Sixth Hungarian Conference on Computer Graphics and Geometry, Budapest, 2012

Real-Time Outline Visualization for Triangle Meshes on Modern GPUs

Balázs Hajagos, László Szécsi

Abstract

This paper presents an outline visualization technique which renders the silhouettes and creases of triangle mesh models in real time. We perform preprocessing of the geometry in order to augment vertex records with crease edge information, and we use the geometry shader to identify and render relevant outline edges. We show that the approach can be efficiently implemented to offer robust, high quality outline rendering in real-time.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image Generation: Line and curve generation

1. Introduction

Outline visualization is extensively used in a wide range of applications, from CAD systems to non-photorealistic rendering (NPR). It can clarify the shape of a complex 3D object or may highlight essential features. The human visual system processes seen images by identifying shapes separated by discontinuities. Outline rendering provides strong cues for shape separation, substituting for subtle and expensively rendered real-world cues like scattered lighting and shadows, and providing a stronger visual language in stylistic rendering. Cartoon shading[13], in particular, relies on edge visualization to convey shape information, in lieu of realistic shading. Hatching[15], which may convey shape by applying hatch lines following object curvature, is also combined with outline rendering in most artistic styles.

There are two classes of outlines that need to be drawn, both indicating some kind of perceived discontinuity (see Figure 1). *Silhouettes* appear at discontinuities in image space, where a continuous object surface appears to end. For manifold surface models this can happen only where the surface folds behind itself, meaning that outlines are located on the border of the visible (camera-facing) and not visible (back-facing) part of the object surface. The other class of displayed outlines — called *creases* — indicate discontinuities in the surface normals, and they are defined by the topology of the mesh itself, independent of the view direction or the camera settings.

In stylistic rendering, outline drawings may feature lines other than silhouettes and creases. Suggestive contours[3] and apparent ridges[5] define outlines based on surface curvature characteristics. While these can provide superior visual cues, especially in absence of additional shading, they are less fit if we aim at minimal-overhead real-time rendering[2].

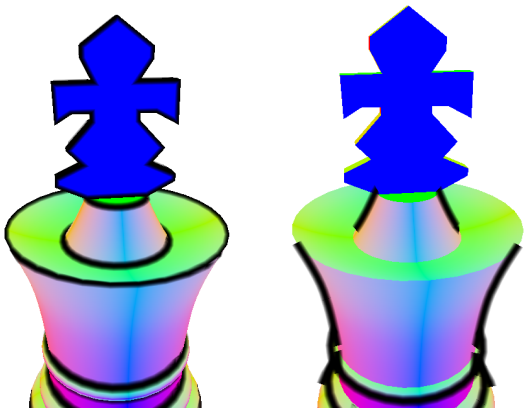

Figure 1: *Crease (left) and silhouette (right) outlines.*

We propose a solution that can render outlines with a performance similar to regular incremental triangle mesh rendering, but meeting quality standards set by offline NPR methods. The particular contribution of this work is

- a preprocessing algorithm which augments mesh vertices with crease information, and
- a real-time outline rendering algorithm which

– does not need multiple passes,
– does not use image processing filters,
– renders antialiased, continuous, textured, alpha-blended outlines of any width without seems or folds, and
– provides flicker-free animation.

The goal is to provide a solution which can replace costly edge detection filters with a more flexible, geometry-aware method in real-time applications, and offer a faster alternative in stylistic rendering.

2. Previous work

There are two well known approaches to outline rendering. The first one works in image space with the use of normal and depth maps[8]. Edge pixels — those which lie near discontinuities in these maps — can be found using edge detection filters. What level of image-space discontinuity warrants outline edges must be adjusted by fine-tuning filter parameters and applying mask textures[10]. Object-space consistency of outlines during animations is also subject to those parameters. However, the main problem with this approach is the excessive texture access bandwidth and the absence of real scalability in line features.

The other approach works in world space and generates new triangle strip geometry to visualize the outlines. In this case we do not need to search on per pixels basis.

Raskar[9] distinguishes front-facing and back-facing triangles, and augments all of them with cleverly placed quadrilaterals along their edges. A custom depth-tesing algorithm makes sure that they are only visible at outlines, valleys and ridges. The solution does not need any preprocessing, adjacency information or geometry shaders, but it is prone to z-fighting artifacts and cracks showing up in wider outlines. Also, valleys and ridges disregard triangle mesh topology, and thus may not coincide with actual creases. In today's applications, mesh adjacency information is widely used, and can be seen as given, thus the brute-force approach of the method no longer seems justified.

There are two basic ways to define silhouettes on triangle meshes. The approach by Markosian et al.[7] operates on the discrete triangle mesh geometry itself, selecting those edges as silhouette edges which separate front-facing and back-facing triangles. The silhouette looks smooth, but it often backtracks in the image plane. The definition by Hertzmann and Zorin[4] avoids this problem, as it considers the smooth surface instead of the triangulated one, reconstructing silhouettes from the vertex normals. For a given vertex $\mathbf{a}$ with normal $\mathbf{n_a}$ and vector $\mathbf{c_a}$ to the camera, we define the scalar field $f(\mathbf{a}) = \mathbf{n_a} \cdot \mathbf{c_a}$, extending f to triangle interiors by linear interpolation. Silhouettes are taken to be the zeroset of f, yielding clean, closed polylines whose segments traverse faces in the mesh — rather than following edges, as in the Markosian method.

Kalnins et al.[6] presents an impressive toolbox of NPR techniques, including aesthetical, parametrizable outlines. They combine Hertzmann-and-Zorin-style silhouette segments into continuous curves, ensuring artifact-free connections, but incurring an overhead which makes real-time application less appealing.

We propose a method which can do a similarly good job with both silhouette and crease outlines in real time — even though we do not provide arc-length parametrization for the outline curves. The method works in world space and uses the geometry shader to generate outline segments. In order to ensure that these segments fit together perfectly and form artifact-free outlines, we use both adjacency information and crease edge information — which must be precomuted for a mesh. This algorithm fulfills peformance and quality requirements posed in various applications, ranging from games through CAD systems to production rendering.

Self-similar and procedural texturing and deformation of outlines has also been addressed by previous research[1, 14]. These techniques make it possible to texture and deform arbitrarily parametrized curves in a visually uniform way. Therefore, they are in good synergy with out approach.

3. Triangle mesh representation

There are two algorithms making up the method: the preprocessing which augments vertex records with crease edge information, and the rendering algorithm which can be implemented using a geometry shader. The preprocessing phase is motivated by the data needs of the rendering one. In this latter phase, we render the triangle mesh geometry with adjacency information. In the geometry shader, we need to find both crease and sihouette segments, and instantiate geometry for displaying them. The shader processing a triangle primitive can receive data from six vertices: those six whose indices have been pre-calculated into the index buffer. In a standard adjacency-aware mesh, these are the three vertices of the triangle, and those vertices of the neighbouring three triangles that do not coincide with the first three.

In a manifold triangle mesh[12, 11], all edges have two adjacent triangles. As the mesh is represented as a set of triangles, every edge is specified twice — once for each triangle. Therefore, an edge as defined by a triangle is called a *halfedge*. Any of the three halfedges of a triangle might lie on a *crease edge*. At crease edges, surface normals are not continuous, meaning that the vertex normals of at least one of the edge vertices are different for the two triangles. In triangle mesh models, this is accomplished by duplicating the vertex, creating two vertex records with identical positions, but different normals. In fact, we define crease edges topologically, as edges where the two spatially adjacent triangles do not share two vertex records. Both halfedges on a crease edge are *crease halfedges*. Creases separate meshes into topologically connected triangle groups, called *smoothing groups* in modelling. These groups are delimited by

loops of crease halfedges. We allow two such halfedges to coincide: there might be crease edges with the two adjacent triangles belonging to the same smoothing group. In order to differentiate vertex records with well-defined normals from geometry corners, we will refer to the latter explicitly as *geometry vertices*, and to the former as *topology vertices* or just vertices. Apart from those at creases, vertices and geometry vertices are identical. We use the term *crease vertex* for any vertex on a crease edge, even if it does not have spatially coinciding duplicates.

4. The proposed method

When adjacency data for a triangle mesh is calculated, geometric — and not topological — connectivity is considered. Where a triangle has a crease halfedge, the non-coincident vertex of the neighbouring triangle is stored (see Figure 2). Its other vertices are not recorded, and not passed on to the geometry shader during rendering. Thus, in order to be able to identify crease halfedges in a shader, we need additional information in vertex records.

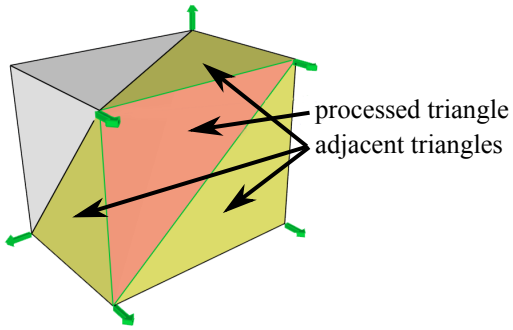

Figure 2: *Normals of vertices referenced in triangle adjacency data.*

Crease halfedges form continuous strips along crease outlines. The crease segments output by the geometry shader should fit together without discontinuities. Therefore, the directions of previous and next segments must be known. These can also not be discerned from adjacent vertex data, as adjacent segments can very well run along non-edge-adjacent triangles. As a topology vertex can have at most two crease halfedges, it would be straighforward to store their directions in vertex records. However, as this would unnecessarily increase record size and slightly impact rendering performace, we propose a representation that is more lightweight.

One of the crease segments will always be known in the geometry shader, as one of the triangle's halfedges. The information stored in a vertex must allow us to compute the direction of the other, connecting crease segment. Therefore, we store the difference of the segment directions, which we call the *crease difference*. This is also the bisector on which one segment's direction must be reflected to get the other's. During preprocessing, we augment all vertex records with a crease difference element, which consists of the non-normalized direction difference vector, and a verification value, which can be computed as the first element of the cross product of the segment directions. If the vertex is not part of a crease halfedge, the verifier is set to 2, and we call the crease difference *invalid*. During rendering, in the geometry shader we can easily tell if the triangle potentially has a crease halfedge — at least two crease differences are valid — and we have the directions of previous and next crease segments simply by adding or subtracting the crease differences to or from the direction of the crease halfedge. In order to discard those triangle halfedges that connect crease vertices but are not crease halfedges themselves, we need to perform two tests. First, the computed adjacent segment direction must be unit length. Second, its cross product with the halfedge direction has to produce the verifier. Otherwise, albeit the halfedge runs between crease outlines, it is not a crease halfedge.

4.1. Preprocessing

In order to compute the crease differences we process all triangles of the mesh. One option is to use the following mesh traversal algorithm (depiced in Figure 3):

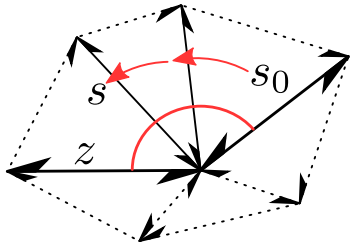

Figure 3: *The search for the crease halfedges.*

First, we initialize all crease difference verifiers to 2. Then, for every triangle of the mesh:

1. If the triangle has a crease halfedge s_0, we process both its vertices. With **w** as the current vertex, and halfedge s being s_0 initially:
   a. If the triangle halfedge z — that shares **w** with s — is a crease halfedge, then the crease difference for **w** is the direction difference between s_0 and z, and we can proceed to the next vertex.
   b. Otherwise, let s be the opposite halfedge of the former z, and repeat step 1.a. for the adjacent triangle which contains halfedge s.

The search definitely stops, because if there is no other crease edge, the computation will find the opposite halfedge of the initial crease halfedge.

Alternatively, if the topology information is not available,

but we can customize adjacency generation, then crease difference computation can be integrated into it. For every vertex, an incoming and an outgoing crease direction has to be maintained. Adjacency generation first finds the topology vertices that belong to the same geometry vertex (typically called *point reps* in this context). Second, all halfedges in the mesh are processed to find matching pairs, for which endpoint point reps are identical. Whenever such a pair of halfedges is found, we can check if their endpoint vertices are different, and therefore they form a crease edge. For crease edges, their direction must be written into the outgoing and incoming crease direction variables of the start and end vertices, respectively. After the adjacency computation is complete, the difference of incoming an outgoing directions can simply be computed.

4.2. Crease edge rendering

We render the mesh with geometry shaders processing its triangles. If two vertices in a triangle have valid crease differences, the halfedge between them is a potential crease halfedge, which must be extruded into a crease outline segment. We reflect the halfedge direction onto the crease differences of the end vertices to get the previous and next segment directions. Then we verify that the halfedge is a crease halfedge as described in Section 4.

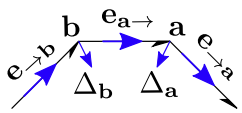

Figure 4: *Nomenclature for crease segment directions.*

Formally (see Figure 4), if $\Delta_\mathbf{a}$ and $\Delta_\mathbf{b}$ are crease differences at vertices $\mathbf{a}$ and $\mathbf{b}$, respectively, then the crease segment directions are:

$$\mathbf{e}_{\rightarrow\mathbf{a}} = \mathbf{a} \hat{-} \mathbf{b},$$

$$\mathbf{e}_{\mathbf{a}\rightarrow} = \mathbf{e}_{\rightarrow\mathbf{a}} + \Delta_\mathbf{a}, \quad \mathbf{e}_{\rightarrow\mathbf{b}} = -\mathbf{e}_{\rightarrow\mathbf{a}} - \Delta_\mathbf{b},$$

where we use $\hat{}$ over an operator to denote normalization of the result.

To render the halfedge segment we create a quad with two vertices at the original halfedge vertices, and two vertices *inset* (see Figure 5). The inset vector must bisect the crease halfedges in screen space, but lie in the plane of the triangle in world space (see Figure 6). A screen-space-bisecting inset direction $\mathbf{d}_\mathbf{a}$ can be found for a vertex $\mathbf{a}$, using the vector to the camera $\mathbf{c}_\mathbf{a}$ and crease segment directions $\mathbf{e}_{\rightarrow\mathbf{a}}$ and $\mathbf{e}_{\mathbf{a}\rightarrow}$ as

$$\mathbf{d}_\mathbf{a} = \mathbf{e}_{\rightarrow\mathbf{a}} \hat{\times} \mathbf{c}_\mathbf{a} \hat{+} \mathbf{e}_{\mathbf{a}\rightarrow} \hat{\times} \mathbf{c}_\mathbf{a},$$

where we use $\hat{}$ over an operator to denote normalization of the result. The cross products compute screen-space crease segment normals, and the sum computes the bisector. The actual world-space inset direction $\mathbf{r}_\mathbf{a}$ is then found as the vector perpendicular to the surface normal $\mathbf{n}_\mathbf{a}$ and in the plane of $\mathbf{c}_\mathbf{a}$ and $\mathbf{d}_\mathbf{a}$:

$$\mathbf{r}_\mathbf{a} = \mathbf{n}_\mathbf{a} \hat{\times} (\mathbf{c}_\mathbf{a} \times \mathbf{d}_\mathbf{a}).$$

The actual quad vertex $\mathbf{x}_\mathbf{a}$ which ensures screen space crease width of k is

$$\mathbf{x}_\mathbf{a} = \mathbf{a} + k\mathbf{r}_\mathbf{a} \frac{|\mathbf{c} - \mathbf{a}|}{\mathbf{r}_\mathbf{a} \cdot \mathbf{e}_{\rightarrow\mathbf{a}}},$$

where the numerator scales the inset by distance and the denominator accounts for the angle between the inset and the edge direction. Such a quad will only cover one side of the crease edge. The other half is rendered when the adjacent triangle is processed. The quad can also be textured in the pixel shader, and its edge can be softened using alpha-blending. The screen space width k is adjustable and the edge segments will be connected seamlessly.

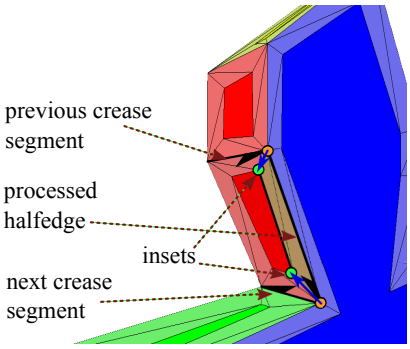

Figure 5: *Crease segment quad construction.*

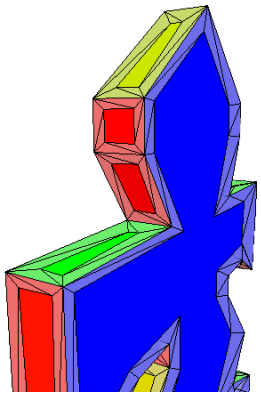

Figure 6: *Crease edge geometry rendered in wireframe.*

4.3. Silhouette rendering

Rendering the silhouette outlines is accomplished by implementing the Hertzmann[4] approach in the geometry shader (see Figure 7). A vertex $\mathbf{a}$ is defined to be front-facing if its normal $\mathbf{n_a}$ points towards the camera, satisfying $\alpha_\mathbf{a} > 0$, where $\alpha_\mathbf{a} = \mathbf{n_a} \cdot \mathbf{c_a}$. A triangle contains a silhouette segment if exactly one or two of its vertices are front-facing. In this case the triangle has two edges intersected by the silhouette. The intersection point $\mathbf{y_{ab}}$ on the edge between vertices $\mathbf{a}$ and $\mathbf{b}$ can be found using linear interpolation.

$$\mathbf{y_{ab}} = \frac{\alpha_\mathbf{a}\mathbf{b} - \alpha_\mathbf{b}\mathbf{a}}{\alpha_\mathbf{a} - \alpha_\mathbf{b}}.$$

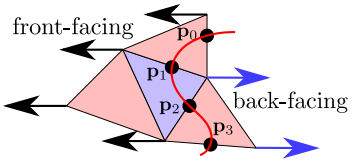

Figure 7: *Creating silhouette edges.*

The same process can be repeated for the adjacent triangles known in the geometry shader, alltogether producing four points on edges. These define a Catmull-Rom spline which can be vectorized into a triangle strip of given width. However, there is no guarantee the curve will not double back in screen space, so it is safer to just generate a single quad for one triangle, but adjusting its endpoints to fit the adjacent segments.

5. Depth testing

As crease segment quads are aligned on the surfaces, they can be subjected to classic, accurate depth testing. However, if surfaces are not flat, and the outlines are wide in world space, the flat outline geometry will intersect the object surface. Therefore, a slight bias should be used, and a smooth fade out is helpful to diminish artifacts. These can be accomplished by implementing a custom depth test in the pixel shader.

Silhouette outline quads cannot be aligned on surfaces. Therefore, a stronger bias is necessary for rendering them, or, in most cases, it is even more robust to test against backface depth.

6. Results and limitations

The algorithm can be easily added to any rendering system. In our test application (see Figures 8, 9, 10 and 11 for screenshots), which used deferred toon shading with depth map shadows, rendering outlines only reduced the frame rate by $3-5\%$, compared to the case without outlines. As the method relies on the geometry shader, combination with other techniques also exploiting it might not be trivial. Also, we require exact crease differences, which might not be preserved in case of mesh deformation or character animation. These cases would require less lightweight crease edge indicators. Crease edges on contours appear as half width crease edges, as the back-facing neighbour does not emit a crease edge quad. A robust way to eliminate this nuance is left for future research.

Figure 8: *Pictures from a real time application.*

Figure 9: *Closer view from the same application.*

7. Acknowledgements

This work was supported by OTKA K-719922 and 101527, as well as the project of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

Figure 10: *Wider edges.*

Figure 11: *Rook with thinner edges.*

References

1. P. Bénard, F. Cole, A. Golovinskiy, and A. Finkelstein. Self-similar texture for coherent line stylization. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, pages 91–97. ACM, 2010.
2. D. DeCarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pages 15–145. ACM, 2004.
3. D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 848–855. ACM, 2003.
4. A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.
5. T. Judd, F. Durand, and E. Adelson. Apparent ridges for line drawing. In *ACM Transactions on Graphics (TOG)*, volume 26, page 19. ACM, 2007.
6. R.D. Kalnins, L. Markosian, B.J. Meier, M.A. Kowalski, J.C. Lee, P.L. Davidson, M. Webb, J.F. Hughes, and A. Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Transactions on Graphics*, 21(3):755–762, 2002.
7. L. Markosian, M.A. Kowalski, D. Goldstein, S.J. Trychin, J.F. Hughes, and L.D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420. ACM Press/Addison-Wesley Publishing Co., 1997.
8. M. Nienhaus and J. Doellner. Edge-enhancement-an algorithm for real-time non-photorealistic rendering. *Journal of WSCG*, 11(2), 2003.
9. R. Raskar. Hardware support for non-photorealistic rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 41–47. ACM, 2001.
10. J. Shin. A stylised cartoon renderer for toon shading of 3d character models. Master's thesis, University of Canterbury, UK, 2006.
11. László Szirmay-Kalos, György. Antal, and Ferenc Csonka. *Háromdimenziós grafika, animáció és játékfejlesztés*. ComputerBooks, Budapest, 2003.
12. L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. http://www.iit.bme.hu/~szirmay.
13. T. Umenhoffer. Graphics techniques in the Turing game project. In L. Szirmay-Kalos and G. Renner, editors, *V. Magyar Számítógépes Grafika és Geometria Konferencia*, pages 33–39, Budapest, 2010.
14. T. Umenhoffer, M. Magdics, and K. Zsolnai. Procedural generation of hand-drawn like line art. In Z. Kató and K. Palágyi, editors, *Képfeldolgozók és Alakfelismerők VIII. Konferenciája*, pages 502–514, Szeged, 2011.
15. T. Umenhoffer, L. Szécsi, and L. Szirmay-Kalos. Hatching for motion picture production. In *Computer Graphics Forum*, volume 30, pages 533–542, 2011.