Lecture 7: Hidden Line Removal

This lecture concerns the removal of hidden lines from wire frame models seen in perspective projection. We will assume that the data describing the scene is organised in such a way that the topological information can be obtained. In other words, we set up a list of the 'faces' or 'facets' of the objects in the scene. Each 'face' record is comprised of a list of 'lines', and each line record is an index to a pair of points. The topological information can be stored either by means of pointer structures or in arrays. Algorithms of this kind are appropriate for line drawings (e.g. for architectural visualisation) or for engineering drawings where the hidden lines may be shown as feint or dotted lines. They are generally faster than the painter's algorithm and z buffer algorithm introduced below.

Hidden Parts

In the previous lectures, we have considered only wire frame models, whereby a scene is depicted by drawing only its edges. For animations, the polygons represent solid objects (like the wall of a building), and each face is usually opaque. This leads to the problem of removing the hidden parts of a polygon. The simplest, though not the most efficient algorithm for doing this is to draw the polygons in the order of their distance from the viewer. As each polygon is filled with a colour, the parts of those already drawn are occluded by the new polygon. In order to do this, it is necessary to transform all the vertices of the scene to the viewing coordinate system, in which the viewpoint is the origin and the view direction is the positive z axis. Then, the order of the polygons is sorted according to the smallest z coordinate of the vertices. This simple algorithm will fail on certain cases of overlapping (shown in Diagram 6.4). It will be very inefficient in cases where there are many polygons in a scene that extends far into the distance.

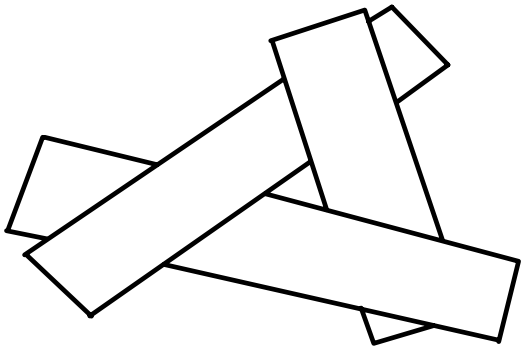

Diagram 7.1: Failing case for the painter's algorithm

A better method is the Z buffer, which is commonly available in hardware. At each pixel a record is kept of the z distance of the polygon that is currently shown. Each distance is initialised to a high value before starting. When a new polygon is being drawn, a check is made at each pixel to see whether it nearer than the polygon currently shown, and the pixel is only set if it is. If the polygons are sorted into order, and drawn starting with the nearest to the viewpoint, then drawing can stop as soon as all the pixels have been set. Alternatively, the Z buffer can be used without sorting. Providing that all the polygons are processed, the drawing can be done in any order.

Single Convex Objects

For a single convex object it is easy to verify that if any point on a face is obscured, the whole face cannot be seen. Thus we need simply test each face for visibility and if the test is successful, then draw all the lines of that face. The simplest test is to make use of the half space property of planes, namely that for a plane with equation f(x,y,z)=0, all points on one side of the plane will have positive f(xi,yi,zi) and all points on the other side of the plane will have negative f(xj,yj,zj). Suppose we know an internal point of the convex object, [xi,yi,zi]. If that point is on the same side of the plane of the face as the viewpoint, then the face must be obscured. This gives us a simple pseudocode algorithm:

<* find an internal point by averaging the vertices *>
for <*each face of the object *>

```
        <* find the plane equation ax + by + cz + d = 0  i.e. f(x,y,z)=0 *>
        <* for the viewpoint find sign(f(0,0,0)) which is the sign of d *>
        <* for a point inside the object find sign(f(xi,yi,zi) )*>
        if sign(f(0,0,0)) not equal to  sign(f(xi,yi,zi))
        then  <*draw all the edges of the face *>
        end if
    end for;
```

Note that:

(i) An internal point may be found by taking the average of all the vertices of a convex object. It would increase efficiency to compute and store it with the other points in the data structure. It can be transformed in the same way as the vertices when changing the viewpoint.

(ii) The plane equation parameters a,b,c,d can be found from three vertices of a face **V1**,**V2**,**V3**. Two vectors on the plane are **p1** = **V1**-**V2** and **p2** = **V1** - **V3**. The vector product gives the normal to the plane **n** = **p1**x**p2** = [a,b,c]. Having thus found a b and c we re-arrange the equation into the form d = -ax - by - cz, and by substituting any one vertex into the equation we can calculate d.

Multiple convex objects and/or concave objects.

With concave objects we must compare every line making up the scene with every object face, and determine what part of the line (if any) is obscured by the face. A complication comes from the fact that the visible parts of a line will not necessarily be continuous. Thus, the line being tested will need to be represented by a list of line segments. For simplicity we will assume that all the faces are convex polygons, giving the possibilities shown in Diagram 7.2. We will consider later how to extend the algorithm to avoid this restriction.

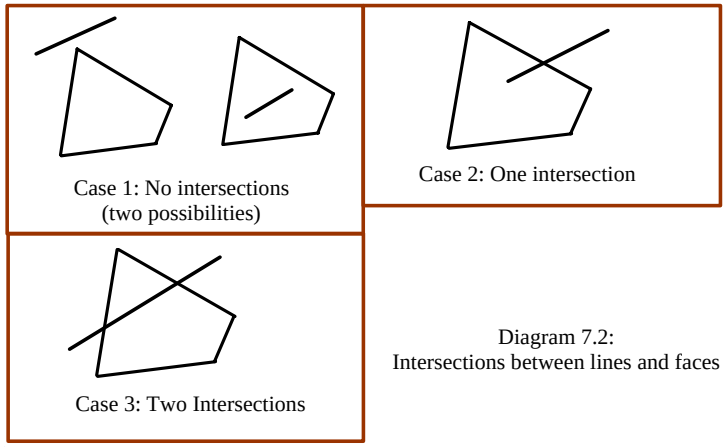

Diagram 7.2:
Intersections between lines and faces

```
    for <*each line of the scene*>
        <*set up a list of line segments (this initial list will have just one item on it) *>
        for <*each face of each object*>
        for <*each line segment on the list*>
            <* compute all intersection points between the line segment and the face's edges *>
            if <* no intersections *>
            then   if <*both points inside the polygon *>
                       then  if  <* the face obscures the line segment *>
                                 then <* delete the line segment from the list *>
                                 end if
                       end if
            end if
            if <*one intersection*>
            then   <* find which part of the line is inside the polygon *>
                       if <*the face obscures the contained part of the line*>
                       then <* replace the line segment being tested with the visible part *>
                       end if
            end if
            if <*two intersections*>
            then   if <* the face obscures the inner part of the line (between the intersections)*>
```

then <* replace the line segment being tested with the outer two segments *>
end if
end if
end for
end for
<* draw all line segments that remain on the list (if any) *>
end for

Notes:

(i) This algorithm is partly computed in object space (3D), and partly in image space (2D). For efficiency, it is possible to precompute the projection of each point of the scene and store it in the data structure along with the 3D data.

(ii) To compute the intersections between the polygon edges and the line segment being tested we equate the two equations:

line segment: $\mathbf{p} = \mu\,\mathbf{p2} + (1-\mu)\,\mathbf{p1}$

face edge: $\mathbf{p} = \nu\,\mathbf{p4} + (1-\nu)\,\mathbf{p3}$

at intersection: $\mu\,\mathbf{p2} + (1-\mu)\,\mathbf{p1} = \nu\,\mathbf{p4} + (1-\nu)\,\mathbf{p3}$

when separated into the x and y components we have two equations to solve for μ and ν. It will be necessary to test for the case of the two lines being parallel before solving for the intersection. Having solved for μ and ν it is necessary to check that the intersection is on the line segment and on the edge, that is that μ and ν are in the range [0..1]. Care is also needed in the case where a line intersects at a vertex, i.e. when $\nu=0$ or when $\nu=1$. There is a remote possibility that the intersection is counted twice, and the condition for an intersection should therefore be written $0<\nu<=1$.

(iii) When testing whether a polygon obscures a part of a line it is necessary to use the three dimensional data. We need to find a three dimensional point on the line segment being tested. Unfortunately a general point on a line in 3D space ($\mathbf{pi}+\mu(\mathbf{pj}-\mathbf{pi})$) does not project to the point ($\mathbf{pi}'+\mu(\mathbf{pj}'-\mathbf{pi}')$) in two dimensions. So the only way we can be sure we are looking at the right part of the line in 3D is to invert the perspective projection. Suppose that the mid-point of the line segment that we are testing in 2 dimensions is **m**. This is the projected point in 2D, and it will have a coordinate in the form (x,y,f) where f is the focal length of the perspective projection. In three dimensional space the points **pi** and **pj** are the end points of the corresponding line (i.e. the 3D vertices of the object). Then, the corresponding point to **m** in 3D is given by the intersection of the two lines:

object edge: $\mathbf{p} = \mu\,\mathbf{p2} + (1-\mu)\,\mathbf{p1}$

projector: $\mathbf{p} = \nu\,\mathbf{m}$

at intersection: $\mu\,\mathbf{p2} + (1-\mu)\,\mathbf{p1} = \nu\,\mathbf{m}$

which we can solve for μ, ν and hence **p** as above. Note that this time there is a valid intersection if $0<\mu<1$ and $\nu>0$, and there should always be a valid intersection.

Having solved for **p**, we now determine whether the viewpoint and point **p** are on the same side of the plane of the face using the same method that was applied for single convex objects (above). If they are on different sides, the face obscures the line segment.