2

III

NPR Effects Using the Geometry Shader

Pedro Hermosilla and Pere-Pau Vázquez

2.1 Introduction

Non-photorrealistic rendering (NPR) techniques [Achorn et al. 03, Gooch and Gooch 01] have been here for quite a while [Saito and Takahashi 90]. In contrast to traditional rendering, these techniques deal with geometric entities such as silhouettes, which makes them not easily amenable to GPU algorithms, although some papers already address some NPR algorithms in hardware [Dietrich 00, Mitchell et al. 02, Everitt 02, Card and Mitchell 02]. With the arrival of more modern graphics hardware that includes the geometry shader stage, some of these techniques can be implemented in hardware, making them real time in many cases [McGuire and Hughes 04, Dyken et al. 08, Doss 08]. In this chapter we present a set of techniques that can be implemented using the GPU by taking advantage of the geometry shader pipeline stage. Concretely, we show how to make use of the geometry shader in order to render objects and their silhouettes in a single pass, and to imitate pencil drawing.

2.2 Previous Work

Silhouette rendering has been studied extensively. Two major groups of algorithms require the extraction of silhouettes in real time: shadow volume-based approaches and non-photorealistic rendering [Gooch and Gooch 01].

From the literature, we may extract two different approaches: object-space and image-space algorithms. However, most modern algorithms work in either image space or hybrid space. For the purposes of this chapter, we are interested in GPU-based algorithms, and these are the ones we will present. We refer the interested reader to the works of [Isenberg et al. 03] and [Hartner et al. 03] for overviews and deep comparisons on CPU-based silhouette extraction algorithms.

GPU-assisted algorithms may compute the silhouette either using multiple rendering passes [Mitchell et al. 02] or in a single pass. Single pass methods usually use some sort of precomputation in order to store adjacency information into the vertices [Card and Mitchell 02], or make use of the geometry shader feature [Doss 08], as this may query adjacency information. These algorithms generate the silhouette in a single rendering pass, though still a first geometry pass is required for the object itself.

One of the first attempts to extract silhouettes using hardware is due to [Raskar 01], where a new stage at the rendering pipeline is introduced: the primitive shader. At this stage, polygons are treated as single primitives, similar to the way actual geometric shaders do. Raskar's proposal also requires modification of the incoming geometry. For instance, extending back faces to render silhouettes, and adding polygons in order to render ridges and valleys.

[Card and Mitchell 02] pack adjacent normals into the texture coordinates of vertices and render edges as degenerated quads, which expand if they are detected to belong to a silhouette edge in the vertex processor. This is a single pass algorithm that requires rendering extra geometry for the silhouette extraction. This approach is also used by [Achorn et al. 03]. [McGuire and Hughes 04] extend this technique to store the four vertices of the two faces adjacent to each edge, instead of explicit face normals. This allows the authors to construct correct face normals under animation and add textures to generate artistic strokes.

In [Ashikhmin 04], silhouettes are generated without managing adjacency information through a multiple rendering passes algorithm that reads back the frame buffer in order to determine face visibility. More recently, [Dyken et al. 08] extract silhouettes from a triangle mesh and perform an adaptive tessellation in order to visualize the silhouette with smooth curvature. However, this system neither textures the silhouettes nor extrudes the silhouette geometry. [Doss 08] develops an algorithm similar to the one presented here: he extrudes the silhouettes, but with no guarantee of continuity between the extrusions generated from different edges; consequently, gaps are easily noticeable as the silhouette width grows.

A completely different approach is used by [Gooch et al. 99], as they note that environment maps can be used to darken the contour edges of a model but, as a result, the rendered lines have uncontrolled variable thickness. The same idea was refined by [Dietrich 00], who took advantage of the GPU hardware available at that moment (GeForce 256). [Everitt 02] used MIP-maps to achieve similar effects. In all of these cases, it is difficult to fine-tune an artistic style because there is no support geometry underlying the silhouette.

The approach presented here is conceptually similar to [Raskar 01], but takes advantage of modern hardware. We also borrow ideas from [Doss 08] and [McGuire and Hughes 04] for silhouette geometry generation. In contrast to these approaches, we generate both the silhouette and the object in a single pass, and we present an algorithm for correct texturing with coherent and continuous texture coordinates along the entire silhouette.

2.3 Silhouette Rendering

Silhouette rendering is a fundamental element in most NPR effects, as it plays an important role in object shape understanding. In this section we present a novel approach for the detection, generation, and texturing of a model in a single rendering pass. First we will present an overview of our algorithm, and then we will detail how each of the steps is implemented.

2.3.1 Algorithm Overview

In order to carry out the entire process in a single step, we will take advantage of some of the modern features of GPUs; concretely, we will make an extensive use of the geometry shader. This stage permits triangle operations, with knowledge of adjacent triangles, and the generation of new triangles to the geometry.

Our process for silhouette rendering performs the following steps at the different stages of the pipeline (Figure 2.1):

- *Vertex shader.* Vertices are transformed in the usual way to camera space.
- *Geometry shader.* In this stage, edges that belong to the silhouette are detected by using the information of the current triangle and its adjacency, and the corresponding geometry is generated.

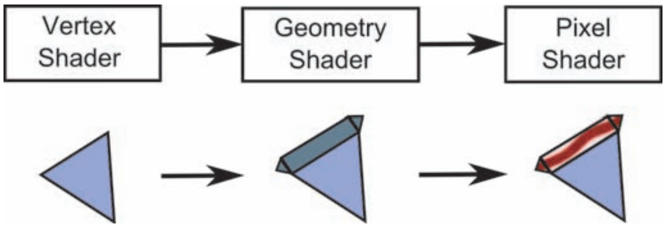


Figure 2.1. Pipeline overview: the vertex shader (left) transforms the vertex coordinates of the incoming geometry; the second step (geometry shader) generates new geometry for the silhouette of the object. Finally, the fragment shader generates correct texture coordinates.

- *Pixel shader.* For each rasterized fragment, its texture coordinates are generated and pixels are shaded according to the color obtained from the texture.

Before we may send a mesh throughout the pipeline, we first perform a special reordering of the indices of the triangles. This will make the adjacency information available at the geometry shader level. In order to access such information, we send six indices per triangle (instead of the normal three), ordered as depicted in Figure 2.2. The central triangle, identified by indices 0, 4, and 2 is the one to be analyzed. The remaining adjacent triangles are needed to show if any of the edges of the central triangle belong to the silhouette.

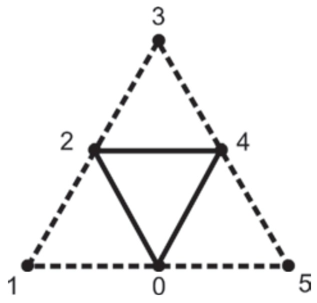


Figure 2.2. Index sort.

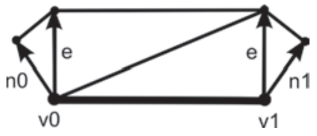


Figure 2.3. Edge geometry.

2.3.2 Silhouette Detection and Geometry Generation

We consider a closed triangle mesh with consistently oriented triangles. The set of triangles is denoted, $T_1 \dots T_N$. The set of vertices is $v_1 \dots v_n$ in $\Re^3$, and normals are given by triangles: n_t is the normal of a triangle $T_t = [v_i, v_j, v_k]$, using the notation by [Dyken et al. 08]. This triangle normal is defined as the normalization of the vector $(v_j - v_i) \times (v_k - v_i)$. Given an observer at position $x \in \Re^3$, we may say a triangle is *front facing* in v if $(v - x) \cdot n \leq 0$, otherwise it is *back facing*. The silhouette of a triangle mesh is the set of edges where one of the adjacent triangles is front facing while the other is back facing. In order to detect a silhouette in a triangulated mesh we need to process any triangle, together with the triangles that share an edge with it. This test is performed at the geometry shader level for each edge of the triangle being processed. In order to avoid duplicate silhouettes when processing both the front facing and the back facing triangles, we only generate silhouettes for the front-facing triangles. The code in Listing 2.1 shows how to detect a silhouette edge at the geometry shader level.

As shown in Figure 2.3, once an edge (indicated by $\overline{v0v1}$) has been determined as a silhouette one, we generate the geometry that will act as the silhouette by

```
[maxvertexcount(21)]
void main( triangleadj VERTEXin input[6],
         inout TriangleStream<VERTEXout> TriStream )
{

 //Calculate the triangle normal and view direction.
 float3 normalTrian = getNormal( input[0].Pos.xyz,
   input[2].Pos.xyz, input[4].Pos.xyz );
 float3 viewDirect = normalize(-input[0].Pos.xyz
   - input[2].Pos.xyz - input[4].Pos.xyz);

 //If the triangle is frontfacing
 [branch]if(dot(normalTrian,viewDirect) > 0.0f)
 {

  [loop]for(uint i = 0; i < 6; i+=2)
  {

   //Calculate the normal for this triangle.
   float auxIndex = (i+2)%6;
   float3 auxNormal = getNormal( input[i].Pos.xyz,
     input[i+1].Pos.xyz, input[auxIndex].Pos.xyz );
   float3 auxDirect = normalize(- input[i].Pos.xyz
     - input[i+1].Pos.xyz - input[auxIndex].Pos.xyz);

   //If the triangle is backfacing
   [branch]if(dot(auxNormal,auxDirect) <= 0.0f)
   {

    //Here we have a silhouette edge.

   }
  }
 }
}
```

Listing 2.1. Geometry shader silhouette detection code.

applying the algorithm in [McGuire and Hughes 04]. It consists of four triangles. The central triangles forming the quad are generated by extruding the edges' vertices using as the extrusion direction of a vector orthogonal to the edge and view directions. The remaining triangles are generated by extruding the vertices from the edge in the direction of the vertex normal as projected on screen. The generation of such geometry can be done either in world space or in screen space. We usually use screen space because this way is easier to obtain a silhouette geometry of constant size in screen. The code needed to generate this geometry appears in Listing 2.2.

```
//Transform the positions to screen space.
float4 transPos1 = mul(input[i].Pos,projMatrix);
transPos1 = transPos1/transPos1.w;
float4 transPos2 = mul(input[auxIndex].Pos,projMatrix);
transPos2 = transPos2/transPos2.w;

//Calculate the edge direction in screen space.
float2 edgeDirection = normalize(transPos2.xy - transPos1.xy);

//Calculate the extrude vector in screen space.
float4 extrudeDirection = float4(normalize(
  float2(-edgeDirection.y,edgeDirection.x)),0.0f,0.0f);

//Calculate the extrude vector along the vertex
//normal in screen space.
float4 normExtrude1 = mul(input[i].Pos + input[i].Normal
  ,projMatrix);
normExtrude1 = normExtrude1 / normExtrude1.w;
normExtrude1 = normExtrude1 - transPos1;
normExtrude1 = float4(normalize(normExtrude1.xy),0.0f,0.0f);
float4 normExtrude2 = mul(input[auxIndex].Pos
  + input[auxIndex].Normal,projMatrix);
normExtrude2 = normExtrude2 / normExtrude2.w;
normExtrude2 = normExtrude2 - transPos2;
normExtrude2 = float4(normalize(normExtrude2.xy),0.0f,0.0f);

//Scale the extrude directions with the edge size.
normExtrude1 = normExtrude1 * edgeSize;
normExtrude2 = normExtrude2 * edgeSize;
extrudeDirection = extrudeDirection * edgeSize;

//Calculate the extruded vertices.
float4 normVertex1 = transPos1 + normExtrude1;
float4 extruVertex1 = transPos1 + extrudeDirection;
float4 normVertex2 = transPos2 + normExtrude2;
float4 extruVertex2 = transPos2 + extrudeDirection;

//Create the output polygons.
VERTEXout outVert;

outVert.Pos = float4(normVertex1.xyz,1.0f);
TriStream.Append(outVert);
outVert.Pos = float4(extruVertex1.xyz,1.0f);
TriStream.Append(outVert);
outVert.Pos = float4(transPos1.xyz,1.0f);
TriStream.Append(outVert);
outVert.Pos = float4(extruVertex2.xyz,1.0f);
TriStream.Append(outVert);
outVert.Pos = float4(transPos2.xyz,1.0f);
TriStream.Append(outVert);
```

```
outVert.Pos = float4(normVertex2.xyz,1.0f);
TriStream.Append(outVert);

TriStream.RestartStrip();
```

Listing 2.2. Geometry shader silhouette generation.

In some cases, this solution may produce an error when the extrusion direction has a different direction than the projected normal version. There are several ways to solve this. One of the simplest ones consists of changing the direction of the projected normal, as commented in [Hermosilla and Vázquez 09]. Some cases also might require different silhouette geometry (see [McGuire and Hughes 04] for more details).

2.3.3 Silhouette Texturing

Once the silhouette geometry has been generated, it becomes obvious that texturing this geometry will increase the number of effects that can be achieved. In order to properly texture the silhouette geometry, we need to generate texture coordinates. Texture coordinates generation is a bit tricky, as we need to generate continuous coordinates along the entire silhouette. Therefore we may not simply assign coordinates from 0 to 1 for each edge, as this would cause irregular coordinate distribution if the edges are not created all with the same length. Instead we need a global strategy for coordinate generation because each triangle of the silhouette will not be aware of the neighbor triangles' coordinates.

From the two texture coordinates u and v, coordinate v can be simply defined, because it changes from zero to one as long as we go away from the object, as depicted in Figure 2.4.

Coordinate u has to be generated in such a way that its value is continuous along the silhouette of the object. In order to make sure that two adjacent edges will generate coherent texture coordinates, we will build a function that depends on the position of the projection of the vertices on screen. As a consequence, the coordinates will be continuous because neighbor edges share a vertex. This is achieved when the geometry shader sends the x- and y-coordinates of the

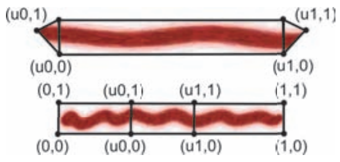


Figure 2.4. The v-coordinate has a value of 0 for the edge vertex and 1 for the extruded vertices.

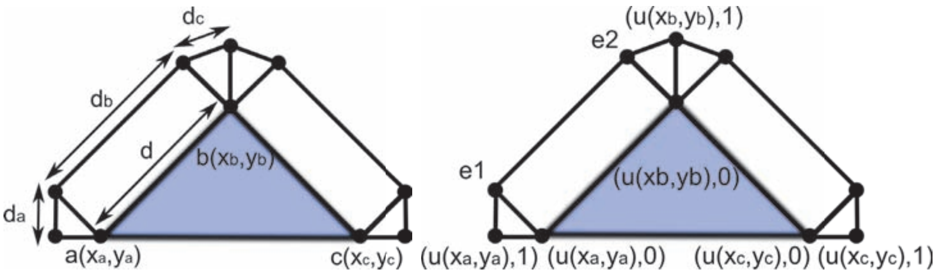


Figure 2.5. The u-coordinates are generated from the edge vertex coordinates in screen space. The first vertex of the edge and the vertex extruded from the first vertex normal gets the u-coordinate from the coordinates of the first vertex (a) The other edge endpoint, and the vertex extruded from the second vertex normal gets the u-coordinate from the coordinates of the second vertex (b) The vertices extruded from the extrusion vector ($e1$ & $e2$) obtain their u-coordinates by interpolation, as show in Equation (2.1).

generated vertices in screen, together with v-coordinate. The pixel shader will receive such coordinates as interpolated values, and will generate the corresponding u value. Figure 2.5 shows how this information is used.

Vertices e receive their coordinates from linear interpolation as shown in the following equations:

$$\begin{gathered}
e1.ux = x_a + (|\vec{ab}| * ((d * d_a)/(d_a + d_b + d_c))) \\
e1.uy = y_a + (|\vec{ab}| * ((d * d_a)/(d_a + d_b + dc))) \\
e1.v = 0 \\
\\
e2.ux = x_b + (|\vec{ba}| * ((d * d_a)/(d_a + d_b + d_c))) \\
e2.uy = x_b + (|\vec{ba}| * ((d * d_a)/(d_a + d_b + d_c))) \\
e2.v = 0.
\end{gathered} \tag{2.1}$$

The pixel shader will receive those coordinates interpolated and will use them to compute the final texture coordinates.

In order to compute the final u component, we will transform components x and y into polar coordinates. The reference system will be the screen space position of the center of the bounding box of the object, and the x- and y- axes will be those of the viewport. Therefore, we will have polar coordinates computed as

- Polar angle: α will be the angle between the x-axis, and the vector with initial point at the origin of the coordinates system, and final point at (x, y).

- Distance: d is the distance from (x, y) to the origin of the coordinates system.

```
float4 main(PIXELin inPut):SV_Target
{
 //Initial texture coordinate.
 float2 coord = float2(0.0f,inPut.UV.z);

 //Vector from the projected center bounding box to
 //the location.
 float2 vect = inPut.UV.xy - aabbPos;

 //Calculate the polar coordinate.
 float angle = atan(vect.y/vect.x);
 angle = (vect.x < 0.0f)?angle+PI:
   (vect.y < 0.0f)?angle+(2*PI):angle;

 //Assign the angle plus distance to the u texture coordinate.
 coord.x = ((angle/(2*PI)) + (length(vect)*lengthPer))*scale;

 //Get the texture color.
 float4 col = texureDiff.Sample(samLinear,coord);

 //Alpha test.
 if(col.a < 0.1f)
  discard;

 //Return color.
 return col;
}
```

Listing 2.3. Silhouette Texturing.

Finally, we compute u as indicated in the following equation:

$$u = (\frac{\alpha}{2 * \pi}) + (k * d).$$

As we may see, the polar angle is divided by 2π in order to transform it into a value in the [0..1] range. The distance is weighted by a factor k that may be changed interactively. For objects of a sphere-like shape, k value can be set to close to 0, but for objects with edges that roughly point to the origin of coordinates, the value k must be different from 0. Otherwise, texture coordinates at both ends of those edges would be similar. The code that implements this is shown in Listing 2.3.

This algorithm may produce small artifacts in edges that are projected on the screen close to the center of the bounding box. However, these are not visible in most of the models we tested.

2.3.4 Single Pass Geometry and Silhouette Rendering

Most silhouette rendering algorithms perform two passes, one for the geometry, and another one for the silhouettes. This means that the geometry is sent two times into the rendering pipeline. We can avoid this by taking further advantage of the geometry shader with little modifications to the original code. This is achieved by simply rendering the triangle being analyzed by the geometry shader, even if it does not have any edge belonging to the silhouette. This can be done thanks to the fact that the geometry shader may output more than a single triangle.

So far, the pixel shader code deals only with edges and textures them accordingly. In order to render the triangles belonging to the geometry in a single pass, we must inform the pixel shader of the sort of triangle that originated the rasterized fragment: silhouette or geometry. We encode this information in the texture coordinates. As we are passing three coordinates, we will use one of them—in this case the v-coordinate—to encode this information. For triangles belonging to this geometry, we assign the value 2. This way, the pixel shader can easily distinguish between both kinds of triangles, and shade them accordingly.

2.3.5 Results

We can see some results of our algorithm in Figure 2.6. The algorithm presented here achieves real-time performance, as can be seen in Table 2.1. These results were obtained on a 6 GB Quad Core PC equipped with a GeForce 9800 GX2 GPU. The viewport resolution (key for image space algorithms) was 1680×1050. Note that even complex objects (such as the Buddha model), with more than 1M polygons, achieve interactive framerates.

Models	**Triangles**	**FPS**
Buddha	1087716	8.50
Armadillo	345944	21.07
Asian Dragon	225588	25.75
Dragon	100000	60.07
Bunny	69666	110.78
Atenea	15014	337.59

Table 2.1. Framerates obtained with the textured silhouette algorithm on a GeForce 9800 GX2 GPU with a viewport resolution of 1680×1050.

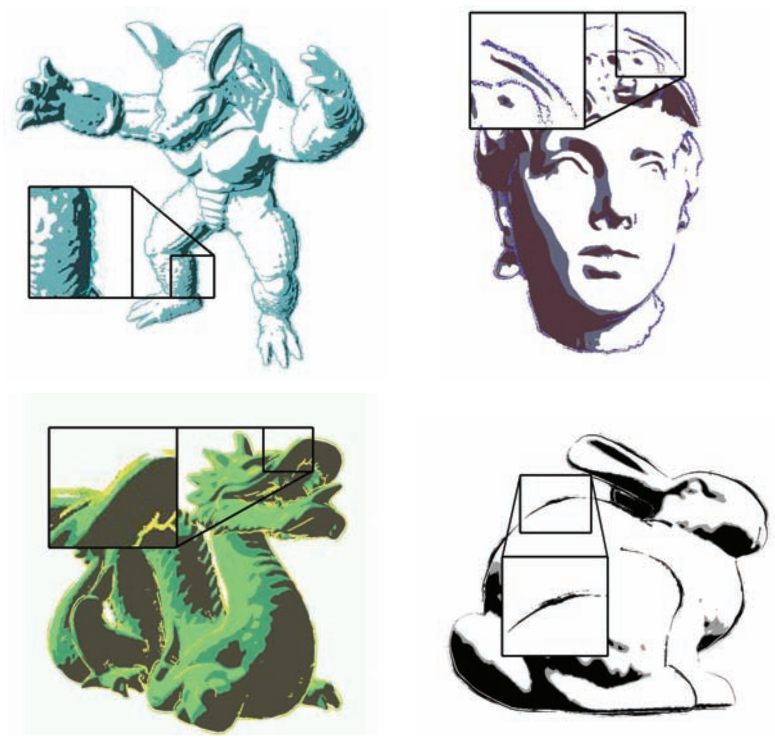


Figure 2.6. Images showing the algorithm in action. Silhouettes have been generated and textured in real time.

2.4 Pencil Rendering

In this section we will present how to implement pencil rendering in the geometry shader. This is based on the technique presented by [Lee et al. 06].

2.4.1 Algorithm Overview

The original technique by [Lee et al. 06] works in the following way. First, the minimal curvature at each vertex is computed. Then, triangles are sent through the pipeline with this value as a texture coordinate for each vertex. In order to shade the interior of a triangle, the curvatures at the vertices are used to rotate a pencil texture in screen space. This texture is rotated three times in screen space, one for each curvature, and the result is combined with blending. Several textures with different tones are used at the same time, stored in an array of textures. The correct one is selected according to illumination.

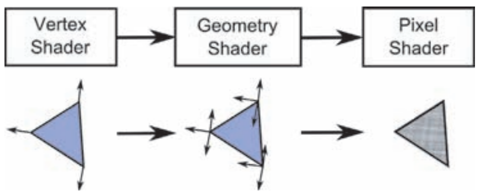


Figure 2.7. Pipeline overview: the vertex shader transforms the vertices to screen space; the geometry shader assigns the vertex curvatures of the triangle to the three vertices. Finally, the pixel shader generates texture coordinates for the three curvatures and calculates the final color.

We may implement this algorithm using the GPU pipeline in the following way (Figure 2.7):

- *Vertex shader.* Vertices are transformed to screen coordinates. Vertex curvature is transformed too, and only x- and y-components are passed through as a two-dimensional vector.
- *Geometry shader.* The curvature values are assigned to each vertex as texture coordinates.
- *Pixel shader.* Final color is computed.

2.4.2 Geometry Shader Optimization

This technique has an important shortcoming: It is necessary to make a copy of each vertex for each triangle that shares it. This is because each pixel receives the interpolated curvature by using each vertex, and the three curvatures are required unchanged. Each duplicated vertex is assigned the three curvatures of the vertices of the triangle in order to make each fragment get the three exact curvatures.

In order to avoid vertex duplication, we will use the geometry shader. At the geometry shader level, we receive the three vertices of a triangle, with its corresponding curvatures. These curvatures are assigned as three texture coordinates to the vertices of the output triangle in the same order. Thus, the fragment shader will receive the three values without interpolation. The code corresponding to the geometry shader appears in Listing 2.4.

2.4.3 Pixel Texturing

The final color composition is performed in the following way: the fragment shader receives the coordinates of the fragment, together with the curvatures. We will use components x and y of the fragment in screen space as texture coordinates. These coordinates are scaled to the range [0..1].

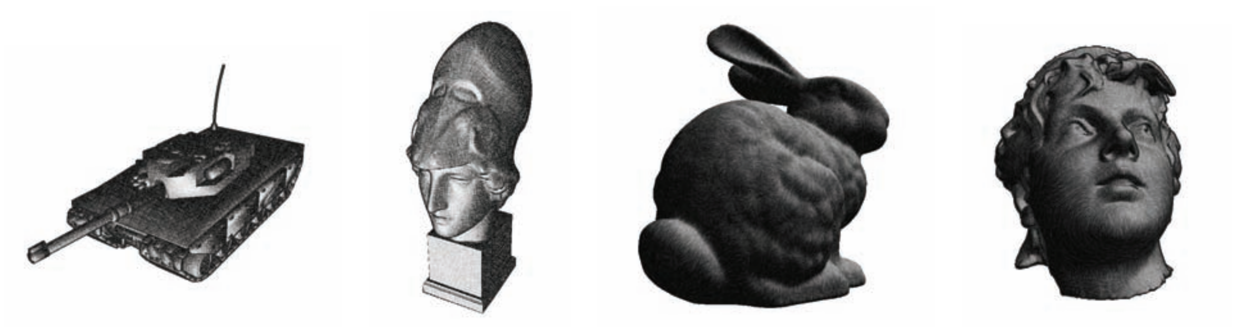
```
[maxvertexcount(3)]
void main( triangle VERTEXin input[3],
   inout TriangleStream<VERTEXout> TriStream )
{
 //Assign triangle curvatures to the three vertices.
 VERTEXout outVert;
 outVert.Pos = input[0].Pos;
 outVert.norm = input[0].norm;
 outVert.curv1 = input[0].curv;
 outVert.curv2 = input[1].curv;
 outVert.curv3 = input[2].curv;
 TriStream.Append(outVert);

 outVert.Pos = input[1].Pos;
 outVert.norm = input[1].norm;
 outVert.curv1 = input[0].curv;
 outVert.curv2 = input[1].curv;
 outVert.curv3 = input[2].curv;
 TriStream.Append(outVert);

 outVert.Pos = input[2].Pos;
 outVert.norm = input[2].norm;
 outVert.curv1 = input[0].curv;
 outVert.curv2 = input[1].curv;
 outVert.curv3 = input[2].curv;
 TriStream.Append(outVert);

 TriStream.RestartStrip();
}
```

Listing 2.4. Pencil geometry shader.

In order to orient the texture according to the surface curvature, and to avoid deformations inside large triangles, the paper texture is oriented by using the three curvatures at the vertices, and blending them with equal weights. The implementation has three steps: First, the angles between curvatures and the x-axis are computed. Then, three two-dimensional rotation matrices are built using these angles. Finally, these matrices are used to transform the texture coordinates obtained from the fragment coordinates, and this yields three new texture coordinates. These are the ones used for final texturing.

The model may be shaded by using the dot product between the light direction and the surface normal in order to access a texture array of different tones. We use a single texture but modified with a function that changes brightness and contrast according to the incident illumination at each point. We use the following function:

```
p = 1.0 - {max}({dot}(light,normal),0.0)
colorDest = {pow}(colorSrc,p*S + O).
```



$O = 5; S = 0.15$ $O = 5; S = 0.15$ $O = 5; S = 1$ $O = 5; S = 1$

Figure 2.8. Pencil rendering results.

The resulting color of texture mapping is powered to a value in the range $[O..S + O]$. This value is determined from the dot product between the light and normal vectors. This will darken dark regions and lighten lighter regions, as can be seen in Figure 2.8 where we show the comparison using different values of O and S. The code corresponding to the geometry shader is shown in Listing 2.5.

```
float4 main(PIXELin inPut):SV_Target
{
 float2 xdir = float2(1.0f,0.0f);
 float2x2 rotMat;
 //Calculate the pixel coordinates.
 float2 uv = float2(inPut.Pos.x/width,inPut.Pos.y/height);

 //Calculate the rotated coordinates.
 float2 uvDir = normalize(inPut.curv1);
 float angle = atan(uvDir.y/uvDir.x);
 angle = (uvDir.x < 0.0f)?angle+PI:
  (uvDir.y < 0.0f)?angle+(2*PI):angle;
 float cosVal = cos(angle);
 float sinVal = sin(angle);
 rotMat[0][0] = cosVal;
 rotMat[1][0] = -sinVal;
 rotMat[0][1] = sinVal;
 rotMat[1][1] = cosVal;
 float2 uv1 = mul(uv,rotMat);

 uvDir = normalize(inPut.curv2);
 angle = atan(uvDir.y/uvDir.x);
 angle = (uvDir.x < 0.0f)?angle+PI:
  (uvDir.y < 0.0f)?angle+(2*PI):angle;
 cosVal = cos(angle);
 sinVal = sin(angle);
 rotMat[0][0] = cosVal;
 rotMat[1][0] = -sinVal;
 rotMat[0][1] = sinVal;
 rotMat[1][1] = cosVal;
 float2 uv2 = mul(uv,rotMat);

 uvDir = normalize(inPut.curv3);
 angle = atan(uvDir.y/uvDir.x);
 angle = (uvDir.x < 0.0f)?angle+PI:
  (uvDir.y < 0.0f)?angle+(2*PI):angle;
 cosVal = cos(angle);
 sinVal = sin(angle);
 rotMat[0][0] = cosVal;
 rotMat[1][0] = -sinVal;
 rotMat[0][1] = sinVal;
 rotMat[1][1] = cosVal;
 float2 uv3 = mul(uv,rotMat);

 //Calculate the light incident at this pixel.
 float percen = 1.0f - max(dot(normalize(inPut.norm),
  lightDir),0.0);

 //Combine the three colors.
 float4 color = (texPencil.Sample(samLinear,uv1)*0.333f)
  +(texPencil.Sample(samLinear,uv2)*0.333f)
  +(texPencil.Sample(samLinear,uv3)*0.333f);

 //Calculate the final color.
 percen = (percen*S) + O;
 color.xyz = pow(color.xyz,float3(percen,percen,percen));
 return color;
}
```

Listing 2.5. Pencil pixel shader.

Models	Triangles	FPS
Buddha	1087716	87.71
Armadillo	345944	117.22
Asian Dragon	225588	199.20
Dragon	100000	344.28
Bunny	69666	422.20
Atenea	15014	553.55

Table 2.2. Framerates obtained with our implementation of the pencil rendering algorithm on a GeForce 9800 GX2 GPU graphics card and a viewport resolution of 1680×1050.

2.4.4 Results

Table 2.2 shows the framerates obtained with the pencil rendering technique. Note that we obtain high framerates because the implementation is relatively cheap, and that from the numbers we can deduce that the timings depend strongly on vertex count rather than rasterized fragments count.

2.5 Acknowledgments

The authors want to thank the reviewers for their valuable comments. This work has been supported by project TIN2007-67982-C02-01 of the Spanish Government.

Bibliography

[Achorn et al. 03] Brett Achorn, Daniel Teece, M. Sheelagh T. Carpendale, Mario Costa Sousa, David Ebert, Bruce Gooch, Victoria Interrante, Lisa M. Streit, and Oleg Veryovka. "Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems." In *SIGGRAPH 2003*. ACM Press, 2003.

[Ashikhmin 04] Michael Ashikhmin. "Image-Space Silhouettes for Unprocessed Models." In *GI '04: Proceedings of Graphics Interface 2004*, pp. 195–202. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2004.

[Card and Mitchell 02] Drew Card and Jason L. Mitchell. "Non-Photorealistic Rendering with Pixel and Vertex Shaders." In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, edited by Wolfgang Engel. Plano, Texas: Wordware, 2002.

[Dietrich 00] Sim Dietrich. "Cartoon Rendering and Advanced Texture Features of the GeForce 256 Texture Matrix, Projective Textures, Cube Maps, Texture Coordinate Generation and Dotproduct3 Texture Blending." Technical report, NVIDIA, 2000.

[Doss 08] Joshua Doss. "Inking the Cube: Edge Detection with Direct3D 10." http://www.gamasutra.com/visualcomputing/archive, 2008.

[Dyken et al. 08] Christopher Dyken, Martin Reimers, and Johan Seland. "Real-Time GPU Silhouette Refinement Using Adaptively Blended Bézier Patches." *Computer Graphics Forum* 27:1 (2008), 1–12.

[Everitt 02] Cass Everitt. "One-Pass Silhouette Rendering with GeForce and GeForce2." White paper, NVIDIA Corporation, 2002.

[Gooch and Gooch 01] Amy A. Gooch and Bruce Gooch. *Non-Photorealistic Rendering.* A K Peters, 2001. ISBN: 1568811330, 250 pages.

[Gooch et al. 99] Bruce Gooch, Peter-Pike J. Sloan, Amy A. Gooch, Peter Shirley, and Richard Riesenfeld. "Interactive Technical Illustration." In *1999 ACM Symposium on Interactive 3D Graphics*, pp. 31–38, 1999.

[Hartner et al. 03] Ashley Hartner, Mark Hartner, Elaine Cohen, and Bruce Gooch. "Object Space Silhouette Algorithms.", 2003. Unpublished.

[Hermosilla and Vázquez 09] Pedro Hermosilla and Pere-Pau Vázquez. "Single Pass GPU Stylized Edges." In *Proceedings of IV Iberoamerican Symposium in Computer Graphics*, pp. 47–54, 2009.

[Isenberg et al. 03] Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. "A Developer's Guide to Silhouette Algorithms for Polygonal Models." *IEEE Comput. Graph. Appl.* 23:4 (2003), 28–37.

[Lee et al. 06] Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. "Real-Time Pencil Rendering." In *NPAR '06: Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, pp. 37–45. New York: ACM, 2006.

[McGuire and Hughes 04] Morgan McGuire and John F. Hughes. "Hardware-Determined Feature Edges." In *NPAR '04: Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pp. 35–47. New York: ACM, 2004.

[Mitchell et al. 02] Jason L. Mitchell, Chris Brennan, and Drew Card. "Real-Time Image-Space Outlining for Non-Photorealistic Rendering." In *Siggraph 02*, 2002.

[Raskar 01] Ramesh Raskar. "Hardware Support for Non-photorealistic Rendering." In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 41–46. ACM Press, 2001.

[Saito and Takahashi 90] Takafumi Saito and Tokiichiro Takahashi. "Comprehensible Rendering of 3-D Shapes." *SIGGRAPH90* 24:3 (1990), 197–206.