1.5

Silhouette Geometry Shaders

Jörn Loviscach

Introduction

Silhouettes suffer much from a coarse tessellation of 3D objects. We propose two different methods to improve their look: first, a method to render semi-transparent halos with sub-polygon accuracy, see Figure1.5.1, second, a method to fake curved silhouettes for convex parts of 3D meshes, see Figure 1.5.2. In spirit, the latter method resembles the CPU-based "Silhouette Clipping" [Sander00]. Both presented approaches employ special pseudo-geometry besides the original mesh. A vertex shader detects the silhouette and extrudes quadrangles from the pseudo-geometry along it. For the halo, we use one quadrangle per triangle of the original mesh; we extrude quadrangles from its edges, however, to smooth the silhouette visually.

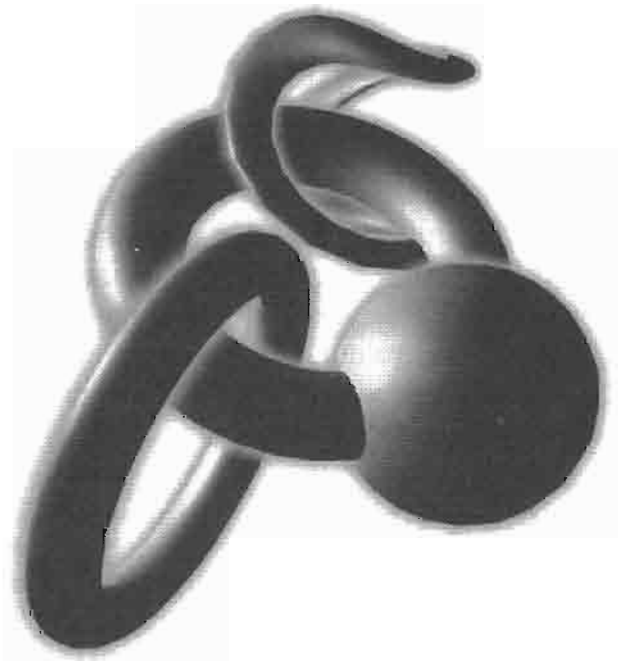

FIGURE 1.5.1 *A soft shining halo can be implemented by extruding quadrangles from the silhouette.*

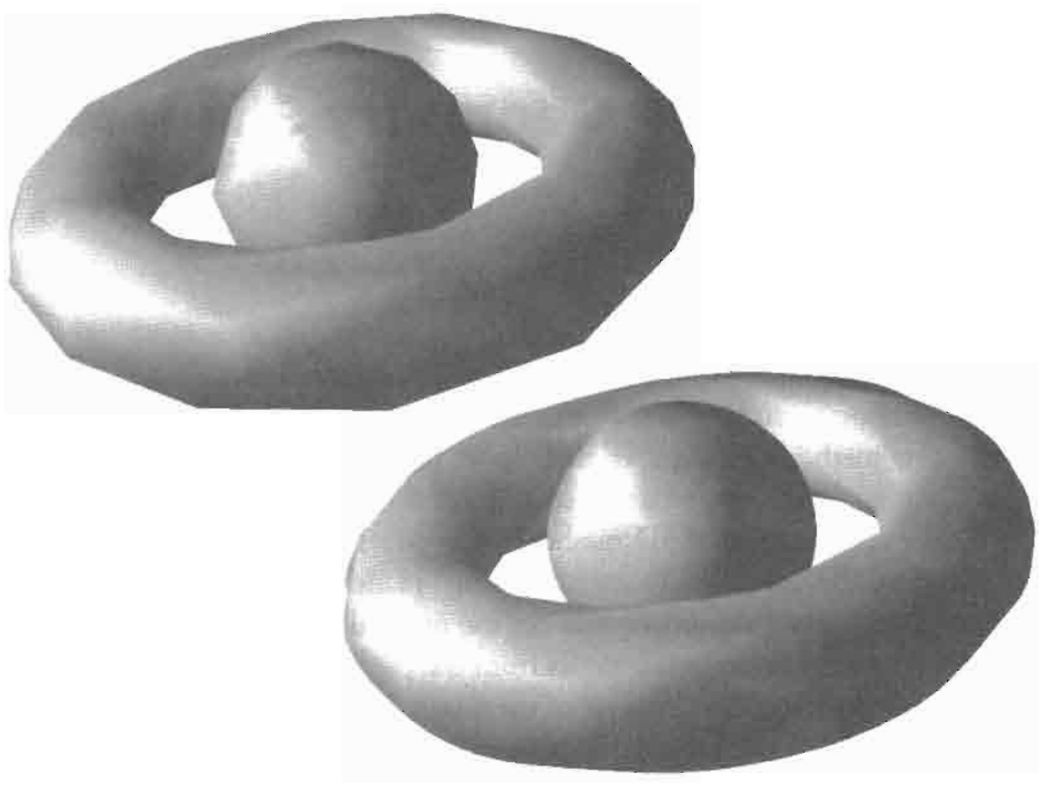

FIGURE 1.5.2 *To hide the angular look of a coarse mesh (left) we add curved fins to its silhouette (right).*

The extrusion of the edges of a mesh via a vertex shader has already been used for cartoon effects [Card02, McGuire04, Loviscach04] and for the computation of soft shadows [Arvo04]. In a straightforward manner [McGuire04] it can be applied to generate shadow volumes. Because this effect has not been used much, we elected to also include a shadow volume computation in the demo code for the present chapter. The silhouette extraction via a vertex shader is the same for the shadow volumes as for the curved silhouettes.

Sub-Polygon Accuracy Halos

The typical way to extract silhouettes from a mesh is to find edges that are adjacent to both a forward-facing and a backward-facing polygon. While this criterion is geometrically precise, it introduces complex zigzag structures [Isenberg02], which may not be favorable for many applications.

The demo code shows a simple example of such an application: the silhouette edges are extruded into a halo, which is perpendicular to the local viewing direction. They are painted with an alpha gradient in order to simulate a smooth glow. The glow is rendered using additive alpha blending. Here, zigzag layers of the halo become immediately visible. The resulting artifacts may be diminished by using alpha blending with the maximum operation. However, this workaround introduces strange shapes where the halos of glowing objects intersect in screen space.

Sub-polygon accuracy silhouettes [Hertzmann99, see also Appendix A of McGuire04] come to the rescue. They do not run along the edges of the original mesh but across its faces, see Figure 1.5.3. For each vertex the dot product of the vertex normal **n** and a unit vector **v** to the viewing position is formed. If the result is positive for three vertices of a triangle, it is considered as facing forward; if the resul't is negative for all, it is considered as facing backward. In both cases it is discarded as not containing part of the silhouette. All other triangles contain two edges that intersect the silhouette, see Figure 1.5.4.

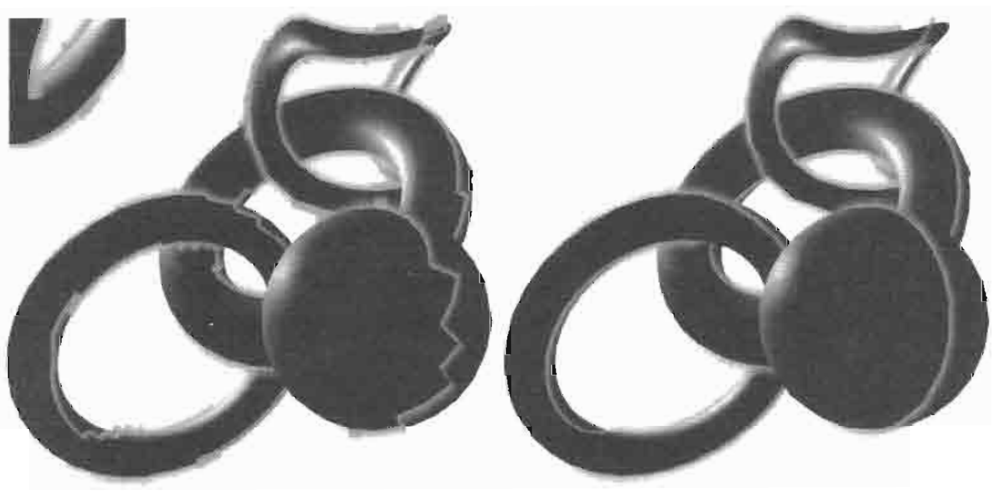

FIGURE 1.5.3 *While edge-based silhouettes (left: seen from a different view point) display strong zigzag structures resulting in artifacts (inset upper left), sub-polygon accuracy silhouettes (right) are virtually clean.*

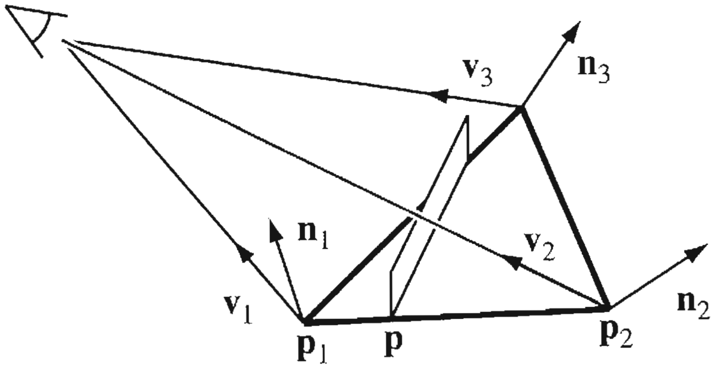

FIGURE 1.5.4 *The silhouette line is computed through linear interpolation and extruded into a quadrangle.*

There is a silhouette between the vertices 1 and 2 if $\mathbf{n}_1 \cdot \mathbf{v}_1$ and $\mathbf{n}_2 \cdot \mathbf{v}_2$ have different signs, where $\mathbf{v}_1$, $\mathbf{v}_2$ denote the local directions to the viewing position. Thus, it suffices to check whether $(\mathbf{n}_1 \cdot \mathbf{v}_1)\cdot(\mathbf{n}_2 \cdot \mathbf{v}_2)<0$. The actual intersection point $\mathbf{p}$ on the edge 1-2 can be estimated through linear interpolation: $\mathbf{p} = ((\mathbf{n}_1 \cdot \mathbf{v}_1)\mathbf{p}_2 - (\mathbf{n}_2 \cdot \mathbf{v}_2)\mathbf{p}_1)/(\mathbf{n}_1 \cdot \mathbf{v}_1 - \mathbf{n}_2 \cdot \mathbf{v}_2)$, where $\mathbf{n}_1$, $\mathbf{n}_2$, $\mathbf{v}_1$, $\mathbf{v}_2$ are the vertex normals and the unit vectors to the viewing position at the edge's end points. To generate a halo, the line across the triangle from one intersection with an edge to the other has to be extruded.

To implement this through a vertex shader, we generate a vertex buffer containing four vertices per face of the original mesh and an index buffer connecting them to two triangles forming a quadrangle. The shader determines unit vectors from all three points to the viewing position and computes the dot products with the corresponding vertex normals. Thus, all of this data has to be present within every vertex. We use four vertices per triangle with the following attributes:

$$\mathbf{p}_1, \mathbf{n}_1, \mathbf{p}_3, \mathbf{n}_3, \mathbf{p}_2, \mathbf{n}_2, u = 1, v = 0 \tag{1.5.1}$$

$$\mathbf{p}_1, \mathbf{n}_1, \mathbf{p}_2, \mathbf{n}_2, \mathbf{p}_3, \mathbf{n}_3, u = 0, v = 0 \tag{1.5.2}$$

$$\mathbf{p}_1, \mathbf{n}_1, \mathbf{p}_3, \mathbf{n}_3, \mathbf{p}_2, \mathbf{n}_2, u = 1, v = 1 \tag{1.5.3}$$

$$\mathbf{p}_1, \mathbf{n}_1, \mathbf{p}_2, \mathbf{n}_2, \mathbf{p}_3, \mathbf{n}_3, u = 0, v = 1 \tag{1.5.4}$$

Two vertices contain the points in the order 1-2-3, the other two use the order 1-3-2. This change in arrangement allows simplifying the logic of the shader: We do not need to test all three, but only two edges per vertex for an intersection with the silhouette. If there is a silhouette between the first two points, the shader sets the resulting position to the linear interpolation of them. If there is not, the shader tests if there is a silhouette between the last two points and sets the resulting position correspondingly. If either test fails, there is no silhouette at all on the triangle, and the position is set to a fixed far position. The latter will happen to all four vertices so that the triangle is not only reduced to zero area but will also be clipped away. As an example, consider a triangle where the silhouette intersects the edges 2-3 and 1-3. The position of the first and the fourth vertex will be set to the intersection on the edge 2-3; the second and the third vertex will be positioned on the edge 1-3.

The v coordinate of the vertices is used to control the extrusion: The 3D point determined as described beforehand is offset by v times a vector pointing outside the mesh. Here, we employ the interpolated normal vector $\mathbf{n} = ((\mathbf{n}_1 \cdot \mathbf{v}_1)\mathbf{n}_2 - (\mathbf{n}_2 \cdot \mathbf{v}_2)\mathbf{n}_1)/(\mathbf{n}_1 \cdot \mathbf{v}_1 - \mathbf{n}_2 \cdot \mathbf{v}_2)$, which is in good approximation perpendicular to the local viewing direction. The u coordinate is only used for illustration.

The resulting kind of halo rendering works fine for convex objects. Concave objects may show isolated bursts of light on their faces. This effect, however, is much more intense with a silhouette generated from the edges of the mesh instead of the proposed method.

Thanks to their smoothness, sub-polygon accuracy silhouettes may also prove helpful in the generation of soft shadows through quadrangles appended to the silhouette in a shadow map rendering [Arvo04]. Somewhat surprisingly, sub-polygon accuracy silhouettes do not prove very useful to render shadow volumes: the relationship between the vertex normals and the actual geometry of the mesh is quite complex; in principle, the vertex normals may even be chosen freely. This leads to situations where back-facing parts of a shadow volume lie outside of the corresponding front-facing part. Such geometry causes a bright area in the shadow, as can be seen with complex meshes in the demo code. Note that to get useful shadow volumes at all, the silhouettes have to be oriented correctly. As it turns out, all that is needed in the vertex shader is an inversion according to the sign of $\mathbf{n}_1 \cdot \mathbf{v}_1$.

Sub-polygon accuracy silhouettes offer the advantage that the corresponding vertex and index buffers can easily be built from the data of the original mesh. There is no need to collect the edges of the mesh. The data needed amounts to *six* vectors plus one float per vertex with four vertices per *triangle*. An edge-based halo (see next section) needs *four* vectors plus one float per vertex with four vertices per *edge*. On a closed mesh, there are three half edges for every triangle, so that in total the sub-polygon accuracy method uses a little less data.

Curved Silhouettes

The convex parts of polygonal silhouettes can be smoothed by appending curved fins to them. We use a pixel shader to cut these fins from quadrangles that are perpendicular to the silhouette, see Figure 1.5.5. The quadrangles themselves are extruded from pseudo-geometry based on the edges of the original mesh.

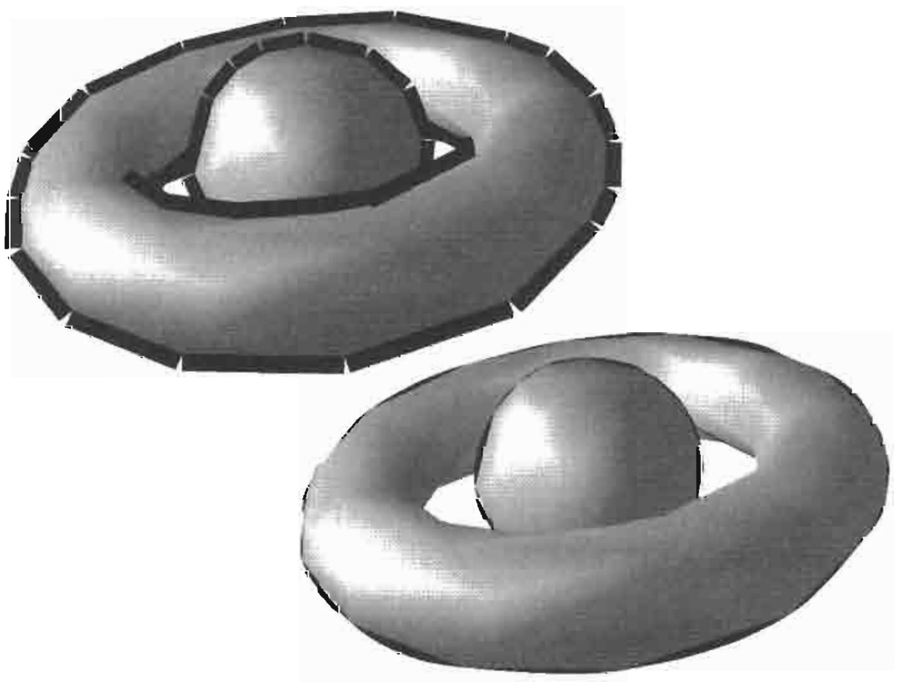

FIGURE 1.5.5 *Quadrangles that extend perpendicularly from the silhouette (left) are trimmed to only display the curved fins (right).*

The vertex buffer of this pseudo-geometry contains four vertices per edge of the original mesh; again, an index buffer is used to connect those four vertices to two triangles forming a quadrangle. The attributes of the four vertices are selected as follows:

$$\mathbf{p}_1, \mathbf{n}_1, \mathbf{p}_2, \mathbf{n}_{\mathrm{I}}, \mathbf{n}_{\mathrm{II}}, u = 0, v = 0 \tag{1.5.5}$$

$$\mathbf{p}_2, \mathbf{n}_2, \mathbf{p}_1, \mathbf{n}_{\mathrm{I}}, \mathbf{n}_{\mathrm{II}}, u = 1, v = 0 \tag{1.5.6}$$

$$\mathbf{p}_2, \mathbf{n}_2, \mathbf{p}_1, \mathbf{n}_{\mathrm{I}}, \mathbf{n}_{\mathrm{II}}, u = 1, v = 1 \tag{1.5.7}$$

$$\mathbf{p}_1, \mathbf{n}_1, \mathbf{p}_2, \mathbf{n}_{\mathrm{I}}, \mathbf{n}_{\mathrm{II}}, u = 0, v = 1 \tag{1.5.8}$$

Here, $\mathbf{n}_1$, $\mathbf{n}_2$, $\mathbf{p}_1$, $\mathbf{p}_2$ are the positions and normals of the edge's end points, $\mathbf{n}_{\mathrm{I}}$ and $\mathbf{n}_{\mathrm{II}}$ are the face normals of the adjacent triangles, see Figure 1.5.6.

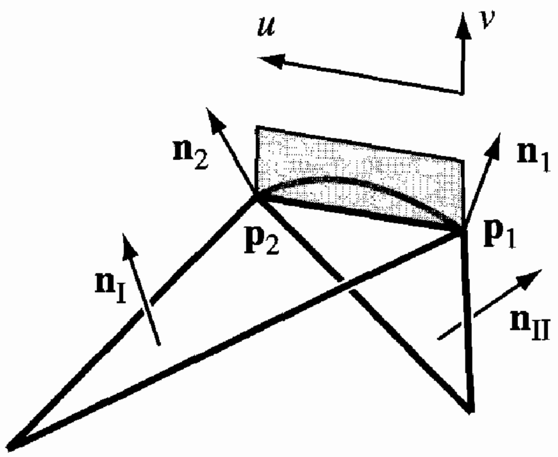

FIGURE 1.5.6 *The face normals are used to decide if an edge is to be extruded. The vertex normals, however, control the form of the curved fin.*

The vertex shader uses the first position vector to determine a vector v to the viewing position. Then it checks if $(\mathbf{v} \cdot \mathbf{n}_{\mathrm{I}}) \cdot (\mathbf{v} \cdot \mathbf{n}_{\mathrm{II}}) < 0$ which means that $\mathbf{v} \cdot \mathbf{n}_{\mathrm{I}}$ and $\mathbf{v} \cdot \mathbf{n}_{\mathrm{II}}$ have different sign, which in turn means that one of the adjacent triangles is front-facing and the other is back-facing: We have a silhouette edge. If that is not the case, the position is set to a fixed far point.

Again, v is used to control the extrusion of the edge. Using the vertex normal $\mathbf{n}_1$ projected perpendicular to the local viewing direction, the silhouette edge can be extruded into a halo. It may also be extruded along the light ray to form a shadow volume. Both uses are demonstrated in the accompanying code. Again we have to take care that the polygons bounding the shadow volumes are drawn with correct orientation. The u coordinate and the position of the other point of the edge are needed neither for halos nor for shadow volumes. Note a difference between point and directional light sources. The shadow volumes for a directional light source located behind the viewer converge at the vanishing point of the light's direction. Thus, they can be drawn extruding only a single triangle from each silhouette edge.

For curved silhouettes, we extrude the silhouette edge perpendicular to itself and perpendicular to the local viewing direction. To this end we form the cross product of the local viewing direction and the vector along the edge. The latter vector is computed using the position of both end points of the edge. It could also be computed using the normals of the two adjacent edges, but that would be computationally more intensive. Anyway, we need both end positions of the edge as attributes of every single vertex for a different reason.

The extrusion leads to quadrangles along the silhouette that appear to the viewer almost perfectly as rectangles. This simplifies the computation of texture coordinates. Furthermore, the interpolation of texture coordinates would incur a shearing effect along the diagonal of a non-rectangular quadrangle. That the extruded quadrangles do not close to a continuous silhouette does not bother us, because we will trim away from the left, upper, and right side in the pixel shader.

Using the vertex normal, the vertex shader computes the illumination at the original position of every vertex. This is automatically linearly interpolated on the way to the pixel shader so that the color gradient on the fin exactly matches that of the adjacent triangle.

We trim away the unwanted pixels of the extruded quadrangle using the HLSL instruction clip in the pixel shader. The *uv* coordinates are handed to the pixel shader so that every extruded quadrangle carries interpolated values of u ranging from 0 to 1 along the edge of the mesh and v from 0 to 1 approximately perpendicular to that. We build a cubic function $u \cdot (1-u) \cdot ((1-u)\mathbf{a} + u\mathbf{b})$ for certain **a**, **b** and check if v (apart from a scaling) is above that value. If so, the pixel is clipped; otherwise it is painted. **a** and **b** are determined from the vertex normals in such a way that the resulting curve runs perpendicular to the projection of the vertex normal onto the screen. This is the part of the computation, where we need the other point of the edge: to compute the vector from one point of the edge to the other.

The cubic method ensures that curves on two quadrangles that meet at a single vertex have the same tangent: no crease will appear between them. While the cubic expression may read complex, it can be reduced: we write **a** at one vertex of the edge, **b** at the other, and the automatic linear interpolation happening between vertex shader and pixel shader will yield $(1-u)\mathbf{a} + u\mathbf{b}$, the last factor of the cubic expression.

In this method, we do not care about perfectly accurate computation of screen-oriented rectangles. This may be done at the cost of working in screen coordinates, which means that the vertex shader has to do perspective division. Given the approximating nature of the method we elected to stick with less accuracy but also less overhead.

Conclusion and Outlook

The proposed methods can be used in many different ways to improve the look of silhouettes for intermediate levels of tessellation. On very coarse meshes they have severe difficulties in estimating the original form. In the demo code we use the mesh simplification of DirectX in order to show the behavior at different levels of tessellation.

The relatively high number of vertex attributes still poses a high load on the memory interface. This is aggravated by the circumstance that the vertices cannot be reused: while a vertex in a regular mesh may be shared by all adjacent polygons, the vertices of the pseudo-geometries introduced here depend too much on their geometric context. Using quadrangles generated through "instancing" under Shader Model 3.0 may offer improvements here. Furthermore, this shader model may offer a drastic speedup of both presented methods, because it allows an "early out" in the shader and will let it finish quickly if there is no silhouette. However, first experiments show that current hardware profits from this only if the number of code lines that are branched over is much larger than used here for silhouette geometry.

References

[Arvo04] Arvo, Jukka, and Westerholm, Jan, "Hardware Accelerated Soft Shadows Using Penumbra Quads," *Journal of WSCG,* Vol. 12 (2004), No. 1, pp. 11–18.

[Card02] Card, Drew, and Mitchell, Jason L., "Non-Photorealistic Rendering with Pixel and Vertex Shaders," *ShaderX: Vertex and Pixel Shaders Tips and Tricks,* ed. Wolfgang Engel, Wordware, 2002, pp. 319–333.

[Hertzmann99] Hertzmann, Aaron, "Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines," SIGGRAPH Course Notes, 1999.

[Isenberg02] Isenberg, Tobias, Halper, Nick, and Strothotte, Thomas, "Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes," *Computer Graphics Forum* (EUROGRAPHICS 2002), Vol. 21 (2002), No. 3, pp. 249–258.

[Loviscach04] Loviscach, Jörn, "Stylized Haloed Outlines on the GPU," SIGGRAPH 2004 Poster.

[McGuire04] McGuire, Morgan, and Hughes, John F., "Hardware-Determined Feature Edges," Proc. NPAR 2004, pp. 135–145.

[Sander04] Sander, Pedro V., Gu, Xiafeng, Gortler, Steven, Hoppe, Hugues, and Snyder, John, "Silhouette Clipping," Proc. SIGGRAPH 2000, pp. 327–334.