.1)

Number

??

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 ElementRef and DrawArrays, or just DrawElements?

RESOLVED: All of them. It applies to ElementRef 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 ElementRef, 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 dereferenced, 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

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Sec</th>
<th>Attrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMITIVE_RESTART_NV</td>
<td>IsEnabled</td>
<td>B</td>
<td>FALSE</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>PRIMITIVE_RESTART_INDEX_NV</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>0</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
</tbody>
</table>
Name

NV_vertex_program2

Name Strings

GL_NV_vertex_program2

Notice

Copyright NVIDIA Corporation, 2000-2002.

IP Status

NVIDIA Proprietary.

Status


Version

Last Modified Date: $Date: 2002/09/09$
NVIDIA Revision: $Revision: #2$

Number

UNASSIGNED

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.

* Branch instructions allow a program to perform computed goto by computing an index (using an address register), looking up an entry in
a branch table, and jumping to the instruction corresponding to that entry.

* 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 (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, SPL, 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 exponentiated 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?

An application could conceivably use a single generic program with a set of specifically tuned subroutines. For example, if a program evaluated the OpenGL lighting model using multiple lights (some local, some infinite), it could provide a small set of lighting routines tuned to specific light types and use the address register and branch table to select the proper routine at run time.

# c[0].xyz holds the light position
# c[0].w holds the light type (0=disabled, 1=infinite, 2=local)
# A0.x holds the offset of the program parameter holding the attributes of the light in question.
# A0.y will be loaded with the light type.
JMPTABLE {
    disabledLight,
    infiniteLight,
    localLight,
};
...
disabledLight:
    RET;                   # do nothing
localLight:
    # perform computations assuming the light is local
    ...
    RET;

infiniteLight:
    # perform computations assuming the light is infinite
    ...
    RET;

    ...
    ARL   A0.y, c[A0.x].w;
    CAL   [A0.y];
    ...

Note that this example can be further optimized by noting that the "disabled" (index 0) case does nothing. The condition code register can be used to avoid the call/return here.

ARLC A0.y, c[A0.x].w;
CAL   [A0.y] (GT.y);

There are certainly better uses of this functionality than the examples above.

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 \( o[\text{CLP}x] \geq 0 \), which matches the definition in the spec.

```c
# R0 holds eye coordinates
# c[0] holds eye-space clip plane coefficients
DP4 o[\text{CLP}0].x, R0, c[0];
```

Note that the clip plane or clip distance volume corresponding to the \( o[\text{CLP}n] \) 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.

```c
# 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[\text{CLP}0].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
# 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 \( \text{counter} > 0 \) function();
   SUBC R15.x, R15.x, c[1].x;  # \text{counter} = \text{counter} - 1;
   BRA startLoop (GT.x);      # if \( \text{counter} > 0 \) 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
# 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 \( \text{counter} > 0 \) function();
   # Note: A1.y can be used for indexing in function().
```
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

\[ c_n(P) \geq 0, \]

where \( c_n(P) \) is the interpolated value of the clip distance \( \text{CLP}_n \) 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.

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 \( \text{CLIP PLANE}_i \) 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 \( \text{CLIP PLANE}_i = \text{CLIP PLANE}_0 + i \).

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

```c
void VertexAttribs{1234}sfdvNV(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

```c
VertexAttribs{1234}sfdvNV(index, n, coords)
```
is the same (assuming no errors) as the command sequence

```c
#define NUM k /* where k is 1, 2, 3, or 4 components */
int i;
for (i=n-1; i>=0; i--) {
    VertexAttrib{NUM}sfdvNV(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

```c
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

```c
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

```c
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

```c
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.

<table>
<thead>
<tr>
<th>Vertex Result Register Name</th>
<th>Description</th>
<th>Component Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPOS</td>
<td>Homogeneous clip space position</td>
<td>(x,y,z,w)</td>
</tr>
<tr>
<td>COL0</td>
<td>Primary color (front-facing)</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>COL1</td>
<td>Secondary color (front-facing)</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>BFC0</td>
<td>Back-facing primary color</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>BFC1</td>
<td>Back-facing secondary color</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>FOGC</td>
<td>Fog coordinate</td>
<td>(f,<em>,</em>,*)</td>
</tr>
<tr>
<td>PSIZ</td>
<td>Point size</td>
<td>(p,<em>,</em>,*)</td>
</tr>
<tr>
<td>TEX0</td>
<td>Texture coordinate set 0</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX1</td>
<td>Texture coordinate set 1</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX2</td>
<td>Texture coordinate set 2</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX3</td>
<td>Texture coordinate set 3</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX4</td>
<td>Texture coordinate set 4</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX5</td>
<td>Texture coordinate set 5</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX6</td>
<td>Texture coordinate set 6</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>TEX7</td>
<td>Texture coordinate set 7</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>CLP0(*)</td>
<td>Clip distance 0</td>
<td>(d,<em>,</em>,*)</td>
</tr>
<tr>
<td>CLP1(*)</td>
<td>Clip distance 1</td>
<td>(d,<em>,</em>,*)</td>
</tr>
<tr>
<td>CLP2(*)</td>
<td>Clip distance 2</td>
<td>(d,<em>,</em>,*)</td>
</tr>
<tr>
<td>CLP3(*)</td>
<td>Clip distance 3</td>
<td>(d,<em>,</em>,*)</td>
</tr>
<tr>
<td>CLP4(*)</td>
<td>Clip distance 4</td>
<td>(d,<em>,</em>,*)</td>
</tr>
<tr>
<td>CLP5(*)</td>
<td>Clip distance 5</td>
<td>(d,<em>,</em>,*)</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>Vertex Attribute</th>
<th>Conventional</th>
<th>Conventional</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Number</td>
<td>Per-vertex Parameter</td>
<td>Per-vertex Parameter Command</td>
<td>Component</td>
</tr>
<tr>
<td>0</td>
<td>vertex position</td>
<td>Vertex</td>
<td>x,y,z,w</td>
</tr>
<tr>
<td>1</td>
<td>vertex weights</td>
<td>VertexWeightEXT</td>
<td>w,0,0,1</td>
</tr>
<tr>
<td>2</td>
<td>normal</td>
<td>Normal</td>
<td>x,y,z,1</td>
</tr>
<tr>
<td>3</td>
<td>primary color</td>
<td>Color</td>
<td>r,g,b,a</td>
</tr>
<tr>
<td>4</td>
<td>secondary color</td>
<td>SecondaryColorEXT</td>
<td>r,g,b,1</td>
</tr>
<tr>
<td>5</td>
<td>fog coordinate</td>
<td>FogCoordEXT</td>
<td>fc,0,0,1</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>texture coord 0</td>
<td>MultiTexCoord(GL_TEXTURE0_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>9</td>
<td>texture coord 1</td>
<td>MultiTexCoord(GL_TEXTURE1_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>10</td>
<td>texture coord 2</td>
<td>MultiTexCoord(GL_TEXTURE2_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>11</td>
<td>texture coord 3</td>
<td>MultiTexCoord(GL_TEXTURE3_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>12</td>
<td>texture coord 4</td>
<td>MultiTexCoord(GL_TEXTURE4_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>13</td>
<td>texture coord 5</td>
<td>MultiTexCoord(GL_TEXTURE5_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>14</td>
<td>texture coord 6</td>
<td>MultiTexCoord(GL_TEXTURE6_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>15</td>
<td>texture coord 7</td>
<td>MultiTexCoord(GL_TEXTURE7_ARB, ...)</td>
<td>s,t,r,q</td>
</tr>
</tbody>
</table>

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> <vp2-branchTable>
                 <programText>

<optionSequence> ::= <option> <optionSequence>
                   | ""
```
<option> ::= "OPTION" <vp1-option> ";" | "OPTION" <vp2-option> ";"

<vp1-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> ::= "ABS" | "ABSC" | "FLR" | "FLRC" | "FRC" | "FRCC" | "LITC" | "MOVC" | "SSG" | "SSGC"

<SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrc>

<SCALARop> ::= "EXP" | "LOG" | "RCP" | "RSQ" | <vp2-SCALARop>
\[<\text{vp2-SCALARop}> ::= \text{"COS"} | \text{"COSC"} | \text{"EX2"} | \text{"EX2C"} | \text{"LG2"} | \text{"LG2C"} | \text{"EXPC"} | \text{"LOGC"} | \text{"RCPC"} | \text{"RSQC"}\]

\[<\text{BINop-instruction}> ::= <\text{BINop}> <\text{maskedDstReg}> \text{","} <\text{vectorSrc}> \text{","} <\text{vectorSrc}>\]

\[<\text{BINop}> ::= \text{"ADD"} | \text{"DP3"} | \text{"DP4"} | \text{"DST"} | \text{"MAX"} | \text{"MIN"} | \text{"MUL"} | \text{"SGE"} | \text{"SLT"} | <\text{vp11-BINop}> | <\text{vp2-BINop}>\]

\[<\text{vp11-BINop}> ::= \text{"DPH"}\]

\[<\text{vp2-BINop}> ::= \text{"ADDC"} | \text{"DP3C"} | \text{"DP4C"} | \text{"DPHC"} | \text{"DSTC"} | \text{"MAXC"} | \text{"MINC"} | \text{"MULC"} | \text{"SEQ"} | \text{"SEQC"} | \text{"SFL"} | \text{"SFLC"} | \text{"SGEC"} | \text{"SGTC"} | \text{"SLTC"} | \text{"SLEC"} | \text{"SNE"} | \text{"SNEC"} | \text{"STR"} | \text{"STRC"}\]

\[<\text{TRIop-instruction}> ::= <\text{TRIop}> <\text{maskedDstReg}> \text{","} <\text{vectorSrc}> \text{","} <\text{vectorSrc}> \text{","} <\text{vectorSrc}>\]

\[<\text{TRIop}> ::= \text{"MAD"} | <\text{vp2-TRIop}>\]

\[<\text{vp2-TRIop}> ::= \text{"MADC"}\]

\[<\text{vp2-BRA-instruction}> ::= <\text{vp2-BRANCHop}> <\text{vp2-branchTarget}> <\text{vp2-branchCondition}>\]

\[<\text{vp2-BRANCHop}> ::= \text{"BRA"} | \text{"CAL"}\]
<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>

<vp2-branchTarget> ::= <vp2-absoluteBranch>
                     | <vp2-indexedBranch>

<vp2-absoluteBranch> ::= <vp2-branchLabel>

<vp2-indexedBranch> ::= "]" <scalarAddr> <vp2-indexedBrOffset> "]"

<vp2-indexedBrOffset> ::= "" 
                        | "+" <vp2-indexedBrBase>

<vp2-indexedBrBase> ::= decimal integer from 0 to 15 inclusive

<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 "]"

NVIDIA OpenGl Extension Specifications for CineFX NV_vertex_program2
<vtxResultRegName> ::= "HPOS"
| "COL0"
| "COL1"
| "BFC0"
| "BFC1"
| "FOGC"
| "PSIZ"
| "TEX0"
| "TEX1"
| "TEX2"
| "TEX3"
| "TEX4"
| "TEX5"
| "TEX6"
| "TEX7"

<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"
| "." "z"
| "." "w"
| "." "x" "y"
| "." "y" "z"
| "." "z" "w"
| "." "x" "z"
| "." "x" "y" "z"
| "." "y" "z" "w"
| "." "z" "w" "w"
| "." "z" "w" "w"
| "." "y" "z" "w"

<optionalSign> ::= ""
| "-"
| <vp2-optionalSign>

<vp2-optionalSign> ::= "+"
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.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Register Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;OPOS&quot;</td>
<td>0</td>
<td>object position</td>
</tr>
<tr>
<td>&quot;WGHT&quot;</td>
<td>1</td>
<td>vertex weight</td>
</tr>
<tr>
<td>&quot;NRML&quot;</td>
<td>2</td>
<td>normal</td>
</tr>
<tr>
<td>&quot;COL0&quot;</td>
<td>3</td>
<td>primary color</td>
</tr>
<tr>
<td>&quot;COL1&quot;</td>
<td>4</td>
<td>secondary color</td>
</tr>
<tr>
<td>&quot;FOGC&quot;</td>
<td>5</td>
<td>fog coordinate</td>
</tr>
<tr>
<td>&quot;TEX0&quot;</td>
<td>8</td>
<td>texture coordinate 0</td>
</tr>
<tr>
<td>&quot;TEX1&quot;</td>
<td>9</td>
<td>texture coordinate 1</td>
</tr>
<tr>
<td>&quot;TEX2&quot;</td>
<td>10</td>
<td>texture coordinate 2</td>
</tr>
<tr>
<td>&quot;TEX3&quot;</td>
<td>11</td>
<td>texture coordinate 3</td>
</tr>
<tr>
<td>&quot;TEX4&quot;</td>
<td>12</td>
<td>texture coordinate 4</td>
</tr>
<tr>
<td>&quot;TEX5&quot;</td>
<td>13</td>
<td>texture coordinate 5</td>
</tr>
<tr>
<td>&quot;TEX6&quot;</td>
<td>14</td>
<td>texture coordinate 6</td>
</tr>
<tr>
<td>&quot;TEX7&quot;</td>
<td>15</td>
<td>texture coordinate 7</td>
</tr>
</tbody>
</table>

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, and 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.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inputs</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(*)</td>
<td>v</td>
<td>v</td>
<td>absolute value</td>
</tr>
<tr>
<td>ADD</td>
<td>v,v</td>
<td>v</td>
<td>add</td>
</tr>
<tr>
<td>ARL</td>
<td>v</td>
<td>as</td>
<td>address register load</td>
</tr>
<tr>
<td>DP3</td>
<td>v,v</td>
<td>ssss</td>
<td>3-component dot product</td>
</tr>
<tr>
<td>DP4</td>
<td>v,v</td>
<td>ssss</td>
<td>4-component dot product</td>
</tr>
<tr>
<td>DPH(*)</td>
<td>v,v</td>
<td>ssss</td>
<td>homogeneous dot product</td>
</tr>
<tr>
<td>DST</td>
<td>v,v</td>
<td>v</td>
<td>distance vector</td>
</tr>
<tr>
<td>EXP</td>
<td>s</td>
<td>v</td>
<td>exponential base 2 (approximate)</td>
</tr>
<tr>
<td>LIT</td>
<td>v</td>
<td>v</td>
<td>compute light coefficients</td>
</tr>
<tr>
<td>LOG</td>
<td>s</td>
<td>v</td>
<td>logarithm base 2 (approximate)</td>
</tr>
<tr>
<td>MAD</td>
<td>v,v,v</td>
<td>v</td>
<td>multiply and add</td>
</tr>
<tr>
<td>MAX</td>
<td>v,v</td>
<td>v</td>
<td>maximum</td>
</tr>
<tr>
<td>MIN</td>
<td>v,v</td>
<td>v</td>
<td>minimum</td>
</tr>
<tr>
<td>MOV</td>
<td>v</td>
<td>v</td>
<td>move</td>
</tr>
<tr>
<td>MUL</td>
<td>v,v</td>
<td>v</td>
<td>multiply</td>
</tr>
<tr>
<td>RCC(*)</td>
<td>s</td>
<td>ssss</td>
<td>reciprocal (clamped)</td>
</tr>
<tr>
<td>RCP</td>
<td>s</td>
<td>ssss</td>
<td>reciprocal</td>
</tr>
<tr>
<td>RSQ</td>
<td>s</td>
<td>ssss</td>
<td>reciprocal square root</td>
</tr>
<tr>
<td>SGE</td>
<td>v,v</td>
<td>v</td>
<td>set on greater than or equal</td>
</tr>
<tr>
<td>SLT</td>
<td>v,v</td>
<td>v</td>
<td>set on less than</td>
</tr>
<tr>
<td>SUB(*)</td>
<td>v,v</td>
<td>v</td>
<td>subtract</td>
</tr>
</tbody>
</table>

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. The instructions and their respective input and output parameters are summarized in Table X.4.
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", and "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. "paramRegister" 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

NVIDIA Proprietary
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,

```c
# 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 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.

```cpp
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, or can be found in the branch table, as described below. Instruction labels can be defined or used at any point in the body of a program, and can be used in instructions or branch table entries before being defined in the program string.

VP2 vertex programs can contain a branch table, as specified in the grammar rule <vp2-branchTable>. The branch table is a list of up to 16 instruction labels. The first entry in the branch table is assigned an index of zero, and subsequent labels are assigned sequentially increasing indices. When an branch instruction refers to the branch table (by matching the <vp2-indexedBranch> rule), the contents of an address register are added to a constant offset, and clamped to the range [0,N], where N is the index of the last entry of the branch table, to obtain a branch table index. Vertex program execution will continue with the instruction corresponding to the branch table entry matching the computed index.

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 or the branch table. 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);     # not executed
MOV R0,R1;
label1:
   BRA label2 (LT.wyzw);
MOV R0,R2;              # executed
label2:
```

de 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 an executed branch.

A vertex program will fail to load if an instruction or branch table entry refers to a label that is not defined in the program string.

A vertex program will terminate abnormally if it executes a branch instruction that computes a branch table index that does not identify a valid entry in the branch table, except if the branch instruction is conditional and the condition evaluates to FALSE. 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.

\[
\begin{align*}
\text{tmp} &= \text{VectorLoad}(\text{op0}); \\
\text{result}.x &= \text{abs}(\text{tmp}.x); \\
\text{result}.y &= \text{abs}(\text{tmp}.y); \\
\text{result}.z &= \text{abs}(\text{tmp}.z); \\
\text{result}.w &= \text{abs}(\text{tmp}.w);
\end{align*}
\]

The following special-case rules apply to absolute value operation:

1. \(\text{abs}(\text{NaN}) = \text{NaN}\).
2. \(\text{abs}(-\text{INF}) = \text{abs}(\text{+INF}) = \text{+INF}\).
3. \(\text{abs}(-0.0) = \text{abs}(+0.0) = +0.0\).

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.

```c
    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.

```c
    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.

\[
\text{tmp} = \text{VectorLoad}(\text{op0}); \\
i\text{result}.x = \text{round}(\text{tmp}.x); \\
i\text{result}.y = \text{round}(\text{tmp}.y); \\
i\text{result}.z = \text{round}(\text{tmp}.z); \\
i\text{result}.w = \text{round}(\text{tmp}.w); \\
\text{if}(i\text{result}.x < -512) i\text{result}.x = -512; \\
\text{if}(i\text{result}.x > 511) i\text{result}.x = 511; \\
\text{if}(i\text{result}.y < -512) i\text{result}.y = -512; \\
\text{if}(i\text{result}.y > 511) i\text{result}.y = 511; \\
\text{if}(i\text{result}.z < -512) i\text{result}.z = -512; \\
\text{if}(i\text{result}.z > 511) i\text{result}.z = 511; \\
\text{if}(i\text{result}.w < -512) i\text{result}.w = -512; \\
\text{if}(i\text{result}.w > 511) i\text{result}.w = 511;
\]

The rounding function, \text{round}(x), returns the nearest integer to <x>. If the fractional portion of <x> is 0.5, \text{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 or at a computed offset in the branch table. The following pseudocode describes the operation of the instruction:

\[
\text{if}(\text{TestCC}(\text{cc.c***}) || \text{TestCC}(\text{cc.*c**}) || \text{TestCC}(\text{cc.**c*}) || \text{TestCC}(\text{cc.***c})) \{ \\
\quad \text{if}(\text{branchUsesTable}) \{ \\
\quad \quad \text{index} = \text{ScalarLoad}(\text{addrReg}) + \text{branchTableOffset}; \\
\quad \quad \text{if}(\text{index} < 0) \{ \\
\quad \quad \quad \text{index} = 0; \\
\quad \quad \} \else \text{if}(\text{index} >= \text{branchTableSize}) \{ \\
\quad \quad \quad \text{index} = \text{branchTableSize} - 1; \\
\quad \quad \} \else \{ \\
\quad \quad \quad \text{label} = \text{branchTable}[\text{index}]; \\
\quad \quad \} \else \{ \\
\quad \quad \quad \text{label} = \text{branchLabel}; \\
\quad \quad \} \quad // \text{continue execution at instruction following label} \\
\quad \} \else \{ \\
\quad \quad // \text{do nothing} \\
\quad \} 
\]

In the pseudocode, <branchUsesTable> is true if the instruction matches the <vp2-indexedBranch> grammar rule, <branchTableOffset> is the base offset into the branch table specified by <vp2-indexedBrOffset>,

NVIDIA Proprietary
<branchTableSize> is the number of entries in the branch table, <branchTable> is an array holding the branch table, and <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 or at a computed offset in the branch table. 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:

```plaintext
if (TestCC(cc.c***) || TestCC(cc.*c**) ||
    TestCC(cc.**c*) || TestCC(cc.***c)) {
  if (branchUsesTable) {
    index  = ScalarLoad(addrReg) + branchTableOffset;
    if (index < 0) {
      index = 0;
    } else if (index >= branchTableSize) {
      index = branchTableSize - 1;
    }
    label = branchTable[index];
  } else {
    label = branchLabel;
  }

  if (callStackDepth >= 4) {
    // terminate vertex program
  } else {
    callStack[callStackDepth] = nextInstruction;
    callStackDepth++;
  }

  // continue execution at instruction following label
} else {
  // do nothing
}
```

In the pseudocode, <branchUsesTable> is true if the instruction matches the <vp2-indexedBranch> grammar rule, <branchTableOffset> is the base offset into the branch table specified by <vp2-indexedBrOffset>, <branchTableSize> is the number of entries in the branch table, <branchTable> is an array holding the branch table, <branchLabel> is the label specified in the instruction matching the <vp2-branchLabel> grammar rule, <callStackDepth> is the 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].

\[
\text{tmp} = \text{ScalarLoad(op0)}; \\
\text{result}.x = \text{ApproxCosine(tmp)}; \\
\text{result}.y = \text{ApproxCosine(tmp)}; \\
\text{result}.z = \text{ApproxCosine(tmp)}; \\
\text{result}.w = \text{ApproxCosine(tmp)};
\]

The approximation function ApproxCosine is accurate to at least 22 bits with an angle in the range [0,2*PI].

\[
| \text{ApproxCosine(x)} - \cos(x) | < 1.0 / 2^{22}, \text{if } 0.0 <= x < 2.0 \times \text{PI}.
\]

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.

\[
\text{tmp0} = \text{VectorLoad(op0)}; \\
\text{tmp1} = \text{VectorLoad(op1)}; \\
\text{result}.x = (\text{tmp0}.x \times \text{tmp1}.x) + (\text{tmp0}.y \times \text{tmp1}.y) + (\text{tmp0}.z \times \text{tmp1}.z); \\
\text{result}.y = (\text{tmp0}.x \times \text{tmp1}.x) + (\text{tmp0}.y \times \text{tmp1}.y) + (\text{tmp0}.z \times \text{tmp1}.z); \\
\text{result}.z = (\text{tmp0}.x \times \text{tmp1}.x) + (\text{tmp0}.y \times \text{tmp1}.y) + (\text{tmp0}.z \times \text{tmp1}.z); \\
\text{result}.w = (\text{tmp0}.x \times \text{tmp1}.x) + (\text{tmp0}.y \times \text{tmp1}.y) + (\text{tmp0}.z \times \text{tmp1}.z);
\]
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.

\[
\begin{align*}
tmp0 &= \text{VectorLoad}(\text{op0}); \\
tmp1 &= \text{VectorLoad}(\text{op1}); \\
\text{result}.x &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + (tmp0.w \times \text{tmp1}.w); \\
\text{result}.y &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + (tmp0.w \times \text{tmp1}.w); \\
\text{result}.z &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + (tmp0.w \times \text{tmp1}.w); \\
\text{result}.w &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + (tmp0.w \times \text{tmp1}.w); 
\end{align*}
\]

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.

\[
\begin{align*}
tmp0 &= \text{VectorLoad}(\text{op0}); \\
tmp1 &= \text{VectorLoad}(\text{op1}); \\
\text{result}.x &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + \text{tmp1}.w; \\
\text{result}.y &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + \text{tmp1}.w; \\
\text{result}.z &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + \text{tmp1}.w; \\
\text{result}.w &= (tmp0.x \times \text{tmp1}.x) + (tmp0.y \times \text{tmp1}.y) + \\
&\quad (tmp0.z \times \text{tmp1}.z) + \text{tmp1}.w; 
\end{align*}
\]

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 \([\text{NA}, d^2, d^2, \text{NA}]\) and the second operand should be of the form \([\text{NA}, 1/d, \text{NA}, 1/d]\), where \(\text{NA}\) 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:

```c
 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.

```c
 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:

\[ |\text{Approx2ToX}(x) - 2^x| < \frac{1.0}{2^{22}}, \text{if } 0.0 \leq x < 1.0, \]

and, in general,

\[ |\text{Approx2ToX}(x) - 2^x| < \left(\frac{1.0}{2^{22}}\right) \times (2^{\text{floor}(x)}). \]

The following special-case rules apply to exponential approximation:

1. \(\text{Approx2ToX}(\text{NaN}) = \text{NaN}\).
2. \(\text{Approx2ToX}(-\text{INF}) = +0.0\).
3. \(\text{Approx2ToX}(+\text{INF}) = +\text{INF}\).
4. \(\text{Approx2ToX}(+/0.0) = +1.0\).

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

\[ \text{result.x} \times f(\text{result.y}), \]

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:

\[
\begin{align*}
\text{tmp} &= \text{ScalarLoad}(\text{op0}); \\
\text{result.x} &= 2^{\text{floor}(\text{tmp})}; \\
\text{result.y} &= \text{tmp} - \text{floor}(\text{tmp}); \\
\text{result.z} &= \text{RoughApprox2ToX}(\text{tmp}); \\
\text{result.w} &= 1.0;
\end{align*}
\]

The approximation function is accurate to at least 11 bits:

\[ | \text{RoughApprox2ToX}(x) - 2^x | < 1.0 / 2^{11}, \text{ if } 0.0 \leq x < 1.0, \]

and, in general,

\[ | \text{RoughApprox2ToX}(x) - 2^x | < (1.0 / 2^{11}) \times (2^{\text{floor}(x)}). \]

The following special cases apply to the EXP instruction:

1. \( \text{RoughApprox2ToX}(\text{NaN}) = \text{NaN}. \)
2. \( \text{RoughApprox2ToX}(\text{-INF}) = +0.0. \)
3. \( \text{RoughApprox2ToX}(\text{+INF}) = +INF. \)
4. \( \text{RoughApprox2ToX}(+/0.0) = +1.0. \)

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.

```c
    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.

```c
    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.

```cpp
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:

\[ | \text{ApproxLog2}(x) - \log_2(x) | < 1.0 / 2^{22}. \]

The following special-case rules apply to logarithm approximation:

1. \( \text{ApproxLog2}(\text{NaN}) = \text{NaN} \).
2. \( \text{ApproxLog2}(+\text{INF}) = +\text{INF} \).
3. \( \text{ApproxLog2}(+/-0.0) = -\text{INF} \).
4. \( \text{ApproxLog2}(x) = \text{NaN}, -\text{INF} < x < -0.0. \)
5. \( \text{ApproxLog2}(-\text{INF}) = \text{NaN} \).

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 \((n \cdot VP_{pli}, \text{as in the vertex lighting equations in Section 2.13.1})\). The "y" component of the operand is assumed to hold a specular dot product \((n \cdot h_i)\). 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 \((n \cdot VP_{pli})\). The "z" component of the result vector receives the value that should be multiplied by the specular light/material product \((f_i \cdot (n \cdot h_i)^{s_{rm}})\). 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) = +INF, if -0.0 <= x < +1.0,
   +0.0, if x > +1.0,
8. 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:

\[ |\text{RoughApproxLog2}(x) - \log_2(x) | < \frac{1.0}{2^{11}}. \]

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.

```c
    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.

```c
    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.

```c
    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 \([-2^{+64}, -2^{-64}]\).

\[
\begin{align*}
\text{tmp} &= \text{ScalarLoad}(\text{op0}); \\
\text{result.x} &= \text{ClampApproxReciprocal}(\text{tmp}); \\
\text{result.y} &= \text{ClampApproxReciprocal}(\text{tmp}); \\
\text{result.z} &= \text{ClampApproxReciprocal}(\text{tmp}); \\
\text{result.w} &= \text{ClampApproxReciprocal}(\text{tmp});
\end{align*}
\]

The approximation function is accurate to at least 22 bits:

\[
| \text{ClampApproxReciprocal}(x) - (1/x) | < 1.0 / 2^{22}, \text{ if } 1.0 \leq x < 2.0.
\]

The following special-case rules apply to reciprocation:

1. \(\text{ClampApproxReciprocal}(\text{NaN}) = \text{NaN}\).
2. \(\text{ClampApproxReciprocal}(+\text{INF}) = +2^{-64}\).
3. \(\text{ClampApproxReciprocal}(-\text{INF}) = -2^{-64}\).
4. \(\text{ClampApproxReciprocal}(+0.0) = +2^{64}\).
5. \(\text{ClampApproxReciprocal}(-0.0) = -2^{64}\).
6. \(\text{ClampApproxReciprocal}(x) = +2^{-64}, \text{ if } -2^{64} < x < +\text{INF}\).
7. \(\text{ClampApproxReciprocal}(x) = -2^{-64}, \text{ if } -\text{INF} < x < -2^{-64}\).
8. \(\text{ClampApproxReciprocal}(x) = +2^{64}, \text{ if } +0.0 < x < +2^{-64}\).
9. \(\text{ClampApproxReciprocal}(x) = -2^{64}, \text{ if } -2^{-64} < x < -0.0\).

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.

\[
\begin{align*}
\text{tmp} &= \text{ScalarLoad}(\text{op0}); \\
\text{result.x} &= \text{ApproxReciprocal}(\text{tmp}); \\
\text{result.y} &= \text{ApproxReciprocal}(\text{tmp}); \\
\text{result.z} &= \text{ApproxReciprocal}(\text{tmp}); \\
\text{result.w} &= \text{ApproxReciprocal}(\text{tmp});
\end{align*}
\]

The approximation function is accurate to at least 22 bits:

\[
| \text{ApproxReciprocal}(x) - (1/x) | < 1.0 / 2^{22}, \text{ if } 1.0 \leq x < 2.0.
\]

The following special-case rules apply to reciprocation:

1. \(\text{ApproxReciprocal}(\text{NaN}) = \text{NaN}\).
2. \(\text{ApproxReciprocal}(+\text{INF}) = +0.0\).
3. \(\text{ApproxReciprocal}(-\text{INF}) = -0.0\).
4. \(\text{ApproxReciprocal}(+0.0) = +\text{INF}\).
5. \(\text{ApproxReciprocal}(-0.0) = -\text{INF}\).
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:

```c
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.

```c
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. \((<x> == <y>) \text{ and } (<y> == <x>)\) always produce the same result.
2. \((\text{NaN} == <x>)\) is FALSE for all \(<x>\), including \(\text{NaN}\).
3. \((+\text{INF} == +\text{INF})\) and \((-\text{INF} == -\text{INF})\) are TRUE.
4. \((-0.0 == +0.0)\) 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.

\[
\begin{align*}
\text{tmp0} & = \text{VectorLoad(op0)}; \\
\text{tmp1} & = \text{VectorLoad(op1)}; \\
\text{result}.x & = (\text{tmp0}.x \geq \text{tmp1}.x) \ ? \ 1.0 : 0.0; \\
\text{result}.y & = (\text{tmp0}.y \geq \text{tmp1}.y) \ ? \ 1.0 : 0.0; \\
\text{result}.z & = (\text{tmp0}.z \geq \text{tmp1}.z) \ ? \ 1.0 : 0.0; \\
\text{result}.w & = (\text{tmp0}.w \geq \text{tmp1}.w) \ ? \ 1.0 : 0.0; \\
\end{align*}
\]

The following special-case rules apply to SGE:

1. \((\text{NaN} \geq <x>)\) and \((<x> \geq \text{NaN})\) are FALSE for all \(<x>\).
2. \((+\text{INF} \geq +\text{INF})\) and \((-\text{INF} \geq -\text{INF})\) are TRUE.
3. \((-0.0 \geq +0.0)\) and \((+0.0 \geq -0.0)\) 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.

\[
\begin{align*}
\text{tmp0} & = \text{VectorLoad(op0)}; \\
\text{tmp1} & = \text{VectorLoad(op1)}; \\
\text{result}.x & = (\text{tmp0}.x > \text{tmp1}.x) \ ? \ 1.0 : 0.0; \\
\text{result}.y & = (\text{tmp0}.y > \text{tmp1}.y) \ ? \ 1.0 : 0.0; \\
\text{result}.z & = (\text{tmp0}.z > \text{tmp1}.z) \ ? \ 1.0 : 0.0; \\
\text{result}.w & = (\text{tmp0}.w > \text{tmp1}.w) \ ? \ 1.0 : 0.0; \\
\end{align*}
\]

The following special-case rules apply to SGT:

1. \((\text{NaN} > <x>)\) and \((<x> > \text{NaN})\) 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 \([0, 2\pi]\).

```c
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 \([0, 2\pi]\).

\[
|\text{ApproxSine}(x) - \sin(x)| < \frac{1.0}{2^{22}}, \text{ if } 0.0 \leq x < 2.0 \times \pi.
\]

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.

```c
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.

```cpp
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.

```cpp
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.

\[
\begin{align*}
\text{tmp0} &= \text{VectorLoad}(\text{op0}); \\
\text{tmp1} &= \text{VectorLoad}(\text{op1}); \\
\text{result.x} &= \text{tmp0.x} - \text{tmp1.x}; \\
\text{result.y} &= \text{tmp0.y} - \text{tmp1.y}; \\
\text{result.z} &= \text{tmp0.z} - \text{tmp1.z}; \\
\text{result.w} &= \text{tmp0.w} - \text{tmp1.w};
\end{align*}
\]

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

\[
\begin{align*}
\text{void VertexAttribPointerNV}(\text{uint index, int size, enum type, sizei stride, const void *pointer});
\end{align*}
\]

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. \(\text{pointer}\) 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 \text{EnableClientState} command and disabled with the \text{DisableClientState} command. The value of the argument to either command is \text{VERTEX_ATTRIB_ARRAY}_i\_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 \text{VERTEX_ATTRIB_ARRAY}_i\_NV = \text{VERTEX_ATTRIB_ARRAY}_0\_NV + \(i\).

When vertex program mode is enabled, the \text{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 \text{ArrayElement} (\text{DrawArrays}, \text{DrawElements}, and \text{DrawRangeElements}) assume the operation of \text{ArrayElement} described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the \text{ArrayElement} command transfers the \(i\)th 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 \text{VertexAttrib}[\text{size}][\text{type}]v, where \text{size} is one of \([1,2,3,4]\), and \text{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 \text{ArrayElement}.

If the vertex attribute array 0 is enabled, it is as though \text{VertexAttrib}[\text{size}][\text{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 \text{Vertex}[\text{size}][\text{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
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:

```plaintext
<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

```c
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

\[ C = P \times M \]

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, ProgramParameter4fNV, 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.)

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM_PARAMETER_NV</td>
<td>256xR4</td>
<td>GetProgramParameterNV</td>
<td>(0,0,0,0)</td>
<td>program parameters</td>
<td>2.14.1.2</td>
<td>-</td>
</tr>
</tbody>
</table>

(Modify Table X.7. Vertex Program Per-vertex Execution State. "VP1" and "VP2" refer to the VP1 and VP2 execution environments, respectively.)

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>12xR4</td>
<td>-</td>
<td>(0,0,0,0)</td>
<td>VP1 temporary registers</td>
<td>2.14.1.4</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>16xR4</td>
<td>-</td>
<td>(0,0,0,0)</td>
<td>VP2 temporary registers</td>
<td>2.14.1.4</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>15xR4</td>
<td>-</td>
<td>(0,0,0,1)</td>
<td>vertex result registers</td>
<td>2.14.1.4</td>
<td>-</td>
</tr>
<tr>
<td>Z4</td>
<td>-</td>
<td></td>
<td>(0,0,0,0)</td>
<td>VP1 address register</td>
<td>2.14.1.3</td>
<td>-</td>
</tr>
<tr>
<td>2xZ4</td>
<td>-</td>
<td></td>
<td>(0,0,0,0)</td>
<td>VP2 address registers</td>
<td>2.14.1.3</td>
<td>-</td>
</tr>
</tbody>
</table>