NV_path_rendering Frequently Asked Questions

NVIDIA Corporation

June 13, 2011

The document collects answers to frequently asked questions about NVIDIA's NV_path_rendering OpenGL extension for GPU-accelerated path rendering.

1. What is NV_path_rendering?

NV_path_rendering is an OpenGL extension supported by CUDA-capable NVIDIA GPUs to GPU-accelerate path rendering.

2. What is path rendering?

Path rendering is a well-established resolution-independent approach to 2D computer graphics characterized by the specification of graphics objects as *paths*. These paths, sometimes called *outlines*, specify an object to render as a sequence of commands for drawing connected lines, curves, and arcs. These paths may be concave, may self-intersect, can contain holes, and may be arbitrarily complex.

Salient features of path rendering systems include the ability to both fill and stroke paths (see the figure below), to apply constant coloring as well as color gradients, restricting the rendering of one path to the region within another arbitrary so-called clipping path, arranging the rendering of paths into a hierarchy of objects with nested transformations, arranging paths into layers and blending among those layers, and embellishing the process of stroking paths with support for end caps, join styles, and dashing.

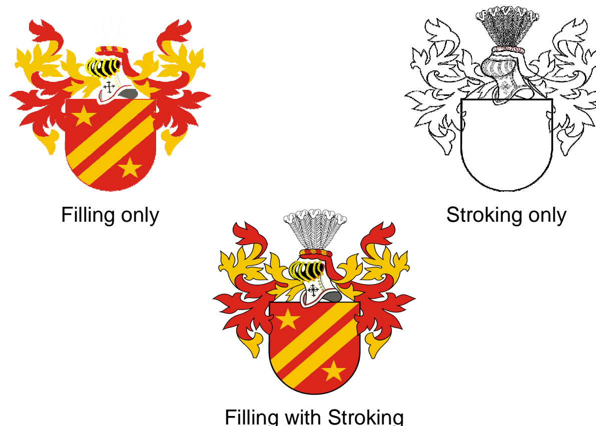

Figure 1: Example of how filling and stroking together complete a scene.

3. What existing standards support path rendering?

Numerous standards incorporate path rendering. Every time you open a PDF document or visit a web site relying on Flash or Silverlight, you are using path rendering. Drawings

in applications such as Adobe Illustrator or PowerPoint are represented in the resolution-independent manner of path rendering. PostScript, TrueType, and OpenType fonts all define the glyphs in fonts with paths. The latest HTML 5 standard for the web incorporates support for and Scalable Vector Graphics (SVG) standard and "canvas" rendering.

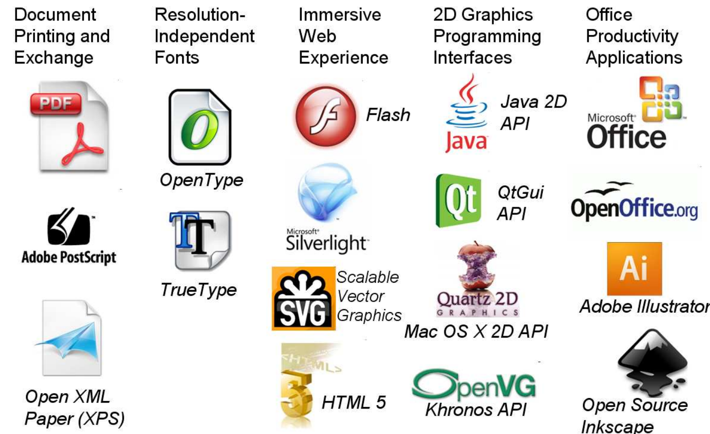

Figure 2: Various software standards based on path rendering in various categories.

4. Is NV_path_rendering yet another standard for path rendering?

No, instead NV_path_rendering is a means to GPU-accelerate existing path rendering standards on NVIDIA GPUs. Implementations of existing standards can re-target their path rendering to benefit from the substantial quality and performance benefits of GPU-acceleration. For example, a web browser could be implemented to render SVG through NV_path_rendering to achieve immersive web experiences at higher resolutions and quality levels than possible with only CPU-based path rendering.

Through study of the requirements of existing path rendering standards, NV_path_rendering provides the full gamut of path rendering functionality.

5. What drivers support NV_path_rendering?

Support for NV_path_rendering first appeared in the Release 275.33 drivers in June 2011.

All CUDA-capable NVIDIA GPUs support NV_path_rendering so GeForce 8 and later GPUs can all GPU-accelerate path rendering with the extension. GeForce 7 and earlier GPUs do not support NV_path_rendering.

NV_path_rendering supports Windows XP, Windows Vista, Windows 7, Linux, FreeBSD, and Solaris. Both 32-bit and 64-bit operating systems are supported.

6. Is there example source code demonstrating NV_path_rendering?

Yes. Download the NVIDIA Path Rendering SDK (**NVprSDK.zip**) that contains full source code for over a dozen interesting path rendering examples. Included in the SDK is an extended examples (**nvpr_svg**) that demonstrates GPU-accelerated SVG content compared to standard CPU-based path rendering APIs such as Cairo, Skia, Qt, and OpenVG as well as Direct2D.

7. Is there a tutorial to help understand the NVIDIA Path Rendering SDK examples?

Yes, read the "*Getting Started with NV_path_rendering*" whitepaper which walks through the SDK's **nvpr_whitepaper** example.

8. Are pre-compiled versions of the NVIDIA Path Rendering SDK examples available?

Yes. Download the **NVprDEMOs.zip** for pre-compiled 32-bit Windows binaries, ready to run on XP, Vista, or Windows 7.

9. How does NV_path_rendering work?

The NV_path_rendering OpenGL extension provides new API commands for specifying path objects. Each path object is a sequence of 2D drawing commands and their associated coordinates. Additional path parameters for each path object specify values for the stroke width, end cap style, line join style, and various dashing parameters.

Once a path is specified, the path is rendered through a combination of "stencil" and "cover" commands. Used in sequence, these commands implement a "stencil, then cover" (StC) method for rendering paths. There are two flavors of stencil and cover operations: filling and stroking.

The "stencil fill" step can identify all the sample locations within the framebuffer that are "inside" the filled region of an indicated path. The stencil samples of such locations are modified to indicate the location is within the filled path. The "stencil fill" operation only modifies the stencil state of the framebuffer; no color values are updated during the "stencil fill" step.

Next a subsequent "cover fill" step colors the pixels within the filled path based on the stencil information generated by the "stencil fill" step. The standard OpenGL stencil test is used to restrict the pixel coloring to the region within the filled path. This cover step can also reset the stencil values back to their pre-"stencil fill" state so the StC method can be repeated for any additional paths to be rendered. The cover step gets its name because the cover step guarantees covering enough of the framebuffer to guarantee any pixels with stencil values modified by the "stencil" step will be tested and colored as necessary by the cover step.

The "stencil stroke" and "cover stroke" operations operate in a similar manner, but the stencil values updated by the "stencil stroke" step are the points contained in the stroked region of the indicated transformed path. Likewise, the "cover stroke" step guarantees the rasterization of covering geometry that conservatively covers that stroked region.

The GPU's ability to alternate rapidly between the stencil and cover operations is vital to the ability of NVIDIA GPUs to implement the NV_path_rendering StC approach. In some cases, the GPU is over 100 times faster at complex path rendering than CPU-based path rendering approaches.

Other commands in the NV_path_rendering APIs support additional functionality such as geometric queries on paths, stenciling or covering batches of path objects in a single command, and managing path object names.

10. How is shading performed with NV_path_rendering?

During the cover step, the current OpenGL fragment processing is performed. This could be fixed-function processing or programmable shading performed by a programmable shader written in Cg, GLSL, or even assembly.

In conventional path rendering, shading is typically limited to solid coloring, color gradients, or image textures. With NV_path_rendering, there are no limitations on what kind of programmable shading for path rendering is possible. For example, the image below from the **nvpr_shaders** example shows a Cg shader applying a bump-mapped shader effect to path rendered text.

Figure 3: Path rendered text with a bump mapped shader applying per-pixel lighting.

11. Can NV_path_rendering mix with 3D rendering?

Yes. Path rendered content can be depth tested during the stencil and cover steps. This makes it possible to seamlessly mix conventional 3D GPU rendering with GPU-accelerated path rendering.

The image below from the **nvpr_tiger3d** example shows how a 3D wire-frame teapot can be mixed with path rendered content. The "stencil, then cover" approach of NV_path_rendering makes the mixing 3D and path rendering straightforward.

Figure 4: 3D rendering of a teapot mixed with path rendered tigers and overlaid path rendered text.

12. Can NV_path_rendering clip path rendering to other arbitrary paths?

Yes. The images below from the **nvpr_svg** example show how a tiger can be clipped to a heart path. By stenciling the clipping path into the stencil buffer first, and then performing stencil testing of the "stencil, then cover" steps to render the tiger, NV_path_rendering makes arbitrary path clipping straightforward and extremely efficient.

Figure 5: Clipping a complex path scene to an arbitrary clipping path.

13. How does NV_path_rendering fit into the existing OpenGL graphics pipeline?

Traditionally OpenGL has had a vertex pipeline for rendering geometric primitives such as triangles and a pixel pipeline for texture downloads and drawing images. NV_path_rendering adds a third path rendering pipeline to complement the existing vertex and pixel pipelines as shown in this figure.

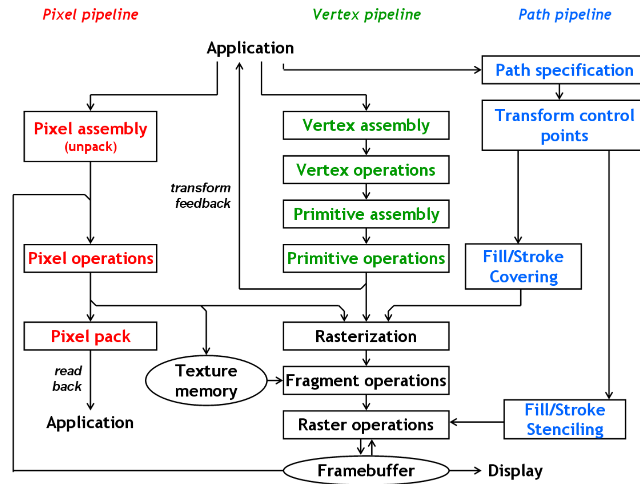

Figure 6: How the NV_path_rendering pipeline fits into the OpenGL API.

14. Does NV_path_rendering support arbitrary projective transformations of paths?

Conventional OpenGL supports arbitrary projective transformations of 3D content, however most path rendering standards constrain the supporting range of transformations of paths to 2D transformations for rotation, translation, scaling, and shearing, but not projection. NV_path_rendering extends the range of available transformation to the full variety of *3D* transformations *including* projection. This is one of the reasons NV_path_rendering can mix seamlessly with arbitrary 3D OpenGL rendering.

Just as shading computations performed on the GPU for 3D rendering are naturally perspective-correct, shading computations done in the cover step of NV_path_rendering usage is also fully perspective-correct.

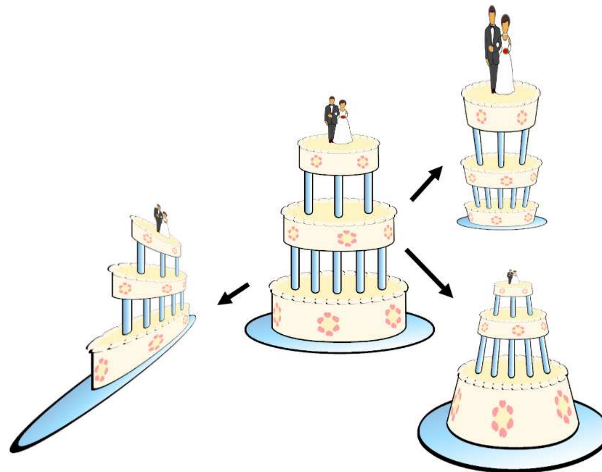

Figure 7: Examples of projective path rendering.

15. How are transformations specified with NV_path_rendering?

The standard OpenGL modelview and projection matrices and their associated matrix stacks control the transformation of paths for both the stencil and cover steps.

16. Does NV_path_rendering support vertex, geometry, or tessellation shaders?

No. Path objects are not specified with vertexes but rather with control points and other path coordinates corresponding to distinct path commands so these programmable shaders operating on vertexes do not make sense in a path rendering context.

This situation is not unique to NV_path_rendering but rather reflects the rendering paradigm encompassed by path rendering generally.

17. If there is no vertex shader, how do colors and texture coordinates for shading get generated?

NV_path_rendering has commands to generate colors, texture coordinates, and the fog coordinate as a linear combination of the path's coordinate space. Consistent with other path rendering standards such as SVG, the generated values can be a function either of path space or a normalized bounding box coordinate space and can further be a transformed version of this coordinate system. This is similar in approach to OpenGL's fixed-function texture coordinate generation facility (i.e. glTexGenfv, etc.). This process includes user-defined clip plane processing (using OpenGL's existing glClipPlane facility) and view frustum clipping.

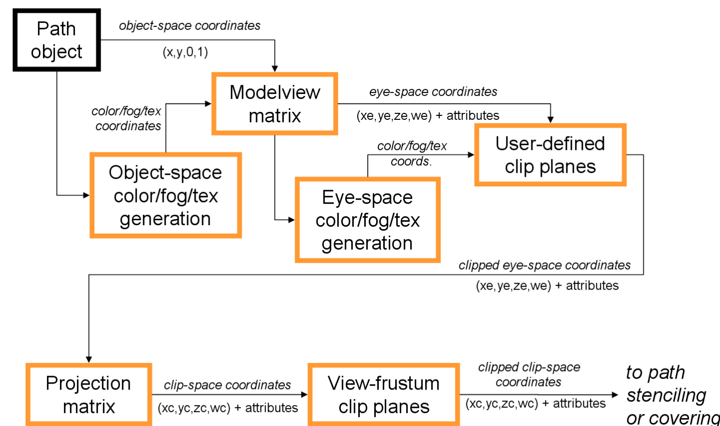

Figure 8: NV_path_rendering's path transformation, clipping, and coordinate generation pipeline.

18. So paths rendered by NV_path_rendering can be clipped by OpenGL clip planes?

Yes. And this clipping is fully consistent with the way 3D geometry is clipped by frustum and user-defined clip planes. Clip planes can be used to discard rendering that is trivially outside a clipping path or to restrict rendering to an arbitrarily transformed box.

The image below shows how a path rendering scene can be clipped by every combination of 6 user defined clip planes.

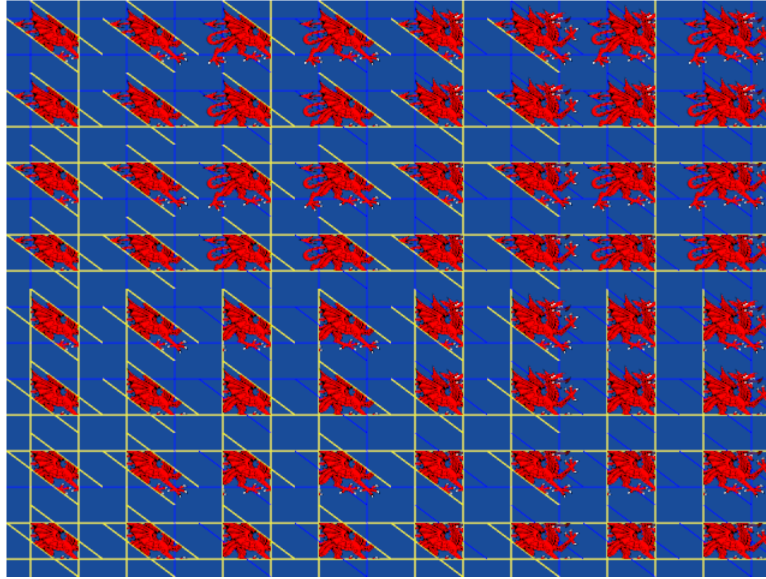

Figure 9: Scene showing a Welsh dragon clipped to all 64 combinations of 6 clip planes enabled & disabled.

19. How are path objects specified in NV_path_rendering?

Path objects can be specified in one of five ways:

- With a text strings conforming to a standard grammar to describe a path. NV_path_rendering supports both the SVG path syntax and the complete PostScript user path construction grammar.
- From data supplied by an array of path commands with a corresponding array of path coordinates.
- By naming ranges of Unicode character points from a specified font. The font can be named by its system name ("Palantino"), or a filename of a TrueType or PostScript font file (pala.ttf), or a standard font name ("Sans") guaranteeing the same glyphs outlines on every operating system supporting NV_path_rendering.
- By linear combination of two or more existing path objects with the same path command sequence.
- By linear transformation of an existing path object to a new coordinate system.

20. So does NV_path_rendering provide first-class resolution-independent font support?

Yes.

Basic glyph metrics can be queried from a path object created from a Unicode character point of a specified font.

NV_path_rendering provides additional stencil and cover commands for *instanced* path rendering where a sequence of path objects, each with their own independent transformation, can be either stenciled together or covered together by a single API command. This provides efficient rendering of arbitrary text. The sequence of path objects to rendered in an instanced batch can be specified in multiple formats including UTF-8 and UTF-16 strings.

NV_path_rendering provides a query for kerning separations suited to return an array of transformations suitable for rendering properly kerned instanced path objects for a string of text.

21. What specific glyph metrics does NV_path_rendering provide?

Per-glyph metrics and aggregate per-font face metrics are provided corresponding to the horizontal and vertical metrics provided by the FreeType2 library as shown in the image below:

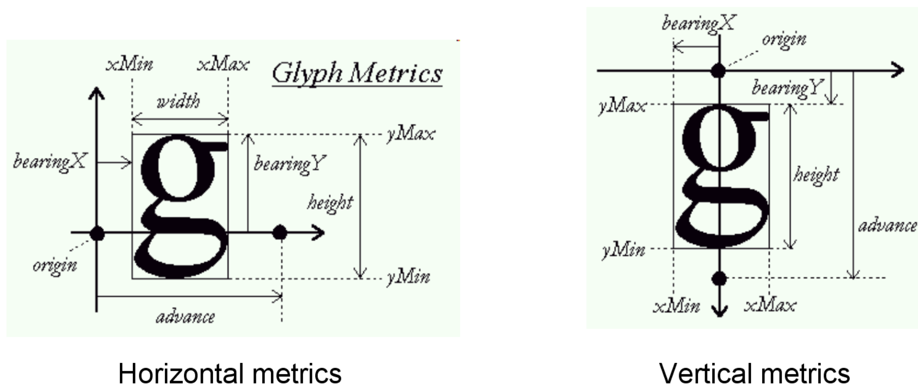

Figure 10: Available glyph metrics to query (image credit: FreeType 2 Tutorial).

22. Does NV_path_rendering use font hinting such as TrueType hints?

No, glyph outlines are generated simply from each character's master outline.

TrueType hinting is a process of improving the legibility of an outline given knowledge of the underlying device grid geometry. TrueType hinting makes sense when the mapping to device coordinates is known, fixed, and orthographic such as in a terminal window, word processor, or dialog box. The primary goal of font hinting is to optimize legibility, particularly when rendering to relatively coarse pixel grids; this forces a trade-off with geometric accuracy. NV_path_rendering is designed to render paths, including paths representing glyphs, under arbitrary transformations with excellent geometric accuracy.

Increasing screen density and resolution and high-quality antialiasing diminishes the need for TrueType hinting. Apple has recognized this ignoring almost all font hints in Mac OS X. If your application puts a premium on font legibility for small point sizes, NV_path_rendering may not deliver sufficient legibility. However if you require dynamically moving text under arbitrary transformations, NV_path_rendering is very well suited to fast font rendering.

Rather than assuming font hinting is a mandatory requirement for legible font rendering, evaluate the font rendering quality of NV_path_rendering with a sufficient number of samples per pixel (8 or more) and judge for yourself.

23. Can a path object be modified once created?

Yes.

glPathCoordsNV replaces the complete set of path coordinates of a specified path object with a new set. glPathSubCoordsNV replaces a sub-range of path coordinates with a new set. glPathSubCommandsNV deletes a specified number of commands and their corresponding coordinates and replaces them with a new (potentially different sized) set of path commands and corresponding coordinates.

Path parameters can also be modified once a path object is created.

24. Can all the application-specified state of a path object be queried?

Yes.

25. Does NV_path_rendering support all the standard stroking embellishments?

Yes.

NV_path_rendering supports round, square, flat (butt), and triangular end and dash caps. Distinct initial and terminal caps can be specified.

NV_path_rendering supports round, bevel, non-existent, PostScript-style mitered, and SVG-style mitered join styles with a configurable miter limit.

NV_path_rendering supports dashing with both a dash array (supporting both even and odd dash array lengths) and a dash offset. An OpenVG-style dash offset reset parameter specifies whether a MOVETO command resets the dash offset or simply continues the current dash pattern. Consistent with SVG, a client length parameter scales the dash array based on ratio of the client length to the path length computed by NV_path_rendering.

26. Does NV_path_rendering support cubic Bezier path commands?

Yes. Both conventional and smooth cubic Bezier paths are supported.

Cubic Bezier commands have both absolute and relative flavors.

27. Does NV_path_rendering support partial elliptical arc path commands?

Yes. Both SVG-style and OpenVG-style partial elliptical arcs are supported.

SVG-style arcs use an end-point parameterization based on seven coordinates.

OpenVG-style arcs use an end-point parameterization based on five coordinates where the clockwise/counter-clockwise and large/small arc flags are folded into the path command.

Partial elliptical arc commands have both absolute and relative flavors.

28. Does NV_path_rendering support PostScript-style circular arc path commands?

Yes. NV_path_rendering supports path commands corresponding to the PostScript ARC, ARCN, and ARCT path commands.

29. What is the sub-pixel quality of the resulting path rendering?

Modern GPUs maintain color, stencil, and depth state for several samples per pixel through a technique known as multisampling. NV_path_rendering exploits GPU multisampling to render antialiased paths. The specific number of samples per pixel is determined by the application. Maintaining 8 or 16 samples per pixel provides path rendering quality comparable to CPU-based path renderers. NVIDA GPUs support rendering to 32 samples per pixel as well.

Varying the number of samples per pixel provides a tradeoff between rendering quality and performance (and memory consumption). High-end GPUs can often support 16 and 32 samples per pixel with nominal performance degradation.

NVIDIA GPUs are designed to scatter the sample positions within a pixel (sometimes called jittered sampling). CPU-based path renderers typically rely on regular sample grids to minimize complexity and thereby improve performance. Jittered sampling grids are widely acceptable to provide better quality rendering results.

Requesting 4, 2, or even just 1 sample per pixel are also possible. GPUs are also flexible in their ability to render to off-screen surfaces or use the accumulation buffer to further improve the quality of path rendering.

Conventional CPU-based path renderers often maintain just a single color value per pixel. This leads to a host of quality artifacts. Among these is an artifact known as conflation where coverage information is conflated with opacity information leading to color

bleeding. NV_path_rendering is renders without conflation artifacts because coverage is maintained independently and accurately for every sub-pixel color sample.

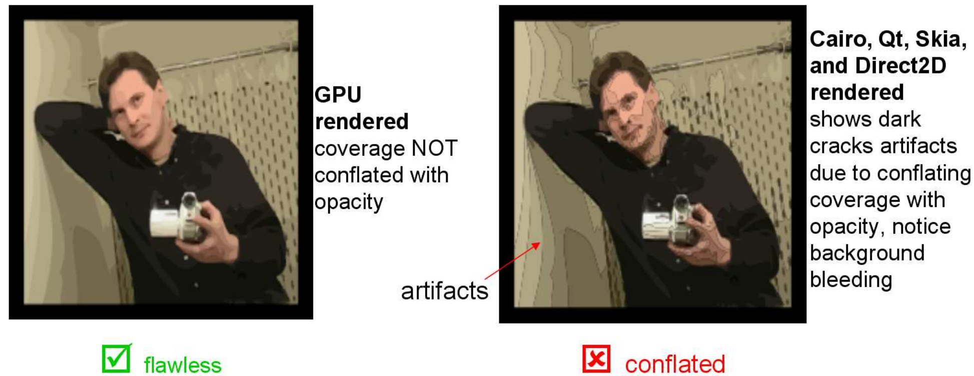

Figure 11: Conflation-free rendering with NV_path_rendering compared to other renderers.

30. What is the numerical quality of the path filling?

NV_path_rendering performs direct per-sample analytical winding number computations. The path outline is never approximated or tessellated into linear segments. This results in an extremely accurate determination of the path's winding number with respect to a given stencil location.

31. What is the numerical quality of the path stroking?

NV_path_rendering performs direct per-sample analytical point containment computations with respect to linear strokes and quadratic Bezier strokes and round features such as end caps or cusps. Stroked path edges are never approximated or tessellated into linear segments. Cubic Bezier segments and partial elliptical arcs are approximated to a high degree of accuracy to sequences of analytically accurate quadratic Bezier segments.

The image below shows the kind of stroking inaccuracies present in two widely used path rendering APIs compared to results rendered by NV_path_rendering and the OpenVG reference implementation.

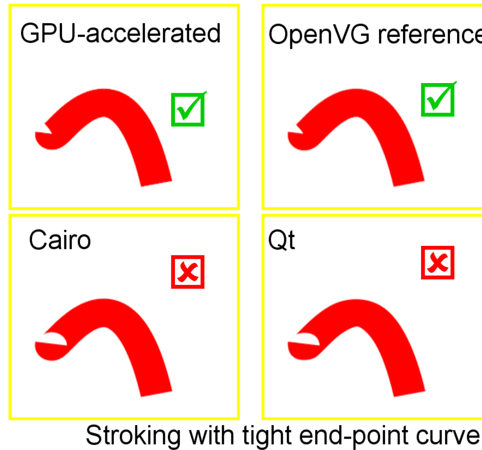

Figure 12: Stroking quality comparison.

32. Does NV_path_rendering fully accelerate path rendering into the sRGB color space?

Yes. Core OpenGL since version 3.0 supports both sRGB textures and framebuffers. All NVIDIA GPUs supporting NV_path_rendering support sRGB-correct blending and texture filtering. CPU-based path rendering systems do not typically properly correct blending and filtering because of the prohibitive expense, but sRGB correct rendering has a negligible cost for GPU-accelerated path rendering with NV_path_rendering. The image below shows the perceptually linear color response achieved by rendering with a sRGB-corrected color space.

Figure 13: Radial color gradient transitioning between saturated red and saturated blue.

33. How do I implement complex blend modes?

Path rendering systems often support complex blend modes such as SOFTLIGHT or COLORDODGE that cannot be implemented with the standard GPU blending functionality.

With "stencil, then cover" path rendering, these blend modes can be implemented within a programmable fragment shader that directly reads the sample's color value from framebuffer as a texture. While such access is typically undefined in OpenGL, NVIDIA's NV_texture_barrier extension provides well-defined behavior under the specific conditions that are guaranteed by "stencil, then cover" path rendering. This is because a glTextureBarrierNV operation can be performed prior to each "cover" step when drawing a path. What the texture barrier operation guarantees is that all caches and

pipelines within the GPU that might hold texture or framebuffer data have been flushed. Execution of the glTextureBarrierNV command is pipelined so is relatively inexpensive in the overall execution of your path rendering. Your shader still has to guarantee you read just the color value corresponding to the fragment shader execution.

All drivers supporting NV_path_rendering also support NV_texture_barrier.

34. If I use Direct3D in my application, how do I use NV_path_rendering?

While NV_path_rendering is an OpenGL extension, NVIDIA provides a set of DirectX interoperability extensions allowing OpenGL rendering to be directed into Direct3D surfaces as well as allowing OpenGL to texture from Direct3D textures. See the OpenGL WGL_NV_DX_interop extension for more details.

35. Is doing the two-step "stencil, then cover" process slow?

No; it's actually quite efficient and faster than other approaches. When the driver implementation is optimized for this pattern, the two-step "stencil, then cover" process is extremely efficient.

NVIDIA GPUs in particular are designed to double their rasterization throughput when performing simple stencil-only rendering. NVIDIA GPUs are also extremely efficient at discarding shading and further fragment processing when the stencil test fails.

If you play with the **nvpr_svg** example in the NVprDEMOs.zip, you'll find that NV_path_rendering is typically many times (in a few cases even 100x faster) than other well-known path renderers. This includes even includes Direct2D.

36. Still couldn't one step be faster than two?

The process of rendering path really is a two-step process. Determining the coverage for an arbitrarily complex path constructed from curved edges that may or may not self-intersect or form holes is a difficult task. Separating this task from the distinct shading task (performed during the cover step) makes the streaming processor array of CUDA cores in an NVIDIA GPU operate very coherently. The GPU can pipeline stenciling and covering steps within the GPU so they operate in true parallel for maximum performance.

For applications that wish to minimize application overhead, the stencil and cover commands to render a particular path object can be display listed in OpenGL. NVIDIA's OpenGL driver is optimized for multi-core systems so that when a multi-core system executes a display list, the driver effort can be forwarded to a second thread of execution. If you play with toggling display listing in the **nvpr_svg** example, you can observe as much as a further 20% speed-up over non-display listed path rendering.

There are secondary benefits from decoupling the coverage determination for path rendering (the "stencil" step) from the shading (the "cover" step). Rendering a path that is clipped to an arbitrary path is straightforward. You stencil the clipping path into the stencil buffer, cover the clipping path to (instead of shading) transfer the net clipping result to the high-bit of the stencil buffer, then stencil the rendered path into the stencil

buffer’s lower bits while testing against the high-bit. When the rendering path is covered, you’ll skip shading the clipped pixels.

37. Why is Direct2D slower if it also uses the GPU?

While Direct2D does leverage the GPU to accelerate path rendering, the standard Direct2D path rendering algorithm remains CPU-bottlenecked.

With the NV_path_rendering approach, path objects are “baked” into a resolution-independent form once and then path rendered entirely on the GPU from that point on. This allows the NV_path_rendering approach to depend more fully on the GPU for its acceleration.

38. How do I get my specific questions about NV_path_rendering answered?

Send email to **nvpr-support@nvidia.com**