OpenGL ES 2.0 Development for the Tegra Platform

Version 100106.01

Contents

Introduction

This documentation provides platform-specific details for developing shader-based OpenGL ES 2.0 (GLES2) applications on the Tegra platform. The information in this documentation is designed to be OS-independent, and represents the capabilities of the Tegra OpenGL ES 2.0 hardware and driver on all supported operating systems.

This document does not detail performance optimizations; it only covers rendering feature- and compatibility-related items. Performance optimizations for Tegra are described in other documentation that may be available from the Tegra Developers' site.

Shader Development

This section provides guidelines and details of shader development on the Tegra platform. It discusses specific shader features and limitations on the Tegra as well as differences between shaders supplied to GLES2 as source code and as precompiled binaries. It also provides information on the shader compiler tools required to generate precompiled binary shaders for the Tegra.

GLSL-ES Shaders

The Tegra supports OpenGL ES 2.0 and its shading language, GLSL-ES. Basically a subset of desktop GLSL, GLSL-ES removes all of the fixed-function language constructs, and also removes language constructs for GL features that are not a part of OpenGL ES 2.0 core, such as 1D and 3D textures.

General GLSL-ES features and uses are outside the scope of this document. Developers should refer directly to the GLSL specification, which is currently downloadable from the Khronos group website:

http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.14.pdf

Obviously, developers can also refer to applications and shaders provided online and in NVIDIA's Tegra Applications SDK for basic examples of GLSL-ES.

Source versus Binary Shaders

As a mobile platform, the Tegra GLES2 drivers are focused on optimal performance and memory footprint while maximizing the feature set. The driver supports both source code shaders provided to GLES2 at runtime, as well as platform-specific binary shaders compiled offline specifically for Tegra. Applications may choose one or the other path (or even a mixture of both) as desired. Precompiled shaders provide the highest performance at runtime, as they can be used directly by the hardware. However, there are limitations and hints that must be provided when using precompiled shaders that will be detailed in other sections of this document. Source code shaders, on the other hand, are extremely flexible and can even be generated as strings on the fly in application code. However, the driver must compile (and in some cases recompile) these shaders at runtime on the Tegra itself, increasing memory footprint and incurring performance overhead at points when the shaders must be recompiled. This can include recompiling the shader during rendering if a never-before-seen combination of some key rendering states is used. The states that can cause shader recompilation are the same as those listed later in this document as requiring pragmas when used in binary, precompiled shaders.

Both shaders provided to GLES2 as source at runtime and precompiled shaders can use the same shader source code in almost all cases. But there are significant cases in which the two may need to differ, which will be detailed in other sections of this documentation.

Precompiling Shaders

The generation of binary, precompiled shaders is dependent upon the host PC OS (generally Windows or Linux). The resulting shaders can be OS-image-specific, although these incompatibilities are avoided whenever possible. See the documentation supplied with the particular Platform Support pack for details on compiling shaders and the resulting binary shaders.

Loading Shaders

Source Code Shaders

Shaders are loaded using the OpenGL ES standard functions: `glShaderSource`, `glCompileShader`, and `glLinkProgram`.

Binary Shaders

Shaders are loaded as binaries on the Tegra platformvia the OpenGL ES standard functions: `glShaderBinary` and `glLinkProgram`. The shader format enumerant is listed in `gl2ext.h`, and is `GL_NVIDIA_PLATFORM_BINARY_NV`.

Binary Shader-Specific Limitations and Requirements

This section includes information on binary, precompiled shader limitations and requirements. In order to optimally support binary shaders on Tegra, the compiled shader binary needs to be specialized for several bits of OpenGL-ES state. In particular, these states are:

Alpha Blending : OpenGL-ES fixed-function alpha blending is performed inside the shader processor on Tegra. To support this, the compiler must accept the blend state and perform the specified blend equation inside the compiled fragment program. Simply setting the alpha blending state in the application C code will not suffice. A pragma in the shader code or a command-line argument to the shader compiler is required.

Texture format : Tegra supports unsigned fixed-point, signed fixed-point, thin floating-point and wide-floating point texture formats. Depending upon which texture format is used for a given sampler, the compiler may need to perform different post-fetch operations, or assume different return value registers. Thus, if you are using floating-point or signed textures ("classic" OpenGL texture types are generally unsigned fixed-point), a pragma in the shader code or a command-line argument to the shader compiler is required. Tegra supports up to 16 simultaneous samplers.

Framebuffer format(s) : When reading from or writing to the framebuffer, the compiler will need to be aware of whether the framebuffer format is fixed- or floating- point. If the framebuffer format is floating-point, a pragma in the shader code or a command-line argument to the shader compiler is required. Tegra supports up to 8 simultaneous color targets (multiple render targets, a.k.a. MRT).

Write Masking : OpenGL-ES allows applications to selectively enable writes to individual color components. To support this, the shader compiler must be given the write mask; the setting it in the C-code GL calls will not suffice. A pragma in the shader code or a command-line argument to the shader compiler is required.

These required pragmas and tags ***DO NOT apply to shaders provided to GLES2 as source code***, as the driver can recompile the shader as needed for the specific fixed-function settings.

Alpha Blending and Binary Shaders

When used with binary shaders, Tegra requires that alpha blending modes be built into the binary shaders. The alpha blending mode as set by the C code calls to OpenGL ES will not set the blending mode on Tegra.

Binary shaders on Tegra do not use the OpenGL ES `glBlend` mode settings to determine blending. The blending mode must be compiled into the shader.

The Tegra blending mode can be set in one of two ways. One method involves an addition to the shader source itself (a pragma), while the other involves an additional compiler flag be passed to the shader compiler.

In the case of the shader source addition, the programmer simply adds a pragma to the top of the fragment shader source code that declares the alpha blending modes to be used. This is convenient to specify, but does have the drawback of "burning" the alpha blending mode into the shader source itself. Precompiled, that shader will always use the given blending mode. But the pragma is ignored in shaders supplied to GLES2 at runtime as source code, with one exception: Any calls to `glValidateShader` will test the pragma against the current blend settings.

The compiler flag option allows for the same source shader code to be compiled into multiple binary shaders with different blending modes. However, it requires a more advanced content pipeline on the part of the app, which can determine the required settings when compiling the shaders.

In order to ensure that source code shaders and precompiled binary shaders match, the blending mode burned into the shader should match the OpenGL ES shading mode set in C code at the time of each draw call. If these modes do not match, calls to `glValidateShader` will fail. Also, the draw call itself may generate a GL error (but the geometry will be drawn based on the shader's burned-in blending mode).

Alpha Blending Pragma/Compile Option Syntax

The syntax of both the pragma and the compile time option are equivalent, and basically follow the OpenGL ES blending mode enumerations.

Pragma Syntax:

```
#pragma profilepragma blendoperation( <drawbufferName>, <rgbEquation>,
            <rgbSrcFactor>, <rgbDstFactor>, [ <alphaEquation>,
            <alphaSrcFactor>, <alphaDstFactor>] )
```

Compile Time Syntax:

```
-profileopt -
blendoperation,<drawbufferName>,<rgbEquation>,<rgbSrcFactor>,
<rgbDstFactor>[,<alphaEquation>,<alphaSrcFactor>,<alphaDstFactor>]
```

`<drawbufferName>` should be the name of a draw buffer that is written in the shader, i.e.: `gl_FragColor` or `gl_FragData[0] - gl_FragData[7]`. The name of the semantics used to describe each output is also accepted, i.e. COLOR or COLOR0-COLOR7 (`COL` and `COL0-COL7` are also accepted).

`<rgbEquation>` and `<alphaEquation>` may be one of:

```
GL_NO_OPERATION
GL_FUNC_ADD
GL_FUNC_SUBTRACT
GL_FUNC_REVERSE_SUBTRACT
```

If `<rgbEquation>` or `<alphaEquation>` is `GL_NO_OPERATION`, no operation is performed.

If `<rgbEquation>` or `<alphaEquation>` is `GL_FUNC_ADD`, alpha blending performs the operation:

```
<rgbEquation>  dst.rgb = <rgbSrcFactor>*src.rgb + <rgbDstFactor>*dst.rgb
<alphaEquation> dst.a  = <alphaSrcFactor>*src.a + <alphaDstFactor>*dst.a
```

If `<rgbEquation>` or `<alphaEquation>` is `GL_FUNC_SUBTRACT`, alpha blending performs the operation:

```
<rgbEquation>  dst.rgb = <rgbSrcFactor>*src.rgb - <rgbDstFactor>*dst.rgb
<alphaEquation> dst.a  = <alphaSrcFactor>*src.a - <alphaDstFactor>*dst.a
```

Finally, if `<rgbEquation>` or `<alphaEquation>` is `GL_FUNC_REVERSE_SUBTRACT`, alpha blending performs the operation:

```
<rgbEquation>  dst.rgb = <rgbSrcFactor>*dst.rgb - <rgbDstFactor>*src.rgb
```

```
  <alphaEquation> dst.a  = <alphaSrcFactor>*dst.a - <alphaDstFactor>*src.a
```

`<rgbDstFactor>` and `<alphaDstFactor>` may be one of:

```
GL_ZERO
GL_ONE
GL_SRC_COLOR
GL_ONE_MINUS_SRC_COLOR
GL_DST_COLOR
GL_ONE_MINUS_DST_COLOR
GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA
GL_CONSTANT_COLOR
GL_ONE_MINUS_CONSTANT_COLOR
GL_CONSTANT_ALPHA
GL_ONE_MINUS_CONSTANT_ALPHA
```

Blending Pragma Examples

Some examples of pragmas and the resulting blend modes include:

"Standard" alpha-based blending:

```
#pragma profilepragma blendoperation(gl_FragColor, GL_FUNC_ADD,
GL_SRC_ALPHA,
    GL_ONE_MINUS_SRC_ALPHA)
```

Color modulation:

```
#pragma profilepragma blendoperation(gl_FragColor, GL_FUNC_ADD,
GL_ZERO,
    GL_SRC_COLOR)
```

Color add:

```
#pragma profilepragma blendoperation(gl_FragColor, GL_FUNC_ADD,
GL_ONE, GL_ONE)
```

Alpha blend the colors, add the alphas:

```
#pragma profilepragma blendoperation( gl_FragColor, GL_FUNC_ADD,
                      GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA,
                      GL_FUNC_ADD, GL_ONE, GL_ONE)
```

Texture Format and Binary Shaders

Pragma Syntax:

```
#pragma profilepragma samplerformat( <samplerName>: <format> )
```

Configuration File Syntax:

```
-profileopts -samplerformat,<samplerName>,<format>
```

`<samplerName>` should be the name of a sampler uniform appearing inside the shader. `<format>` should be one of:

```
fixed_unsigned (default)
fixed_signed
half_float_luminance_alpha
half_float_packed
half_float
```

`fixed_unsigned` corresponds to hardware texture formats of:

```
A8, I8, L8, L8A8, B2G3R3, B5G6R5, B5G5R5A1, B4G4R4A4, A1B5G5R5,
A4B4G4R4,
R8G8B8A8, B8G8R8A8, DXT1, DXT1C, DXT3, DXT5, ETC, LATC1 and LATC2
```

fixed_signed corresponds to software texture formats of:

```
COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC1_EXT (hardware: LATC1) and
COMPRESSED_SIGNED_LUMINANCE_ALPHA_LATC2_EXT (hardware: LATC2)
```

`half_float_luminance_alpha` corresponds to hardware texture formats of:

```
A16_float, L16_float and L16A16_float
```

`half_float_packed` corresponds to hardware texture formats of:

```
R11G11B10_float
```

`half_float` corresponds to hardware texture formats of:

```
R16G16B16A16_float
```

If no format is specified for a sampler, a default format of `fixed_unsigned` is assumed. If a format is specified for a sampler not appearing in the shader, a warning is generated. If `<format>` is not one of the supported formats, a warning is generated, and compilation proceeds as if the format was specified as fixed_unsigned.

Texture Format Pragma Examples

```
#pragma profilepragma samplerformat( bumpmap: fixed_signed )
#pragma profilepragma samplerformat( specularCube: half_float_packed )
// all other samplers default to fixed_unsigned
```

Framebuffer Format and Binary Shaders

Pragma Syntax:

```
#pragma profilepragma drawbufferformat( <drawbufferName>: <format> )
```

Configuration File Syntax:

```
-profileopts -drawbufferformat,<drawbufferName>,<format>
```

<drawbufferName> should be the name of a draw buffer that is written in the shader, i.e.: gl_FragColor or gl_FragData[0] - gl_FragData[7]. The name of the semantics used to describe each output is also accepted, i.e. COLOR or COLOR0- COLOR7 (COL and COL0-COL7 are also accepted). <format> should be one of:

```
fixed_unsigned (default)
half_float_luminance_alpha
half_float_packed
half_float
```

The list of hardware formats mirrors the list of texture formats. fixed_signed is not supported, since Tegra does not support any renderable signed formats.

Framebuffer Pragma Examples

```
#pragma profilepragma drawbufferformat(gl_FragData[3]:
half_float_luminance_alpha)
#pragma profilepragma drawbufferformat(gl_FragData[0]:
half_float_packed
// any other gl_FragData outputs default to fixed_unsigned
```

Write Masking and Binary Shaders

Currently, on Tegra with binary shaders, the write mask set by `glColorMask` is not sufficient to enable write masking; the write mask set via this function must be matched with pragmas in the shader.

Binary shaders on Tegra using writemasking must match the OpenGL ES `glColorMask` mode with a masking mode pragma that must be compiled into the shader. `glValidateProgram` can be used to verify this matching state.

Pragma Syntax:

```
#pragma profilepragma colorwritemask(<drawbuffername>, <red>, <green>,
<blue>, <alpha> )
```

Configuration File Syntax:

```
profileopts -
colorwritemask,<drawbuffername>,<red>,<green>,<blue>,<alpha>
```

`<drawbufferName>` should be the name of a draw buffer that is written in the shader, i.e.: `gl_FragColor` or `gl_FragData[0] – gl_FragData[7]`. The name of the semantics used to describe each output is also accepted, i.e. `COLOR` or `COLOR0`- `COLOR7` (`COL` and `COL0–COL7` are also accepted).

Legal values for `<red>`, `<green>`, `<blue>` and `<alpha>` are:

`false` or `true`

Write-masking behaves identically to the definition in section 4.2.2 of the OpenGL 2.0 Specification (page 214). A value of "true" indicates that the resulting color value (after blending, if necessary) will be stored in the framebuffer. A value of "false" indicates that the original framebuffer color value should be maintained.

If unspecified, the default value for `colorwritemask` is `TRUE, TRUE, TRUE, TRUE`.

EGL Development Notes

EGL Configurations

EGL's method of sorting configurations returned from queries is often counter-intuitive. Applications using `eglChooseConfig` ***must not*** simply select the first returned configuration, nor should they request only one configuration. An example of a common and confusing case is requesting a 565 RGB configuration. Owing to section 3.4 of the EGL spec, `eglChooseConfig` must return the ***deepest*** color buffer first, even if it is deeper than the requested format, and ***even if the requested format could have been matched exactly***. In other words, an implementation that supports 565 and 8888 must return 8888 earlier in the list than 565, even if 565 is requested. The EGL spec notes the following in a footnote to 3.4:

> "This rule places configs with deeper color buffers first in the list returned by eglChooseConfig. Applications may find this counterintuitive, and need to perform additional processing on the list of configs to find one best matching their requirements. For example, specifying RGBA depths of 5651 could return a list whose first config has a depth of 8888."

The side-effects of this can be detrimental in subtle ways. For example, if an application requests 565 but uses the first returned configuration blindly, on Tegra they will receive an 8888 configuration. Rendering to an 8888 back buffer:

1) Can require additional memory bandwidth above rendering to 565
2) Will silently disable dithering (as there is no need for it)

If this 8888 backbuffer is used with a 565 onscreen surface (the default on most OS images), then the 8888 surface will be converted to 565 during the backbuffer-to-frontbuffer copy. This format-converting copy is not dithered, and can result in far worse color banding than would be seen with a correctly dithered 565 backbuffer direct-copied into a 565 frontbuffer. Simply searching the returned configurations for a 565 configuration can increase performance and visual quality!

Applications should always request an array of multiple configurations, and should query important attributes such as red, green and blue depths of each, performing their own manual sorting and filtering of the resulting array. EGL's behavior is defined by the spec; Tegra's driver cannot deviate from the proscribed order.

Tegra OpenGL ES Limits

The following table lists the current limits of various OpenGL ES 2.0 values as returned by the driver. Note that these are intended as guidelines – applications should always query important values from the particular driver being used.

GL_SUBPIXEL_BITS	4
GL_ALIASED_POINT_SIZE_RANGE	(1, 256)
GL_ALIASED_LINE_WIDTH_RANGE	(1, 256)
GL_MAX_ARRAY_TEXTURE_LAYERS_EXT	2048
GL_MAX_CUBE_MAP_TEXTURE_SIZE	2048
GL_MAX_TEXTURE_SIZE	2048
GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT	15
GL_MAX_VIEWPORT_DIMS	(3839, 3839)
GL_NUM_COMPRESSED_TEXTURE_FORMATS	9
GL_NUM_SHADER_BINARY_FORMATS	1
GL_SHADER_BINARY_FORMATS	GL_NVIDIA_PLATFORM_BINARY_NV
GL_MAX_VERTEX_ATTRIBS	16
GL_MAX_VERTEX_UNIFORM_VECTORS	256
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS	0
GL_MAX_VARYING_VECTORS	15
GL_MAX_TEXTURE_IMAGE_UNITS	16
GL_MAX_FRAGMENT_UNIFORM_VECTORS	1024
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS	16
GL_MAX_COLOR_ATTACHMENTS_NV	8
GL_MAX_RENDERBUFFER_SIZE	3839
GL_MAX_DRAW_BUFFERS_ARB	8

Tegra-Supported OpenGL ES 2.0 Extensions

The following are extensions supported by the OpenGL ES 2.0 drivers in most Tegra OS images. However, this list is not universal. Applications should call `glGetString(GL_EXTENSIONS)` to verify the support for key extensions. The first group of extensions are GL- or OES-standard, and the documentation can be found at http://www.opengl.org/ or http://www.khronos.org/ . Notes are made only as needed.

- `GL_ARB_draw_buffers`
- `GL_ARB_half_float_pixel`
- `GL_EXT_packed_float`
- `GL_EXT_texture_array`
- `GL_EXT_texture_compression_latc`
- `GL_EXT_texture_filter_anisotropic`
- `GL_OES_compressed_ETC1_RGB8_texture`
- `GL_OES_EGL_image`
- `GL_OES_fbo_render_mipmap`
- GL_OES_shader_binary
    - (Indicated by the value of GL_NUM_SHADER_BINARY_FORMATS being nonzero). Indicates that the implementation supports precompiled binary shaders. All of the demos use this capability on Tegra. See the nv_shader helper library for details
- `GL_OES_texture_float`
- `GL_OES_vertex_half_float`
- `GL_EXT_texture_compression_dxt1`
    - The implementation supports specifying textures with the GL_COMPRESSED_RGB[A]_S3TC_DXT1_EXT formats. Not exported on Tegra, but supported
- `GL_EXT_texture_compression_s3tc`
    - The implementation supports specifying textures with the `GL_COMPRESSED_RGBA_S3TC_DXT[1,3,5]_EXT` formats.
- `GL_OES_framebuffer_object`
    - (Required extension.) Framebuffer objects are supported. Not exported as an extension string on Tegra, but supported
- `GL_OES_mapbuffer`
    - The implementation supports the `glMapBufferOES` and `glUnmapBufferOES` functions. These are exported directly and do not need to be queried.
- `GL_OES_rgb8_rgba8`
    - The implementation supports `GL_RGBA8_OES` and `GL_RGB8_OES` as FBO color buffer formats.
- `GL_OES_stencil8`

    - Indicates that the implementation can support an 8-bit stencil buffer for render targets. Not exported as an extension string on Tegra, but supported.
- GL_OES_texture_half_float
    - Indicates that the GL_HALF_FLOAT_OES token is accepted by glTex[Sub]Image2D and allows for the creation and use of 16-bit floating-point textures (1 sign bit, 5 exponent bits, 10 mantissa bits). These are supported as texture formats and FBO formats. An example of loading a half-float RGBA texture is:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_HALF_FLOAT_OES, img);
```

The following extensions are NVIDIA-specific; their extension specs (as available) are at the end of this chapter.

- GL_NV_coverage_sample
- GL_NV_depth_nonlinear
- GL_NV_fbo_color_attachments
- GL_NV_read_buffer
- GL_NV_draw_path

NVIDIA-Proprietary Tegra OpenGL ES 2.0 Extension Specs

NV_shader_framebuffer_fetch

```
Name

    NV_shader_framebuffer_fetch

Name Strings

    GL_NV_shader_framebuffer_fetch

Contact

    Gary King, NVIDIA Corporation (gking 'at' nvidia.com)

Notice

    Copyright NVIDIA Corporation, 2005 - 2006

Status

    NVIDIA Proprietary

Version

    Last Modified: 2006/04/28
    NVIDIA Revision: 0.9

Number

    XXXX  Not Yet XXXX

Dependencies

    This extension is written against the OpenGL-ES Shading Language
    1.10 Specification.

Overview

    This extension provides a mechanism whereby a fragment shader
    written in the OpenGL-ES Shading Language (GLSL-ES) may use
    the color values stored in the active draw buffers as
    well-defined input values.

Issues

    1.  How should this functionality be exposed?

    RESOLVED:  Four options were considered for this functionality:

        A)  Defining reads from gl_FragColor and gl_FragData (prior
            to any writes) to result in the existing framebuffer
            values.

        B)  Defining new read-only built-in variables corresponding
            to the existing framebuffer data (e.g., gl_LastFragColor).
```

```
        C)  Defining new built-in functions which return the existing
            framebuffer data.

        D)  Defining a new programmable stage (e.g., a Sample Shader)
            which takes the fragment shader output values and
            the existing framebuffer data as inputs.

    This extension has chosen option B, as it provides the best mix of
    language / API simplicity and programmer flexibility.  Reusing
    the existing built-in variables (option (A)) unnecessarily
    complicates the language, since it requires shader compilers to
    perform flow analysis to determine whether or not framebuffer loads
    are required.  Option (C) requires adding either multiple entry
    points (one for each gl_FragData array entry), or also adding
    keywords to specify which buffer should be read.  Option (D) is
    interesting, but leads to a variety of questions regarding what
    functionality should be included in the new stage (e.g., "should
    texturing be included?")  Therefore, Option B has been chosen
    due to its comparative simplicity and available functionality.

    2. What value should gl_LastFragColor contain when the
       ARB_draw_buffers extension is in use?

    RESOLVED: Two options were considered for this functionality:

        A) To mirror ARB_draw_buffers specification the value in
           gl_LastFragColor should be dependent on all active
           draw buffers.

       B) The value on gl_LastFragColor should be the same as the one
          in gl_LastFragData[0].

    This extension has chosen option B, for its simplicity. Option
    A may follow the spirit of the original ARB_draw_buffers
    specification more closely but the value of gl_LastFragData becomes
    somewhat undefined. There is no good way to combine the values
    from multiple draw buffers.


New Functions and Entry Points

    None

New Builtin Variables

   mediump vec4 gl_LastFragData[gl_MaxDrawBuffers]
   mediump vec4 gl_LastFragColor

Changes to the OpenGL-ES Shading Language 1.10 Specification, Chapter 7

    Add after the last sentence of Paragraph 2 of Section 7.2, Fragment
    Shader Special Variables ("These variables may be written to more [...]":

    "...  To access the existing framebuffer values (e.g., to implement
    a complex blend operation inside the shader), fragment shaders
    should use the read-only input values gl_LastFragColor and
    gl_LastFragData."

   Insert new paragraph after Paragraph 7 of Section 7.2 ("If a shader
   statically assigns [...]")
```

```
"Similarly, if a shader using the NV_shader_framebuffer_fetch extension
statically assigns a value to gl_FragColor, it may not read any element
of gl_LastFragData.  If a shader using the NV_shader_framebuffer_fetch
extension statically writes a value to any element of gl_FragData, it
may not read from gl_LastFragColor.  That is, use of the inputs defined
in the NV_shader_framebuffer_fetch extension must mirror the outputs
used in the shader program."
```

NV_coverage_sample

```
Name

    NV_coverage_sample

Name Strings

    GL_NV_coverage_sample
    EGL_NV_coverage_sample

Contact

    Gary King, NVIDIA Corporation (gking 'at' nvidia.com)

Notice

    Copyright NVIDIA Corporation, 2005 - 2007

Status

    NVIDIA Proprietary

Version

    Last Modified Date:  2007/03/20
    NVIDIA Revision: 1.0

Number

    XXXX Not Yet XXXX

Dependencies

    Written based on the wording of the OpenGL 2.0 specification
    and the EXT_framebuffer_object specification.

    Written based on the wording of the EGL 1.2 specification.

    Requires OpenGL-ES 2.0 and OES_framebuffer_object.

    Requires EGL 1.1.

Overview

    Anti-aliasing is a critical component for delivering high-quality
    OpenGL rendering.  Traditionally, OpenGL implementations have
    implemented two anti-aliasing algorithms: edge anti-aliasing
    and multisampling.

    Edge anti-aliasing computes fractional fragment coverage for all
    primitives in a rendered frame, and blends edges of abutting
    and/or overlapping primitives to produce smooth results.  The
    image quality produced by this approach is exceptionally high;
    however, applications are render their geometry perfectly ordered
    back-to-front in order to avoid artifacts such as bleed-through.
    Given the algorithmic complexity and performance cost of performing
    exact geometric sorts, edge anti-aliasing has been used very
    sparingly, and almost never in interactive games.
```

```
    Multisampling, on the other hand, computes and stores subpixel
    (a.k.a. "sample") coverage for rasterized fragments, and replicates
    all post-alpha test operations (e.g., depth test, stencil test,
    alpha blend) for each sample.  After the entire scene is rendered,
    the samples are filtered to compute the final anti-aliased image.
    Because the post-alpha test operations are replicated for each sample,
    all of the bleed-through and ordering artifacts that could occur with
    edge anti-aliasing are avoided completely; however, since each sample
    must be computed and stored separately, anti-aliasing quality is
    limited by framebuffer storage and rendering performance.

    This extension introduces a new anti-aliasing algorithm to OpenGL,
    which dramatically improves multisampling quality without
    adversely affecting multisampling's robustness or significantly
    increasing the storage required, coverage sampling.

    Coverage sampling adds an additional high-precision geometric
    coverage buffer to the framebuffer, which is used to produce
    high-quality filtered results (with or without the presence of a
    multisample buffer).  This coverage information is computed and stored
    during rasterization; since applications may render objects where the
    specified geometry does not correspond to the visual result (examples
    include alpha-testing for "imposters," or extruded volume rendering
    for stencil shadow volumes), coverage buffer updates may be masked
    by the application, analagous to masking the depth buffer.

IP Status

    NVIDIA Proprietary

New Procedures and Functions

    void CoverageMaskNV( boolean mask )
    void CoverageOperationNV( enum operation )

New Tokens


    Accepted by the <attrib_list> parameter of eglChooseConfig
    and eglCreatePbufferSurface, and by the <attribute>
    parameter of eglGetConfigAttrib

    (Note: these enumerants reuse the values GLX_VIDEO_OUT_DEPTH_NV -
    GLX_VIDEO_OUT_COLOR_AND_ALPHA_NV from the GLX_NV_video_out extension)

    EGL_COVERAGE_BUFFERS_NV           0x20C5
    EGL_COVERAGE_SAMPLES_NV           0x20C6

    (Note: these enumerants reuse the values REGISTER_COMBINERS_NV -
    VARIABLE_G_NV of the NV_register_combiners extension)

    Accepted by the <internalformat> parameter of
    RenderbufferStorageEXT and the <format> parameter of ReadPixels

    COVERAGE_COMPONENT_NV             0x8522

    Accepted by the <internalformat> parameter of
    RenderbufferStorageEXT

    COVERAGE_COMPONENT4_NV            0x8523
```

Accepted by the <operation> parameter of CoverageOperationNV

```
COVERAGE_ALL_FRAGMENTS_NV         0x8524
COVERAGE_EDGE_FRAGMENTS_NV        0x8525
COVERAGE_AUTOMATIC_NV             0x8526
```

Accepted by the <attachment> parameter of FramebufferRenderbuffer, and GetFramebufferAttachmentParameteriv

```
COVERAGE_ATTACHMENT_NV            0x8527
```

Accepted by the <buf> parameter of Clear

```
COVERAGE_BUFFER_BIT_NV            0x8000
```

Accepted by the <pname> parameter of GetIntegerv

```
COVERAGE_BUFFERS_NV               0x8528
COVERAGE_SAMPLES_NV               0x8529
```

Changes to Chapter 4 of the OpenGL 2.0 Specification

Insert a new section, after Section 3.2.1 (Multisampling)

"3.2.2 Coverage Sampling

Coverage sampling is a mechanism to antialias all GL primitives: points, lines, polygons, bitmaps and images. The technique is similar to multisampling, with all primitives being sampled multiple times at each pixel, and a sample resolve applied to compute the color values stored in the framebuffer's color buffers. As with multisampling, coverage sampling resolves color sample and coverage values to a single, displayable color each time a pixel is updated, so antialiasing appears to be automatic at the application level. Coverage sampling may be used simultaneously with multisampling; however, this is not required.

An additional buffer, called the coverage buffer, is added to the framebuffer. This buffer stores additional coverage information that may be used to produce higher-quality antialiasing than what is provided by conventional multisampling.

When the framebuffer includes a multisample buffer (3.5.6), the samples contain this coverage information, and the framebuffer does not include the coverage buffer.

If the value of COVERAGE_BUFFERS_NV is one, the rasterization of all primitives is changed, and is referred to as coverage sample rasterization. Otherwise, primitive rasterization is referred to as multisample rasterization (if SAMPLE_BUFFERS is one) or single-sample rasterization (otherwise). The value of COVERAGE_BUFFERS_NV is queried by calling GetIntegerv with <pname> set to COVERAGE_BUFFERS_NV.

During coverage sample rasterization the pixel fragment contents are modified to include COVERAGE_SAMPLES_NV coverage values. The value of COVERAGE_SAMPLES_NV is an implementation-dependent constant, and is queried by calling GetIntegerv with <pname> set to COVERAGE_SAMPLES_NV.

The command

```
CoverageOperationNV(enum operation)
```

may be used to modify the manner in which coverage sampling is performed for all primitives. If <operation> is COVERAGE_ALL_FRAGMENTS_NV, coverage sampling will be performed and the coverage buffer updated for all fragments generated during rasterization. If <operation> is COVERAGE_EDGE_FRAGMENTS_NV, coverage sampling will only be performed for fragments generated at the edge of the primitive (by only updating fragments at the edges of primitives, applications may get better visual results when rendering partially transparent objects). If <operation> is COVERAGE_AUTOMATIC_NV, the GL will automatically select the appropriate coverage operation, dependent on the GL blend mode and the use of gl_LastFragColor / gl_LastFragData in the bound fragment program. If blending is enabled, or gl_LastFragColor / gl_LastFragData appears in the bound fragment program, COVERAGE_AUTOMATIC_NV will behave identically to COVERAGE_EDGE_FRAGMENTS_NV; otherwise, COVERAGE_AUTOMATIC_NV will behave identically to COVERAGE_ALL_FRAGMENTS_NV. The default coverage operation is COVERAGE_AUTOMATIC_NV."

Insert a new section, after Section 3.3.3 (Point Multisample Rasterization)

"3.3.4 Point Coverage Sample Rasterization

If the value of COVERAGE_BUFFERS_NV is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (POINT_SMOOTH) is enabled or disabled. Point rasterization produces fragments using the same algorithm described in section 3.3.3; however, sample points are divided into SAMPLES multisample points and COVERAGE_SAMPLES_NV coverage sample points.

Rasterization for multisample points uses the algorithm described in section 3.3.3. Rasterization for coverage sample points uses implementation-dependent algorithms, ultimately storing the results in the coverage buffer."

Insert a new section, after Section 3.4.4 (Line Multisample Rasterization)

"3.4.5 Line Coverage Sample Rasterization

If the value of COVERAGE_BUFFERS_NV is one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (LINE_SMOOTH) is enabled or disabled. Line rasterization produces fragments using the same algorithm described in section 3.4.4; however, sample points are divided into SAMPLES multisample points and COVERAGE_SAMPLES_NV coverage sample points.

Rasterization for multisample points uses the algorithm described in section 3.4.4. Rasterization for coverage sample points uses implementation-dependent algorithms, ultimately storing results in the coverage buffer."

Insert a new section, after Section 3.5.6 (Polygon Multisample Rasterization)

"3.5.7 Polygon Coverage Sample Rasterization

If the value of COVERAGE_BUFFERS_NV is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (POLYGON_SMOOTH) is enabled or disabled. Polygon rasterization produces fragments using the same algorithm described in section 3.5.6; however, sample points are divided into SAMPLES multisample points and COVERAGE_SAMPLES_NV coverage sample points.

Rasterization for multisample points uses the algorithm described in section 3.5.7. Rasterization for coverage sample points uses implementation-dependent algorithms, ultimately storing results in the coverage buffer."

Insert a new section, after Section 3.6.6 (Pixel Rectangle Multisample Rasterization)

"3.6.7 Pixel Rectangle Coverage Sample Rasterization

If the value of COVERAGE_BUFFERS_NV is one, then pixel rectangles are rasterized using the algorithm described in section 3.6.6."

Modify the first sentence of the second-to-last paragraph of section 3.7 (Bitmaps) to read:

"Bitmap Multisample and Coverage Sample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one; or if the value of COVERAGE_BUFFERS_NV is one, then bitmaps are rasterized using the following algorithm. [...]"

Insert after the first paragraph of Section 4.2.2 (Fine Control of Buffer Updates):

"The coverage buffer can be enabled or disabled for writing coverage sample values using

```
    void CoverageMaskNV( boolean mask );
```

If <mask> is non-zero, the coverage buffer is enabled for writing; otherwise, it is disabled. In the initial state, the coverage buffer is enabled for writing."

And change the text of the last 2 paragraphs of Section 4.2.2 to read:

"The state required for the various masking operations is three integers and two bits: an integer for color indices, an integer for the front and back stencil values, a bit for depth values, and a bit for coverage sample values. A set of four bits is also required indicating which components of an RGBA value should be written. In the initial state, the integer masks are all ones, as are the bits controlling the depth value, coverage sample value and RGBA component writing.

Fine Control of Multisample Buffer Updates

When the value of SAMPLE_BUFFERS is one, ColorMask, DepthMask, CoverageMask, and StencilMask or StencilMaskSeparate control the modification of values in the multisample buffer. [...]"

Change paragraph 2 of Section 4.2.3 (Clearing the Buffers) to read:

"is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, STENCIL_BUFFER_BIT, ACCUM_BUFFER_BIT and COVERAGE_BUFFER_BIT_NV, indicating the buffers currently enabled for color writing, the depth buffer, the stencil buffer, the accumulation buffer and the virtual-coverage buffer, respectively. [...]"

Insert a new paragraph after paragraph 4 of Section 4.3.2 (Reading Pixels) (beginning with "If there is a multisample buffer ..."):

"If the <format> is COVERAGE_COMPONENT_NV, then values are taken from the coverage buffer; again, if there is no coverage buffer, the error INVALID_OPERATION occurs. When <format> is COVERAGE_COMPONENT_NV, <type> must be GL_UNSIGNED_BYTE. Any other value for <type> will generate the error INVALID_ENUM. If there is a multisample buffer, the values are undefined."

Modifications to the OES_framebuffer_object specification

Add a new table at the end of Section 4.4.2.1 (Renderbuffer Objects)

"

Sized internal format	Base Internal Format	C Samples
COVERAGE_COMPONENT4_NV	COVERAGE_COMPONENT_NV	4

Table 1.ooo Desired component resolution for each sized internal format that can be used only with renderbuffers"

Add to the bullet list in Section 4.4.4 (Framebuffer Completeness)

"An internal format is 'coverage-renderable' if it is COVERAGE_COMPONENT_NV or one of the COVERAGE_COMPONENT_NV formats from table 1.ooo. No other formats are coverage-renderable"

Add to the bullet list in Section 4.4.4.1 (Framebuffer Attachment Completeness)

"If <attachment> is COVERAGE_ATTACHMENT_NV, then <image> must have a coverage-renderable internal format."

Add a paragraph at the end of Section 4.4.4.2 (Framebuffer Completeness)

"The values of COVERAGE_BUFFERS_NV and COVERAGE_SAMPLES_NV are derived from the attachments of the currently bound framebuffer object. If the current FRAMEBUFFER_BINDING_OES is not 'framebuffer-complete', then both COVERAGE_BUFFERS_NV and COVERAGE_SAMPLES_NV are undefined. Otherwise, COVERAGE_SAMPLES_NV is equal to the number of coverage samples for the image attached to COVERAGE_ATTACHMENT_NV, or zero if COVERAGE_ATTACHMENT_NV is zero."

Additions to the EGL 1.2 Specification

Add to Table 3.1 (EGLConfig attributes)

Attribute	Type	Notes

EGL_COVERAGE_BUFFERS_NV	integer	number of coverage buffers
EGL_COVERAGE_SAMPLES_NV	integer	number of coverage samples per pixel

Modify the first sentence of the last paragraph of the "Buffer Descriptions and Attributes" subsection of Section 3.4 (Configuration Management), p. 16

"There are no single-sample depth, stencil or coverage buffers for a multisample EGLConfig; the only depth, stencil and coverage buffers are those in the multisample buffer. [...]"

And add the following text at the end of that paragraph:

"The <coverage buffer> is used only by OpenGL ES. It contains primitive coverage information that is used to produce a high-quality anti-aliased image. The format of the coverage buffer is not specified, and its contents are not directly accessible. Only the existence of the coverage buffer, and the number of coverage samples it contains, are exposed by EGL.

EGL_COVERAGE_BUFFERS_NV indicates the number of coverage buffers, which must be zero or one. EGL_COVERAGE_SAMPLES_NV gives the number of coverage samples per pixel; if EGL_COVERAGE_BUFFERS_NV is zero, then EGL_COVERAGE_SAMPLES_NV will also be zero."

Add to Table 3.4 (Default values and match criteria for EGLConfig attributes)

Attribute	Default	Selection Criteria	Sort Order	Sort Priority
EGL_COVERAGE_BUFFERS_NV	0	At Least	Smaller	7
EGL_COVERAGE_SAMPLES_NV	0	At Least	Smaller	8

And renumber existing sort priorities 7-11 as 9-13.

Modify the list in "Sorting of EGLConfigs" (Section 3.4.1, pg 20)

" [...]

5. Smaller EGL_SAMPLE_BUFFERS
6. Smaller EGL_SAMPLES
7. Smaller EGL_COVERAGE_BUFFERS_NV
8. Smaller EGL_COVERAGE_SAMPLES_NV
9. Smaller EGL_DEPTH_SIZE
10. Smaller EGL_STENCIL_SIZE
11. Smaller EGL_ALPHA_MASK_SIZE
12. Special: [...]
13. Smaller EGL_CONFIG_ID [...]"

Usage Examples

(1) Basic Coverage Sample Rasterization

```
glCoverageMaskNV(GL_TRUE);
glDepthMask(GL_TRUE);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

while (1)
```

```
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
                GL_COVERAGE_BUFFER_BIT_NV);
        glDrawElements(...);
        eglSwapBuffers(...);
    }
```

(2) Multi-Pass Rendering Algorithms

```
    while (1)
    {
        glDepthMask(GL_TRUE);
        glCoverageMaskNV(GL_TRUE);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
                GL_COVERAGE_BUFFER_BIT_NV);

        //  first render pass: render Z-only (occlusion surface), with
        //  coverage info.  color writes are disabled

        glCoverageMaskNV(GL_TRUE);
        glDepthMask(GL_TRUE);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
        glDepthFunc(GL_LESS);
        glDrawElements(...);

        //  second render pass: set Z test to Z-equals, disable Z-writes &
        //  coverage writes.  enable color writes.  coverage may be
        //  disabled, because subsequent rendering passes are rendering
        //  identical geometry -- since the final coverage buffer will be
        //  unchanged, we can disable coverage writes as an optimization.

        glCoverageMaskNV(GL_FALSE);
        glDepthMask(GL_FALSE);
        glDepthFunc(GL_EQUAL);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
        glDrawElements(...);

        eglSwapBuffers();
    }
```

(3) Rendering Translucent Objects on Top of Opaque Objects

```
    while (1)
    {
        glDepthMask(GL_TRUE);
        glCoverageMaskNV(GL_TRUE);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
                GL_COVERAGE_BUFFER_BIT_NV);

        // render opaque, Z-buffered geometry with coverage info for the
        // entire primitive.  Overwrite coverage data for all fragments, so
        // that interior fragments do not get resolved incorrectly.

        glDepthFunc(GL_LESS);
        glCoverageOperationNV(GL_COVERAGE_ALL_FRAGMENTS_NV);
        glDrawElements(...);

        // render translucent, Z-buffered geometry.  to ensure that visible
```

```
        // edges of opaque geometry remain anti-aliased, change the
        // coverage operation to just edge fragments.  this will maintain
        // the coverage information underneath the translucent geometry,
        // except at translucent edges.

        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glCoverageOperationNV(GL_COVERAGE_EDGE_FRAGMENTS_NV);
        glEnable(GL_BLEND);
        glDrawElements(...);
        glDisable(GL_BLEND);

        eglSwapBuffers();
    }
```

(4) Rendering Opacity-Mapped Particle Systems & HUDs on Top of Opaque Geometry

```
    while (1)
    {
        glDepthMask(GL_TRUE);
        glCoverageMaskNV(GL_TRUE);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
                GL_COVERAGE_BUFFER_BIT_NV);

        // render opaque, Z-buffered geometry, with coverage info.
        glDepthFunc(GL_LESS);
        glDrawElements(...);

        // render opacity-mapped geometry.  disable Z writes, enable alpha
        // blending. also, disable coverage writes -- the edges of the
        // geometry used for the HUD/particle system have alpha values
        // tapering to zero, so edge coverage is uninteresting, and
        // interior coverage should still refer to the underlying opaque
        // geometry, so that opaque edges visible through the translucent
        // regions remain anti-aliased.

        glCoverageMaskNV(GL_FALSE);
        glDepthMask(GL_FALSE);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glEnable(GL_BLEND);
        glDrawElements(...);
        glDisable(GL_BLEND);

        eglSwapBuffers();
    }
```

Issues

1. Is any specific discussion of coverage sampling resolves required, particularly with respect to application-provided framebuffer objects?

   RESOLVED: No. Because the coverage sampling resolve is an implementation-dependent algorithm, it is always legal behavior for framebuffer read / copy functions to return the value in the selected ReadBuffer as if COVERAGE_BUFFERS_NV was zero. This allows textures attached to the color attachment points of framebuffer objects to behave predictably, even when COVERAGE_BUFFERS_NV is one.

Implementations are encouraged, whenever possible, to use the highest-quality coverage sample resolve supported for calls to eglSwapBuffers, eglCopyBuffers, ReadPixels, CopyPixels and CopyTex{Sub}Image.

2. Should all render buffer & texture types be legal sources for image resolves and coverage attachment?

   RESOLVED: This spec should not place any arbitrary limits on usage; however, there are many reasons why implementers may not wish to support coverage sampling for all surface types.

   Implementations may return FRAMEBUFFER_UNSUPPORTED_OES from CheckFramebufferStatusOES if an object bound to COVERAGE_ATTACHMENT_NV is incompatible with one or more objects bound to DEPTH_ATTACHMENT_OES, STENCIL_ATTACHMENT_OES, or COLOR_ATTACHMENTi_OES.

Revision History

#1.0 - 20.03.2007

Renumbered enumerants. Reformatted to 80 columns.

NV_depth_nonlinear

```
Name

    NV_depth_nonlinear

Name Strings

    GL_NV_depth_nonlinear
    EGL_NV_depth_nonlinear

Contact

    Gary King, NVIDIA Corporation (gking 'at' nvidia.com)

Notice

    Copyright NVIDIA Corporation, 2005 - 2007.

Status

    NVIDIA Proprietary

Version

    Last Modified: 2007/03/20
    NVIDIA Revision: 1.0

Number

    XXXX  Not Yet XXXX

Dependencies

    Written based on the wording of the OpenGL 2.0 Specification and
    EGL 1.2 Specification.

    Requires EGL 1.1.

    Requires OpenGL-ES 1.0.

    OES_framebuffer_object affects the wording of this specification.

Overview

    Due to the perspective divide, conventional integer Z-buffers have
    a hyperbolic distribution of encodings between the near plane
    and the far plane.  This can result in inaccurate depth testing,
    particularly when the number of depth buffer bits is small
    and objects are rendered near the far plane.

    Particularly when the number of depth buffer bits is limited
    (desirable and/or required in low-memory environments), artifacts
    due to this loss of precision may occur even with relatively
    modest far plane-to-near plane ratios (e.g., greater than 100:1).

    Many attempts have been made to provide alternate encodings for
    Z-buffer (or alternate formulations for the stored depth) to
    reduce the artifacts caused by perspective division, such as
    W-buffers, Z-complement buffers and floating-point 1-Z buffers.
```

This extension adds a non-linear encoded Z buffer to OpenGL, which can improve the practically useful range of, e.g. 16-bit depth buffers by up to a factor of 16, greatly improving depth test quality in applications where the ratio between the near and far planes can not be as tightly controlled.

IP Status

NVIDIA Proprietary

New Procedures and Functions

None

New Tokens

(Note: these enumerant values reuse the GLX_VIDEO_OUT_COLOR_NV - GLX_VIDEO_OUT_ALPHA_NV values from the GLX_NV_video_out extension, and the COMBINER3_NV value from the NV_register_combiners extension)

Accepted as a valid sized internal format by all functions accepting sized internal formats with a base format of DEPTH_COMPONENT

```
DEPTH_COMPONENT16_NONLINEAR_NV     0x8553
```

Accepted by the <attrib_list> parameter of eglChooseConfig and eglCreatePbufferSurface, and by the <attribute> parameter of eglGetConfigAttrib

```
EGL_DEPTH_ENCODING_NV              0x20C3
```

Accepted as a value in the <attrib_list> parameter of eglChooseConfig and eglCreatePbufferSurface, and returned in the <value> parameter of eglGetConfigAttrib

```
EGL_DEPTH_ENCODING_NONE_NV         0
EGL_DEPTH_ENCODING_NONLINEAR_NV    0x20C4
```

Changes to the OpenGL 2.0 Specification

Add the following line to table 3.16 (p. 154)

Sized Internal Format	Base Internal Format	D Bits
DEPTH_COMPONENT16_NONLINEAR_NV	DEPTH_COMPONENT	16

Changes to the EGL 1.2 Specification

Add the following line to table 3.1 (p. 14)

Attribute	Type	Notes
EGL_DEPTH_ENCODING_NV	enum	Type of depth-buffer encoding employed

Modify the description of the depth buffer in Section 3.4 (p. 15)

"The depth buffer is used only by OpenGL ES. It contains fragment depth (Z) information generated during rasterization. EGL_DEPTH_SIZE indicates the depth of this buffer in bits, and EGL_DEPTH_ENCODING_NV indicates which alternate depth-buffer encoding (if any) should be used. Legal values for EGL_DEPTH_ENCODING_NV are: EGL_DONT_CARE, EGL_DEPTH_ENCODING_NONE_NV and EGL_DEPTH_ENCODING_NONLINEAR_NV."

Add the following line to table 3.4 (p. 20)

Attribute	Default	Selection Criteria	Sort Order	Sort Priority
EGL_DEPTH_ENCODING_NV	EGL_DONT_CARE	Exact	None	-

Issues

None

Revision History

#1.0 - 20.03.2007

Renumbered enumerants. Reformatted to 80 columns.

NV_draw_path

```
Name

    NV_draw_path

Name Strings

    GL_NV_draw_path

Contact

   Jussi Rasanen, NVIDIA Corporation (jrasanen 'at' nvidia.com)
   Tero Karras, NVIDIA Corporation (tkarras 'at' nvidia.com)

Notice

    Copyright NVIDIA Corporation, 2008

Status

    NVIDIA Proprietary

Version

    Last Modified: 2008/09/16
    NVIDIA Revision: 0.11

Number

    XXXX Not Yet XXXX

Dependencies

    Written based on the wording of the OpenGL 2.0 Specification.

    Requires OpenGL-ES 2.0.

Overview

    This extension adds functionality to render planar Bezier paths. Use cases
    for this extension include acceleration of vector graphics content and
    text rendering.

    A path is defined as a number of segments, representing either straight
    lines, or quadratic or cubic Bezier curves, and can be either filled or
    stroked. Filling corresponds to generating the fragments that lie within
    the interior of the path. Stroking corresponds to generating the fragments
    that lie within a region defined by sweeping a straight-line pen along the
    path.

    Path segments are specified using a command array and a vertex coordinate
    array. When a path is drawn, the command array is processed sequentially.
    There are two categories of commands: ones that cause a path segment to be
    drawn, and ones that affect how path is rendered. Depending on its type,
    each command consumes a variable number of coordinates from the vertex
    coordinate array.

    When filling a path, the order in which the path segments are specified is
    disregarded. The only requirement is that they form zero or more closed
```

contours. If a path contains unclosed contours, its interior and thus the resulting set of fragments is undefined.

The contours of a path may have self-intersecting geometry and overlap with each other. For such paths, the interior is determined using a fill rule. Two fill rules, even-odd and non-zero, are provided. The direction of path segments matters only with the non-zero fill rule, as explained below.

When stroking a path, additional cap and join styles may be applied at the start and end of path segments. Joins are automatically generated between pairs of segments whose corresponding commands are adjacent. Caps are generated based on explicit path commands.

Rendering quality can be controlled per path by specifying the maximum deviation from the ideal curve in window space.

Path rendering pipeline

The path rendering pipeline consists of three stages: transformation and texture coordinate generation, fill and stroke rasterization, and fragment shader. This extension provides a minimal fixed function transformation and texture coordinate generation stage. Programmable vertex shaders are not supported in the context of path rendering.

Path definition

Paths are defined as a combination of an immutable sequence of commands and an associated mutable sequence of vertex coordinates. Each command consumes zero or more vertex coordinates. Path commands are represented as unsigned bytes, whereas the data type of the vertex coordinates is specified separately for each path. Path vertices are always two-dimensional.

The following table lists the available path commands:

Path command	Coords	Notes
MOVE_TO_NV	2	Change the current position
LINE_TO_NV	2	Draw a straight line
QUADRATIC_BEZIER_TO_NV	4	Draw a quadratic Bezier curve
CUBIC_BEZIER_TO_NV	6	Draw a cubic Bezier curve
START_MARKER_NV	0	Record the current position
CLOSE_NV	0	Draw line to the recorded position
STROKE_CAP0_NV	0	Use cap style 0 in adjacent segment
STROKE_CAP1_NV	0	Use cap style 1 in adjacent segment
STROKE_CAP2_NV	0	Use cap style 2 in adjacent segment
STROKE_CAP3_NV	0	Use cap style 3 in adjacent segment

Transformation and texture coordinate generation

Path vertices specified by the vertex coordinate sequence are converted to the homogenous form (x, y, 0, 1) by the transformation stage, and then transformed from model space to clip space using the MATRIX_PATH_TO_CLIP_NV matrix.

To facilitate texture mapping and color gradients, the path vertices are also transformed using each of the MATRIX_PATH_COORD[0-3]_NV matrices. A built-in fragment shader varying array gl_PathCoord of type vec4 receives the corresponding interpolated values. The number of elements in the gl_PathCoord array is 4. Although gradients and texture coordinates can also be implemented using the gl_FragCoord built-in fragment shader variable, it is generally more efficient to use gl_PathCoord, avoiding unnecessary per-fragment matrix multiplications.

MATRIX_PATH_COORD[0-3]_NV and MATRIX_PATH_TO_CLIP_NV can define a homogenous perspective transformation. It is up to the fragment shader to normalize the interpolated coordinates if necessary.

Filling a path

When filling a path, the path segments must form zero or more closed contours. If any of the contours are left open, the resulting set of fragments is undefined. This requirement can be rephrased as follows, depending on the value of FILL_RULE_NV:

NON_ZERO_NV:
Each two-dimensional point has an equal number of path segments starting and ending at it.

EVEN_ODD_NV
Each two-dimensional point has an even number of path segments starting or ending at it.

Note that for any two points to be considered identical, the binary representations of their coordinates must match exactly.

To determine the segments to draw, the path commands are processed sequentially. The following temporary values are maintained during the process:

i: Current vertex coordinate index, initially 0.
cp: Current position, initially (0, 0).
sp: Start position, initially undefined.

Each path command is processed depending on its type as follows. c[i] is used to denote the i'th value in the vertex coordinate array.

MOVE_TO_NV:
Replace the current position.
cp = (c[i+0], c[i+1]), i += 2.

LINE_TO_NV:
Draw a straight line from <cp> to (c[i+0], c[i+1]).
cp = (c[i+0], c[i+1]), i += 2.

QUADRATIC_BEZIER_TO_NV:
Draw a quadratic Bezier curve from <cp> to (c[i+2], c[i+3]) using (c[i+0], c[i+1]) as the control point.
cp = (c[i+2], c[i+3]), i += 4.

CUBIC_BEZIER_TO_NV:
Draw a cubic Bezier curve from <cp> to (c[i+4], c[i+5]) using (c[i+0], c[i+1]) and (c[i+2], c[i+3]) as the control points.
cp = (c[i+4], c[i+5]), i += 6.

START_MARKER_NV:
Replace the start position.

sp = cp.

CLOSE_NV:
If <sp> is undefined, ignore the command.
Otherwise, draw a straight line from <cp> to <sp>.
cp = sp.

STROKE_CAP[0-3]_NV:
Ignore the command.

START_MARKER_NV and CLOSE_NV commands can be used to implement subpath closure found in many vector graphics content formats. For filled paths, an explicit LINE_TO_NV command to the start position will produce the same result as CLOSE_NV. For stroked paths, the difference is that CLOSE_NV will join the closing line segment to the segment following the START_MARKER_NV command.

A fill rule is applied to determine if any given point is contained within the interior of the path. The fill rules are defined by projecting a ray from the point in question to infinity and counting the intersections of the ray and path segments. When looking along the direction of the ray, segments intersecting from left to right increment the counter and right to left segment intersections decrement the counter. If the fill rule is NON_ZERO_NV, the point is within the interior if the final count is non-zero. If the fill rule is EVEN_ODD_NV, the point is within the interior if the final count is odd. The counter must support at least 255 intersections. For more complex paths, the results are undefined.

Curves may be approximated within a limit specified by the PATH_QUALITY_NV parameter. The limit defines the radius of a disc in the window space. Placing the disc at each sampling point, the following rules are used to determine whether to generate the corresponding fragments:

* If the disc is entirely inside the path, generate a fragment.
* If the disc is entirely outside the path, do not generate a fragment.
* If the disc is partially inside the path, whether to generate a fragment is up to the implementation.

Stroking a path

Stroking is performed by sweeping a straight-line pen along each path segment, generating fragments for the sampling points touched by the pen. Additionally, cap and join styles may be applied at the start and end of the segments.

Cap and join styles are selected for each path segment based on the path commands adjacent to the one specifying the segment, and the values of the path parameters. The general rule is that the end of a segment is joined to start of the following segment if they are specified by adjacent path commands. If the start or end of a segment is not joined, a cap is generated instead.

Fill rule is not applied when stroking. Instead, a fragment is generated for each sampling point inside the stroke. Even in case the stroke sweeps over a sampling point multiple times, only one fragment is generated.

Dashing is not supported directly. Instead, this extension allows implementing dashing in user code by generating the corresponding.

Paths are stroked in a coordinate space distinct from the path user space and the clip space. The transformation from the stroke space to the path user space is is controlled by the MATRIX_STROKE_TO_PATH_NV matrix. Both the path user space and the stroke space are two-dimensional, and thus only the 2x2 upper-left components of the matrix are used.

Conceptually, stroking a path consists of five steps. First, the path segments are transformed from the path user space to the stroke space using the inverse of the stroke-to-path matrix. Second, the set of points affected by the stroke is determined in the stroke space, using a straight-line pen that extends one unit into each direction. Third, the set of points is transformed from the stroke space back to the path user space using the stroke-to-path matrix. Fourth, the points are further transformed from the path user space to the clip space using the path-to-clip matrix. Fifth, a fragment is generated for each sampling point contained by the set of transformed points.

The stroke-to-path matrix allows specifying stroke width independent of how the path itself is transformed. Two common scenarios include scaling stroke, where the stroke width varies as the path-to-clip transformation changes, and non-scaling stroke, where the width remains constant in the clip space. For scaling stroke, the stroke-to-path matrix should be specified as an identity matrix multiplied by half of the desired stroke width in the path user space. For non-scaling stroke, it should be specified as the inverse of the path-to-clip matrix multiplied by half of the stroke width in the clip space.

The style of all joins is determined by the STROKE_JOIN_STYLE_NV path parameter, which can be set to one of the following values:

JOIN_MITER_NV:
    Extend the incoming and outgoing stroke outlines until they intersect. If the distance between the intersection point and the center point exceeds STROKE_MITER_LIMIT_NV in the stroke space, apply a bevel join instead.

JOIN_ROUND_NV:
    Connect the incoming and outgoing stroke outlines with a circular arc segment in the stroke space, corresponding to a radius of one unit.

JOIN_BEVEL_NV:
    Connect the incoming and outgoing stroke outlines with a straight line.

JOIN_CLIPPED_MITER_NV:
    Same as JOIN_MITER_NV if STROKE_MITER_LIMIT_NV is not exceeded. Otherwise, clip the extended outlines and connect them with a straight line. The clipping is done against a line whose distance from the center point is equal to STROKE_MITER_LIMIT_NV in the stroke space, and whose orientation is symmetrical with regards to the outlines.

The style of a start cap depends on the previous path command, and the style of an end cap depends on the next command. If the command is not STROKE_CAP[0-3]_NV, the cap style is STROKE_CAP_BUTT_NV. Otherwise, the style is determined by the corresponding STROKE_CAP[0-3]_STYLE_NV path parameter, each opf which can be set to one of the following values:

CAP_BUTT_NV:
    Terminate the segment with a straight line connecting the two outline endpoints.

CAP_ROUND_NV:
Terminate the segment with a semicircle with radius equal to one in the stroke space.

CAP_SQUARE_NV:
Terminate the segment with a rectangle extending one unit along the path tangent.

CAP_TRIANGLE_NV:
Terminate the segment with a triangle with two vertices at the stroke outline endpoints, and a third vertex one unit along the path tangent.

As with fill, the path commands are processed sequentially, maintaining the following temporary values:

i: Current vertex coordinate index, initially 0.
cp: Current position, initially (0, 0).
ct: Current tangent, initially undefined.
cs: Pending cap style, initially butt.
sp: Start position, initially undefined.
st: Start tangent, initially undefined.

Each command is processed as follows, depending on its type:

MOVE_TO_NV:
* If <ct> is defined, draw a butt end cap at <cp>.
* cp = (c[i+0], c[i+1]), ct = undefined, cs = butt, i += 2.

LINE_TO_NV, QUADRATIC_BEZIER_TO_NV, and CUBIC_BEZIER_TO_NV:
* Draw the segment corresponding to the command type.
* If <ct> is undefined, draw a start cap of style <cs> at <cp>.
* If <ct> is defined, draw a join at <cp> between <ct> and the start tangent of the segment.
* If the previous command is START_MARKER_NV, replace <st> with the start tangent of the segment.
* cp = end point, ct = end tangent, i += num.

START_MARKER_NV:
* If <ct> is defined, draw a butt end cap at <cp>.
* ct = undefined, cs = butt, sp = cp, st = undefined.

CLOSE_NV:
* If <sp> is undefined, ignore the command.
* Draw a straight line from <cp> to <sp>.
* If <ct> is undefined, draw a start cap of style <cs> at <cp>.
* If <ct> is defined, draw a join at <cp> between <ct> and the direction of the line.
* If <st> is undefined, draw a butt end cap at <sp>.
* If <st> is defined, draw a join at sp between the direction of the line and <st>.
* cp = sp, ct = undefined, cs = butt.

STROKE_CAP[0-3]_NV:
* If <ct> is defined, draw an end cap of the specified style.
* ct = undefined, cs = style specified by the command.

Path programs

Paths are drawn using a special type of program object called a path program. Path programs function like normal program objects, except that they do not allow a vertex shader to be specified. Path programs are created with a new function CreatePathProgramNV().

Fragment shader

Fragment shader depth values are obtained by transforming the homogenous vertex coordinates (x, y, 0, 1) into the clip space. This enables mixing 3D and path geometry using depth buffering.

Since path programs do not support vertex shaders, path fragment shaders cannot make use of user-defined varyings. Instead, this extension adds built-in variables gl_PathCoord[0-3] of type vec4 that receive interpolated vertex positions transformed with their respective MATRIX_PATH_COORD[0-3]_NV matrices.

The value of gl_FrontFacing is undefined when rendering paths.

The rest of the pipeline

When rendering paths, stencil functionality and backface culling are not applied. Blending, dithering, depth test, scissor test, polygon offset, and multisampling are applied as with other primitives.

Even though stencil test and operation are unavailable when rendering paths, the original contents of the stencil buffer are retained.

Path buffers

Path buffers facilitate efficient rendering of animated text or other instanced path geometry by making it possible to render multiple path objects with a single draw call. A path buffer contains a list of path object handles and associated translation vectors.

Invariance rules

Changing path parameters, viewport, transformations and clipping parameters may result in a different set of pixels to be rendered.

New Procedures and Functions

```
uint CreatePathNV(          enum datatype,
                            sizei numCommands,
                            const ubyte* commands );

void DeletePathNV(          uint path );

void PathVerticesNV(        uint path,
                            const void* vertices );

void PathParameterfNV(      uint path,
                            enum paramType,
                            float param );

void PathParameteriNV(      uint path,
                            enum paramType,
                            int param );

uint CreatePathProgramNV(   void );

void PathMatrixNV(          enum target,
```

```
                                const float* value );

    void DrawPathNV(            uint path,
                                enum mode );

    uint CreatePathbufferNV(    sizei capacity );

    void DeletePathbufferNV(    uint buffer );

    void PathbufferPathNV(      uint buffer,
                                int index,
                                uint path );

    void PathbufferPositionNV(  uint buffer,
                                int index,
                                float x,
                                float y );

    void DrawPathbufferNV(      uint buffer,
                                enum mode );

New Types

    None

New Tokens

    Accepted as the <paramType> parameter of PathParameterNV:

        PATH_QUALITY_NV          0x8ED8
        FILL_RULE_NV             0x8ED9
        STROKE_CAP0_STYLE_NV     0x8EE0
        STROKE_CAP1_STYLE_NV     0x8EE1
        STROKE_CAP2_STYLE_NV     0x8EE2
        STROKE_CAP3_STYLE_NV     0x8EE3
        STROKE_JOIN_STYLE_NV     0x8EE8
        STROKE_MITER_LIMIT_NV    0x8EE9

    Values for the ILL_RULE_NV path parameter:

        EVEN_ODD_NV              0x8EF0
        NON_ZERO_NV              0x8EF1

    Values for the CAP[0-3]_STYLE_NV path parameter:

        CAP_BUTT_NV              0x8EF4
        CAP_ROUND_NV             0x8EF5
        CAP_SQUARE_NV            0x8EF6
        CAP_TRIANGLE_NV          0x8EF7

    Values for the JOIN_STYLE_NV path parameter:

        JOIN_MITER_NV            0x8EFC
        JOIN_ROUND_NV            0x8EFD
        JOIN_BEVEL_NV            0x8EFE
        JOIN_CLIPPED_MITER_NV    0x8EFF

    Accepted as the <target> parameter of PathMatrixNV:

        MATRIX_PATH_TO_CLIP_NV   0x8F04
```

```
        MATRIX_STROKE_TO_PATH_NV 0x8F05
        MATRIX_PATH_COORD0_NV    0x8F08
        MATRIX_PATH_COORD1_NV    0x8F09
        MATRIX_PATH_COORD2_NV    0x8F0A
        MATRIX_PATH_COORD3_NV    0x8F0B

    Accepted as the <mode> parameter of DrawPathbufferNV:

        FILL_PATH_NV             0x8F18
        STROKE_PATH_NV           0x8F19

    Accepted as path commands by CreatePathNV:

        MOVE_TO_NV               0x00
        LINE_TO_NV               0x01
        QUADRATIC_BEZIER_TO_NV   0x02
        CUBIC_BEZIER_TO_NV       0x03
        START_MARKER_NV          0x20
        CLOSE_NV                 0x21
        STROKE_CAP0_NV           0x40
        STROKE_CAP1_NV           0x41
        STROKE_CAP2_NV           0x42
        STROKE_CAP3_NV           0x43

Additions to Chapter 2 of the OpenGL ES Specification

    Add the following error conditions to Chapter 2.8, under DrawArrays:

    "An INVALID_OPERATION error is generated if the current program is a path
    program."

    Add the following error conditions to Chapter 2.8, under DrawElements:

    "An INVALID_OPERATION error is generated if the current program is a path
    program."

    Add the following error conditions to Chapter 2.15, under AttachShader.

    "An INVALID_OPERATION error is generated if the program is a path program
    and the shader is a vertex shader."

    Add the following error conditions to Chapter 2.15, under LinkProgram.

    "Linking a program without a vertex shader will not fail if the program is
    a path program."

Additions to Chapter 3 of the OpenGL ES Specification

    Add a new section between sections 3.5 (Polygons) and 3.6 (Pixel
    Rectangles)

    "3.6 Paths

    This extension adds a new type of primitive, paths, to OpenGL ES'
    primitives - points, lines, polygons, pixel rectangles and bitmaps.

    3.6.1 Path objects

    New path objects are created with the call
```

```
uint CreatePathNV( enum datatype,
                   sizei numCommands,
                   const ubyte* commands );
```

where <datatype> is the vertex data type and it must be one of [UNSIGNED_]BYTE, [UNSIGNED_]SHORT, [UNSIGNED_]INT, FLOAT, FIXED, <numCommands> is the number of commands in the path definition and <commands> is a pointer to an unsigned byte array of commands. Valid commands are listed below. The function returns a non-zero handle to the object or 0 on error.

An INVALID_ENUM error is generated if <datatype> is not one of the values specified above. An INVALID_VALUE error is generated if <numCommands> is less than zero or <numCommands> is greater than zero and <commands> is NULL or the <commands> array contains an invalid command.

TODO note that path objects can be shared between multiple contexts

Path objects are deleted with the command

```
void DeletePathNV( uint path );
```

where <path> is the handle to the path object to delete. If the path is assigned to one or more path buffers, path resources are freed only when the last reference to the path is removed. Path handle is invalid after a call to DeletePathNV. An INVALID_VALUE error is generated if the path object does not exist.

Path vertices are specified with the command

```
void PathVerticesNV( uint path,
                     const void* vertices );
```

where <path> is the handle to the path object and <vertices> is a pointer to an array of vertices. <vertices> must contain at least as many coordinate tuples as is consumed by the associated path commands, otherwise the results are undefined, and may lead to a program crash. If <vertices> contains more coordinates than consumed by the path commands, the rest are silently ignored.

An INVALID_VALUE error is generated if the specified <path> object does not exist or if <vertices> is NULL and the path command requires vertices.

Path parameters are set using the commands

```
void PathParameterfNV( uint path,
                       enum paramType,
                       float param );

void PathParameteriNV( uint path,
                       enum paramType,
                       int param );
```

where <path> is the path object handle, <paramType> is the parameter to set and <param> is the value of the parameter.

The following symbols are accepted as <paramType>:

PATH_QUALITY_NV
    Maximum allowed deviation from the ideal path measured in pixels. The

```
    default value is 0.5 pixels.
FILL_RULE_NV
    Fill rule to use for filling paths. <param> must be either
    EVEN_ODD_NV or NON_ZERO_NV. The default value is EVEN_ODD_NV.
STROKE_CAPn_STYLE_NV
    cap style for the cap index n used when stroking a path. The default
    values are CAP_BUTT_NV.
STROKE_JOIN_STYLE_NV
    join style used when stroking a path. The default value is
    JOIN_MITER_NV.
STROKE_MITER_LIMIT_NV
    miter limit used when stroking a path with miter joins. If a join
    angle exceeds the limit, a miter join is converted into a bevel join.
    The default value is 4.

If paramType is PATH_QUALITY_NV in PathParameteriNV(), param is converted
to a float. If paramType is not PATH_QUALITY_NV in PathParameterfNV(),
param is converted to an int.

An INVALID_VALUE error is generated if the <path> object does not exist.
An INVALID_ENUM error is generated if <paramType> is not any of the above.
An INVALID_VALUE error is generated if <paramType> is PATH_QUALITY_NV and
<param> <= 0 or <paramType> is STROKE_MITER_LIMIT_NV and <param> < 1. An
INVALID_ENUM error is generated if <paramType> is FILL_RULE_NV and param
is not a valid fill rule, <paramType> is STROKE_CAPn_STYLE_NV and <param>
is not a valid cap style, or <paramType> is STROKE_JOIN_STYLE_NV and
<param> is not a valid join style.

Path transformations are set using the call

    void PathMatrixNV( enum target,
                       const float* value );

where <value> must specify a 4x4 matrix. The following values are accepted
as the <target> parameter:

MATRIX_PATH_TO_CLIP_NV
    used for transforming path vertices into clip space when drawing a
    path.
MATRIX_STROKE_TO_PATH_NV
    used for transforming the pen when stroking a path. The vertices are
    subsequently transformed into clip space by MATRIX_PATH_TO_CLIP_NV
    matrix. Only the top-left 2x2 submatrix is used.
MATRIX_PATH_COORDn_NV
    used for generating values for gl_PathCoord[0-3] varyings for fragment
    shader by transforming vertex positions.

The default value for all matrices is the identity matrix.

An INVALID_ENUM error is generated if <target> is not any of the above. An
INVALID_VALUE error is generated if value is NULL.

A path is rendered using the call

    void DrawPathNV( uint path,
                     enum mode );

where <path> is the path to be drawn and <mode> must be either
FILL_PATH_NV or STROKE_PATH_NV.
```

An INVALID_VALUE error is generated if the <path> does not exist. An INVALID_OPERATION error is generated if there is no current program, the current program is not a path program, stencil test is enabled, polygon mode is not GL_FILL, or shade model is not GL_SMOOTH. An INVALID_ENUM is generated if <mode> is not FILL_PATH_NV or STROKE_PATH_NV.

3.6.2 Path programs

A path program is a special type of program object that otherwise behaves like a normal program object, but allows attaching only a fragment shader. Path programs are created using the command

```
uint CreatePathProgramNV( void );
```

The function returns 0 on error (i.e. OUT_OF_MEMORY).

TODO describe LinkProgram error conditions here for clarity?

3.6.3 Path buffers

Path buffers can be used for efficiently rendering multiple instances of a set of path objects with a single draw call. Each path in a path buffer has an associated position vector that allows specifying a model space position offset for that path. Path buffers are created using the function

```
uint CreatePathbufferNV( sizei capacity );
```

where <capacity> is the number of paths in a path buffer. This function returns a non-zero handle to a path buffer object or 0 on error.

An INVALID_VALUE error is generated if capacity < 0.

TODO note that path buffer objects can be shared between multiple contexts

Path buffers are deleted using the call

```
void DeletePathbufferNV( uint buffer );
```

where <buffer> is the handle to the path buffer. Path buffer handle is invalid after a call to DeletePathbufferNV.

An INVALID_VALUE error is generated if the path buffer does not exist.

A path can be added to or removed from a path buffer with the function

```
void PathbufferPathNV( uint buffer,
                       int index,
                       uint path );
```

where <buffer> is the path buffer object handle, <index> is the index of the path buffer slot and <path> is the path object handle. Path buffer paths are mutable and can be re-specified later. Calling PathbufferPathNV with <path> set to zero removes path from the path buffer and leaves the slot corresponding to <index> empty.

An INVALID_VALUE error is generated if the path buffer object <buffer> does not exist, or <index> is less than zero, or greater than or equal to the path buffer capacity.

Path buffer path position vector is specified using the call

```
void PathbufferPositionNV( uint buffer,
                           int index,
                           float x,
                           float y );
```

where <buffer> is the path buffer object handle, <index> is the index of the path buffer slot and <x> and <y> specify the translation. Path buffer path translations are mutable and can be re-specified later.

An INVALID_VALUE error is generated if the path buffer does not exist or index < 0 or index >= path buffer capacity.

All paths in a path buffer are rendered using the command

```
void DrawPathbufferNV( uint buffer,
                       enum mode );
```

An INVALID_VALUE error is generated if <buffer> does not exist. An INVALID_OPERATION error is generated if there is no current program, the current program is not a path program, stencil test is enabled, polygon mode is not GL_FILL, or shade model is not GL_SMOOTH. An INVALID_ENUM is generated if <mode> is not FILL_PATH_NV or STROKE_PATH_NV. The effect of a DrawPathbufferNV call is the same as if DrawPathNV was called for each individual path reference in the path buffer, ordered from the first index to the last."

Add the following to Chapter 3.11 in the section Shader Inputs.

"The value of gl_FrontFacing is undefined if the current program is a path program."

"If the current program is a path program and fragment shader has defined varying variables gl_PathCoord[0-3], they will recieve interpolated vertex coordinates transformed with their respective MATRIX_PATH_COORD[0-3]_NV".

Additions to Chapter 4 of the OpenGL ES Specification

Add the following to the end of Chapter 4.1.5 Stencil Test.

"Stencil functionality is not applied when rendering paths. Path rendering will generate an error if stencil testing is enabled."

Issues

1. Should we use vertex shader or fixed function transform?

   RESOLUTION: Introduce minimal fixed function transform and texgen functionality.

   DISCUSSION: The problem with vertex shader is that it allows changing vertex/control point positions arbitrarily, but path geometry only makes sense if it remains planar. A similar problem occurs with vertex attributes: since attributes of three vertices define interpolation on a plane, the attributes of the rest of the vertices cannot be chosen freely for the interpolation to remain well-defined. In effect, this forces vertex attributes to be derived from vertex positions, which can be described by a matrix multiplication. Furthermore, many implementations are expected to cache the results of flattening and/or triangulation, which is much simpler in case of fixed function transform.

Downsides to using fixed function transformation include the lack of support for morphing. We expect most content to not use morphing, so a viable alternative is modifying the path coordinates. Another downside is that we're reintroducing fixed function transform stage into ES2. In our opinion, the downsides of adopting vertex shader solution outweigh this concern.

2. Should we leverage VBOs/vertex arrays for path coordinates and commands?

   RESOLUTION: Involving VBOs would make the extension much messier.

3. Should we have elliptical arc segments?

   RESOLUTION: Not in this version. It is fairly straightforward to convert arcs into quadratic beziers in an application when content is loaded.

4. Should we have vertex indices?

   RESOLUTION: No. This would unnecessarily complicate the API.

5. Should we use 1 - 4 component vectors as vertex position?

   RESOLUTION: No. It is not clear how non-planar geometry would be rendered.

6. Should we support perspective transformations?

   RESOLUTION: Yes.

7. Should we support stroking?

   RESOLUTION: Yes, all vector graphics formats have stroking. Stroking is a rather involved process both implementation-wise and computationally, so it is a good candidate for being implemented in a driver.

8. Should we allow modification of paths?

   RESOLUTION: Modifying coordinates is needed for animation, especially since we ignore vertex shader. There is no good use case for modifying commands.

9. What happens in case a contour is not closed?

   RESOLUTION: The result is undefined. Alternatively we could have an error check, but that is an extra burden for implementations and extra work in the common case where the path data is ok.

10. Should we support edge antialiasing?

    RESOLUTION: No. ES2 doesn't support edge antialiasing for other primitives either, and this extension is compatible with multisampling.

11. What invariance requirements should we impose?

    SUGGESTION:
    1) Rendering the same path with the same state must generate the same

```
    pixels. This is the normal GL invariance requirement.

 2) If two adjoining paths have a shared curve defined by exactly the
    same vertices (bitwise exact), there can be no gaps. The curve
    direction can change and the invariant must still hold.

 3) UNRESOLVED: If two paths have a shared curve, plus one of the paths
    has extra geometry that intersects the shared curve, does the
    no-gap requirement still have to hold? This is problematic for
    implementations that generate extra vertices at intersection points
    in the process of triangulation/path simplification.
```

```
Revision History

    #0.11 - 2008/09/16 - Tero Karras
    #0.10 - 2008/09/15 - Tero Karras
    #0.9  - 2008/09/12 - Jussi Rasanen
```

NV_system_time

```
Name

    NV_system_time

Name Strings

    EGL_NV_system_time

Contact

    Jason Allen, NVIDIA Corporation (jallen 'at' nvidia.com)

Dependencies

    Requires EGL 1.2

Overview

    This extension exposes an alternative method of querying the system time
    from the driver instead of the operating system.

Issues

    Add 64 bit types?

      Yes, EGL doesn't support any 64 bit types so this extension adds int64
      and uint64 types.

New Types

    typedef long int EGLint64NV;
    typedef unsigned long int EGLuint64NV;

New Procedures and Functions

    uint64NV GetSystemTimeFrequencyNV(void);
    uint64NV GetSystemTimeNV(void);

New Tokens

    None

Description

    The command:

        uint64NV GetSystemTimeFrequencyNV(void);

    returns the frequency of the system timer, in counts per second. The
    frequency will not change while the system is running.

    The command:

        uint64NV GetSystemTimeNV(void);

    returns the current value of the system timer. The system time in seconds
    can be calculated by dividing the returned value by the frequency returned
    by the GetSystemTimeFrequencyNV command.
```

```
    Multiple calls to GetSystemTimeNV may return the same values, applications
    need to be careful to avoid divide by zero errors when using the interval
    calculated from successive GetSystemTimeNV calls.

Usage Example

    EGLuint64NV frequency = eglGetSystemTimeFrequencyNV();

    loop
    {
        EGLuint64NV start = eglGetSystemTimeNV() / frequency;

        // draw

        EGLuint64NV end = eglGetSystemTimeNV() / frequency;

        EGLuint64NV interval = end - start;
        if (interval > 0)
            update_animation(interval);

        eglSwapBuffers(dpy, surface);
    }
```

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com