Non-Photorealistic Rendering with Pixel and Vertex Shaders

Drew Card
3D Application Research Group
ATI Research

Jason L. Mitchell
3D Application Research Group
ATI Research

Introduction

Development of cutting-edge graphics techniques like programmable pixel and vertex shaders are often motivated by a desire to achieve photorealistic renderings for gaming or simulation. In this paper, we apply vertex and pixel shaders to Non-Photorealistic Rendering (NPR). In many types of images such as cartoons and technical drawings, photorealism is not desirable. In the case of technical illustrations, non-photorealistic rendering techniques are used to enhance understanding of the scene or object being drawn without obscuring important features such as outlines. In other cases, we simply hope to simulate other media such as cel-shaded cartoons, wood-block prints or hatched line drawings for stylistic purposes. In the following sections, we will apply Direct3D pixel and vertex shaders to implement and extend recent research efforts in non-photorealistic rendering.

Rendering Outlines

Rendering of object outlines is a common step in non-photorealistic rendering. In this section, we will present a geometric approach to outline rendering, which uses vertex shaders to determine silhouette edges. These outlines are used *with* the NPR shading techniques discussed in subsequent sections. An image-space approach to outlining will be presented at the end of the paper.

We consider the silhouette outline to be the minimum set of lines that is needed to represent the contour and shape of the object. Silhouette edges represent more than just the outer edges of the object but also points of surface discontinuity (e.g. a sharp edge or crease in a surface).

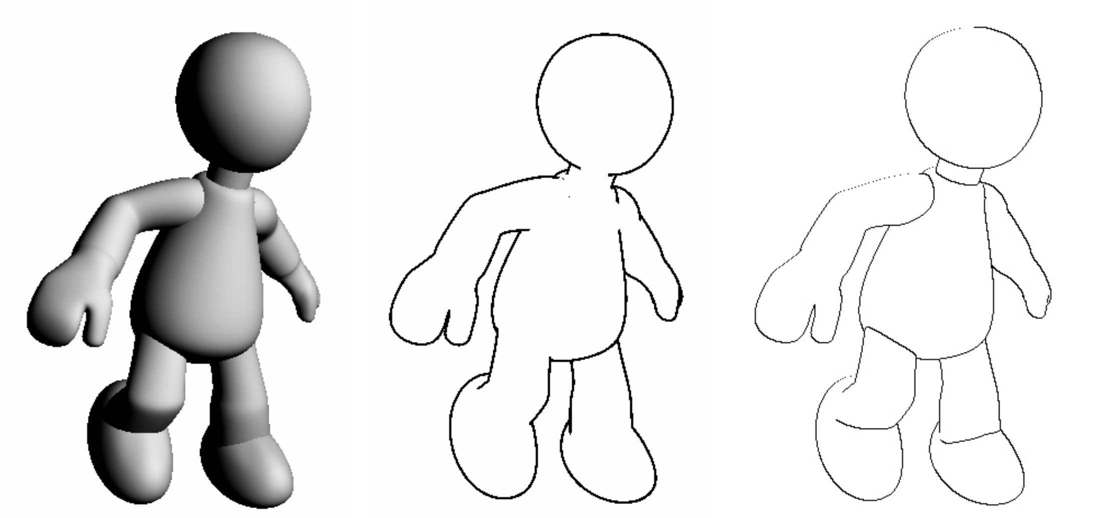


Figure 1 – N·L, silhouette outlines, image filtered outlines

The basic algorithm involves ~~drawing every edge~~ of an object as a quadrilateral fin and doing the silhouette determination in the vertex shader. This is very similar to the way that [Lengyel01] faded the “fins” used in fur rendering. The goal of this shader is very simple; if the edge is a silhouette, the vertex shader renders the quad fin, otherwise the vertex shader renders a degenerate (unseen) fin.

The vertex shader determines if an edge is a silhouette by comparing the face normals of the triangles that share the edge (n_{face0} and n_{face1}). If one normal is front facing with respect to the viewer and the other is back facing, then the edge is a silhouette. This algorithm works perfectly except in the case of edges that are not shared by more than one triangle. These kinds of edges are considered “boundary edges” and need to be drawn all of the time. Boundary edges only have one face normal associated with them, so there is no second normal to be used in the comparison. In order to ensure that boundary edges are always drawn, the second shared normal is chosen such that it is the negation of the first normal ($n_{face1} = -n_{face0}$). This results in boundary edges always being drawn since one normal will always be facing the viewer and the other always facing away from the viewer. With this organization of data, one vertex buffer and one rendering call can be used to draw all of the quad fins and the vertex shader will handle both regular edges and boundary edges.

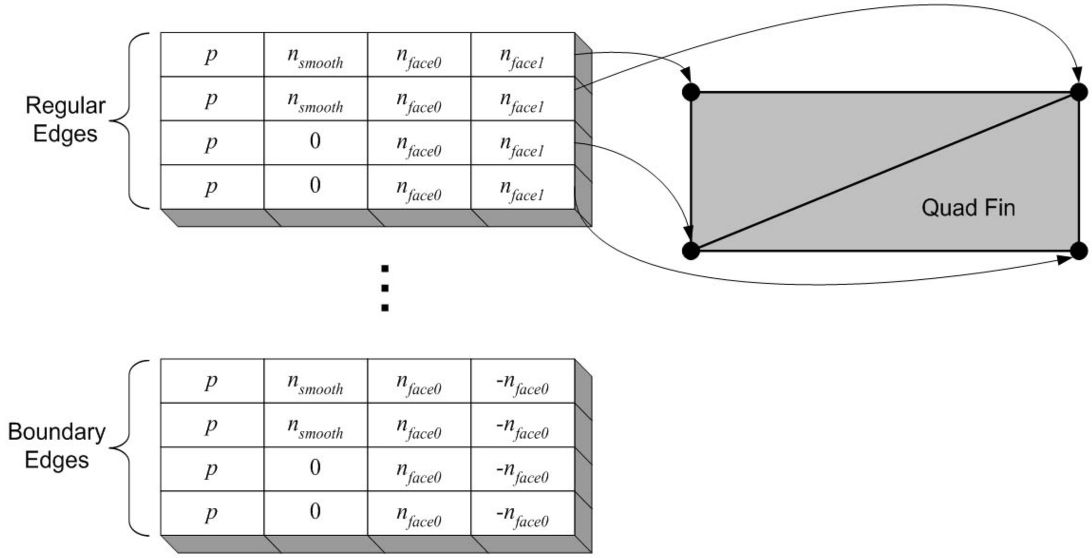


Figure 2 – Vertex buffer organization for silhouette edge rendering

The data for drawing the object will typically be stored in memory in an organization similar to that shown in Figure 2. As shown, every vertex *of every edge* is composed of the vertex position (p) along with three normal values corresponding to the vertex's smooth normal (n_{smooth}) and the face normals of the triangles sharing the edge (n_{face0} and n_{face1}). The application should then render each edge quad fin, possibly passing the additional normal information in a separate data stream from the rest of the vertex. In order to reduce the memory bandwidth hit from reading in the "extra" data (n_{face0} and n_{face1}), one optimization would be to quantize the face normals to byte 4-tuples or short 4-tuples.

camera space

The vertex shader computes the view vector in camera space by multiplying the vertex by the view matrix and normalizing. The shader then transforms the face normals n_{face0} and n_{face1} into camera space and dots them with the view vector. If the edge is a silhouette edge, one of these dot products will be negative and the other will be positive. The shader checks for this condition by multiplying the two dot products together and checking for a value less than zero. If the value is less than zero, the vertex offset is set to zero (unchanged), otherwise the vertex offset is set to one. The vertex offset is then multiplied by the smooth normal and added to the untransformed vertex position. Note that two of the four vertices for each quad fin have $n_{smooth} = 0$. This acts as a mask of the fin vertex displacement and causes two of the fin vertices to stick to the model while the other two are displaced to cause the fin to be visible.

```
// NPR outline shader
// c0-3         view matrix
// c4-7         view projection matrix
// c8
// c9           (0.0, 0.0, 0.0, 1.0f)
// c10          line width scalar

vs.1.1
m4x4    r0, v0, c0   // compute the view vector
dp3     r1, r0, r0   // normalize the view vector
rsq     r1, r1
mul     r0, r0, r1

m3x3    r1, v7, c0   // multiply normal 1 by the view matrix
m3x3    r2, v8, c0   // multiply normal 2 by the view matrix
dp3     r3, r0, r1   // dot normal 1 with the view vector
dp3     r4, r0, r2   // dot normal 2 with the view vector
mul     r3, r3, r4   // multiply the dot products together
slt     r3, r3, c9   // check if less than zero

mov     oD0, c9      // set the output color

dp4     r0, v0, c6   // compute the vertex depth
mul     r0, r0, c10  // multiply by a line thickness scalar
mul     r3, r3, r0   // multiply the thickness by the smooth normal

mul     r3, v3, r3   // multiply by the normal offset
add     r0, v0, r3   // add in the offset
mov     r0.w, c9.w   // swizzle in a one for the w value
m4x4    oPos, r0, c4 // transform the vertex by the model view projection
```

Listing 1 – Outline vertex shader

Hidden line removal is handled via the depth buffer. We assume that a shaded version of the model is rendered before the outlines to fill the z buffer with values that will cause hidden outlines to fail the z test. The following pseudo code outlines this process:

1. Preprocess the geometry into quad fins
   a. For each vertex of each edge store the edge vertex, the smooth surface normal, and the two face normals which share said edge; One should have the smooth normal and the other should have the smooth normal field set to zero
   b. If edge is unshared (only used in one face) store the negation of the one face normal as the second normal
2. Render a shaded version of the geometry to fill the z buffer
3. Enable outline vertex shader and initialize the shader constant storage
4. Set up stream mappings to pass in the additional normal data
5. Render the edges as triangles
6. Vertex shader breakdown:
   a. Compute the view vector by transforming the vertex into eye space (multiply by the view matrix) and normalize
   b. Dot each face normal with the view vector
   c. Multiply the resulting dot products together

d. Check for a value less than zero
e. Multiply the smooth normal by the result of the less than zero test
f. Compute the vertex depth (dot the vertex with the third row of the view projection matrix)
g. Multiply the vertex depth by the line thickness factor to get a normal scale value
h. Multiply the smooth normal by the normal scale value
i. Add the smooth normal to the untransformed vertex
j. Transform the vertex and output

There are some drawbacks associated with the previous algorithm. Along with the hassle of preprocessing the geometry and storing extra edge data, boundary edges may potentially be drawn incorrectly when a quad fin points straight at the viewer. This is because the algorithm currently only scales the edge along the smooth surface normal, therefore leaving no means to screen-align the quadrilateral edge. This could be addressed by re-working the algorithm to also screen-align the quad. Later in this paper, we present an image-space approach to rendering outlines, which requires no preprocessing and does not exhibit the same boundary edge issue.

In the next section, we will discuss methods for shading the interior of the object to achieve different stylized results.

Cartoon Lighting Model

One method of cartoon shading is to create banded regions of color to represent varying levels of lighting. Recent examples of 3D games using cel-shading techniques are *Cel Damage* by Pseudo Interactive and the *Jet Set Radio* games (called *Jet Grind Radio* in some markets) by Sega/Smilebit. A common technique illustrated in [Lake00] is a technique called *hard shading* which shades an object with two colors that make a hard transition where $N \cdot L$ crosses zero. [Lake00] indexes into a 1D texture map to antialias the transition, while the method shown here computes the colors analytically. Figure 3 shows an approach which uses three colors to simulate ambient (unlit), diffuse (lit) and specular (highlight) illumination of the object.

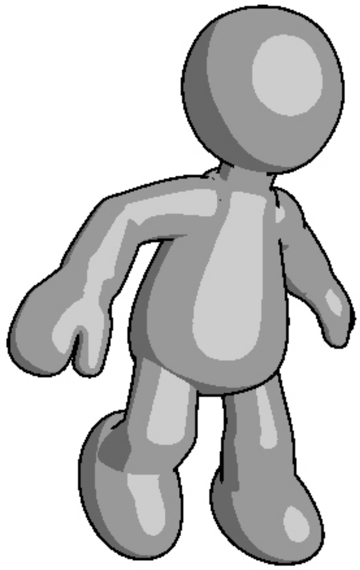


Figure 3 - Cartoon shaded with outlines

This is accomplished using a vertex shader that computes the light vector at each vertex and passes it into the pixel shader as the first texture coordinate. The vertex shader also computes the half angle vector at each vertex and passes it into the pixel shader as the second texture coordinate. The pixel shader analytically computes the pixel color. As shown in the listing below, the pixel shader first computes N · L and N · H. If the N · L term is above a specified threshold, the diffuse color is output, otherwise the ambient color is output. If the N · H term is above a specified threshold, then the specular color replaces the color from the N · L term. This same analytical method could be expanded to use any number of banded regions.

```
// Cartoon vertex shader
// c9 is the light position
// c10 is the view projection matrix
// c14 is the view matrix
vs.1.1

// output the vertex multiplied by the mvp matrix
m4x4 oPos, v0, c10

// compute the normal in eye space
m3x3  r0, v3, c14
mov   oT0, r0 // write the normal to tex coord 0

// compute the light vector
sub   r0, c9, v0
dp3   r1, r0, r0
rsq   r1, r1
mul   r0, r0, r1
m3x3  r1, r0, c14 // transform the light vector into eye space
mov   oT1, r1 // write the light vector to tex coord 1

// compute half vector
m4x4  r0, v0, c14 // transform the vertex position into eye space
dp3   r3, r0, r0 // normalize to get the view vector
rsq   r3, r3
mul   r0, r0, r3
```

```
add   r0, r1, -r0 // add the light vector and the view vector = half angle
dp3   r3, r0, r0 // normalize the half angle vector
rsq   r3, r3
mul   r0, r0, r3
mov  oT2, r0 // write the half angle vector to tex coord 2
```

Listing 2 – Cartoon shading vertex shader code

```
// Cartoon shading pixel shader
//
ps.1.4

def c0, 0.1f, 0.1f, 0.1f, 0.1f // falloff 1
def c1, 0.8f, 0.8f, 0.8f, 0.8f // falloff 2
def c2, 0.2f, 0.2f, 0.2f, 1.0f // dark
def c3, 0.6f, 0.6f, 0.6f, 1.0f // average
def c4, 0.9f, 0.9f, 1.0f, 1.0f // bright

// get the normal and place it in register 0
texcrd r0.xyz, t0

// get the light vector and put it in register 1
texcrd r1.xyz, t1

// compute n dot l and place it in register 3
dp3 r3, r0, r1

// subtract falloff 1 from the n dot l computation
sub r4, r3, c0

// check if n dot l is greater than zero
// if yes use average color otherwise use the darker color
cmp_sat r0, r4, c3, c2

// subtract falloff 2 from the n dot l computation
sub r4, r3, c1

// check if n dot l is greater than zero
// if yes use bright color otherwise use whats there
cmp_sat r0, r4, c4, r0
```

Listing 3 – Cartoon shading pixel shader code

The ambient and diffuse bands help to visualize the shape of the object while the specular highlight gives insight into the properties of the surface of the object. If the goal of the cartoon shader is only to represent the object's shape then the shader could omit the specular portion and replace it with any number of additional diffuse regions.

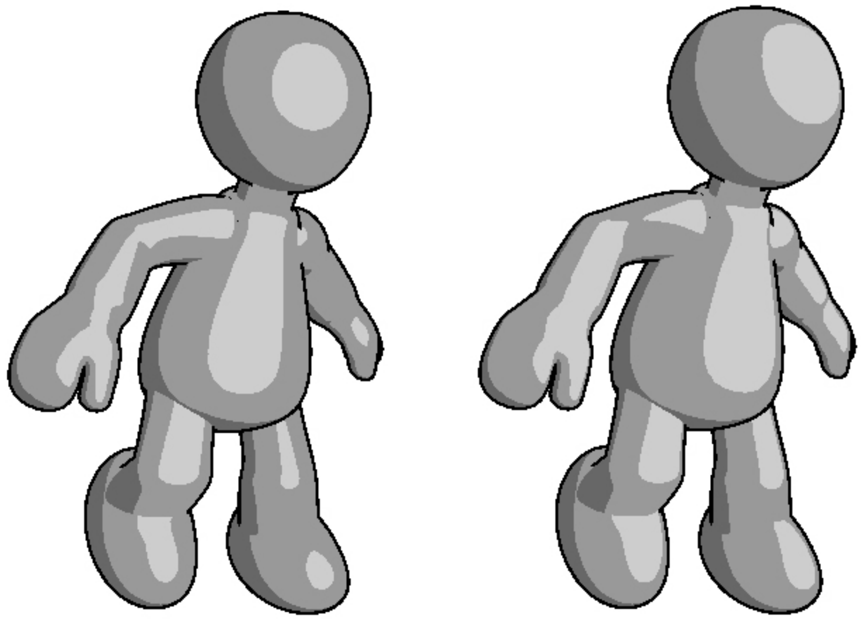


Figure 4 - Cartoon shaded object with specular and with multiple diffuse regions

Hatching

Another method of NPR shading is hatching, which is commonly used in pen and ink drawings to show shape and differentiate between lit and unlit regions of an object. The density of the hatch pattern signifies how much light the surface is reflecting at that point. The current state of the art in real-time hatching is illustrated in [Praun01]. This technique uses an array of hatch patterns ranging from very sparse (well lit) to very dense (unlit) hatching called tonal art maps. N · L is computed per-vertex and used to determine a weighted-average of the tones in the tonal art map. Per-pixel, the tonal art maps are blended together according to the weights interpolated from the vertices. The result is a hatched image which is well antialiased.

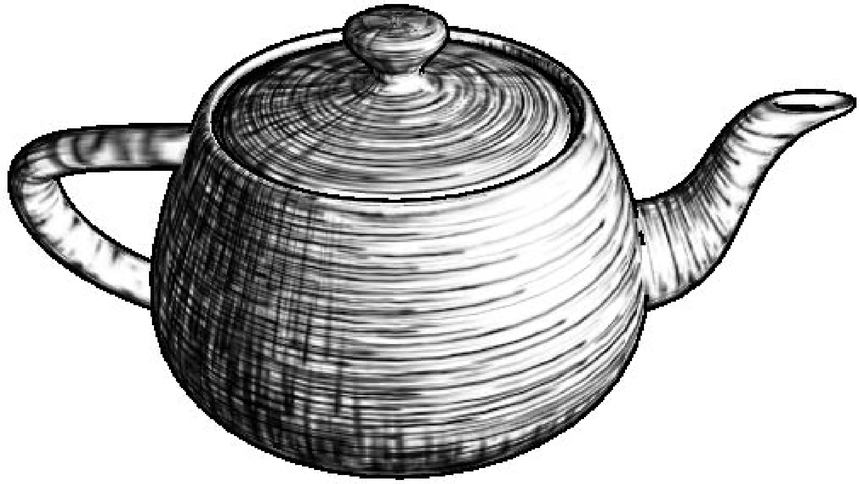


Figure 5 - Hatched object

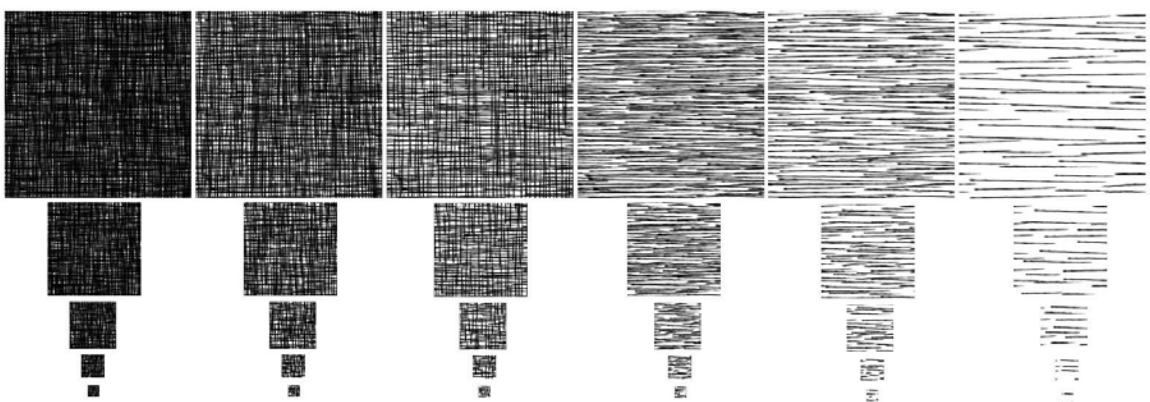


Figure 6 - Tonal Art Map from [Praun01]

```
// Hatching vertex shader
//
// c0  0 (0.0, 0.0, 0.0, 0.0)
// c1  1 (1.0, 1.0, 1.0, 1.0)
// c2  2 (2.0, 2.0, 2.0, 2.0)
// c3  3 (3.0, 3.0, 3.0, 3.0)
// c4  4 (4.0, 4.0, 4.0, 4.0)
// c5  5 (5.0, 5.0, 5.0, 5.0)
// c6  6 (6.0, 6.0, 6.0, 6.0)
// c7  7 (7.0, 7.0, 7.0, 7.0)
// c8  brightness
// c9  light position
// c10 view projection matrix
// c14 view matrix
//
vs.1.1

m4x4   oPos, v0, c10 // output the vertex multiplied by the mvp matrix

mov    oT0, v7 // write out the texture coordinate
mov    oT1, v7

mov    r1, v3 // normalize the normal
mov    r1.w, c0
dp3    r2, r1, r1
rsq    r2, r2
mul    r1, r1, r2

sub    r2, c9, v0 // compute light vector and normalize
dp3    r3, r2, r2
rsq    r3, r3
mul    r2, r2, r3

dp3    r3, r2, r1 // compute the light factor (n dot l) times six clamp at zero
mul    r3, r3, c8

mov    r5.x, c5.x // seed the blend weights
mov    r5.y, c4.x
mov    r5.z, c3.x
mov    r5.w, c0.x

mov    r6.x, c2.x
mov    r6.y, c1.x
mov    r6.z, c0.x
mov    r6.w, c0.x

sub    r5, r3, r5 // sub each weights initial value from the light factor
sub    r6, r3, r6

max    r5, r5, c0 // get rid of everything less than zero
sge    r7, c2, r5 // flag all weights that are <= 2
mul    r5, r5, r7 // zero out weights > 2
sge    r7, r5, c1 // flag all weights that are >= 1
mul    r7, r7, c2 // subtract all weights that are greater than or equal to one from 2
sub    r5, r7, r5
slt    r7, r5, c0 // flag all weights that are < 0 and negate
sge    r8, r5, c0 // flag all spots that are >= 0
add    r7, -r7, r8 // add the flags
mul    r5, r5, r7 // should negate the negatives and leave the positives

max    r6, r6, c0 // same as above only on the second set of weights
sge    r7, c2, r6
mul    r6, r6, r7
sge    r7, r6, c1
mul    r7, r7, c2
sub    r6, r7, r6
slt    r7, r6, c0
sge    r8, r6, c0
```

```
add    r7, -r7, r8
mul    r6, r6, r7

sge    r8, c1, r3 // check for total shadow and clamp on the darkest texture
mov    r7, c0
mov    r7.z, r8.z
add    r6, r6, r7
min    r6, r6, c1

mov    oT2.xyz, r5 // write the 123 weights into tex coord 3
mov    oT3.xyz, r6 // write the 456 weights into tex coord 4
```

Listing 4 – Hatching Vertex Shader

```
// Hatching pixel shader
ps.1.4

texld   r0, t0          // sample the first texture map
texld   r1, t1          // sample the second texture map
texcrd  r2.rgb, t2.xyz // get the 123 texture weights and place it in register 2
texcrd  r3.rgb, t3.xyz // get the 456 texture weights and place it in register 3
dp3_sat r0, 1-r0, r2   // dot the reg0 (texture values) with reg2 (texture weights)
dp3_sat r1, 1-r1, r3   // dot the reg1 (texture values) with reg3 (texture weights)
add_sat r0, r0, r1     // add reg 0 and reg1
mov_sat r0, 1-r0       // complement and saturate
```

Listing 5 – Hatching Pixel Shader

Gooch Lighting

The Gooch lighting model introduced in [Gooch98] is designed to provide lighting cues without obscuring the shape of the model, the edge lines or specular highlights. This technique, designed to model techniques used by technical illustrators, maps the -1 to 1 range of the diffuse $N \cdot L$ term into a cool-to-warm color ramp. This results in diffuse lighting cues which are shown as hue changes rather than color intensity changes. This diffuse lighting model is designed to work *with* the silhouette and feature-edge lines discussed earlier in this paper. It essentially results in a reduction in the dynamic range of the diffuse shading so that the edge lines and specular highlights are never obscured. A similar technique is used in the game *Half-Life* by Valve Software [Birdwell01]. The *Half-Life* engine first computes a single approximate aggregate light direction. The -1 to 1 result of the per-vertex N.L from this aggregate light direction is then scaled and biased into the 0 to1 range rather than simply clamped at zero. This eliminates the flat look that would otherwise be apparent on the side of a game character that faces away from the light.

As shown in [Gooch98], the classic lighting equation can be generalized to the following, which allows us to experiment with cool-to-warm colors.

$$I = \left(\frac{(1+n\cdot l)}{2}\right) * k_{warm} + \left(1 - \frac{(1+n\cdot l)}{2}\right) * k_{cool}$$

$$k_{warm} = k_{yellow} + \beta * k_d$$

$$k_{cool} = k_{blue} + \alpha * k_d$$

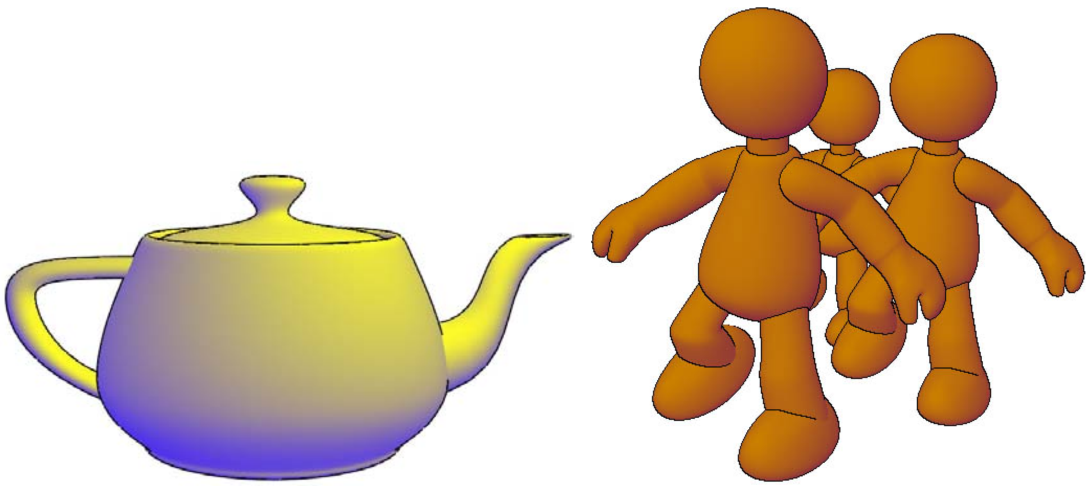


Figure 7 – Gooch lighting with outlining on a teapot and cartoon characters

```
// Gooch Lighting vertex shader
// c9 is the light position
// c10 is the view projection matrix
// c14 is the view matrix
vs.1.1

m4x4   oPos, v0, c10 // output the vertex multiplied by the vp matrix
sub    r0, c9, v0 // compute the light vector and normalize
dp3    r1, r0, r0
rsq    r1, r1
mul    r0, r0, r1
mov    r1, v3  // compute the normal
mov    oT0, r1 // write the normal to tex coord 0
mov    oT1, r0 // write the light vector to tex coord 1
```

Listing 6 – Gooch Lighting Vertex Shader

```
// Gooch Lighting pixel shader
// c0 is alpha (eg. {0.4, 0.4, 0.4, 1.0})
// c1 is beta (eg. {0.5, 0.5, 0.5, 1.0})
// c2 is kyellow (eg. {0.5, 0.5, 0.0, 1.0})

// c4 is kd
// c5 is (1.0, 1.0, 1.0, 1.0)
ps.1.4

texcrd r0.xyz, t0  // get the normal and place it in register 0
texcrd r1.xyz, t1  // get the light vector and put it in register 1
dp3    r3, r0, r1  // compute n dot l and place it in register 3
add_d2 r3, r3, c5  // normalize the n dot l range

mul_sat r0, c4, c0 // compute the cool factor
add_sat r0, r0, c2
mul_sat r0, r0, r3

mul_sat r1, c4, c1 // compute the warm factor
add_sat r1, r1, c3
mad_sat r0, r1, 1-r3, r0 // add the warn and cool together and output
```

Listing 7 – Gooch Lighting Pixel Shader

In the preceding sections, we have concentrated on shader techniques which render non-photorealistic images directly into the frame buffer. In the following section, we will look at image-space techniques which require rendering into textures and subsequent processing of these rendered images to produce non-photorealistic images.

Image-Space Techniques

As discussed in "Image Processing with 1.4 Pixel Shaders in Direct3D," it is possible to render 3D scenes into textures for subsequent image processing. One technique developed in [Saito90] and refined in [Decaudin96], is to render the depth and world-space normals of objects in a scene into a separate buffer. This rendered image is subsequently post-processed to extract edges which can be composited with a hatched, Gooch shaded or cartoon shaded scene. We will show a Direct3D implementation of this technique as well as our own extension which thickens the lines using morphological techniques. One advantage of an image-space approach to determining outlines is that it is independent of the rendering primitives used to render the scene or even whether the models are particularly well-formed. A scene which contains N-Patch primitives [Vlachos01], for example, will work perfectly well with an image-space approach as will interpenetrating geometry such as the classic Utah teapot. This approach even works with user clip planes (or the front clip plane), correctly outlining areas of normal or depth discontinuity in the final image, without any application intervention at the modeling level. Another advantage is that this approach does not require the creation and storage of the auxiliary outline buffers discussed in the first section of this paper.

The first step of this technique is to use a vertex shader to render the world space normals and depths of a scene into a texture map. The vertex shader scales and biases the world space normals from the -1 to 1 range into the 0 to 1 range and writes them to diffuse *r*, *g* and *b* (**`oD0.xyz`**). The eye-space depth is written into diffuse alpha (**`oD0.w`**). This interpolator is simply written out to the RGBA render target (i.e. a texture) by the pixel shader. One important detail is that the clear color for the scene should be set to world-space +z so that the filter will interact properly with the objects at all orientations. An image of some cartoon characters rendered with this technique is shown in the following figures. The RGBA texture containing world space normals and eye-space depths is shown in Figure 8.

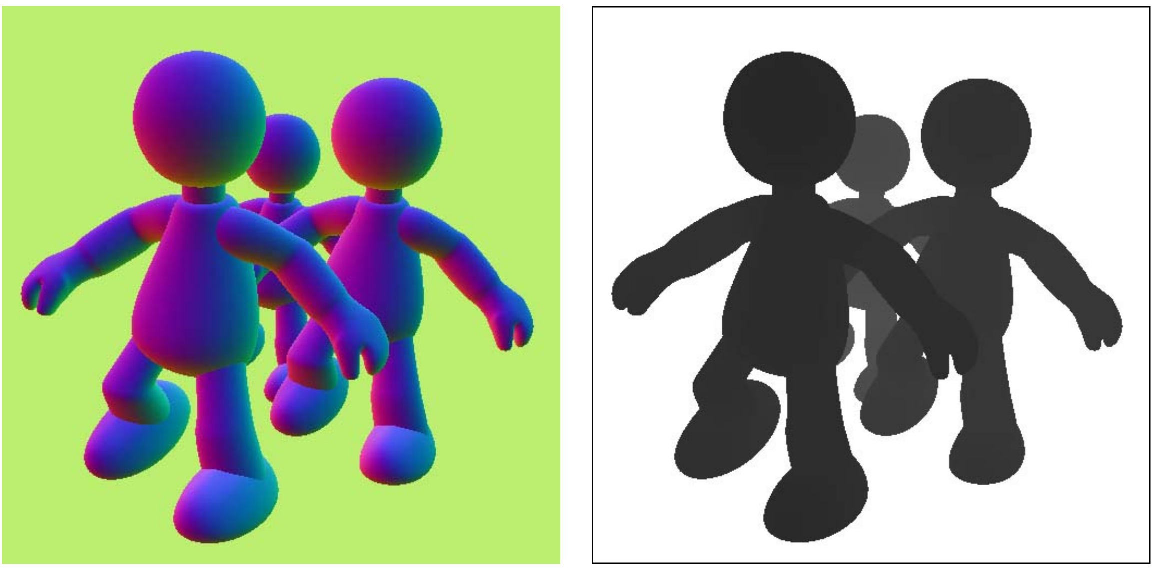


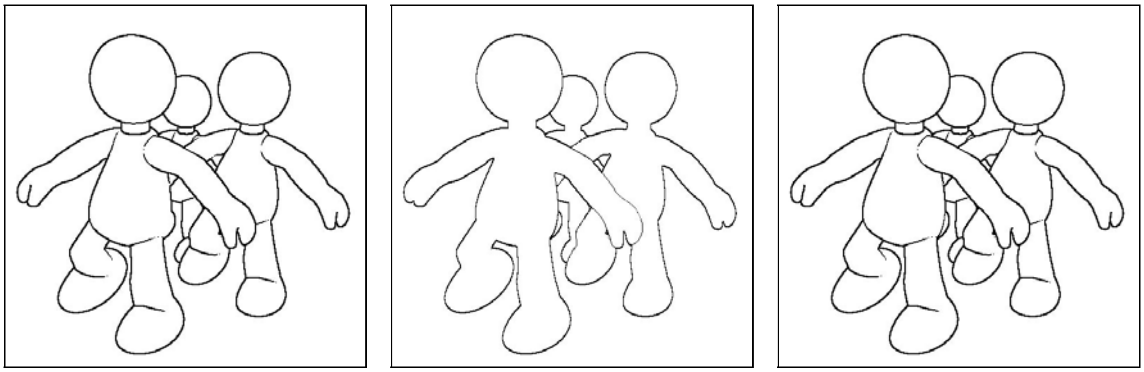
Figure 8 – World Space Normals and Eye Space Depth as in [Saito90] and [Decaudin96]. These are rendered to RGB and A of a renderable texture map.

The vertex shader used to render this scene is shown in Listing 8:

```
vs.1.1
m4x4 oPos, v0, c0
mov   r0, v3
mov   r0.w, c12.w
add   r0, c8, r0
mul   r0, r0, c9
m4x4  r1, v0, c4
sub   r1.w, r1.z, c10.x
mul   r0.w, r1.w, c11.x
mov  oD0, r0
```

Listing 8 – World Space Normals and Eye Space depth vertex shader

The next step is to use image processing techniques to extract the desired outlines. The normals and depths are effectively processed independently to isolate different classes of edges. Discontinuities in the normals occur at internal creases of an object, while depth discontinuities occur at object silhouettes, as shown in Figure 9. Note that while the normal discontinuity filter picks up the edge at the top of the leg and the depth discontinuity filter does not, the union of the edge pixels from the two filters produces a reasonable outline for the object.



(a) (b) (c)

Figure 9 – (a) Edges from world space normal discontinuities, (b) depth discontinuities and (c) both

```
 // Normal discontinuity filter for Non-Photorealistic Rendering
 ps.1.4
 def c0,  1.0f,  1.0f, 1.0f, 1.0f
 def c1, -0.85f, 0.0f, 1.0f, 1.0f
 def c2,  0.0f,  0.0f, 0.0f, 0.0f

 // Sample the map five times
 texld r0, t0 // Center Tap
 texld r1, t1 // Down/Right
 texld r2, t2 // Down/Left
 texld r3, t3 // Up/Left
 texld r4, t4 // Up/Right
 dp3 r1.rgb, r0_bx2, r1_bx2  // Dot products with center pixel (Signed result -1 to 1)
 dp3 r2.rgb, r0_bx2, r2_bx2
 dp3 r3.rgb, r0_bx2, r3_bx2
 dp3 r4.rgb, r0_bx2, r4_bx2

 // Subtract threshold
 add r1, r1, c1.r
 add r2, r2, c1.r
 add r3, r3, c1.r
 add r4, r4, c1.r

 phase

 // Make black/white based on threshold
 cmp r1.rgb, r1, c0.r, c2.r
+mov r1.a, c0.a
 cmp r2.rgb, r2, c0.r, c2.r
+mov r2.a, c0.a
 cmp r3.rgb, r3, c0.r, c2.r
+mov r3.a, c0.a
 cmp r4.rgb, r4, c0.r, c2.r
+mov r4.a, c0.a

 mul r0.rgb, r1, r2
 mul r0.rgb, r0, r3
 mul r0.rgb, r0, r4
+mov r0.a, r0.r
```

Listing 9 – Determining edges from an image of a scene's world space normals

```
// 5-tap depth-discontinuity filter
ps.1.4
def c0, -0.02f, 0.0f, 1.0f, 1.0f

texld r0, t0 // Center Tap
texld r1, t1 // Down/Right
texld r2, t2 // Down/Left
texld r3, t3 // Up/Left
texld r4, t4 // Up/Right

add r1, r0.a, -r1.a     // Take four deltas
add r2, r0.a, -r2.a
add r3, r0.a, -r3.a
add r4, r0.a, -r4.a

cmp r1, r1, r1, -r1     // Take absolute values
cmp r2, r2, r2, -r2
cmp r3, r3, r3, -r3
cmp r4, r4, r4, -r4

phase

add r0.rgb, r1, r2     // Accumulate the absolute values
add r0.rgb, r1, r3
add r0.rgb, r1, r4

add r0.rgb, r0, c0.r  // Subtract threshold
cmp r0.rgb, r0, c0.g, c0.b
+mov r0.a, r0.r
```

Listing 10 – Determining edges from an image of a scene's eye-space depth

Compositing the Edges

Once we have the image containing the world-space normals and depth, we composite the edge-filtered result with the frame buffer which already contains a hatched, cartoon or Gooch shaded image. The output of the edge detection shader is either black or white, so we use a multiplicative blend (src*dst) with the image already in the frame buffer:

```
d3d->SetRenderState (D3DRS_ALPHABLENDENABLE,TRUE);
d3d->SetRenderState (D3DRS_SRCBLEND,  D3DBLEND_DESTCOLOR);
d3d->SetRenderState (D3DRS_DESTBLEND, D3DBLEND_ZERO);
```

This frame buffer operation is nice because we can multipass edge filters with the frame buffer and get the aggregate edges. In the NPR sample on the book CD, for example, we do one pass for normal discontinuities and one for depth discontinuities. It is worth noting that it would be possible to process both normal discontinuities and depth discontinuities using 1.4 pixel shaders and co-issue pixel shader instructions, but we chose to use a larger filter kernel (and thus more instructions) in the sample shown here.

Depth Precision

We have found that 8 bits of precision for eye-space depth works well for the simple scenes we have tested, but we expect this to be a problem for more aggressive NPR applications such as games with large environments. In scenes of large game environments, using only 8 bits of precision to represent eye-space depth will cause some which are close to each other to “fuse” together if their world-space normals are also similar. Because of this, it might be necessary to use techniques which spread the eye-space depth across multiple channels or to simply rely upon future generations of hardware which will provide higher precision pixels and texels.

Alpha Test for Efficiency

Since the black edge pixels are a very small subset of the total pixels in the scene, we can alpha test the edge image to save frame buffer bandwidth during the composite. Note that the last instruction in the depth and normal discontinuity shaders moves the red channel of the filter result into the alpha channel of the pixel. This is done so that the alpha test functionality which follows the pixel shader can be used to kill the pixel rather than composite it with the frame buffer, speeding up performance. Since we want to kill pixels which are white, we set an alpha reference value of something between white and black (0.5f) and use an alpha compare function of `D3DCMP_GREATER`:

```
d3d->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
d3d->SetRenderState(D3DRS_ALPHAREF, (DWORD) 0.5f);
d3d->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATER);
```

Shadow Outlines

In addition to outlining object silhouettes, it is also desirable to outline shadows in a scene. We have added this functionality to a stencil shadow application in the RADEON 8500 SDK on the ATI website as shown in Figure 10 below.

Figure 10 (a) shows a scene using the hatching technique from [Praun01] alone. In addition to this, we use stencil shadows to generate an image in a texture which contains normals and depths similar to the preceding technique. The application renders the scene into the texture with world space normals and depths, using the stencil buffer to write to only pixels which are not in shadow. The application then re-renders the geometry to the pixels in shadow, but negates the world space normals. This results in an image like that shown in Figure 10 (b), where the alpha channel (not shown) contains depths. The same normal and depth discontinuity filters used above are applied to this image to determine both object and shadow edges in one pass. These edges are composited over a hatched scene which already contains areas in shadow which have been hatched with the densest hatching pattern to simulate shadow. Figure 10 (d) shows this technique along with coloration of the TAMs and per-pixel TAM weight determination.

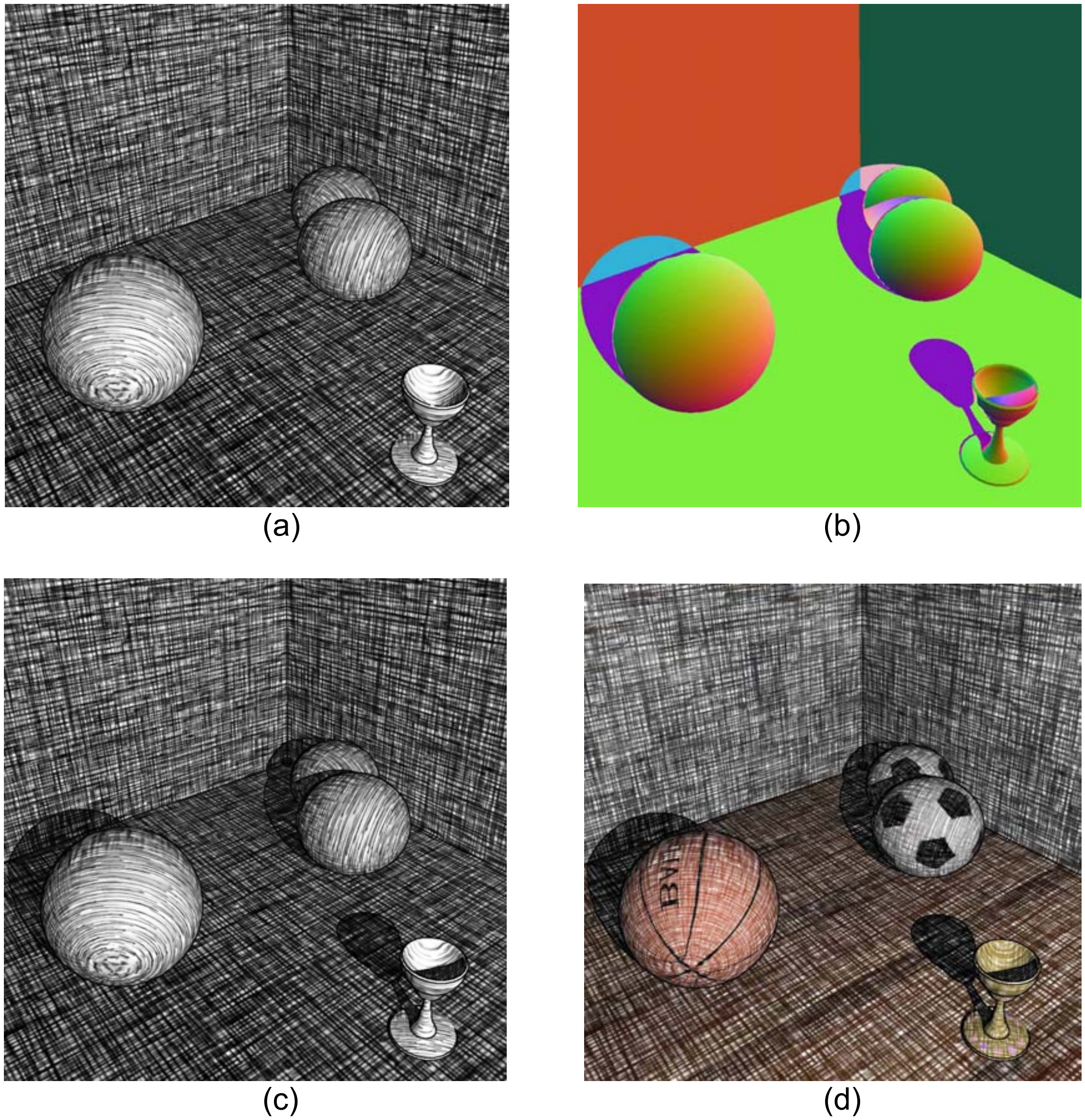


Figure 10 – (a) A plain outlined hatched scene. (b) The renderable texture containing world-space normals for non-shadowed pixels and negated normals for shadowed pixels. (c) A hatched scene with shadows, object outlines and shadow outlines. (d) Adding base texture coloring and per-pixel TAM weight calculation.

Thickening Outlines with Morphology

The outlines generated by the above technique are a few pixels thick and look fine for many NPR applications, but some applications may want thicker lines. Rather than directly composite the edges onto a shaded image in the back buffer, we can render the edges into a separate texture and apply morphological operations as shown in "Image Processing with 1.4 Pixel Shaders in Direct3D." To thicken the lines, use dilation; to thin them or break them up, use erosion. Do give a different style, we could thicken the lines by ping-pong rendering between renderable textures. After performing the desired number of dilations, composite the thickened edge image back onto the shaded frame buffer as discussed above.

Summary of Image Space Technique.

Rendering a scene with the image-space outlining technique shown here is done in the following steps:

1. Render shaded scene to back buffer
2. Render world-space normals and depths to a texture map
3. If thickening lines using morphology,
    a. Clear renderable texture to white
    b. Draw quad into texture using world-space normal discontinuity filter. Use alpha test and src*dst blending
    c. Draw quad into texture using depth discontinuity filter. Use alpha test and src*dst blending
    d. Dilate edges
    e. Composite with shaded scene in back buffer by drawing full screen quad using alpha test and src*dst blending
4. Else using edges directly
    a. Draw full-screen quad over whole screen using world-space normal discontinuity filter. Use alpha test and src*dst blending
    b. Draw full-screen quad over whole screen using depth discontinuity filter. Use alpha test and src*dst blending

Conclusion

We have presented Direct3D shader implementations of some recent developments in non-photorealistic rendering including outlines, cartoon shading, hatching, Gooch lighting and image-space techniques. For an excellent overview of these and many other NPR techniques, please refer to the recent book *Non-Photorealistic Rendering* by Gooch & Gooch. Gooch & Gooch also have an excellent online reference for NPR research: http://www.cs.utah.edu/npr/

Acknowledgments

Thanks to Eli Turner for 3D modeling and Chris Brennan for implementing the shadow outlining technique.

References

[Birdwell01] Ken Birdwell, Valve Software, Personal Communication, 2001

[Decaudin96] Philippe Decaudin, "Cartoon-looking rendering of 3D-scenes," Technical Report INRIA 2919, Universite de Technologie de Compiegne, France, June 1996.

[Gooch98] Amy Gooch, Bruce Gooch, Peter Shirley and Elaine Cohen, "A non-photorealistic lighting model for automatic technical illustration." Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, edited by Michael Cohen, pp. 447-452, Reading, MA: Addison-Wesley, 1998.

[Praun01] Emil Praun, Hugues Hoppe, Matthew Webb, Adam Finkelstein, "Real-time Hatching," Computer Graphics (SIGGRAPH 2001 Proceedings), pages 581-586.

[Lake00] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. "Stylized Rendering Techniques For Scalable Real-Time 3D Animations," Non-Photorealistic Animation and Rendering, 2000.

[Lengyel01] Jed Lengyel, Emil Praun, Adam Finkelstein and Hugues Hoppe, "Real-time fur over arbitrary surfaces," *ACM Symposium on Interactive 3D Graphics 2001*, pp 227-232.

[Saito90] Takafumi Saito and Tokiichiro Takahashi, "Comprehensible Rendering of 3-D Shapes." Computer Graphics (SIGGRAPH '90 Proceedings), pages 197-206, 1990.

[Vlachos01] Alex Vlachos, Jörg Peters, Chas Boyd and Jason L. Mitchell, "Curved PN Triangles," ACM Symposium on Interactive 3D Graphics, 2001.