A Silhouette Edge Lab for Computer Graphics

Paul Nixon, Tufts University

1 Background

In computer graphics, the generation of the silhouette edge is one of the most important fundamental techniques in non-photorealistic rendering. The silhouette edge appears as the outline of a three-dimensional object or between its parts, but for any viewing angle it can be generated from the usual data required for more common photorealistic rendering techniques.

The usual representation of an object in computer graphics is to specify a set of three-dimensional points (also called vertices), and then to specify how those points are arranged into triangles. To draw the triangles, the points are transformed from the three-dimensional space in which they are specified into a two-dimensional space analogous to the screen where they are to be displayed. From the points which make up each triangle, the normal vector perpendicular to the plane of the triangle is used to simulate the bouncing of light, allowing for photorealistic shading.

There are many algorithms involved in simulating the physics of how light interacts with objects, but generation of the silhouette edge is not one of them. In fact, the silhouette edge is defined by the boundary between faces which interact with light at all and those which do not.

The silhouette edge is made up of those edges on the boundary between faces which can be seen onscreen and faces which are hidden from view. Its generation is one of the times when computer graphics operates not on faces or on vertices, but on the edges, the places where two faces meet.

When displayed on its own, it appears as the "outline" of a three-dimensional shape, or the border between parts of the shape. It is used in situations which require simulation of artists' hand-drawn rendering techniques. Sometimes this is because an outline can make shapes easier for the viewer to discern, and sometimes this is for purely artistic purposes such as the creation of a "cel-shaded" art style for a video game or movie.

2 Motivation

The generation of the silhouette edge was chosen for an inclass lab for several reasons.

It was important to expose students to the techniques involved in non-photorealistic rendering. It was important to have the students work with the geometric principles involved in generation of the silhouette edge, and also the data structures involved.

However, the generation of the silhouette edge was not suitable for inclusion in the out-of-class assignments, because each of those assignments built off the previous one, and focused together on creation of successive techniques in traditional photorealistic rendering.

The data structure problem in generation of the silhouette edge is that the list of all edges must be found before the silhouette edge (a subset of the total edge list) can be calculated. Shapes are specified only as vertices and faces, so generation of the edge list requires iterating through the list of faces and finding all the places where two faces meet (that is, where two points are shared between a pair of faces). To do this requires knowledge of the relation between data and geometry, to find a new statistic of the existing data. But it also requires knowledge of the computational representation of data, for there is more than one data structure which can be used to fulfill the requirements of the edge list from which the silhouette edge can be synthesized.

The more heavily geometric aspect of the problem comes afterwards, when the edge list has been created and the silhouette edge is to be found from within it. This requires calculations for each face based on the face's orientation with respect to the viewer. Instead of these calculations being used simply to shade each face as in most computer graphics algorithms, they are used for a more drastic choice of whether or not to display an edge at all.

There are more steps involved in cel-shading an object from its silhouette edge, but they were deemed infeasible. A silhouette edge will include edges which on their own have all the properties of the silhouette edge, but would be hidden from view in reality by other parts of the object in between the edge and the viewer. To simulate this phenomenon, called occlusion, correctly, each pixel which might be colored as part of the silhouette edge must be checked against each face to see if the face is in the way preventing the pixel from being seen. To do this in real time is possible, but it requires more advanced techniques than the students had learned at this point in the course.

Also, the viewer's look vector towards each face is dependent on both the viewer's position and the position of the face itself. In a perspective view, the resulting look vector is different for each face, but an orthographic projection was used instead so that the look vector would be the same for all faces, simplifying the geometric calculations involved.

To keep the amount of work to a feasible level for students to complete in one hour, some peripheral parts of the program to find and display the silhouette edge were provided to the students. This included the parser for the mesh files containing faces and vertices, along with a user interface and basic program to load them and display the solid shapes. Parsing was considered an important skill, however, and the same parser had in fact been the assignment for a previous lab assignment.

3 Problems

The data structure part of the silhouette edge lab involved reading from two structures and creating a third. The vertices, which were read in first from the mesh file, were stored as triples of floating point numbers, as an array by the program in the same order as they were listed in the file. The triangular faces those vertices were arranged into were read from the file afterwards, stored both in the file and in the program as triples of indices into the vertex list.

The use of arrays for both the vertex list and the face list made sense because these data structures had a known size indicated by the mesh file header and didn't need to support lookup by anything other than index.

However, the number of edges isn't necessarily known before the edge list is done being created, and lookup of an edge within the edge list by its constituent vertices is useful during this process. So an array isn't the best way to store the edge list.

The set of edges comprising the silhouette edge can change based on the angle at which the shape is being viewed, so the list of all edges must be traversed every time the viewing angle is changed. The need for speed is a theme in computer graphics programming, and the reason this is an interesting data structures problem is the difference in speed between an adequate data structure and a more optimized one.

The geometric part of the problem is to find and display the silhouette edge, working from the list of all edges. This changes every time the viewing angle is changed, hence it is important that it can be calculated quickly.

Conversely, it is not important to store the actual silhouette edge; it can be sent to the graphics hardware and then discarded. It is at this stage that the edge list is traversed, checking each pair of adjacent faces to see if the edge between them is a part of the silhouette edge. The necessary condition is that one face is facing towards the viewer and the other face is facing away. This can be detected by calculating the angle between the viewer's look vector and the surface normal vector for each face. If the angle is less than 90 degrees, the face is towards the viewer and in view, whereas if the angle is 90 degrees or greater, it cannot be seen. So the algorithm must check this angle for every face.

Since the normal vector for a face does not change, it makes sense to precompute and store this for all faces. A less obvious optimization is to compute the angle for each face prior to checking the two faces in each edge, since each face is almost certain to be a member of multiple edges.

4 Solutions

Building the edge list was the more difficult part of the problem for most students. For the data representation of an edge to be useful, each edge must contain some representation of both the vertexes contained by the edge (for drawing the edge), and of both the faces containing the edge (for determining if the edge will be drawn).

The usual approach is to iterate through the list of faces, and for each pair of vertexes in a face, either add a new edge to the list of edges, or add a new face to the faces in an existing edge. The aspect of this which lends itself to more than one approach is the lookup of an edge within the edge list.

The obvious approach is to use some sort of hash table, but the hashing scheme is less obvious, since an edge is uniquely identified by no one piece of data but by a pair of vertexes. A less obvious approach is to simply iterate through the entire edge list every time an edge is searched for. This is an $O(n^2)$ algorithm, but its use here can be justified by the realization that the edge list only needs to be built once per object, during the initial load of the object file. If this approach is chosen, the edge list can be made up of a singly-linked list, giving the advantages of an easily expandable and quickly-iterable data structure.

My own solution to the data structure problem was to use an edge list made up of an array of linked lists. The array was the same size as the vertex list and indexed by the lower of the two vertex indices for a given edge. The linked lists were used because a vertex could have several edges for which it was the lower-numbered vertex.

This provides a O(n) algorithm (with a constant factor for the number of faces per vertex).

The students were not required to use the same data structure, but the declaration for it was included and some groups chose to use it.

In retrospect, this was a mistake. The use of a hybrid data structure was confusing to the students, produced more trouble spots in code, and made even my own solution more difficult to write. These disadvantages were not outweighed by the advantage of faster edge list creation, and a simple singly-linked list would have been a more sensible solution.

5 Results

The lab was completed by students in groups of two or three.

There were about ten groups, with one professor and three teaching assistants present.

This allowed the students to work collaboratively and to get help whenever both members of any team got stuck.

Most students figured out the algorithm for calculating the edge list within the first half hour of the hour long lab session.

However, the data structure proved to be a point of confusion for several teams (and a point of contention for one team who spent fifteen minutes arguing about hashing schemes).

When the lab session ended, half of the teams in the room had completed the assignment, and most of the others had either the algorithm or the data structure.

When the results were demoed for grading the following week, most groups had completed the assignment successfully, and the one group which had not completely rendered the silhouette edge was able to render most segments of it.

Six groups sent me their code afterwards (frustratingly, the team with an incomplete silhouette edge was not among them). Among the code I saw, three teams used my hybrid data structure for the edge list. The final code did not show any signs of the struggles they may have had in using it. One team used a standard library vector, and one team built their own expanding array. These teams had increased runtimes for edge list generation, but the silhouette edge was still produced correctly and in approximately the same amount of time. Runtimes were especially bad for the expanding array, probably due to the cost of allocating a larger array and copying from the old one when the array ran out of space.

One data structure difference within the individual edges was the choice of whether to maintain an ordering condition of the faces within the edge, or to check each combination when searching for an edge within the edge list. This was a requirement when using the hybrid data structure as the array was indexed by the lower-numbered edge, but neither the team using the standard library vector nor the team who built their own expandable array decided to maintain this condition. The result of this was that the extra traversal steps for the hybrid data structure had approximately the same code size as the extra checks for the non-hybrid data structures.

The geometric parts of the algorithm were almost identically similar among different teams' implementations, since the mathematical definition of the silhouette edge did not leave much room for creativity. All groups used the sign of the dot product (scalar product) instead of the actual angle for deciding if a face was facing towards or away from the viewer, but only one team had the idea of multiplying two dot products to check if their signs were opposite.