The edge buffer: A data structure for easy silhouette rendering

John W. Buchanan*
Electronic Arts (Canada), Inc.
4330 Sanderson Way,
Electronic Arts Centre
Burnaby, B.C. V5G 4X1
juancho@ea.com

Mario C. Sousa†
Department of Computing Science
University of Alberta
Edmonton, AB
Canada, T6G 2H1
mario@cs.ualberta.ca

Abstract

Cartoon rendering of 3d models relies heavily on accent lines to portray the important features of a model. In addition to this it has been shown that highlighting silhouette edges can significantly enhance the comprehension of technical images. In this paper we introduce *the edge buffer*. This data structure allows us to highlight silhouette edges, boundary edges, and artist defined edges. This edge buffer is used *a-priori* to define which edges are to be rendered when visible. The edge buffer is also updated each time the object is rendered so that silhouette edges can be drawn. We discuss the difference between silhouette edges and boundary edges and show how the edge buffer allows both types of edges can be drawn. The use of the edge buffer only requires that a front/back computation be available and that the object being rendered be represented in a vertex/polygon representation

Keywords: NPR, Edge highlighting, silhouette rendering

1 Introduction and previous work

Rendering objects in a cartoon style requires the use of edges to highlight different areas of the object. Many times the silhouette edges of the object are also drawn to clearly delineate the object. The method for detecting silhouette edges is fairly straight forward. The basic approach is to find edges that are shared between front facing and back facing polygons. A front facing polygon is defined as a polygon whose normal points towards the viewer[1].

If the viewing vector (V) is defined as a vector from the eye to the viewing plane the a front facing polygon is detected by looking at the sign of the dot product between N(the polygon normal) and V. If the dot product $N \cdot V > 0$ then the polygon is front facing, if $N \cdot V < 0$ then the polygon is back facing and if $N \cdot V = 0$ then the polygon is perpendicular to the viewing direction as illustrated in Figure 1

*Part of this work was done while the first author was employed as a research scientist at Radical Entertainment in Vancouver Canada.

†This research was funded in by NSERC.

[1]This assumes the fairly standard definition of a polygon normal where the normal points away from the polygon thus defining the outside direction for the polygon.

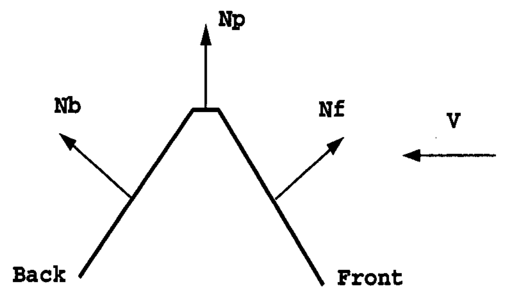

Figure 1: The front/back facing property of a polygon can be calculated from the sign of the dot product $N \cdot V$

If this test is done after the objects and their normals have been transformed into camera space the test reduces to a sign test of the z component of the projected normal since the projected view vector is simply $(0, 0, 1)$.

The test for front/back facing can be done either in model space or in camera space. In some situations there is no need to project the object normals into camera space so the test should be done in object space.

Image space algorithms

The use of silhouette edges has been discussed by several people, in particular Saito and Takahashi [6] showed how silhouette edges can be used to enhance the display of images. The silhouette edges can be rendered by highlighting the discontinuities in the z-buffer derivative. In a similar fashion creases or folds in the image can be rendered by highlighting discontinuities in the second derivative of the z-buffer. Rosignac [5] used the z-buffer to render silhouette edges by first rendering the object in white then rendering the scene in thick wire-frame mode. The polygons in the second pass are pushed away in the z direction. This results in a silhouette display of the objects. In [4] Raskar and Cohen presented three ways to add silhouette edges using front/back polygon computation. They used z-buffer equality, z-buffer offset, and polygon growing. In each of these methods the front facing polygons are rendered first with z-buffer write enabled. The back facing polygons are then rendered using one of three methods to produce the silhouette edges. This is an extension of the work presented by Rossignac[5].

Object space methods

Markosian *et. al.* [3] presented a method for quickly approximating the display of an object by rendering the silhouette of the object.

Their work differs from ours in that they do not want to process all of the polygons and thus focus on approximating the silhouette. In their paper they presented a probabilistic silhouette rendering technique that is based on Appel's [1] work on hidden surface removal. Gooch *et. al.* [2] presented a cartoon illumination model that used silhouette rendering as one of its parameters. In particular they presented two methods for highlighting the silhouettes, the first used a environment map with the circumference of the texture being set to black. This results in a darkening of the object where the normals are perpendicular to the viewing direction, thus a silhouette is added to the object image. The second method that they presented assumes that all of the polygons are being processed. The edges of the model are stored as lines on the surface of the Gauss sphere. Any line that intersects the projection plane corresponds to a silhouette edge. Their method works well for orthographic projection but is difficult to generalize to a perspective framework. If processed correctly the silhouette edges can be found accurately without touching all of the polygons.

The context in which we wanted to add silhouette edges to objects was in relatively low-end consumer PC applications[2]. On these machines reading from the z-buffer is relatively expensive or impossible if there is no z-buffer[3]. The second criterium is that we are always going to process all of the polygons that define the object. Thus we are not concerned with methods that require preprocessing to define the silhouette edges. The third criterium was that artists should be able to define which edges in the model must be displayed.

2 The edge buffer

In this paper we introduce the edge buffer silhouette technique. Our technique is based on two assumptions, the first that front/back facing computations can or are being performed for the polygons. The second assumption is that the polygonal mesh is being stored using an indexed vertex mesh representation. Our technique does not require the submission of the back facing polygons, it handles open objects, and easily incorporates artist defined edges and other computed edges. The edge buffer requires little additional storage[4] We introduce the edge buffer in steps. First we show how the idea was formulated to work for the simple case of closed polyhedral objects, we then show how the technique easily extends to boundary edges on open objects, and finally we show how this data structure can easily be extended to store artist defined edges.

2.1 Closed polyhedral objects

The edge buffer stores a minimum of 2 bits per edge in the model. The bits correspond to a front facing flag (F) and a back facing flag (B). In our implementation we have stored these flags in a hash table. The hash table is accessed using the lowest valued vertex-index of the edge. The second vertex index is stored as a field in the edge buffer entry. Thus for the object in Figure 2 the complete edge buffer has the following entries.

[2]*In grant terms:*Highly interactive 3d environments, or in *real terms*: Video Games

[3]The Sony Playstation^{TM} has no hardware z-buffer.

[4]In our implementation we have used 32 bits per edge. Five bits for the edge information that we describe, and 24 bits for the index of the second vertex.

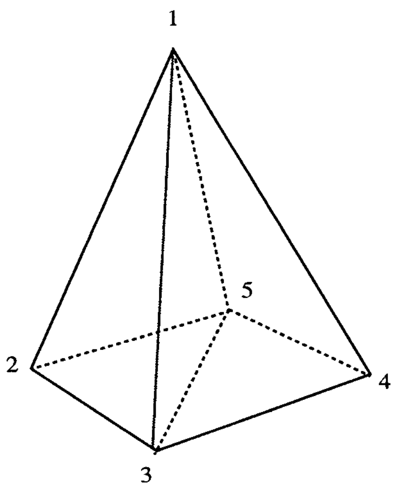

Figure 2: A closed polyhedral object defined by 5 vertices and 6 polygons.

Vertex	*VFB*	*VFB*	*VFB*	*VFB*
1	200	300	400	500
2	300	500	x00	x00
3	400	500	x00	x00
4	500	x00	x00	x00
5	x00	x00	x00	x00

In our implementation we stored this as a static array that is part of the object definition. Notice that the last row of the edge buffer is always empty since any edge that contains the last vertex will be represented earlier in the table. This initialization is performed as a pre-rendering setup for each frame.

As the polygons are being passed to the rendering engine we update the edge buffer on a per-polygon basis. The FB flags for the edges in the current polygon are updated depending on whether the polygon is front facing or back facing. For a front facing polygon we XOR a 1 value into the F field and for a back facing polygon we XOR a 1 into the B field. The polygon defined by the vertices 1,2, and 3 (P[1,2,3]) is a front facing polygon. After P[1,2,3] is processed the edge buffer has the following values (new values are in bold font):

Vertex	*VFB*	*VFB*	*VFB*	*VFB*
1	**210**	**310**	400	500
2	**310**	500	x00	x00
3	400	500	x00	x00
4	500	x00	x00	x00
5	x00	x00	x00	x00

After polygon P[1,3,4] is processed the edge buffer contains the following values:

Vertex	*VFB*	*VFB*	*VFB*	*VFB*
1	210	**300**	**410**	500
2	310	500	x00	x00
3	**410**	500	x00	x00
4	500	x00	x00	x00
5	x00	x00	x00	x00

Notice that the entry for the edge $(1,3)$ now has the value 00 even though the edge has been visited twice. The F bit has been set twice with the XOR operation and thus now has the value 0. Now consider the effect of processing polygon P[3,4,5].

Vertex	*VFB*	*VFB*	*VFB*	*VFB*
1	210	300	410	500
2	310	500	x00	x00
3	**411**	**501**	x00	x00
4	**501**	x00	x00	x00
5	x00	x00	x00	x00

Notice that the entry for the edge $(3,4)$ now has the value 11 for FB. This indicates that the edge $(3,4)$ is shared by a front and a back facing polygon. The remaining polygons are processed similarly finally producing the following edge buffer table:

Vertex	*VFB*	*VFB*	*VFB*	*VFB*
1	**211**	**300**	**411**	**500**
2	**311**	**500**	x00	x00
3	**411**	**500**	x00	x00
4	**500**	x00	x00	x00
5	x00	x00	x00	x00

Thus the edges that define the silhouette of this object are $\{(1,2),(2,3),(3,4),(1,4)\}$. The rendering of the silhouette can now be done in a second pass using the edges defined by this process. Since the vertices for these edges are all part of a front facing polygon we know that they will be transformed to screen space. In our implementation we simply call a line drawing routine with the screen projections of the appropriate vertices.

2.2 Open polyhedral objects and artist edges.

One of the advantages of closed polyhedral objects is that back facing polygons need not be submitted to the rendering pipeline. Unfortunately if the object is an open polyhedral object all of its polygons need to be submitted to the rendering pipeline, this is illustrated in Figure 3. After processing the polygons P[1,2,3,4] and P[3,7,6,4] the edges $(1,4)$,$(4,6)$ will have their front facing flags set. Similarly after polygons P[1,2,8,5] and P[8,7,6,5] have been processed the edges $(1,5)$ and $(5,6)$ will have their back facing flags set. After all the polygons in this open box have been processed the edges $(1,2)$, $(2,3)$, $(3,7)$, and $(6,7)$ will have both the F and B flags set. Following our initial interpretation of the edge buffer table would result in the rendering of edges with FB = '11'. Clearly, as shown in Figure 4, this is not a correct rendering of the silhouette of this open box. This problem can be trivially addressed by rendering all edges whose FB flags are not identical to '00'. When we do this we have a better rendering of the box (Figure 5). In particular the edges $(1,4)$ and $(4,6)$ would not have been drawn had this box been a closed object.

There is still a problem with this rendering of the box. A traditional illustration of the box would completely render the edge $(4,3)$ and would partially render the edge $(5,8)$ (Figure 6). Unfortunately unless we want to use the second derivate of the z-buffer there is no easy way to automatically detect and highlight these edges.

Given a model that is to be rendered there may always be an edge or set of edges in this object that must always be rendered. In order to address this almost arbitrary set of edges we decided to allow the artist to control which edges are crucial in the rendering of the object.

In order to allow an artist to a-priori define which edges we added an additional bit (A) to the edge buffer field. The combination of the artist flag with the FB flags allows us to decide which edges to render. In the case of an open object we simply render any edges

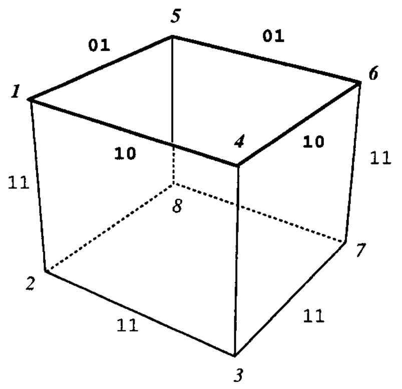

Figure 3: A box with the top polygon missing. This results in edges that will never be defined as silhouette edges. In the context of this paper we refer to the top edges of this object as boundary edges.

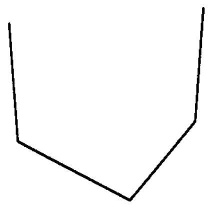
Figure 4: Using the edge buffer results in an incomplete rendering of the silhouette of the open box. This is because the boundary edges do not end up with a FB = 11 labeling.

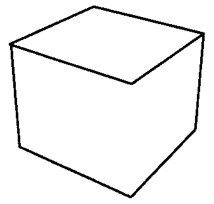
Figure 5: By rendering the edges with $FB \neq 00$ we correctly render the boundary edges for front and back facing polygons.

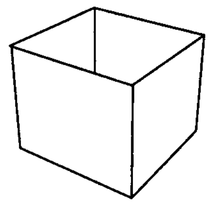
Figure 6: By allowing the artist to designate which edges are to be always drawn we get a better rendering of the open cube.

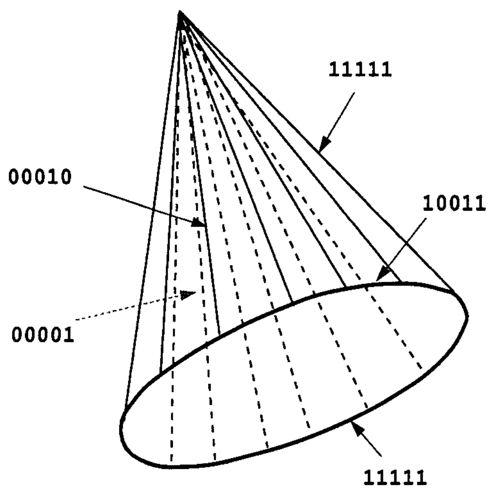

Figure 7: A closed polyhedral object showing the use of the A bit. The edges around the base are allways rendered thus highlighting the essential shape of the object. In the demo program the artist edges are rendered in red when they are not silhouette edges. The cone is labeled with the AFBF^{a}B^a values that are evaluated for this view.

whose AFB flags are not 000. In the case of a closed object we may want to ignore back facing artist edges. Unfortunately, a back facing artist edge will have FB = 00 thus we are not able to determine whether this edge should be drawn or not. The addition of two additional bits with a different processing/update method allows us to drop back facing artist edges. The two additional flags that we add are F^a and B^a for front absolute and back absolute. Like the related F and B flags the initial value for these flags is 0. When a front facing edge is being updated we simply OR the current value of F^a with 1, similarly for B^a. Thus for a front facing artist edge the resulting flags will be AFBF^{a}B^a = '10010' and for a back facing artist edge the flags will be AFBF^{a}B^a = '10001'. The addition of F^a and B^a allows us to easily determine when there is a back facing artist edge. If we are processing a closed object we can simply ignore that edge. An example of the use of artist defined edges can be seen in Figure 7, additionally the example program that can be found at http://www.cs.ualberta.ca/~juancho/edge.html contains an example of this technique. Three objects are rendered, a sphere, a cone, and a cube. The sphere contains no artist edges, the cone contains artists edges around its base, and all of the edges of the cube are artist edges. In this example the artist edges are drawn red when they are not silhouette edges.

3 Conclusion

In this paper we presented the edge buffer. This fairly compact data structure allows us to process silhouette edges, boundary edges and artist edges. The processing of these edges is done in a consistent framework. This work is applicable in situations where the object is stored in a indexed vertex polygon mesh and all of the polygons are going to be processed. The overhead of this technique is minimal in storage cost and requires two binary operations per edge. This approach to silhouette rendering is compatible with most graphics pipelines that the authors are familiar with. The increased computation cost is negligible adding less than 1% rendering time in most situations where we looked at using this method.

The main requirements of this technique are that the objects be stored as a indexed polygon mesh and that the front and back facing computation be done. Since this computation is performed in most modern pipelines we feel that these requirements are minimal and should not deter the implementation of this technique in a variety of applications.

4 Acknowledgements

This work was the result of many random interactions. In particular we would like to acknowledge: Paul Lalonde for putting up with us during the time we worked on this; The members of the Graphics group at the University of Alberta for sitting through a couple of presentations; Oleg Veryovka (Mainframe) and Lisa Streit (UBC) for discussions on this and many other topics (not necessarily related).

References

[1] A. Appel, F.J. Rohlf, and A.J. Stein. The haloed line effect for hidden line elimination. volume 13, pages 151–157, August 1979.

[2] B. Gooch, P.J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *Proc. of 1999 ACM Symposium on Interactive 3D Graphics*, pages x–x, April 1999.

[3] L. Markosian, M.A. Kowalski, S.J. Trychin, L.D. Bourdev, D. Goldstein, and J.F. Hughes. Real-time nonphotorealistic rendering. In *Proc. of SIGGRAPH '97*, pages 415–420, August 1997.

[4] R. Raskar and M. Cohen. Image precision silhouette edges. In *Proc. of 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

[5] J. Rossignac and M. van Emmerik. Hidden contours on a framebuffer. In *Proc. of the 7th Workshop on Computer Graphics Hardware, Eurographics*, September 1992.

[6] T. Saito and T. Takahashi. Comprehensible rendering of 3d shapes. In *Proc. of SIGGRAPH '90*, pages 197–206, August 1990.