Getting Started with NV_path_rendering

Mark J. Kilgard
NVIDIA Corporation

May 20, 2011

In this tutorial, you will learn how to GPU-accelerate path rendering within your OpenGL program. This tutorial assumes you are familiar with OpenGL programming in general and how to use OpenGL extensions.

Conventional OpenGL supports rendering images (pixel rectangles and bitmaps) and simple geometric primitives (points, lines, polygons).

NVIDIA's **`NV_path_rendering`** OpenGL extension adds a new rendering paradigm, known as path rendering, for rendering filled and stroked paths. Path rendering approach is not novel—but rather a standard part of most resolution-independent 2D rendering systems such as Adobe's PostScript, PDF, and Flash; Microsoft's TrueType fonts, Direct2D, Office drawings, Silverlight, and XML Paper Specification (XPS); W3C's Scalable Vector Graphics (SVG); Sun's Java 2D; Apple's Quartz 2D; Khronos's OpenVG; Google's Skia; and the Cairo open source project. What *is* novel is the ability to mix path rendering with arbitrary OpenGL 3D rendering and imaging, all with full GPU acceleration.

With the **`NV_path_rendering`** extension, path rendering becomes a first-class rendering mode within the OpenGL graphics system that can be arbitrarily mixed with existing OpenGL rendering and can take advantage of OpenGL's existing mechanisms for texturing, programmable shading, and per-fragment operations.

Unlike geometric primitive rendering, paths are specified on a 2D (non-projective) plane rather than in 3D (projective) space. Even though the path is defined in a 2D plane, every path can be transformed into 3D clip space allowing for 3D view frustum & user-defined clipping, depth offset, and depth testing in the same manner as geometric primitive rendering.

Both geometric primitive rendering and path rendering support rasterization of edges defined by line segments; however, path rendering also allows path segments to be specified by Bezier (cubic or quadratic) curves or partial elliptical arcs. This allows path rendering to define truly curved primitive boundaries unlike the straight edges of line and polygon primitives. Whereas geometric primitive rendering requires convex polygons for well-defined rendering results, path rendering allows (and encourages!) concave and curved outlines to be specified. These paths are even allowed to self-intersect and contain holes.

When filling closed paths, the winding of paths (counterclockwise or clockwise) determines whether pixels are inside or outside of the path.

Paths can also be stroked whereby, conceptually, a fixed-width *brush* is pulled along the path such that the brush remains orthogonal to the gradient of each path segment. Samples within the sweep of this brush are considered inside the stroke of the path.

This extension supports path rendering through a sequence of three operations:

1. *Path specification* is the process of creating and updating a path object consisting of a set of path commands and a corresponding set of 2D vertices.

   Path commands can be specified explicitly from path command and coordinate data, parsed from a string based on standard grammars for representing paths, or specified by a particular glyph of standard font representations. Also new paths can be specified by weighting one or more existing paths so long as all the weighted paths have consistent command sequences.

   Each path object contains zero or more subpaths specified by a sequence of line segments, partial elliptical arcs, and (cubic or quadratic) Bezier curve segments. Each path may contain multiple subpaths that can be closed (forming a contour) or open.

2. *Path stenciling* is the process of updating the stencil buffer based on a path's coverage transformed into window space.

   Path stenciling can determine either the filled or stroked coverage of a path.

   The details of path stenciling are explained within the core of the specification.

   Stenciling a stroked path supports all the standard embellishments for path stroking such as end caps, join styles, miter limits, dashing, and dash caps. These stroking properties specified are parameters of path objects.

3. *Path covering* is the process of emitting simple (convex & planar) geometry that (conservatively) “covers” the path's sample coverage in the stencil buffer. During path covering, stencil testing can be configured to discard fragments not within the actual coverage of the path as determined by prior path stenciling.

   Path covering can cover either the filled or stroked coverage of a path.

To render a path object into the color buffer, an application specifies a path object and then uses a two-step rendering process. First, the path object is stenciled whereby the path object's stroked or filled coverage is rasterized into the stencil buffer. Second, the path object is covered whereby conservative bounding geometry for the path is transformed and rasterized with stencil testing configured to test against the coverage information written to the stencil buffer in the first step so that only fragments covered by the path are written during this second step. Also during this second step written pixels typically have their stencil value reset (so there's no need for clearing the stencil buffer between rendering each path).

Here is an example of specifying and then rendering a five-point star and a heart as a path using Scalable Vector Graphics (SVG) path description syntax:

```
GLuint pathObj = 42;
const char *svgPathString =
  // star
  "M100,180 L40,10 L190,120 L10,120 L160,10 z"
  // heart
  "M300 300 C 100 400,100 200,300 100,500 200,500 400,300 300Z";
glPathStringNV(pathObj, GL_PATH_FORMAT_SVG_NV,
               (GLsizei)strlen(svgPathString), svgPathString);
```

Alternatively applications oriented around the PostScript imaging model can use the PostScript user path syntax instead:

```
const char *psPathString =
  // star
  "100 180 moveto"
  " 40 10 lineto 190 120 lineto 10 120 lineto 160 10 lineto closepath"
  // heart
  " 300 300 moveto"
  " 100 400 100 200 300 100 curveto"
  " 500 200 500 400 300 300 curveto closepath";
glPathStringNV(pathObj, GL_PATH_FORMAT_PS_NV,
               (GLsizei)strlen(psPathString), psPathString);
```

The PostScript path syntax also supports compact and precise binary encoding and includes PostScript-style circular arcs.

Or the path's command and coordinates can be specified explicitly:

```
static const GLubyte pathCommands[10] =
  { GL_MOVE_TO_NV, GL_LINE_TO_NV, GL_LINE_TO_NV, GL_LINE_TO_NV,
    GL_LINE_TO_NV, GL_CLOSE_PATH_NV,
    'M', 'C', 'C', 'Z' };  // character aliases
static const GLshort pathCoords[12][2] =
  { {100, 180}, {40, 10}, {190, 120}, {10, 120}, {160, 10},
    {300,300}, {100,400}, {100,200}, {300,100},
    {500,200}, {500,400}, {300,300} };
glPathCommandsNV(pathObj, 10, pathCommands, 24, GL_SHORT, pathCoords);
```

Before rendering to a window with a stencil buffer, clear the stencil buffer to zero and the color buffer to black:

```
glClearStencil(0);
glClearColor(0,0,0,0);
glStencilMask(~0);
glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Use an orthographic path-to-clip-space transform to map the [0..500]x[0..400] range of the star's path coordinates to the [-1..1] clip space cube:

```
glMatrixLoadIdentityEXT(GL_PROJECTION);
glMatrixOrthoEXT(GL_PROJECTION, 0, 500, 0, 400, -1, 1);
glMatrixLoadIdentityEXT(GL_MODELVIEW);
```

Stencil the path:

```
glStencilFillPathNV(pathObj, GL_COUNT_UP_NV, 0x1F);
```

The **0x1F** mask means the counting uses modulo-32 arithmetic. In principle the star's path is simple enough (having a maximum winding number of 2) that modulo-4 arithmetic would be sufficient so the mask could be **0x3**. Or a mask of all 1's (~0) could be used to count with all available stencil bits.

Now that the coverage of the star and the heart has been rasterized into the stencil buffer, cover the path with a non-zero fill style (indicated by the **GL_NOTEQUAL** stencil function with a zero reference value):

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_NOTEQUAL, 0, 0x1F);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
glColor3f(1,1,0); // yellow
glCoverFillPathNV(pathObj, GL_BOUNDING_BOX_NV);
```

The result is a yellow star (with a filled center) to the left of a yellow heart.

The **GL_ZERO** stencil operation ensures that any covered samples (meaning those with non-zero stencil values) are zeroed when the path cover is rasterized. This allows subsequent paths to be rendered without clearing the stencil buffer again.

A similar two-step rendering process can draw a white outline over the star and heart.

Before rendering, configure the path object with desirable path parameters for stroking. Specify a wider 6.5-unit stroke and the round join style:

```
glPathParameteriNV(pathObj, GL_PATH_JOIN_STYLE_NV, GL_ROUND_NV);
glPathParameterfNV(pathObj, GL_PATH_STROKE_WIDTH_NV, 6.5);
```

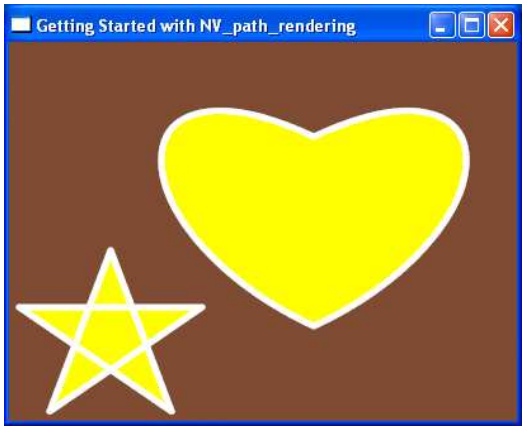
Now stencil the path's stroked coverage into the stencil buffer, setting the stencil to 0x1 for all stencil samples within the transformed path.

```
glStencilStrokePathNV(pathObj, 0x1, ~0);
```

Cover the path's stroked coverage (with a hull this time instead of a bounding box; the choice doesn't really matter here) while stencil testing that writes white to the color buffer and again zero the stencil buffer.

```
glColor3f(1,1,1); // white
glCoverStrokePathNV(pathObj, GL_CONVEX_HULL_NV);
```

The rendering result is:



In this example, constant color shading is used but the application can specify their own arbitrary shading and/or blending operations, whether with Cg compiled to fragment program assembly, GLSL, or fixed-function fragment processing.

More complex path rendering is possible such as clipping one path to another arbitrary path. This is because stencil testing (as well as depth testing, depth bound test, clip planes, and scissoring) can restrict path stenciling.

Now let's render the word “OpenGL” atop the star and heart.

First create a sequence of path objects for the glyphs for the characters in “OpenGL”:

```
GLuint glyphBase = glGenPathsNV(6);
const unsigned char *word = "OpenGL";
const GLsizei wordLen = (GLsizei)strlen(word);
const GLfloat emScale = 2048;  // match TrueType convention
GLuint templatePathObject = ~0;  // Non-existant path object
glPathGlyphsNV(glyphBase,
               GL_SYSTEM_FONT_NAME_NV, "Helvetica", GL_BOLD_BIT_NV,
               wordLen, GL_UNSIGNED_BYTE, word,
               GL_SKIP_MISSING_GLYPH_NV, templatePathObject, emScale);
glPathGlyphsNV(glyphBase,
               GL_SYSTEM_FONT_NAME_NV, "Arial", GL_BOLD_BIT_NV,
               wordLen, GL_UNSIGNED_BYTE, word,
               GL_SKIP_MISSING_GLYPH_NV, templatePathObject, emScale);
glPathGlyphsNV(glyphBase,
               GL_STANDARD_FONT_NAME_NV, "Sans", GL_BOLD_BIT_NV,
               wordLen, GL_UNSIGNED_BYTE, word,
               GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
```

Glyphs are loaded for three different fonts in priority order: Helvetica first, then Arial, and if neither can be loaded, use the standard sans-serif font which is guaranteed to exist. If a prior **`glPathGlyphsNV`** is successful and specifies the path object range, the subsequent **`glPathGlyphsNV`** commands silently avoid re-specifying the already existent path objects.

Priority font loading in this manner is consistent with the font-family convention of HTML where fonts are listed in priority order. Because OpenGL loads the glyphs from the system fonts, OpenGL applications can portably access outline fonts for path rendering. Letting the OpenGL driver manage the font outlines is smart because the driver keeps the path data for the glyphs cached efficiently on the GPU.

Now query the *kerned* glyph separations for the word “OpenGL” and accumulate a sequence of horizontal translations advancing each successive glyph by its kerning distance with the following glyph.

```
GLfloat xtranslate[6+1];  // wordLen+1
xtranslate[0] = 0;  // Initial glyph offset is zero
glGetPathSpacingNV(GL_ACCUM_ADJACENT_PAIRS_NV,
                   wordLen+1, GL_UNSIGNED_BYTE,
                   "\000\001\002\003\004\005\005", // repeat last
                                                   // letter twice
                   glyphBase,
                   1.0f, 1.0f,
                   GL_TRANSLATE_X_NV,
                   xtranslate+1);
```

Next determine the font-wide vertical minimum and maximum for the font face by querying the per-font metrics of any one of the glyphs from the font face.

```
GLfloat yMinMax[2];
glGetPathMetricRangeNV(GL_FONT_Y_MIN_BOUNDS_NV|GL_FONT_Y_MAX_BOUNDS_NV,
                       glyphBase, /*count*/1,
                       2*sizeof(GLfloat),
                       yMinMax);
```

Use an orthographic path-to-clip-space transform to map the word's bounds to the [-1..1] clip space cube:

```
glMatrixLoadIdentityEXT(GL_PROJECTION);
glMatrixOrthoEXT(GL_PROJECTION,
                 0, xtranslate[6], yMinMax[0], yMinMax[1],
                 -1, 1);
glMatrixLoadIdentityEXT(GL_MODELVIEW);
```

Stencil the filled paths of the sequence of glyphs for “OpenGL”, each transformed by the appropriate 2D translations for spacing.

```
glStencilFillPathInstancedNV(6, GL_UNSIGNED_BYTE,
                             "\000\001\002\003\004\005",
                             glyphBase,
                             GL_PATH_FILL_MODE_NV, 0xFF,
                             GL_TRANSLATE_X_NV, xtranslate);
```

Cover the bounding box union of the glyphs with 50% gray.

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_NOTEQUAL, 0, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
glColor3f(0.5,0.5,0.5); // 50% gray
glCoverFillPathInstancedNV(6, GL_UNSIGNED_BYTE,
                           "\000\001\002\003\004\005",
                           glyphBase,
                           GL_BOUNDING_BOX_OF_BOUNDING_BOXES_NV,
                           GL_TRANSLATE_X_NV, xtranslate);
```

Voila, the word “OpenGL” in gray is now stenciled into the framebuffer atop the star and heart:

Instead of solid 50% gray, the cover operation can apply a linear gradient that changes from green (RGB=0,1,0) at the top of the word “OpenGL” to blue (RGB=0,0,1) at the bottom of “OpenGL”:

```
const GLfloat rgbGen[3][3] = {
  { 0,  0, 0 },  // red   = constant zero
  { 0,  1, 0 },  // green = varies with y from bottom (0) to top (1)
  { 0, -1, 1 }   // blue  = varies with y from bottom (1) to top (0)
};
glPathColorGenNV(GL_PRIMARY_COLOR, GL_PATH_OBJECT_BOUNDING_BOX_NV,
                 GL_RGB, &rgbGen[0][0]);
```

With the gradient applied, the result looks like:

Instead of loading just the glyphs for the characters in “OpenGL”, the entire character set could be loaded. This allows the characters of the string to be mapped (offset by the **`glyphBase`**) to path object names. A range of glyphs can be loaded like this:

```
const int numChars = 256;  // ISO/IEC 8859-1 8-bit character range
GLuint fontBase;
glyphBase = glGenPathsNV(numChars);
glPathGlyphRangeNV(fontBase,
                   GL_SYSTEM_FONT_NAME_NV, "Helvetica", GL_BOLD_BIT_NV,
                   0, numChars,
                   GL_SKIP_MISSING_GLYPH_NV, templatePathObject,
                   emScale);
glPathGlyphRangeNV(fontBase,
                   GL_SYSTEM_FONT_NAME_NV, "Arial", GL_BOLD_BIT_NV,
                   0, numChars,
                   GL_SKIP_MISSING_GLYPH_NV, templatePathObject,
                   emScale);
glPathGlyphRangeNV(fontBase,
                   GL_STANDARD_FONT_NAME_NV, "Sans", GL_BOLD_BIT_NV,
                   0, numChars,
                   GL_USE_MISSING_GLYPH_NV, templatePathObject,
                   emScale);
```

Given a range of glyphs loaded as path objects, accumulated *kerned* spacing information can now be queried for the string:

```
GLuint fontBase;
GLfloat kerning[6+1];  // wordLen+1
fontBase = glGenPathsNV(numChars);
kerning[0] = 0;  // Initial glyph offset is zero
glGetPathSpacingNV(GL_ACCUM_ADJACENT_PAIRS_NV,
                   7, GL_UNSIGNED_BYTE, "OpenGLL", // repeat L to get
                                                   // final spacing
                   fontBase,
                   1.0f, 1.0f,
                   GL_TRANSLATE_X_NV,
                   kerning+1);
```

Using the range of glyphs, stenciling and covering the instanced paths for “OpenGL” can be done this way:

```
glStencilFillPathInstancedNV(6, fontBase,
                             GL_UNSIGNED_BYTE, "OpenGL",
                             GL_PATH_FILL_MODE_NV, 0xFF,
                             GL_TRANSLATE_X_NV, kerning);

glCoverFillPathInstancedNV(6, fontBase,
                           GL_UNSIGNED_BYTE, "OpenGL",
                           GL_BOUNDING_BOX_OF_BOUNDING_BOXES_NV,
                           GL_TRANSLATE_X_NV, kerning);
```

Path rendering is resolution-independent so simply resizing the window simply rescales the path rendered content, including the text. Notice that resizing the window automatically resizes the star, heart, and text.

More generally you can apply arbitrary transformations to rotate, scale, translate, and project paths. This code performed prior to the instanced stencil and cover operations to render the "OpenGL" string cause the word to rotate:

```
float center_x = (0 + kerning[6])/2;
float center_y = (yMinMaxFont[0] + yMinMaxFont[1])/2;
glMatrixTranslatefEXT(GL_MODELVIEW, center_x, center_y, 0);
glMatrixRotatefEXT(GL_MODELVIEW, angle, 0, 0, 1);
glMatrixTranslatefEXT(GL_MODELVIEW, -center_x, -center_y, 0);
```

This scene shows the text rotated by an angle of 10 degrees:

Because **`NV_path_rendering`** uses the GPU for your path rendering, the rendering performance is very fast. Please consult the NVIDIA Path Rendering SDK (NVpr SDK) to find the ready-to-compiler-and-run source code for this example as well as many more intricate examples demonstrating GPU-accelerated path rendering.

To resolve questions or issues, send email to nvpr-support@nvidia.com