An Introduction to NV_path_rendering

Mark J. Kilgard
NVIDIA Corporation
June 8, 2011
Purpose of this Presentation

- Overview of GPU-accelerated path rendering
  - Using “stencil, then cover”
- Explain and demonstrate the **NV_path_rendering** API
  - Aimed primarily at programmers
- Introduce you to the content of NVIDIA’s NVpr SDK
What is path rendering?

- A rendering approach
  - Resolution-independent two-dimensional graphics
  - Occlusion & transparency depend on rendering order
    - So called “Painter’s Algorithm”
  - Basic primitive is a path to be filled or stroked
    - Path is a sequence of path commands
    - Commands are
      - moveto, lineto, curveto, arcto, closepath, etc.

- Standards
  - **Content**: PostScript, PDF, TrueType fonts, Flash, Scalable Vector Graphics (SVG), HTML5 Canvas, Silverlight, Office drawings
  - **APIs**: Apple Quartz 2D, Khronos OpenVG, Microsoft Direct2D, Cairo, Skia, Qt::QPainter, Anti-grain Graphics,
## 3D Rendering vs. Path Rendering

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>GPU 3D rendering</th>
<th>Path rendering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensionality</td>
<td>Projective 3D</td>
<td>2D, typically affine</td>
</tr>
<tr>
<td>Pixel mapping</td>
<td>Resolution independent</td>
<td>Resolution independent</td>
</tr>
<tr>
<td>Occlusion</td>
<td>Depth buffering</td>
<td>Painter’s algorithm</td>
</tr>
<tr>
<td>Rendering primitives</td>
<td>Points, lines, triangles</td>
<td>Paths</td>
</tr>
<tr>
<td>Primitive constituents</td>
<td>Vertices</td>
<td>Control points</td>
</tr>
<tr>
<td>Constituents per primitive</td>
<td>1, 2, or 3 respectively</td>
<td>Unbounded</td>
</tr>
<tr>
<td>Topology of filled primitives</td>
<td>Always convex</td>
<td>Can be concave, self-intersecting, and have holes</td>
</tr>
<tr>
<td>Degree of primitives</td>
<td>$1^{\text{st}}$ order (linear)</td>
<td>Up to $3^{\text{rd}}$ order (cubic)</td>
</tr>
<tr>
<td>Rendering modes</td>
<td>Filled, wire-frame</td>
<td>Filling, stroking</td>
</tr>
<tr>
<td>Line properties</td>
<td>Width, stipple pattern</td>
<td>Width, dash pattern, capping, join style</td>
</tr>
<tr>
<td>Color processing</td>
<td>Programmable shading</td>
<td>Painting + filter effects</td>
</tr>
<tr>
<td>Text rendering</td>
<td>No direct support ($2^{\text{nd}}$ class support)</td>
<td>Omni-present ($1^{\text{st}}$ class support)</td>
</tr>
<tr>
<td>Raster operations</td>
<td>Blending</td>
<td>Brushes, blend modes, compositing</td>
</tr>
<tr>
<td>Color model</td>
<td>RGB or sRGB</td>
<td>RGB, sRGB, CYMK, or grayscale</td>
</tr>
<tr>
<td>Clipping operations</td>
<td>Clip planes, scissoring, stenciling</td>
<td>Clipping to an arbitrary clip path</td>
</tr>
<tr>
<td>Coverage determination</td>
<td>Per-color sample</td>
<td>Sub-color sample</td>
</tr>
</tbody>
</table>
**CPU vs. GPU at Rendering Tasks over Time**

Pipelined 3D Interactive Rendering

Path Rendering

*Goal of NV_path_rendering is to make path rendering a GPU task*
What is NV_path_rendering?

- OpenGL extension to GPU-accelerate path rendering
- Uses “stencil, then cover” (StC) approach
  - Create a path object
  - **Step 1:** “Stencil” the path object into the stencil buffer
    - GPU provides fast stenciling of filled or stroked paths
  - **Step 2:** “Cover” the path object and stencil test against its coverage stenciled by the prior step
    - Application can configure arbitrary shading during the step
    - More details later
- Supports the union of functionality of all major path rendering standards
  - Includes all stroking embellishments
  - Includes first-class text and font support
  - Allows this functionality to mix with traditional 3D and programmable shading
OpenGL Path Rendering API Structure

- Path object management
- Path data specification
  - String-based path specification
  - Data-based (command array + coordinate array) path specification
  - Font- and glyph-based path specification
  - Linear combination (interpolation) of existing paths
- Path parameters
  - stroking parameters (end caps, join styles, dashing, dash caps)
  - quality parameters (cubic approximation)
- Path rendering
  - Path stenciling (fill & stroke)
  - Path covering (fill & stroke)
- Path object queries
- Instanced path rendering
- Querying glyph metrics from glyph path objects
- Geometric queries on path objects
Path Object Management

- Standard OpenGL GLuint object names
  - app-generated, not returned by driver
  - important for font glyphs & instancing
- Standard *is-a* query and *generate* & *delete* commands
  - `glIsPathNV`, `glGenPathsNV`, `glDeletePathsNV`
  - Familiar to anyone using OpenGL objects
Path Specification

- Several ways
  - strings
    - standard grammars exist for encoding paths as strings
      - SVG and PostScript both have standard string encodings
    - `glPathStringNV`
  - data
    - array of path commands with corresponding coordinates
    - `glPathCommandsNV` initially
    - `glPathSubCommands`, `glPathCoords`, `glPathSubCoords` for updates
  - fonts
    - given a range of glyphs in named fonts, created a path object for each glyph
    - `glPathGlyphsNV`, `glPathGlyphRangeNV`
  - linear combination of existing paths
    - interpolate one, two, or more existing paths
    - requires paths “match” their command sequences
    - `glCopyPathNV`, `glInterpolatePathsNV`, `glCombinePathsNV`
  - linear transformation of existing path
    - `glTransformPathNV`
Enumeration of Path Commands

- Very standard
  - move-to \((x, y)\)
  - close-path
  - line-to \((x, y)\)
  - quadratic-curve \((x_1, y_1, x_2, y_2)\)
  - cubic-curve \((x_1, y_1, x_2, y_2, x_3, y_3)\)
  - smooth-quadratic-curve \((x, y)\)
  - smooth-cubic-curve \((x_1, y_1, x_2, y_2)\)
  - elliptical-arc \((r_x, r_y, x\text{-axis-rotation}, \text{large-arc-flag}, \text{sweep-flag}, x, y)\)

- Other variations
  - Relative (relative-line-to, etc.) versions
  - Horizontal & vertical line versions
  - OpenVG-style elliptical arcs
  - PostScript-style circular arcs

- Idea: provide union of path commands of all major path rendering standards
## Path Command Tokens

<table>
<thead>
<tr>
<th>Command</th>
<th>Relative version</th>
<th>Number of Scalar Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_MOVE_TO_NV</td>
<td>GL_RELATIVE_MOVE_TO_NV</td>
<td>2</td>
</tr>
<tr>
<td>GL_LINE_TO_NV</td>
<td>GL_RELATIVE_LINE_TO_NV</td>
<td>2</td>
</tr>
<tr>
<td>GL_HORIZONTAL_LINE_TO_NV</td>
<td>GL_RELATIVE_HORIZONTAL_LINE_TO_NV</td>
<td>1</td>
</tr>
<tr>
<td>GL_VERTICAL_LINE_TO_NV</td>
<td>GL_RELATIVE_VERTICAL_LINE_TO_NV</td>
<td>1</td>
</tr>
<tr>
<td>GL_QUADRATIC_CURVE_TO_NV</td>
<td>GL_RELATIVE_QUADRATIC_CURVE_TO_NV</td>
<td>4</td>
</tr>
<tr>
<td>GL_CUBIC_CURVE_TO_NV</td>
<td>GL_RELATIVE_CUBIC_CURVE_TO_NV</td>
<td>6</td>
</tr>
<tr>
<td>GL_SMOOTH_QUADRATIC_CURVE_TO_NV</td>
<td>GL_RELATIVE_SMOOTH_QUADRATIC_CURVE_TO_NV</td>
<td>2</td>
</tr>
<tr>
<td>GL_SMOOTH_CUBIC_CURVE_TO_NV</td>
<td>GL_RELATIVE_SMOOTH_CUBIC_CURVE_TO_NV</td>
<td>4</td>
</tr>
<tr>
<td>GL_SMALL_CCW_ARC_TO_NV</td>
<td>GL_RELATIVE_SMALL_CCW_ARC_TO_NV</td>
<td>5</td>
</tr>
<tr>
<td>GL_SMALL_CW_ARC_TO_NV</td>
<td>GL_RELATIVE_SMALL_CW_ARC_TO_NV</td>
<td>5</td>
</tr>
<tr>
<td>GL_LARGE_CCW_ARC_TO_NV</td>
<td>GL_RELATIVE_LARGE_CCW_ARC_TO_NV</td>
<td>5</td>
</tr>
<tr>
<td>GL_LARGE_CW_ARC_TO_NV</td>
<td>GL_RELATIVE_LARGE_CW_ARC_TO_NV</td>
<td>5</td>
</tr>
<tr>
<td>GL_CIRCULAR_CCW_ARC_TO_NV</td>
<td>n/a</td>
<td>5</td>
</tr>
<tr>
<td>GL_CIRCULAR_CW_ARC_TO_NV</td>
<td>n/a</td>
<td>5</td>
</tr>
<tr>
<td>GL_CIRCULAR_TANGENT_ARC_TO_NV</td>
<td>n/a</td>
<td>5</td>
</tr>
<tr>
<td>GL_ARC_TO_NV</td>
<td>GL_RELATIVE_ARC_TO_NV</td>
<td>7</td>
</tr>
<tr>
<td>GL_CLOSE_PATH_NV</td>
<td>n/a</td>
<td>0</td>
</tr>
</tbody>
</table>
Path String Format Grammars

- **GL_PATH_FORMAT_SVG_NV**
  - Conforms to BNF in SVG 1.1 specification
  - ASCII string encoding
  - Very convenient because readily available in SVG files
  - Supports SVG-style partial elliptical arcs
  - Examples:
    - "M100,180 L40,10 L190,120 L10,120 L160,10 z" // star
    - "M300 300 C 100 400,100 200,300 100,500 200,500 400,300 300Z" // heart

- **GL_PATH_FORMAT_PS_NV**
  - Conforms to PostScript’s sub-grammar for user paths
  - Allows more compact path encoding than SVG
    - Includes binary encoding, includes accounting for byte order
    - Includes ASCII-85 encoding
  - Supports PostScript-style circular arcs
  - Examples:
    - "100 180 moveto 40 10 lineto 190 120 lineto 10 120 lineto 160 10 lineto closepath" // star
    - "300 300 moveto 100 400 100 200 300 100 curveto 500 200 500 400 300 300 curveto closepath" // heart
FYI: Complete SVG Grammar

```
svg-path:
  wsp* moveto-drawto-command-groups? wsp*
moveto-drawto-command-groups:
  moveto-drawto-command-group
  moveto-drawto-command-group wsp* moveto-drawto-command-groups
moveto-drawto-command-group:
  moveto wsp* drawto-commands?
drawto-commands:
  drawto-command
  drawto-command wsp* drawto-commands
drawto-command:
  closepath
  lineto
  vertical-lineto
  curveto
  smooth-curveto
  quadratic-bezier-curveto
  smooth-quadratic-bezier-curveto
  elliptical-arc
moveto:
  ("M" | "m") wsp* moveto-argument-sequence
moveto-argument-sequence:
  coordinate-pair
  coordinate-pair comma-wsp? lineto-argument-sequence
closepath:
  ("Z" | "z")
lineto:
  ("L" | "l") wsp* lineto-argument-sequence
lineto-argument-sequence:
  coordinate-pair
  coordinate-pair comma-wsp? lineto-argument-sequence
horizontal-lineto:
  ("H" | "h") wsp* horizontal-lineto-argument-sequence
horizontal-lineto-argument-sequence:
  coordinate
  coordinate comma-wsp? horizontal-lineto-argument-sequence
vertical-lineto:
  ("V" | "v") wsp* vertical-lineto-argument-sequence
vertical-lineto-argument-sequence:
  coordinate
  coordinate comma-wsp? vertical-lineto-argument-sequence
curveto:
  ("C" | "c") wsp* curveto-argument-sequence
curveto-argument-sequence:
  curveto-argument
curveto-argument:
  coordinate-pair comma-wsp? coordinate-pair comma-wsp? coordinate-pair
  coordinate-pair comma-wsp? coordinate-pair comma-wsp? coordinate-pair
smooth-curveto:
  ("S" | "s") wsp* smooth-curveto-argument-sequence
smooth-curveto-argument-sequence:
  smooth-curveto-argument
smooth-curveto-argument:
  smooth-curveto-argument comma-wsp? smooth-curveto-argument-sequence
smooth-curveto-argument:
  smooth-curveto-argument comma-wsp? smooth-curveto-argument-sequence
```
FYI: Complete PS Grammar (1)

ps-path:
  | ps-wsp* user-path? ps-wsp*
  | ps-wsp encoded-path ps-wsp*
user-path:
  | user-path-cmd ps-wsp+ user-path
user-path-cmd:
  | setbbox
  | ps-moveto
  | rmoveto
  | ps-lineto
  | rlineto
  | ps-curveto
  | rcurveto
  | arc
  | arcn
  | arct
  | ps-closepath
  | ucache
setbbox:
  | numeric-value numeric-value numeric-value numeric-value setbbox-cmd
setbbox-cmd:
  | "setbbox"
  | #x02 #x0f
ps-moveto:
  | numeric-value numeric-value moveto-cmd
moveto-cmd:
  | "moveto"
  | #x02 #x6b
rmoveto:
  | numeric-value numeric-value rmoveto-cmd
rmoveto-cmd:
  | "rmoveto"
  | #x02 #x86
ps-lineto:
  | numeric-value numeric-value lineto-cmd
lineto-cmd:
  | "lineto"
  | #x02 #x8b
rlineto:
  | numeric-value numeric-value rlinoeto-cmd
rlineto-cmd:
  | "rlineto"
  | #x02 #x93
ps-curveto:
  | numeric-value numeric-value numeric-value numeric-value numeric-value numeric-value curveto-cmd
curveto-cmd:
  | "curveto"
  | #x02 #x2b
rcurveto:
  | numeric-value numeric-value numeric-value numeric-value numeric-value numeric-value rcurveto-cmd
curveto-cmd:
  | "rcurveto"
  | #x02 #x7a

arc:
  | numeric-value numeric-value numeric-value numeric-value numeric-value numeric-value arc-cmd
arc-cmd:
  | "arc"
  | #x02 #x05
arcn:
  | numeric-value numeric-value numeric-value numeric-value numeric-value arcn-cmd
arcn-cmd:
  | "arcn"
  | #x02 #x06
arct:
  | numeric-value numeric-value numeric-value numeric-value numeric-value arct-cmd
arct-cmd:
  | "arct"
  | #x02 #x07
ps-closepath:
  | "closepath"
  | #x02 #x16
ucache:
  | "ucache"
  | #x02 #x11
encoded-path:
  | data-array ps-wsp" operator-string
data-array:
  | ("ps-wsp" numeric-value-sequence? ")"
  | homogeneous-number-array
  | ascii85-homogeneous-number-array
operator-string:
  | hexadecimal-binary-string
  | ascii85-string
  | short-binary-string
  | be-long-binary-string
  | le-long-binary-string
hexadecimal-binary-string:
  | "<" ps-wsp-" chars* hexadecimal-sequence ps-wsp-" >"
hexadecimal-sequence:
  | hexadecimal-digit
  | hexadecimal-digit ps-wsp-" chars* hexadecimal-sequence
hexadecimal-digit:
  | digit
  | "a"..."f" |
  | "A"..."F"
short-binary-string:
  | #x8e one-byte (one-byte)^n
  | where n is the value of the one-byte production decoded
  | as an unsigned integer, 0 through 255
be-long-binary-string:
  | #x8f two-bytes (one-byte)^n
  | where n is the value of the two-bytes production decoded
  | as an unsigned integer, 0 through 65535, decoded in big-endian byte order
le-long-binary-string:
  | #x90 two-bytes (one-byte)^n
  | where n is the value of the two-bytes production decoded
  | as an unsigned integer, 0 through 65535, decoded in little-endian byte order
numeric-value-sequence:
  numeric-value:
    numeric-value numeric-value-sequence
numeric-value:
  number ps-wsp+
  | radix-number ps-wsp+
  | be-integer-32bit
  | le-integer-32bit
  | be-integer-16bit
  | le-integer-16bit
  | be-integer-8bit
  | le-integer-8bit
  | be-fixed-16bit
  | le-fixed-16bit
  | le-fixed-32bit
  | be-float-ieee
  | le-float-ieee
  | native-float-ieee
be-integer-32bit:
  #x84 four-bytes
le-integer-32bit:
  #x85 four-bytes
be-integer-16bit:
  #x86 two-bytes
le-integer-16bit:
  #x87 two-bytes
le-integer-8bit:
  #x88 one-byte
be-fixed-32bit:
  #x89 #x0..#x1F four-bytes
de-fixed-32bit:
  #x89 #x80..#x9F four-bytes
be-float-ieee:
  #x8A four-bytes
le-float-ieee:
  #x95 #x0..#x1F two-bytes
d-fixed-32bit:
  #x95 #x80..#x9F two-bytes
be-fixed-16bit:
  #x95 #x20..#x2F two-bytes
le-fixed-16bit:
  #x95 #xA0..#xAF two-bytes
be-float-ieee:
  #xB0 two-bytes
le-float-ieee:
  #x95 #x30 two-bytes
native-float-ieee:
  #x95 ( #x31 | #xB1 ) two-bytes
radix-number:
  base "#" base-number
base:
  digit-sequence
base-number:
  base-digit-sequence
base-digit-sequence:
  base-digit
  | base-digit base-digit-sequence
base-digit:
  digit
  | "a".."z"
  | "A".."Z"
  | ascii85-string:
    "<~" (#x21..#x75 | "z" | psp-wsp ) "~>"
ascii85-homogeneous-number-array:
  "<~" (#x21..#x75 | "z" | psp-wsp ) "~>" one-byte:
  #x0..#xFF
two-bytes:
  #x0..#xFF #x0..#xFF
four-bytes:
  #x0..#xFF #x0..#xFF #x0..#xFF #x0..#xFF
numeric-value-sequence:
  numeric-value:
    numeric-value numeric-value-sequence
numeric-value:
  number ps-wsp+
  | radix-number ps-wsp+
  | be-integer-32bit
  | le-integer-32bit
  | be-integer-16bit
  | le-integer-16bit
  | be-integer-8bit
  | le-integer-8bit
  | be-fixed-16bit
  | le-fixed-16bit
  | le-fixed-32bit
  | be-float-ieee
  | le-float-ieee
  | native-float-ieee
be-integer-32bit:
  #x84 four-bytes
le-integer-32bit:
  #x85 four-bytes
be-integer-16bit:
  #x86 two-bytes
le-integer-16bit:
  #x87 two-bytes
le-integer-8bit:
  #x88 one-byte
be-fixed-32bit:
  #x89 #x0..#x1F four-bytes
de-fixed-32bit:
  #x89 #x80..#x9F four-bytes
be-float-ieee:
  #x8A four-bytes
le-float-ieee:
  #x95 #x0..#x1F two-bytes
d-fixed-32bit:
  #x95 #x80..#x9F two-bytes
be-fixed-16bit:
  #x95 #x20..#x2F two-bytes
le-fixed-16bit:
  #x95 #xA0..#xAF two-bytes
be-float-ieee:
  #xB0 two-bytes
le-float-ieee:
  #x95 #x30 two-bytes
native-float-ieee:
  #x95 ( #x31 | #xB1 ) two-bytes
radix-number:
  base "#" base-number
base:
  digit-sequence
base-number:
  base-digit-sequence
base-digit-sequence:
  base-digit
  | base-digit base-digit-sequence
base-digit:
  digit
  | "a".."z"
  | "A".."Z"
Settable Path Parameters

- Filling has just a few parameters
  - default fill mode
  - default fill mask
  - default fill cover mode

- Stroking has many
  - stroke width (floating-point number)
  - end caps (flat, square, round, triangular)
  - join styles (round, bevel, miter)
    - miter limit (floating-point number)
  - dash array count + dash array
    - array of floats in multiple of stroke width
  - client length (floating-point) scales dash array
  - dash offset reset for OpenVG (move-to-continues, move-to-resets)
  - dash offset (floating-point)
  - dash cap (flat, square, round, triangular)
  - stroke over sample count (integer)
  - default stroke cover mode
  - default stroke mask

---

<table>
<thead>
<tr>
<th>GL_FLAT</th>
<th>GL_ROUND_NV</th>
<th>GL_SQUARE_NV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GL_MITER_NV</th>
<th>GL_ROUND_NV</th>
<th>GL_BEVEL_NV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

various dash styles
### glPathParameter Parameters

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH_STROKE_WIDTH_NV</td>
<td>float</td>
<td>Non-negative</td>
</tr>
<tr>
<td>PATH_INITIAL_END_CAP_NV</td>
<td>enum</td>
<td>GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV</td>
</tr>
<tr>
<td>PATH_TERMINAL_END_CAP_NV</td>
<td>enum</td>
<td>GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV</td>
</tr>
<tr>
<td>PATH_INITIAL_DASH_CAP_NV</td>
<td>enum</td>
<td>GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV</td>
</tr>
<tr>
<td>PATH_TERMINAL_DASH_CAP_NV</td>
<td>enum</td>
<td>GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV</td>
</tr>
<tr>
<td>PATH_MITER_LIMIT_NV</td>
<td>float</td>
<td>Non-negative</td>
</tr>
<tr>
<td>PATH_DASH_OFFSET_NV</td>
<td>float</td>
<td>Any value</td>
</tr>
<tr>
<td>PATH_DASH_OFFSET_RESET_NV</td>
<td>enum</td>
<td>GL_MOVE_TO_RESET_NV, GL_MOVE_TO_CONTINUES_NV</td>
</tr>
<tr>
<td>PATH_CLIENT_LENGTH_NV</td>
<td>float</td>
<td>Non-negative</td>
</tr>
<tr>
<td>PATH_SAMPLE_QUALITY_NV</td>
<td>float</td>
<td>Clamped to [0,1] range</td>
</tr>
<tr>
<td>PATH_STROKE_OVERSAMPLE_COUNT_NV</td>
<td>integer</td>
<td>Non-negative</td>
</tr>
<tr>
<td>PATH_FILL_MODE_NV</td>
<td>enum</td>
<td>GL_COUNT_UP_NV, GL_COUNT_DOWN_NV, GL_INVERT</td>
</tr>
<tr>
<td>PATH_FILL_MASK_NV</td>
<td>integer</td>
<td>Any value</td>
</tr>
<tr>
<td>PATH_FILL_COVER_MODE_NV</td>
<td>enum</td>
<td>GL_CONVEX_HULL_NV, GL_MULTI_HULLS_NV, GL_BOUNDING_BOX_NV</td>
</tr>
<tr>
<td>PATH_STROKE_COVER_MODE_NV</td>
<td>enum</td>
<td>GL_CONVEX_HULL_NV, GL_MULTI_HULLS_NV, GL_BOUNDING_BOX_NV</td>
</tr>
</tbody>
</table>
Dash Array State

- Dashing specified as an array of lengths
  
  ```c
  void glPathDashArrayNV(GLuint path,
                         GLsizei dashCount,
                         const GLfloat *dashArray);
  ```

- Defines alternating “on” and “off” sequence of dash segment lengths
  - Odd dash pattern “doubled” so [1,3,2] is treated as the pattern [1,3,2,1,3,2]
  - Dash count of zero means not dashed
    - Initial state of path objects

- Has its own dedicated query
  
  ```c
  void glGetPathDashArrayNV(GLuint name,
                             GLfloat *dashArray);
  ```
Dashing Content Examples

Frosting on cake is dashed elliptical arcs with round end caps for “beaded” look; flowers are also dashing.

Same cake missing dashed stroking details

Artist made windows with dashed line segment

Technical diagrams and charts often employ dashing

All content shown is fully GPU rendered

Dashing character outlines for quilted look
Rendering Path Objects

- Stencil operation
  - only updates stencil buffer
  - `glStencilFillPathNV`, `glStencilStrokePathNV`
- Cover operation
  - `glCoverFillPathNV`, `glCoverStrokePathNV`
  - renders hull polygons guaranteed to “cover” the region updated by corresponding stencil
- Two-step rendering paradigm
  - stencil, then cover (StC)
- Application controls cover stenciling and shading operations
  - Gives application considerable control
- No vertex, tessellation, or geometry shaders active during either step
  - Why? Paths have control points and rasterized regions, **not** vertices or triangles
Path Filling vs. Stroking

just filling

just stroking

filling + stroke = intended content
## Stencil, then Stroke Command Prototypes

### Filling
- Stencil step
  ```c
  void glStencilFillPathNV(
    GLuint path,
    GLenum fillMode,
    GLuint mask)
  ```

- Cover step
  ```c
  void glCoverFillPathNV(
    GLuint path,
    GLenum coverMode)
  ```

### Stroking
- Stencil step
  ```c
  void glStencilStrokePathNV(
    GLuint path,
    GLint reference,
    GLuint mask)
  ```

- Cover step
  ```c
  void glCoverStrokePathNV(
    GLuint path,
    GLenum coverMode)
  ```
Excellent Geometric Fidelity for Stroking

- Correct stroking is hard
  - Lots of CPU implementations approximate stroking
- GPU-accelerated stroking avoids such short-cuts
  - GPU has FLOPS to compute true stroke point containment

![GPU-accelerated stroking](image1)
![OpenVG reference](image2)

Stroking with tight end-point curve
Let’s draw a green concave 5-point star

Path specification by string of a star

```c
GLuint pathObj = 42;
const char *pathString = "M100,180 L40,10 L190,120 L10,120 L160,10 z";
g1PathStringNV(pathObj, GL_PATH_FORMAT_SVG_NV,
    strlen(pathString), pathString);
```

Alternative: path specification by data

```c
static const GLubyte pathCommands[5] = {
    GL_MOVE_TO_NV, GL_LINE_TO_NV, GL_LINE_TO_NV, GL_LINE_TO_NV,
    GL_LINE_TO_NV, GL_CLOSE_PATH_NV
};
static const GLshort pathVertices[5][2] =
    { {100,180}, {40,10}, {190,120}, {10,120}, {160,10} };
g1PathCommandsNV(pathObj, 6, pathCommands, GL_SHORT, 10, pathVertices);
```
Initialization

- 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);
  ```

- Specify the Path's Transform
  
  ```
  glMatrixModeEXT(GL_PROJECTION);
glOrthoEXT(GL_MODELVIEW, 0, 200, 0, 200, -1, 1);
  ```

- Nothing really specific to path rendering here
Render star with non-zero fill style

Stencil path
```c
glStencilFillPathNV(pathObj, GL_COUNT_UP_NV, 0x1F);
```

Cover path
```c
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);
```

Alternative: for even-odd fill style

Just program glStencilFunc differently
```c
glStencilFunc(GL_NOTEQUAL, 0, 0x1);  // alternative mask
```
**“Stencil, then Cover”**

Path Fill Stenciling

- **Specify a path**
- **Specify arbitrary path transformation**
  - Projective (4x4) allowed
  - Depth values can be generated for depth testing
- **Sample accessibility determined**
  - Accessibility can be limited by any or all of:
    - Scissor test, depth test, stencil test, view frustum, user-defined clip planes, sample mask, stipple pattern, and window ownership
- **Winding number w.r.t. the transformed path is computed**
  - Added to stencil value of accessible samples

**Stencil, then Cover**

Path Fill Stenciling

- Specify a path
- Specify arbitrary path transformation
  - Projective (4x4) allowed
  - Depth values can be generated for depth testing
- Sample accessibility determined
  - Accessibility can be limited by any or all of:
    - Scissor test, depth test, stencil test, view frustum, user-defined clip planes, sample mask, stipple pattern, and window ownership
- Winding number w.r.t. the transformed path is computed
  - Added to stencil value of accessible samples

**Diagram**

- Path object
- Path front-end
- Per-path transform fill region operations
  - Projective transform
  - Clipping & scissoring
  - Window, depth & stencil tests
  - Path winding number computation
- Per-sample operations
  - Winding number w.r.t. the transformed path is computed
- Stencil update:
  - +, -, or invert
- Stencil buffer
- Fill stenciling specific
“Stencil, then Cover” Path Fill Covering

- Specify a path
- Specify arbitrary path transformation
  - Projective (4x4) allowed
  - Depth values can be generated for depth testing
- Sample accessibility determined
  - Accessibility can be limited by any or all of
    - Scissor test, depth test, stencil test, view frustum, user-defined clip planes, sample mask, stipple pattern

1. path front-end
2. projective transform
3. clipping & scissoring
4. window, depth & stencil tests
5. stencil update
6. programmable path shading
7. color buffer

Cover fill path command
Adding Stroking to the Star

- **After the filling, add a stroked “rim” to the star like this...**

- Set some stroking parameters *(one-time)*:
  ```
  glPathParameterfNV(pathObj, GL_STROKE_WIDTH_NV, 10.5);
  glPathParameteriNV(pathObj, GL_JOIN_STYLE_NV, GL_ROUND_NV);
  ```

- **Stroke the star**
  - Stencil path
    ```
    glStencilStrokePathNV(pathObj, 0x3, 0xF); // stroked samples marked “3”
    ```
  - Cover path
    ```
    glEnable(GL_STENCIL_TEST);
    glStencilFunc(GL_EQUAL, 3, 0xF); // update if sample marked “3”
    glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
    glColor3f(1,1,0); // yellow
    glCoverStrokePathNV(pathObj, GL_BOUNDING_BOX_NV);
    ```

- non-zero fill style
- even-odd fill style
“Stencil, then Cover” Path Stroke Stenciling

- Specify a path
- Specify arbitrary path transformation
  - Projective (4x4) allowed
  - Depth values can be generated for depth testing
- Sample accessibility determined
  - Accessibility can be limited by any or all of
    - Scissor test, depth test, stencil test, view frustum, user-defined clip planes, sample mask, stipple pattern, and window ownership
- Point containment w.r.t. the stroked path is determined
  - Replace stencil value of contained samples

Stencil stenciling specific

Path front-end
  → Projective transform
  ↓ Clipping & scissoring
  ↓ Window, depth & stencil tests
  ↓ Stroke point containment
  ↓ Stencil update: replace

Sample accessibility

Path object

Stencil buffer
“Stencil, then Cover” Path Stroke Covering

- Specify a path
- Specify arbitrary path transformation
  - Projective (4x4) allowed
  - Depth values can be generated for depth testing
- Sample accessibility determined
  - Accessibility can be limited by any or all of
    - Scissor test, depth test, stencil test, view frustum, user-defined clip planes, sample mask, stipple pattern, and window ownership
- Conservative covering geometry uses stencil to “cover” stroked path
  - Determined by prior stencil step

cover stroke path command

- path front-end
- projective transform
- clipping & scissoring
- window, depth & stencil tests
- stencil update
  - typically zero
- programmable path shading
- color buffer
- programmable path shading
- color buffer
- sample accessibility
- path object

stroked
Path Object State

- **Path commands**
  - Unbounded number of commands allowed

- **Path coordinates**
  - Match up with commands
  - Example: each cubic Bezier segment has 6 coordinates
    - (x1, y1), (x2, y2), (x3, y3)
  - Initial control point (x0, y0) is implicit based on prior path command’s end-point

- **Path parameters**
  - Stroke width, end caps, join styles, dash pattern, etc.

- **Glyph metrics**
  - When path object is created from a font
Path Object Queries

- All settable path object state is queriable
  - just like all conventional OpenGL state
- glGetPathParameter\{i,f\}vNV
- glGetPathParameter\{i,f\}NV
- glGetPathCommandsNV
- glGetPathCoordsNV
- Can also query \textit{derived} state of path objects
  - GL\_PATH\_COMMAND\_COUNT\_NV
  - GL\_PATH\_COORD\_COUNT\_NV
  - GL\_DASH\_ARRAY\_COUNT\_NV
  - GL\_COMPUTED\_LENGTH\_NV
  - GL\_PATH\_OBJECT\_BOUNDING\_BOX\_NV
  - GL\_PATH\_FILL\_BOUNDING\_BOX\_NV
  - GL\_PATH\_STROKE\_BOUNDING\_BOX\_NV
Supported Glyph Metrics

- Based on FreeType2 metrics
  - Provides both per-glyph & per-font face metrics

**Image credit:** FreeType 2 Tutorial
<table>
<thead>
<tr>
<th>Bit field name</th>
<th>Glyph metric tag</th>
<th>Bit number from LSB in bitmask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_GLYPH_WIDTH_BIT_NV</td>
<td>width</td>
<td>0</td>
<td>Glyph's width</td>
</tr>
<tr>
<td>GL_GLYPH_HEIGHT_BIT_NV</td>
<td>height</td>
<td>1</td>
<td>Glyph's height</td>
</tr>
<tr>
<td>GL_GLYPH_HORIZONTAL_BEARING_X_BIT_NV</td>
<td>hBearingX</td>
<td>2</td>
<td>Left side bearing for horizontal layout</td>
</tr>
<tr>
<td>GL_GLYPH_HORIZONTAL_BEARING_Y_BIT_NV</td>
<td>hBearingY</td>
<td>3</td>
<td>Top side bearing for horizontal layout</td>
</tr>
<tr>
<td>GL_GLYPH_HORIZONTAL_BEARING_ADVANCE_BIT_NV</td>
<td>hAdvance</td>
<td>4</td>
<td>Advance width for horizontal layout</td>
</tr>
<tr>
<td>GL_GLYPH_VERTICAL_BEARING_X_BIT_NV</td>
<td>vBearingX</td>
<td>5</td>
<td>Left side bearing for vertical layout</td>
</tr>
<tr>
<td>GL_GLYPH_VERTICAL_BEARING_Y_BIT_NV</td>
<td>vBearingY</td>
<td>6</td>
<td>Top side bearing for vertical layout</td>
</tr>
<tr>
<td>GL_GLYPH_VERTICAL_BEARING_ADVANCE_BIT_NV</td>
<td>vAdvance</td>
<td>7</td>
<td>Advance height for vertical layout</td>
</tr>
<tr>
<td>GL_GLYPH_HAS_KERNING_NV</td>
<td>-</td>
<td>8</td>
<td>True if glyph has a kerning table</td>
</tr>
</tbody>
</table>
# Per-Font Face Metric Names

<table>
<thead>
<tr>
<th>Bit field name</th>
<th>Bit number from LSB in bitmask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_FONT_X_MIN_BOUNDS_NV</td>
<td>16</td>
<td>Horizontal minimum (left-most) of the font bounding box. The font bounding box (this metric and the next 3) is large enough to contain any glyph from the font face.</td>
</tr>
<tr>
<td>GL_FONT_Y_MIN_BOUNDS_NV</td>
<td>17</td>
<td>Vertical minimum (bottom-most) of the font bounding box.</td>
</tr>
<tr>
<td>GL_FONT_X_MAX_BOUNDS_NV</td>
<td>18</td>
<td>Horizontal maximum (right-most) of the font bounding box.</td>
</tr>
<tr>
<td>GL_FONT_Y_MAX_BOUNDS_NV</td>
<td>29</td>
<td>Vertical maximum (top-most) of the font bounding box.</td>
</tr>
<tr>
<td>GL_FONT_UNITS_PER_EM_NV</td>
<td>20</td>
<td>Number of units in path space (font units) per Em square for this font face. This is typically 2048 for TrueType fonts, and 1000 for PostScript fonts.</td>
</tr>
<tr>
<td>GL_FONT_ASCENDER_NV</td>
<td>21</td>
<td>Typographic ascender of the font face. For font formats not supplying this information, this value is the same as GL_FONT_Y_MAX_BOUNDS_NV.</td>
</tr>
<tr>
<td>GL_FONT_DESCENDER_NV</td>
<td>22</td>
<td>Typographic descender of the font face (always a positive value). For font formats not supplying this information, this value is the same as GL_FONT_Y_MIN_BOUNDS_NV.</td>
</tr>
<tr>
<td>GL_FONT_HEIGHT_NV</td>
<td>23</td>
<td>Vertical distance between two consecutive baselines in the font face (always a positive value).</td>
</tr>
<tr>
<td>GL_FONT_MAX_ADVANCE_WIDTH_NV</td>
<td>24</td>
<td>Maximal advance width for all glyphs in this font face. (Intended to make word wrapping computations easier.)</td>
</tr>
<tr>
<td>GL_FONT_MAX_ADVANCE_HEIGHT_NV</td>
<td>25</td>
<td>Maximal advance height for all glyphs in this font face for vertical layout. For font formats not supplying this information, this value is the same as GL_FONT_HEIGHT_NV.</td>
</tr>
<tr>
<td>GL_FONT_UNDERLINE_POSITION_NV</td>
<td>26</td>
<td>Position of the underline line for this font face. This position is the center of the underlying stem.</td>
</tr>
<tr>
<td>GL_FONT_UNDERLINE_THICKNESS_NV</td>
<td>27</td>
<td>Thickness of the underline of this font face.</td>
</tr>
<tr>
<td>GL_FONT_HAS_KERNING_NV</td>
<td>28</td>
<td>True if font face provides a kerning table</td>
</tr>
</tbody>
</table>
Glyph Spacing, including Kerning

- NV\_path\_rendering tries to avoid text layout
  - But kerning requires more than per-glyph metrics
- Kerning occurs when a font face specifies how a particular pair of glyphs should be spaced when adjacent to each other
  - For example: the “A” and “V” often space tighter than other glyphs
- glEnablePathSpacingNV returns horizontal spacing for a sequence of path objects
- Three modes
  - GL\_ACCUM\_ADJACENT\_PAIRS\_NV—spacing can be immediately passed to instanced path rendering commands
  - GL\_AJACENT\_PAIRS\_NV
  - GL\_FIRST\_TO\_REST\_NV
- Provides independent scale factors for the advance and kerning terms—set kerning term to zero to ignore kerning
- Returns an array of 1- or 2-component spacing based on GL\_TRANSLATE\_X or GL\_TRANSLATE\_2D
Instanced Path Rendering

- Stencil multiple path objects in a single call
  - Efficient, particularly for text
  - Minimizes state changes
- Also cover multiple paths in a single call
  - `glStencilFillPathInstancedNV`
  - `glStencilStrokePathInstancedNV`
  - `glCoverFillPathInstancedNV`
  - `glCoverStencilPathInstancedNV`
- Operation
  - Takes an array of path objects, each with its own transform
  - Each path object covered gets a unique instance ID
  - Or can have a `GL_BOUNDING_BOX_OF_BOUNDING_BOXES_NV` mode to cover with a single box
Instanced Filling
Function Prototypes

- **Instanced Filling**

```c
void glStencilFillPathInstancedNV(GLsizei numPaths,
                                  GLenum pathNameType, const void *paths,
                                  GLuint pathBase,
                                  GLenum fillMode,
                                  GLuint mask,
                                  GLenum transformType,
                                  const GLfloat *transformValues);
```

```c
void glCoverFillPathInstancedNV(GLsizei numPaths,
                                GLenum pathNameType, const void *paths,
                                GLuint pathBase,
                                GLenum coverMode,
                                GLenum transformType,
                                const GLfloat *transformValues);
```
Instanced Stroking
Function Prototypes

Instanced Filling

```c
void glStencilStrokePathInstanceNV(GLsizei numPaths,
    GLenum pathNameType, const void *paths,
    GLuint pathBase,
    GLint reference,
    GLuint mask,
    GLenum transformType,
    const GLfloat *transformValues);
```

```c
void glCoverStrokePathInstanceNV(GLsizei numPaths,
    GLenum pathNameType, const void *paths,
    GLuint pathBase,
    GLenum coverMode,
    GLenum transformType,
    const GLfloat *transformValues);
```
First-class, Resolution-independent Font Support

- Fonts are a standard, first-class part of all path rendering systems
  - Foreign to 3D graphics systems such as OpenGL and Direct3D, but natural for path rendering
  - Because letter forms in fonts have outlines defined with paths
    - TrueType, PostScript, and OpenType fonts all use outlines to specify glyphs
- NV_path_rendering makes font support easy
  - Can specify a range of path objects with
    - A specified font
    - Sequence or range of Unicode character points
- No requirement for applications use font API to load glyphs
  - You can also load glyphs “manually” from your own glyph outlines
  - Functionality provides OS portability and meets needs of applications with mundane font requirements
Three Ways to Specify a Font

- **GL_SYSTEM_FONT_NAME_NV**
  - Corresponds to the system-dependent mapping of a name to a font
  - For example, “Arial” maps to the system’s Arial font
  - Windows uses native Win32 fonts services
  - Linux uses fontconfig + freetype2 libraries

- **GL_STANDARD_FONT_NAME_NV**
  - Three built-in fonts, same on all platforms
  - “Sans”, “Serif”, and “Mono”
  - Based on DejaVu fonts
  - Guaranteed to be available no matter what

- **GL_FONT_FILE_NAME_NV**
  - Use freetype2 to load fonts from a system file name
  - Requires freetype2 DLL to be available on Windows
  - Just works in Linux
Font API Example: Initialization

- Allocate unused path object range for glyphs
  
  ```c
  GLuint glyphBase = glGenPathsNV(6);
  ```

- Load glyphs for a sequence of characters
  
  ```c
  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);
  ```

- Web-style alternative font faces
  
  ```c
  glPathGlyphsNV(glyphBase,
                 GL_SYSTEM_FONT_NAME_NV, "Arial", GL_BOLD_BIT_NV,
                 wordLen, GL_UNSIGNED_BYTE, word,
                 GL_SKIP_MISSING_GLYPH_NV, templatePathObject, emScale);
  ```

  ```c
  glPathGlyphsNV(glyphBase,
                 GL_STANDARD_FONT_NAME_NV, "Sans", GL_BOLD_BIT_NV,
                 wordLen, GL_UNSIGNED_BYTE, word,
                 GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
  ```
Font API Example: Initialization

- Allocate unused path object range for glyphs

```c
GLuint glyphBase = glGenPathsNV(6);
```

- Load glyphs for a sequence of characters

```c
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);
```

- Web-style alternative font faces

```c
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_MISSIGN_GLYPH_NV, templatePathObject, emScale);
```
Font API Example: Pre-rendering

**Simple horizontal layout**

```c
const char *text = "OpenGL";

GLfloat xtranslate[6+1]; // wordLen+1

glfwGetPathSpacingNV(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);
```

**Query per-font face metrics**

```c
GLfloat yMinMax[2];

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

**Initialize canvas-to-window transform**

```c
glfwLoadIdentityEXT(GL_PROJECTION);
glfwOrthoEXT(GL_PROJECTION,
             0, xtranslate[6], yMinMax[0], yMinMax[1],
             -1, 1); // [zNear..zFar]
```
Path API Example: Rendering

- **Clear window**

  ```
  // Has the window's pixels been damaged due to exposure or resizing?
  if (glutLayerGet(GLUT_NORMAL_DAMAGED)) {
    // Yes, stencil clear to zero is needed.
    glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
  } else {
    // No, just color clear is needed.
    glClear(GL_COLOR_BUFFER_BIT);
  }
  ```

- **Stencil “Hello World”**

  ```
  glStencilFillPathInstancedNV(numChars, fontBase,
   GL_UNSIGNED_BYTE, text,
   GL_DEFAULT_NV, 0x0, // use obj’s default count mode & fill mask
   GL_TRANSLATE_1D_NV, xoffsets);
  ```

- **Cover “Hello World”**

  ```
  glEnable(GL_STENCIL_TEST);
  // accept only non-zero fragments (as determined by stencil step)
  glStencilFunc(GL_NOT_EQUAL, 0, 0xFF);
  glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO); // reset non-0 stencil back to 0
  glColor3f(0,0,1); // blue
  glCoverFillPathInstancedNV(numChars, fontBase,
   GL_UNSIGNED_BYTE, text,
   GL_BOUNDING_BOX_OF_BOUNDING_BOXES_NV, // coverage mode
   GL_TRANSLATE_X_NV, xoffsets);
  
  glDisable(GL_STENCIL_TEST);
  ```
Font API Example: Loose Ends

- Present frame
  ```c
  glutSwapBuffers();
  ```

- Clean up
  ```c
  glDeletePathsNV(glyphBase, 6);
  ```
Mapping Entire Font Character Set

- Allocate unused path object range for glyphs
  
  ```
  const int unicodeRange = 0x110000;  // 1,114,112 Unicode chars
  GLuint glyphBase = glGenPathsNV(unicodeRange);
  ```

- Load glyphs for a range of Unicode character points
  
  ```
  const GLfloat emScaleObject = 2048;  // match TrueType convention
  GLuint templatePathObject = ~0;  // Non-existant path object
  glPathGlyphRangeNV(glyphBase,
                     GL_SYSTEM_FONT_NAME_NV, "Helvetica", GL_BOLD_BIT_NV,
                     /*first character*/0, /*count*/unicodeRange,
                     GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
  ```

- Web-style alternative font faces
  
  ```
  glPathGlyphRangeNV(glyphBase,
                     GL_SYSTEM_FONT_NAME_NV, "Arial", GL_BOLD_BIT_NV,
                     /*first character*/0, /*count*/unicodeRange,
                     GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
  ```

  ```
  glPathGlyphRangeNV(glyphBase,
                     GL_STANDARD_FONT_NAME_NV, "Sans", GL_BOLD_BIT_NV,
                     /*first character*/0, /*count*/unicodeRange,
                     GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
  ```
Several commands take a sequence of path objects
- The instanced commands such as
  - `glStencilFillPathInstancedNV`
  - `glGetPathMetricsNV`
  - `glGetPathSpacingNV`

The type of the sequence array can be
- `GL_UNSIGNED_BYTE`
- `GL_UNSIGNED_SHORT`, essentially UCS-2
- `GL_UNSIGNED_INT`
- `GL_2_BYTES`, `GL_3_BYTES`, and `GL_4_BYTES`
- `GL_UTF8_NV` 8-bit Unicode Transformation Format
- `GL_UTF16_NV` 16-bit Unicode Transformation Format

Allowing UTF modes means Unicode strings can be directly passed to OpenGL for path rendering.
Handling Common Path Rendering Functionality: Filtering

- GPUs are highly efficient at image filtering
  - Fast texture mapping
    - Mipmapping
    - Anisotropic filtering
    - Wrap modes
- CPUs aren't really

GPU

Moiré artifacts

Qt

Cairo
Handling Uncommon Path Rendering Functionality: Projection

- Projection “just works”
  - Because GPU does everything with perspective-correct interpolation
Projective Path Rendering Support Compared

- **GPU**: flawless
- **Skia**: yes, but bugs
- **Cairo**: unsupported
- **Qt**: unsupported

Correct: ✔️
Incorrect: ×
Unsupported: 〉

Correct: ✔️
Correct: ✔️
Unsupported: 〉
Unsupported: 〉
Unsupported: 〉
Unsupported: 〉
Path Geometric Queries

- **glIsPointInFillPathNV**
  - determine if object-space \((x,y)\) position is inside or outside path, given a winding number mask

- **glIsPointInStrokePathNV**
  - determine if object-space \((x,y)\) position is inside the stroke of a path
  - accounts for dash pattern, joins, and caps

- **glGetPathLengthNV**
  - returns approximation of geometric length of a given sub-range of path segments

- **glPointAlongPathNV**
  - returns the object-space \((x,y)\) position and 2D tangent vector a given offset into a specified path object
  - Useful for “text follows a path”

- Queries are modeled after OpenVG queries
Accessible Samples of a Transformed Path

- When stenciled or covered, a path is transformed by OpenGL’s current modelview-projection matrix
  - Allows for arbitrary 4x4 projective transform
  - Means (x,y,0,1) object-space coordinate can be transformed to have depth
- Fill or stroke stenciling affects “accessible” samples
- A samples is *not* accessible if any of these apply to the sample
  - clipped by user-defined or view frustum clip planes
  - discarded by the polygon stipple, if enabled
  - discarded by the pixel ownership test
  - discarded by the scissor test, if enabled
  - discarded by the depth test, if enabled
  - displaced by the polygon offset from `glPathStencilDepthOffsetNV`
- discarded by the depth test, if enabled
- discarded by the (implicitly enabled) stencil test
  - specified by `glPathStencilFuncNV`
  - where the read mask is the bitwise AND of the `glPathStencilFuncNV` read mask and the bit-inversion of the effective mask parameter of the stenciling operation
Mixing Depth Buffering and Path Rendering

- PostScript tigers surrounding Utah teapot
  - Plus overlaid TrueType font rendering
  - No textures involved, no multi-pass
3D Path Rendering Details

- Stencil step uses
  
  ```
  GLfloat slope = -0.05;
  GLint bias = -1;
  glPathStencilDepthOffsetNV(slope, bias);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
  ```

- Stenciling step uses
  
  ```
  glPathCoverDepthFuncNV(GL_ALWAYS);
  ```

- Observation
  
  - Stencil step is testing—but not writing—depth
    - Stencil won’t be updated if stencil step fails depth test at a sample
  - Cover step is writing—but not testing—depth
    - Cover step doesn’t need depth test because stencil test would only pass if prior stencil step’s depth test passed
  - Tricky, but neat because minimal mode changes involved
Without glPathStencilDepthOffset
Bad Things Happen

- Each tiger is layered 240 paths
  - Without the depth offset during the stencil step, all the—essentially co-planar—layers would Z-fight as shown below
Path Transformation Process

Path object

Object-space color/fog/tex generation

Modelview matrix

Eye-space color/fog/tex generation

User-defined clip planes

Projection matrix

View-frustum clip planes

to path stenciling or covering

object-space coordinates

(x,y,0,1)

color/fog/tex coordinates

eye-space coordinates

(xe,ye,ze,we) + attributes

color/fog/tex coords.

clip-space coordinates

(xc,yc,zc,wc) + attributes

clipped eye-space coordinates

(xe,ye,ze,we) + attributes

colored clip-space coordinates

(xc,yc,zc,wc) + attributes
Clip Planes Work with Path Rendering

- Scene showing a Welsh dragon clipped to all 64 combinations of 6 clip planes enabled & disabled
Path Rendering Works with Scissoring and Stippling too

- Scene showing a tiger scissoring into 9 regions
- Tiger with two different polygon stipple patterns
Rendering Paths Clipped to Some Other Arbitrary Path

- Example clipping the PostScript tiger to a heart constructed from two cubic Bezier curves

![Unclipped tiger](image1.png) ![Tiger with pink background clipped to heart](image2.png)
Complex Clipping Example

tiger is 240 paths

cowboy clip is the union of 1,366 paths

result of clipping tiger to the union of all the cowboy paths
During the “cover” step, you can do arbitrary fragment processing
- Could be
  - Fixed-function fragment processing
  - OpenGL assembly programs
  - Cg shaders compiled to assembly with Cg runtime
  - OpenGL Shading Language (GLSL) shaders
  - Your pick—they all work!

Remember:
- Your vertex, geometry, and tessellation shaders are ignored during the cover step
  - (Even your fragment shader is ignored during the “stencil” step)
Example of Bump Mapping on Path Rendered Text

- Phrase “Brick wall!” is path rendered and bump mapped with a Cg fragment shader.
Good anti-aliasing is a big deal for path rendering
- Particularly true for font rendering of small point sizes
- Features of glyphs are often on the scale of a pixel or less

`NV_path_rendering` needs multiple stencil samples per pixel for reasonable antialiasing
- Otherwise, image quality is poor
- 4 samples/pixel bare minimum
- 16 samples/pixel is pretty sufficient
  - But this requires expensive 2x2 supersampling of 4x multisampling—not good for low-end
  - 16x is extremely memory intensive

Alternative: quality vs. performance tradeoff
- Fast enough to render multiple passes to improve quality

Approaches
- Accumulation buffer
- Alpha accumulation
Anti-aliasing Strategy Benefits

- Benefits from GPU’s existing hardware AA strategies
  - Multiple color-stencil-depth samples per pixel
    - 4, 8, or 16 samples per pixel
  - Rotated grid sub-positions
  - Fast downsampling by GPU
  - Avoids conflating coverage & opacity
    - Maintains distinct color sample per sample location
  - Centroid sampling
- Fast enough for temporal scheme
  - >>60 fps means multi-pass improves quality

GPU rendered coverage NOT conflated with opacity

Cairo, Qt, Skia, and Direct2D rendered shows dark cracks artifacts due to conflating coverage with opacity, notice background bleeding
GPU Advantages

- **Fast, quality filtering**
  - Mipmapping of gradient color ramps essentially free
  - Includes anisotropic filtering (up to 16x)
  - Filtering is *post*-conversion from sRGB

- **Full access to programmable shading**
  - No fixed palette of solid color / gradient / pattern brushes
  - Bump mapping, shadow mapping, etc.—it’s all available to you

- **Blending**
  - Supports native blending in sRGB color space
    - Both colors converted to linear RGB
    - Then result is converted stored as sRGB

- **Freely mix 3D and path rendering in same framebuffer**
  - Path rendering buffer can be depth tested against 3D
  - So can 3D rendering be stenciled against path rendering

- Obviously performance is MUCH better than CPUs
Improved Color Space: sRGB Path Rendering

- Modern GPUs have native support for perceptually-correct for:
  - sRGB framebuffer blending
  - sRGB texture filtering
  - No reason to tolerate uncorrected linear RGB color artifacts!
  - More intuitive for artists to control
- Negligible expense for GPU to perform sRGB-correct rendering
  - However quite expensive for software path renderers to perform sRGB rendering
  - Not done in practice

Radial color gradient example moving from saturated red to blue

Linear RGB transition between saturated red and saturated blue has dark purple region

sRGB perceptually smooth transition from saturated red to saturated blue
<table>
<thead>
<tr>
<th>Benchmark Scenes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiger</td>
<td>Dragon</td>
</tr>
<tr>
<td>Coat of Arms</td>
<td>Cowboy</td>
</tr>
<tr>
<td>Round Dogs</td>
<td>Buonaparte</td>
</tr>
<tr>
<td>Butterfly</td>
<td>Embrace</td>
</tr>
<tr>
<td>Spikes</td>
<td>Japanese Strokes</td>
</tr>
</tbody>
</table>

*Images of Tiger, Dragon, Round Dogs, Butterfly, Spikes, Coat of Arms, Cowboy, Buonaparte, Embrace, Japanese Strokes.*
GPU-accelerated Path Rendering Speed Factor

- GL/NVpr
- GL/Cairo
- GL/Qt
- GL/Skia
- GL/D2D
- GL/WARP

Strokes

Tiger  Dragon  Round Dogs  Butterfly  Spikes  Coat of Arms  Cow boy  Buonaparte  Embrace  Japanese Strokes
Benchmark Test Configuration & Assumptions

- **CPU**
  - 2.9 GHz i3 Nehalem
  - Only using a single-core

- **GPU**
  - Fermi GTX 480, so assuming fastest available GPU
  - 16 samples/pixel

- **Ten window resolutions**
  - 100x100 (lowest) to 1,000x1,000 (highest)
  - In 100 pixel increments

- **Ten test scenes**
  - Variety of path complexity, stroking vs. filling, and gradients
    - Scenes shown on next slide
  - Scenes measured rendering from “resolution-independent” representation (static pre-tessellation dis-allowed)
Getting a Driver with NV_path_rendering

- Operating system support
  - 2000, XP, Vista, Windows 7, Linux, FreeBSD, and Solaris
  - No Mac support
- GPU support
  - GeForce 8 and up (Tesla and beyond)
  - More efficient on Fermi GPUs
  - Current performance can be expected to improve
- Available now for preview in the Release 275
  - GeForce 275.33 driver now public
  - We need your feedback
Learning NV_path_rendering

- White paper + source code available
  - “Getting Started with NV_path_rendering”
- Explains
  - Path specification
  - “Stencil, then Cover” API usage
  - Instanced rendering for text and glyphs
NV_path_rendering SDK Examples

- A set of NV_path_rendering examples of varying levels of complexity
  - Most involved example is an accelerated SVG viewer
    - Not a complete SVG implementation

- Compiles on Windows and Linux
  - Needs Visual Studio 2008 for Windows
SDK Example Walkthrough (1)

**pr_basic**: simplest example of path filling & stroking

**pr_hello_world**: kerned, underlined, stroked, and linear gradient filled text

**pr_welsh_dragon**: filled layers

**pr_gradient**: path with holes with texture applied
SDK Example Walkthrough (2)

**pr_font_file**: loading glyphs from a font file with the GL_FONT_FILE_NV target

**pr_korean**: rendering UTF-8 string of Korean characters

**pr_shaders**: use Cg shaders to bump map text with brick-wall texture
SDK Example Walkthrough (3)

**pr_text_wheel**: render projected gradient text as spokes of a wheel

**pr_tiger**: classic PostScript tiger rendered as filled & stroked path layers

**pr_warp_tiger**: warp the tiger with a free projective transform

click & drag the bounding rectangle corners to change the projection
**pr_tiger3d**: multiple projected and depth tested tigers + 3D teapot + overlaid text

**pr_svg**: GPU-accelerated SVG viewer

**pr_pick**: test points to determine if they are in the filled and/or stroked region of a complex path
Very close to fully functional but…
Errata—a few things not working yet

- Instance ID not set for instanced rendering
- GL_MULTI_HULLS_NV for covering paths
- glTransformPathNV for circular arcs
- glTransformPathNV for projective transforms
- Ignored parameters GL_SAMPLE_QUALITY_NV and GL_PATH_OVERSAMPLE_COUNT_NV

*Early Release 275 drivers have a bug (now fixed) where destroying an OpenGL context after using NV_path_rendering can cause the driver to crash*

- Future drivers will fix these deficiencies
- Expect performance to improve too
Feedback and Contacts

- We need your feedback
  - Issues?
  - Questions?

- Contact us by emailing
  - nvpr-support@nvidia.com