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.
- 3D rendering is typically drawn using projective 3D viewing transformations; path rendering typically restricts its drawing to affine 2D transformations.
- 3D rendering is off-loaded from the CPU to GPUs that are extremely fast; path rendering is traditionally performed by slower CPU-based rendering.

The last reason is probably the most frustrating one. If 3D rendering happens on the GPU in video memory while path rendering is done by the CPU is system memory buffers, it is very hard to “mix” the two types of rendering. Even when there are APIs, such as OpenVG or Direct2D designed to be GPU accelerated, these APIs do not naturally allow 3D and path rendering to mix in a single depth-buffered 3D scene.

Mixing means more than just compositing a path-rendered image into a framebuffer containing 3D rendering. Mixing means the 3D rendering and path rendering should depth test with respect to each other while not losing the blended layering effects crucial to path rendering. Mixing also means the same programmable fragment shaders (also known as pixel shaders) used to shade 3D rendering can also shade path rendering.

NV_path_rendering makes path rendering a first-class rendering option within the OpenGL graphics system so now you really can mix high-performance path rendering and 3D in the same scene.

Mixing first-class 3D rendering and path rendering enables a whole new level of user interface approaches. Once 3D and path rendering mix freely, there's no need to separate 2D and 3D API elements. For example, both types of elements can seamlessly co-exist in a stereo environment.

Motivational Images

Figure 1 shows an example of four instances of the classic PostScript tiger arranged in a 3D scene with the one equally classic 3D teapot. Notice that not only is the teapot properly occluding the tigers, but the tigers are mutually occluding each other *and* occluding the teapot.

Figure 1: Depth-buffered 3D scene of classic path rendered PostScript tigers surrounding the Utah teapot.

The tigers are not simply images of the tiger stored in a texture. As shown in Figure 2, the eye lashes of the tiger (indeed all the details in each tiger) are drawn from resolution-independent curved paths. You can zoom into any detail in the scene and without ever getting pixilated results.

Figure 2: Zoomed view demonstrating the resolution-independent nature of the tigers

Depth Buffering Both 3D Rendering and Path Rendering

Rendering these scenes above is simpler than you might guess.

Framebuffer Configuration

The scene is depth buffered so the example must allocate a depth buffer and clear it every frame. The scene also uses "stencil, then cover" path rendering using NV_path_rendering so the framebuffer configuration also needs a stencil buffer. The scene is also antialiased using multisampling with 8 samples per pixel. The GLUT command below allocates such a framebuffer is

```
glutInitDisplayString("rgb depth stencil~4 double samples~8");
```

(The tilde operator in the string above says allocate at least the following number of stencil bits or sample but favor framebuffer configurations as close as possible to the requested number.)

Clear Configuration and Clearing

The example sets the values to clear the color, depth, and stencil values:

```
glClearColor(0.1, 0.3, 0.6, 0.0);  // nice blue background
glClearDepth(1.0);
glClearStencil(0);
```

Every frame requires clearing the color and depth buffers. However the stencil buffer only needs clearing if the window has been reshaped or damaged. Assuming a Boolean variable called **`force_stencil_clear,`** indicating when a stencil clear is necessary, this code clears the buffers before each frame:

```
if (force_stencil_clear) {
  glClear(GL_COLOR_BUFFER_BIT |
          GL_STENCIL_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT);
  force_stencil_clear = false;
} else {
  glClear(GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT);
}
```

Initial Viewing Conditions

NV_path_rendering uses the same OpenGL modelview and projection matrix state used for conventional fixed-function vertex processing. So you can manipulate these transformations the same way you would manipulate them for 3D rendering.

First, a GLUT reshape callback (called when the window is first realized and when resized) establishes a conventional 3D viewing frustum and viewport

```
static void reshape(int w, int h)
{
  glViewport(0,0, w,h);
  window_width = w;
  window_height = h;
  float aspect_ratio = window_width/window_height;
  glMatrixLoadIdentityEXT(GL_PROJECTION);
  float near = 1,
        far = 1200;
  glMatrixFrustumEXT(GL_PROJECTION, -aspect_ratio,aspect_ratio,
    1, -1, near,far);
  force_stencil_clear = true;
}
```

This frustum contains everything in eye-space from 1 unit from the eye to 1,200 units from the eye. Our 3D scene with the tigers and teapot will be positioned within this viewing frustum.

(**`glMatrixLoadIdentityEXT`** and **`glMatrixFrustumEXT`** are selector-free versions of **`glLoadIdentity`** and **`glFrustum`** introduced by the EXT_direct_state_access extension; uses these commands avoids mistakenly depending on the prior matrix mode selector set by **`glMatrixMode`**. EXT_direct_state_access is always available when NV_path_rendering is supported.)

When we initialize OpenGL rendering state, we translate the camera location down the negative Z axis so we can view the scene centered at the origin in world space:

```
glMatrixLoadIdentityEXT(GL_MODELVIEW);
glMatrixTranslatefEXT(GL_MODELVIEW, 0,0,ztrans);
```

`ztrans` is initially -150 units.

3D Transformations can transform Path Object Rendering

Further 3D modeling transformations will be configured when rendering the teapot and each tiger instance. These additional per-object 3D transformations can be concatenated onto the projection and modelview transformations configured above.

Each path making up the tiger is lies in a plane and so is fundamentally 2D. In other words, each path is "flat" in the sense that the path is specified with 2D (x,y) positions, without a Z axis or any sense of 3D. Yet these 2D (x,y) positions can be treated as 3D homogeneous positions by simply reinterpreting them as $(x,y,0,1)$ positions. This is similar to the way you can use **`glVertex2f(x,y)`** and **`glVertex2f(x,y,0,1)`** interchangeably. By promoting the coordinate space of path objects from 2D positions to homogeneous 3D positions, paths can be transformed by arbitrary 3D projective transformations.

Such 3D transformations of paths treated this way generate Z values that can be transformed into window-space depth coordinates and depth tested with respect to any other geometric primitives (or paths!) rendered with OpenGL 3D rendering. These paths can also be clipped by clip planes. Varying window-space W coordinate values also facilitate perspective-correct interpolations of attributes such as colors and texture coordinates.

Rendering State Configuration

Depth buffering provides the proper 3D occlusion so enable depth testing:

```
glEnable(GL_DEPTH_TEST);
```

For NV_path_rendering, stencil testing is needed during the "cover" steps when drawing the tiger paths. In expectation of using stencil testing, configure the stencil testing state so that zero acts as a neutral stencil value for filling and stroking operations:

```
glStencilFunc(GL_NOTEQUAL, 0, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
```

In other words, if the cover operations identify non-zero stencil values in the framebuffer, these correspond to color samples within the fill or stroke of the path and so are rendered (instead of being discarded). Such rendered samples are also reset to a stencil value of zero in expectation of the next path to be rendered.

Configuring a Depth Offset for Path Stenciling

All the steps so far have are quite ordinary. When stenciling the paths that make up each tiger instance, 3D transformations will generate window-space depth values that can be depth tested against previously rendered depth values.

But path-rendered objects, such as the tigers, are typically made up of many layers and this introduces a potential issue. Each tiger is really 140 paths layers, with each one layered atop the prior path. In the terminology of depth testing, these layers would all be approximately co-planar when transformed into window space ahead of depth testing. This will result in ambiguous depth testing results as different paths of the tiger wind up

slightly in front or slightly behind other paths due to numerical differences in the transformation of each path.

OpenGL has a mechanism known as *depth offset*, more commonly known as *polygon offset*, for dealing with similar problems with co-planar and nearly co-planar triangles. Essentially OpenGL can bias forward or backward depth values based on depth offset state consisting of an integer bias term specified in units of depth precision and an additional floating-point terms based on the maximum X and Y slope of the polygonal primitive in window space. The **glPolygonOffset** command configures this state.

For path rendering, NV_path_rendering introduces additional depth offset state that applies to stenciling paths objects. The path stenciling depth offset is configured like this:

```
GLfloat slope = -0.05;
GLint bias = -1;
glPathStencilDepthOffsetNV(slope, bias);
```

The depth offset state set by **glPolygonOffset** only applies to rendering polygonal primitives in OpenGL (as its name implies); however the depth offset set by **glPathStencilDepthOffsetNV** applies only to stenciled paths. The reason for having a special depth offset for path stenciling will be clear when the entire 3D rendering process for paths is explained.

Notice only a very small slope offset (-0.05) and a single unit of bias offset (-1) are needed to ensure proper 3D rendering of paths.

Keep in mind that during the path stencil operations, *only* the stencil buffer is going to be updated. While the depth test is performed during path stencil, the depth buffer is *not* being written. The depth test can only restrict the accessibility of the samples to be updated by stenciling of the path. If the depth test fails for a given framebuffer stencil sample otherwise covered by the filled or stroked region of a path, the depth test is going to discard these stencil updates. The path stenciling depth offset therefore is simply offsetting the interpolated depth value at a sample position which could change the depth test result, but it is not ever going to write that offset depth value into the depth buffer. Importantly, the depth offset can allow the depth test to pass during a path stencil operation when the path's plane in 3D space is co-planar with prior paths rendered into the same plane.

Configuring a Depth Function for Path Covering

During path stencil operations, such as performed by NV_path_rendering's **glStencilFillPathNV** and **glStencilStrokePathNV** commands, the standard OpenGL depth test is applied, if enabled through **glEnable(GL_DEPTH_TEST)**, using the standard depth test function set by **glDepthFunc**.

For reasons that will make sense upon further explanation, the cover step for rendering paths, such as performed by the **glCoverFillPathNV** and **glStencilFillPathNV** commands, has its own depth function for depth testing that is distinct from the standard depth function used for path stenciling and rendering all other OpenGL rendering. For

rendering the tiger paths in 3D, the depth function for path covering should be first specified as:

```
glPathCoverDepthFuncNV(GL_ALWAYS);
```

This means whether or not the depth test passes or fails during the path covering step, *always* pass the depth test *and* write the depth value (when the depth test is enabled).

Eureka: 3D Path Rendering!

So why does this enable 3D path rendering?

Think about what happens when we perform the two steps of "stencil, then cover" path rendering with the following state configured:

```
glEnable(GL_DEPTH_TEST);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_NOTEQUAL, 0, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
GLfloat slope = -0.05;
GLint bias = -1;
glPathStencilDepthOffsetNV(slope, bias);
glPathCoverDepthFuncNV(GL_ALWAYS);
```

During the path stenciling step, the depth test is performed but using an interpolated depth value offset based on the **`slope`** and **`bias`** values above. If the depth test fails at a sample, that means there is no stencil update for the sample location. This is crucial because if there is no change in the stencil value of a covered stencil location, the stencil test configured during the path cover step is not going to update the location's color value.

But if the stencil test does pass (implying the stencil test passed in the prior "stencil" step for the path), that means the enabled depth test is also going to pass during the following path "cover" step (because the path coverage depth function is **`GL_ALWAYS`** so it cannot fail). This means the depth value of the covered sample passing the stencil test is going to be written. Notice this is *not* an "offset" version of the depth value, but a "true" interpolated depth value that is written to the depth value.

This is desirable because the offset depth value is not really correct, but simply offset enough to avoid co-planar depth testing artifacts. As we layer all 140 layers of each tiger, different depth values will be written (all essentially co-planar), but no offset depth values are written. Even if 10,000 layers were drawn, it isn't going to "advance forward" the depth of the path rendered layers.

This also means the generated depth values could be used for other purposes such as constructing shadow maps or performing constructive solid geometry (CSG) operations. The point is an accurate depth buffer is being generated through the described 3D path rendering process.

Is This Really Necessary?

To help appreciate that this really works, it helps to see what happens if you render the scene *without* the crucial **`glPathStencilDepthOffsetNV`** command as shown in Figure

3. Each tiger becomes a mess of co-planar depth fighting because the depth testing during the stencil steps is ambiguous after the first layer is rendered.

Figure 3: Scene rendered with (left) and without (right) stencil step's proper depth offset.

Drawing the Tigers

Here is the code fragment to draw the four tigers:

```
float separation = 60;
glEnable(GL_STENCIL_TEST);
int numTigers = 4;
for (int i=0; i<numTigers; i++) {
  float angle = i*360/numTigers;
  glMatrixPushEXT(GL_MODELVIEW); {
    glRotatef(angle, 0,1,0);
    glTranslatef(0, 0, -separation);
    glScalef(0.3, 0.3, 1);
    renderTiger(filling, stroking);
  } glMatrixPopEXT(GL_MODELVIEW);
}
```

Notice that each tiger is drawn with a different modelview matrix transformation (rotate/translate/scale) pushed on the modelview stack to draw each tiger.

The **`renderTiger`** routine is not special. The same **`renderTiger`** routine can draw a simple 2D version of the PostScript tiger as shown in Figure 4. It simply loops over the 140 paths that make up the tiger performing pairs of **`glStencilFillPathNV`** and **`glCoverFillPathNV`** operations for each path and, if the path is also stroked, **`glStencilStrokePathNV`** and **`glCoverStrokePathNV`** for each such stroked path.

Figure 4: Tiger rendered by **`renderTiger`** with a simple 2D orthographic view by the **`nvpr_tiger`** example in the NVIDIA Path Rendering SDK.

Drawing the Teapot

So far, the state configuration and drawing has been for rendering the path-rendered tigers. This is primarily because nothing, except remembering to not leave the stencil test enabled, is required to render conventional 3D objects such as the teapot into the scene.

The following code draws the teapot:

```
glDisable(GL_STENCIL_TEST);
glColor3f(0,1,0);

glMatrixPushEXT(GL_MODELVIEW); {
  glScalef(1,-1,1);
  if (wireframe_teapot) {
    glutWireTeapot(teapotSize);
  } else {
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glutSolidTeapot(teapotSize);
    glDisable(GL_LIGHTING);
  }
} glMatrixPopEXT(GL_MODELVIEW);
```

The code draw the teapot as a solid object with conventional lighting applied or draw the teapot in green wireframe. Figure 5 shows a scene with a green wireframe teapot.

Figure 5: An enlarged green wireframe teapot protruding through an arrangement of five tigers to show how depth testing is correct even for complicated wireframe geometry.

Notice that the teapot and tigers both have valid depth values in the OpenGL window's single depth buffer. This makes it possible for you to mix any conventional 3D rendering with any NV_path_rendering-based path rendering. There is no performance penalty for mixing the two rendering paradigms.

Because depth testing is order-independent, the tigers and teapot could be rendered in an arbitrary order and the scene would be rendered correctly.

Overlaid Path Rendered Text

And you do not have to limit yourself to depth-tested path rendering mixed with the 3D objects. You can use path rendering to overlay fancy resolution-independent text too. Figure 6 shows how you can do this. When rendering the text, depth testing is simply disabled by calling `glDisable(GL_DEPTH_TEST)`. The text shown has a linear color gradient applied to the filling while each letter is also highlighted by an underlying stroke version of each glyph in gray.

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.

Conventional Path Rendering Shading is Simple and Non-programmable

The tiger artwork is simple in its shading. Each of the 140 layered paths that make up the tiger uses constant color shading; but arbitrarily complex shading of path rendered content is also possible with "stencil, then cover" path rendering.

Conventional path rendering also uses linear and radial color gradients and image texturing as demonstrated in Figure 7. While these examples may appear 3D, they are really creations of a skilled 2D artist rather than truly 3D.

Figure 7: Examples of path rendering content relying on linear and radial color gradients.

Using Arbitrary Programmable Fragment Shaders to Cover Paths

NV_path_rendering allows the same 3D shaders that make modern 3D scenes so visually rich also available for shading path rendering. The exact same fragment shader that you use for 3D content can be applied to path rendering too. How you author the shader is totally up to you. You can use Cg, GLSL, assembly extensions, or fixed-function fragment processing; the choice is yours.

Figure 8 shows a simple example of applying a per-pixel bump mapping fragment shader, written in Cg, to some path rendered text. The text says “Brick wall!” so it makes sense to shade the text with a fragment shader that applies a normal map texture encoding the brick-and-mortar surface variations of a brick wall surface. The diffuse lighting on the wall responds to a circling light source. Flat regions of the normal map are shaded red while bumpy regions are shaded white. The yellow dot in the scene indicates the light source position. Notice how the diffuse lighting is darker further away from the light source.

Figure 8: Path rendered brick wall with bump map brick fragment shader applied.

Zooming into the scene, we can highlight the difference in shading when the light source is nearby versus when it is far away as shown in Figure 9.

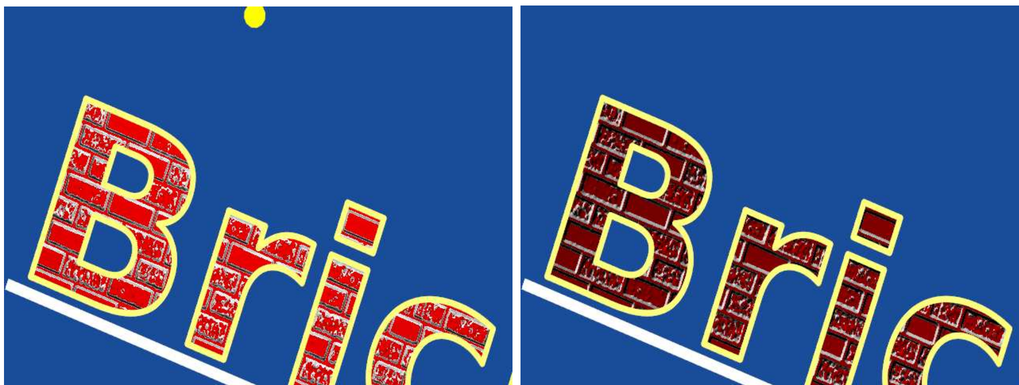

Figure 9: Left zoomed scene shows nearby light; right scene shows far away light.

With the "stencil, then cover" approach of NV_path_rendering, configuring a fragment shader is simple. During the "cover" step, however OpenGL is currently configured for fragment shading determines how the covered path is shaded.

Brick Wall Bump Map Shader Example

For this particular example, the Cg fragment shader is:

```
float3 expand(float3 v)
{
  return (v-0.5)*2;  // Expand a range-compressed vector
}

void bumpmap(float2 normalMapTexCoord : TEXCOORD0,
             float3 lightDir          : TEXCOORD1,

         out float4 color : COLOR,

     uniform float3 lightPos,
     uniform sampler2D normalMap)
{
  // Normalizes light vector with normalization cube map
  float3 light = normalize(lightPos - float3(normalMapTexCoord,0));
  // Sample and expand the normal map texture
  float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
  float3 normal = expand(normalTex);
  // Diffuse lighting
  float diffuse = dot(normal,light);

  // Decide the brick color based on how flat (red)
  // or angled (red) the surface is.
  float3 red = float3(1,0,0);
  float3 white = float3(1,1,1);
  float3 brick = normal.z > 0.9 ? red : white;

  color = float4(brick*diffuse,1);
}
```

The brick wall normal map pattern accessed by the **`normalMap`** sampler is stored in the a 2D texture such that the signed normal vectors are range-compressed into unsigned RGB color values as shown in

Figure 10.

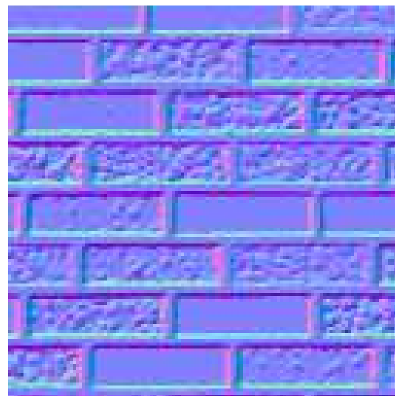

Figure 10: Normal map texture image.

Configuring the Shader for "Stencil, then Cover" Path Rendering

This application code renders the filled glyphs saying "Brick Wall!":

```
const char *message = "Brick wall!"; /* the message to show */
messageLen = strlen(message);
glStencilFillPathInstancedNV((GLsizei)messageLen,
                             GL_UNSIGNED_BYTE, message, glyphBase,
                             GL_PATH_FILL_MODE_NV,
                             ~0,  /* Use all   stencil bits */
                             GL_TRANSLATE_X_NV, xtranslate);

const GLfloat coeffs[2*3] = { 10,0,0, 0,1,0 };
glPathTexGenNV(GL_TEXTURE0,
  GL_PATH_OBJECT_BOUNDING_BOX_NV, 2, coeffs);

cgGLBindProgram(myCgFragmentProgram);
cgGLEnableTextureParameter(myCgFragmentParam_normalMap);
cgGLEnableProfile(myCgFragmentProfile);

glCoverFillPathInstancedNV((GLsizei)messageLen,
                           GL_UNSIGNED_BYTE, message, glyphBase,
                           GL_BOUNDING_BOX_OF_BOUNDING_BOXES_NV,
                           GL_TRANSLATE_X_NV, xtranslate);

cgGLDisableProfile(myCgFragmentProfile);
```

The **cgGL**-prefixed routines are part of the Cg Runtime API for configuring the Cg fragment shader for use during OpenGL rendering.

The **glStencilFillPathInstancedNV** and **glCoverFillPathInstancedNV** commands take the string message as an array of path objects name offsets. These offsets, each corresponding to an ASCCI character, will be added to **glyphBase** to generate a sequence of path objects corresponding to a range of glyphs loaded for a standard sans-serif font. (The initialization of these glyph path objects is not shown here but in the example code.) The array **xtranslate** is an array of GLfloat values that indicates the absolute X translation of each glyph from the next in the sequence. NV_path_rendering includes a query **glGetPathSpacingNV** to make it easy to compute a set of kerned glyph offsets for a string of characters. (For more details about text rendering with NV_path_rendering, see the *Getting Started with NV_path_rendering* whitepaper.)

Generating Texture Coordinates for the Cover Step

Unlike 3D rendering, path rendering does not use vertex-level shading operations because paths are not specified with vertices. (Instead paths are specified with path commands and their associated coordinates. Technically, a path specifies a 2D trajectory or contour on a logical 2D plane so the kind of vertex-level operations such as vertex, geometry, and tessellation shading applied to meshed polygonal geometry are not applicable to paths.)

Without conventional vertex processing and their associated per-vertex attributes, there needs to be some other way to generate linearly varying attributes such as colors, texture coordinate sets, and the fog coordinate over a path's filled or stroked region. NV_path_rendering includes a set of commands to generate such attributes from linear

functions of the object-space, eye-space, or bounding box coordinates of the paths being rendered.

In the example code above, the **glPathTexGenNV** command, similar in spirit to the conventional **glTexGenfv** command, computes the texture coordinates for the rendering as a function of the bounding box of the instanced path objects. This command configures the texture coordinate generation:

```
const GLfloat coeffs[2*3] = { 10,0,0, 0,1,0 };
glPathTexGenNV(GL_TEXTURE0,
  GL_PATH_OBJECT_BOUNDING_BOX_NV, 2, coeffs);
```

With path texture coordinate generation configured by this command, the s and t texture coordinates are computed based on the linear coefficients in the **coeffs** array as

$$s = 10\times x + 0\times y + 0$$
$$t = 0\times x + 1\times y + 0$$

where x and y are positions within the normalized [0,1] range of the bounding box. This scales the [0,1] ×[0,1] square to a [0,10] ×[0,1] region. This mapping is because the text "Brick Wall!" has about a 10-to-1 aspect ratio. This means the normal map texture tiles (repeats) ten times horizontally.

Alternative attribute generation modes to **GL_PATH_OBJECT_BOUNDING_BOX_NV** are **GL_OBJECT_LINEAR** and **GL_EYE_LINEAR** to generate attributes as a linear combination of object-space object coordinates or eye-space (post-modelview matrix transform) coordinates.

Easy to Modify Shading and Font Choices

Once the shading of path rendered content is under programmable control, it becomes much easier to modify and control the resulting path rendering. By modifying the Cg shader, the text coloration is easy to change. For example, these alternative lines modify how the **brick** color is computed:

```
float3 green = float3(0,1,0);
float3 magenta = float3(1,0,1);
float3 brick = normal.z > 0.9 ? green : magenta;
```

Figure 11 shows the result of using this modified Cg shader and picking the standard "Serif" font.

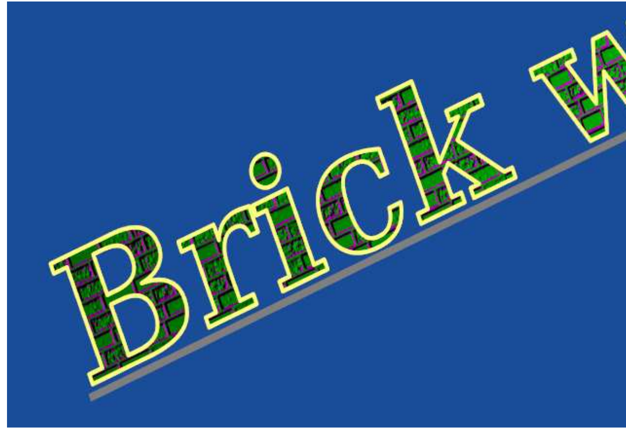

Figure 11: Rendering result with a modified shader and different font choice.

More Information

The source code for the examples shown is freely available in the NVIDIA Path Rendering SDK. **`nvpr_tiger3d`** is the example rendering the tigers and teapot shown in Figure 1, Figure 2, Figure 3, Figure 5, and Figure 6. **`nvpr_tiger`** is the simple 2D example shown in Figure 4. **`nvpr_shaders`** is the programmable shading example shown in Figure 8, Figure 9, and Figure 11.

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