Subdivision of Silhouette Edges in Interactive Time

Florian Boulnois, Michael Andryauskas

Advanced Computer Graphics

6/5/2011

Abstract:

Our project aims to detect and subdivide the silhouette edges of triangle polygonal meshes. Such silhouetting is useful for character and object prototyping, and may be used to give 3D objects a sketched or cartoony feel. Furthermore, subdividing the entire mesh may be a very expensive operation, and the silhouette provides a good target for subdivision without dramatically increasing polygon count.

Introduction:

Silhouetting is an important part of various subjects in computer graphics, including technical illustrations (Cole et al., 2008), line drawings (Judd, Durand and Adelson, 2007) and non-photorealistic rendering (NPR) (Hartner et al., 2003). Hertzmann and Zorin (2000) showed that such silhouetting can also be used to calculate shadow volumes. Such silhouettes can also have important applications in video games, where objects or characters may need to be quickly recognized (Mitchell, Francke, and Eng, 2007). Benichou and Elber (1999) define a silhouette edge as follows: E_i is a silhouette edge if one of its two associated polygons is locally visible from the view vector $\vec{V}$ and the other is locally invisible. A polygon is considered locally visible if the dot product between the view vector $\vec{V}$ and the polygon normal $\vec{N}< 0$, and invisible if the dot product between the view vector $\vec{V}$ and the polygon normal $\vec{N} > 0$. Such edges are extremely valuable since they provide significant cues to the objects shape. Note that these silhouette edges are view dependent, that is, changing the camera angle will change the set of

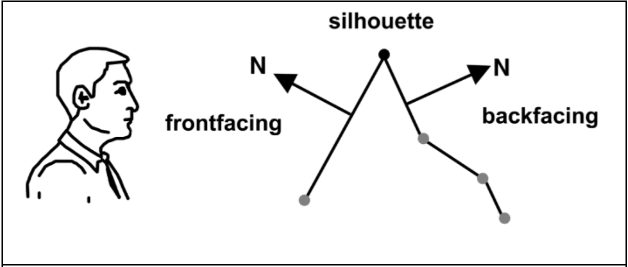

Figure i: Example of front-facing and back-facing polygons. (Hartner et al., 2003)

silhouette edges. Silhouette edges have another interesting property, in that a chain of silhouette edges always form loops on watertight models. Thus, such chains can be followed along the model rapidly.

Subdivision is another important part of computer graphics, because it allows relatively low polygon count models to have higher polygon counts. This is useful for a variety of applications, including real time graphics on mobile devices (Hertzmann and Zorin, 2000). However, on such restricted devices, subdividing the entire model may be costly and wasteful – because the silhouette provides the shape and feature curves of the surface, it may be better to simply subdivide the silhouette. Zorin and Hertzmann outline several constraints for a good subdivision scheme: Interpolation, Locality, Symmetry, Generality, Smoothness, and Simplicity. They present a modified butterfly subdivision scheme which can satisfy all these requirements.

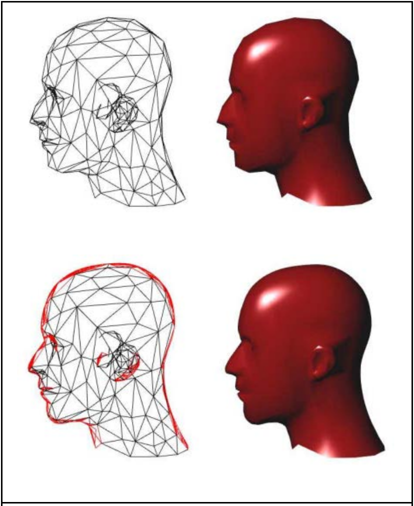
Figure ii: An example of silhouette smoothing. (Wang et al., 2010)

The butterfly subdivision scheme is simple to implement, but unfortunately can only work with vertices that have 6 edges (valence 6). Zorin and Hertzmann expand this algorithm to work on arbitrary valence meshes. Their subdivision scheme computes a new vertex for each edge midpoint of a triangle, weighing the position of the vertex based on the surrounding vertices. In the typical subdivision scheme, those other vertices of the polygons make the shape of a butterfly, hence the name. In the modified scheme, they consider additional polygons and edges for the weighing scheme which allows them to generalize the system for arbitrary valence meshes.

Starting from the work of Zorin and Hertzmann (2000) and Wang et al, we present a method to rapidly subdivide silhouette edges by using a modified butterfly subdivision scheme. We only consider the boundary edge case in the Zorin and Hertzmann modified subdivision scheme which uses the weight of 4 points to generate new vertices and triangles and generalize that to our subdivision algorithm.

Methods:

The algorithms were tested and compiled on Lenovo T410 laptops with a 2.53GHz dual-core processor with 4GB RAM and an Nvidia NVS 3100m GPU (512MB). The framework from the first assignment (Cutler, 2011) on Gouraud shading, mesh simplification, and subdivision was used and expanded to fit our needs.

The silhouette edge detection was implemented by Florian. The first structure that was modified was the mesh code, as the silhouette edge detection needs to figure which edges are part of the silhouette in relation to the camera. The entire camera object (including viewing position and direction) was thus passed into the mesh code so that the silhouette edges could then be calculated. The algorithm then iterates through all edges and figured out which ones were silhouette edges. As mentioned previously, one face adjoining such edges have a normal which points towards the camera and the other has a normal which points away from the camera. It is thus simply a matter of comparing the position of the camera to the normal of the face using the dot product of those vectors. If the dot product is negative, the normal of the face is pointing in the opposite direction to the camera (a front-facing polygon). If the dot product is positive, the normal of the face is in is the same direction as the camera (a back-facing polygon). Once those edges are found, they need to be delivered in a useful format for silhouette subdivision. The simplest and easiest method to deliver those edges to be subdivided is by delivering a vector of vector of silhouette edges. Each vector contains a loop of contiguous

Silhouette edge detection pseudocode:

```
For each edge in the mesh
        Get polygons of edge
        Test if polygons are silhouette polygons
        If they are, store edge in vector s
While there are still silhouette edges in s
        Go through all silhouette edges
        Find out which edges link together, deleting them from s
```

silhouette edges. These vectors are themselves packed in a vector.

The silhouette subdivision was implemented by Michael. It uses a modified butterfly subdivision scheme which only considers the two edges immediately adjacent to an edge in a loop of silhouette edges and uses those four vertices to calculate the position of a new vertex in the midpoint of the edge. The weights are as follows: $v0 = -\frac{1}{16}, v1 = \frac{9}{16}, v2 = \frac{9}{16}, v3 = -\frac{1}{16}$. The new triangles for the subdivision are then generated using the boundary edge case of the modified butterfly subdivision scheme. A new edge is made from the midpoint of the triangle to the opposing vertex if there is one silhouette edge. New edges are created to the center of the triangle if there are two or three silhouette edges. The edges are subdivided twice. The silhouette edges of the triangle are recalculated when the camera moves.

Silhouette subdivision pseudocode:

```
For each series of silhouette edges
        For each edge
        Find immediately adjacent edges
        Check for sharpness (if needed)
        Calculate point on edge based on
adjacent silhouette edges
        Store silhouette triangles and their
silhouette edges
For each triangle
        If not a silhouette triangle, do not modify
        Else subdivide appropriately
Render triangles
```

Several other features were included to help visualize and debug the silhouette subdivision, implemented by both Florian and Michael. Silhouette edges can be visualized by pressing E. Gouraud shading is available and can be triggered using G. The mesh can be reset to its original state by pressing R. The program can be quit by pressing Q. The silhouette can be manually subdivided using M. Pressing P will subdivide the silhouette in real-time. Normally, a sharp crease of more than 90 degrees or more from one edge to another is considered a corner and is not subdivided in the regular manner. However, we have made this an option instead, as it looks better. Pressing V will force the algorithm to consider sharp creases and not subdivide those.

Results and Discussion:

Originally, the plan to detect silhouette edges was through a propagation algorithm, in which a single silhouette edge is detected. The algorithm then exploits the fact that silhouette edges must share additional silhouette edges and such chains of silhouette edges create loops, and would then follow that chain until a duplicate of one of the edges was detected. The advantage to such a method is that not all edges in the mesh would need to be tested. Unfortunately, such a method also has several drawbacks, including how to decide which silhouette edge to start at and detecting other silhouette edge chain loops (there might be several in a scene). Other silhouette edge algorithms were investigated from the paper "Object Space Silhouette Algorithims [sic]" (Hartner et al. 2003), but ultimately rejected due to implementation complexity or unsuitability with the subdivision scheme we would later use.

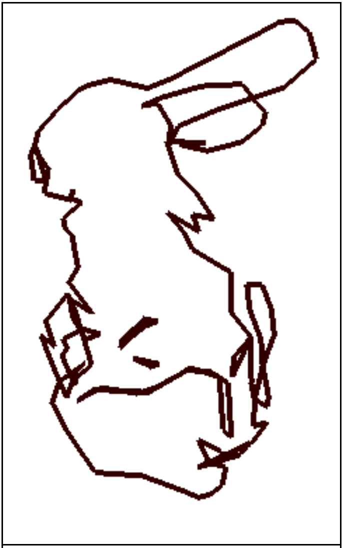
Figure 1: Exploiting loops. The model used is the 1000 face bunny.

Instead, a much simpler algorithm was implemented that could then be progressively refined. These chains of silhouette edges were then delivered in a vector that could easily be processed and visualized. Although the regular butterfly subdivision scheme requires all vertices of the mesh to have a degree of 6, the modified subdivision scheme can work with vertices of any degree. We only consider the boundary edge condition. This uses four vertices from three contiguous edges which are weighed to produce a new vertex in the center of the middle edge. Sharpness of edges can be dynamically considered using an on/off switch. Programming the entire assignment took about 20 hours.

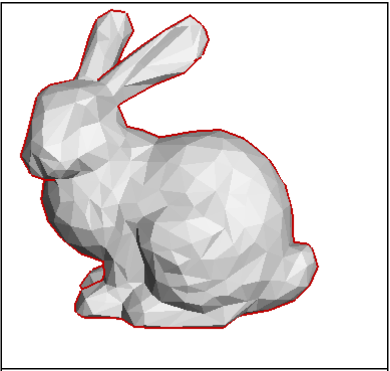
Figure 2: Silhouette edge detection of 1000 face bunny.

Several traditional computer graphics objects were rendered, along with simpler models used to debug our system. In a typical scene (for example, the 1k bunny), only 100-200 edges are silhouette edges (representing about 3-7% of all edges). Our simple examples are the sphere and cube, and for such scenes, our algorithm runs almost in real time 25-30 fps. Our medium difficulty examples are the 200 polygon bunny. Our hard examples are the knot and the bunny, which both work well with the silhouette edge subdivision. Unfortunately, our algorithm pops when rotating the model in real time. This is normal and is due to the local remeshing around the silhouette edges, which may be discontinuous from frame to frame. Additionally, our algorithm cannot deal with saddle shaped structures such as the edges that make up the inside of a torus, because in such a case the entire saddle shape structure would be covered by the edges around it. Thus, a different weighing scheme would be required.

Ultimately, we have implemented a silhouette subdivision scheme which is easy to use, has various tweakable parameters, and works well for low to medium polygon meshes, without incurring significant computational cost. In the future, we might improve our subdivision scheme by including the several other vertex weighing features in butterfly subdivision, and including a more intelligent silhouette edge detection algorithm, such as a propagation algorithm. We might also try to accelerate the underlying vector structures we used such that rendering time could be done much more quickly, so that the program would truly be in real time.

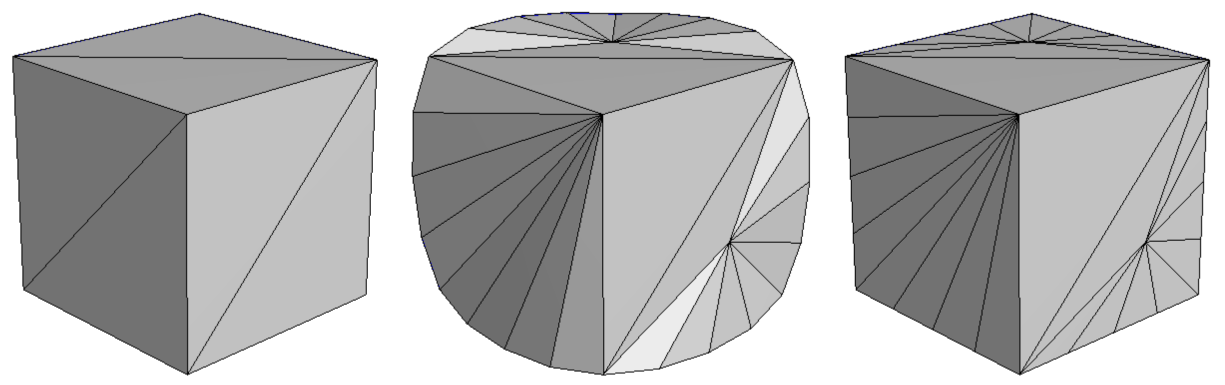

Figure 3: On the left, the original cube model. In the middle, the silhouette edges of the cube have been subdivided, ignoring edge crease sharpness. On the right, the silhouette edges have been subdivided, but the edge crease sharpness is taken into account.

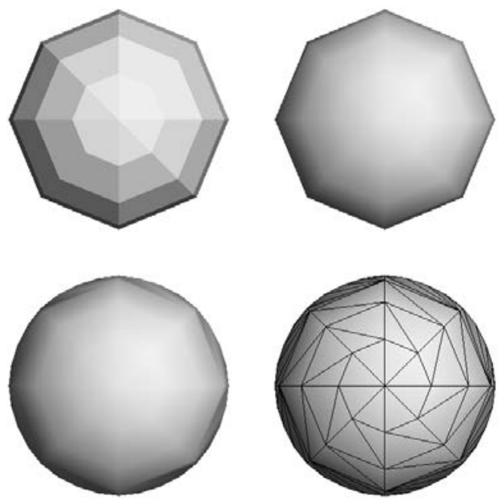
Figure 4: Silhouette subdivision of the sphere. Although the model is fairly simple, the subdivision makes the sphere look round.

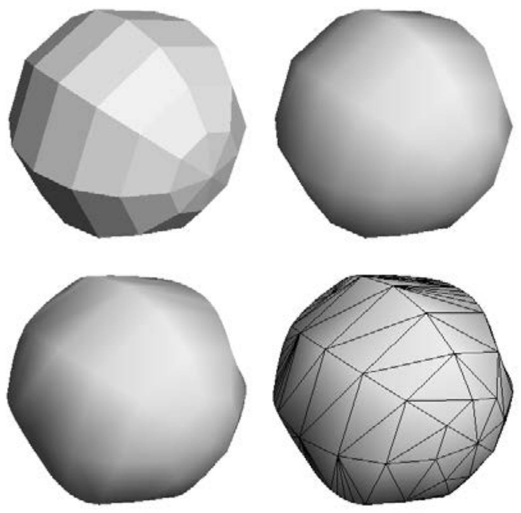
Figure 5: Silhouette subdivision of the sphere. Due to the edges making up the silhouette edges in this particular view, the subdivision appears odd.

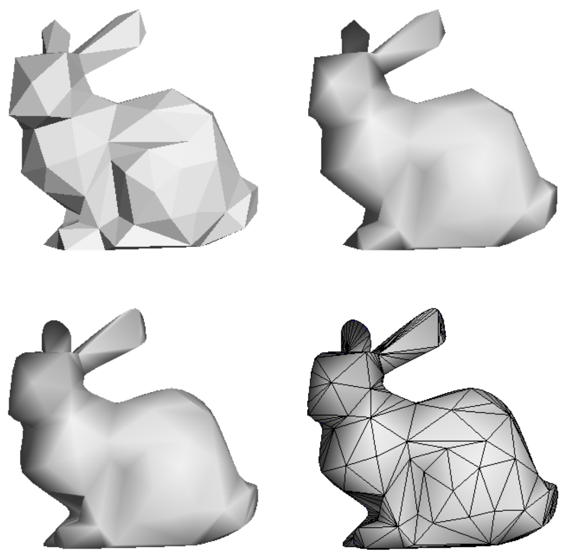
Figure 6: Silhouette subdivision of the 200 polygon bunny. The bunny appears smooth even though the entire model has not been subdivided.

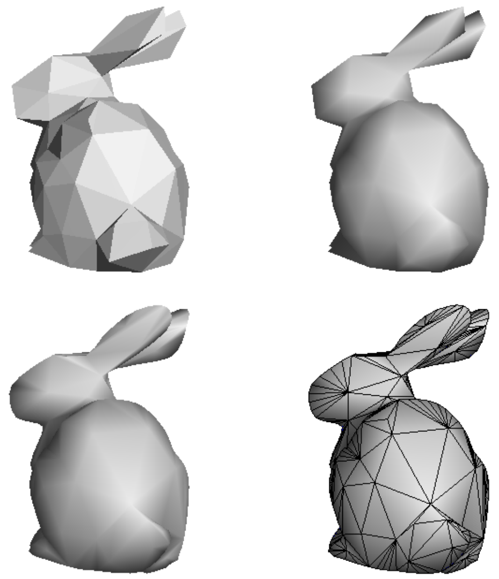
Figure 7: Another view of the 200 polygon bunny. The bunny appears smooth from this angle as well, especially the ears, nose, and paws.

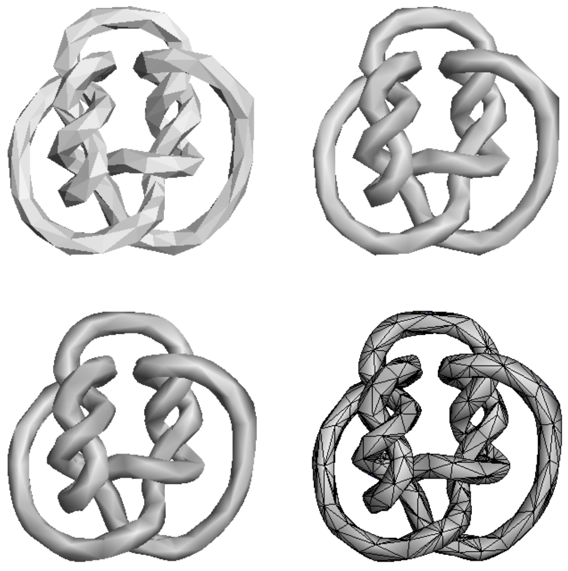

Figure 8: Silhouette subdivision of the knot. The knot appears smooth after the subdivision, especially because many of its edges make up the silhouette.

References:

1. Cole F., Golovinskiy A., Limpaecher A., Stoddart Barros H., Finkelstein, A., Funkhouser T., and Rusinkiewicz S. (2008). “Where Do People Draw Lines?” In proceedings of ACM Transactions on Graphics, SIGGRAPH 2008.
2. Adelson E., Judd T., Durand F., (2007). “Apparent Ridges for Line Drawing” In proceedings of ACM Transactions on Graphics, SIGGRAPH 2007.
3. Hertzmann, A., and Zorin, D. (2000). “Illustrating Smooth Surfaces.” In Proceedings of ACM Transactions on Graphics, SIGGRAPH 2000.
4. Mitchell, J., Francke, M., and Eng, D. (2007) “Illustrative Rendering in Team Fortress 2.” Valve Software. URL: http://www.valvesoftware.com/publications/2007/NPAR07_IllustrativeRenderingInTeamFortress2.pdf
5. Benichou, F., and Elber, G. (1999). “Output Sensitive Extraction of Silhouettes from Polygonal Geometry”. In proceedings of the Seventh Pacific Conference on Computer Graphics and Applications (PG99).
6. Hartner A., Hartner M., Cohen E., Gooch B., (2003). “Object Space Silhouette Algorithims [sic]”.
7. Wang L., Tu C., Wang W., Meng X., Chan B., Yan, D. (2010). “Silhouette Smotthing for Real-time Rendering of Mesh Surfaces”.