

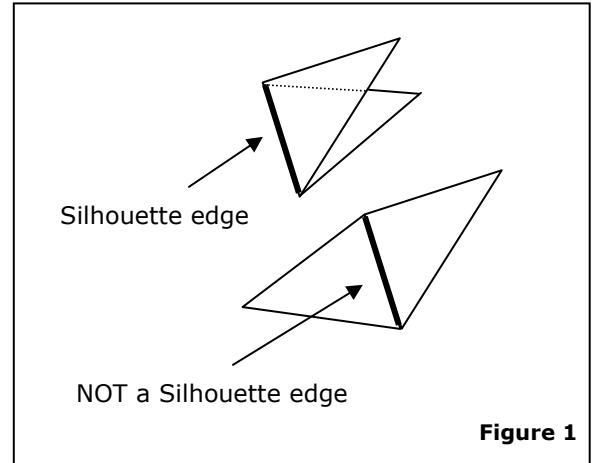
# Silhouette Edge Detection Algorithms for use with 3D Models

**David A. Hostetler**

Graphics Algorithms and 3D Technologies (G3D)  
Intel Architecture Labs (IAL)

## Introduction

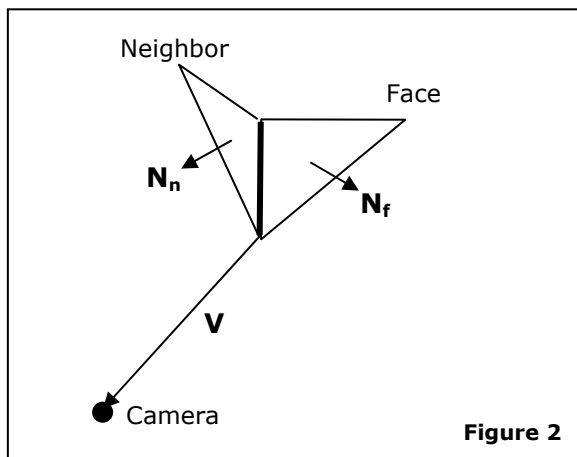
In computer graphics, a silhouette refers to the outline of an object. Since the objects with which we're concerned are three-dimensional models composed of connected triangles, we define a silhouette edge as any edge of any triangle in the model which, when the model is projected onto a two-dimensional plane, forms a portion of the outline of the model. It is important to note that an object's outline is dependent upon the angle at which it is viewed, i.e. the perspective of the viewer or camera. Thus, silhouette edges are valid only so long as two conditions are met. First, the orientation of the projection plane does not change, and second, the model's orientation and composition do not change. In other words, if the camera is moving, or the model is moving, animating, tessellating, etc. then the set of silhouette edges must be redetected after each change.



A more procedural description of a silhouette edge, and one that will govern the discussion of the different algorithms is the following. An edge is formed by two adjacent triangles. These triangles have two important characteristics. The first is the angle between them and the second is their orientation relative to a given perspective, or view vector. These two characteristics combine to determine whether or not the given edge is perceived as part of an outline or silhouette. Basically, if one of the triangles of the pair faces towards the viewer and the other triangle of the pair does not, then their common edge is a silhouette edge. A triangle is 'facing towards the viewer' if the angle between the triangle's face normal and the vector from the triangle's center to the camera location is less than  $90^\circ$  (Figure 1). Throughout the discussions, *view vector* is used to refer to the vector between the camera location and a single specific vertex in the scene. In contrast, *camera vector* is used to identify the center axis of the current view frustum, i.e. the specific direction in which the camera is oriented.

## The Contenders:

### Algorithm A – Original:



This algorithm is perhaps the most intuitive. The steps are as follows (refer to Figure 2):

For each frame:

For each triangle in the mesh:

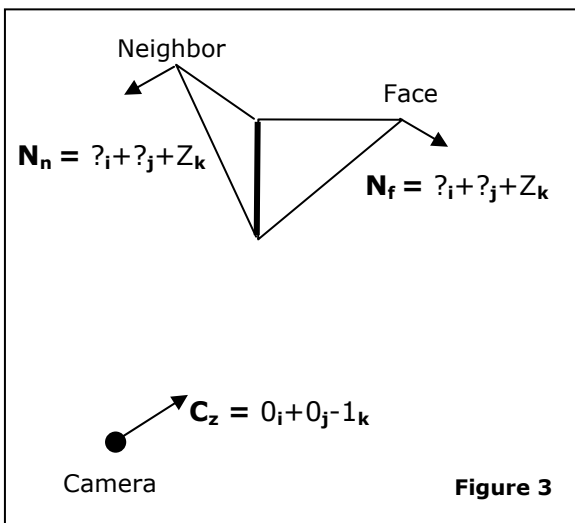
- 1) If necessary, generate the unnormalized face normal ( $\mathbf{N}_n, \mathbf{N}_f$ ).  
(1 cross product w/out normalization: 6 mults, 3 adds)
- 2) For each edge in each triangle:
  - 1) Generate a view vector pointing from one of the vertices of the edge to the location of the camera ( $\mathbf{V}$ ).

- (1 vector addition: 3 adds)
- 2) Take the dot product of the view vector with the face normals of the two neighboring triangles which form the edge ( $\mathbf{N} \bullet \mathbf{V}$ ).  
(2 dot products: 6 mults, 4 adds)
- 3) Compare the dot products for a sign difference. If the signs are opposite, the edge is a silhouette edge.  
(3 Boolean ops and 4 compares)

Silhouette edges are detected by essentially determining the angle between the view vector and each of the face normals. (The angle is not really determined, only whether or not the normal projects positively or negatively upon the view vector.) This algorithm works very well. It is not, however, completely accurate. It has the potential for misdetection, labeling edges as silhouettes that aren't and not labeling edges that in fact are silhouettes. The reason that this can occur is because the view vector is created from one of the triangle vertices and **not** from the center of the triangle. In practice, misdetection with this algorithm is rare, and it produces high fidelity results.

**Algorithm B – Fast Transform:**

This algorithm was generated in pursuit of the realization that a change in coordinate systems could greatly simplify the detection process. The steps are as follows (refer to Figure 3):



For each frame:

- 1) Construct a ModelView matrix from the current viewer orientation.  
(3 dot products, 3 multiplies: 12 mults, 6 adds)

For each vertex in the mesh:

- 1) Transform the vertex normal into camera space via the ModelView matrix.  
(4 multiplies, 3 adds)

For each triangle in the mesh:

For each edge in each triangle:

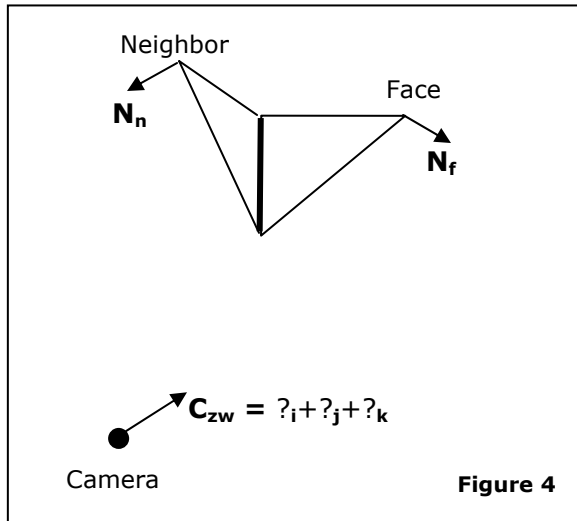
- 1) Use the vertex normals for the two opposing vertices of the face/neighbor pair. I.e., use the third vertex of each triangle, not either one of the two vertices constructing the shared edge ( $\mathbf{N}_{vn}, \mathbf{N}_{vf}$ ).  
Compare the Z components of the transformed vertex normals for a sign difference. If the signs are opposite, the edge is a silhouette edge.  
(3 Boolean ops and 4 compares)

Conceptually, this algorithm is performing basically the same steps as the Original algorithm. The view vector in this case is the camera vector, which is implicitly the Z-axis of the camera's orientation (i.e. the camera is defined to be looking along its Z-axis). The normal vectors for each face are no longer the true face normals as they were before, but instead are the vertex normals of the two opposing vertices (in the code, these are referred to as the *corner vertices*). The coordinate transformation allows for the elimination of the two dot products in favor of simply testing the sign of the normals' Z components. This algorithm proved to be disappointing visually. There can be a great deal of misdetection, stemming from the fact that both the camera vector and the normals are approximations. The Z-axis of the camera is not equivalent to the true view vector from the camera to the triangle. The vertex normals are generally not equivalent to the face normal, unless no smoothing or other

preprocessing has been done. If the vertex normals *are* equivalent to the true face normals, or even if the algorithm is used on the face normals themselves, then the error is proportional to the angle between the true view vector and the camera space Z-axis. At any rate, the error introduced by the two approximations is enough to cause sufficient misdetection to make the algorithm unusable.

### Algorithm C – Fast Dot Products:

This algorithm is akin to the *Fast Transform*, in that alternative vectors are used as approximations to true face normals and a true view vector. The steps are as follows (refer to Figure 4):



For each frame:

For each triangle in the mesh:

For each edge in each triangle:

- 1) Use the pre-existing world space representation of the viewing vector ( $\mathbf{C}_{zw}$ ). Use the vertex normals of the two opposing vertices of the face/neighbor pair ( $\mathbf{N}_{vn}$ ,  $\mathbf{N}_{vf}$ ). Take the dot product of the camera vector with the vertex normals.  
(2 dot products: 6 mults, 4 adds)
- 2) Compare the dot products for a sign difference. If the signs are opposite, the edge is a silhouette edge.  
(3 Boolean ops and 4 compares)

This algorithm is basically the same as the *Fast Transform*, except that the steps take place in world space coordinates. Thus, instead of using the implicit camera vector in camera space (i.e. the camera's Z-axis) we use the same vector in world space. This vector is maintained by the *Camera* class and does not need to be generated by the SED algorithm. However, instead of simply having the Z coordinates of the vertex normals in camera space, the dot products of the vertex normals with the camera vector in world space are needed. The results of this algorithm are identical to those of the *Fast Transform* algorithm, and the same problems apply.

### Algorithm D – True SED:

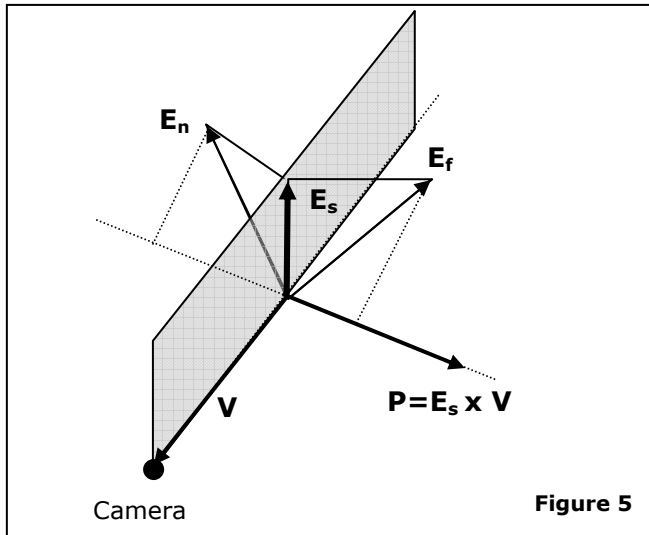
This algorithm is unlike any of the previous, and stems from a different approach to silhouette edge detection. The theory is that a silhouette edge can also be detected using a plane formed by the true view vector and the shared edge vector itself. In this case, the view vector is from the camera location to one of the vertices of the shared edge. Basically, the view is straight down the edge of the plane. The steps are as follows (refer to Figure 5):

For each frame:

For each triangle in the mesh:

For each edge in each triangle:

- 1) Form 4 vectors. The first is the view vector from a vertex of the shared edge to the camera position ( $\mathbf{V}$ ). The second is the shared edge vector itself, formed via the two vertices of the edge ( $\mathbf{E}_s$ ). The last two vectors are edges on each triangle of the face/neighbor pair, with the head at the opposing vertex of each triangle and the tail at one of the vertices of the shared edge ( $\mathbf{E}_f$  and  $\mathbf{E}_n$ ).



- (4 vector additions: 12 adds)
- 2) Form a vector normal to the plane of the view vector and the shared edge vector, by taking the cross product of the two ( $\mathbf{P} = \mathbf{E}_s \times \mathbf{V}$ ). This will be called the plane vector, for lack of a better name. (1 cross product: 6 mults, 3 adds)
  - 3) Take the dot product of the plane vector with the two edge vectors, **not** the shared edge vector ( $\mathbf{P} \cdot \mathbf{E}_f$  and  $\mathbf{P} \cdot \mathbf{E}_n$ ). (2 dot products: 6 mults, 4 adds)
  - 4) Compare the dot products for a sign difference. If the signs are equal, the edge is a silhouette edge. (3 Boolean ops and 4 compares)

For the scenario depicted in Figure 5, the two dot products have opposite signs and so the shared edge is not a silhouette edge. The key to the algorithm is the construction of the  $\mathbf{P}$  vector. It is vital because it is perpendicular to both the edge and the view vector. It is helpful to think of the two triangles as being able to pivot about the shared edge between them, i.e. that  $\mathbf{E}_s$  can be considered an axis of rotation for the faces. No matter how they are rotated about a fixed  $\mathbf{E}_s$ , the orientations of  $\mathbf{V}$ ,  $\mathbf{P}$ , and  $\mathbf{E}_s$  will never change.

Because  $\mathbf{P}$  is perpendicular to  $\mathbf{V}$ , we can take the dot product of  $\mathbf{P}$  with both  $\mathbf{E}_f$  and  $\mathbf{E}_n$  to determine which side of the  $\mathbf{VE}_s$  plane each triangle is on. To put it another way, the dot product of each of the vectors with  $\mathbf{P}$  gives either a positive or negative projection of that vector along  $\mathbf{P}$ , indicating upon which side of the  $\mathbf{VE}_s$  plane the third vertex of each triangle resides. If both vertices reside on the same side (i.e. the dot products have the same sign) then one triangle must be occluding the other and their shared edge is perceived as a silhouette edge. If the vertices lie on opposite sides, then both faces are visible relative to that particular view vector. It is worth noting that either of the non-shared edges of a given triangle can be used to project onto  $\mathbf{P}$ . Since both vectors for the triangle share the same end point (the corner vertex) they will have identical projections. Also, the angle between  $\mathbf{V}$  and  $\mathbf{E}_s$  does not matter, since the resulting  $\mathbf{P}$  will always be perpendicular to the two, which is the important thing. To help visualize this, imagine the two faces (and their shared edge) swiveling around  $\mathbf{P}$  in Figure 5. The projections of the two vectors onto  $\mathbf{P}$  do not change. In the figure,  $\mathbf{E}_f$  will always project positively and  $\mathbf{E}_n$  will always project negatively, and the shared edge will never be a silhouette edge.

### Performance Estimates:

The following table shows the type and quantity of operations necessary for each of the four algorithms described. A dot product requires 3 multiplies and 2 adds. A cross product requires 6 multiplies and 3 adds. A vector addition consists of 3 adds.

If we assume there are  $n$  faces in the mesh, then the following relationships represent reasonable assumptions. Obviously, the ratio of vertices to faces and edges to vertices will vary depending on the size and shape of the mesh, so these are only intended to be rough estimates for an average case.

Faces:  $f = n$ ;  
 Vertices:  $v = 2f = 2n$ ;  
 Edges:  $e = 6v/2 = 6(2n)/2 = 6n$ ;

The ratio of vertices to faces ( $v/f=2$ ) is based on observations made by Chris L. Gorman (G3D). The ratio of edges to vertices ( $e/v=6/2=3$ ) is based on the fact that a nonborder vertex is shared among 6

edges, but that each edge is shared among 2 faces, hence the division by 2. For any reasonably large mesh, nonborder vertices will far outnumber border vertices.

### Operations per Frame

| Algorithm         | Multiplies                                  | Adds                                      | Bool Ops            | Compares            |
|-------------------|---|---|---------------------|---------------------|
| Original          | $6f+6e$<br>$= 6n+36n$<br>$= \mathbf{42n}$   | $3f+7e$<br>$= 3n+42n$<br>$= \mathbf{45n}$ | $3e = \mathbf{18n}$ | $4e = \mathbf{24n}$ |
| Fast Transform    | $12+4v$<br>$= 12+8n$<br>$= \sim\mathbf{8n}$ | $6+3v$<br>$= 6+6n$<br>$= \sim\mathbf{6n}$ | $3e = \mathbf{18n}$ | $4e = \mathbf{24n}$ |
| Fast Dot Products | $6e = \mathbf{36n}$                         | $4e = \mathbf{24n}$                       | $3e = \mathbf{18n}$ | $4e = \mathbf{24n}$ |
| True SED          | $12e = \mathbf{72n}$                        | $19e = \mathbf{114n}$                     | $3e = \mathbf{18n}$ | $4e = \mathbf{24n}$ |

### Summary

It appears as though the *Original* SED algorithm has withstood the test and remains the best choice for detecting silhouette edges in a real-time rendering environment. Its near perfect accuracy and middle of the road op count make the algorithm a no-brainer compromise. The speedy *Fast Transform* algorithm is the down and dirtiest of the bunch and its super low op count makes it attractive. However, unless you're working with models that don't have any smoothed edges, it's results are just too ugly. The *True SED* algorithm is the only choice if you absolutely positively have to detect every silhouette edge and only silhouette edges, but since all of its ops are done on a per edge basis, the calculations are steep compared to the others.