Programming with NV_path_rendering: An Annex to the SIGGRAPH paper *GPU-accelerated Path Rendering*

Mark J. Kilgard
*NVIDIA Corporation**

Abstract

This annex provides a practical overview of the OpenGL-based programming interface described in our SIGGRAPH Asia 2012 paper *GPU-accelerated Path Rendering*.

Keywords: NV_path_rendering, path rendering, vector graphics, OpenGL, stencil buffer

1 Introduction

Our SIGGRAPH Asia paper *GPU-accelerated Path Rendering* [Kilgard and Bolz 2012] describes a system for accelerating vector graphics on GPUs. NVIDIA has implemented the system and has been shipping the functionality for its GeForce and Quadro GPUs since the summer of 2011. We refer the reader to that paper for the motivation and technical underpinning of the `NV_path_rendering` [Kilgard 2012] OpenGL extension. In particular, that paper explains our "Stencil, then Cover" (StC) approach to filling and stenciling paths.

In this annex to the forementioned paper we explain the programming interface in more detail. The intended audience for this annex is developers evaluating and learning to program `NV_path_rendering`. You should be familiar with the OpenGL [Segal and Akeley 2012] programming interface. Familiarity with path rendering standards such as PostScript or SVG is helpful.

Figure 1 shows how our new path pipeline co-exists with the existing pipelines in OpenGL for pixel and vertex processing. Your application can *mix* traditional OpenGL usage with path rendering.

2 Path Object Specification

Before an application can render paths, it must create a path object corresponding to each path. A path object is a container for the sequence of path commands and corresponding coordinates for the path. Additionally, each path object maintains per-object parameters (see Section 3) and the "baked" GPU-resident state needed to stencil and cover the path object. Like other types of objects in OpenGL, path objects are named by 32-bit unsigned integers. Names of path objects can be generated, tested for existence, and deleted respectively with `glGenPathsNV`, `glIsPathNV`, and `glDeletePathsNV` commands—matching the mechanism OpenGL uses for texture, buffer, and display list objects.

2.1 Path Segment Commands

The path commands supported by `NV_path_rendering` are the union of path commands from all major path rendering standards. We designed `NV_path_rendering` to be a low-level interface upon which all major path rendering standards can be hosted. Eliminating any semantic friction between the path commands of various standards and our interface is important to us.

For example, PostScript [Adobe Systems 1985] provides three commands (`arc`, `arcn`, and `arct`) for specifying circular arc segments. While standards developed after PostScript sought to generalize circular arc segments to an elliptical form, our interface supports circular arc commands to match PostScript's parameterization. So rather than require an application to convert such PostScript circular paths into some alternate form, the circular arc commands are handled with semantics exactly matching PostScript. Likewise, OpenVG [Khronos Group 2008] has a four elliptical arc segment commands, each expecting five coordinate values; whereas SVG has a single elliptical arc segment command with five continuous coordinate values and two Boolean coordinates.

For line segments, the general line segment command takes an (x, y) control point—but horizontal and vertical line segments take a single horizontal or vertical coordinate respectively.

For Bézier curve segments, commands exist for smooth Bézier segments, matching up with the prior command's segment to provide C1 continuity.

Where appropriate, we provide relative and absolute versions of all path commands. With relative commands, path coordinates indicating a position are relative to the end point of the prior path segment.

In addition to eliminating the semantic gap between other path standards and our interface, we note that paths can be represented with fewer path coordinates when the variety of available path commands is broad. Also, editing of the sequence of path commands and coordinates is straightforward when each standard's path command vocabulary is supported directly. Table 1 organizes the supported path commands.

Path objects are specified in four ways:

1. Explicitly, from a sequence of path commands and their corresponding path coordinates.
2. From a string conforming to a standard grammar for specifying a paths. Both PostScript and SVG have standard grammars for paths—and we support both.
3. From a Unicode character point of an outline font. A font can be specified with a system name (such as Helvetica or Arial), an outline font filename, or a built-in font.
4. Derived from one or more existing path objects. The new path may be the result of an arbitrary projective transform of an existing path, or the linear weighting of exiting paths with matching command sequences.

2.2 Explicit Path Specification

The command

```
void glPathCommandsNV(GLuint path,
                      GLsizei numCmds,
                      const GLubyte *cmds,
                      GLsizei numCoords,
                      GLenum coordType,
                      const void *coords);
```

*e-mail: mjk@nvidia.com

Path command	Relative version	Number of scalar coordinates	Character alias	Origin
GL_MOVE_TO_NV	✓	2	M/m	*all*
GL_LINE_TO_NV	✓	2	L/l	*all*
GL_HORIZONTAL_LINE_NV	✓	1	H/h	SVG
GL_VERTICAL_LINE_NV	✓	1	V/v	SVG
GL_QUADRATIC_CURVE_TO_NV	✓	4	Q/q	SVG
GL_CUBIC_CURVE_TO_NV	✓	6	C/c	*all*
GL_SMOOTH_QUADRATIC_CURVE_TO_NV	✓	2	T/t	SVG
GL_SMOOTH_CUBIC_CURVE_TO_NV	✓	4	S/s	SVG
GL_SMALL_CCW_ARC_TO_NV	✓	5	-	OpenVG
GL_SMALL_CW_ARC_TO_NV	✓	5	-	OpenVG
GL_LARGE_CCW_ARC_TO_NV	✓	5	-	OpenVG
GL_LARGE_CW_ARC_TO_NV	✓	5	-	OpenVG
GL_ARC_TO_NV	✓	7	A/a	SVG
GL_CIRCULAR_CCW_ARC_TO_NV	✗	5	-	PostScript
GL_CIRCULAR_CW_ARC_TO_NV	✗	5	-	PostScript
GL_CIRCULAR_TANGENT_ARC_TO_NV	✗	5	-	PostScript
GL_RECT_NV	✗	4	-	PDF
GL_DUP_FIRST_CUBIC_CURVE_TO_NV	✗	4	-	PDF
GL_DUP_LAST_CUBIC_CURVE_TO_NV	✗	4	-	PDF
GL_RESTART_PATH_NV	✗	0	-	PostScript
GL_CLOSE_PATH_NV	✗	0	-	*all*

Table 1: *Path commands supported by* NV_path_rendering. *The character alias column provides an ASCII alias for the absolute/relative version of token. The "all" for origin means the path command is common to all path rendering standards.*

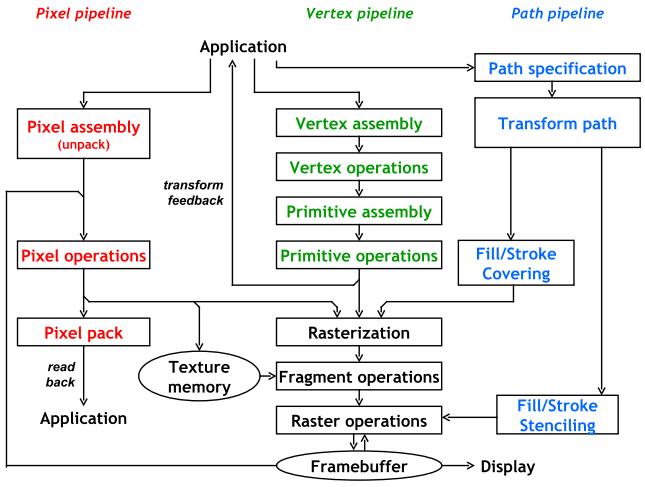

Figure 1: *High-level data flow of OpenGL showing pixel, vertex, and new path pipelines.*

specifies a new path object named *path* where *numCmds* indicates the number of path commands, read from the array *commands*, with which to initialize that path's command sequence. These path commands reference coordinates read sequentially from the *coords* array. The type of the coordinates read from the *coords* array is determined by the *coordType* parameter which must be one of GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, or GL_FLOAT. Coordinates supplied in more compact data types allows paths to be stored more efficiently.

The *numCmds* elements of the *cmds* array must be tokens (or character aliases) from Table 1. The command sequence matches the element order of the *cmds* array. Each command references a number of coordinates specified by the "Number of scalar coordinates" column of Table 1, starting with the first (zero) element of the coords array and advancing by the coordinate count for each command.

The following code fragment creates path object 42 containing the contours of a five-point star and heart:

```
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} };
GLuint pathObj = 42;
glPathCommandsNV(pathObj, 10, pathCommands,
  24, GL_SHORT, pathCoords);
```

The example demonstrates how tokens or character aliases can be used interchangeably to specify a path.

2.3 Grammars for Path Specification

The command

```
glPathStringNV(GLuint path, GLenum format,
               GLsizei length, const void *pathString);
```

specifies a new path object named *path* where format must be either GL_PATH_FORMAT_SVG_NV or GL_PATH_FORMAT_PS_NV, in which case the *length* and *pathString* are interpreted respectively according to SVG's grammar[1] for paths or PostScript's subgrammar for user paths. This code fragment is functionally identical to the prior explicit path specification example but uses an SVG path string:

```
const char *svgPathString =
  // star
  "M100,180 L40,10 L190,120 L10,120 L160,10 z"
  // heart
  "M300 300 "
```

[1]The Backus-Naur Form (BNF) description of the SVG path grammar is found here http://www.w3.org/TR/SVG/paths.html#PathDataBNF

```
  "C100 400,100 200,300 100,500 200,500 400,300 300Z";
glPathStringNV(pathObj, GL_PATH_FORMAT_SVG_NV,
  (GLsizei)strlen(svgPathString), svgPathString);
```

Creating paths from strings has proven very convenient and avoids having each application re-implement standard path grammar parsers.

2.4 Specifying Paths from Glyphs of a Font

Text rendering is a first-class feature in every major path rendering API and standard. Requiring applications to load outlines of glyphs is just too common, not to mention arduous and platform-dependent, so `NV_path_rendering` provides a mechanism for applications to create path objects from glyphs—including contiguous ranges of glyphs indexed by their Unicode character point.

Two commands `glPathGlyphRangeNV` and `glPathGlyphsNV` create a sequence of path objects given a font and a range or sequence of Unicode character points for the font. The font can be specified using a system font name (such as "Arial" or "Helvetica" with a file name for a file in a standard font file format such as TrueType, or a built-in font name (such as "Sans," "Serif," or "Mono") that is guaranteed to be available on every `NV_path_rendering` implementation, regardless of platform.

Once these path objects are populated, these glyph path objects can be stenciled and covered, whether filled or stroked, just like any other path object.

The `glPathGlyphRangeNV` and `glPathGlyphsNV` commands will only create a path object for a given path name if that path object name does not already correspond to an existing path object. This behavior is designed to populate path object ranges with glyph outlines consistent with the `font-family` property of CSS 2.1 [CSS Working Group 2011]. An application can load a sequence of fonts for a given range of path objects repeatedly knowing this will resolve to some supported set of font glyphs eventually.

The `glPathGlyphRangeNV` command has the following prototype:

```
void glPathGlyphRangeNV(GLuint firstPathName,
                        GLenum fontTarget,
                        GLconst void *fontName,
                        GLbitfield fontStyle,
                        GLuint firstGlyph,
                        GLsizei numGlyphs,
                        GLenum handleMissingGlyphs,
                        GLuint pathParameterTemplate,
                        GLfloat emScale);
```

The *emScale* parameter allows fonts of different formats to be loaded with a consistent number of path units per em (a typographic measure of glyph scale). Path coordinates and glyph metrics are scaled to match the specified *emScale*. To ensure all path objects in a range of glyphs have a consistent set of path parameters, the *pathParameterTemplate* path object names a path object from which the new glyph path objects should copy their parameters.

This example shows how a range of path objects for sans serif fonts for the Latin-1 character range can be populated:

```
// Constants
const GLint numChars = 256;   // ISO/IEC 8859-1
                              // 8-bit range
const GLfloat emScale = 2048; // TrueType path
                              // units per em

// Create empty path object for use as parameter template
GLuint pathTemplate = ~0;  // Biggest path name
glPathCommandsNV(pathTemplate,
  0, NULL, 0, GL_FLOAT, NULL);
glPathParameterfNV(pathTemplate,
  GL_PATH_STROKE_WIDTH_NV, emScale*0.1f);
glPathParameteriNV(pathTemplate,
  GL_PATH_JOIN_STYLE_NV, GL_MITER_TRUNCATE_NV);
glPathParameterfNV(pathTemplate,
  GL_PATH_MITER_LIMIT_NV, 1.0);
// Create path object range for Latin-1 character codes
GLuint glyphBase = glGenPathsNV(numChars);
// Typeface names in priority order
struct {
  GLenum fontTarget;
  const char *name;
} font[] = {
  { GL_SYSTEM_FONT_NAME_NV,   "Liberation Sans" },
  { GL_SYSTEM_FONT_NAME_NV,   "Verdana" },
  { GL_SYSTEM_FONT_NAME_NV,   "Arial" },
  // Final standard font provides guaranteed supported
  { GL_STANDARD_FONT_NAME_NV, "Sans" }
};
const int numFonts = sizeof(font)/sizeof(font[0]);
for (int i=0; i< numFonts; i++) {  // For each font
  glPathGlyphRangeNV(glyphBase,
    font[i].fontTarget, font[i].name, GL_BOLD_BIT_NV,
    0, numChars, GL_USE_MISSING_GLYPH_NV,
    pathTemplate, emScale);
}
```

Path objects loaded from glyphs also have associated glyph and font metrics loaded corresponding to their character point. These metrics and spacing information are discussed in Section 5.

2.5 Copied, Weighted, and Transformed Paths

Additional commands for specifying path objects work by generating a new path object from one or more existing path objects. The `glCopyPathNV` command is the simplest and simply copies the state of a named existing path object to another path object name. Path parameters and glyph metrics are copied by `glCopyPathNV`.

The `glInterpolatePathsNV` command takes two (source) path object names and a weighting factor and creates a new (destination) path object that is the linear interpolation based on the weighting factor of the two source paths. The `glWeightPathsNV` command generalizes the `glInterpolatePathsNV` to linear combination of a specified number of source path objects and corresponding weighting factors. All the path objects involved in interpolating or weighting must have identical path command sequences and contain no circular or elliptical arc segment commands. The destination path object's parameters are copied from the first destination path object; glyph metrics are all set invalid (to -1).

The `glTransformPathNV` command take a source path object name and generates a new named destination path object corresponding to the destination path object transformed by an affine linear transform. Path commands such as horizontal or vertical lines or circular arcs may be promoted to a more general path command form as required to perform the transformation. Relative commands are converted to absolute commands, transformed, and then converted back to relative commands.

If the destination path object name refers to an existing path object, that path object is replaced (implicitly deleting the old object) with the new path object. The destination name may be one of the source names.

Assuming the implementation performs a lazy copy of path commands and coordinates, `glCopyPathNV` allows effi-

cient rendering of a path with different stroking parameters. `glInterpolatePathsNV` can help implement Flash's Shape Morph functionality and OpenVG's `vgInterpolatePath` command. `glTransformPathNV` can help implement SVG 1.2's non-scaling stroke functionality and OpenVG's `vgTransformPath`.

3 Path Parameters

Every path object has state in addition to its sequence of path commands and coordinates.

3.1 Settable Parameters

The `glPathParameteriNV`, `glPathParameterfNV`, `glPathParameterivNV`, and `glPathParameterfvNV` commands respectively set path parameters of a specified path object given integer or float data supplied by a scalar parameter or vector array.

Table 2 summarizes the settable parameters. Many parameters deal with embellishments to stroking such as the stroke width, join style, miter limit, end caps, and dash caps.

Name	Type	Initial value
GL_PATH_STROKE_WIDTH_NV	$\Re+$	1.0
GL_PATH_JOIN_STYLE_NV	4-valued	GL_MITER_REVERT_NV
GL_PATH_MITER_LIMIT_NV	$\Re+$	4
GL_PATH_INITIAL_END_CAP_NV	4-valued	GL_FLAT
GL_PATH_TERMINAL_END_CAP_NV	4-valued	GL_FLAT
GL_PATH_INITIAL_DASH_CAP_NV	4-valued	GL_FLAT
GL_PATH_TERMINAL_DASH_CAP_NV	4-valued	GL_FLAT
GL_PATH_DASH_OFFSET_NV	$\Re$	0.0
GL_PATH_DASH_OFFSET_RESET_NV	2-valued	GL_MOVE_TO_CONTINUES_NV
GL_PATH_CLIENT_LENGTH_NV	$\Re+$	0.0
GL_PATH_FILL_MODE_NV	4-valued	GL_COUNT_UP_NV
GL_PATH_FILL_MASK_NV	mask	all 1's
GL_PATH_FILL_COVER_MODE_NV	3-valued	GL_CONVEX_HULL_NV
GL_PATH_STROKE_COVER_MODE_NV	3-valued	GL_CONVEX_HULL_NV
GL_PATH_STROKE_MASK_NV	mask	all 1's

Table 2: *Settable path object parameters.*

3.2 Dashing State

Dashing is an embellishment to stroking where a repeated pattern of enabled stroking and gaps in stroking is applied during stroking. The conventional `glPathParameteriNV`, etc. commands are ill-suited to setting a variable number of dash offsets.

Instead parameters to control the dash pattern of a stroked path are specified by the command

```
void glPathDashArrayNV(GLuint path,
                       GLsizei dashCount,
                       const GLfloat *dashArray);
```

where *path* is the name of an existing path object. A *dashCount* of zero indicates the path object is not dashed; in this case, the *dashArray* is not accessed. Otherwise, *dashCount* provides a count of how many float values to read from the *dashArray* array.

3.3 Computed Parameters and Querying State

All settable path object state is able to be queried; this includes settable parameters with `glGetPathParameterivNV` and `glGetPathParameterfvNV`, the dashing array with `glGetPathDashArrayNV`, the path command array with `glGetPathCommandsNV`, and path coordinate array with `glGetPathCoordsNV`. Additionally, computed parameters for each path object can be queried; see Table 3.

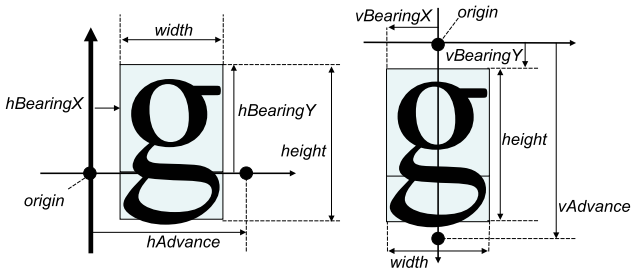

Figure 2: *Glyph metrics.*

Name	Type
GL_PATH_COMMAND_COUNT_NV	N
GL_PATH_COORD_COUNT_NV	N
GL_PATH_COMPUTED_LENGTH_NV	$\Re+$
GL_PATH_OBJECT_BOUNDING_BOX_NV	$4 \times \Re$
GL_PATH_FILL_BOUNDING_BOX_NV	$4 \times \Re$
GL_PATH_STROKE_BOUNDING_BOX_NV	$4 \times \Re$

Table 3: *Computed path object parameters.*

3.4 Glyph and Font Metrics

Path objects created from character points from a font are tagged with additional read-only glyph metrics. These metrics are useful for text layout. Additionally, every glyph has aggregate per-font metrics for its corresponding font. The metrics are obtained directly from the font except for any scaling based on the *emScale*. The `glGetPathMetricRangeNV` and `glGetPathMetricsNV` queries return the glyph and per-font metrics for a range or sequence of path objects respectively.

As shown in Figure 2, the glyph metrics provide each glyph's width and height and (x, y) bearing and advance for both horizontal and vertical layout. These metrics are expressed in path space units.

Additional per-font metrics can be queried from any glyph belonging to a particular font. These metrics include a bounding box large enough to contain any glyph in the font, the native number of font units per em, the font-wide ascender, descender, and height distances, maximum advance for horizontal and vertical layout, underline position and thickness.

Requested metrics are specified in a bitmask and returned to an application-provided array of floats.

4 Rendering Paths via Stencil, then Cover

Once a path object is created, it can be rendered with the "stencil, then cover" approach.

The mapping from path space to clip space and ultimately window space is determined by OpenGL's standard modelview, projection, viewport, and depth range transforms. An OpenGL programmer familiar with OpenGL's `glMatrixMode`, `glRotatef`,

glTranslatef, etc. matrix commands for transforming 3D geometry uses the same commands to manipulate the transformations of path objects.

For example, the code below establishes an orthographic path-to-clip-space to map the $[0..500] \times [0..400]$ region used by the star-and-heart path:

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

This code demonstrates OpenGL's selector-free matrix manipulation commands introduced by the EXT_direct_state_access (DSA) extension [Kilgard 2009].

4.1 Path Rendering in Two Steps

Now we can render the filled and stroked star-and-heart path. We assume the stencil buffer has been initially cleared to zero.

Stencil Step for Filling First we stencil the filled region of the star-and-heart path into the stencil buffer with the glStencilFillPathNV command:

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

The winding number of each sample in the framebuffer w.r.t. the transformed path is added (GL_COUNT_UP) to the stencil value corresponding to each rasterized same. The 0x1F mask indicates that only the five least significant stencil bits are modified—effectively resulting in modulo-32 addition. More typically, this mask will be 0xFF.

Instead of GL_COUNT_UP, we could also subtract (GL_COUNT_DOWN) or invert bits (GL_INVERT) by a count equal to each sample's winding number w.r.t. the transformed path.

Cover Step for Filling Second we conservatively cover the previously stenciled filled region of the star-and heart path. The shading, stencil testing, and blending are fully controlled by the application's OpenGL context state during the cover. So we first enable stencil testing to discard color samples with a stencil value of zero (those not in the path as determined by the prior stencil step); for samples that survive the stencil test, we want to reset the stencil value to zero and shade the corresponding sample's color value green. So:

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

The result is a green star to the left of a green heart.

Stencil Step for Stroking Stroking proceeds similarly in two steps; however before rendering, we 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 we first stencil the stroked coverage for the heart-and-star path into the stencil buffer:

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

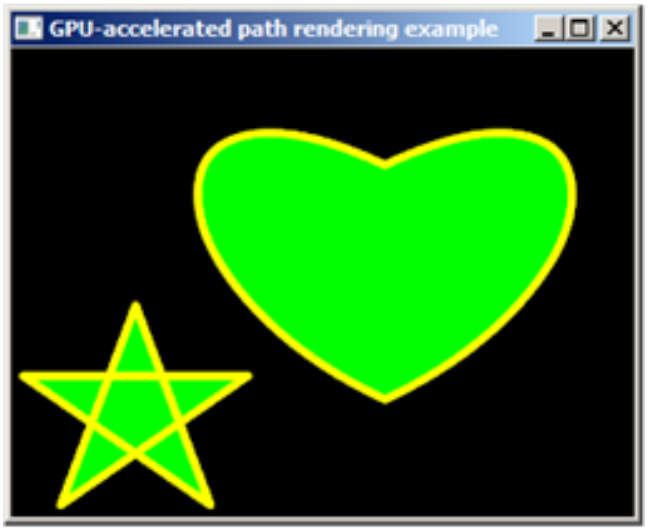

Figure 3: *Filled and stroked path rendering result.*

This computes the point containment of every sample in the framebuffer w.r.t. the stroked path—and if the sample is contained in the path's stroke, the sample's stencil value is set to 0x1 with a write mask of bit-inverted zero (writing all stencil bits).

Cover Step for Stroking Second we conservatively cover the previously stenciled stroked region of the star-and heart path. We leave stencil as configured previously—non-zero values will include the 0x1 value written by the glStencilStrokePathNV command, but we change the color to yellow:

```
glColor3f(1,1,0); // yellow
glCoverStrokePathNV(pathObj, CONVEX_HULL);
```

The complete rendering result is shown in Figure 3.

4.2 Accessible Samples

The fill and stroke stencil/cover commands *conceptually* operate on all samples in the framebuffer. Potentially every sample in the framebuffer could be updated by these commands. However, the set of *accessible samples* be restricted the by current OpenGL context state.

Clip planes, polygon stipple, window ownership, scissoring, stencil testing (on write masked stencil bits during stencil fill/stroke operations), depth testing, depth bounds testing, and the multisample mask all limit, when enabled, the accessible samples.

The cover fill/stroke commands further limit the updated samples to the bounding box or convex hull (depending on the cover mode).

4.3 Path Coordinate Generation

Paths do not have per-vertex attributes such as colors and texture coordinates that are interpolated over geometric primitives as 3D geometry does. Instead varying attributes used by the fragment shader must be generated as a linear function of the path-space coordinate system. The glPathColorGenNV, glPathTexGenNV, and glPathFogGenNV command generated color, texture coordinate sets, and the fog coordinate respectively; the loosely mimic OpenGL's fixed-function glTexGenfvNV, etc. commands.

To match a common idiom in path rendering standards, the path coordinate generation supports mapping a path's path space bounding box to a normalized $[0..1] \times [0..1]$ square.

5 Text Handling

5.1 Instanced Rendering

Efficient rendering of glyphs is very important for any path rendering system. In addition to the ability to create path objects from Unicode character points of fonts and query glyph metrics for such path objects, instanced commands for stenciling and covering sequences of path objects in a single OpenGL command are provided.

`glStencilFillPathInstancedNV` and `glCoverFillPathInstancedNV` for filling and `glStencilStrokePathInstancedNV` and `glCoverStrokePathInstancedNV` for stroking accept arrays of path objects where each path object instance has its own local transformation.

This is intended for rendering spans of characters with corresponding glyph path objects, but could be used for rendering any sequence of path objects. To facilitate text, there is a base path object value to which each path object offset in the sequence is added. This allows a string of ASCII characters to be provided where the base path object value identifies the base of a range of glyphs specified with `glPathGlyphRangeNV`. The array of offsets can even be a UTF-8 or UTF-16 string to facilitate easy rendering of Unicode text.

The instanced cover commands include a special `GL_BOUNDING_BOX_OF_BOUNDING_BOXES` cover mode where the bounding boxes of each locally transformed path object's cover geometry is combined (*unioned*) into a single bounding box.

These instanced commands give the OpenGL implementation the freedom to reorder the geometry sets used during the instanced stencil step for better efficiency in the GPU.

5.2 Spacing and Kerning

Good aesthetics and legibility for horizontal spans of text generally involves appropriate spacing for the glyphs. When this spacing depends on which pairs of glyphs are mutually adjacent, this is called kerning. We provide a `glGetPathSpacingNV` query that accepts a sequence of path objects (in the same way the instanced stencil/cover commands do) and returns an array of translations corresponding to the kerned spacing of the glyphs.

The returned array of translations from `glGetPathSpacingNV` is immediately suitable to pass as the array of translations used for the local transformation sequence when using the instanced stencil/cover commands.

While applications with complex text layout requirements might judge this mechanism insufficiently sophisticated, because the mechanism is simple, cross-platform, and generates kerned glyph translations as expected by the instanced stencil/cover commands, we anticipate this functionality will meet the basic text layout needs of many applications.

6 Geometric Queries

`NV_path_rendering` includes a set of common geometric queries on paths. The queries `glIsPointInFillPathNV` and `glIsPointInStrokePathNV` provide efficient determinations of whether an (x, y) point in a path's local coordinate system is inside or outside the fill or stroke of the path. The query `glGetPathLengthNV` provides a means to obtain arc lengths over specified command sequences for a path. The query `glPointAlongPathNV` returns an (x, y) point and (dx, dy) tangent a given arc length into a path's command sequence.

These queries are intended to be compatible with queries supported by OpenVG.

The crucial rationale for these queries is they are consistent with our implementation's internal computations for being inside/outside the path's stroke or fill and arc length computations (such as for dashing).

7 Antialiasing

Applications are expected to render into multisample framebuffers to achieve acceptable antialiasing quality. Use existing APIs to allocate multisample framebuffer resources. `NV_path_rendering` automatically generates multisample coverage when the framebuffer supports multisampling.

Applications can also use OpenGL's accumulation buffer mechanism with jittered rendering to exceed the base multisampling quality available.

8 More Resources

8.1 NVIDIA Resources

NVIDIA provides developer resources for you to get started programming with `NV_path_rendering`.

`http://developer.nvidia.com/nv-path-rendering`

NVIDIA's `NV_path_rendering` Path Rendering Software Development Kit (NVprSDK) contains eleven complete examples of varying complexity. The most complex example is capable of rendering an interesting subset of SVG.

8.2 Other Resources

The OpenGL Extension Wrangler (GLEW) [Stewart et al.] includes support for the `NV_path_rendering` extension so GLEW provides a hassle-free way to gain access to the extension's OpenGL entry points.

8.3 Driver Availability

The first NVIDIA driver to support `NV_path_rendering` is Release 275.33 (June 2011). Drivers from Release 301.42 (May 2012) have substantially better rendering performance, particularly for GeForce 400 and later GPUs (so-called Fermi or Kepler based GPUs). We are continuing to improve the performance of `NV_path_rendering` so obtain the most recent available driver version.

References

ADOBE SYSTEMS. 1985. *PostScript Language Reference Manual*, 1st ed. Addison-Wesley Longman Publishing Co., Inc. 1

CSS WORKING GROUP. 2011. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, W3C Recommendation*. 3

KHRONOS GROUP, 2008. OpenVG specification version 1.1. 1

KILGARD, M., AND BOLZ, J. 2012. Gpu-accelerated path rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012) 31*, 6 (Nov.), to appear. 1

KILGARD, M., 2009. `EXT_direct_state_access`. http://www.opengl.org/registry/specs/EXT/direct_state_access.txt . 5

KILGARD, M., 2012. `NV_path_rendering`. http://www.opengl.org/registry/specs/NV/path_rendering.txt . 1

SEGAL, M., AND AKELEY, K. 2012. *The OpenGL Graphics System: A Specification (Version 4.3 (Compstibility Profile) - August 6, 2012*. 1

STEWART, N., ET AL. The OpenGL extension wrangler library. http://glew.sourceforge.net/ . 6