OpenGL ES 2.0 Development for the Tegra Platform

Version 111128.01

Contents

INTRODUCTION 3

SHADER DEVELOPMENT 4

EGL DEVELOPMENT NOTES 4

TEGRA OPENGL ES LIMITS 6

TEGRA-SUPPORTED OPENGL ES 2.0 EXTENSIONS 7

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.

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

Source versus Binary Shaders

Earlier versions of the Tegra OpenGL ES drivers supported either source code or precompiled binary shaders, as the shader compilers on the platform matured. However, the many benefits of source code shaders on the platform, especially multiple commercial platforms now outweigh the previous benefits and current issues with precompiled shaders. Only source code shaders are supported in the current Tegra drivers. Some drivers may still export the legacy `GL_NV_platform_binary`, however binary shaders should be considered deprecated and their use is not recommended.

Loading Shaders

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

EGL Development Notes

EGL Configurations

Searching the Returned List

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."

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.

32-bit versus 24-bit

Tegra does not support rendering to 24-bit "888" buffers. Applications wishing to use such formats should request a RGBA8888 format to ensure that the OS does not return a SW-emulated configuration. Requesting 24-bit RGB888 with OpenGL ES1.x on Android can lead to a SW-rendered configuration and decreased performance and available features.

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	0 (may export 1, but use of binary shaders is not recommended)
GL_SHADER_BINARY_FORMATS	(may export GL_NV_platform_binary, but use of binary shaders is not recommended)
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

Extension support in EGL and GLES differs on a device-by-device and OS-by-OS basis on Tegra. Applications should always query for extension support on the target platform. Some of the Khronos-general GLES extensions supported on most Tegra platforms include the following. The specifications for these extensions may be found at http://www.khronos.org/registry/gles/

Vertex and Geometry Extensions	
GL_EXT_packed_float	RGB floating-point textures in one 32bpp format
GL_OES_mapbuffer	Low-overhead buffer updates
GL_OES_vertex_half_float	16-bit float vertex support (1 sign bit, 5 exponent bits, 10 mantissa bits)
FBO and Renderbuffer Extensions	
GL_OES_EGL_image	Cross-API images
GL_OES_EGL_image_external	Cross-API images
GL_OES_EGL_sync	Command-stream synchronization
GL_OES_fbo_render_mipmap	Mipmap-level FBO support
GL_OES_rgb8_rgba8	24 and 32bpp FBOs
Texture Format Extensions	
GL_EXT_bgra	Reversed RGBA texture support
GL_EXT_texture_compression_dxt1	DXT1 texture support
GL_EXT_texture_compression_latc	LA compressed textures
GL_EXT_texture_compression_s3tc	DXT3/5 texture support
GL_EXT_texture_format_BGRA8888	Reversed RGBA texture support
GL_OES_compressed_ETC1_RGB8_texture	ETC1 textures
GL_OES_texture_float	Note that 32-bit floating point textures are accepted, but are converted to 16-bit floating point textures internally, and thus use of this extension is not recommended
GL_OES_texture_half_float	16-bit (1 sign bit, 5 exponent bits, 10 mantissa bits)
Texture Feature Extensions	
GL_EXT_texture_filter_anisotropic	Anisotropic mipmap filtering
GL_EXT_texture_array	1D arrays of 2D textures
GL_EXT_unpack_subimage	Limited stride support for texture updates
GL_EXT_occlusion_query_boolean	

The list below highlights some of the NV-specific extensions and the links to their specifications in the Khronos registry. Several of these extensions do not yet appear in the registry; the specs for these extensions are included at the end of the chapter.

EGL_NV_system_time	http://www.khronos.org/registry/egl/extensions/NV/EGL_NV_system_time.txt
GL_NV_coverage_sample	http://www.khronos.org/registry/gles/extensions/NV/EGL_NV_coverage_sample.txt
GL_NV_depth_nonlinear	http://www.khronos.org/registry/gles/extensions/NV/EGL_NV_depth_nonlinear.txt
GL_NV_draw_buffers	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_draw_buffers.txt
GL_NV_draw_path	Documented at the end of the chapter
GL_NV_fbo_color_attachments	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_fbo_color_attachments.txt
GL_NV_platform_binary	Deprecated; use source-code shaders
GL_NV_read_buffer	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_read_buffer.txt
GL_NV_read_depth	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_read_depth_stencil.txt
GL_NV_read_stencil	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_read_depth_stencil.txt
GL_NV_shader_framebuffer_fetch	Documented at the end of the chapter
GL_NV_texture_compression_s3tc_update	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_texture_compression_s3tc_update.txt
GL_NV_texture_npot_2D_mipmap	http://www.khronos.org/registry/gles/extensions/NV/GL_NV_texture_npot_2D_mipmap.txt

Other Extension Specs

The following extensions specs are not yet in the Khronos registry site. They are included at the end of this chapter.

- GL_NV_draw_path
- GL_NV_shader_framebuffer_fetch

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

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_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."
```

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