Robust, Geometry-Independent Shadow Volumes

Graham Aldridge, Computer Science Department
University of Canterbury, New Zealand
graham.aldridge@canterbury.ac.nz

Eric Woods, HIT Lab NZ
University of Canterbury, New Zealand
eric.woods@hitlabnz.org

Abstract

A novel algorithm is presented that overcomes limitations of existing shadow volume algorithms when dealing with non-manifold geometry and space partitioning. This simple algorithm will generate a shadow volume for any set or subset of polygons, with no restrictions placed on the complexity of the geometric form.

Keywords: Shadow volumes, space partitioning

1 Introduction

Rendering shadows is becoming more popular in real-time computer graphics and shadow volumes are a commonly used technique to achieve this effect. The algorithm used to generate shadow volumes most commonly used today was presented by Crow (CROW 1977), and later generalized by Bergeron (BERGERON 1986). Bergeron presented a general algorithm to generate a shadow volume from a mesh of polygons. However, a major constraint was placed on the mesh: that every edge between polygons must be connected to exactly two polygons, and these polygons must be wound in the same direction. It is possible to overcome these constraints on a case-by-case basis, however, as the complexity of geometry increases, inevitably so do the number of cases. The algorithm also has problems with space partitioning, as generating a shadow volume from a subset of polygons requires further modifications to the algorithm.

This paper presents a redesign of this existing algorithm, with no known limitations on geometry form or complexity. This paper will not describe methods for rendering shadows using shadow volumes. Crow/Bergeron's algorithm will be described and it's limitations outlines, other papers describe how to render a shadow volume using the stencil buffer (EVERITT, KILGARD 2002) Sample code using OpenGL will be included for both algorithms.

2 Background, Crow/Bergeron's algorithm

Shadow volumes are generated by first determining the *silhouette loop* of a polygonal mesh. The silhouette follows the edges between certain polygons. Each edge in the silhouette is connected to two polygons; one of these polygons is facing the light source and the other polygon is not. The silhouette edges are guaranteed to form a silhouette 'loop' if the mesh has no holes, and every edge is connected to exactly two polygons (mathematically, the mesh is two-manifold). The direction of each edge of the silhouette is consistent throughout the entire loop, and will also be consistent with the winding of the lit polygons connected to these edges.

With the silhouette determined, the shadow volume can be constructed. Depending on the stencil buffer technique used, the shadow volume may be required to be completely 'air tight' with no holes. If this is the case, to create a volume, *either* the polygons facing the light, or the polygons not facing the light will form the front *shadow volume cap*.

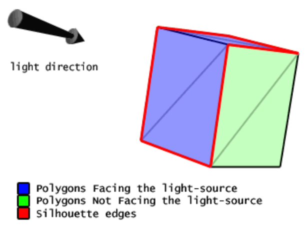
light direction
🟦 Polygons Facing the light-source
🟩 Polygons Not Facing the light-source
🟥 Silhouette edges

Figure 1. A Silhouette edge loop

Either technique has advantages and disadvantages when used with polygonal meshes. Using the polygons facing the light as the front volume cap will result in a larger shadow volume, requiring more rendering fill-rate, and also will produce shadowing artifacts when the light source moves inside the mesh. However, using the polygons not facing the light will require more processing as these faces can otherwise be ignored, and their winding is reversed.

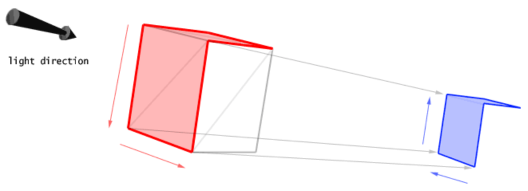

Figure 2. Front and rear shadow volume caps

To begin creating a complete shadow volume, the polygons for the front volume cap are projected away from the light to form the rear volume cap. The projection distance must be enough for the individual polygons to be outside the radius of the light source for point lights, for directional lights the polygons are often projected to infinity. The winding of the polygons on the rear volume cap is reversed.

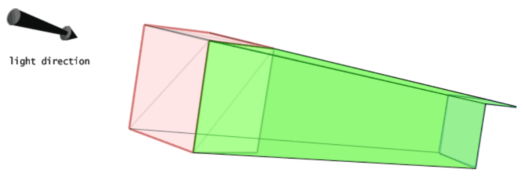

Figure 3: A complete shadow volume

To complete the volume, the silhouette is also projected in a similar way to the rear volume cap. Each segment of the silhouette loop consists of four vertices; two form the original segment of the loop, and two for the projected segment. A single quad can be created for each segment, between the four vertices. This completes the shadow volume.

2.1 C++ Example code

For simplicity, the following example uses triangles. A constraint is made here: because the winding of the polygon on the shadow volume cap needs to be known when computing a silhouette edge, it is required here that each `Edge` object's `vertices` are ordered the same as the winding of `faces[0]`. Therefore, the edge must also be directed opposite to the winding of `faces[1]`. Both `faces` are valid and are not the same face. As a result of this constraint, each `Edge` object is unique.

Data structures:

```
class Triangle
{
public:
     Vertex* vertices[3];
     bool facesLight(Light light);
};

class Edge
{
public:
     Vertex * vertices[2];
     Triangle * faces[2];
};
```

```
class Vertex
{
public:
     float vertex[3];
     float* project(Light light);
};
```

Crow/Bergeron's algorithm using OpenGL, with polygons facing the light forming the volume cap:

```
//render the volume caps
glBegin(GL_TRIANGLES);
for (int t=0; t<triangles.size(); t++)
{
     if (triangles[t].facesLight(light))
     {
          //render the front volume cap
          for (int v=0; v<3; v++)
               glVertex3fv(triangles[t].vertices[v]);
               //render the projected rear volume cap
          for (int v=2; v>=0; v--)
               glVertex3fv(triangles[t].vertices[v]->project(light));
     }
}
glEnd();
//render the projected silhouette
glBegin(GL_QUADS);
for (int e=0; e<edges.size(); e++)
{
     if (edges[e].faces[0]->facesLight(light) !=
          edges[e].faces[1]->facesLight(light))
     {
          //render the projected silhouette edges
          if (edges[e].faces[0]->facesLight(light))
          {
               glVertex3fv(edges[e].vertices[0]);
               glVertex3fv(edges[e].vertices[1]);
               glVertex3fv(edges[e].vertices[1]->project(light));
               glVertex3fv(edges[e].vertices[0]->project(light));
          }else{
               glVertex3fv(edges[e].vertices[1]);
               glVertex3fv(edges[e].vertices[0]);
               glVertex3fv(edges[e].vertices[0]->project(light));
               glVertex3fv(edges[e].vertices[1]->project(light));
          }
     }
}
glEnd();
```

2.2 Disadvantages and limitations

The algorithm is not suited for large meshes with smaller light sources (an example may be a terrain renderer). These cases require space partitioning of the `Polygon` data for efficient operation. This presents a new problem, as the `Edge` data also requires partitioning, unless pointers are added to the `Polygon` class that point to each `Edge` object. Either approach is not optimal due to the memory overhead. Furthermore, because the `Edge` object determines if it is part of the silhouette loop by comparing *both* polygons it is connected to, it is a likely that some of these polygons will not be part of the set of polygons returned by the space partitioner. For a robust volume to be generated, these polygons must not qualify as part of the shadow volume cap.

However, most of the problems inherent in this algorithm relate to building the `Edge` data structures. This problem can be simplified by extending the algorithm to cope with edges that are only connected to a single polygon. This removes the constraint that each edge be unique, but will lead to an extreme case where every edge is associated with only one polygon. In this case rendering will become grossly inefficient, as each individual polygon will produce a complete shadow volume, leading to a significant increase in the number of silhouette edges, and therefore a large increase in the number of projected quads.

Further problems will arise when *more* than two polygons share an edge. This is a rare case on most small meshes (such as a character model), but is a very common problem in larger meshes (such as world geometry) or on meshes with coplanar polygons. This problem can be extremely difficult to counter and is usually dealt with by building world geometry using many smaller meshes, or simply requiring that the geometry is two-manifold. Even with these specific cases dealt with, more will arise. For example, two polygons that share an edge but are wound opposite to each other will not work in this model. The more complex the mesh, the more special cases arise. It is not uncommon, in a large mesh, to have edges connected to three, four, five or more polygons. Furthermore, any attempt to counter these cases will not lead to an optimal solution, as there will undoubtedly be cases where unneeded quads are used to form the shadow volume.

Therefore, it is a logical assumption that to be able to generate a robust shadow volume for any geometry form, the core algorithm needs to be changed. The changes proposed are minor, yet significant, and have the desired effect, while maintaining a similar memory overhead to the existing algorithm.

2.3 Building a new algorithm

A study of the problems in the algorithm above suggest that the path to a solution lies in increasing the number of `Polygons` an `Edge` can connect to. This, however, will not work. The existing algorithm determines if an edge is part of the silhouette by a different light state of the two polygons it is connected to. There is no immediate logical extension to this test when comparing three or more polygons. The reason for this lies in a constraint of the existing algorithm, that `faces[0]`'s winding is the same as the direction of the edge, and the opposite of `faces[1]`. This requirement cannot be achieved with a variable number of faces connected to an edge. Even if an algorithm were developed to solve this problem, it would still fail under certain circumstances that are key to generating robust shadow volumes from non two-manifold geometry.

The solution does not appear to lie in increasing the capabilities of the `Edge` object, in fact, further investigation will show the opposite, where the `Edge` object is simplified.

The core of the problem therefore lies in the winding constraint of the existing algorithm.

In the brute force case where every edge is only connected to a single polygon, every edge will be directed the same as the winding of the polygon it is connected to. All projected silhouette quads will also be wound in the same direction. In the normal case, where edges are connected to two polygons, the winding of the projected silhouette quad will depend on which connected polygon is in the light, and therefore, the winding of that polygon.

This provides a starting point for an algorithm that solves some of the existing algorithms problems. However, problems with space partitioning and edges connected to more than two polygons will still exist.

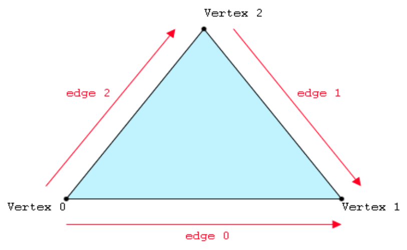

Figure 4. Different directions of edges and winding on a simple polygon

The solution to the space partitioning problem of adding `Edge` pointers to each polygon is extended. For the solution to be optimal the pointers from the `Edge` to the `Polygons` are removed. For this to work the polygon needs to know how the edges it connects to are directed, be it in the same direction as the polygons winding, or opposite. This can either be stored or computed at runtime. If this data is stored, the `Edge` object no longer requires the `vertices` pointers. **`Figure 4`** shows an example polygon where edge 0 is directed the same as the polygons winding, where edge 1 and edge 2 are directed opposite.

In the existing algorithm, when processing a polygon as part of the shadow volume cap, the polygon would only be rendered and projected. Because the polygon now also determines the silhouette edges, it must tag each edge it is connected to as being part of the silhouette edge. Furthermore, as the direction of the edge may be the same as the winding of the polygon or opposite, the edge is either tagged as having a silhouette edge that is wound the same as the edge, or opposite. When processing the edges, if an edge is tagged twice, once wound the same and once opposite, then the edge is *not* a part of the silhouette edge loop, as both polygons it is connected to are part of the shadow volume cap. Therefore edges marked only once become part of the silhouette edge loop. However, there is still a limitation of two polygons per edge. This can be dealt with easily through an extension of the algorithm.

The case where this algorithm still fails is where there are three or more polygons connected to an edge, and two or more are part of the shadow volume cap, and are wound the same. In this case the edge will be tagged *twice* or more for the same winding. In this situation, it may be required for a complete shadow volume that the projected silhouette quad is rendered *more than once*. This is a key requirement for generating robust shadow volumes from any geometry when using stencil buffering.

There is a way to allow this, while simplifying the algorithm. Instead of tagging each edge separately dependent on the winding of the polygon that tagged it, use a counter that is incremented or decremented depending on the polygons winding compared to the edge. After generating the shadow volume cap, if an edge has a value of zero for this counter, it is not a silhouette edge, a value of one will produce a silhouette quad in the same direction as the edge, while a value of negative one produces a quad directed opposite. Values above one or below negative one simply produce more than one quad. This solves the problems with the existing algorithm, while using a similar amount of memory and producing an optimal result using a minimum of quads.

2.4 Summary of the algorithm

Every polygon in a mesh is connected to a number of vertices, and an equal number of edges. These edges may run in the same direction as the polygons winding, or not. This Boolean *direction state* is either stored in the polygon, or computed at run time. Each edge connects between two vertices. The edge need not store these vertices if the polygons connecting to it store its direction state. Edges need not be unique, but for an optimal shadow volume they must be. Each Edge stores a signed counter value, initialised to zero.

To create a volume, either the polygons facing the light, or the polygons not facing the light will form the front and projected rear volume caps. The winding of the polygons projected for the rear volume cap is reversed. When each polygon is processed, all the polygons edges will be processed. If an edge's direction state is the same as the winding of the polygon, the counter for that edge will be incremented. If the direction state is opposite to the winding of the polygon, the counter will be decremented.

Once complete, all edges will have their counter tested. While the counter is positive, a projected silhouette quad will be rendered with the winding the same as the edge's direction. The counter is decremented. While the counter is negative, a projected silhouette quad will be rendered with the winding opposite to the edge's direction. The counter is incremented. Therefore edges with a counter state of zero will be ignored.

2.5 C++ Example code for the new algorithm

For simplicity, the following example uses triangles. Direction state of an edge is computed at run time.

Data structures:

```
class Triangle
{
public:
     Vertex* vertices[3];
     Edge * edges[3];

     bool facesLight(Light light);
};

class Edge
{
public:
     Vertex * vertices[2];
     char counter;
};
```

New Algorithm using OpenGL, with polygons facing the light forming the volume cap:

```
//render the volume caps
glBegin(GL_TRIANGLES);
for (int t=0; t<triangles.size(); t++)
{
     if (triangles[t].facesLight(light))
     {
          //render the front volume cap
          for (int v=0; v<3; v++)
               glVertex3fv(triangles[t].vertices[v]);
               //render the projected rear volume cap
          for (int v=2; v>=0; v--)
               glVertex3fv(triangles[t].vertices[v]->project(light));

          for (int e=0; e<3; e++)
          {
               if (triangles[t].edges[e]->vertices[0]==
                    triangles[t].vertices[e])
                    //edge is directed the same as tirangle winding
                    triangles[t].edges[e]->count++;
               else
                    triangles[t].edges[e]->count--;
          }
     }
}
glEnd();
//render the projected silhouette
glBegin(GL_QUADS);
for (int t=0; t<triangles.size(); t++)
{
     for (int e=0; e<3; e++)
     {
          while (triangles[t].edges[e]->count>0)
          {
               glVertex3fv(triangles[t].edges[e]->vertices[0]);
               glVertex3fv(triangles[t].edges[e]->vertices[1]);
               glVertex3fv(triangles[t].edges[e]->vertices[1]->project(light));
               glVertex3fv(triangles[t].edges[e]->vertices[0]->project(light));
               triangles[t].edges[e]->count--;
          }
          while (triangles[t].edges[e]->count<0)
          {
               glVertex3fv(triangles[t].edges[e]->vertices[1]);
               glVertex3fv(triangles[t].edges[e]->vertices[0]);
               glVertex3fv(triangles[t].edges[e]->vertices[0]->project(light));
               glVertex3fv(triangles[t].edges[e]->vertices[1]->project(light));
               triangles[t].edges[e]->count++;
          }
     }
}
glEnd();
```

2.6 Performance

`Figure 5` shows a complex 3D scene rendered in real-time using shadow volumes. There are 262 lights in the scene. On mid-range 3D hardware (ATI Radeon 9500 series), using the new algorithm, the scene will render at 4.2 frames per second with a polygon count of 2,405,000 (10.1 million polygons / second.) The edge count is 385,077 (10,654 connect to three or more polygons.) Using the brute force algorithm, where every edge connects to one triangle, the scene renders at 2.3 frames per second with a polygon count of 4,440,000 (10.2 million polygons / second) and 763,431 edges.
Before discovery of the new algorithm, approximately 6 months was spent attempting to modify the Crow/Bergeron algorithm to produce optimal, robust shadows for this scene without resorting to brute-force rendering. These attempts were unsuccessful; hence performance data for this algorithm cannot be provided.

2.7 Future work

To complete the algorithm a simple, efficient method for reducing triangles edges to a unique set is required. Experiments using quick-sort to achieve this have so far proved extremely promising.

3 Conclusion

This paper has presented a simple algorithm that solves many problems associated with existing algorithms for computing shadow volumes for stencil buffer shadow rendering. As long as the requirements of the algorithm are adhered to, it will produce a shadow volume with no rendering artifacts, no matter the complexity of the geometry used to construct the volume. Furthermore, the algorithm is significantly easier to use with space partitioning algorithms as only a set of polygons are required to produce a shadow volume, with no need for a set of edges. With careful programming, the algorithm will also use less memory than existing shadow volume algorithms. The algorithm will work for the optimal cases where every edge is unique, or less optimal cases where duplicates exist. There is no need for any case-specific programming with this algorithm; it will always produce a correct solution.

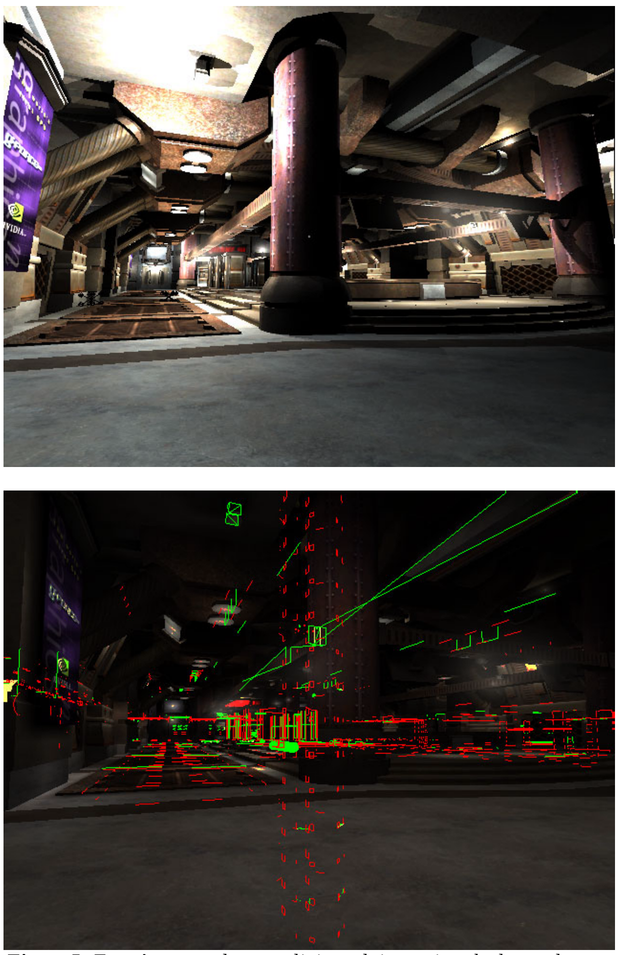

Figure 5. *Top:* An example scene lit in real time using shadow volumes. *Bottom:* Edges connected to three or more polygons are highlighted. Edges are connected to as many as twelve polygons in this scene.

References

EVERITT, CASS AND KILGARD, MARK J. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering, *http://developer.nvidia.com/object/robust_shadow_volumes.html* March, 2002

BERGERON, PHILIPPE. A General Version of Crow's Shadow Volumes, *IEEE Computer Graphics and Applications*. September, 1986, 17-28.

CROW, FRANKLIN. Shadow Algorithms for Computer Graphics, *SIGGRAPH Proceedings* July 1977, vol 11, no. 2, pp. 242-248.