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

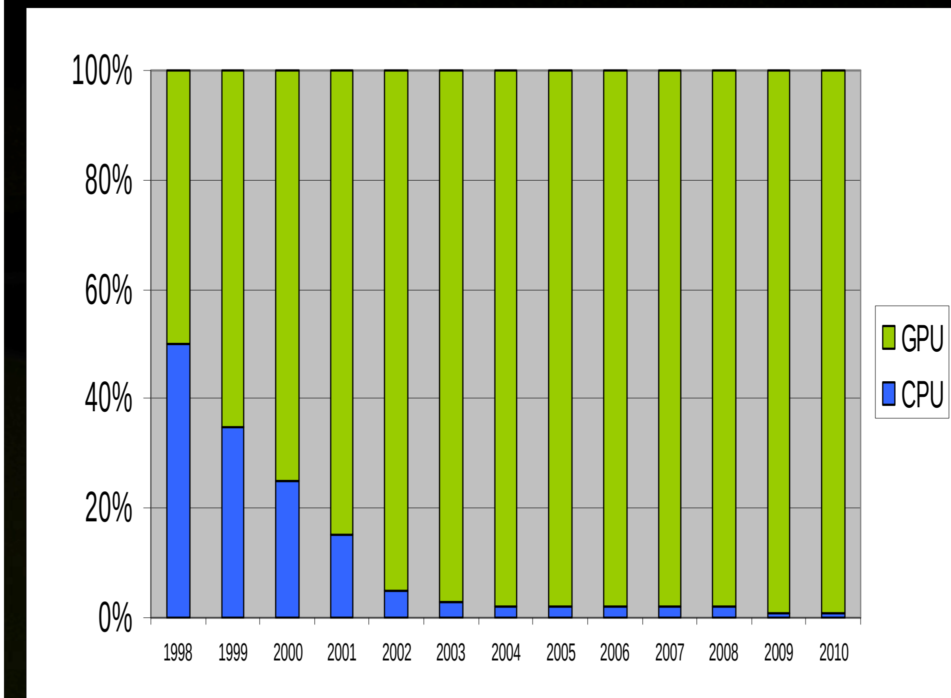
What is path rendering?

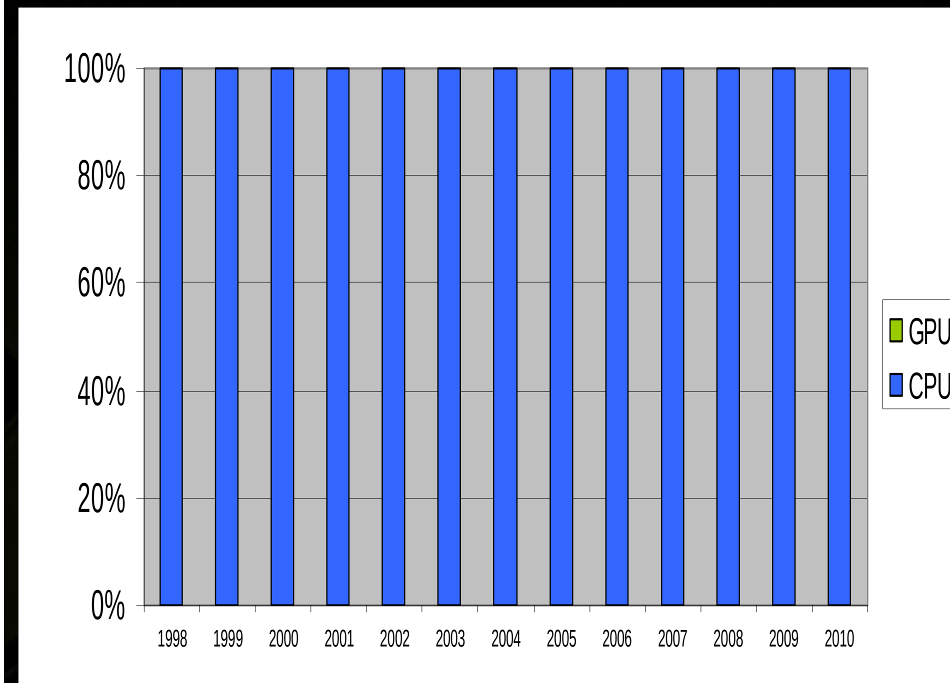
- **A rendering approach**
  - **Resolution-independent two-dimensional graphics**
  - **Occlusion & transparency depend on rendering order**
    - **So called "Painter's Algorithm"**
  - **Basic primitive is a path to be filled or stroked**
    - **Path is a sequence of path commands**
    - **Commands 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

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

CPU vs. GPU at Rendering Tasks over Time





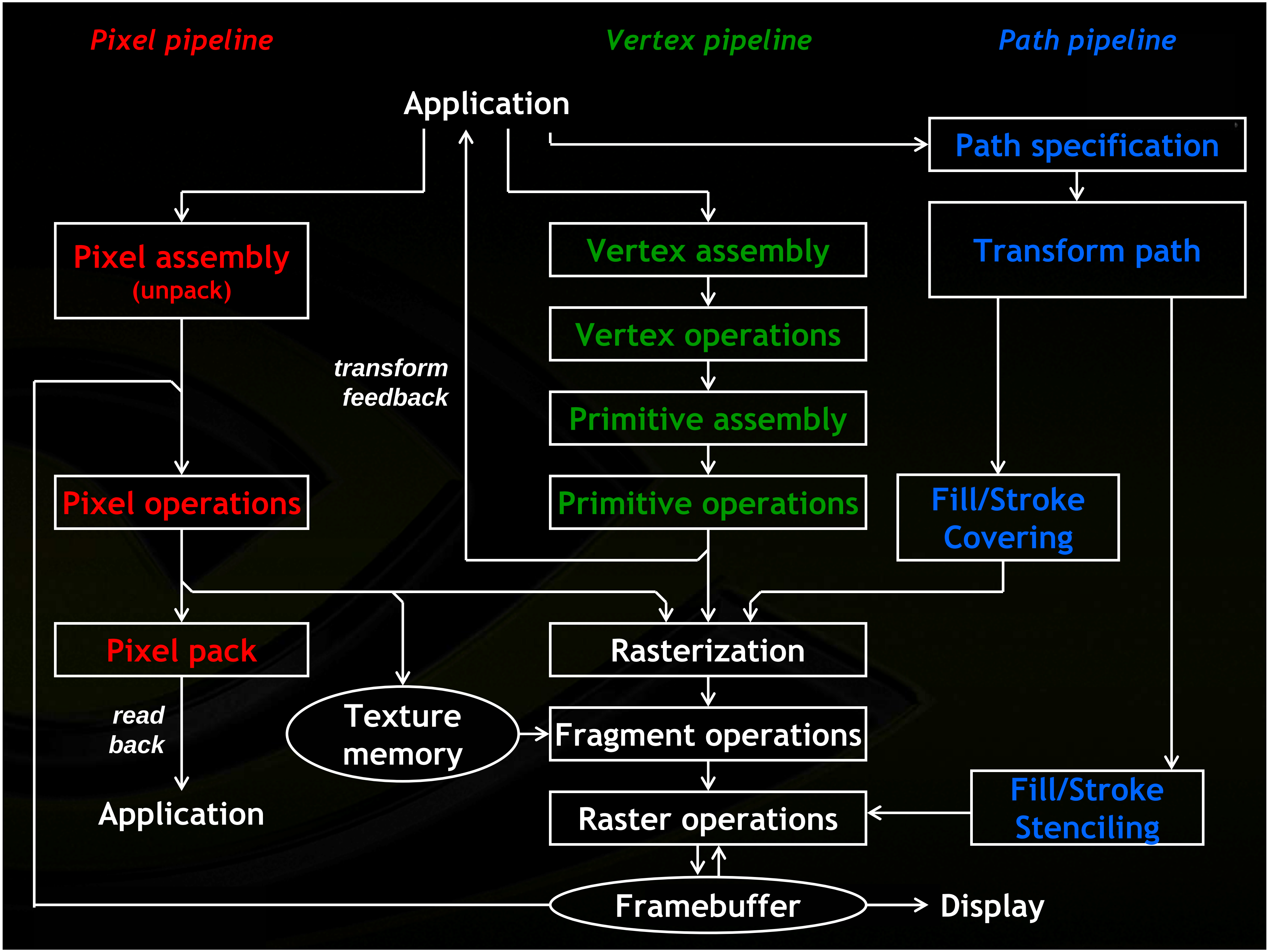
Pipelined 3D Interactive Rendering

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


Pixel pipeline
Vertex pipeline
Path pipeline
Application
Path specification
Transform path
Pixel assembly
(unpack)
Vertex assembly
Vertex operations
transform
feedback
Primitive assembly
Pixel operations
Primitive operations
Fill/Stroke
Covering
Pixel pack
Rasterization
read
back
Texture
memory
Fragment operations
Application
Raster operations
Fill/Stroke
Stenciling
Framebuffer
Display

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 (x1, y1, x2, y2)
  - cubic-curve (x1, y1, x2, y2, x3, y3)
  - smooth-quadratic-curve (x, y)
  - smooth-cubic-curve (x1, y1, x2, y2)
  - elliptical-arc (rx, ry, x-axis-rotation, large-arc-flag, 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

Command	Relative version	Number of Scalar Coordinates
GL_MOVE_TO_NV	GL_RELATIVE_MOVE_TO_NV	2
GL_LINE_TO_NV	GL_RELATIVE_LINE_TO_NV	2
GL_HORIZONTAL_LINE_TO_NV	GL_RELATIVE_HORIZONTAL_LINE_TO_NV	1
GL_VERTICAL_LINE_TO_NV	GL_RELATIVE_VERTICAL_LINE_TO_NV	1
GL_QUADRATIC_CURVE_TO_NV	GL_RELATIVE_QUADRATIC_CURVE_TO_NV	4
GL_CUBIC_CURVE_TO_NV	GL_RELATIVE_CUBIC_CURVE_TO_NV	6
GL_SMOOTH_QUADRATIC_CURVE_TO_NV	GL_RELATIVE_SMOOTH_QUADRATIC_CURVE_TO_NV	2
GL_SMOOTH_CUBIC_CURVE_TO_NV	GL_RELATIVE_SMOOTH_CUBIC_CURVE_TO_NV	4
GL_SMALL_CCW_ARC_TO_NV	GL_RELATIVE_SMALL_CCW_ARC_TO_NV	5
GL_SMALL_CW_ARC_TO_NV	GL_RELATIVE_SMALL_CW_ARC_TO_NV	5
GL_LARGE_CCW_ARC_TO_NV	GL_RELATIVE_LARGE_CCW_ARC_TO_NV	5
GL_LARGE_CW_ARC_TO_NV	GL_RELATIVE_LARGE_CW_ARC_TO_NV	5
GL_CIRCULAR_CCW_ARC_TO_NV	n/a	5
GL_CIRCULAR_CW_ARC_TO_NV	n/a	5
GL_CIRCULAR_TANGENT_ARC_TO_NV	n/a	5
GL_ARC_TO_NV	GL_RELATIVE_ARC_TO_NV	7
GL_CLOSE_PATH_NV	n/a	0

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
   | horizontal-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 comma-wsp? curveto-argument-sequence
curveto-argument:
   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 comma-wsp? smooth-curveto-argument-sequence
smooth-curveto-argument:
   coordinate-pair comma-wsp? coordinate-pair
quadratic-bezier-curveto:
   ( "Q" | "q" ) wsp* quadratic-bezier-curveto-argument-sequence
quadratic-bezier-curveto-argument-sequence:
   quadratic-bezier-curveto-argument
   | quadratic-bezier-curveto-argument comma-wsp?
      quadratic-bezier-curveto-argument-sequence
quadratic-bezier-curveto-argument:
   coordinate-pair comma-wsp? coordinate-pair
smooth-quadratic-bezier-curveto:
   ( "T" | "t" ) wsp* smooth-quadratic-bezier-curveto-argument-sequence
smooth-quadratic-bezier-curveto-argument-sequence:
   coordinate-pair
   | coordinate-pair comma-wsp? smooth-quadratic-bezier-curveto-argument-sequence
elliptical-arc:
   ( "A" | "a" ) wsp* elliptical-arc-argument-sequence
elliptical-arc-argument-sequence:
   elliptical-arc-argument
   | elliptical-arc-argument comma-wsp? elliptical-arc-argument-sequence
elliptical-arc-argument:
   nonnegative-number comma-wsp? nonnegative-number comma-wsp?
      number comma-wsp flag comma-wsp? flag comma-wsp? coordinate-pair
coordinate-pair:
   coordinate comma-wsp? coordinate
coordinate:
   number
nonnegative-number:
   integer-constant
   | floating-point-constant
number:
   sign? integer-constant
   | sign? floating-point-constant
flag:
   "0" | "1"
comma-wsp:
   (wsp+ comma? wsp*) | (comma wsp*)
comma:
   ","
integer-constant:
   digit-sequence
floating-point-constant:
   fractional-constant exponent?
   | digit-sequence exponent
fractional-constant:
   digit-sequence? "." digit-sequence
   | digit-sequence "."
exponent:
   ( "e" | "E" ) sign? digit-sequence
sign:
   "+" | "-"
digit-sequence:
   digit
   | digit digit-sequence
digit:
   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
wsp:
   (#x20 | #x9 | #xD | #xA)
```

FYI: Complete PS Grammar (1)

```
ps-path:
   ps-wsp* user-path? ps-wsp*
   | ps-wsp* encoded-path ps-wsp*
user-path:
   user-path-cmd
   | 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"
   | #x92 #x8F
ps-moveto:
   numeric-value numeric-value moveto-cmd
moveto-cmd:
   "moveto"
   | #x92 #x6B
rmoveto:
   numeric-value numeric-value rmoveto-cmd
rmoveto-cmd:
   "rmoveto"
   | #x92 #x86
ps-lineto:
   numeric-value numeric-value lineto-cmd
lineto-cmd:
   "lineto"
   | #x92 #x63
rlineto:
   numeric-value numeric-value rlineto-cmd
rlineto-cmd:
   "rlineto"
   | #x92 #x85
ps-curveto:
   numeric-value numeric-value numeric-value numeric-value numeric-value numeric-value
          curveto-cmd
curveto-cmd:
   "curveto"
   | #x92 #x2B
rcurveto:
   numeric-value numeric-value numeric-value numeric-value numeric-value numeric-value
          rcurveto-cmd
rcurveto-cmd:
   "rcurveto"
   | #x92 #x7A
arc:
   numeric-value numeric-value numeric-value numeric-value numeric-value arc-cmd
arc-cmd:
   "arc"
   | #x92 #x05
arcn:
   numeric-value numeric-value numeric-value numeric-value numeric-value arcn-cmd
arcn-cmd:
   "arcn"
   | #x92 #x06
arct:
   numeric-value numeric-value numeric-value numeric-value numeric-value arct-cmd
arct-cmd:
   "arct"
   | #x92 #x07
ps-closepath:
   "closepath"
   | #x92 #x16
ucache:
   "ucache"
   | #x92 #xB1
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-chars* ">"
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/
```

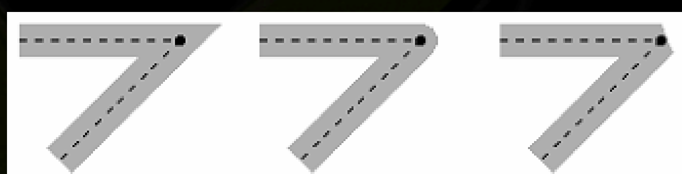
FYI: Complete PS Grammar (2)

```
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
   | le-integer-8bit
   | be-fixed-16bit
   | le-fixed-16bit
   | be-fixed-32bit
   | 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
le-fixed-32bit:
   #x89 #x80..#x9F four-bytes
be-fixed-16bit:
   #x89 #x20..#x2F two-bytes
le-fixed-16bit:
   #x89 #xA0..#xAF two-bytes
be-float-ieee:
   #x8A four-bytes
le-float-ieee:
   #x8B four-bytes
native-float-ieee:
   #x8C four-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"
homogeneous-number-array:
   be-fixed-32bit-array
   | be-fixed-16bit-array
   | be-float-ieee-array
   | native-float-ieee-array
   | le-fixed-32bit-array
   | le-fixed-16bit-array
   | le-float-ieee-array
be-fixed-32bit-array:
   #x95 #x0..#x1F two-bytes ( four-bytes )^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/
be-fixed-16bit-array:
   #x95 #x20..#x2F two-bytes ( two-bytes )^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/
be-float-ieee-array:
   #x95 #x30 two-bytes ( four-bytes )^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-fixed-32bit-array:
   #x95 #x80..#x9F two-bytes ( four-bytes )^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/
le-fixed-16bit-array:
   #x95 #xA0..#xAF two-bytes ( two-bytes )^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/
le-float-ieee-array:
   #x95 #xB0 two-bytes ( four-bytes )^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/
native-float-ieee-array:
   #x95 ( #x31 | #xB1 ) two-bytes ( four-bytes )^n
      /where n is the value of the two-bytes production decoded
       as an unsigned integer, 0 through 65535, decoded in
       the native byte order/
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
ps-wsp:
   ps-wsp-chars
   | ps-comment
ps-wsp-chars:
   ( #x20 | #x9 | #xA | #xC | #xD | #x0 )
ps-comment:
   "%" ( #0..#9 | #xB..#xC | #xE..#xFF )* ( #xD | #xA )
```

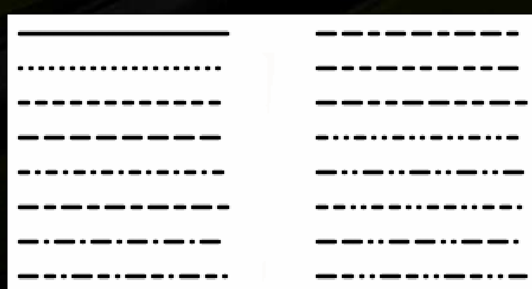
Settable Path Parameters

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

GL_FLAT GL_ROUND_NV GL_SQUARE_NV



GL_MITER_NV GL_ROUND_NV GL_BEVEL_NV



various dash styles

- 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

glPathParameter Parameters

Parameter name	Type	Description
PATH_STROKE_WIDTH_NV	float	Non-negative
PATH_INITIAL_END_CAP_NV	enum	GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV
PATH_TERMINAL_END_CAP_NV	enum	GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV
PATH_INITIAL_DASH_CAP_NV	enum	GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV
PATH_TERMINAL_DASH_CAP_NV	enum	GL_FLAT, GL_SQUARE_NV, GL_ROUND_NV, GL_TRIANGULAR_NV
PATH_JOIN_STYLE_NV	enum	GL_MITER_REVERT_NV, GL_MITER_TRUNCATE_NV, GL_BEVEL_NV, GL_ROUND_NV, GL_NONE
PATH_MITER_LIMIT_NV	float	Non-negative
PATH_DASH_OFFSET_NV	float	Any value
PATH_DASH_OFFSET_RESET_NV	enum	GL_MOVE_TO_RESET_NV, GL_MOVE_TO_CONTINUES_NV
PATH_CLIENT_LENGTH_NV	float	Non-negative
PATH_SAMPLE_QUALITY_NV	float	Clamped to [0,1] range
PATH_STROKE_OVERSAMPLE_COUNT_NV	integer	Non-negative
PATH_FILL_MODE_NV	enum	GL_COUNT_UP_NV, GL_COUNT_DOWN_NV, GL_INVERT
PATH_FILL_MASK_NV	integer	Any value
PATH_FILL_COVER_MODE_NV	enum	GL_CONVEX_HULL_NV, GL_MULTI_HULLS_NV, GL_BOUNDING_BOX_NV
PATH_STROKE_COVER_MODE_NV	enum	GL_CONVEX_HULL_NV, GL_MULTI_HULLS_NV, GL_BOUNDING_BOX_NV

Dash Array State

- Dashing specified as an array of lengths

```
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

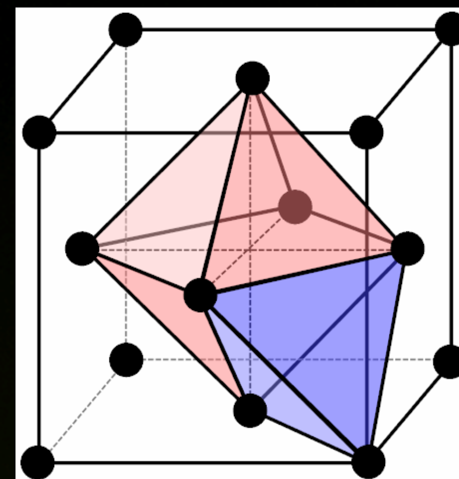
```
void glGetPathDashArrayNV(GLuint name,
                          GLfloat *dashArray);
```

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

This is crazy

Dashing character outlines for quilted look

All content shown is fully GPU rendered

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, <u>not</u> vertices or triangles

Path Filling vs. Stroking

just filling

filling + stroke =
intended content

just stroking

Stencil, then Stroke Command Prototypes

Filling

- Stencil step

```
void glStencilFillPathNV(
  GLuint path,
  GLenum fillMode,
  GLuint mask)
```

- Cover step

```
void glCoverFillPathNV(
  GLuint path,
  GLenum coverMode)
```

Stroking

- Stencil step

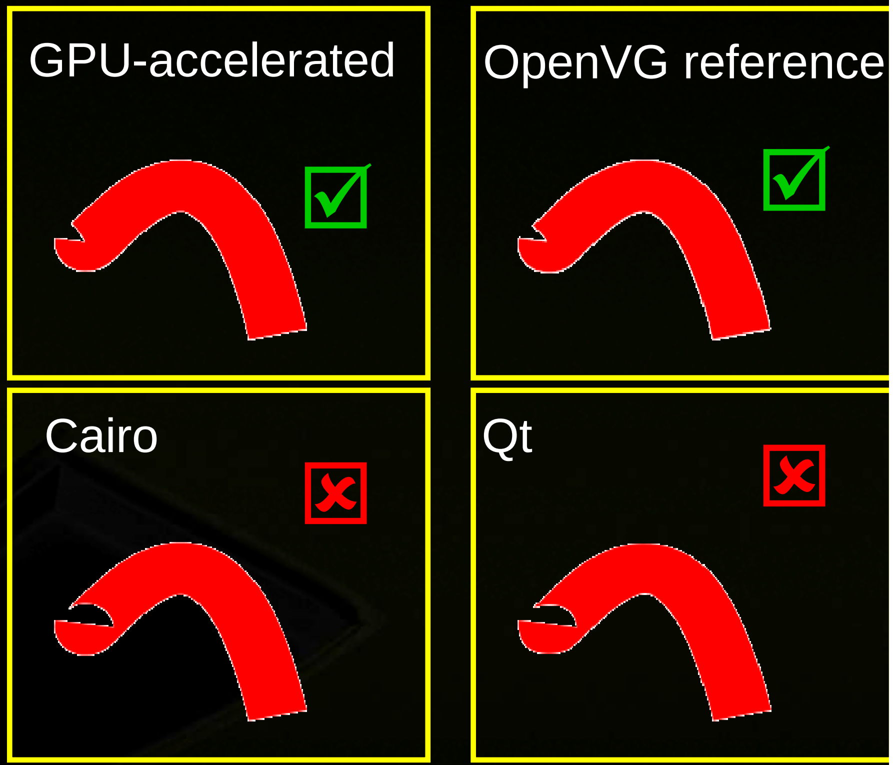
```
void glStencilStrokePathNV(
  GLuint path,
  GLint reference,
  GLuint mask)
```

- Cover step

```
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**



Stroking with tight end-point curve

Path Rendering Example (1 of 3)

- **Let's draw a green concave 5-point star**

even-odd fill style

non-zero fill style

- **Path specification by string of a star**

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

- Alternative: **path specification by data**

```
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} };
glPathCommandsNV(pathObj, 6, pathCommands, GL_SHORT, 10, pathVertices);
```

Path Rendering Example (2 of 3)

- **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**

```
glMatrixIdentityEXT(GL_PROJECTION);
glMatrixOrthoEXT(GL_MODELVIEW, 0,200, 0,200, -1,1); // uses DSA!
```

- **Nothing really specific to path rendering here**

Path Rendering Example (3 of 3)

- Render star with non-zero fill style
  - Stencil path

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

  - Cover path

```
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

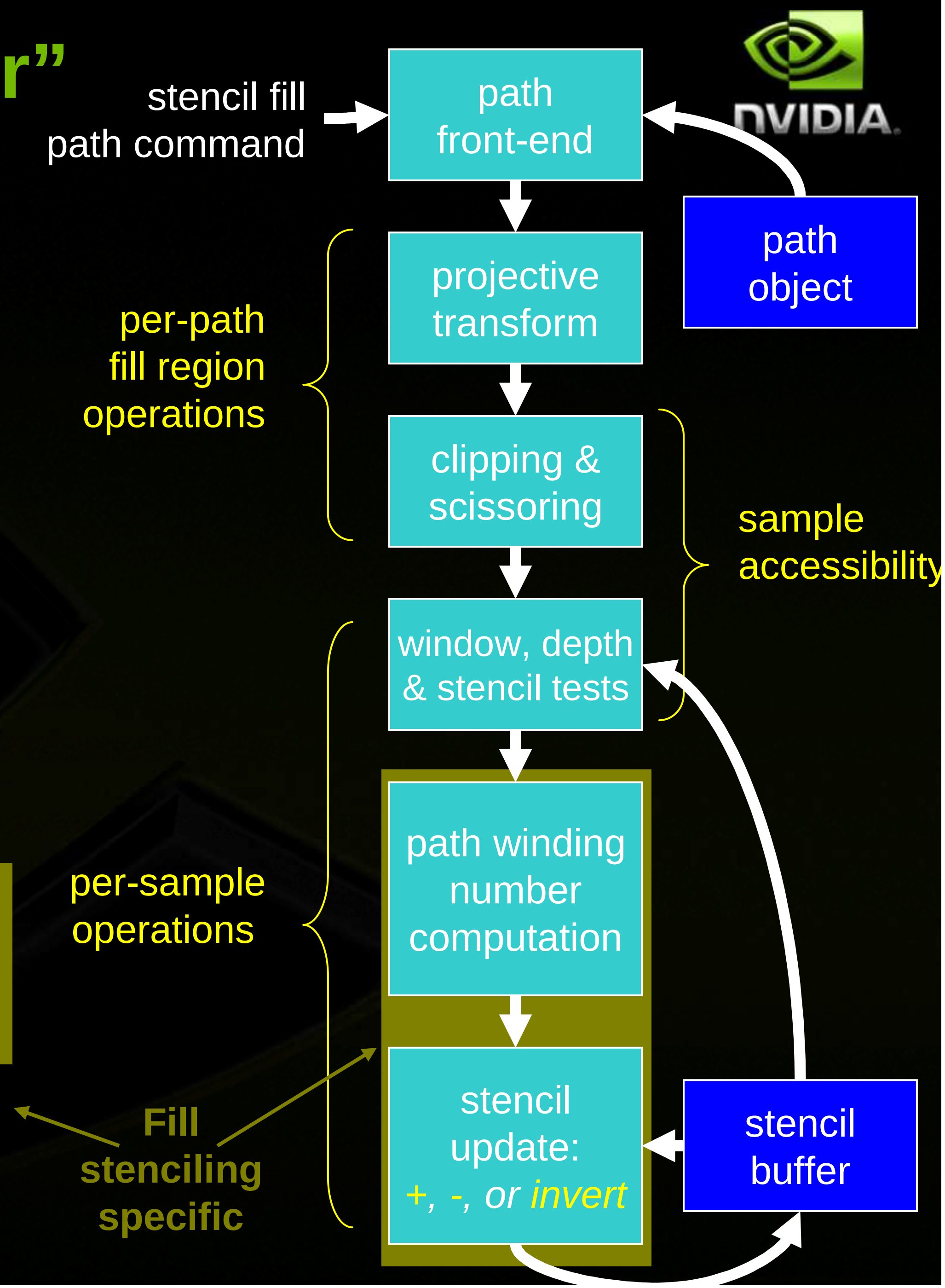
```
glStencilFunc(GL_NOTEQUAL, 0, 0x1); // alternative mask
```

non-zero fill style

even-odd fill style

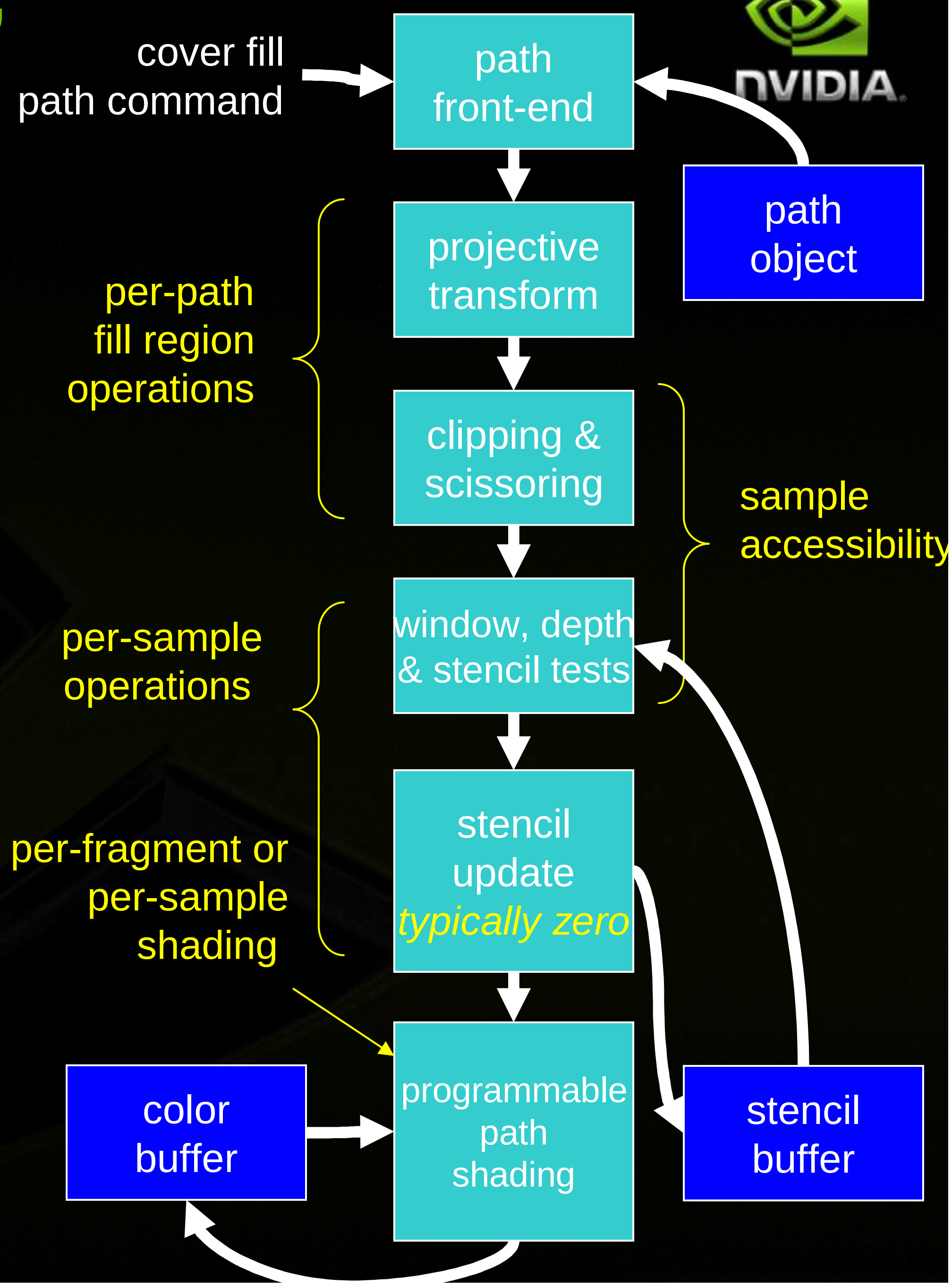
"Stencil, then Cover"
Path Fill Stenciling

- **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 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**



Adding Stroking to the Star

- ***After the filling, add a stroked "rim" to the star like this…***

non-zero fill style

- **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"
```

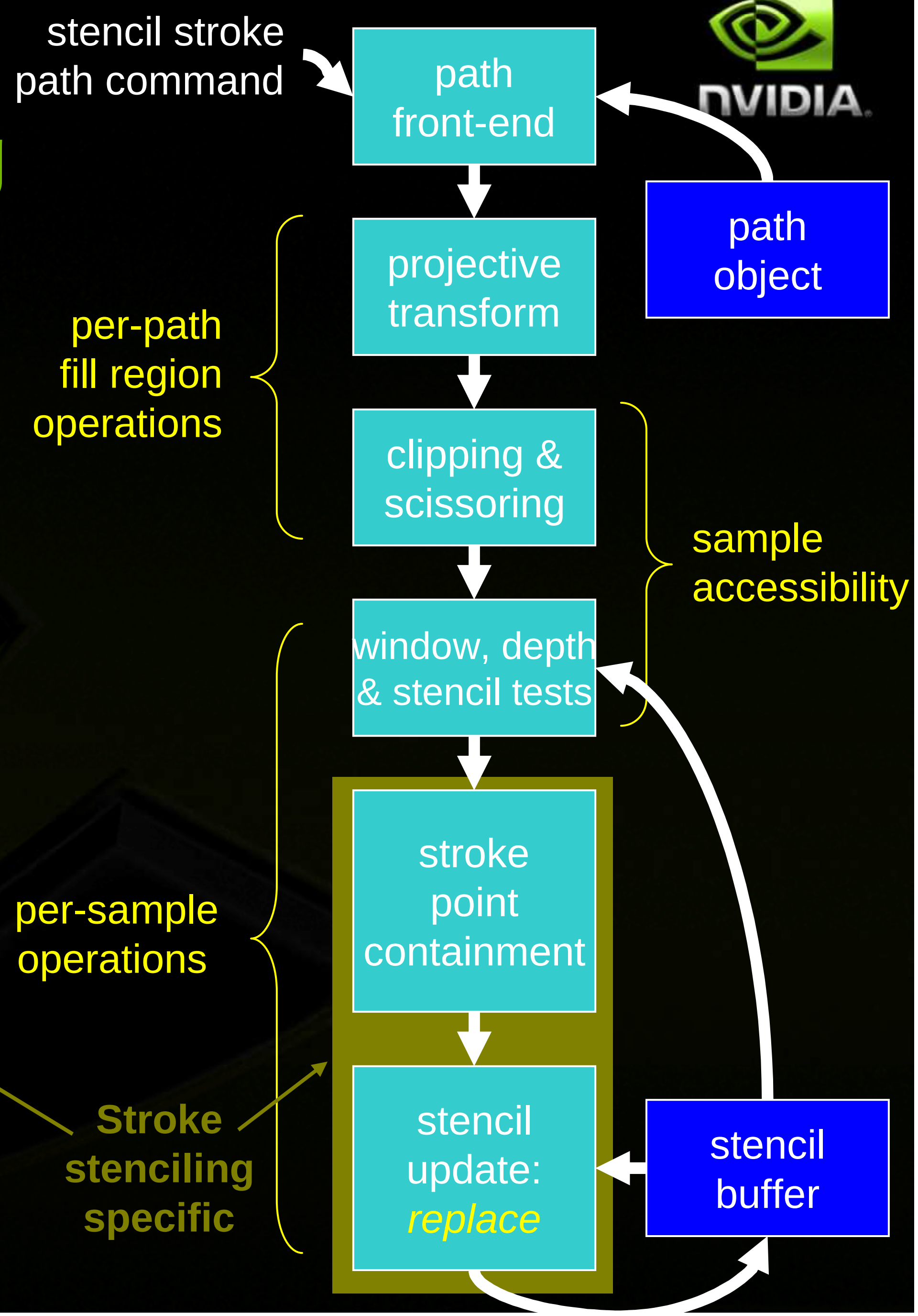
  - **Cover path**

```
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);
```

even-odd fill style

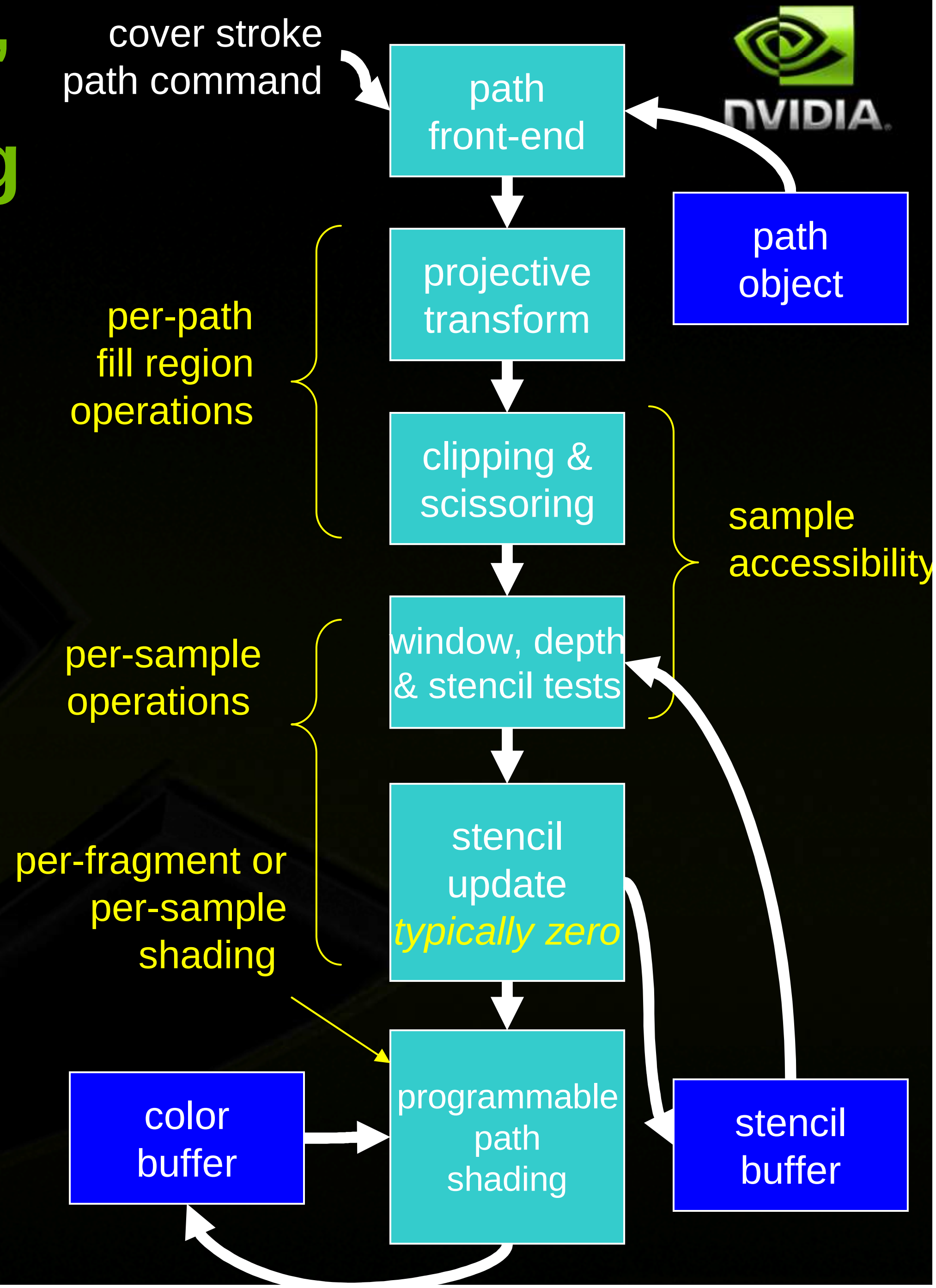
"Stencil, then Cover" Path Stroke Stenciling

- **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, 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**



Path Object State

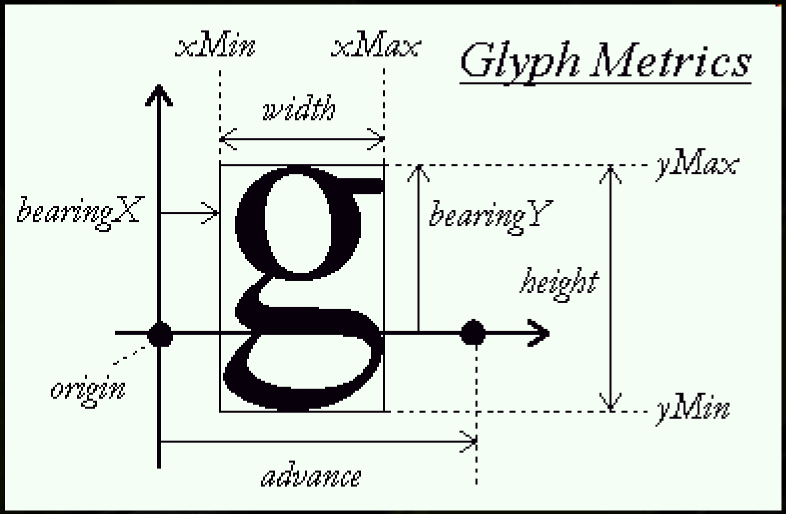
- Path commands
  - Unbounded number of commands allowed
- Path coordinates
  - Match up with commands
  - Example: each cubic Bezier segments has 6 coordinates
    - (x1,y1), (x2,y2), (x3,y3)
    - Initial control point (x0,y0) is implicit based on prior path command's end-point
- Path parameters
  - Stroke width, end caps, join styles, dash pattern, etc.
- Glyph metrics
  - When path object is created from a font

Path Object Queries

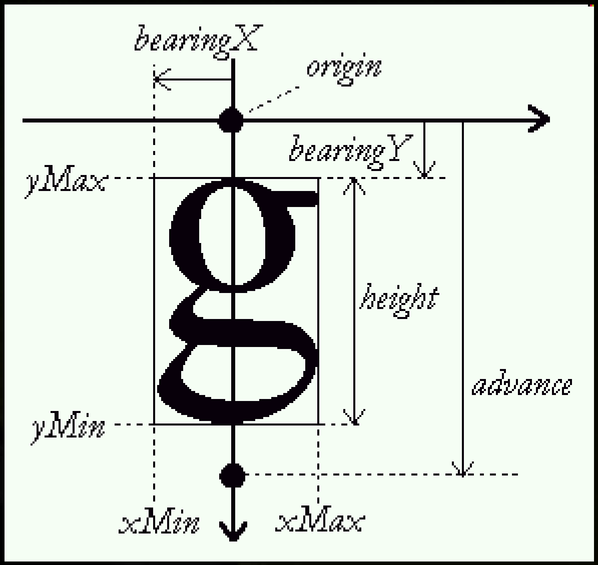
- All settable path object state is queriable
  - just like all conventional OpenGL state
- glGetPathParameter{i,f}vNV
- glGetPathParameter{i,f}NV
- glGetPathCommandsNV
- glGetPathCoordsNV
- Can also query *derived* state of path objects
  - GL_PATH_COMMAND_COUNT_NV
  - GL_PATH_COORD_COUNT_NV
  - GL_DASH_ARRAY_COUNT_NV
  - 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**



Horizontal metrics



Vertical metrics

Image credit: FreeType 2 Tutorial

Per-Glyph Metric Names

Bit field name	Glyph metric tag	Bit number from LSB in bitmask	Description
GL_GLYPH_WIDTH_BIT_NV	*width*	0	Glyph's width
GL_GLYPH_HEIGHT_BIT_NV	*height*	1	Glyph's height
GL_GLYPH_HORIZONTAL_BEARING_X_BIT_NV	*hBearingX*	2	Left side bearing for horizontal layout
GL_GLYPH_HORIZONTAL_BEARING_Y_BIT_NV	*hBearingY*	3	Top side bearing for horizontal layout
GL_GLYPH_HORIZONTAL_BEARING_ADVANCE_BIT_NV	*hAdvance*	4	Advance width for horizontal layout
GL_GLYPH_VERTICAL_BEARING_X_BIT_NV	*vBearingX*	5	Left side bearing for vertical layout
GL_GLYPH_VERTICAL_BEARING_Y_BIT_NV	*vBearingY*	6	Top side bearing for vertical layout
GL_GLYPH_VERTICAL_BEARING_ADVANCE_BIT_NV	*vAdvance*	7	Advance height for vertical layout
GL_GLYPH_HAS_KERNING_NV	-	8	True if glyph has a kerning table

Per-Font Face Metric Names

Bit field name	Bit number from LSB in bitmask	Description
GL_FONT_X_MIN_BOUNDS_NV	16	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.
GL_FONT_Y_MIN_BOUNDS_NV	17	Vertical minimum (bottom-most) of the font bounding box.
GL_FONT_X_MAX_BOUNDS_NV	18	Horizontal maximum (right-most) of the font bounding box.
GL_FONT_Y_MAX_BOUNDS_NV	29	Vertical maximum (top-most) of the font bounding box.
GL_FONT_UNITS_PER_EM_NV	20	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.
GL_FONT_ASCENDER_NV	21	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.
GL_FONT_DESCENDER_NV	22	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.
GL_FONT_HEIGHT_NV	23	Vertical distance between two consecutive baselines in the font face (always a positive value).
GL_FONT_MAX_ADVANCE_WIDTH_NV	24	Maximal advance width for all glyphs in this font face. (Intended to make word wrapping computations easier.)
GL_FONT_MAX_ADVANCE_HEIGHT_NV	25	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.
GL_FONT_UNDERLINE_POSITION_NV	26	Position of the underline line for this font face. This position is the center of the underling stem.
GL_FONT_UNDERLINE_THICKNESS_NV	27	Thickness of the underline of this font face.
GL_FONT_HAS_KERNING_NV	28	True if font face provides a kerning 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
- glGetPathSpacingNV 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

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

Filling specific parameters

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

Instanced Stroking Function Prototypes

- **Instanced Filling**

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

Stroking specific parameters (refers to `reference` and `mask`)

```
void glCoverStrokePathInstancedNV(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 <u>all</u> 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**

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

- **Load glyphs for a sequence of characters**

```
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**

```
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);
```

Font API Example: Initialization

- Allocate unused path object range for glyphs

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

- Load glyphs for a sequence of characters

```
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

```
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);
```

Font API Example: Pre-rendering

- Simple horizontal layout

```
const char *text = “OpenGL";

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

- Query per-font face metrics

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

- Initialize canvas-to-window transform

```
glMatrixLoadIdentityEXT(GL_PROJECTION);
glMatrixOrthoEXT(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**

```
glutSwapBuffers();
```

- **Clean up**

```
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 emScale = 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

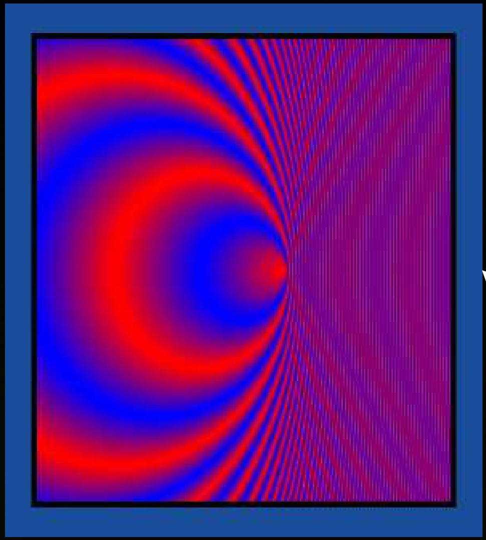
```
glPathGlyphRangeNV(glyphBase,
                   GL_SYSTEM_FONT_NAME_NV, "Arial", GL_BOLD_BIT_NV,
                   /*first character*/0, /*count*/unicodeRange,
                   GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
glPathGlyphRangeNV(glyphBase,
                   GL_STANDARD_FONT_NAME_NV, "Sans", GL_BOLD_BIT_NV,
                   /*first character*/0, /*count*/unicodeRange,
                   GL_USE_MISSING_GLYPH_NV, templatePathObject, emScale);
```

Naming Sequences of Path Objects

- 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

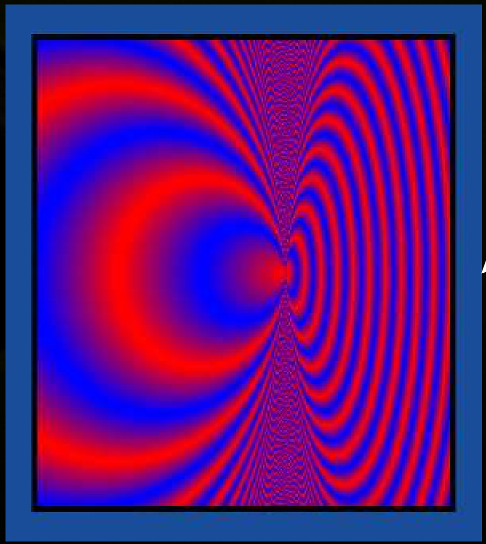
Handling Common Path Rendering Functionality: Filtering

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

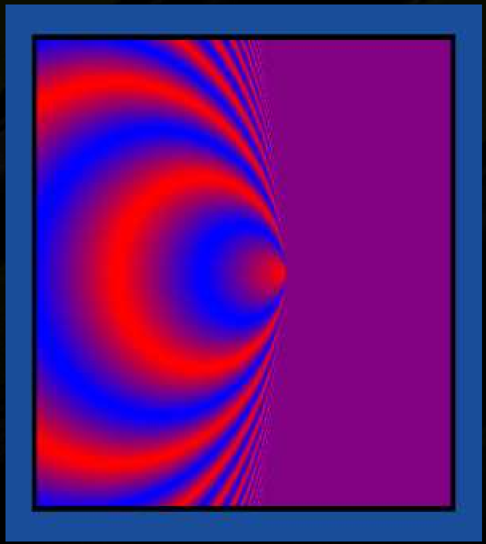


☒ Qt

Moiré artifacts



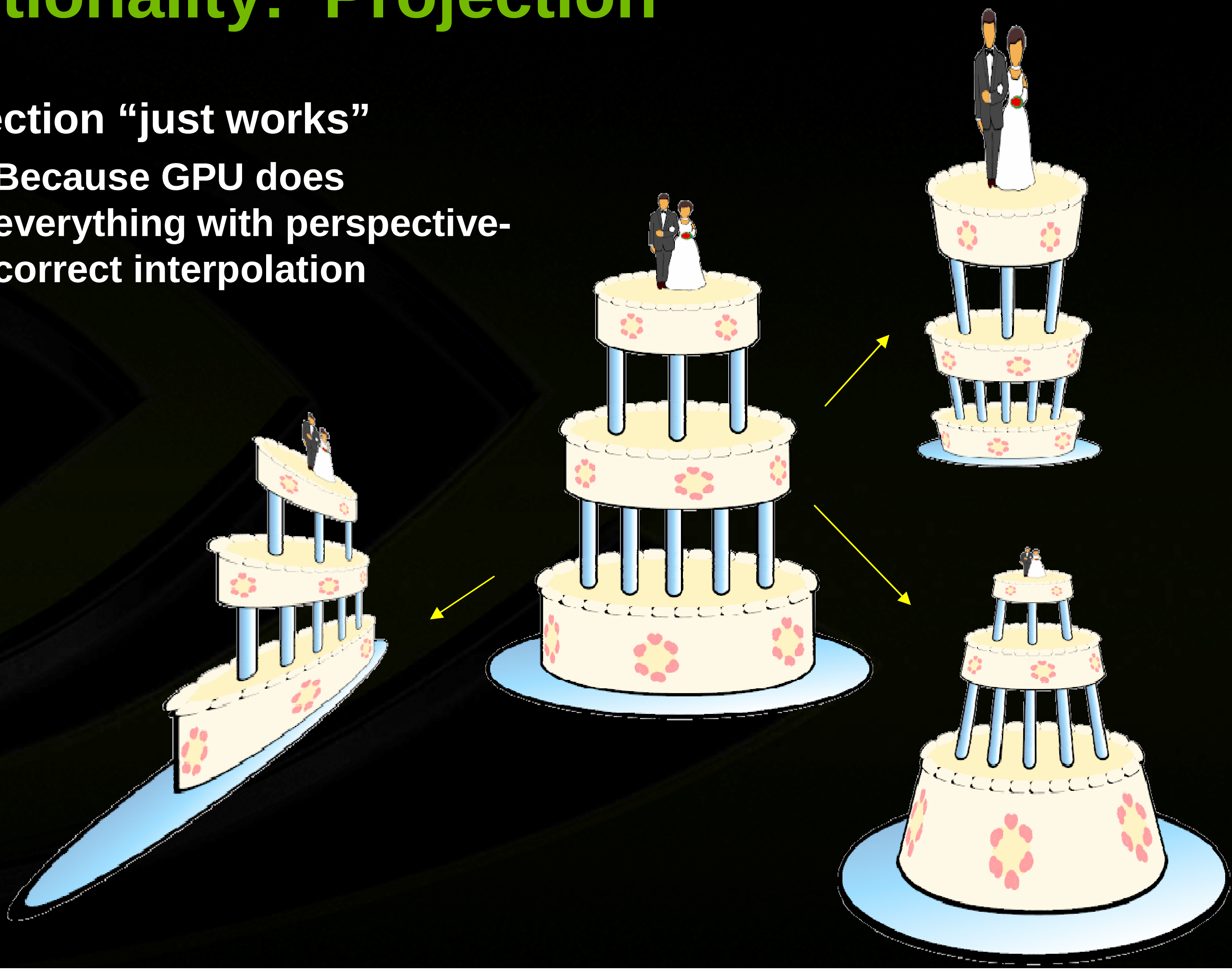
☒ Cairo



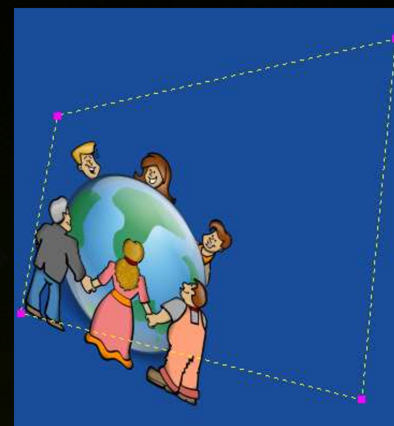
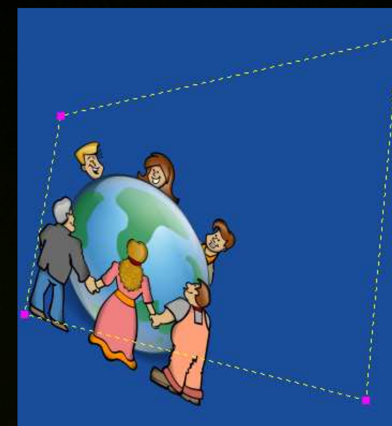
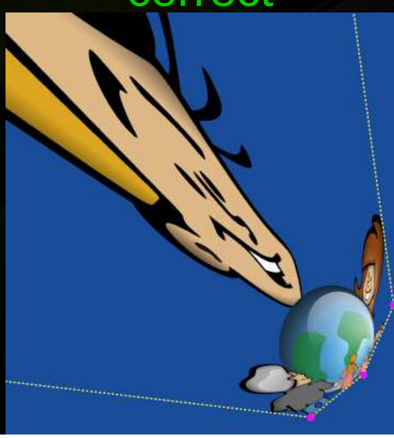
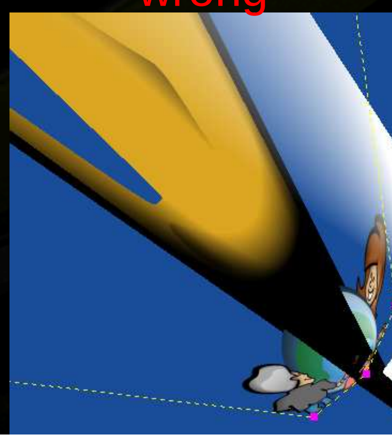
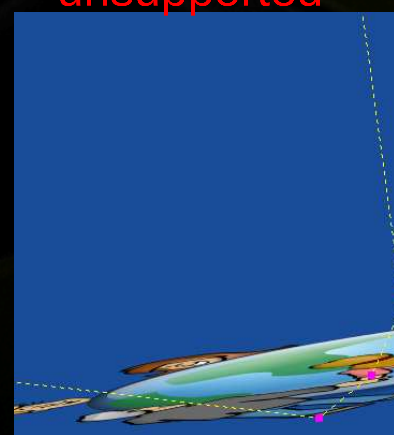
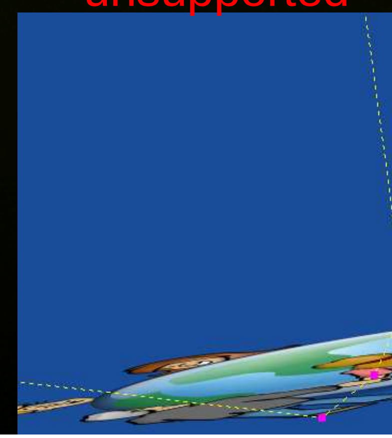
☑ GPU

Handling Uncommon Path Rendering Functionality: Projection

- **Projection "just works"**
  - **Because GPU does everything with perspective-correct interpolation**

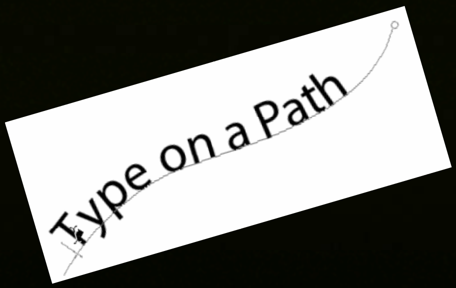


Projective Path Rendering Support Compared

☑ GPU	☹ Skia	☒ Cairo	☒ Qt
flawless	yes, but bugs	unsupported	unsupported
			
correct	correct	unsupported	unsupported
correct	wrong	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);
```

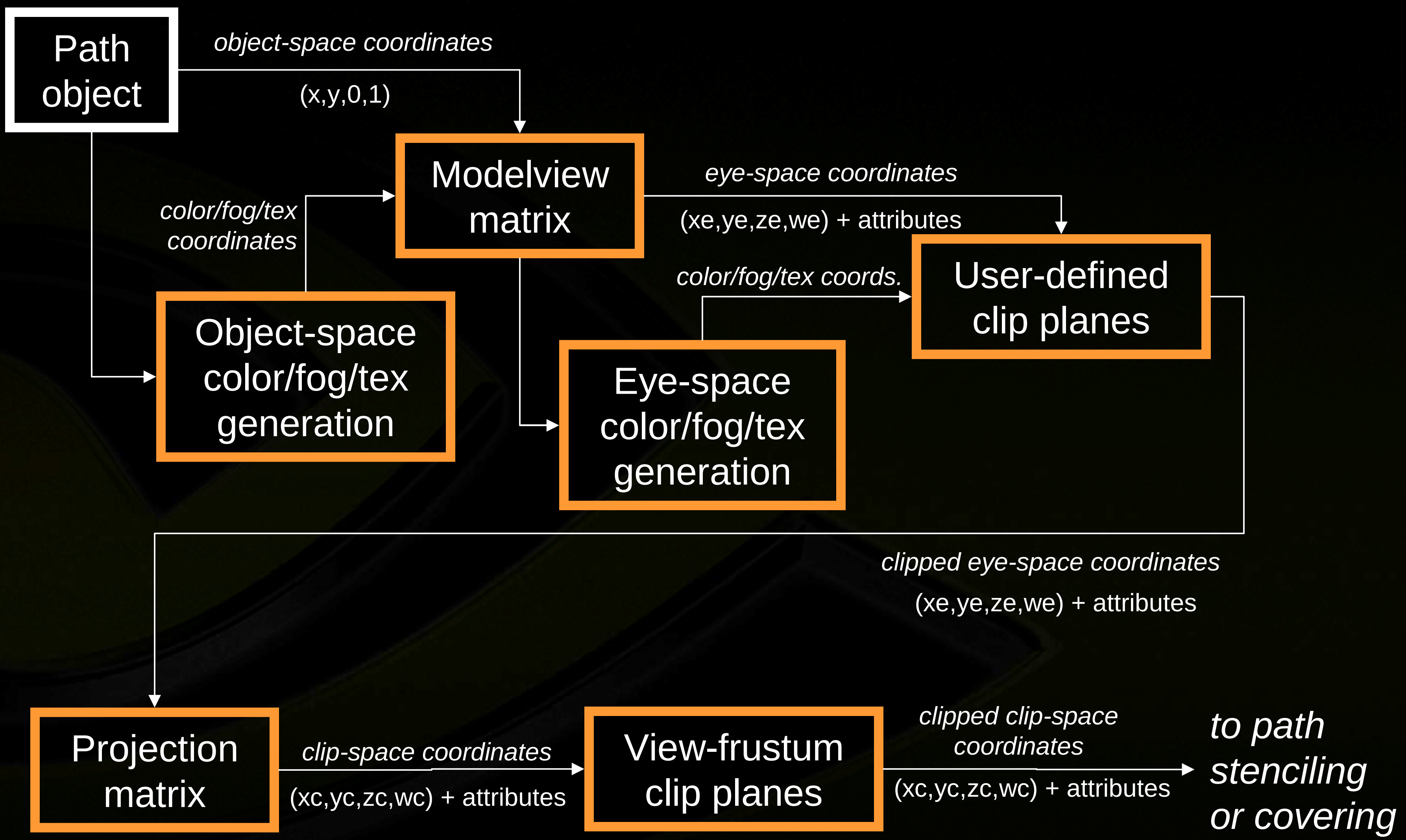
- Observation
  - Stencil step is testing—but not writing—depth
    - Stencil won't be updated if stencil step fails depth test at a sample
  - Cover step is writing—but not testing—depth
    - Cover step doesn't need depth test because stencil test would only pass if prior stencil step's depth test 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**

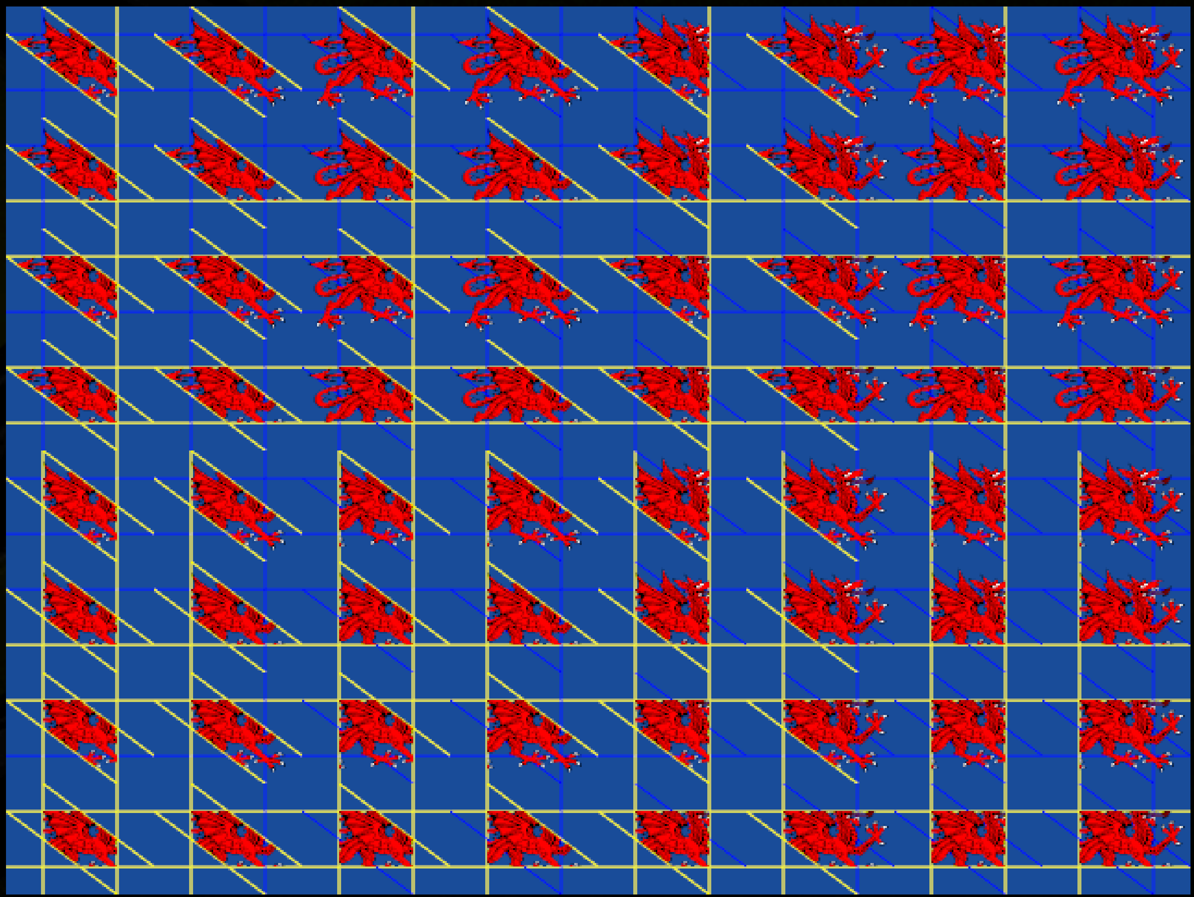
terrible z-fighting artifacts

Path Transformation Process



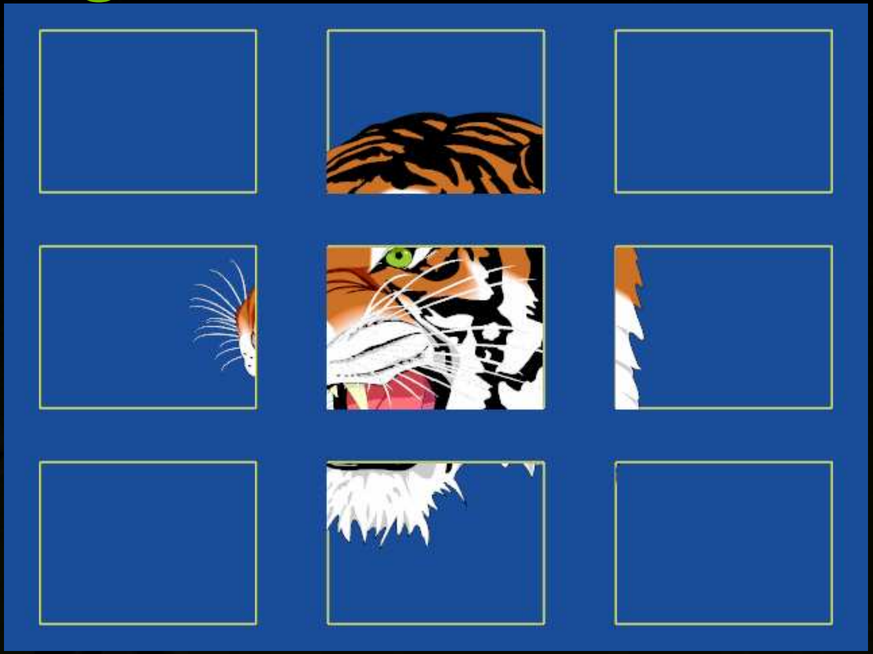
Clip Planes Work with Path Rendering

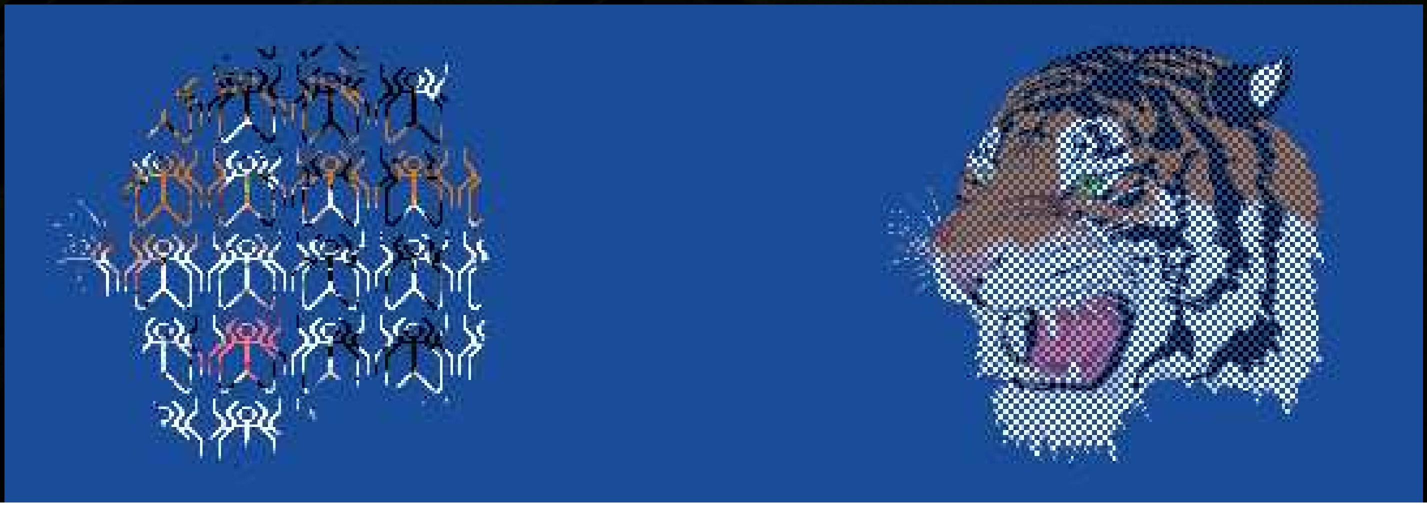
- **Scene showing a Welsh dragon clipped to all 64 combinations of 6 clip planes enabled & disabled**



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

tiger with pink background clipped to heart

Complex Clipping Example

tiger is 240 paths

cowboy clip is
the union of 1,366 paths

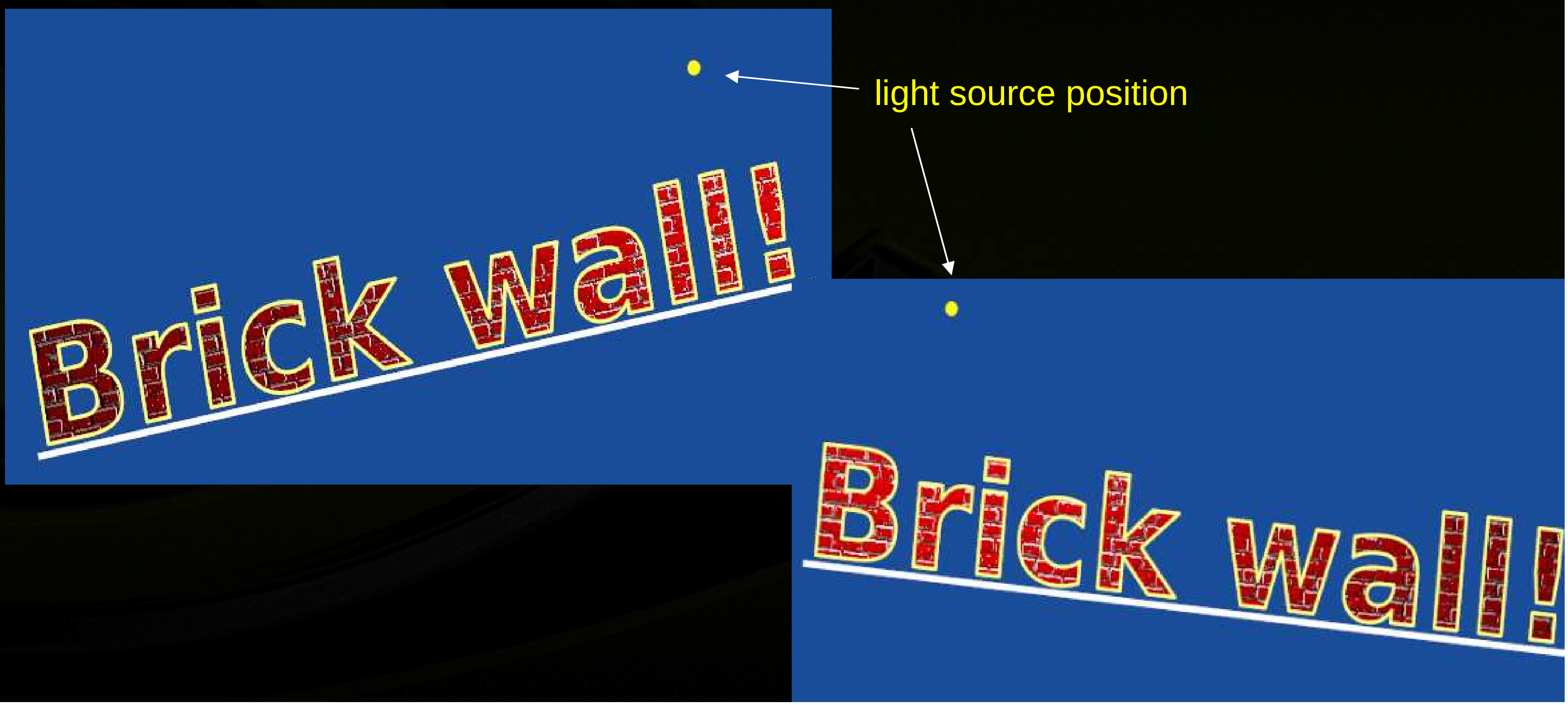
result of clipping tiger
to the union of all the cowboy paths

Arbitrary Programmable GPU Shading with Path Rendering

- 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 <u>ignored</u> 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**



Antialiasing Discussion

- 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

artifacts

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

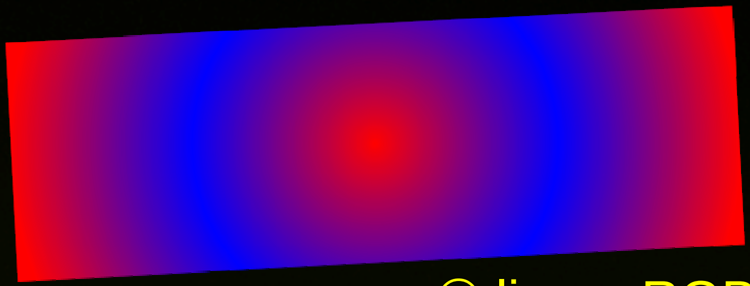
GPU Advantages

- 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

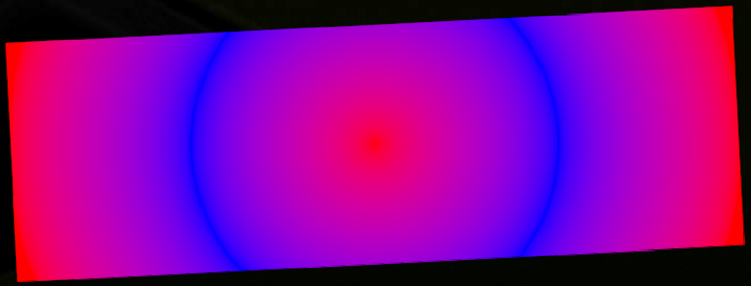
- **Modern GPUs have native support for perceptually-correct for**
  - **sRGB framebuffer blending**
  - **sRGB texture filtering**
  - **No reason to tolerate uncorrected linear RGB color artifacts!**
  - **More intuitive for artists to control**
- **Negligible expense for GPU to perform sRGB-correct rendering**
  - **However <u>quite</u> 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

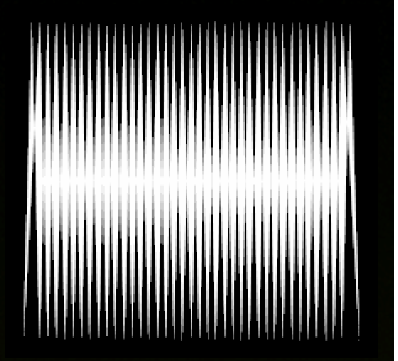
Benchmark Scenes

Tiger

Dragon

Round Dogs

Butterfly



Spikes

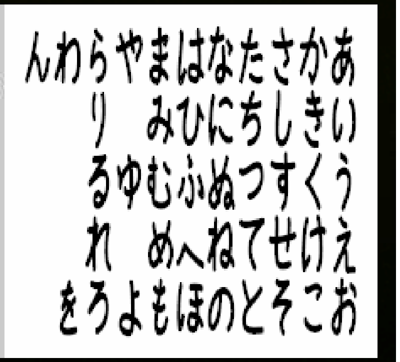
Coat of Arms

Cowboy

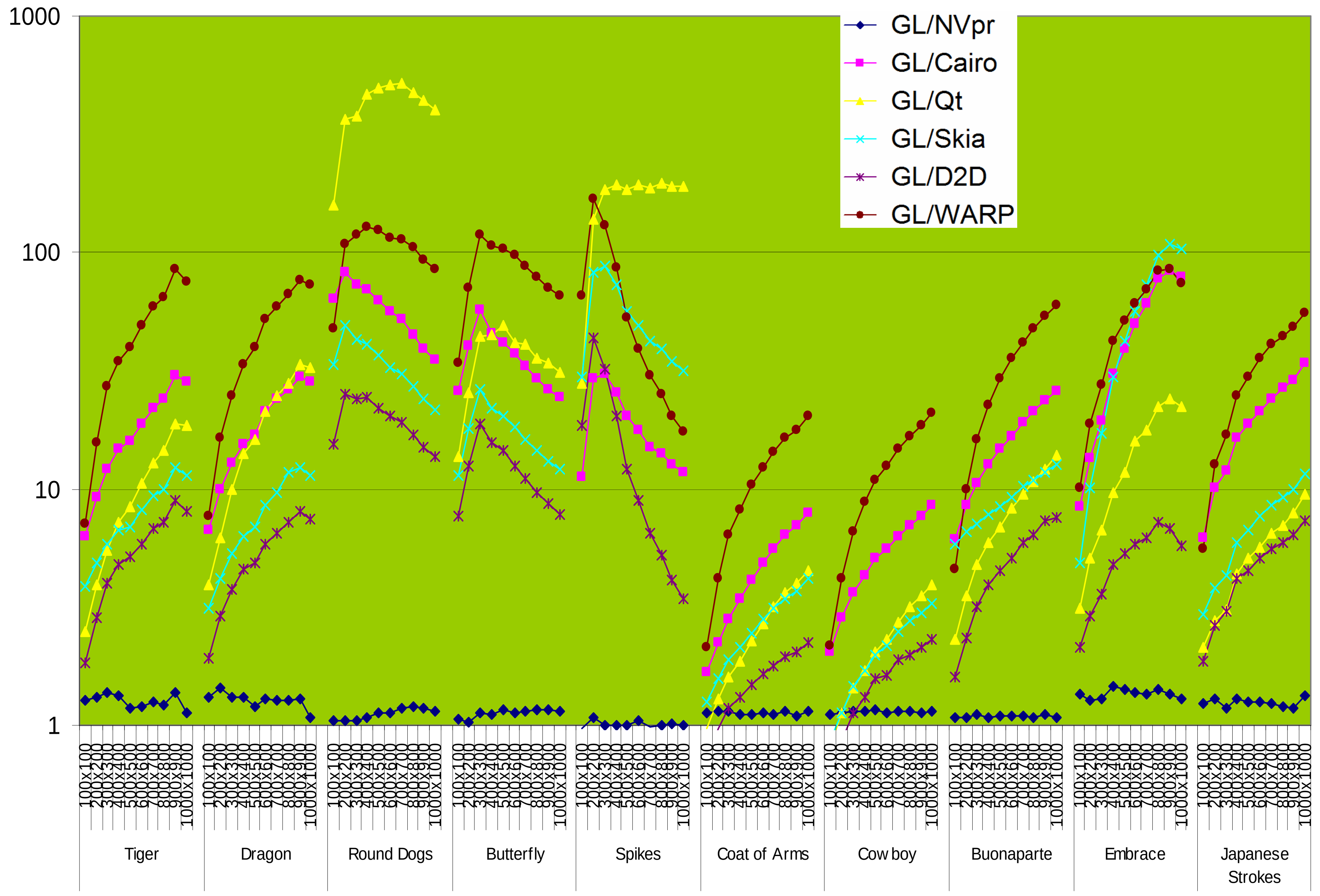
Buonaparte

Embrace

Japanese Strokes



GPU-accelerated Path Rendering Speed Factor



Benchmark Test Configuration & Assumptions

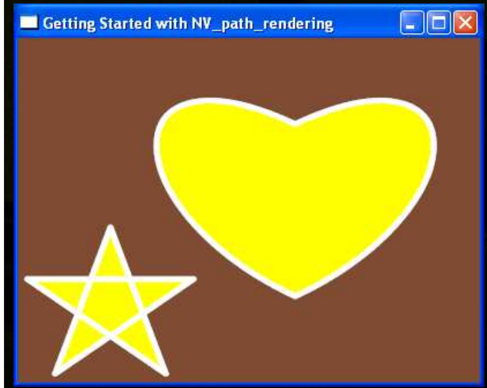
- CPU
  - 2.9 GHz i3 Nehalem
  - Only using a single-core
- GPU
  - Fermi GTX 480, so assuming fastest available GPU
  - 16 samples/pixel
- Ten window resolutions
  - 100x100 (lowest) to 1,000x1,000 (highest)
  - In 100 pixel increments
- Ten test scenes
  - Variety of path complexity, stroking vs. filling, 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**
  - ***http://www.nvidia.com/object/winxp-275.27-beta-driver.html***
  - ***GeForce 275.33 driver now public***
    - ***http://www.nvidia.com/object/winxp-275.33-whql-driver.html***
  - ***We need your feedback***

Learning NV_path_rendering

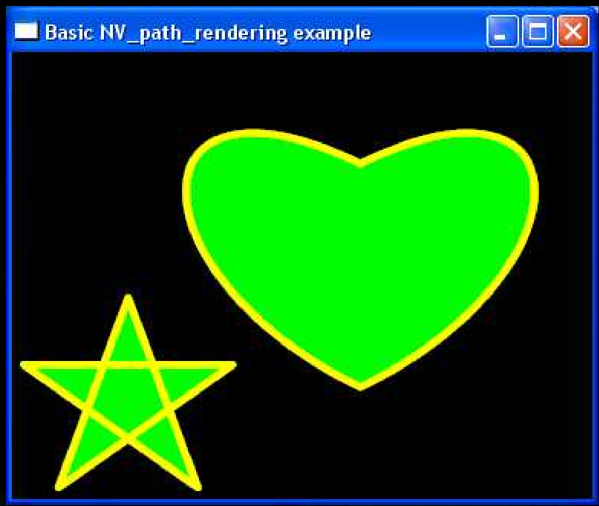
- White paper + source code available
  - "Getting Started with NV_path_rendering"
- Explains
  - Path specification
  - "Stencil, then Cover" API usage
  - Instanced rendering for text and glyphs



NV_path_rendering SDK Examples

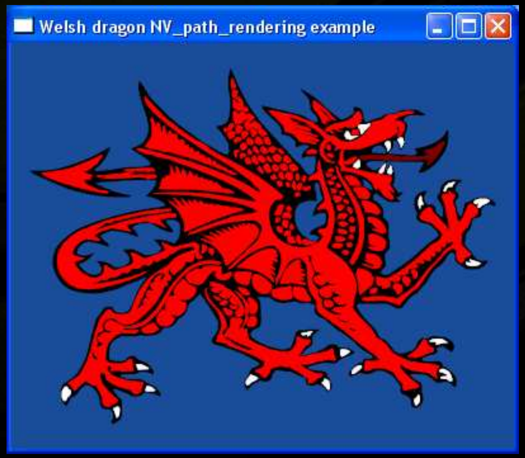
- A set of NV_path_rendering examples of varying levels of complexity
  - Most involved example is an accelerated SVG viewer
    - Not a complete SVG implementation
- Compiles on Windows and Linux
  - Needs Visual Studio 2008 for Windows

SDK Example Walkthrough (1)

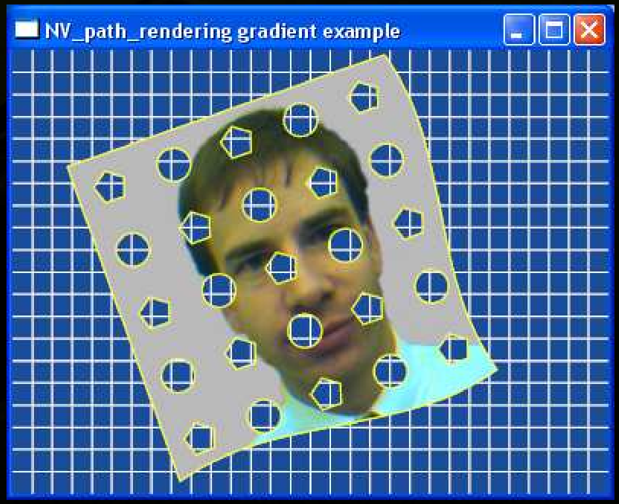


pr_basic: simplest example of path filling & stroking

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

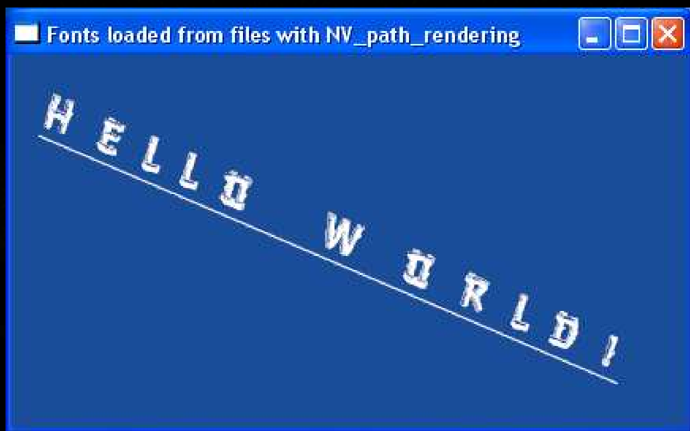


pr_welsh_dragon: filled layers



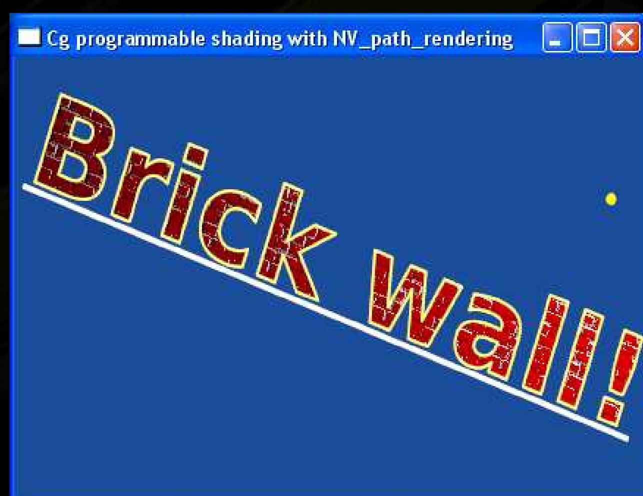
pr_gradient: path with holes with texture applied

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

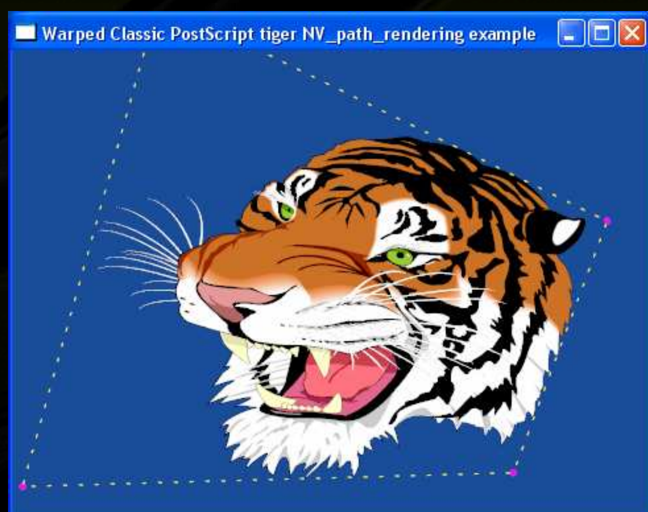


pr_shaders: use Cg shaders to bump map text with brick-wall texture

SDK Example Walkthrough (3)

pr_text_wheel: render projected gradient text as spokes of a wheel

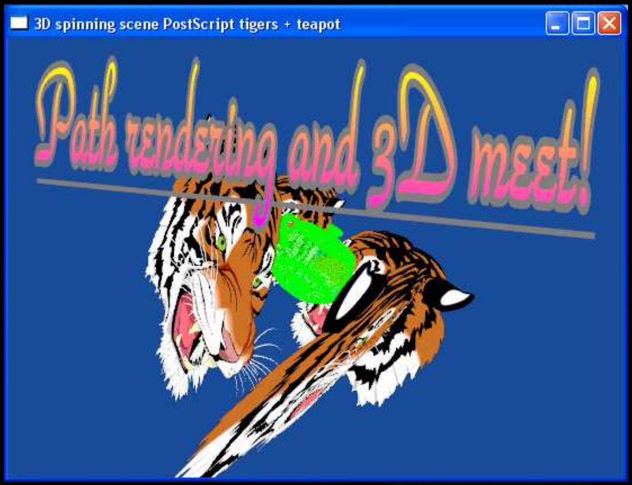
pr_tiger: classic PostScript tiger rendered as filled & stroked path layers



pr_warp_tiger: warp the tiger with a free projective transform

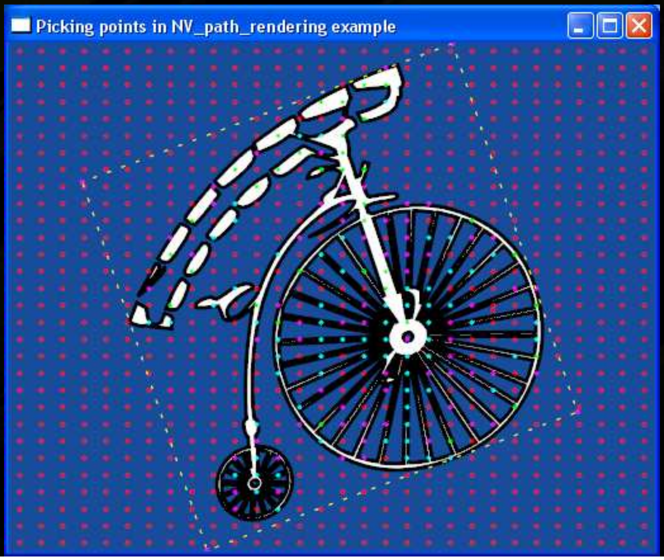
click & drag the bounding rectangle corners to change the projection

SDK Example Walkthrough (4)



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