GPU-Accelerated 2D and Web Rendering

Mark Kilgard

NVIDIA

Talk Details

Location: West Hall Meeting Room 503, Los Angeles Convention Center
Date: Wednesday, August 8, 2012
Time: 2:40 PM – 3:40 PM

Mark Kilgard (Principal Software Engineer, NVIDIA)

Abstract: The future of GPU-based visual computing integrates the web, resolution-independent 2D graphics, and 3D to maximize interactivity and quality while minimizing consumed power. See what NVIDIA is doing today to accelerate resolution-independent 2D graphics for web content. This presentation explains NVIDIA's unique "stencil, then cover" approach to accelerating path rendering with OpenGL and demonstrates the wide variety of web content that can be accelerated with this approach.

Topic Areas: GPU Accelerated Internet; Digital Content Creation & Film; Visualization

Level: Intermediate

Mark Kilgard

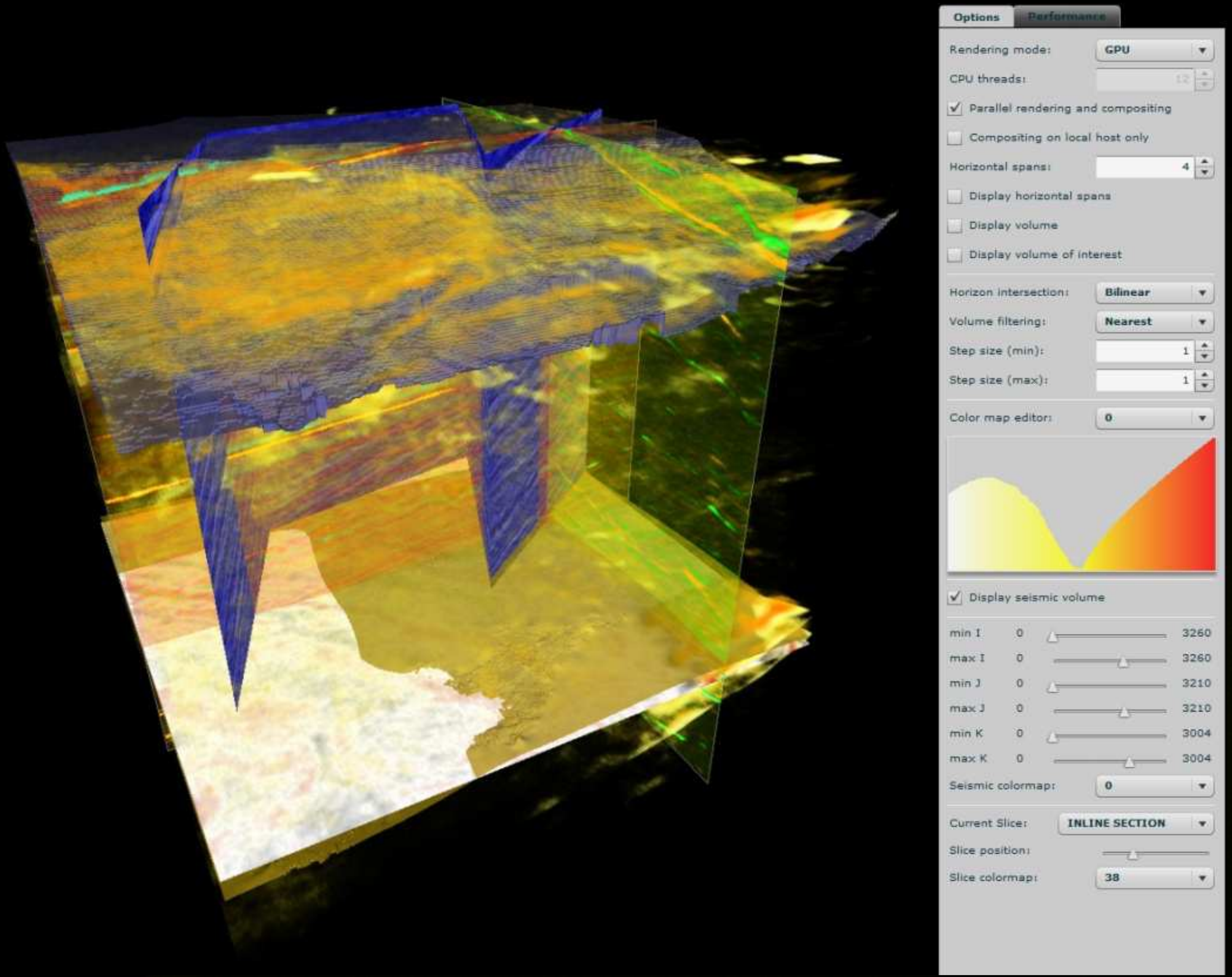
- Principal System Software Engineer
  - OpenGL driver and API evolution
  - Cg ("C for graphics") shading language
  - GPU-accelerated path rendering
- OpenGL Utility Toolkit (GLUT) implementer
- Author of *OpenGL for the X Window System*
- Co-author of *Cg Tutorial*

GPUs are good at a lot of stuff

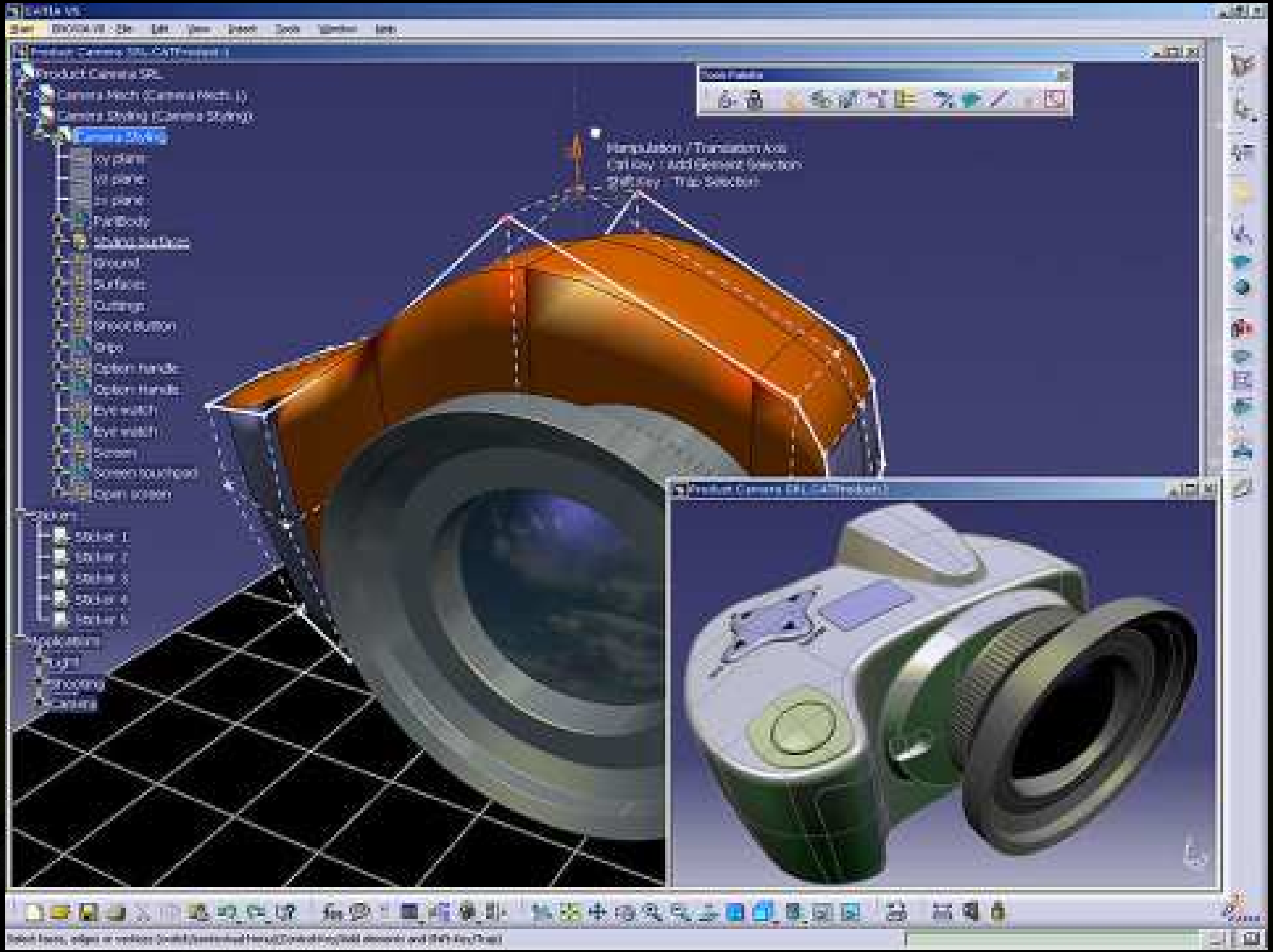
Games

Battlefield 3, EA

Data visualization

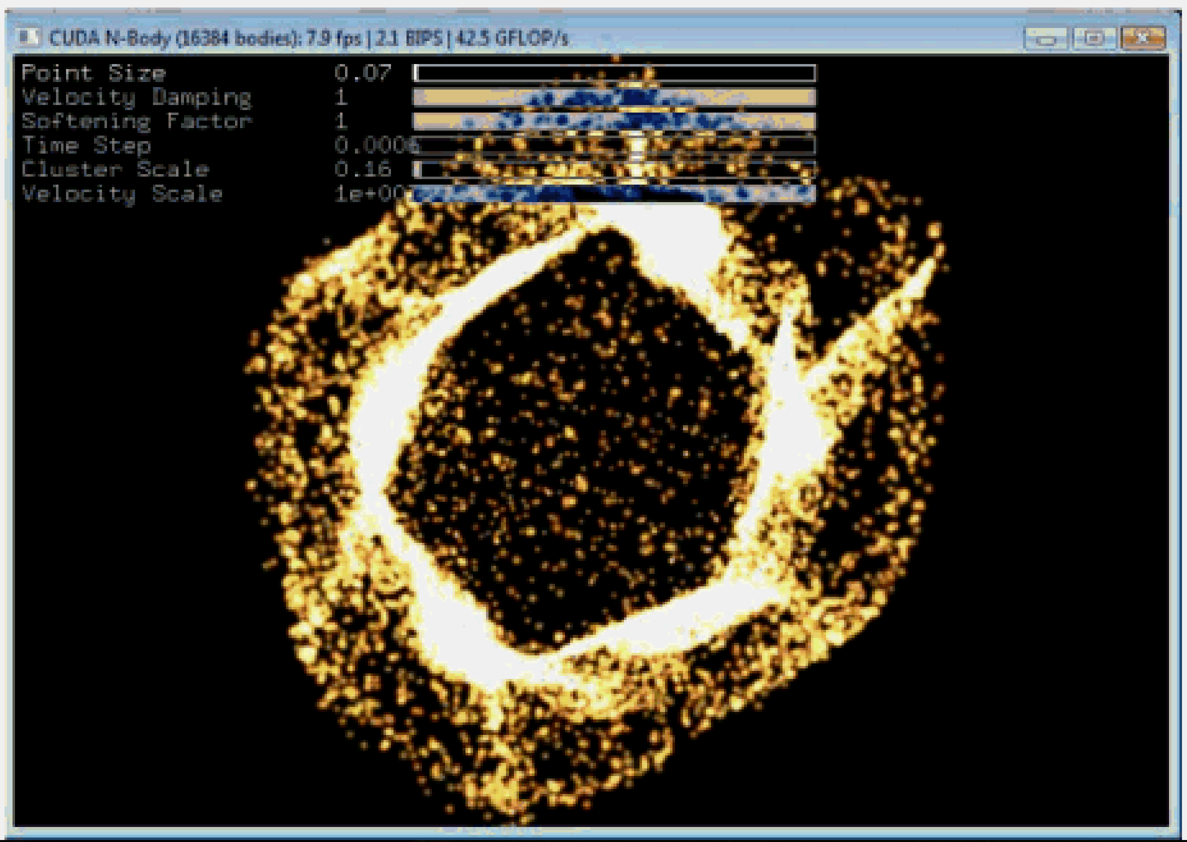


Product design



Catia

Physics simulation



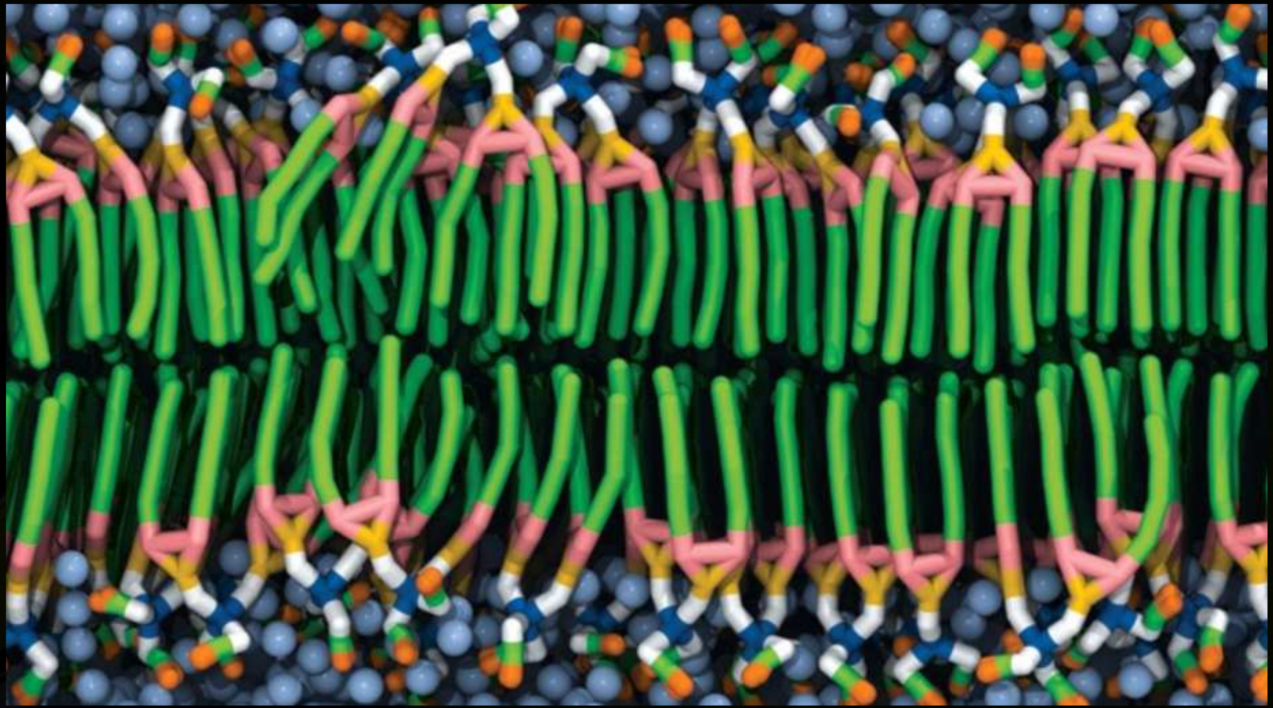
CUDA N-Body

Interactive ray tracing

OptiX

Training

Molecular modeling



NCSA

Impressive stuff

What about advancing 2D graphics?

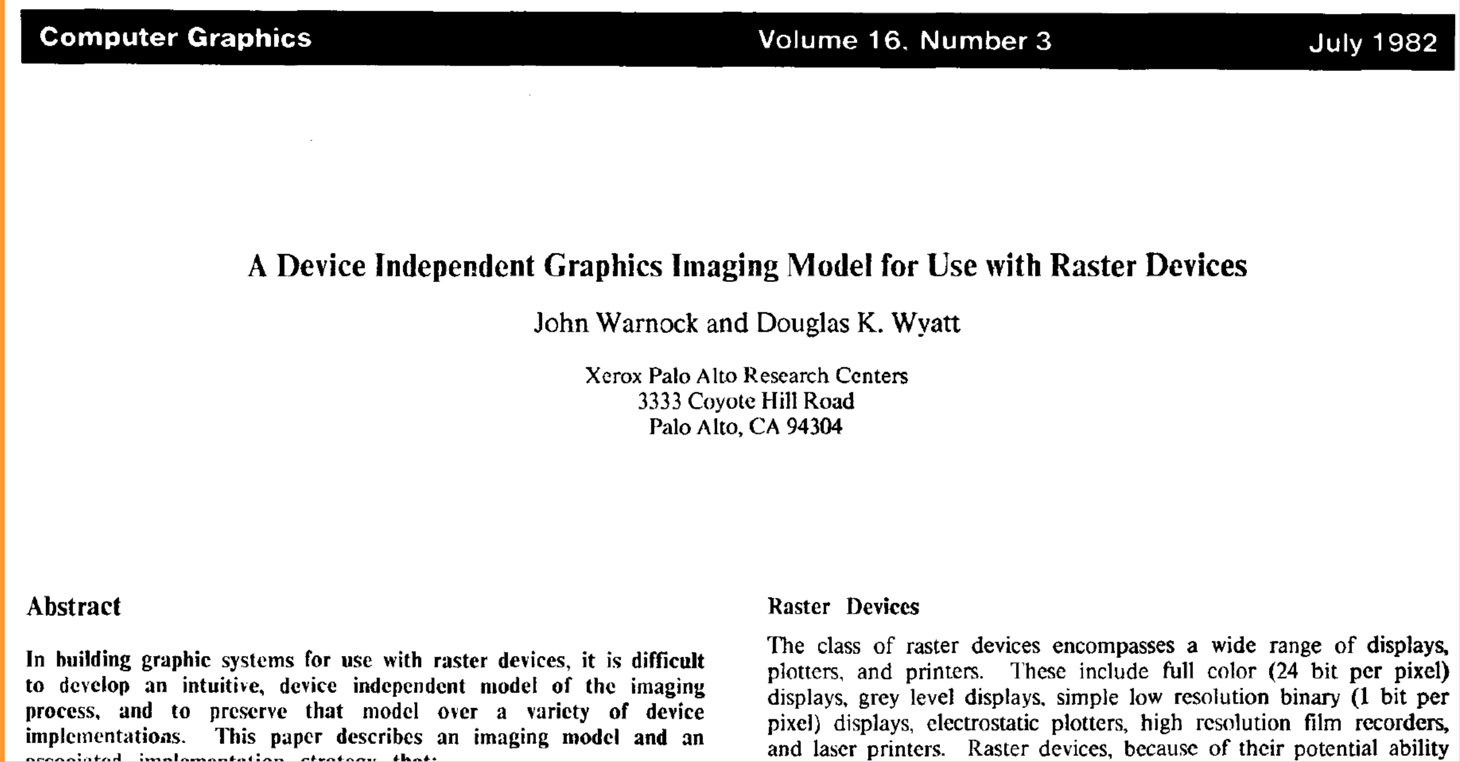
Can GPUs render & improve the immersive web?

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

Seminal Path Rendering Paper

- John Warnock & Douglas Wyatt, Xerox PARC
  - Presented SIGGRAPH 1982
  - Warnock founded Adobe months later



Computer Graphics Volume 16, Number 3 July 1982

A Device Independent Graphics Imaging Model for Use with Raster Devices

John Warnock and Douglas K. Wyatt

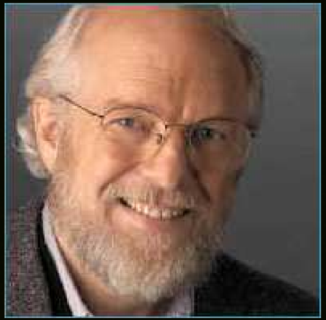
Xerox Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

In building graphic systems for use with raster devices, it is difficult to develop an intuitive, device independent model of the imaging process, and to preserve that model over a variety of device implementations. This paper describes an imaging model and an

Raster Devices

The class of raster devices encompasses a wide range of displays, plotters, and printers. These include full color (24 bit per pixel) displays, grey level displays, simple low resolution binary (1 bit per pixel) displays, electrostatic plotters, high resolution film recorders, and laser printers. Raster devices, because of their potential ability



John Warnock
Adobe founder

Path Rendering Standards

Document Printing and Exchange

- Open XML Paper (XPS)

Resolution-Independent Fonts

- OpenType
- TrueType

Immersive Web Experience

- Flash
- Scalable Vector Graphics
- HTML 5

2D Graphics Programming Interfaces

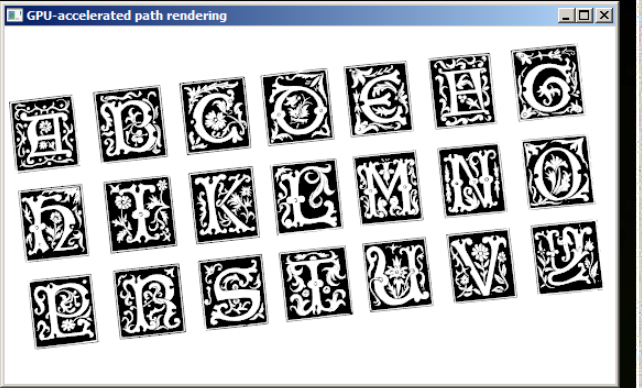
- Java 2D API
- QtGui API
- Mac OS X 2D API
- Khronos API

Office Productivity Applications

- Adobe Illustrator
- Inkscape Open Source

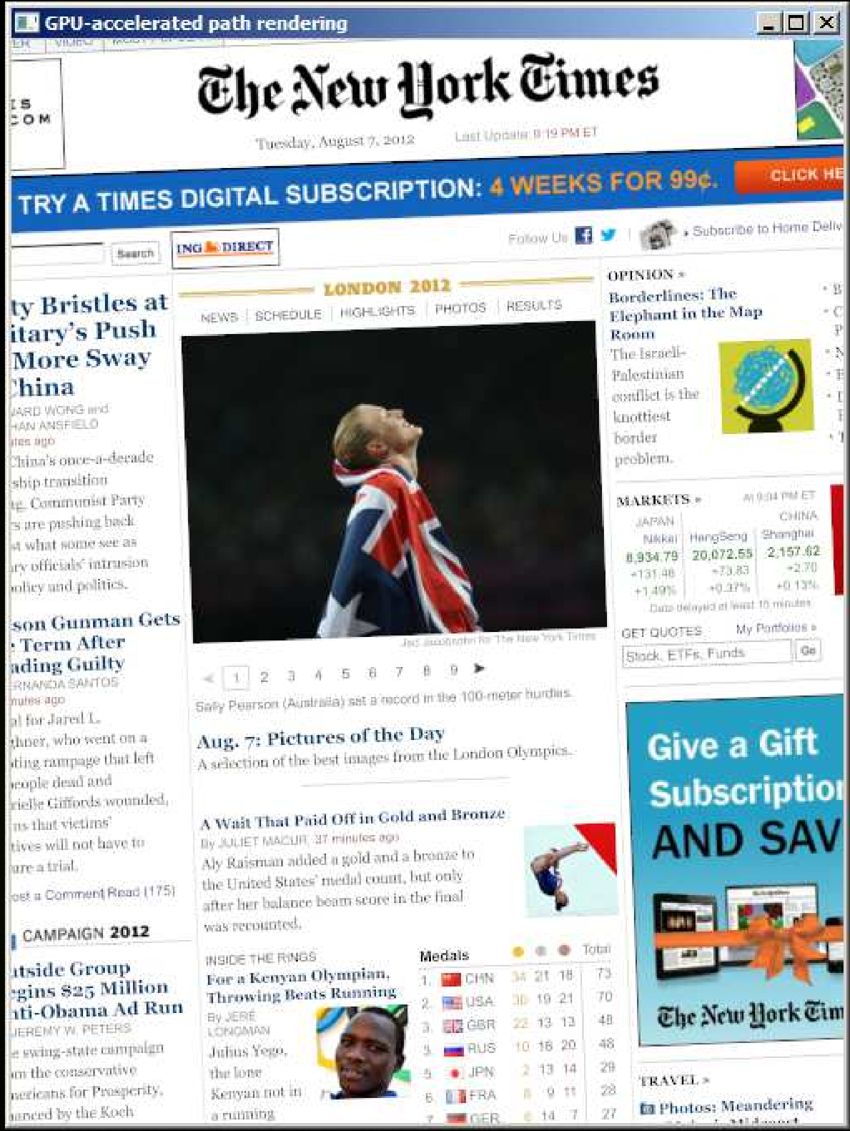
Live Demo

Classic PostScript content



Complex text rendering

Flash content

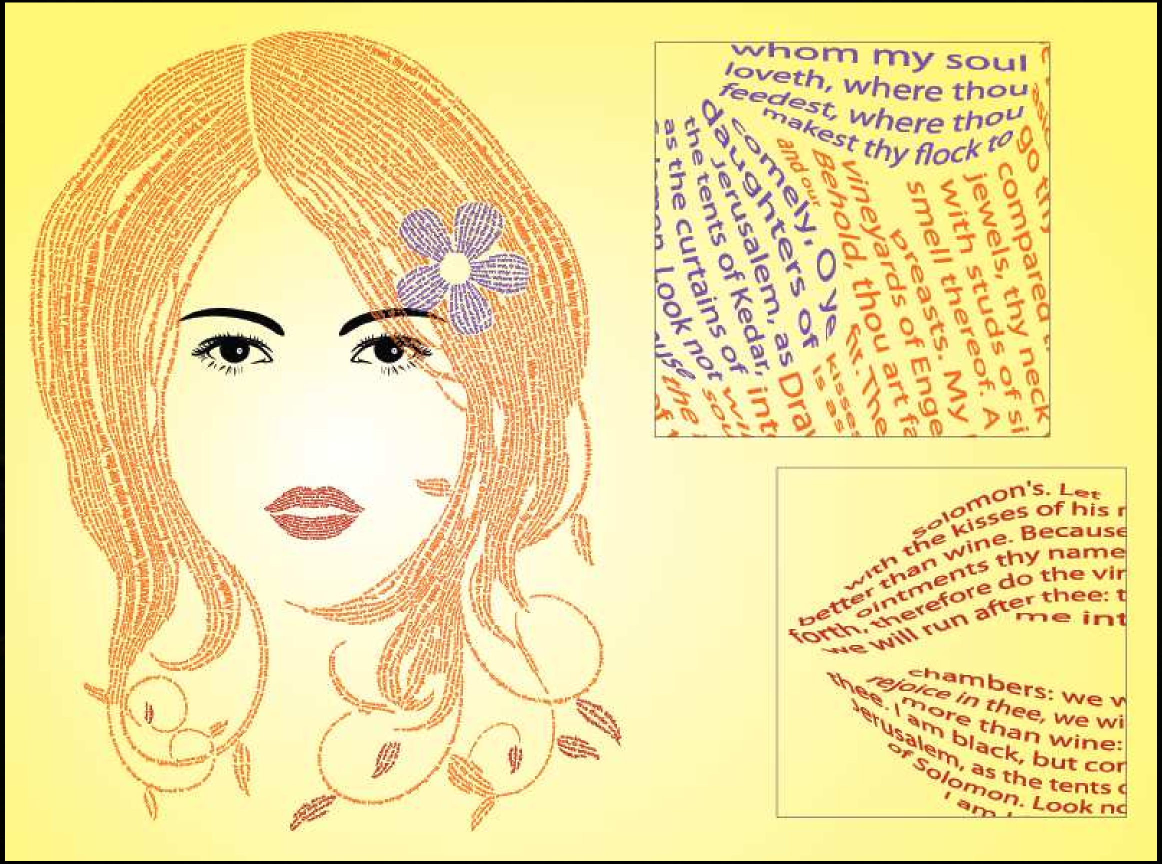


Yesterday’s New York Times rendered from its resolution-independent form

Last Year's SIGGRAPH Results in Real-time

- Ron Maharik, Mikhail Bessmeltsev, Alla Sheffer, Ariel Shamir and Nathan Carr
- SIGGRAPH 2011, July 2011

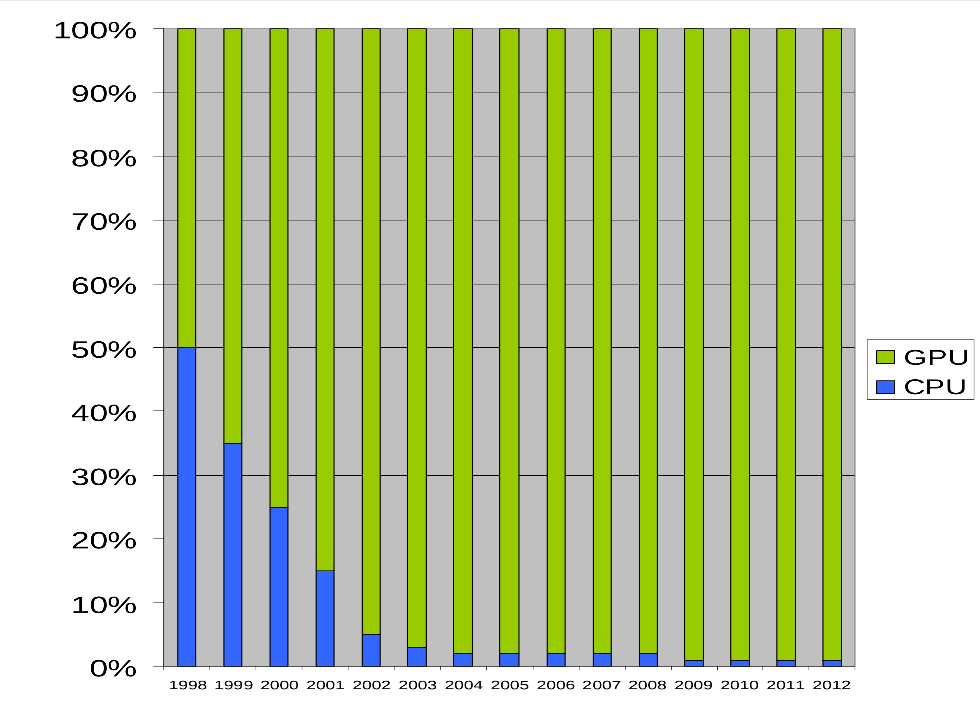
- "Girl with Words in Her Hair" scene
  - 591 paths
  - 338,507 commands
  - 1,244,474 coordinates



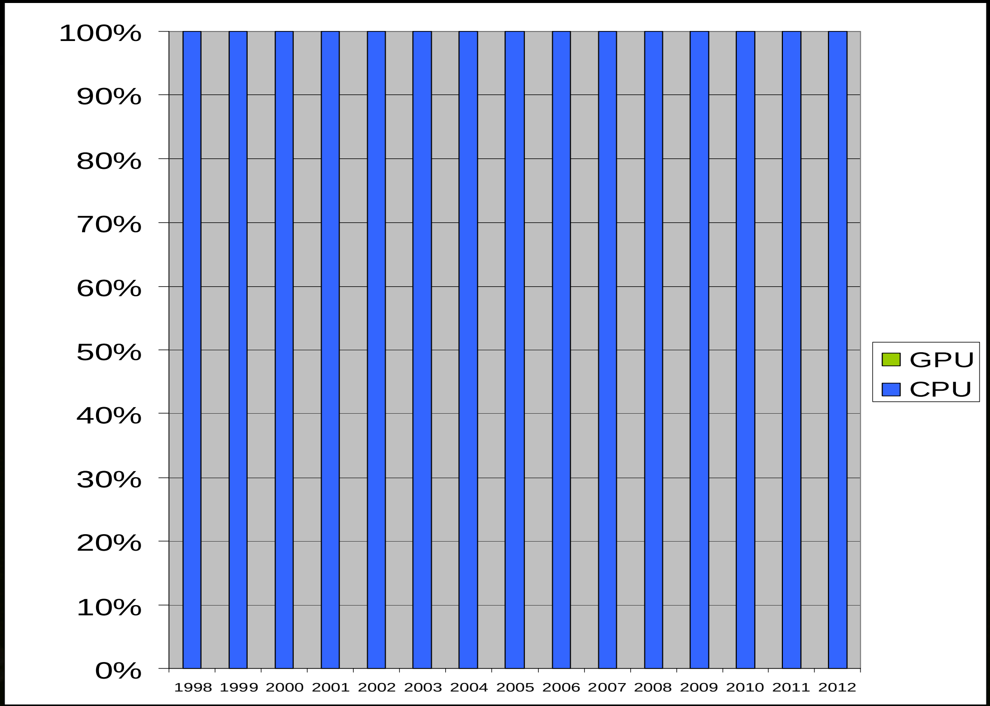
3D Rendering vs. Path Rendering

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

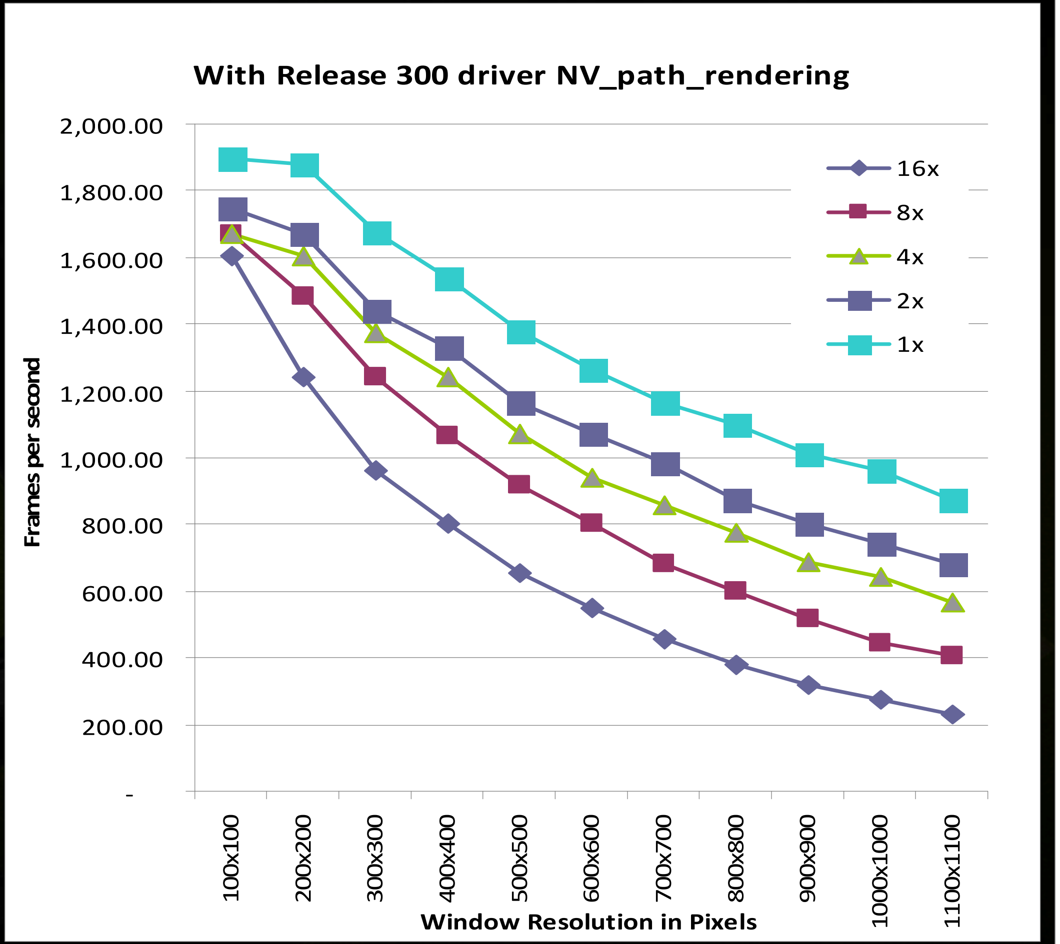
Render all interactive pixels, whether 3D or 2D or web content with the GPU

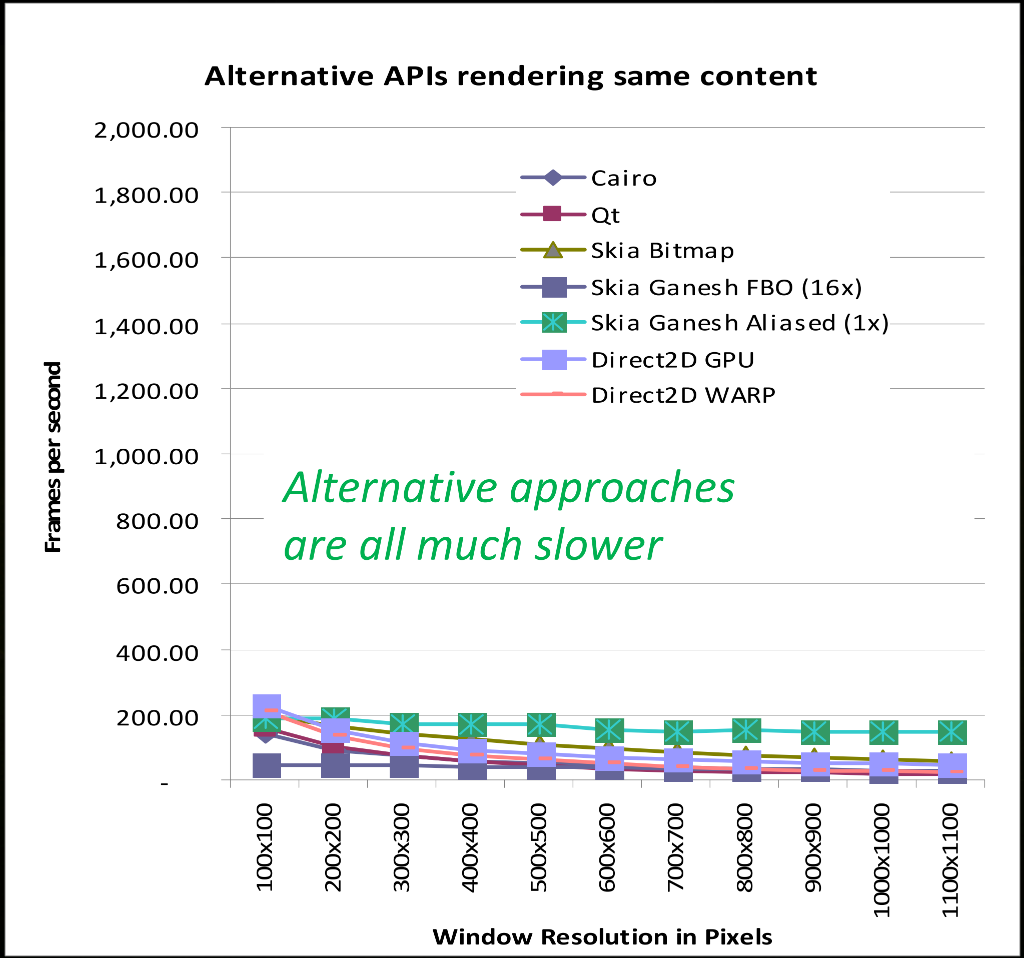
What is NV_path_rendering?

- 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 functionality to mix with traditional 3D and programmable shading

Configuration
GPU: GeForce 480 GTX (GF100)
CPU: Core i7 950 @ 3.07 GHz

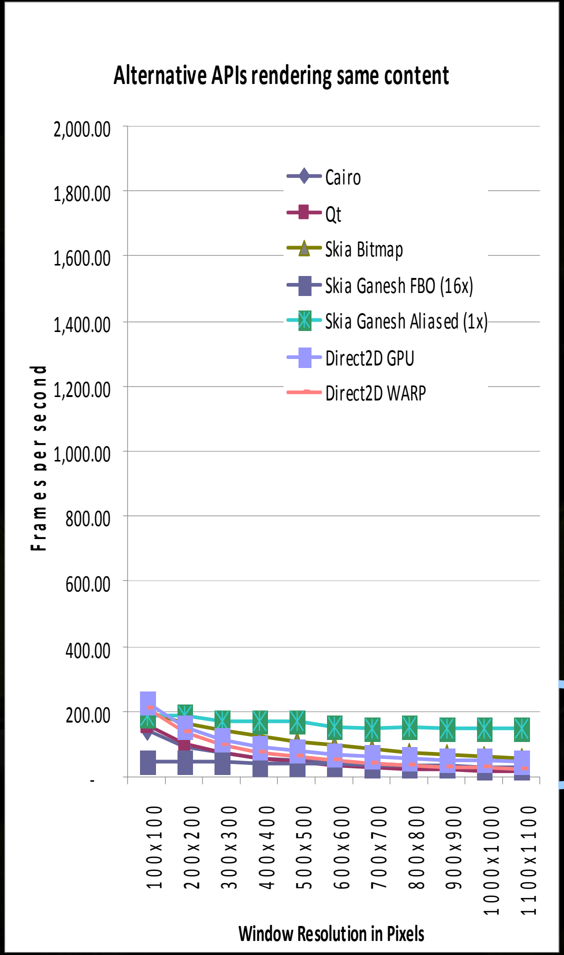
NV_path_rendering Compared to Alternatives

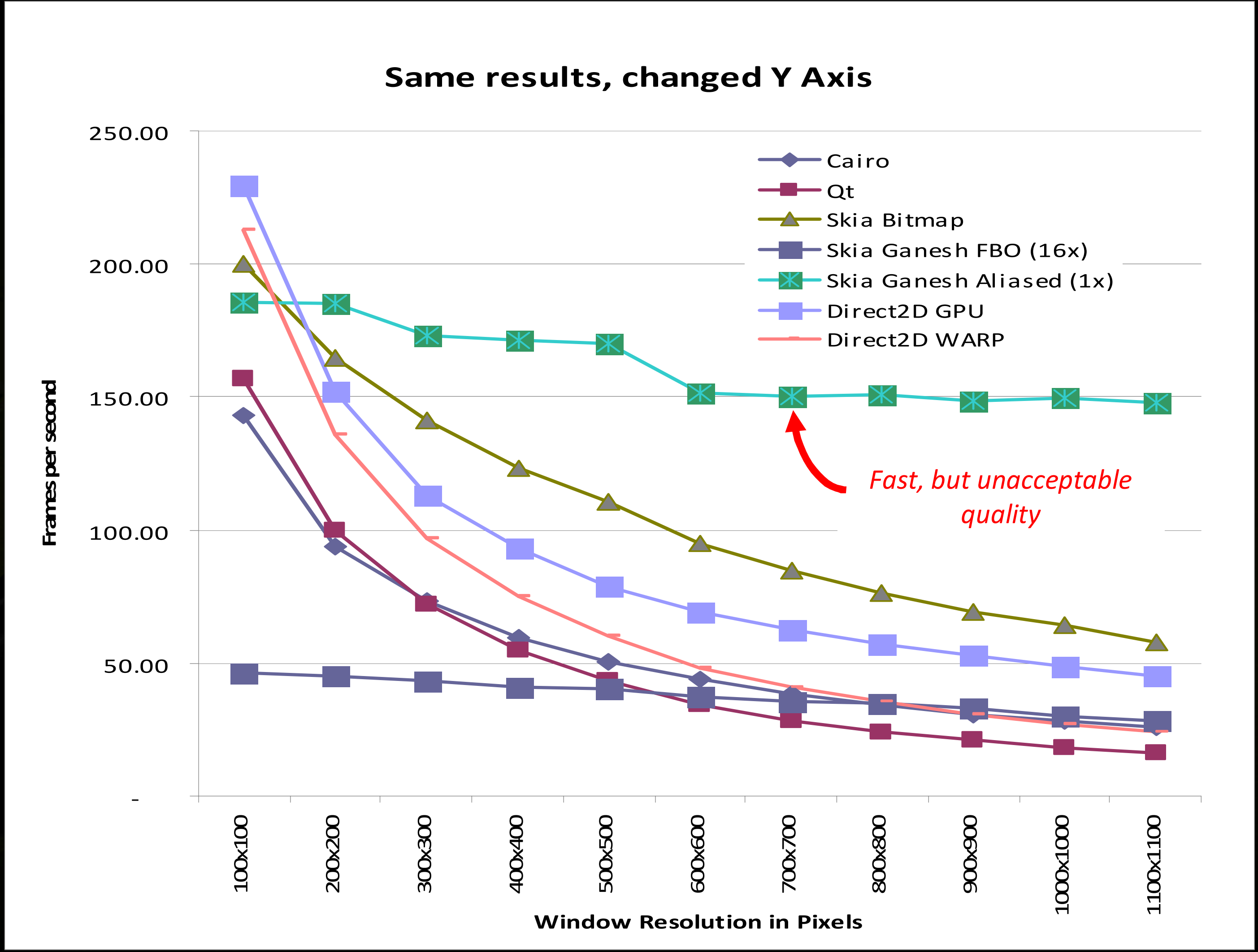




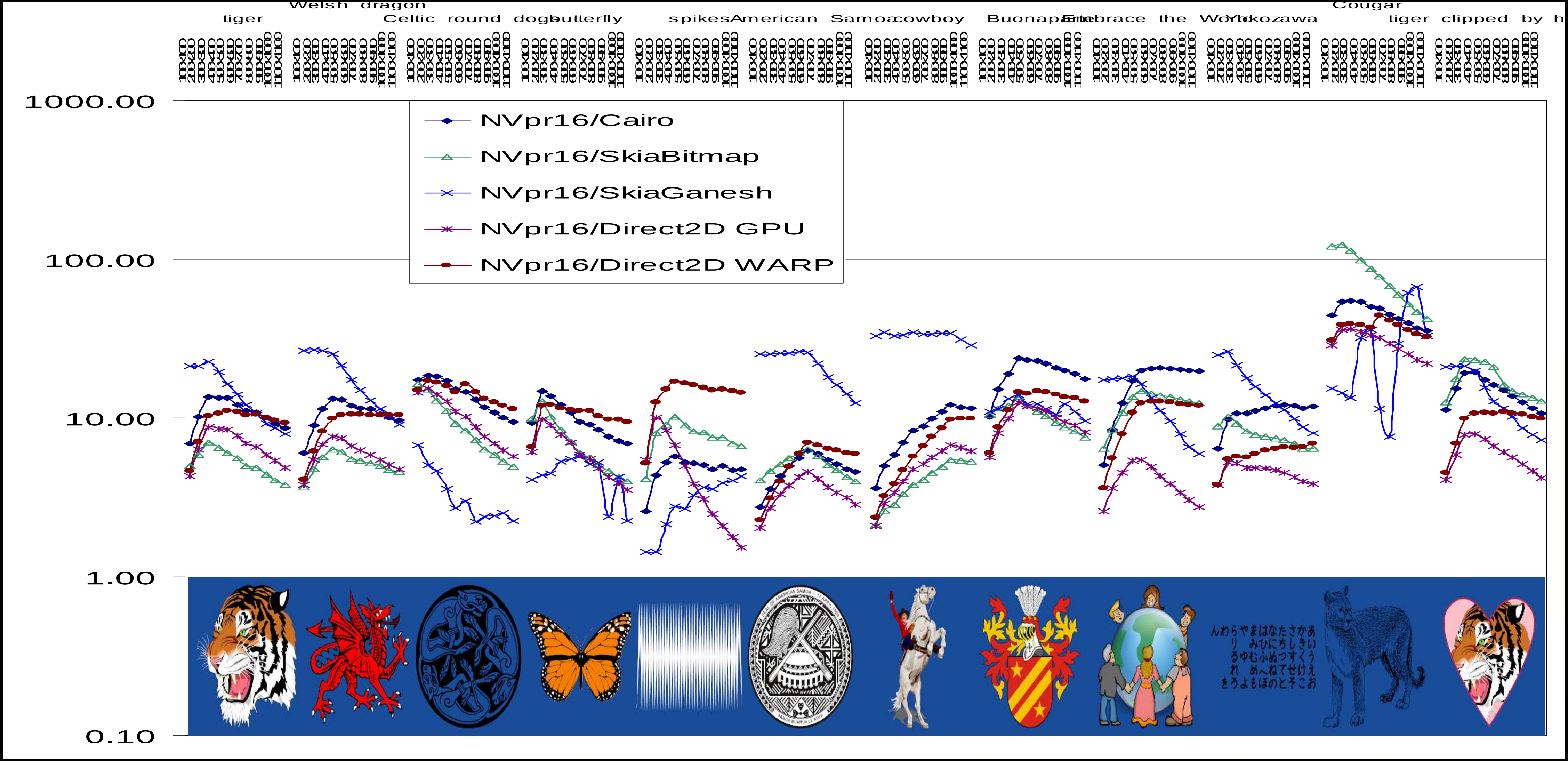
Configuration
GPU: GeForce 480 GTX (GF100)
CPU: Core i7 950 @ 3.07 GHz

Detail on Alternatives



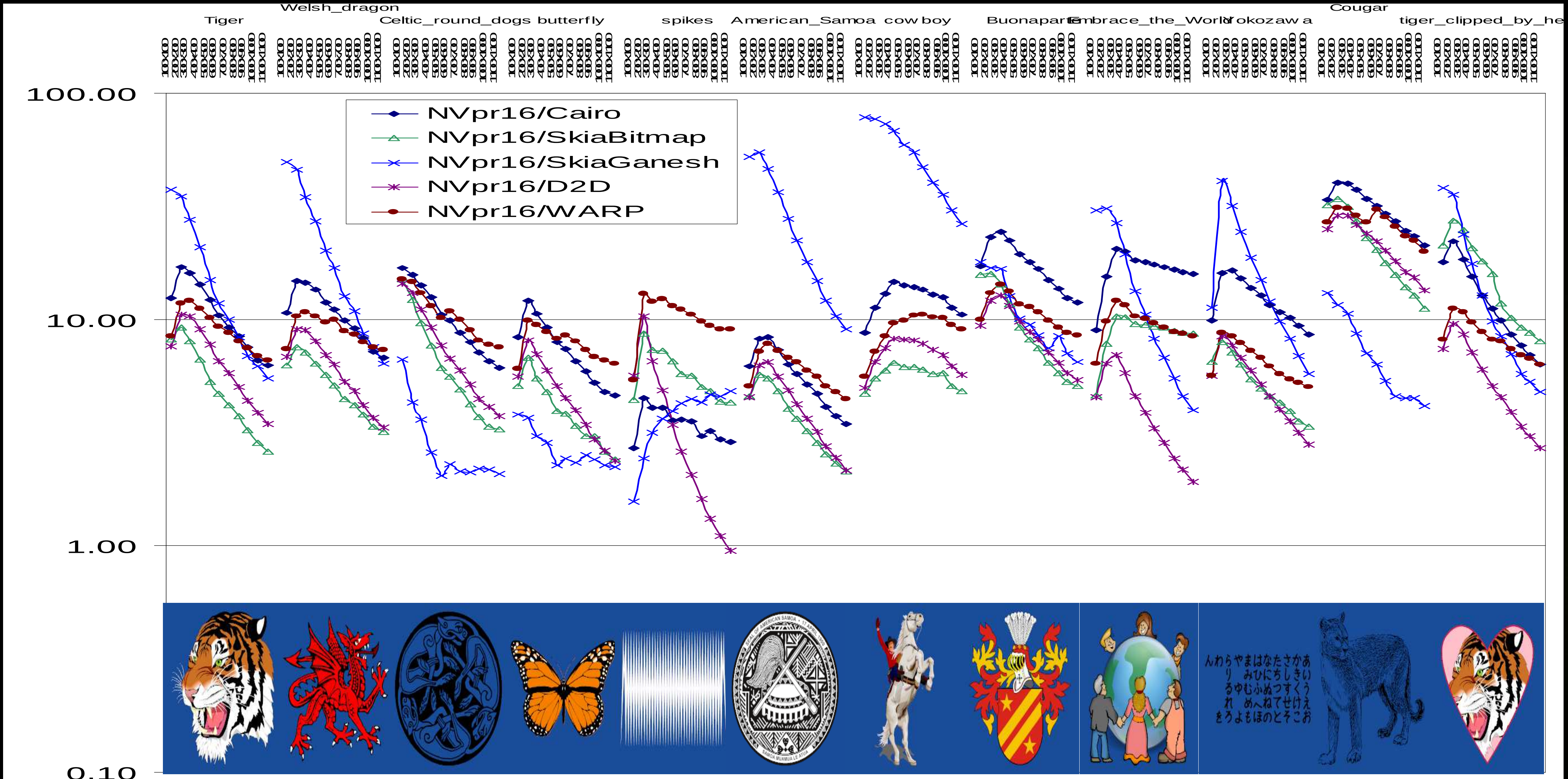


Across an range of scenes…
Release 300 GeForce GTX 480 Speedups over Alternatives

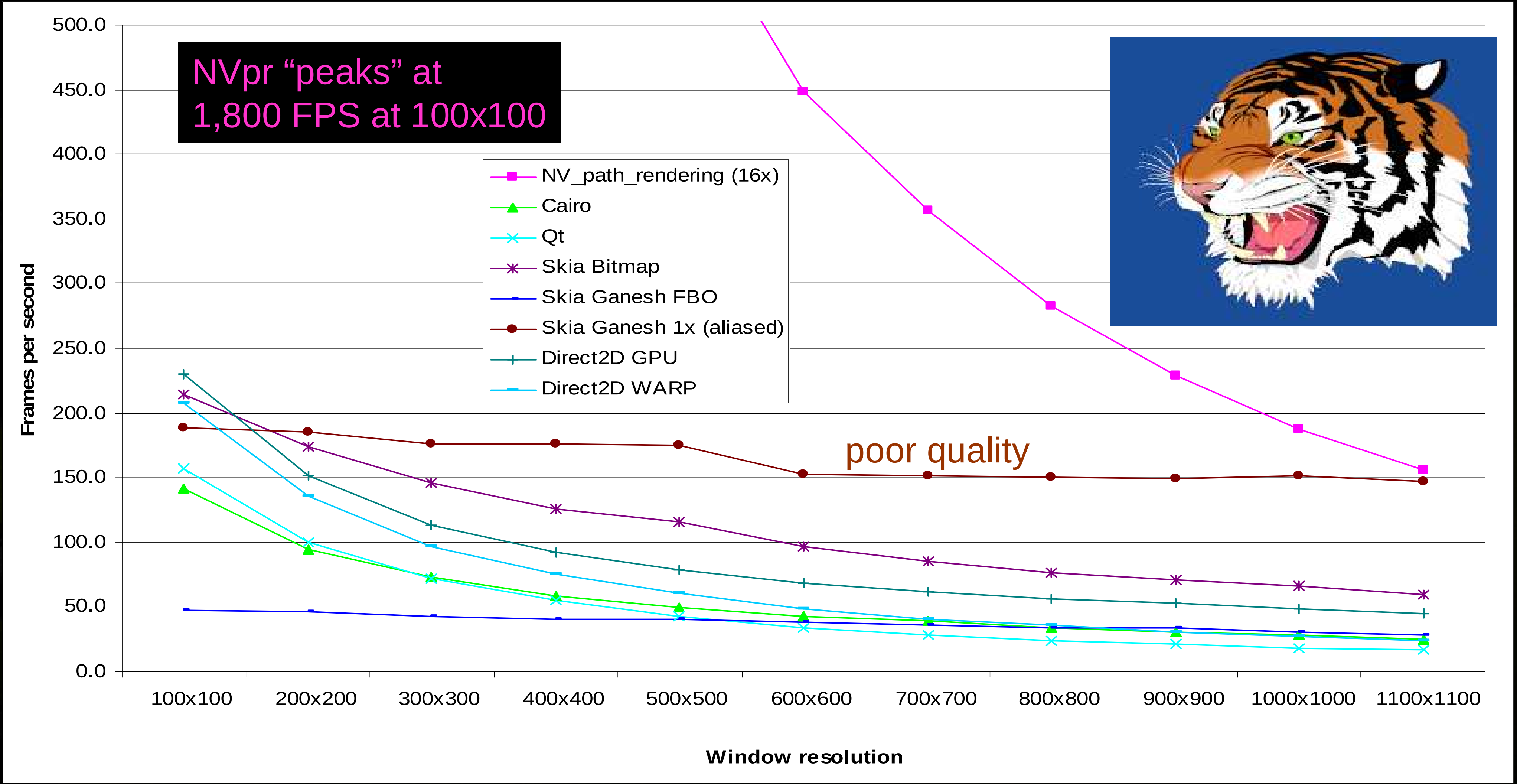


Y axis is <u>logarithmic</u>—shows how many **TIMES** faster NV_path_rendering is that competitor

GeForce 650 (Kepler) Results



Tiger Scene on GeForce 650
Absolute Frames/Second on GeForce 650

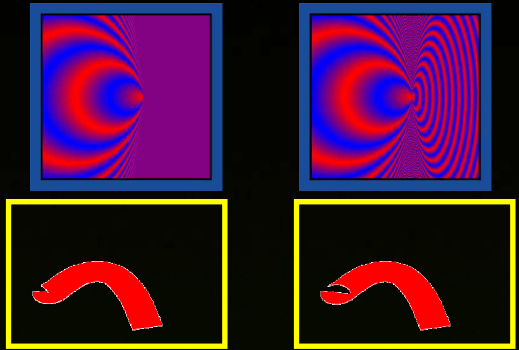


NV_path_rendering is *more* than just matching CPU vector graphics

- **3D and vector graphics mix**

- **Superior quality**

☑ GPU



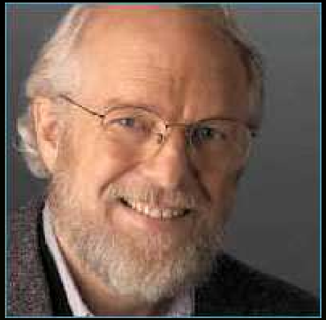
☒ CPU Competitors

- **2D in perspective is free**

- **Arbitrary programmable shader on paths—*bump mapping***

Partial Solutions Not Enough

- Path rendering has 30 years of heritage and history
- Can’t do a 90% solution and Software to change
  - Trying to “mix” CPU and GPU methods doesn’t work
  - Expensive to move software—needs to be an unambiguous win
- Must surpass CPU approaches on all fronts
  - Performance
  - Quality
  - Functionality
  - Conformance to standards
  - More power efficient
  - Enable new applications

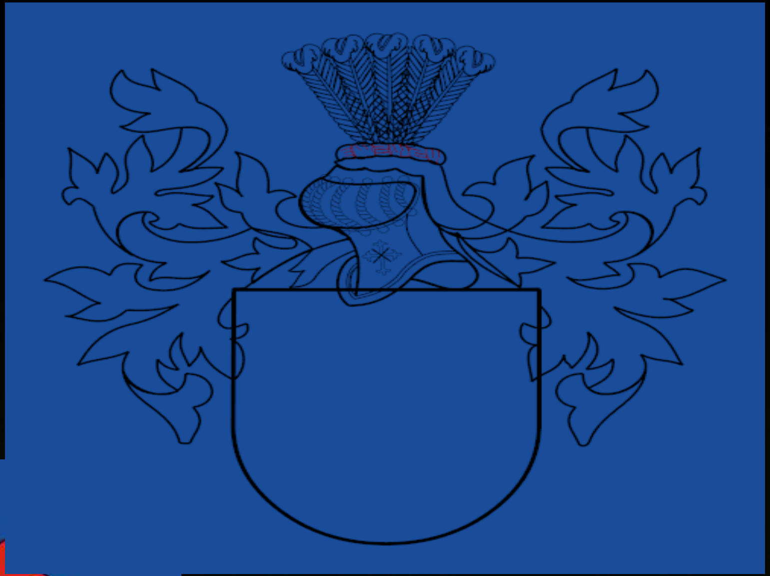


John Warnock
Adobe founder

Inspiration: Perceptive Pixel

Path Filling and Stroking

just filling



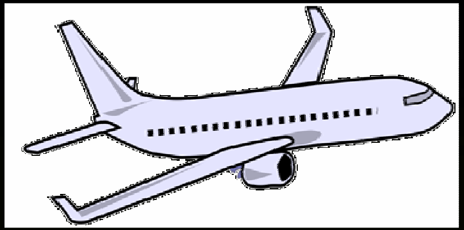
just stroking

filling + stroke =
intended content

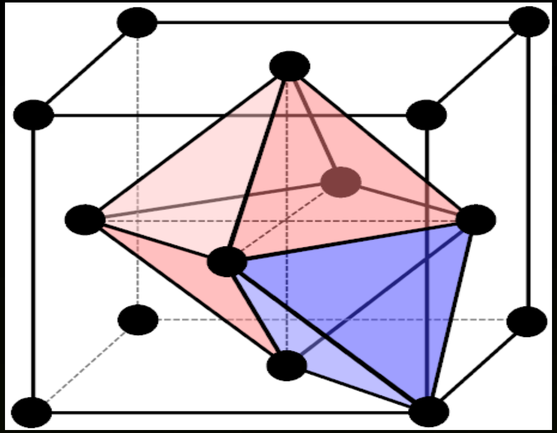
Dashing Content Examples

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



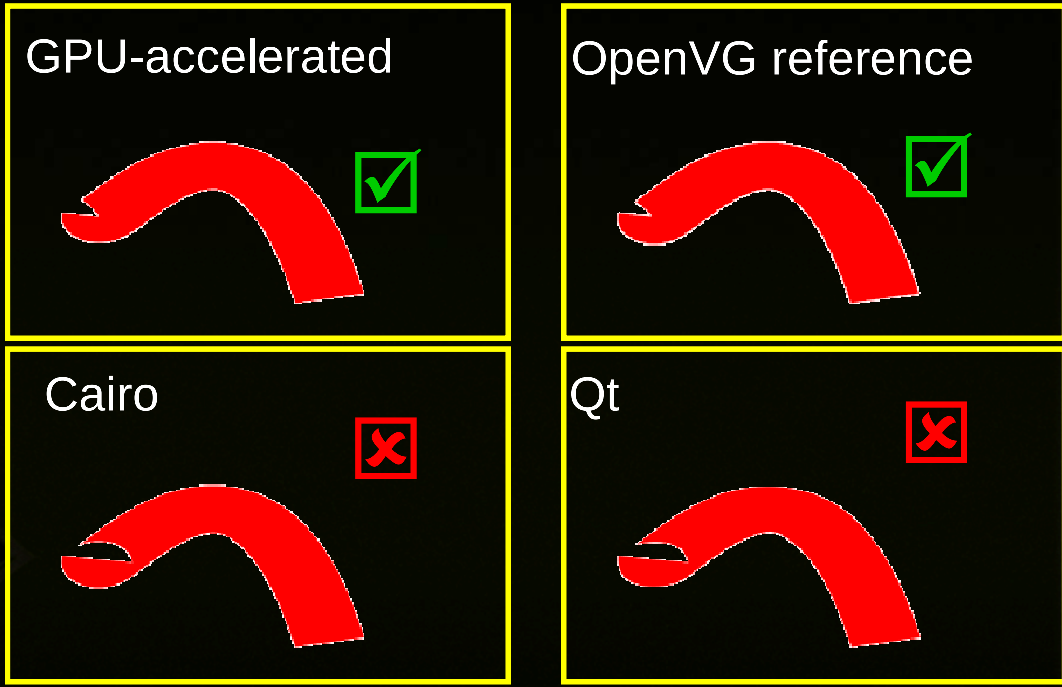
Technical diagrams and charts often employ dashing

Dashing character outlines for quilted look

All content shown is fully GPU rendered

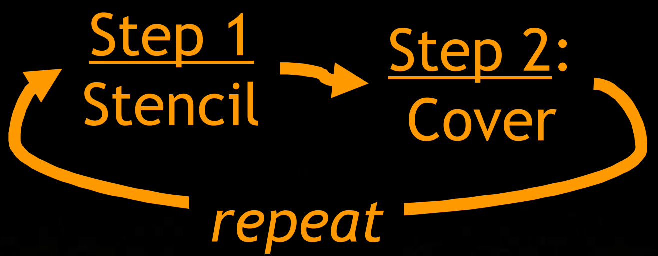
Excellent Geometric Fidelity for Stroking

- **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**

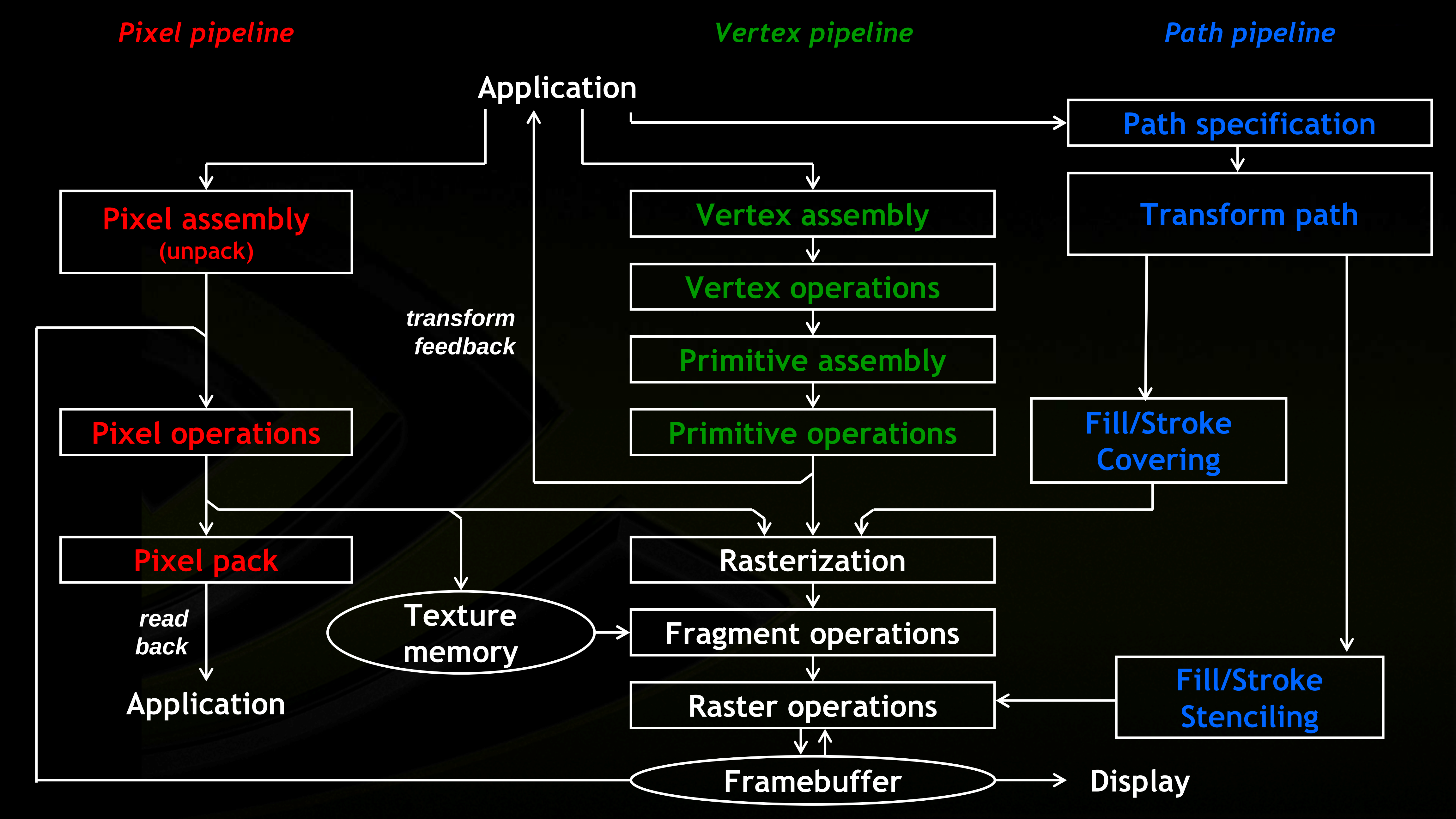


Stroking with tight end-point curve

The Approach



- "Stencil, then Cover" (StC)
- Map the path rendering task from a sequential algorithm…
- …to a pipelined and massively parallel task
- Break path rendering into two steps
  - First, "stencil" the path's coverage into stencil buffer
  - Second, conservatively "cover" path
    - Test against path coverage determined in the 1st step
    - Shade the path
    - And reset the stencil value to render next path

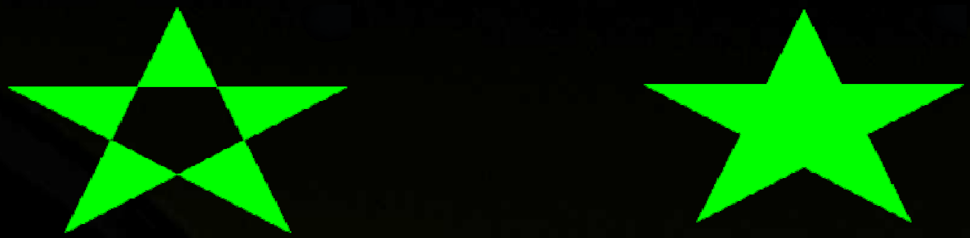

Pixel pipeline
Vertex pipeline
Path pipeline
Application
Path specification
Transform path
Pixel assembly
(unpack)
Vertex assembly
Vertex operations
transform
feedback
Primitive assembly
Pixel operations
Primitive operations
Fill/Stroke
Covering
Pixel pack
Rasterization
read
back
Texture
memory
Fragment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

Key Operations for Rendering Path Objects

- Stencil operation
  - only updates stencil buffer
  - glStencilFillPathNV, glStencilStrokePathNV
- Cover operation
  - glCoverFillPathNV, glCoverStrokePathNV
  - renders hull polygons guaranteed to “cover” 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 steps
  - Why? Paths have control points & rasterized regions, <u>not</u> vertices, triangles

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

DSA = OpenGL's Direct State Access extension (EXT_direct_state_access)

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

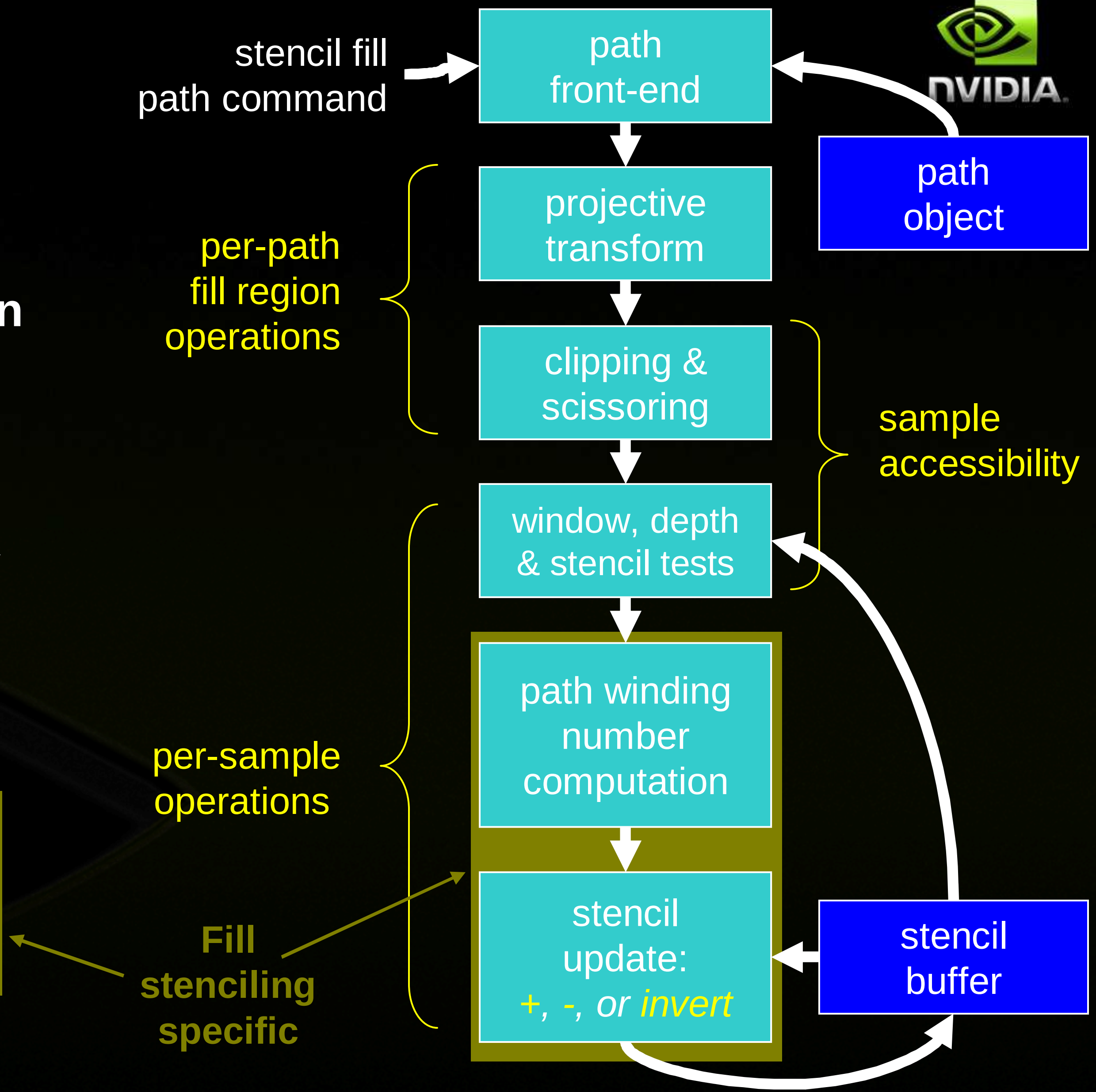
```
glStencilFunc(GL_NOTEQUAL, 0, 0x1);  // alternative mask
```

non-zero fill style

even-odd fill style

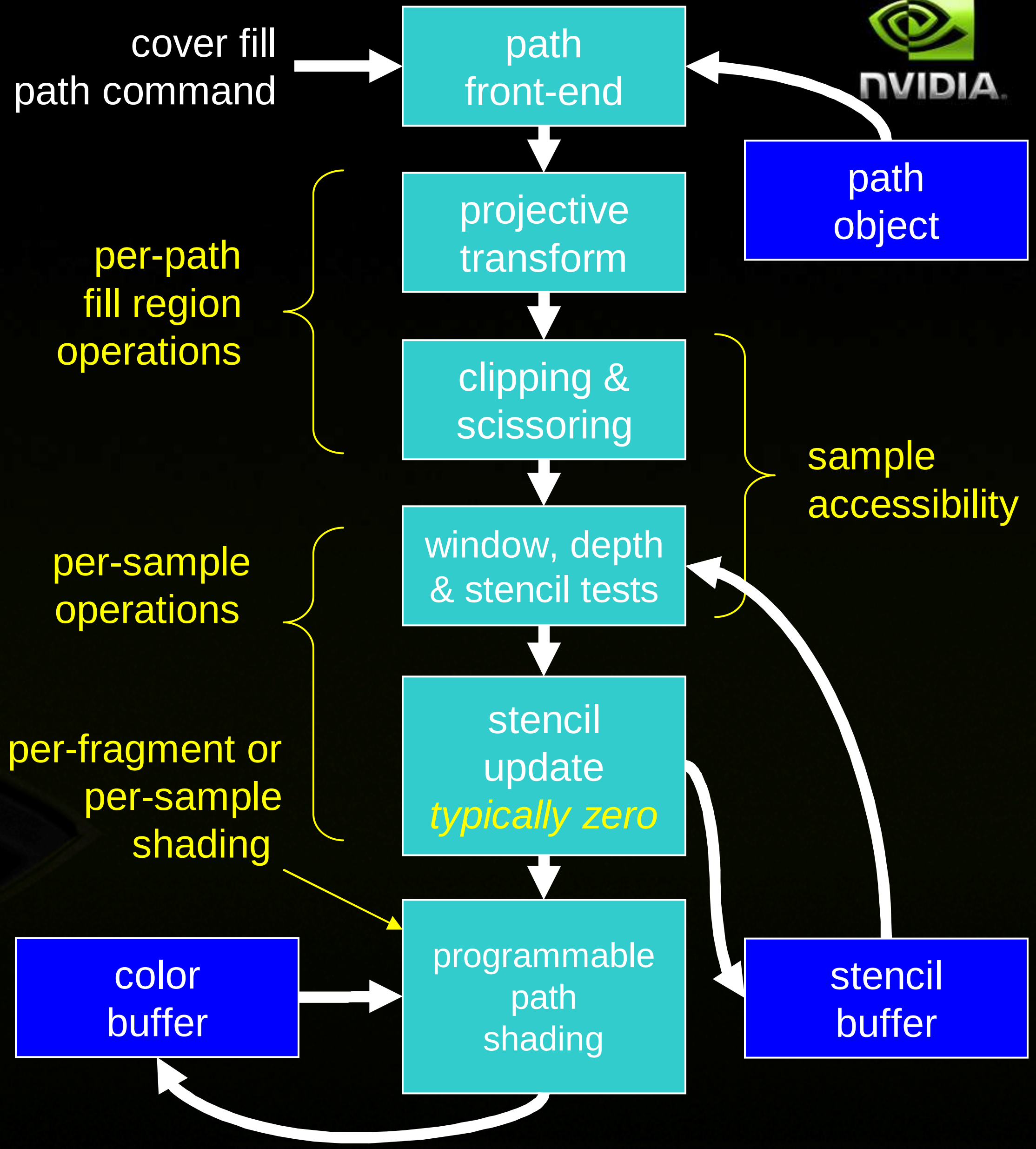
"Stencil, then Cover" Path Fill Stenciling

- **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 mask, stipple pattern, and window ownership**
- **Conservative covering geometry uses stencil to "cover" filled path**
  - **Determined by prior stencil step**



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”
```

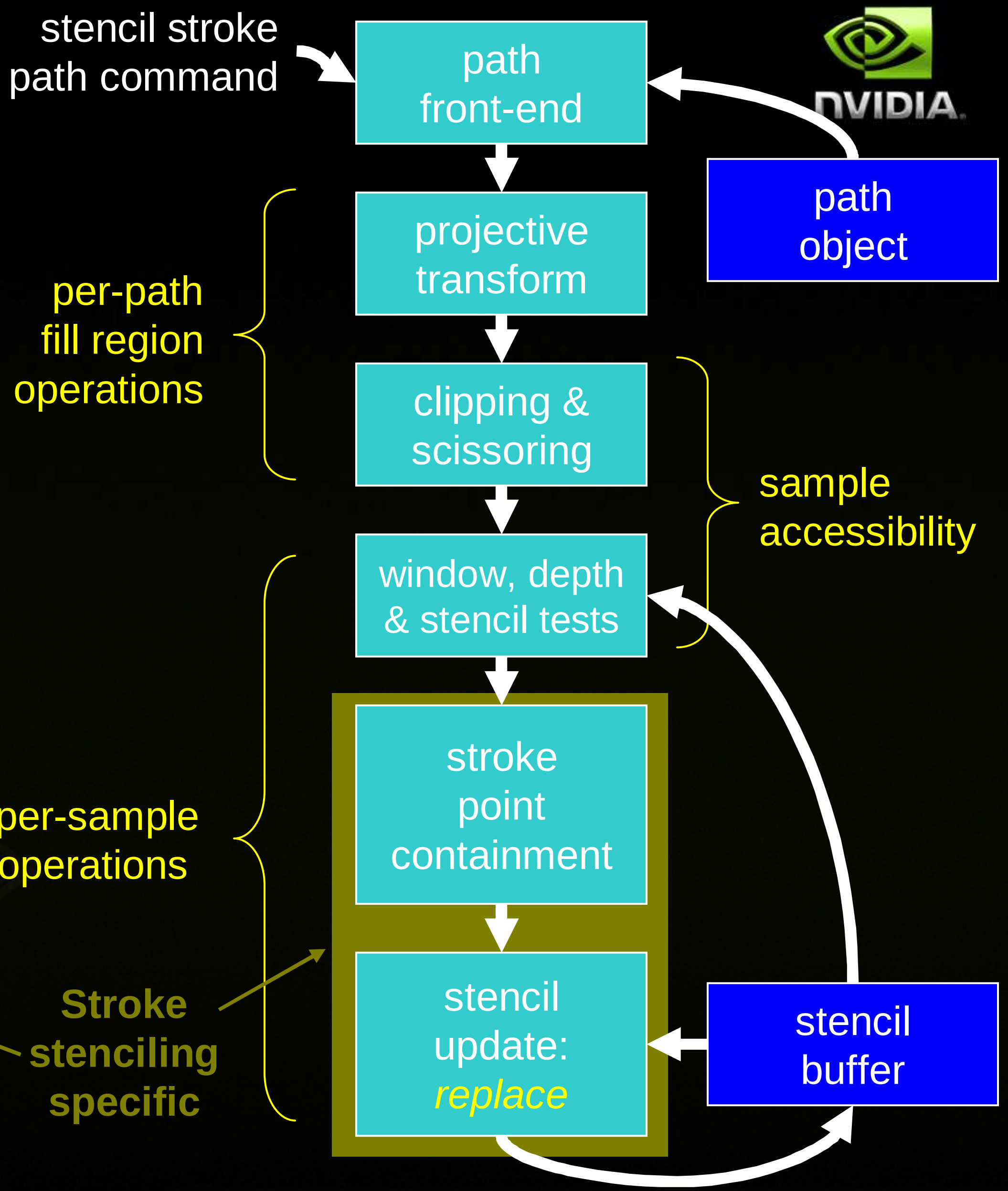
  - **Cover path**

```
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

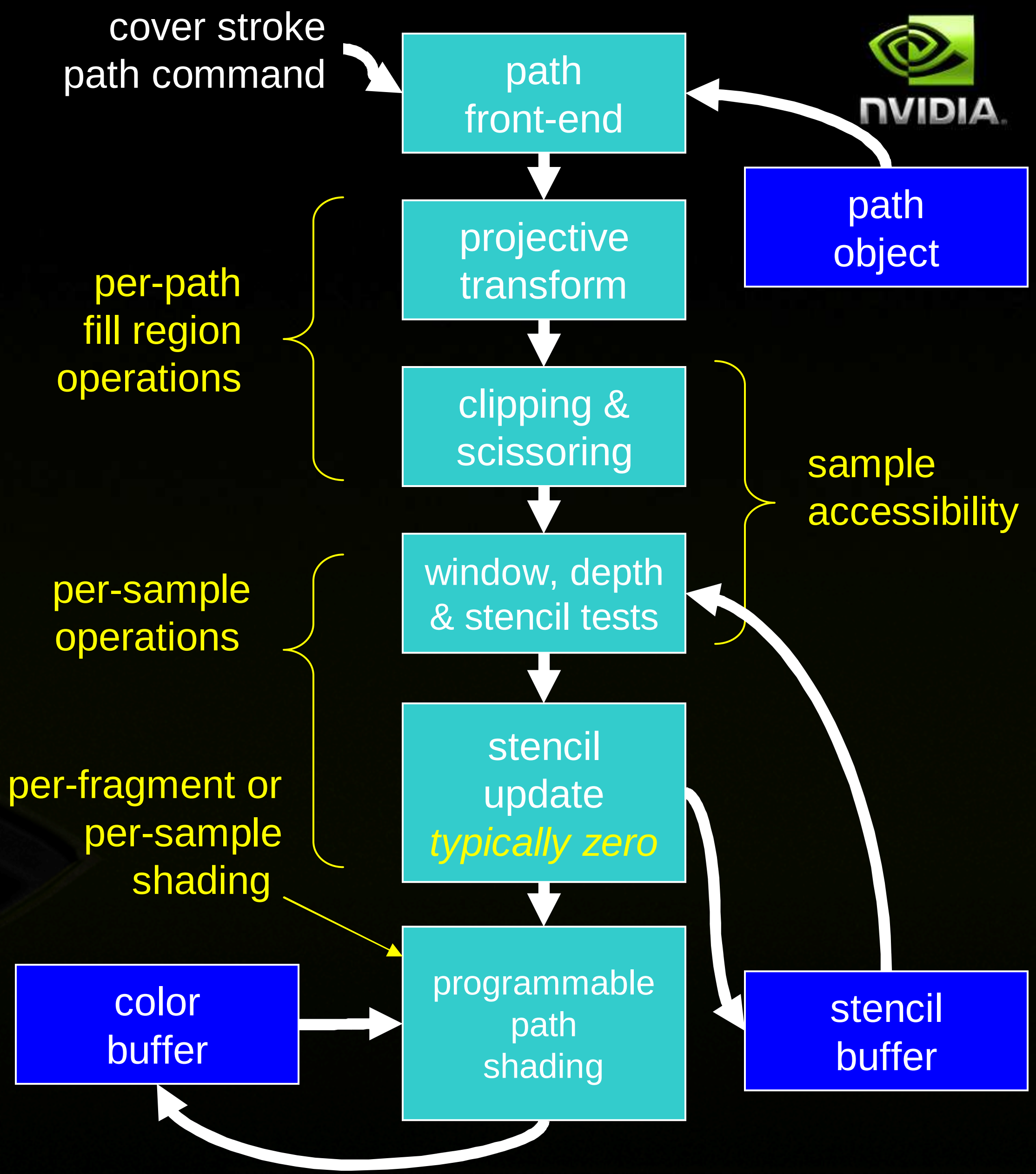
“Stencil, then Cover”
Path Stroke Stenciling

- **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**

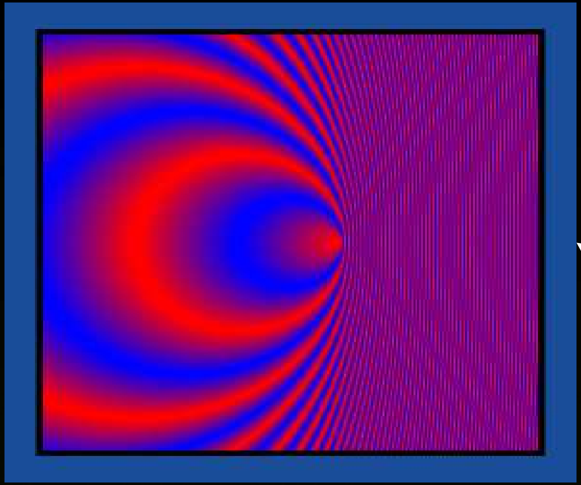


First-class, Resolution-independent Font Support

- 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

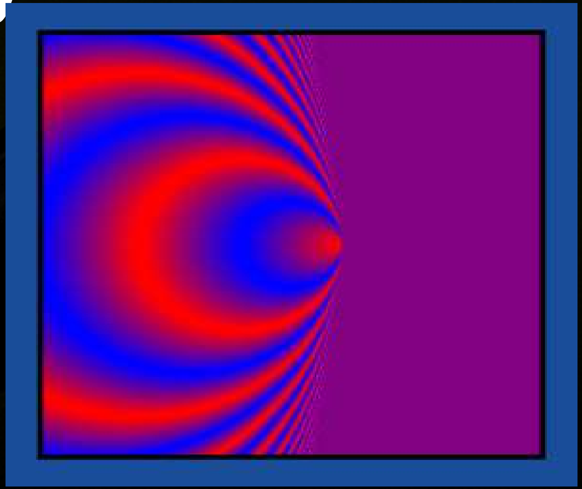
Handling Common Path Rendering Functionality: Filtering

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

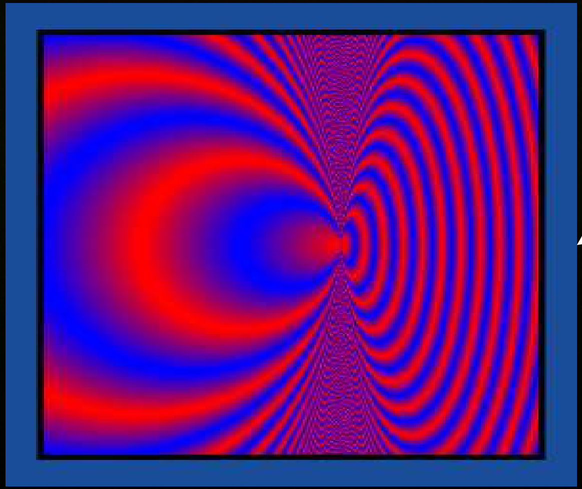


☒ Qt

Moiré artifacts



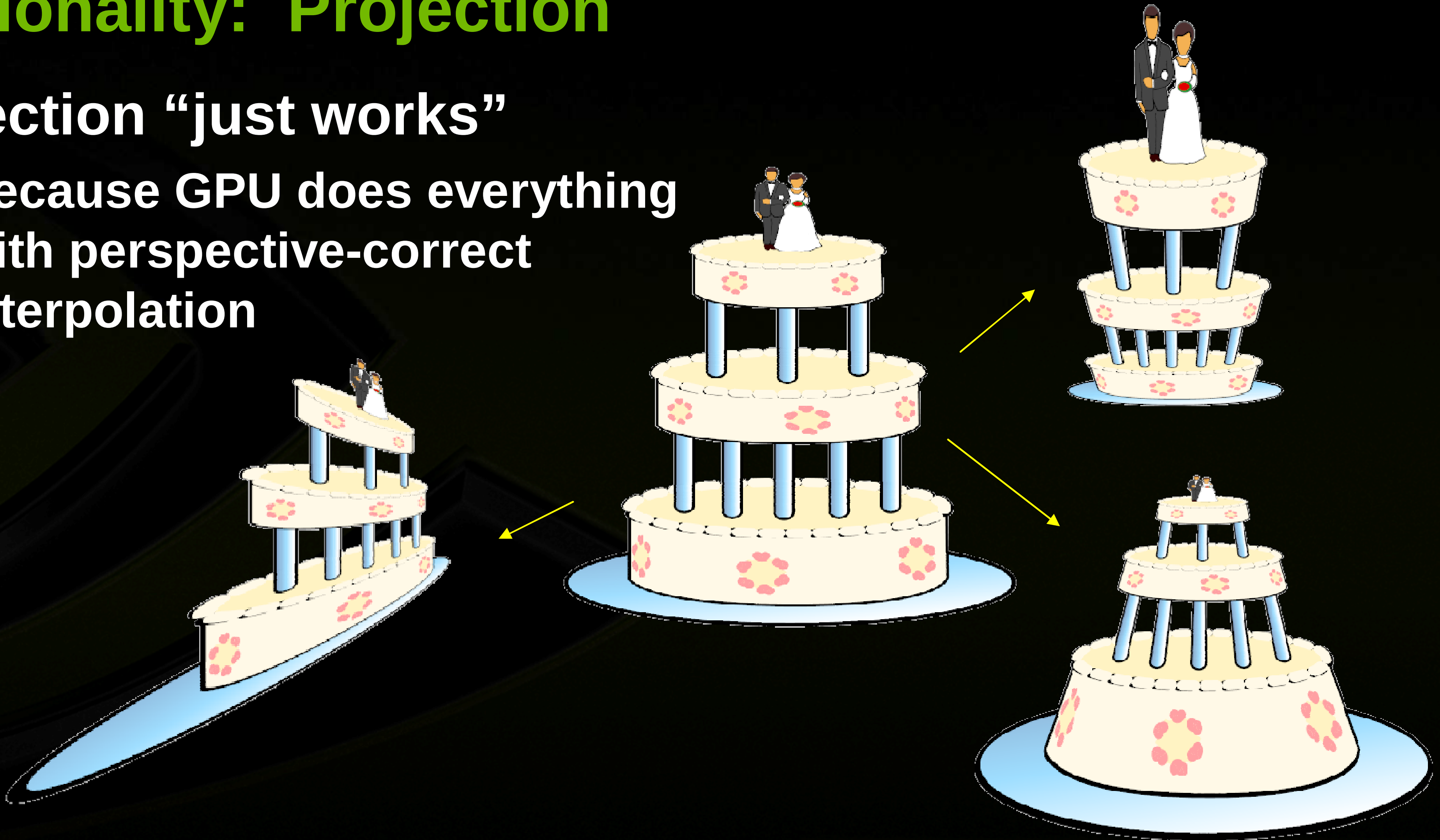
☑ GPU



☒ Cairo

Handling Uncommon Path Rendering Functionality: Projection

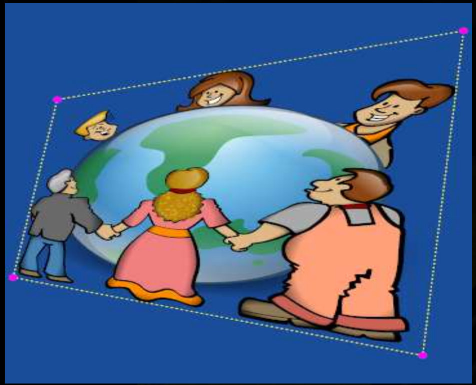
- Projection "just works"
  - Because GPU does everything with perspective-correct interpolation



Projective Path Rendering Support Compared

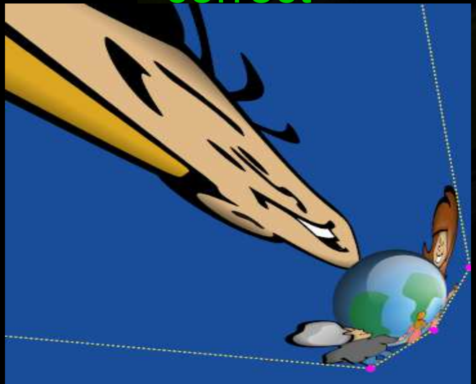
☑ GPU

flawless



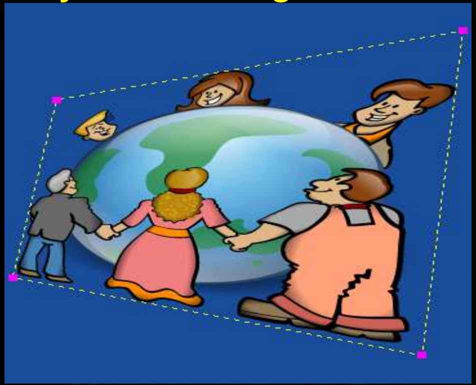
correct

correct



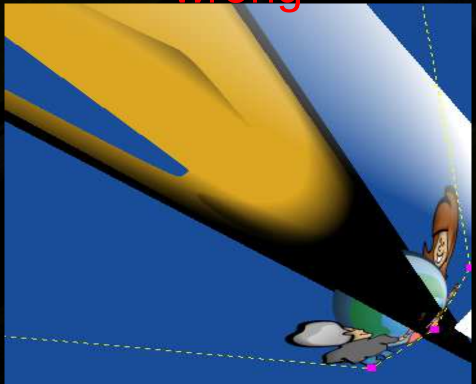
☹ Skia

yes, but bugs



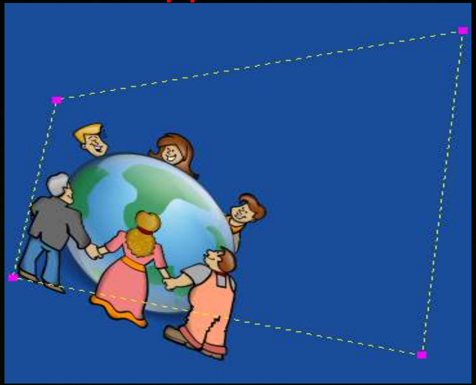
correct

wrong



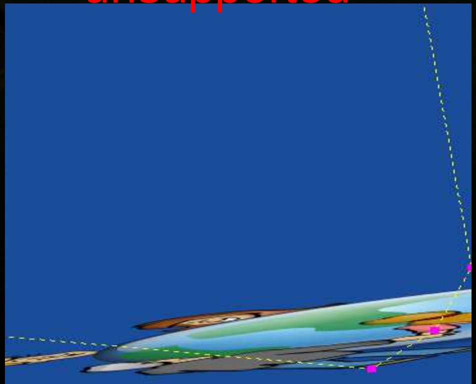
☒ Cairo

unsupported



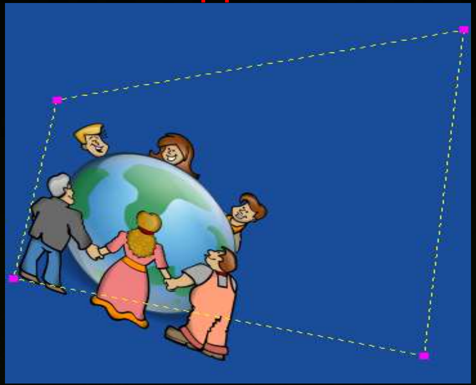
unsupported

unsupported



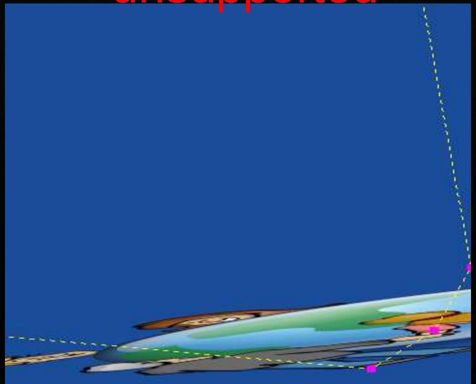
☒ Qt

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

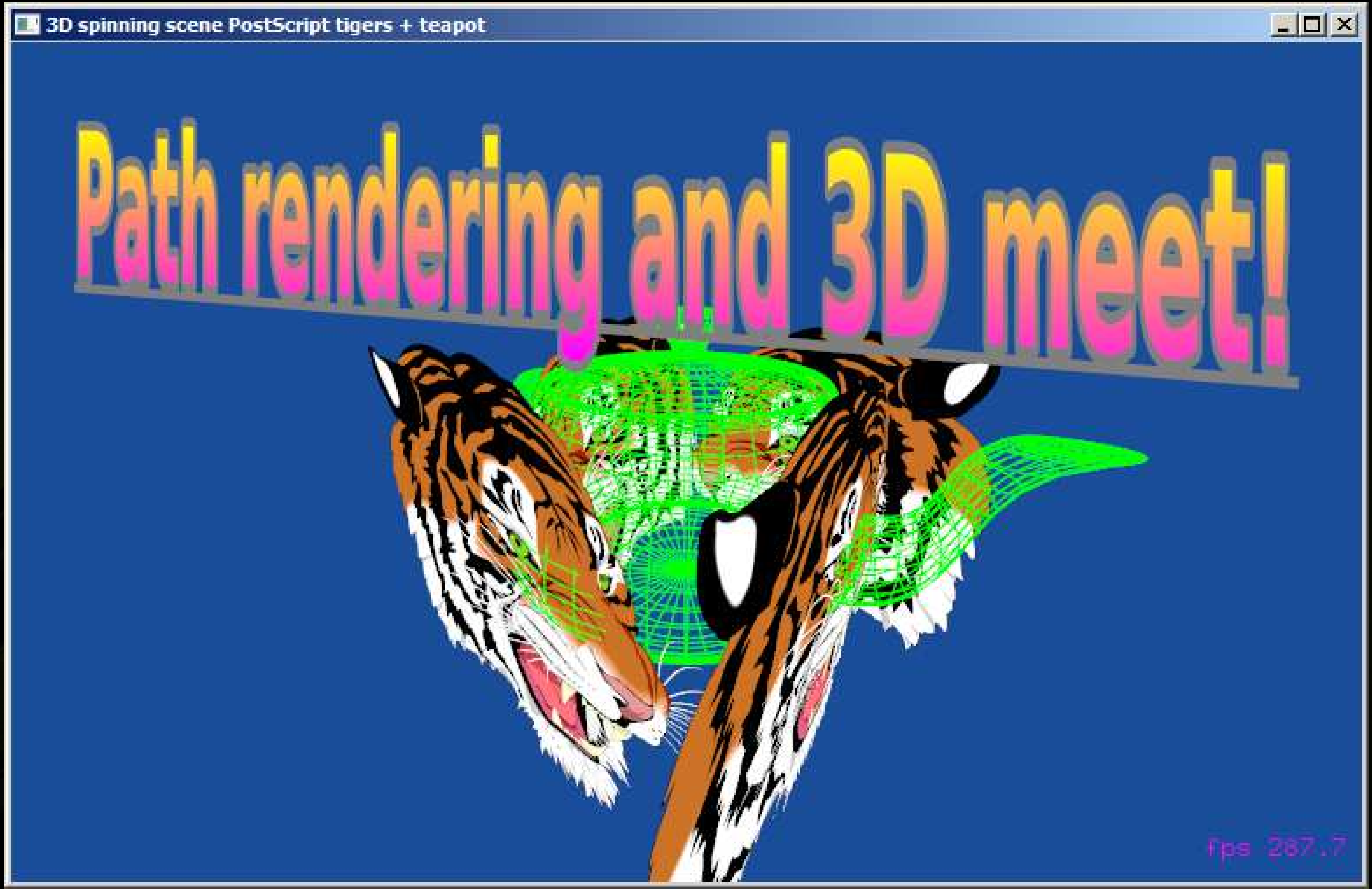
Accessible Samples of a Transformed Path

- 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**

Demo


3D spinning scene PostScript tigers + teapot
Path rendering and 3D meet!
fps 287.7

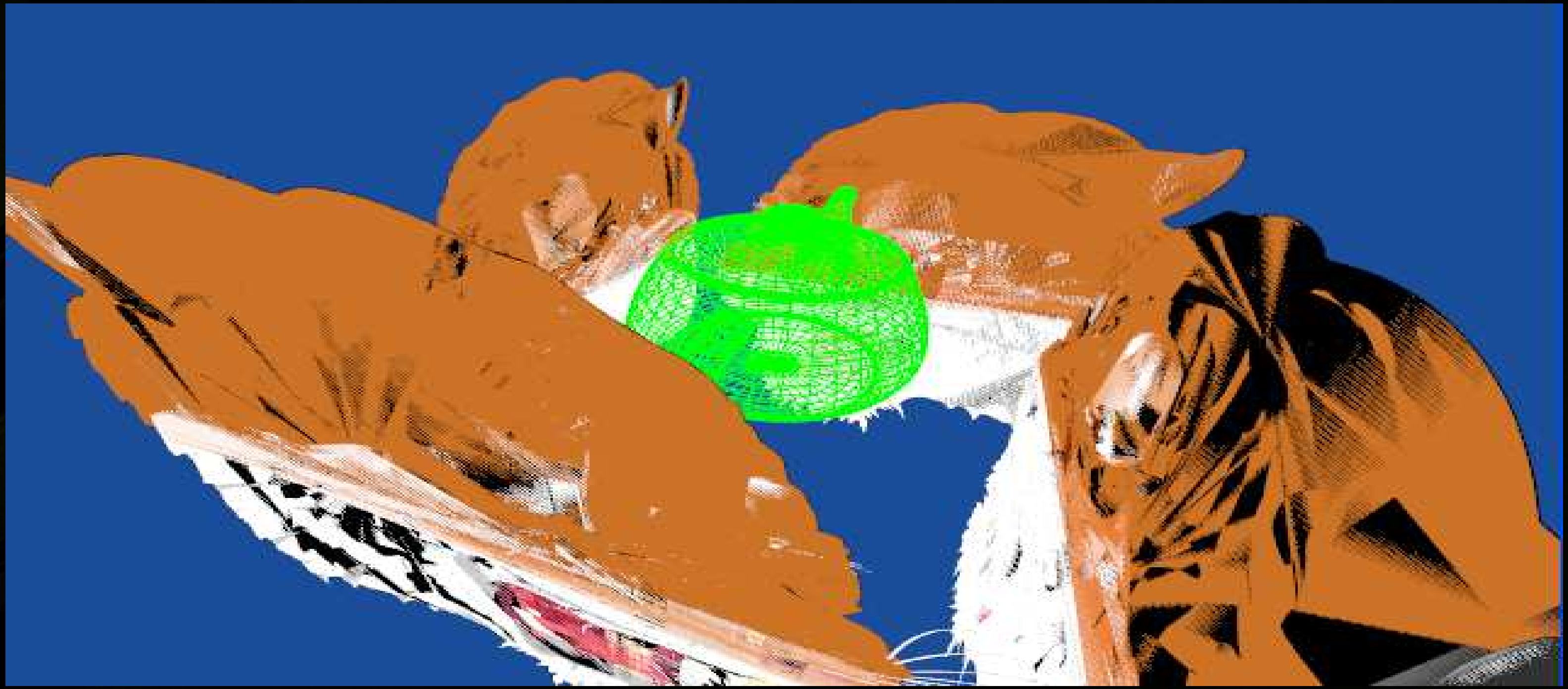
3D Path Rendering Details

- 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

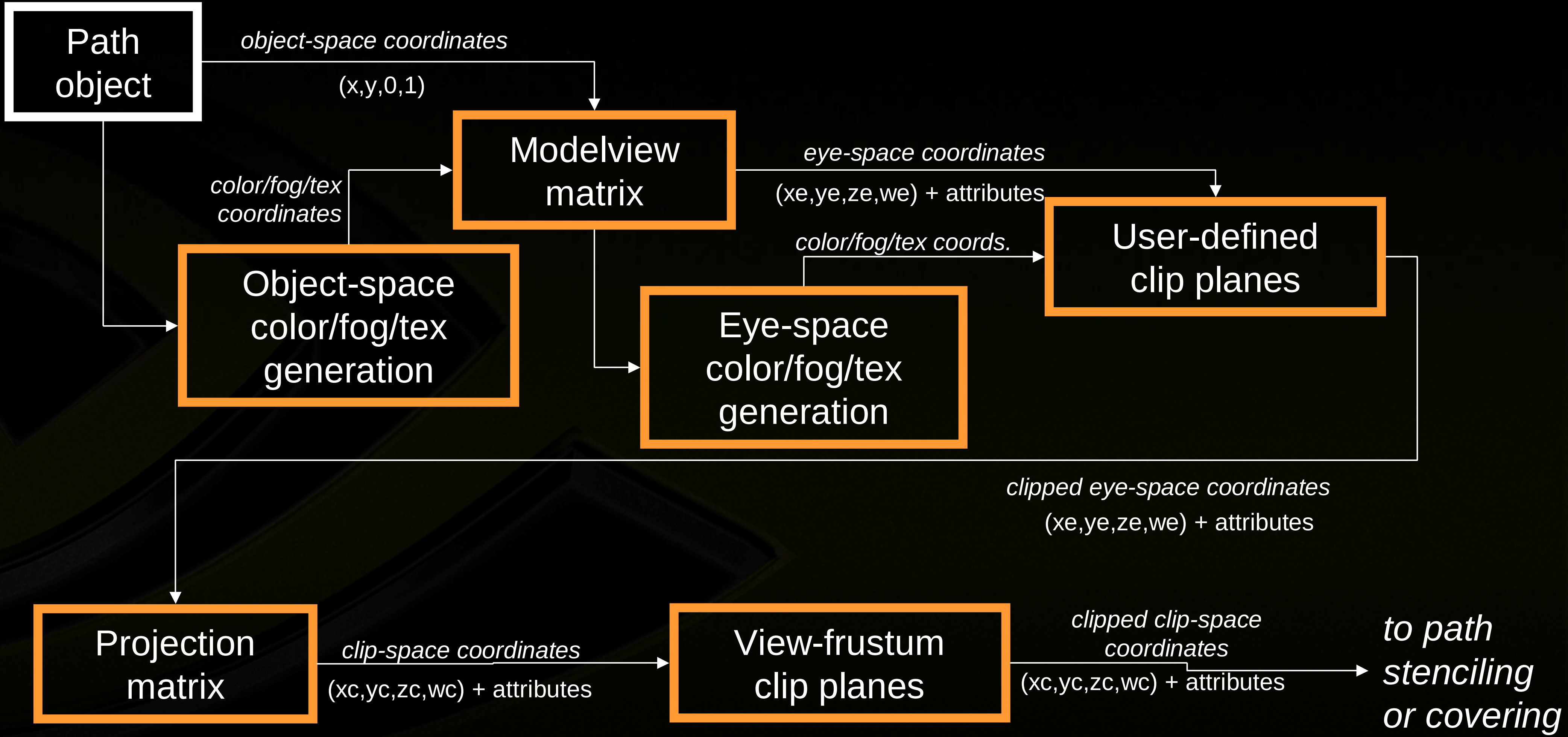
Without glPathStencilDepthOffset Bad Things Happen

- **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**



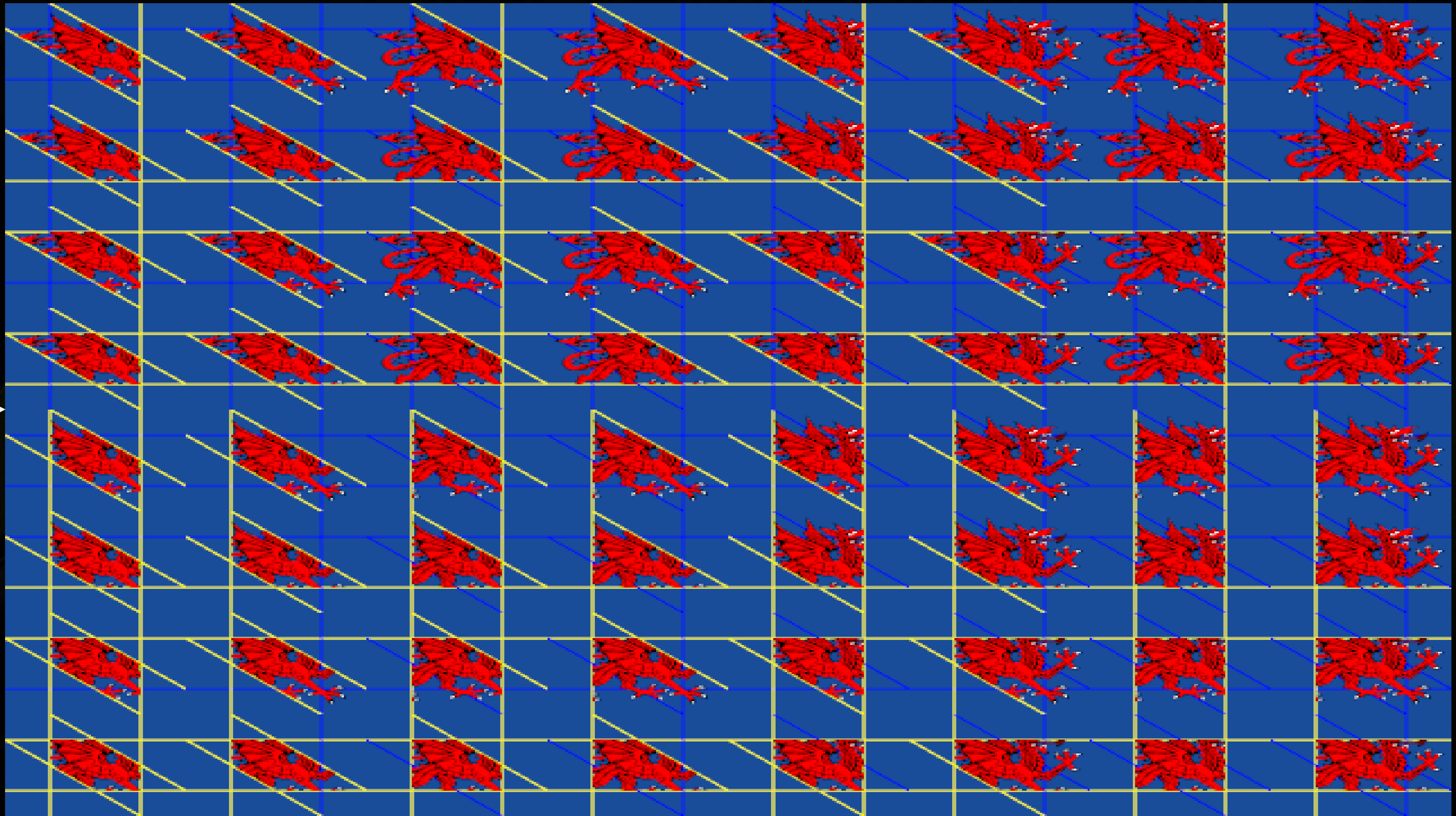
terrible z-fighting artifacts

Path Transformation Process



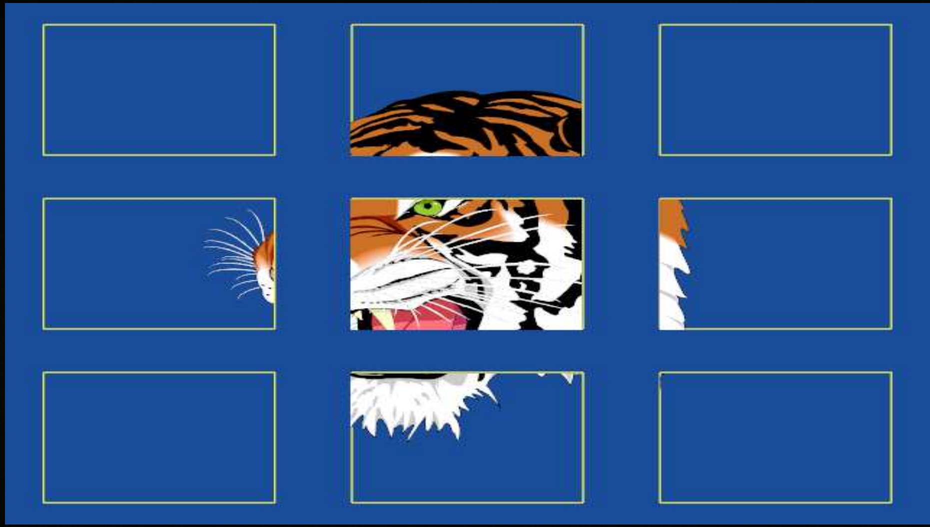
Clip Planes Work with Path Rendering

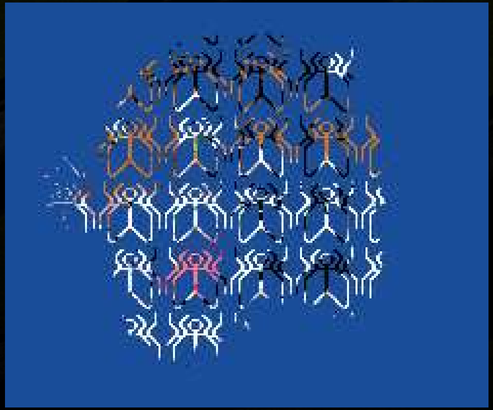
- **Scene showing a Welsh dragon clipped to all 64 combinations of 6 clip planes enabled & disabled**



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

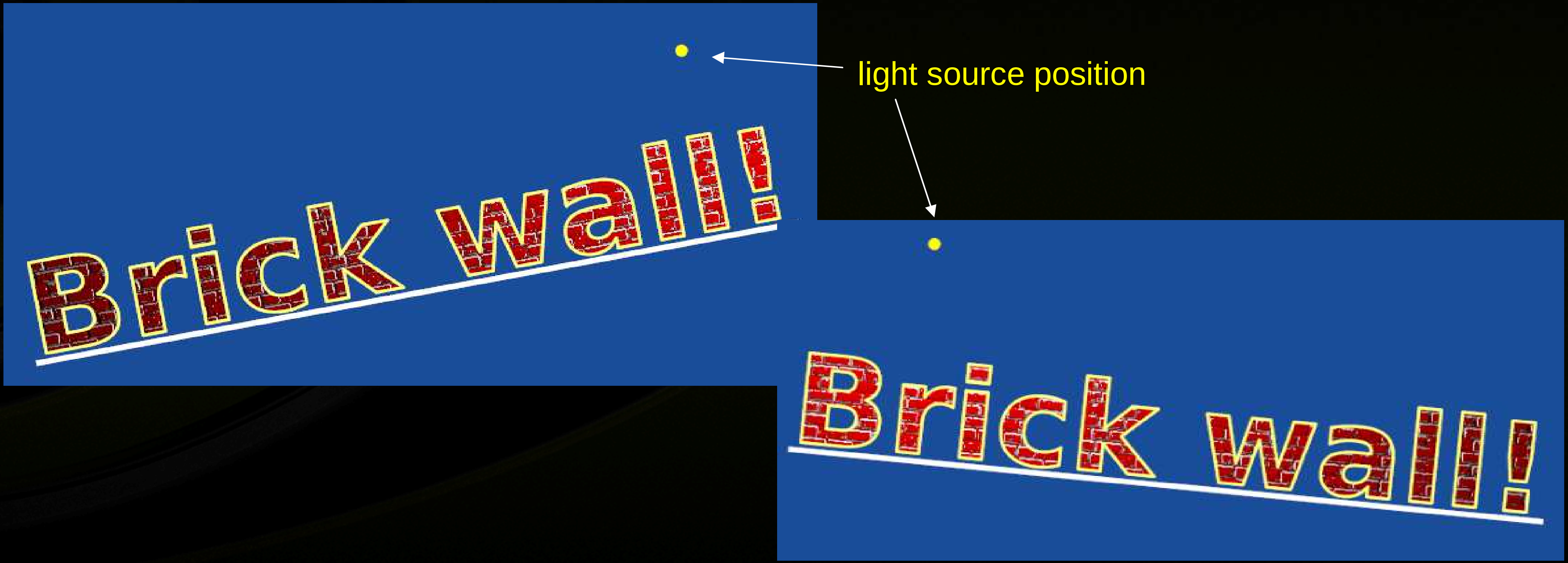
result of clipping tiger
to the union of all the cowboy paths

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, & tessellation shaders <u>ignored</u> during 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**



Anti-aliasing 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 uses multiple stencil samples per pixel for reasonable antialiasing
  - Otherwise, image quality is poor
  - 4 samples/pixel bare minimum
  - 8 or 16 samples/pixel is pretty sufficient
    - But 16 requires expensive 2x2 supersampling of 4x multisampling
    - 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 schemes
  - >>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

Real Flash Scene

conflation artifacts abound, rendered by Skia

conflation is aliasing &
edge coverage percents
are un-predicable in general;
means conflated pixels
flicker when animated slowly

same scene, GPU-rendered without conflation

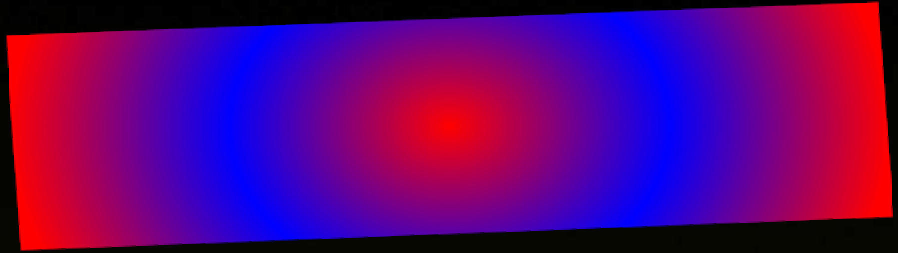
GPU Advantages

- 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

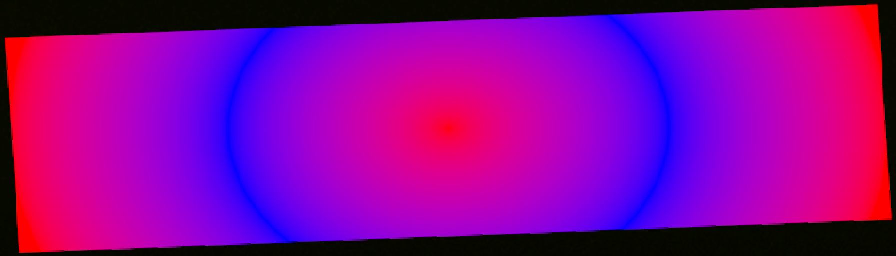
- Modern GPUs have native support for perceptually-correct for
  - sRGB framebuffer blending
  - sRGB texture filtering
  - No reason to tolerate uncorrected linear RGB color artifacts!
  - More intuitive for artists to control
- Negligible expense for GPU to perform sRGB-correct rendering
  - However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

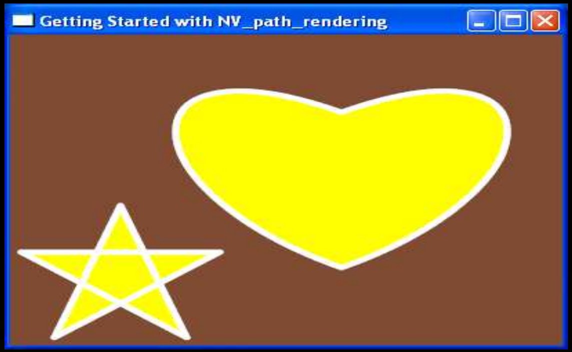
perceptually smooth transition from saturated red to saturated blue

Trying Out NV_path_rendering

- Operating system support
  - 2000, XP, Vista, Windows 7, Linux, FreeBSD, and Solaris
  - Unfortunately no Mac support
- GPU support
  - GeForce 8 and up (Tesla and beyond)
  - Most efficient on Fermi and Kepler GPUs
  - Current performance can be expected to improve
- Shipping since NVIDIA's Release 275 drivers
  - Available since summer 2011
- New Release 300 drivers have remarkable NV_path_rendering performance
  - Try it, you'll like it

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
  - Standard makefiles for Linux
  - Use Visual Studio 2008 for Windows

Whitepapers

- "Getting Started with NV_path_rendering"

Getting Started with NV_path_rendering

Mark J. Kilgard
NVIDIA Corporation

May 20, 2011

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

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

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

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

Unlike geometric primitive rendering, paths are specified on a 2D (non-projective) plane rather than in 3D (projective) space. Even though the path is defined in a 2D plane, every

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

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

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

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

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

Whitepapers

- **“Mixing 3D and Path Rendering”**

Mixing Path Rendering and 3D

Mark J. Kilgard
NVIDIA Corporation

June 20, 2011

In this whitepaper, you will learn how to mix conventional 3D rendering with GPU-accelerated path rendering within your OpenGL program using the NV_path_rendering extension. NV_path_rendering is supported by all CUDA-capable NVIDIA GPUs with Release 275 and later drivers. This whitepaper assumes you are familiar with OpenGL programming in general and how to use OpenGL extensions.

If you are not familiar with NV_path_rendering, first study the *Getting Started with NV_path_rendering* whitepaper.

Normally path rendering and 3D rendering have an oil-and-water relationship for a lot of reasons:

- 3D rendering relies on depth buffering to resolve 3D opaque occlusion; path rendering explicitly depends on the rendering order of layers. Conventional path rendering has no notion of a depth buffer.
- 3D rendering renders on simple primitives with straight (linear) edges such as points, lines, and polygons; path rendering primitives can be arbitrarily complex, contain holes, self-intersections, and have curved edges.
- 3D rendering uses programmable shading for per-pixel effects, typically written in a high-level shading language such as Cg; path rendering relies on artists to layer paths and add filter effects to achieve fancy results.

Figure 6: Teapot and tigers scene with fancy text using the ParkAvenue BT TrueType font and drawn with a 2D projective transform and underlining.

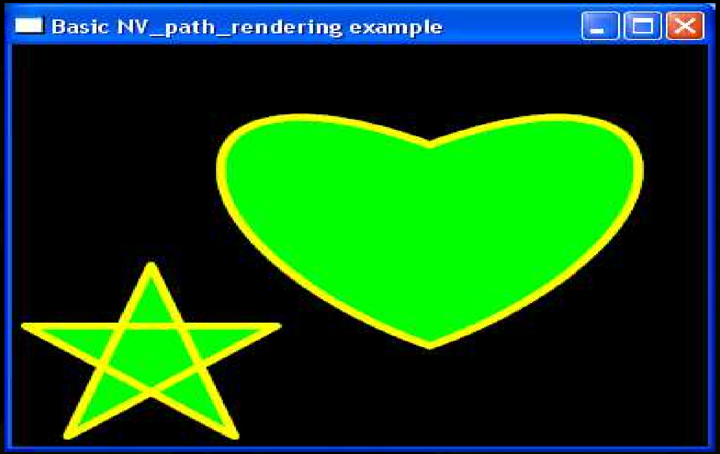
NV_path_rendering supports an “instanced” API for stenciling or covering multiple path objects, typically glyphs, in a single API command. These instanced commands support various per-object transforms of different types, including arbitrary projective 3D transforms.

Programmable Fragment Shading

The discussion so far has discussed how to mix GPU-accelerated path rendering with 3D rendering using a single projective 3D view and single depth buffer but has not addressed programmable shading.

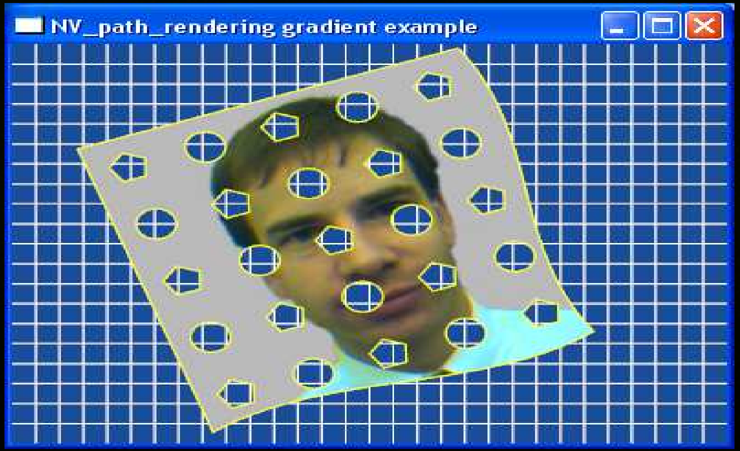
SDK Example Walkthrough (1)

pr_basic—simplest example of path filling & stroking



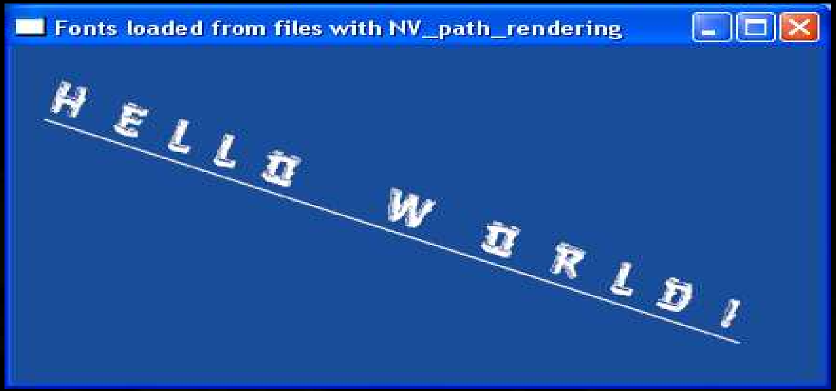
pr_hello_world—kerned, underlined, stroked, and linear gradient filled text

pr_welsh_dragon—filled layers



pr_gradient—path with holes with texture applied

SDK Example Walkthrough (2)

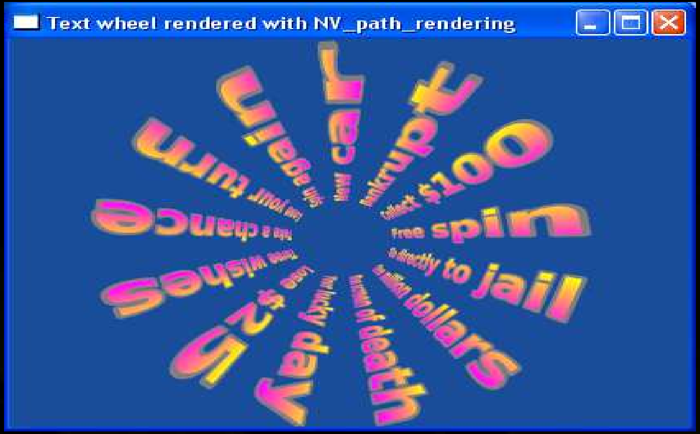


pr_font_file—loading glyphs from a font file with the GL_FONT_FILE_NV target

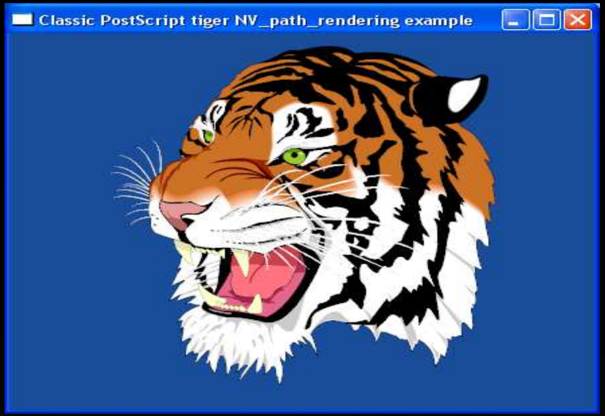
pr_korean—rendering UTF-8 string of Korea characters

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

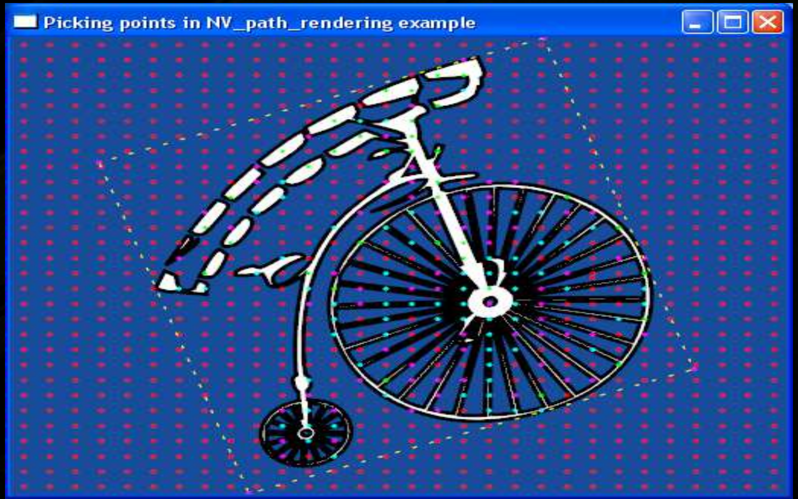
pr_warp_tiger—warp the tiger with a free projective transform

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

Conclusions

- GPU-acceleration of 2D resolution-independent graphics is coming
  - HTML 5 and low-power requirements are demanding it
- "Stencil, then Cover" approach
  - Very fast
  - Quality, functionality, and features
  - Available today through NV_path_rendering
- Shipping today
  - NV_path_rendering resources available

Questions?

More Information

- Best drivers: OpenGL 4.3 beta driver
  - www.nvidia.com/drivers
    - Grab the latest Beta drivers for your OS & GPU
- Developer resources
  - http://developer.nvidia.com/nv-path-rendering
  - Whitepapers, FAQ, specification
  - NVprSDK—software development kit
  - NVprDEMOs—pre-compiled Windows demos
  - OpenGL Extension Wrangler (GLEW) has API support
- Email: nvpr-support@nvidia.com

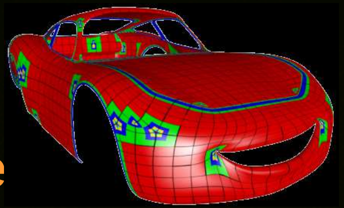
Don't Forget the 20th Anniversary Party

Date: August 8th 2012 (today!)
Location: JW Marriott Los Angeles at LA Live
Venue: Gold Ballroom – Salon 1

Other OpenGL-related NVIDIA Sessions at SIGGRAPH

- GPU Ray Tracing and OptiX
  - Wednesday in West Hall 503, 3:50 PM - 4:50 PM
  - **David McAllister**, OptiX Manager, NVIDIA
  - **Phillip Miller**, Director, Workstation Software Product Management, NVIDIA

- Voxel Cone Tracing & Sparse Voxel Octree for Real-time Global Illumination
  - Wednesday in NVIDIA Booth, 3:50 PM - 4:50 PM
  - **Cyril Crassin**, Postdoctoral Research Scientist, NVIDIA Research
- OpenSubdiv: High Performance GPU Subdivision Surface Drawing
  - Thursday in NVIDIA Booth, 10:00 AM - 10:30 AM
  - **Pixar Animation Studios GPU Team**, Pixar



- nvFX : A New Scene & Material Effect Framework for OpenGL and Dire
  - Thursday in NVIDIA Booth, 2:00 PM - 2:30 PM
  - **Tristan Lorach**, Developer Relations Senior Engineer, NVIDIA