Single Pass GPU Stylized Edges

P. Hermosilla & P.P. Vázquez

MOVING Group, Universitat Politècnica de Catalunya

Abstract

Silhouette detection is a key issue in many non-photorealistic rendering algorithms. Silhouettes play an important role in shape recognition because they provide one of the main cues for figure-to-ground distinction. However, since silhouettes are view-dependent, they must be computed per frame. There has been a lot of research tailored to find an efficient way to do this in real time, although most algorithms require either a precomputation, and/or multiple rendering passes. In this paper we present an algorithm that is able to generate silhouettes in realtime by using the geometric shader stage. In contrast to other approaches, our silhouettes do not suffer from discontinuities. A second contribution is the definition of a continuous set of texture coordinates that allows us coherently texture the object, independently on the length and orientation of the edges. This texture coordinates are also robust to translation and does not produce sudden changes during rotation.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display Algorithms, I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

1. Introduction

Silhouette edges are fundamental for many non-photorealistic algorithms, because they play a critical role in visual interpretation of 3D data. Accurate silhouette determination is not trivial. It can be done in different ways. There are a number of algorithms for determining and rendering silhouettes of 3D objects. In [IFH+03], several algorithms are analyzed and classified according to the space they work on: object-based, image-based, or hybrid algorithms. Most of those algorithms require either a pre-process of the geometry or multiple rendering passes. With the new GPUs geometry shader stage, a new possibility is open, since primitives may be analyzed in a single step, because information of the adjacency is available.

In this paper we take advantage of Geometry Shader stage in order to generate silhouette geometry on the fly. Silhouettes are extruded perpendicular to the viewing direction and textured. In contrast to previous approaches, we guarantee continuity in the generated geometry and we ensure continuity in texture coordinates. As a result we may synthesize images which really look-like hand-drawn versions of our virtual models.

The rest of the paper is organized as follows: Section 2 surveys related work. Section 3 presents our silhouette generation algorithm (see Figure 1). Section 4 details our texture coordinate generation technique. We discuss the results and limitations in Section 5, which also points to some lines for future research.

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 [GG01].

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 concerns of our paper, we are more interested to on where the computation is performed: CPU or GPU. In the first case, silhouettes or silhouette information is extracted in CPU before rendering. In the second case, GPU is used to build the necessary data structures for silhouette rendering, sometimes using multiple render passes.

Multiple-pass methods build the information required to render the silhouette in several passes, and often this information is computed every frame from scratch. Single pass
methods usually use some sort of precomputation in order to store adjacency information into the vertices, or make use (only recently) of the geometry shader feature, as this may query adjacency information. Our method belongs to this last kind of algorithms.

2.1. CPU-based algorithms

Saito and Takahashi presented the G-Buffer [ST90]. In this paper, the authors describe how to extract additional data during the rendering process, store it in G-Buffers, and use it for computing NPR primitives (silhouettes and feature lines). These primitives are then composited into the image to extend the comprehensibility of the shown objects. All this process is carried out in image space.

The work by Isenberg et al. also works in the CPU. They developed a data structure dubbed G-Strokes [IB06], in contrast to strokes, that encode a path that has is modified by a linestyle, G-Strokes are a unique sequence of indices each representing a pointer to a list of 3-D coordinates. Isenberg et al. [IFH∗03] provide an extensive overview on various CPU-based silhouette extraction techniques and their trade-offs. Hartner et al. [HHCG03] benchmark and compare various algorithms in terms of running time and code complexity.

2.2. GPU-based algorithms

Raskar [Ras01] introduces a new stage at the rendering pipeline: primitive shader. At this stage, polygons are treated as single primitives, in a similar way actual geometric shaders do. His proposal was to modify the incoming geometry, for instance extending back faces to render silhouettes, and adding polygons in order to render ridges and valleys. There was no support for texturing, and added lines where very thin. Our approach is very similar to this one, but we perform all the work on the GPU, in geometry shaders. moreover, we may decide the width of the geometry we generate, and we are able to texture it.

Mitchell [MBC02] et al. implemented a hardware-accelerated version of the G-Buffer algorithm in GPUs. As with other approaches, this requires multiple render passes of different geometry elements into textures in order to grab the important information to be able to build the final image in a pixel shader.

Card and Mitchell [CM02] pack adjacent normals into the texture coordinates of vertices and render edges as degenerated quads that are expanded if they are detected to belong to a silhouette edge in the vertex processor. This is a single pass algorithm although it requires rendering extra geometry for the silhouette extraction. This approach is also used by Gooch [ATC∗03]. McGuire and Hughes [MH04] 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.

Ashkimin [Ash04] generates silhouettes without managing adjacency information through a multiple rendering algorithm that reads back the frame buffer in order to determine face visibility.

Dyken et al. [DRS08] extract silhouettes from a triangle mesh and perform an adaptive tessellation in order to visualize the silhouette with smooth curvature. However, this system does not texture the silhouettes and does not extrude the silhouette geometry.

In a recent work, Doss [Dos08] develops an algorithm similar to ours, he extrudes the silhouettes, but no continuity is guaranteed between the extrusions generated from different edges, and, consequently, gaps are easily noticeable as the silhouette width grows.

A completely different approach is due to Gooch et al. [GSG∗99], where 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 [Die00], taking advantages of the current GPU hardware (GeForce 256). Everitt [Eve02] used the MIP-maps to achieve similar effects. In all of these cases, it is difficult to fine tune an artis-

Figure 1: Examples of models rendered with our algorithm for GPU-based stylized edges.
tic style because there is no support geometry underlying the silhouette. Our approach generates new geometry for the required features and we are able to texture it according to an artist defined texture map.

3. Geometry Shader-based silhouettes

3.1. Geometry Shader

Until the appearance of the Shader Model 4.0 it was only possible to process information at vertex and pixel levels, this means that there was no way of treating a primitive at once in the GPU. With the Shader Model 4.0 a new pipeline stage was appeared, the Geometry Shader. This new stage give us the possibility to process a mesh at primitive level with adjacent information (neighbor triangles) and generate a new geometry in the output (see Fig. 2). This is the information that we need to detect and create the silhouette edges.

We consider a closed triangle mesh with consistently oriented triangles. The set of triangles is denoted, $T_1 \ldots T_N$. The set of vertices is $v_1 \ldots v_n$ in $\mathbb{R}^3$, 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 [DRS08]. 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 $x \in \mathbb{R}^3$, 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. In order to be able to access the adjacency information, Geometry Shader require the triangle indices be specially sorted, as it is shown in Fig. 2.

![Figure 2: The triangle currently processed is the one determined by vertices (0,2,4). The ones determined by vertices (0,1,2), (2,3,4) and (4,5,0) are the triangles that share an edge with the current triangle.](image)

3.2. Silhouette Detection and Geometry Creation

As already said, we may detect if a triangle edge belongs to the silhouette respect to the observer by checking if, for both triangles sharing the edge, one is front facing and the other one is back facing. If we apply this method as is, it will generate the same edge two times, one for each triangle that shares the edge. In order to correct this, we only generate the edge if the triangle that is front-facing is the center one, the one that is actually treated by the Geometry Shader. This approach is also taken by Doss [Dos08].

Once a silhouette edge has been detected, we generate new geometry for the silhouette, and this is further processed at the fragment shader stage. This carried out by creating a new polygon, by extruding the edge along the vector perpendicular to the edge direction in screen space. However, this approach produces discontinuities in the start and end of the edges. These become especially visible with the increase in width of the silhouette, as shown in Fig. 3.

![Figure 3: Discontinuity produced when extrude the edge.](image)

In order to correct this, we adopt the approach by McGuire and Hughes [MH04], but we do this at the Geometry Shader stage in a single step (they require three rendering passes). We create two new triangles at the start and ending of the edge, along the projected vertex normal in screen space. This ensures continuity along the silhouette, as depicted in Figure 4.

![Figure 4: v1 and v2 are the edge vertices, and n1 and n2 are the vertex normals.](image)

In some cases, this solution may produce an error when the extrusion direction has a different direction than the projected normal version. In order to solve this, we have designed two possible solutions:

- Invert the extrusion direction in this vertex.
- Invert the direction of the projected normal.

These two solutions are illustrated in Figures 5(a) and 5(b), respectively. As this problem usually occurs when edges are hidden by a front-facing polygon, by reversing the extrude direction the hidden edges are revealed across the visible geometry, and therefore an artifact appears. Therefore, we decided to use the second solution as it is the one that produces more visually pleasing results.

The Geometry Shader pseudo code is shown in Algorithm 1:

Figure 5: v1 and v2 are the edge vertexes, e is the extrude direction, n2 is the vertex normal, e’ is the inverted extrude direction and n2’ is the inverted projected normal.

Algorithm 1: Algorithm that extrudes the silhouette edge.

The results of the applied technique are shown in Figure 6. Image a shows a toon shaded object and image b contains the GPU rendered silhouettes.

4. Silhouette texturing

Once we have a geometry extruded for the silhouette, we want to texture it in order to simulate different artistic styles. In order to do this, we must be able to generate texture coordinates for all the new geometry. The texture coordinates must fulfill two conditions:

- Intra-edge continuity.
- Inter-edge coherence.

For a single edge, texture coordinates must be generated continuously, this means that we should not find sudden jumps in the stylized edge. For a complete silhouette, we must avoid different texture coordinates for vertexes shared by two different edges, and we must guarantee the same (or similar) texture coordinate variation across the whole silhouette.

In order to do this, we generate texture coordinates in screen space, as we explain in the remainder of this section. By assigning the texture coordinates in screen-space, we assure frame coherence and constant size of the texture, independently of the complexity of the source model. Unfortunately, we still have a disadvantage: we cannot ensure a steady growth in any edge direction. However, this creates no visible artifacts in most cases.

Texture coordinates are built using two factors:

- Screen-space angular coordinates with respect to the center of the object.
- Screen-space distance to the center of the object.

As we will show later, both parameters are required because certain objects (such as star-shaped ones) might have edges that have little angular distance between the vertexes that share it.

Figure 6: Top: Toon shaded Asian Dragon model with no silhouette, bottom: The same model with silhouettes.
4.1. Angular coordinates

To create the texture coordinates we project the center of the axis aligned bounding box (AABB) on the screen. Then, we calculate the polar coordinates of the vertexes in screen space with the origin in the projected center of the AABB (Fig. 7).

When we have the polar coordinates of each vertex we can compute the texture coordinates. We calculate the \( u_\alpha \) coordinate using equation (1).

\[
    u_\alpha = \frac{\alpha}{2 \pi},
\]

(1)

where \( \alpha \) is the angular coordinate in radians: the angular distance of vertex \( v \) with respect to the \( X \) axis starting at point \( x \) (see Figure 7).

The final coordinates is obtained by using the interpolation proposed by McGuire and Hughes [MH04] (Fig. 8).

To ensure that the texture coordinates are correctly generated for all pixels, these are created in the pixel shader. This method needs to make some changes in the interpolation (Fig. 8), as we need to store the vertex position in screen-space in instead the \( u \) coordinate. Despite that, this solution release the Geometry shader of some computations, and therefore slightly improves framerates.

We may see the result of this texture coordinate generation in Figure 9.

**Figure 7:** \( v_1 \) and \( v_2 \) are the edge vertexes, \( c \) is the center of the bounding box projected in screen-space, \( x \) is the \( X \) axis direction, \( \alpha_1 \) is the angular coordinate of \( v_1 \), \( \alpha_2 \) is the angular coordinate of \( v_2 \), \( d_1 \) is the radial coordinate of \( v_1 \) and \( d_2 \) is the radial coordinate of \( v_2 \).

**Figure 8:** Texture coordinates.

**Figure 9:** Texture coordinates generated using angular distance.

Note that using the angular distance may not be enough, as for star-shaped objects or other that contain edges whose direction points toward the center of the object, the coordinate values might change just slightly. Thus, for changing textures, we may see little variation across these edges, as shown in Figure 10.

**Figure 10:** Angle-based texture coordinates may generate non coherent information for edges pointing to the center of the object.

This can be solved adding a second parameter to the texture coordinates: the distance to the projected center of the bounding box, as explained next.

4.2. Distance coordinates

In order to improve the coherency in texture coordinates between edges, we added a second parameter: the distance from the endpoints of the edge to the projected center of the object. Therefore, the value for \( u_d \) will be:

\[
    u_d = k \cdot d,
\]

(2)
where \( d \) is the radial coordinate and \( k \) is a constant that indicates the percentage of radial coordinate that is applied to the texture coordinate. Finally, \( u \) coordinates are computed by combining both expressions:

\[
u = u_{\alpha} + u_{d} = \left( \frac{\alpha}{2 \pi} \right) + (k \cdot d),
\]

(3)

For many objects, \( k \) can be close to zero, otherwise, we may adjust it manually with a slider. And the \( v \) coordinate is generated the same way than before.

Apart from that, we may also decide how many times the texture is applied for a variation of coordinates from 0 to 1. This is achieved using a multiplier over the \( u \) resulting coordinates.

This alleviates the coordinate coherency problem, as it is shown in Figures 11 and 12.

![Texture coordinates coherency improved by incorporating distance to angle-based texture coordinates](image)

**Figure 11:** Texture coordinates coherency improved by incorporating distance to angle-based texture coordinates (a and c).

![Texture coordinates coherency improved by incorporating distance to angle-based texture coordinates](image)

**Figure 12:** Texture coordinates coherency improved by incorporating distance to angle-based texture coordinates.

The fragment shader that generates these texture coordinates is sketched in Algorithm 2.

5. Discussion

The method presented here has several advantages over previous works. First, compared with many others, it does not require a preprocessing of geometry: The only real necessary processing is the correct ordering of the indices of the geometry being sent to the GPU. This means that no extra information must be encoded in vertex attributes or textures. Moreover, it is a single-step method, and its cost is affordable, as the amount of processing being done at the Geometry Shader is not too high.

Second, it generates coherent texture coordinates, as shown in Figure 13, as opposed to previous approaches. Thus, we are able to use textures freely to render stylized silhouette lines. Texture coordinates are generated in a deterministic way, by using the angular distance from the X axis traced from the projection of the center of the object onto screen, and the screen-space distance to the center. In many cases, the angular distance is enough, but in some special cases, this might not be sufficient, such as when some edges are almost parallel to X axis or to any rotation of this axis with respect to the center of the object. As a consequence, the user may determine the weight applied to the distance value.

Despite these improvements, texture coordinates may change a lot in regions close to the center of the object (for instance if the edges are roughly defining a circle), and this produces artifacts in texture application, although hardly visible in many cases (see Fig 14).

Texture application allows us to simulate different effects, and some of the parameters are parameterizable, such as edge width or texture repetition. We may see different shadings of the bunny model in Figure 15.

We show the timings in Table 1, note how we obtain...
real time even for models of up to 1M triangles. The results have been obtained with a large viewport resolution: 1680×1050 with a 2GB Quad PC with a GeForce 9800 GX2 GPU. Dyken [DRS08] generates silhouettes on the Geometry Shader, but they are not extruded. Our approach does extrusion and texturing with similar (slightly slower) frame rates. Compared to Doss [Dos08], our silhouettes are continuous, without gaps, and they are textured coherently.

By combining different textures and different shading parameters, we may obtain different effects. In Figure 16 we show several models shaded using different techniques.

In future we want to use our algorithm for crease rendering together with suggestive contours [DFRS03, DFR04].

6. Acknowledgments

This project has been partially supported by TIN2007-67982-C02-01 of the Spanish Government.

References


[Eve02] EVERITT C.: One-Pass Silhouette Rendering

Figure 15: Different stylized lines for the bunny model.

Figure 14: Zoom in of the center of the view, where texture coordinates may introduce small artifacts.
### Table 1: Comparison of framerates of our algorithm for different models.

We render the models using toon shading (column 3 shows the framerate of the toon shading algorithm alone), the method by Doss [Dos08] (fourth column), and our approach without and with texturing of the generated edges (last two columns, respectively).

<table>
<thead>
<tr>
<th>Model</th>
<th>Triangles</th>
<th>Toon Shading</th>
<th>Geom. Silhouette</th>
<th>Our technique</th>
<th>Textured edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Athenea</td>
<td>15014</td>
<td>534</td>
<td>379</td>
<td>365</td>
<td>275</td>
</tr>
<tr>
<td>Bunny</td>
<td>69666</td>
<td>441</td>
<td>159</td>
<td>152</td>
<td>125</td>
</tr>
<tr>
<td>Dragon</td>
<td>100000</td>
<td>429</td>
<td>108</td>
<td>99</td>
<td>65</td>
</tr>
<tr>
<td>Asian Dragon</td>
<td>225588</td>
<td>242</td>
<td>53</td>
<td>49</td>
<td>33</td>
</tr>
<tr>
<td>Armadillo</td>
<td>345944</td>
<td>230</td>
<td>42</td>
<td>39</td>
<td>25</td>
</tr>
<tr>
<td>Dragon</td>
<td>871414</td>
<td>254</td>
<td>20</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>Buda</td>
<td>1087716</td>
<td>220</td>
<td>16</td>
<td>15</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 16: Different stylized lines for different models.


