Incremental Conservative Visibility with General Occluders

CSC2522 Course Project

Yee Whye Teh
Hao Zhang

Contents

1 Introduction

Consider a walk-through in a complex scene consisting of complex polyhedral objects. As the viewpoint moves and as the viewing direction is altered, the scene needs to be rendered over and over again. If the scene contains hundreds of thousands to millions of polygons, the traditional depth buffer algorithm, though equipped with hardware accelerations, can no longer achieve interactive performance.

The major bottle-neck in a rendering algorithm for a highly complex scene is the visibility computation. An obvious goal is to *render* as few polygons as possible in each frame. To this end, fast visibility culling algorithms capable of culling away a significant portion of the invisible polygons is desirable [7]. In most case, an object space bounding volume hierarchy can be used to improve efficiency. Also, levels-of-detail techniques can be applied to reduce the number of polygons in the scene while still maintaining satisfactory image quality.

For most visibility culling algorithms, the number of polygons *processed* is still large. The ultimate goal would be to process as few polygons as possible in each frame, and locate these polygons as quickly as possible. To this end, the intrinsic temporal coherence between successive frames in a walk-through has to be exploited somehow. This is the focus of our current investigation. Let us first motivate our study in more details.

1.1 Depth buffer vs. incremental visibility

In the depth buffer algorithm, after each recorded change of viewing parameters, depth tests are performed on *every* polygon in the scene. Several observations suggest that this approach can be quite inefficient. First of all, the percentage of visible polygons at each stage may be small, so a lot of time is wasted in processing invisible polygons; this is especially costly when sophisticated shading models are used in rendering. Secondly, the visibility status of a polygon will likely remain the same after a small change to the viewing parameters; repeated application of the depth buffer algorithm does not exploit this temporal coherence. So many more polygons than necessary are processed at each rendering step, which would make interactive viewing difficult to achieve, especially for a complex scene with hundreds of thousands to millions of polygons.

An alternate approach is to maintain the visibility information of the scene dynamically, and take advantage of spatial and temporal coherence between successive frames to update visibility incrementally. The visibility information maintained may be just the set of visible polygons. As the viewing parameters change, this set is incrementally updated in an efficient way. Of course, there is extra time and storage cost involved. But the trade-off appears to favor the latter approach as the scene becomes more complex, as long as we have an efficient incremental update algorithm.

1.2 Exact visibility vs. conservative visibility

Exact visibility can be maintained, using *aspect graphs*, for example [3, 6]. An aspect graph partitions the 3D space into a set of regions using surfaces of *visual discontinuity*, or *visual events*. When a viewpoint moves within a region, visibility information does not change. The visibility information stored in each region is a *structural* description, i.e., a set of boundary edges, of the visible portion of each polygon in the scene. Such information changes only when the viewpoint crosses a visual event.

To maintain visibility exactly, two kinds of visual events need to be considered. A *vertex-edge* (*VE*) event is formed by a vertex and an edge, as shown in Figure 1.1(a), and a *triple-edge* (*EEE*) event is a ruled quadric surface formed by three edges which intersect each other when viewed from the view point, as shown in Figure 1.1(b).

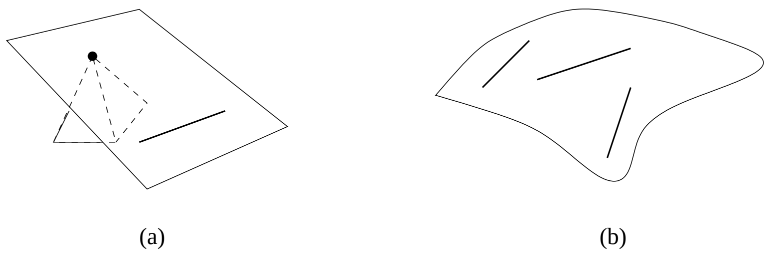

FIGURE 1.1 *A VE event and an EEE event.*

The major drawback of aspect graph is its size. With EEE events, the number of regions can reach $\Theta(n^9)$, where n is the size of the input scene. Also, the visible portion of the scene can be of size $\Theta(n^2)$, e.g, consider a scene with n vertically placed strips in front of n horizontally placed strips.

Hubscheman and Zucker [4] propose an algorithm for incremental update of *exact* visibility in a scene consisting of closed, *convex* polyhedra. Much time is spent in maintaining an exact structural description of the visible portion of each polygon as the viewpoint moves. With hardware-accelerated polygon rendering, we believe it is more efficient to simply maintain a set of visible polygons and render all of them, especially for complex scenes.

Coorg and Teller [1] adopts the notion of *conservative visibility.* They *overestimate* visibility by maintaining a *superset* of the visible polygons. After each change of viewing parameters, the depth buffer algorithm is used to render the set of polygons maintained. As long as all the visible polygons are included, correctness is guaranteed.

1.3 Convex vs. non-convex occluders

One way to make conservative visibility an efficient approach is to ignore EEE events. So only a *linearized* version of the aspect graph is considered, which reduces the computational complexity significantly. Coorg and Teller [1] achieves this by considering occlusion by *single convex* objects only.

Definition 1 **(Conservative visibility of Coorg and Teller [1])** *A polygon is invisible if and only if all its vertices are occluded by a single convex polyhedron.*

In their visibility algorithm, a set of designated *occluders* are identified. Only occlusions caused by these occluders are considered. They motivate this by the observation that in many scenes, a few objects cause most occlusions and checking other objects for occlusion increases the overhead without increasing the number of polygons found to be occluded.

The major drawback of their algorithm lies in the assumption that all occluders must be convex, plus their inability to handle occlusions by multiple objects. In fact, they only describe the occlusion characteristics between two convex polyhedra, that is, even the occludee is taken to be convex.

In most scenes, non-convex occluders exist. When non-convex objects or combinations of small disjoint objects are the most significant occluders, Coorg and Teller's algorithm is not expected to perform much better than depth buffering.

1.4 Conservative visibility with general occluders

In this paper, we investigate visibility with much more general occluders. An occluder can have arbitrary shape and topology. For example, we are able to handle occluders such as a bowl, a tea pot, or a pretzel. An occluder does not even have to be of a connected structure. For example, in a scene of trees with large leaves, occlusion by the set of leaves can be handled.

In other words, we are able to maintain occlusions by *multiple objects*. At first it would appear that to do this, EEE events have to be included. To avoid this, we consider the occluding objects as forming a single occluder. We *preemptively* record a "potential" occlusion or emergence of an occludee before they actually occur. This is motivated by the fact that before an EEE event can occur the three edges involved have to overlap each other in the image plane first. With conservative estimate of the occluder, we can reduce the computational cost involved significantly. Note that the ability to combine a "forest" of small, disjoint, and overlapping occluders is a key feature of the *hierarchical occlusion map* algorithm [7]. We compare this algorithm with ours in Section 7.1.

Our notion of visibility is conservative since there may be polygons, which are completely occluded by single or multiple occluders, but still considered visible by our algorithm. However, such cases are not expected to occur often. Our algorithm is expected to cull away most invisible polygons. Of course, if a polygon is truly visible, our algorithm will definitely render it.

For the scene model, we assume that all the objects are closed polyhedra which are two-dimensional manifolds. Each face is a triangle oriented outwards. All our algorithms apply to objects with arbitrary polygonal faces. In terms of the movement of the viewpoint, we do not allow the view point to penetrate a face of an object.

We present an incremental conservative visibility algorithm which computes the set of visible polygons, taking advantage of the spatial and temporal coherence between successive frames. In each frame, a set of visual events are tested against the movement of the viewpoint. For each visual event crossed by the viewpoint, relevant visibility information is updated. The spatial and temporal coherence is reflected by the fact that the "wavefront" of visibility (or invisibility) moves continuously on an occludee. So we can make use of not only the previously computed results, but also the adjacency information stored in the representation of the input objects.

2 Overview

In this section, we give an overview of our algorithm for incremental conservative visibility and the relevant concepts.

First, let us fix some terminology. An occluder A is not necessarily a single object; it can be a set of objects collectively playing the role of an occluder. However, there is no need to do the same for occludees, so an occludee is just a single object, or a face of an object.

For clarity, let us assume that an occluder and an occludee must be *separable by a plane*. That is, there is a plane P such that the occluder and the occludee lie entirely on opposite sides of P. This assumption makes the occluder-occludee relationship unambiguous. In Section 6.1, we show how this assumption can be enforced by selecting the occluders appropriately.

In our algorithm, all the relevant visibility information is dependent on the position of the viewpoint. The viewpoint is not associated with a specific viewing direction. So a projection is really onto a virtual image sphere, instead of an image plane. But to appeal to our common intuition, we still refer to the image plane in our discussion.

We denote the plane formed by a vertex V and an edge e by $\langle V, e \rangle$. Two edges or faces in 3D are said to *overlap* with respect to a viewpoint Q, if their projection with respect to Q intersect in their respective *interior*. A vertex from one object is said to be *contained* in a face from another object with respect to a viewpoint Q, if the projection of the vertex is contained in the projection of the face with respect to Q. Finally, we denote the projection of an object A onto the image plane by $P(A)$.

2.1 Visual events

Definition 2 **(Visual event)**
A visual event in 3D is a surface which, when crossed by the view point, causes the visible portion of some polygon to change topologically.

Our algorithm ultimately involves the detection and response to visual events. Out of efficiency concerns, we handle only those visual events that are *planar*. A visual event may or may not affect the visibility *status*, i.e., visible or invisible, of a polygon. We distinguish these two types of visual events by the following definition.

Definition 3 **(Critical visual event)**
A visual event is said to be critical if, when it is crossed by the viewpoint, the visibility status of some polygon changes.

In Figure 2.1(a), the shaded occluder does not occlude the triangle completely. So the triangle is visible. As the viewpoint moves across the plane $\langle V, e\rangle$ from above, the triangle will become invisible, as shown in Figure 2.1(b). The reverse process occurs if the viewpoint crosses plane $\langle V, e\rangle$ from below. The visual event $\langle V, e\rangle$ is critical.

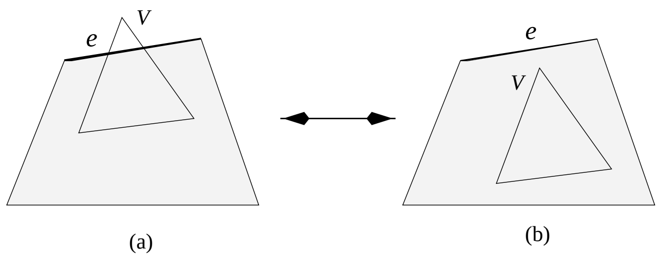

Figure 2.1 *A critical visual event.*

While in Figures 2.2(c) and (d), the visual event $\langle V, e\rangle$ is not critical, since it does not change the visibility status of the triangle. But the visible portion of the triangle changes topologically.

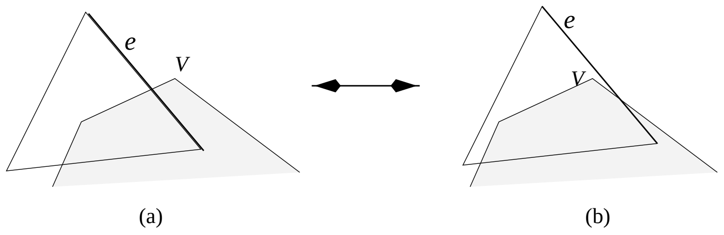

Figure 2.2 *A non-critical visual event.*

2.2 Silhouettes and outlines

Let A be an occluder and B an occludee. It is quite obvious that visibility changes on B can occur only near the *outlines* of the occluder A, as seen from the viewpoint. So it is necessary to maintain the set of edges of A on its outline. For an edge to be on the outline, it has to be a *silhouette edge*.

DEFINITION 4 **(Silhouette edge)**
An edge e is said to be a silhouette edge with respect to a viewpoint Q if and only if one face incident to e is facing Q, and the other face incident to e is facing away from Q. The silhouette of an object is the set of silhouette edges on that object.

A face is facing the viewpoint Q if and only if Q is *above* the plane containing that face, where the "up" direction is the normal direction. Conversely, a face is facing away from Q if and only if Q is *below* the plane containing that face, as shown in Figure 2.3. For simplicity, we call the face facing Q *forward-facing*, and the face facing away from Q *back-facing*.

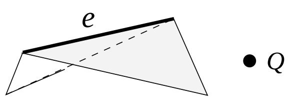

FIGURE 2.3 *A silhouette edge e with respect to viewpoint Q.*

Clearly, not all silhouette edges of A appear on its outline if A is non-convex; this is explained in more details in Section 3.1. But in order to maintain A's outline, we need to maintain its silhouette, since a silhouette edge may emerge on the outline as the viewpoint moves. The silhouette of the occludee B also needs to be maintained, since it contributes to various visual events. To see this, we note that any emergence or disappearance of B behind A must start at B's silhouette. Section 3 is devoted to the maintenance of silhouettes.

In Section 4.1, we define the *actual outline* of an occluder with respect to a viewpoint. It turns out that it is very expensive computationally to maintain the set of edges on the actual outline, since EEE events would have to be considered, as we shall explain in Section 4.2. So we propose the notion of *conservative outline*, which is described briefly in the next section, and the maintenance of *silhouette overlappings* in Section 4.3, which is motivated by the fact that before an EEE event can occur, the three edges involved have to overlap in the image plane.

2.3 Conservative outline

An occludee B, where B can be an object or a face of an object, is *totally occluded* by an occluder A if, for all points P on B, A lies between the view point Q and P. Under the assumption that the occluder A and the occludee B are separated by a plane, this is equivalent to

- the projection $P(B)$ of B is contained in $P(A)$; and
- there exists some point P on B such that A lies between the view point Q and P, i.e. B is behind A.

This gives us a criterion to determine the visibility of any occludee with respect to an occluder. If in the above criterion we replace $P(A)$ by an area *contained* in $P(A)$, then we have a *conservative* visibility criterion. In particular, we may use a *conservative outline* of A whose enclosed area is contained in $P(A)$. Conversely, we can use an area which contains $P(B)$ as an estimate of the occludee B to compute conservative visibility; for example, a bounding volume of B may be used.

The conservative outline is obtained from the actual outline by adding some necessary silhouette edges and deleting some actual outline edges. A precise definition is given in Section 4.4.

2.4 Relevant planes

As the viewpoint moves, we want to make sure that no "emerging" polygon (from invisible to partially visible) is missed. Moreover, we should detect most polygons which disappear (from partially visible to invisible).

To keep track of these visibility changes, we maintain a set of *relevant planes*, a term adopted from Coorg and Teller [1]. When the view point crosses one of these relevant planes, either there is a visibility change or the set of relevant planes needs to be updated, or both. For example, in both Figure 2.1 and 2.2, the plane $\langle V, e \rangle$ is relevant.

There are various types of relevant planes. In Section 5.2, we introduce them in details and classify them according to how they are formed and their functionality. In Section 5.3, we compare our definition of relevant planes to that of Coorg and Teller [1], and point out an error in their definition.

2.5 Visibility maintenance

2.5.1 The "belt" of partially visible triangles

Let us focus on one occluder A and one occludee B. Each *cycle* on the outline of A corresponds to a "belt" of triangles on B, as shown in Figure 2.4. To see why there must be a cycle of edges on the outline, refer to Section 4.1. This belt of triangles act as the "wavefront" of visibility (or invisibility) of the occludee. Since visibility changes occur only near the outline of the occluder, maintaining them can be accomplished by simply keeping track of changes to this belt of triangles.

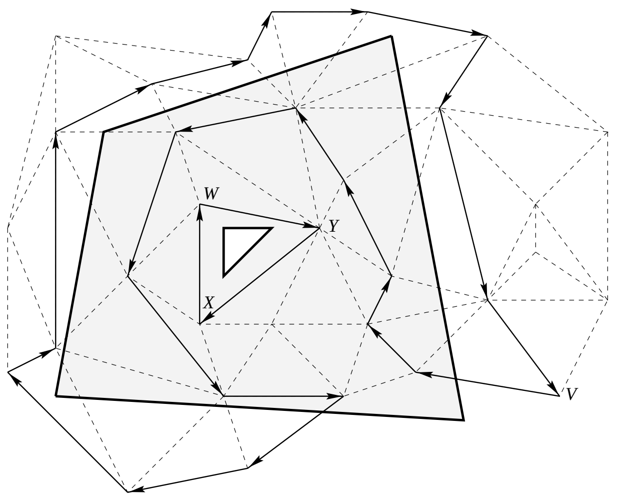

FIGURE 2.4 *There are two belts on the occludee. The boundary of each belt is traced out by arrowed lines. The belt containing vertex V corresponds to the outside outline. The "inner" belt, corresponding to the inside outline, is just a triangle WXY. Note that the inner outline does not overlap any edge of WXY.*

If a triangle f of B, with edges d_1, d_2, d_3 and vertices U_1, U_2, U_3, is on the belt, then either an outline vertex V of A is contained in f, as shown in Figure 2.5(a), or an outline edge e of A overlaps two edges of f, as in Figure 2.5(b). In the former case, if f "leaves" the belt, then one of the planes $\langle V, d_1\rangle$, $\langle V, d_2\rangle$, or $\langle V, d_3\rangle$ must be crossed by the viewpoint. In the latter case, the plane $\langle U_1, e\rangle$ must be crossed. Note that all these aforementioned planes will be relevant planes, which we shall define in Section 5.

The same holds for a face "entering" the belt, in which case other types of relevant planes will be involved.

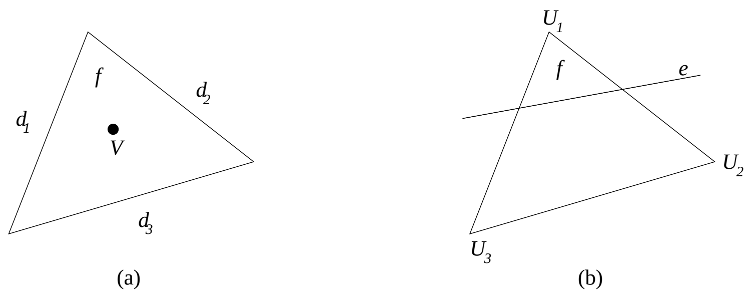

FIGURE 2.5 *A face leaving the belt.*

2.5.2 Updating visibility status of a face

Let f be a face from an occludee B and let A be the occluder. The visibility status of f with respect to A has something to do the number of outline primitives (edges or vertices) overlapping f.

Denote the set of outline edges from A by $\mathcal{OE}(A)$ and the set of outline vertices by $\mathcal{OV}(A)$. If no edge from $\mathcal{OE}(A)$ overlaps an edge of f, and no vertex from $\mathcal{OV}(A)$ is contained in f, then f is either completely occluded by A or it is completely visible with respect to A. The "in-between" configuration is a partial occlusion.

DEFINITION 5 *With respect to a viewpoint Q, an occluder A, and a face f from an occludee B, denote the number of edges in $\mathcal{OE}(A)$ overlapping some edge of f by $\mathcal{N}_E(f, A)$, and the number of vertices from $\mathcal{OV}(A)$ contained in f by $\mathcal{N}_V(f, A)$. The sum of $\mathcal{N}_E(f, A)$ and $\mathcal{N}_V(f, A)$ is denoted by $\mathcal{N}(f, A)$.*

The number $\mathcal{N}(f, A)$ records the number of outline primitives from A overlapping f. When a visual event occurs, $\mathcal{N}(f, A)$ is updated accordingly. If $\mathcal{N}(f, A)$ is positive, we consider f to be visible with respect to the occluder A; if it is zero, then its visibility status can be determined by the nature of the latest visual event. A face is visible if it is visible with respect to all the occluders; otherwise, it is invisible, and is not rendered.

Let us consider an example. Refer to Figures 2.6. In all figures, the visual event is identified by V and e. In (a), we have $\mathcal{N}(f_1, A) = 1$ and $\mathcal{N}(f_2, A) = 3$, and both faces are visible. In (b), $\mathcal{N}(f_1, A) = 2$ and $\mathcal{N}(f_2, A) = 3$, and no visibility changes. In (c), $\mathcal{N}(f_1, A) = 3$ and $\mathcal{N}(f_2, A) = 0$, and f_2 becomes invisible; f_1 is still visible. In (d), $\mathcal{N}(f_1, A) = 4$ and $\mathcal{N}(f_2, A) = 1$, and f_2 becomes visible again. In (e), $\mathcal{N}(f_1, A) = 4$ and $\mathcal{N}(f_2, A) = 1$, and no visibility changes.

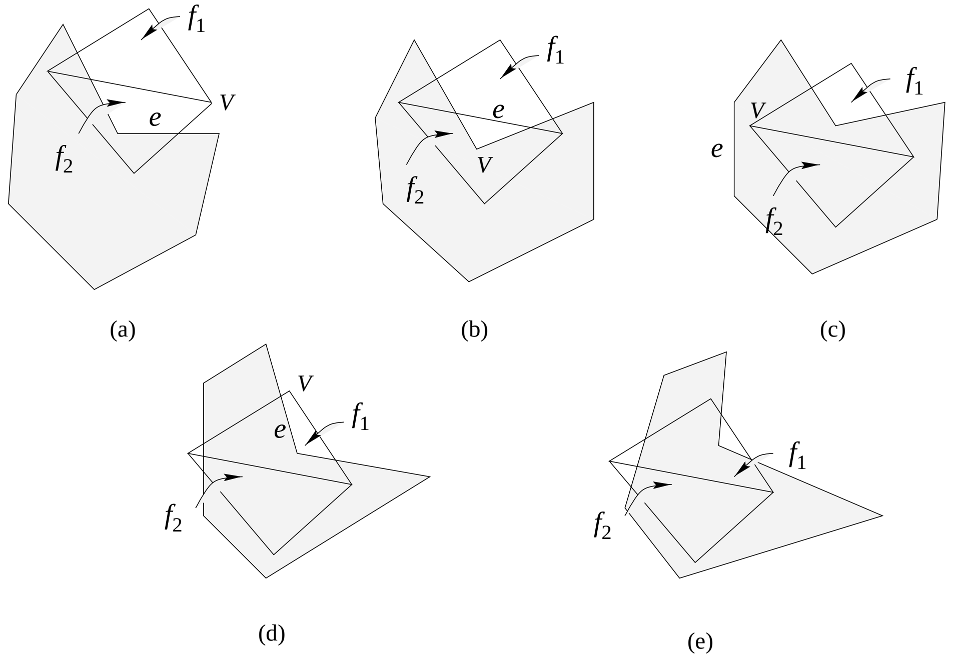

FIGURE 2.6 *Example of visibility update.*

There are cases where $\mathcal{N}_E$ is 0 and what matters is $\mathcal{N}_V$, as shown in Figure 2.7, where the occludee triangle reveals itself through a hole in the occluder.

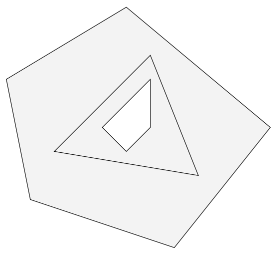

FIGURE 2.7 $\mathcal{N}_E(f, A) = 0$ *and* $\mathcal{N}_V(f, A) = 4$.

2.6 Overview of algorithm

Our algorithm maintains a superset of the visible polygons as the viewpoint moves. A set of occluders is selected according to certain criteria. The selection criteria may vary with the movement of the view point, as we will explain in Section 6.1.

In the initialization stage, we make preparations for our incremental update algorithm. More specifically, we do the following.

1. Select a set of occluders;
2. Initialize the set $\mathcal{S}$ of silhouette edges of all the objects, the set $\mathcal{O}$ of outline edges of the occluders, and the set $\mathcal{R}$ of relevant planes;
3. Cull away all the back-facing polygons with respect to the initial viewpoint;
4. Initialize the set $\mathcal{P}$ of visible polygons and render them using depth buffer algorithm.

For each recorded movement of the viewpoint, we

1. Detect all the relevant planes crossed by the movement;
2. Update $\mathcal{S}$, $\mathcal{R}$, $\mathcal{O}$, and $\mathcal{P}$;
3. Use depth buffer algorithm to render all polygons in $\mathcal{P}$;
4. Reselect the set of occluders if necessary.

3 Dynamic maintenance of silhouettes

3.1 Silhouettes of convex and non-convex objects

We observe that the set of silhouette edges of an object form a graph, possibly disconnected, in which every vertex has even degree; the graph is Eulerian. So the silhouette of any close polyhedral object is composed of a set of silhouette cycles.

For a convex object, the notion of silhouette agrees with our intuition of an outline of the object. When projected onto the image plane, the silhouette form a convex polygon.

For non-convex objects however, it is not so simple. Even if the scene contains just one object, a silhouette edge may not even be visible. In fact, an edge of an object can be a silhouette edge, but it never appears on the outline of the object no matter where the object is viewed from. For such an edge, the dihedral angle made by the two adjacent faces is obtuse ($> \pi/2$). We call such an edge *concave*; all other edges are convex.

When projected onto the image plane, the silhouette of a non-convex object may be self-intersecting. Even a silhouette cycle may be self-intersecting in the image plane; for example, consider the silhouette of a donut.

3.2 Difficulty in silhouette maintenance

Let us call a face incident to a silhouette edge a *silhouette face*, and the plane containing such a face a *silhouette plane*. Unless absolutely needed, we do not distinguish between a face and the plane containing the face. If all the objects are convex, we observe the following.

- The set of silhouette edges need to be updated only when the view point crosses a current silhouette plane.
- If the movement of the viewpoint is sufficiently small, only edges adjacent to a current silhouette edge can become new silhouette edges.

These two observations together make efficient incremental update of the set of silhouette edges for convex objects possible. The first observation ensures that the number of planes we have to test as the viewpoint moves is small, since the complexity of the silhouette of an object is in general much less than that of the object itself [5]. The second observation ensures that the number of candidates for new silhouette edges is small. When only one face is crossed by the viewpoint, we only need to test two edges, since all faces are triangles.

When non-convex objects are included however, we can no longer make the same claim. The following example shows that when the viewpoint moves in small increments, that is, one face is crossed at a time, an edge may become a silhouette edge "abruptly", meaning that such an edge is not adjacent to any current silhouette edge, and no current silhouette plane is crossed.

In Figure 3.1(a), we have part of an object; it is like a small tetrahedron placed on a horizontal square. The viewpoint is above the face ABC. All the triangles in Figure 3.1(a) are visible. The view point is to move in the direction indicated by the solid arrow. Figure 3.1(b) shows a top view of the situation.

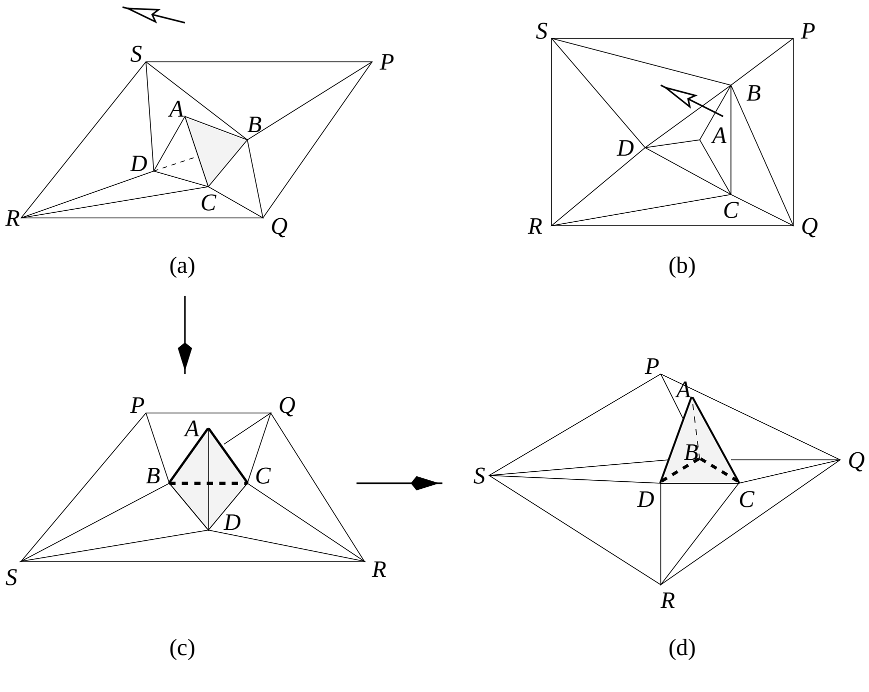

FIGURE 3.1 *Abrupt appearance of silhouette edges.*

When the viewpoint crosses plane ABC, face ABC becomes invisible (back-facing). Edges AC, AB, and BC all become silhouette edges all of a sudden. Figure 3.1(c) shows what the object looks like from this new viewpoint. Now when the viewpoint crosses plane ABD, face ABD becomes invisible. Edges AD and BD become silhouette edges, while edge AB ceases to be a silhouette edge, as shown in Figure 3.1(d).

In order to maintain silhouettes in a scene with general occluders, we have to test more planes than just the current silhouette planes. This leads to the notion of *saddle faces*.

3.3 Saddle faces and silhouette maintenance

DEFINITION 6 **(Saddle face)**
A face is called a saddle face if one of its edges is convex, and one of its edges is concave. The plane containing a saddle face is called a saddle plane.

Clearly, a convex object does not have any saddle face. On the other hand, the number of saddle faces on an object can be proportional to the total number of faces. For the latter point, consider again a donut-shaped object; there should be a lot of saddle faces along the inside ring of the donut.

It turns out that to keep track of silhouette changes, the set of silhouette planes plus the set of saddle planes are sufficient, as the following theorem shows.

THEOREM 1 *Starting from the initial viewing position Q, move the viewpoint continuously. The first face in the scene (or plane containing a face in the scene) crossed by the viewpoint is either a saddle face or a silhouette face with respect to Q.*

Proof: See Appendix 8.1. □

As soon as a face crossed by the viewpoint is detected, updating the silhouettes is easy, as shown by the next two propositions.

PROPOSITION 1 *Any new silhouette edge must be incident to a face just crossed by the viewpoint; the same holds for any edge which just ceases to be a silhouette edge.*

Proof: Let e be a new silhouette edge with incident faces f_1 and f_2. Without loss of generality, let f_1 be forward-facing and f_2 back-facing. Since e was not a silhouette edge, either both f_1 and f_2 were forward-facing or both were back-facing. Therefore, either f_1 or f_2 has just been crossed by the viewpoint. The case where e ceases to be a silhouette edge is similar. □

PROPOSITION 2 *If e is a silhouette edge and one face incident to e is just crossed by the viewpoint, then e is no longer a silhouette edge. Similarly, if e is not a silhouette edge and one face incident to e is just crossed by the viewpoint, then e will become a silhouette edge.*

Proof: Similar to the previous proof. □

Now given a short path traced out by the viewpoint, we can update the set $\mathcal{S}$ of silhouette edges as follows.

1. Detect all the faces crossed by the viewpoint and sort them by crossing time.

2. For each face f crossed in order

```
        For each edge $e$ of $f$
                If $e \in \mathcal{S}$, then $\mathcal{S} = \mathcal{S} - \{e\}$;
                If $e \notin \mathcal{S}$, then $\mathcal{S} = \mathcal{S} \cup \{e\}$.
```

4 Dynamic maintenance of outlines

4.1 Actual outlines

With respect to a view point Q, the *actual outline* of an occluder A is the boundary of the projection $P(A)$ of A onto the image plane. Since A is polyhedral its actual outline is a number of non-intersecting polygons, as shown in Figure 4.1. Each edge on the actual outline is a *segment* of the projection of some silhouette edge of A. Such a silhouette edge is called an *actual outline edge*. Note that sometimes not the whole actual outline edge is on the actual outline, as depicted in Figure 4.1.

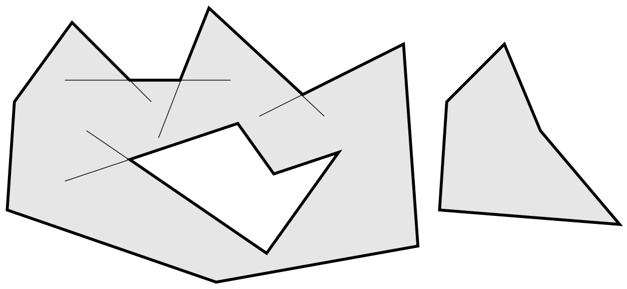

FIGURE 4.1 *Projection of actual outline edges. The thick edges form the actual outline. The thin edges are segments on the actual outline edges that are not on the actual outline.*

An actual outline edge has an orientation on the image plane which tells on which side of the edge the object lies. We call the "object side" the *inside* of the edge, while the other side the *outside*. Clearly, we can organize the actual outline edges into a number of cycles each of which trace out one polygon of the actual outline. Note that an actual outline edge may appear multiple times on such a cycle.

In a general scene in which the only restriction is that the objects are composed of two-dimensional manifolds and the manifolds do not inter-penetrate, the actual outline does not capture all the subtleties of occlusion. For example, in Figure 4.2, the partial occlusion of the "wedge" K by L cannot be detected, since L's silhouette (depicted by a thick edge) does not appear on the actual outline of the object; at the same time, the occlusion of an object G by L where G lies between L and K cannot be detected.

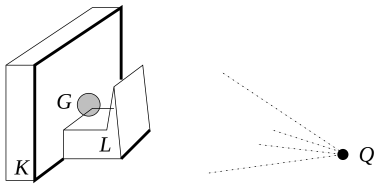

FIGURE 4.2 *Occlusion missed by using actual outlines only.*

The inability to detect these occlusions is really not a big problem. The self-occlusion of an object can simply be ignored, meaning that we simply render all the forward-facing polygons on an occluder. With the assumption that any occluder and any occludee are separable by a plane, there cannot be an object G lying between wedges K and L, as in Figure 4.2. Moreover, by virtue of conservative visibility, an undetected occlusion does not affect the correctness of rendering.

4.2 Difficulty in maintenance of actual outlines

Maintaining the set of outline edges precisely proves to be difficult, since it requires the consideration of EEE events, as illustrated in Figure 4.3: When e is below the crossing V, as shown in (a), it is not an actual outline edge. As the viewpoint moves and e crosses V, it becomes an actual outline edge, as shown in (b). The event at which e crosses V is an EEE event, and the set of viewpoints at which such an event occurs form a quadric surface. The detection of such events is too costly computationally.

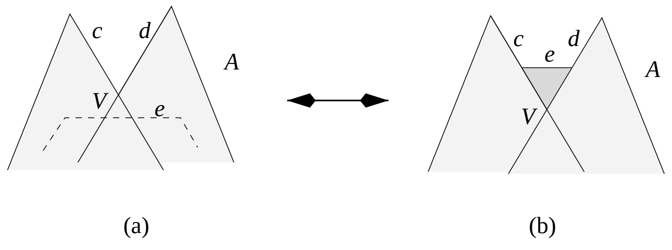

FIGURE 4.3 *Precise maintenance of outlines invovles EEE events.*

A similar problem might arise at the "inside" outline, where a hole might appear or disappear in the projection of the occluder onto the image plane, and an EEE event is involved again. This is depicted in figure 4.4.

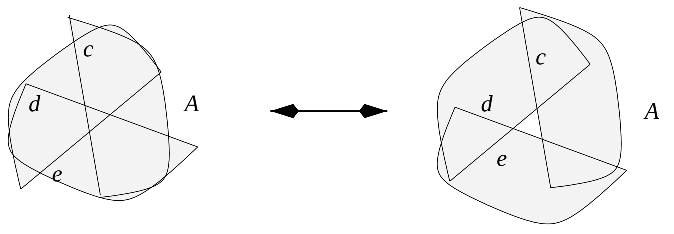

FIGURE 4.4 *A hole may appear or disppear after an EEE event.*

As seen from these two examples, an EEE event occurs only when there are three edges overlapping each other. This observation suggests that we should keep track of triples of overlapping silhouette edges in our algorithm. However, if we are to maintain only a *conservative estimate* of the actual outline, then not all these triples have to be considered.

4.3 Silhouette overlappings

Recall that a conservative estimate of an actual outline encloses an area *contained* in the area enclosed by the actual outline, on the image plane. Let us refer to Figure 4.5. Consider the "outline" O traced out by edges c and d. It is the actual outline if e is below V, as shown in (a), while in (b), where e is above V, it is a conservative outline. In either case, no triangle will be found invisible if it is actually visible. However, in (b), occlusions at the deeper-shaded region are not detected. This problem is fixed whenever the edges e "splits" into two edges after a silhouette face incident to e is crossed by the viewpoint and an outline vertex U emerges, as shown in (c). Note that we are able to detect the emergence of U, since the corresponding visual event is either $\langle U, c\rangle$ or $\langle U, d\rangle$.

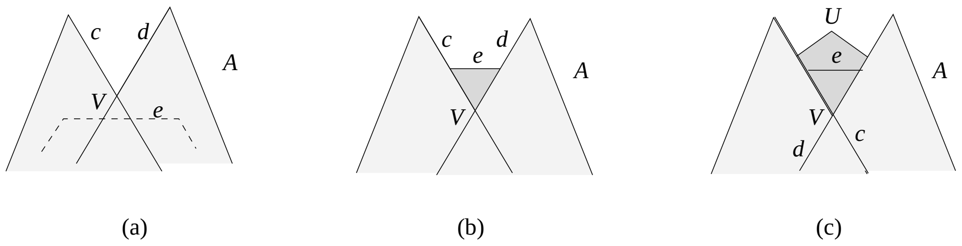

FIGURE 4.5 *Conservative estimate of outside outline.*

It is possible to obtain a conservative estimate of the actual outline from the set of actual outline edges: Whenever there are three consecutive edges on the actual outline, we could remove the *middle* edge, since the other two edges also overlap, and thus "closes the gap" in the actual outline. However, there is one exception. If these three edges alone form a cycle on the actual outline, as shown in Figure 4.4, we operate differently. We shall explain this a little bit later.

The removal of the "middle" edges can only be done *sequentially*, not in parallel. Refer to Figure 4.6. The three edges c, d and e all overlap each other, so do the three edges d, e and f. If we remove both d and e then there will be a "gap" left in the actual outline. As a result, an occluded polygon may "slip through the gap" undetected and remain occluded even though it already becomes visible. But if we remove d first. Then e would be "straddled" between c and f, and they do not all overlap each other, so e remains on the conservative outline.

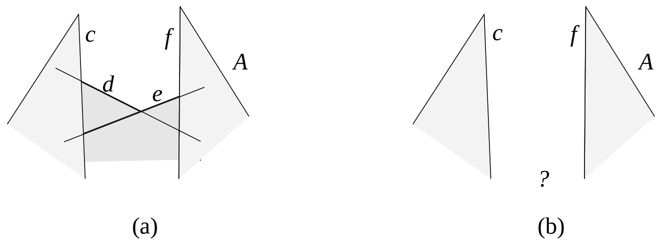

FIGURE 4.6 *Naively removing edges d and e wrecks havoc.*

Using sequential removal, the set of conservative outline edges are still guaranteed to form a cycle. Figure 4.7 shows two examples of conservative outline, on the outside and the inside. The cyclic property of the conservative outline is crucial, since the maintenance of visibility makes use of the belts of triangles, which requires the outlines used to be cyclic.

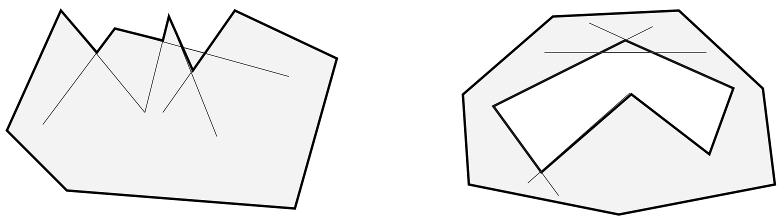

FIGURE 4.7 *Conservative outlines still form cycles.*

New let us consider the case shown in Figure 4.4, where three overlapping silhouette edges c, d, and e alone form a cycle on the actual outline. The three edges have to overlap each other in such a way that the triangle formed by the three points of intersection is either inside or outside all three edges. We should not remove any of the three edges to form a conservative outline, since we still want the conservative outline to be a cycle. A possible solution is to *preemptively* add the three silhouette edges c, d, and e into the conservative outline when we detect that they overlap each other in the interior of $P(A)$ and have their insides/outsides in the right configuration.

If there really is a hole, then the conservative outline would be the same as the actual outline. If there is no hole, then the extra outline edges will do no harm, because no visible triangle can ever be deemed invisible by the extra edges. Although the set of triangles deemed visible might not be correct, nevertheless it will contain the actual set of visible triangles—our algorithm computes conservative visibility.

4.4 Conservative outline

In this section, we make the notion of conservative outline precise. There are two types of vertices on the actual outline: a silhouette vertex, as shown in Figure 4.8(a), and the intersection of two silhouette edges, as shown in (b) and (c). In (a), we want both a and b to be in the conservative outline. The same is true in (b), while in (c), we do not want c to be in the conservative outline.

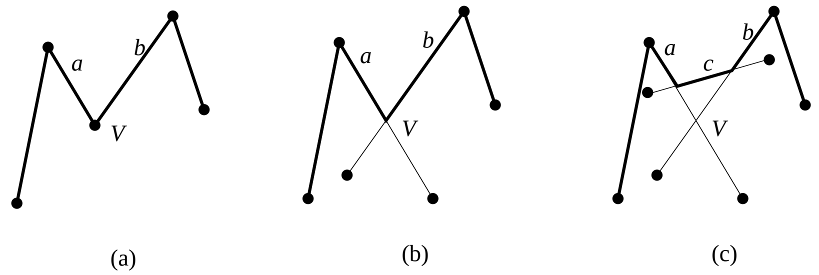

FIGURE 4.8 *The thick line is the actual outline; the thin edges are the actual outline edges; the dots denote silhouette vertices.*

Define the conservative outline of an occluder A as consisting of the following silhouette edges:

- Any three silhouette edges that overlap each other and form a cycle on the actual outline, as shown in Figure 4.4.
- For every cycle on the actual outline, one *subcycle* of actual outline edges in which no three consecutive edges all overlap each other.

In the definition, a list of edges form a cycle if any two consecutive edges either meet at a silhouette vertex, or they overlap in the image plane. The edges in the conservative outline are called conservative outline edges. **For brevity, in all subsequent discussions, we shall call these simply as the outline edges of A.**

4.5 Clipping the conservative outline edges

Consider Figure 4.9. The triangle was initially occluded. After the first visual event, vertex V crossed edge e, moving onto its outside. Locally we believe that the triangle is visible because part of it is now on the outside of the outline edge e. However the segment of e that it is overlapping is not on the conservative outline so it should still be occluded. After the second event, the triangle overlaps both e and d. Although it is still occluded, we should consider it as visible, since its emergence of the triangle from occlusion by moving upwards can only be detected by an EEE event.

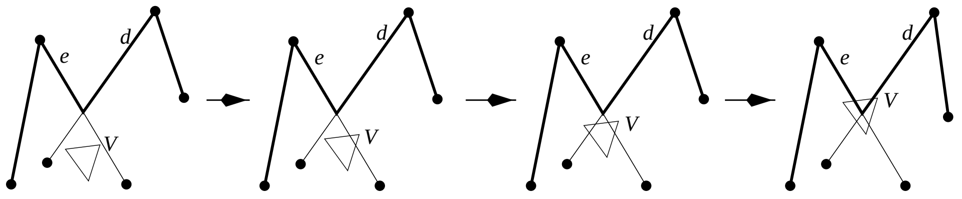

FIGURE 4.9 *The small triangle is still occluded after vertex V moved to the outside of outline edge e.*

To handle this case, we note that outline edge e overlaps with its neighboring outline edge d. To become partially visible, the triangle has to be partially on the outside of both e and d. If we keep track of all such pairs of consecutive outline edges which overlap each other, then we can essentially "clip" away those segments of outline edges which are not on the conservative outline. So we treat each pair of edges as a single primitive of the conservative outline, so that a triangle is partially visible with respect to the pair if and only if it is partially visible with respect to both edges. As a result, less occluded triangles will be considered visible.

4.6 Maintaining silhouette overlappings

To be able to efficiently detect that three silhouette edges overlap each other, for each silhouette edge e, we can keep track of all silhouette edges $\mathcal{C}(e)$ which overlap e. When a silhouette edge e is to overlap another silhouette edge d, go through $\mathcal{C}(e)$ and $\mathcal{C}(d)$ and determine whether there is a silhouette edge in both $\mathcal{C}(e)$ and $\mathcal{C}(d)$. If there is, we have three edges overlapping each other. This gives a worst-case run time of $O(n)$, where n is the number of silhouette edges, and a storage cost of $O(n^2)$. However, the number of silhouette edges overlapping a particular silhouette edge is usually very small, so these are acceptable computational costs.

A more serious slow-down may occur when the silhouette itself changes. This can happen when either a silhouette plane or a saddle plane is crossed by the viewpoint. Let us consider the former case. As shown in Figure 4.10, the silhouette edge e disappears from the outline, and is replaced by two silhouette edges c and d. We term such an event a *splitting* event—one silhouette edge splits into two.

For a convex object, the (outline) silhouette always form a convex polygon on the image plane, so there is no self-overlapping. Also, on the image plane, a new silhouette vertex can only occur in the triangle bounded by the old silhouette edge and its two neighboring silhouette edges. So updating the relevant information when a splitting event involving a convex occluder occurs is quite simple.

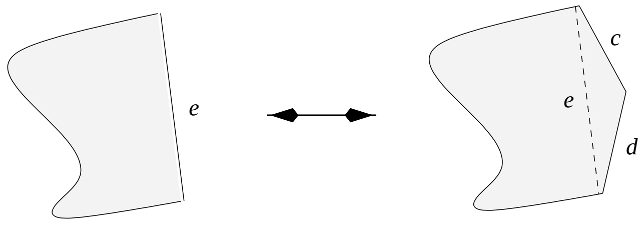

FIGURE 4.10 *A splitting event.*

For non-convex occluders, a new silhouette vertex can appear just about anywhere. Moreover, the new silhouette edges could overlap many other silhouette edges from the same occluder. To further complicate matters, these new overlappings are caused by abrupt appearance of silhouette edges, and they are practically undetectable without going through all the existing silhouette edges.

It takes linear time to exhaustively go through the set of silhouette edges and check whether the new silhouette edges overlap them. A speed-up can be achieved by using a grid partitioning of the 3D space. We will describe this in more details in Section 6.5.3.

5 Relevant planes

5.1 Supporting/separating planes and touch planes

Let A and B be two convex objects. Suppose that initially, A and B do not occlude each other. As the viewpoint moves, A may start to occlude B. In other words, A becomes the occluder and B becomes the occludee. This can happen only when the viewpoint has crossed a *supporting* or *separating* plane of A and B in a certain way.

Between two objects, a supporting plane is formed by a vertex from one object and an edge from the other object such that both objects lie on the same side of the plane, as shown in Figure 5.1(b), where the vertex and the edge forming the supporting plane are highlighted. Separating planes are similar but with two objects lying on opposite sides, as shown in Figure 5.1(a). With respect to a viewpoint, the supporting and separating planes must be formed by vertices and edges both from the silhouettes of the two objects involved.

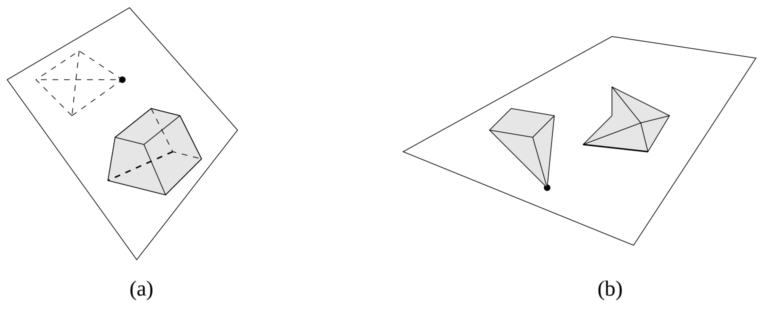

FIGURE 5.1 *A separating and a supporting plane.*

Supporting and separating planes are not sufficient to detect all the occlusions when objects can be non-convex. We need a generalization to handle "concavity" in both the occluder and the occludee. Let us call such generalized supporting and separating planes *touch planes*. Due to concavity, we have the following observations.

- The vertex and the edge forming a touch plane may come from the same object.

- We can no longer require that an object corresponding to a touch plane to lie entirely on one side of the plane; such requirement should at least be "localized". For example, in Figure 5.2(a), both $\langle U, e\rangle$ and $\langle V, d\rangle$ should be touch planes; while $\langle V, c\rangle$ should not be, since it cuts the "cone" rooted at V.

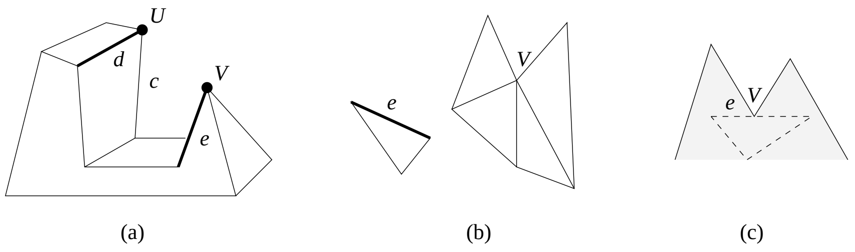

FIGURE 5.2 *Touch planes for non-convex objects.*

Unfortunately, even locally, we cannot have the afore-mentioned requirement. This is illustrated by the example shown in Figure 5.2(b). The plane $\langle V, e\rangle$ should be a touch plane, since when the viewpoint crosses this plane, there will be visibility changes, as shown in Figure 5.2(c). But the cone rooted at V is cut by $\langle V, e\rangle$.

- A plane $\langle V, e\rangle$ is a touch plane if and only if from certain viewpoint, one of the two situations shown in Figure 5.3 occurs.

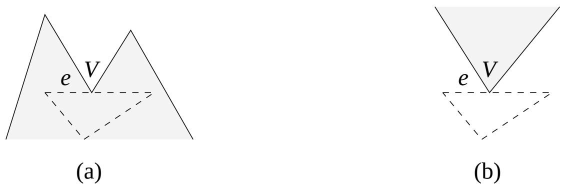

FIGURE 5.3 *Two possible configurations of a touch plane.*

DEFINITION 7 **(Touch plane)**
A touch plane between objects A and B, where A and B can be the same, is formed by a vertex V from A and an edge e from B, such that the two faces of B incident to e lie on the same side of $\langle V, e\rangle$, and there exist two edges a and b of A incident to V satisfying the following.

- *a and b can appear on the outline of A at the same time with respect to some viewpoint.*
- *a and b lie on one side of the plane $\langle V, e\rangle$.*

The plane $\langle V, e \rangle$ is a touch plane with respect to a viewpoint Q if a and b are outline edges with respect to Q.

With respect to a viewpoint, a touch plane must be formed by a vertex and an edge both from the silhouettes of the objects involved. Clearly, all supporting and separating planes are touch planes. When the viewpoint crosses a touch plane $\langle V, e \rangle$, the faces incident to e and the faces incident to V will either start or stop an occluder-occludee relation. Touch planes are essential in the maintenance of outline edges.

5.2 Classification of relevant planes

In Section 2.1, we introduced the concept of visual event. Besides the two kinds of visual events illustrated in Figure 2.1 and 2.2, there are other kinds of visual events that need to be maintained in order to incrementally update visibility, e.g., saddle planes, silhouette planes, and touch planes. We call all these planes the *relevant planes*.

In what follows, we further classify the relevant planes corresponding to an *occluder* A and an *occludee* B (they may be the same object) *with respect to* some viewpoint Q depending on how they are formed and their functionality, where either A or B or both can be non-convex. Note that the terms "plane" and "event" are used interchangeably.

5.2.1 Relevant silhouette and saddle planes (splitting events)

These planes are extensions of silhouette and saddle faces of A and B. Note here that we do mean silhouette planes, *not* outline silhouette planes. When the viewpoint crosses one of these planes, the set of silhouette edges, the set of silhouette overlappings, and the set of outline edges may need to be updated.

All silhouette planes are critical visual events; the face that is crossed by the viewpoint gets its visibility reversed. Note also that the set of relevant silhouette planes is dependent on the viewpoint Q, while the set of relevant saddle planes is not.

5.2.2 Relevant sliding events (SS-planes)

The silhouettes of two objects (or of one non-convex object) can interact by overlapping each other at various points. When the viewpoint moves, the silhouettes of the objects are seen to "slide" across each other. A *sliding event* occurs when a silhouette edge "slides" across a silhouette vertex, and the interactions between the silhouettes will change. We call the plane corresponding to a sliding event an *SS-plane*, where "SS" is for "silhouette-silhouette." Sliding events can be critical although not necessarily.

Relevant to objects A and B, we have the following sliding events, assuming that A and B are different with A being the occluder:

- For each *outline* edge $d = (U, V)$ of A and silhouette edge $e = (W, X)$ of B overlapping each other, the planes $\langle U, e\rangle$, $\langle V, e\rangle$, $\langle W, d\rangle$, and $\langle X, d\rangle$.

  For example, consider three silhouette edges (U, V), (W, X) and (X, Y), as shown in Figure 5.4. In (a), (U, V) overlaps (W, X) but not (X, Y). When the viewpoint crosses the plane $\langle U, V, X\rangle$, (U, V) "slides" across X. Afterward, (U, V) no longer overlaps (W, X) and starts to overlap (X, Y), as shown in (b).

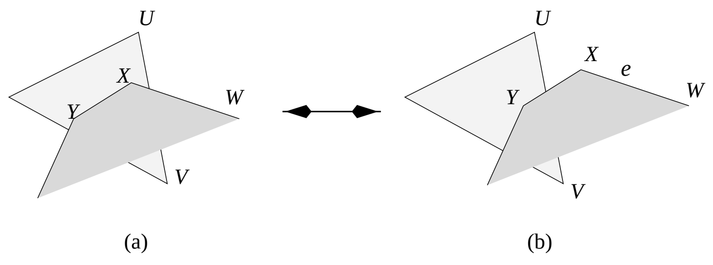

FIGURE 5.4 *Example of a sliding event.*

If A and B are the same occluder, then sliding events involving all *silhouette* edges, not just the outline edges, are relevant.

5.2.3 Relevant covering events (C-planes)

A *covering event* occurs when the viewpoint crosses a plane formed by an *outline* edge from an occluder and an *interior* vertex on an occludee; we call such a plane a *covering event plane*, or *C-plane* for short. We use the term "covering" since the corresponding event resembles the "covering" and "uncovering" of a vertex by an outline, as shown in Figures 5.5(a) and (b).

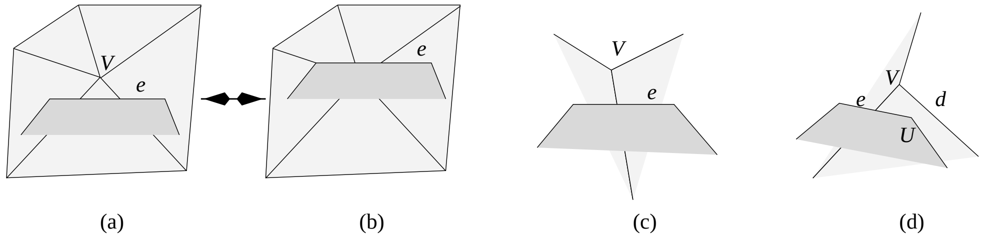

FIGURE 5.5 *Covering plane, touch plane, and non-relevant plane.*

A covering event may or may not be critical. When the viewpoint crosses a C-plane, there may be visibility changes and the set of C-planes may need to be updated.

Relevant to objects A and B, we have the following C-planes:

- For each outline edge e of A, for vertex V of each edge of B which overlaps e, if V is *not* on B's silhouette, then the plane $\langle V, e\rangle$.

  Note that if V is on B's silhouette, then $\langle V, e\rangle$ is either a sliding event, a touch plane, as shown in Figure 5.5(c), or it is not relevant, as shown in Figure 5.5(d), where plane $\langle U, d\rangle$ is relevant. In the situation shown in (d), the plane $\langle V, e\rangle$ cannot be a touch plane since the two silhouette edges incident to V lie on opposite sides of $\langle V, e\rangle$. So before $\langle V, e\rangle$ is crossed by the viewpoint, $\langle U, d\rangle$ has to be crossed first.

5.2.4 Relevant piercing events (P-planes)

A *piercing event plane*, or *P-plane* for short, is formed by an *outline* silhouette vertex from an occluder and an *interior* edge on an occludee. The term "piercing" is adopted since the corresponding event resembles the "piercing" of a face by a "tip" of the outline silhouette, as shown in Figures 5.6(a) and (b).

When the viewpoint crosses a P-plane, the set of C-planes and/or P-planes may need to be updated. A piercing event may be critical, as shown in Figures 5.6(c) and (d).

Relevant to objects A and B, we have the following P-planes:

- For each outline silhouette vertex V of A, for each *non-silhouette* edge e of B adjacent to a face f that contains V, the plane $\langle V, e\rangle$.

  Note that if e is on B's silhouette, then $\langle V, e\rangle$ is either a sliding event, a touch plane, or it is not relevant.

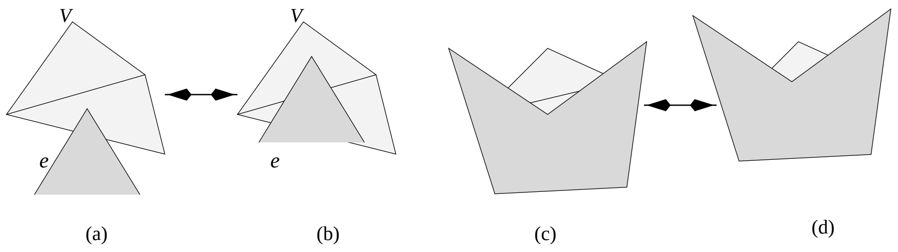

FIGURE 5.6 *Example of piercing events.*

5.2.5 Relevant touch planes

The following touch planes of A and B are relevant, if A and B are different.

- A touch plane with respect to the viewpoint Q, formed by an *outline silhouette* vertex from A and a *silhouette* edge from B.
- A touch plane with respect to Q, formed by an *outline silhouette* edge from A and a *silhouette* vertex from B.

If A and B are the same occluder, then we have to add any touch plane with respect to the viewpoint Q, formed by a silhouette vertex and a silhouette edge from A. These touch planes are used to maintain the outline of A. For example, in Figure 5.7, the plane $\langle V, e\rangle$ is relevant.

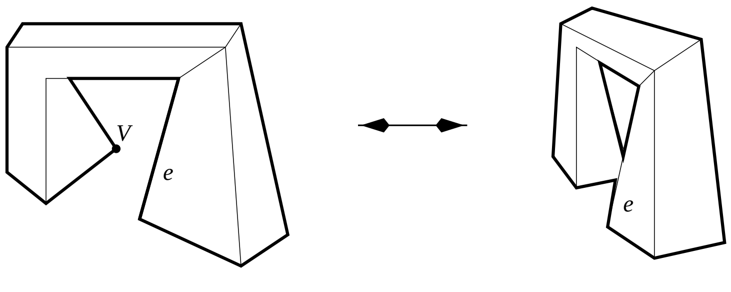

FIGURE 5.7 *Touch event and outline change.*

When a relevant touch plane of two different objects is crossed by the viewpoint, the two objects involved will either start or stop an occluder-occludee relation; so there will be relevant SS-planes, C-planes, and/or P-planes added or removed. If a relevant touch plane of one object is crossed by the viewpoint, the set of outline edges of the object may need to be updated.

5.3 Compare to relevant planes of Coorg and Teller

Comparing our definition of relevant planes to that of Coorg and Teller [1], we find three differences:

1. Coorg and Teller do not consider saddle planes since all occluders are assumed to be convex.
2. Coorg and Teller do not take care of all the sliding events, which we think is an overlook on their part. Without the inclusion of all sliding events, the set of relevant planes cannot possibly be maintained correctly.
3. Coorg and Teller do not consider all the touch planes since supporting and separating planes are sufficient.

Coorg and Teller [1] use the following theorem (Theorem 2 in their paper) to demonstrate that the set of relevant planes, by their definition, from a given viewpoint is sufficient to capture all visual events that occur when the viewpoint changes. However, we believe the absence of certain sliding events makes the result incorrect.

Coorg and Teller's Theorem on Relevant Planes:
Let A (occluder) and B (occludee) be two convex polyhedra and RP be the set of relevant planes from a viewpoint P_1. Let P_2 be a different viewpoint corresponding to a visual event. Then, either the visual event at P_2 is in RP or there exists a viewpoint in the segment $[P_1, P_2]$ that corresponds to a visual event in RP.

It is unclear what a visual event really refers to in the statement. If all relevant planes are considered visual events, then the example in Figure 5.8 gives a counterexample. It is possible that neither $\langle U, e\rangle$ nor $\langle V, d\rangle$ is a supporting plane, where $\langle U, e\rangle$ cuts B and $\langle V, d\rangle$ cuts A. The only plane crossed by the viewpoint from (a) to (b) is $\langle V, d \rangle$, which is not relevant by Coorg and Teller's definition. So without any visual event occurring, $\langle U, e \rangle$ becomes a new visual event. It is worth noting that the plane $\langle V, d \rangle$ is a sliding event by our definition.

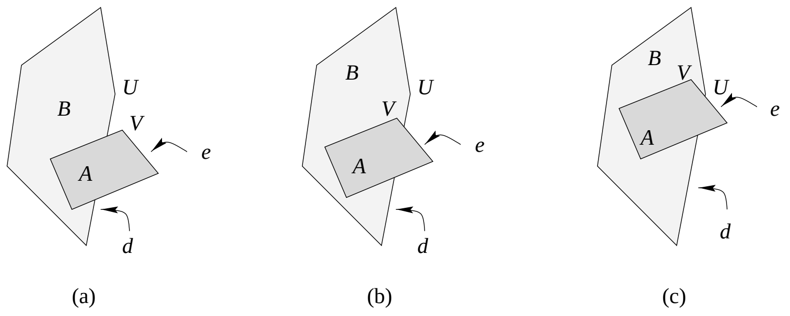

FIGURE 5.8 *Error with Coorg and Teller's Theorem.*

If only critical visual events are considered visual events, then Figures 5.9 provides a counterexample. From (a) to (d), $\langle V, e \rangle$ becomes a new critical visual event without any intermediate critical visual event occurring.

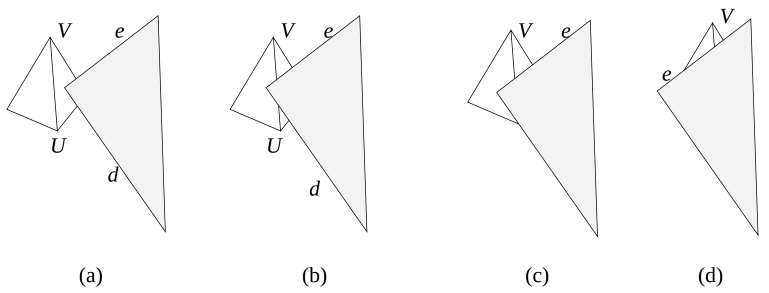

FIGURE 5.9 *Error with Coorg and Teller's Theorem.*

A third possibility is that all but the piercing events are visual events. But then from Figure 5.9(a) to (b), $\langle U, d \rangle$ becomes a new visual event without any intermediate visual event occurring.

5.4 The Relevant Plane Theorem

The following theorem is related to the maintenance of all the relevant planes. It generalizes the theorem of Coorg and Teller. In the theorem, we denote the set of relevant planes with respect to a viewing position by $\mathcal{R}(Q)$.

THEOREM 2 *Let Q_1 be the initial viewing position and suppose that the viewpoint moves from Q_1 continuously to Q_2. If $\mathcal{R}(Q_1) \neq \mathcal{R}(Q_2)$, then at least one plane in $\mathcal{R}(Q_1)$ has been crossed by the viewpoint.*

6 The incremental visibility algorithm

6.1 Selection of occluders

Naturally, the significant occluders should appear large as seen from the viewpoint. So we can use the obvious size and distance criteria to select occluders dynamically. An occluder can be a combination of a set of small, disjoint, overlapping objects.

We do not want the outline structure of a single occluder to be too complex. This is because when an occluder is added to or deleted from our selections, there will be necessary updates to the set of relevant planes; these updates should not take too much time.

Both Coorg and Teller [2] and Zhang et al. [7] suggest techniques for dynamic selection of occluders. All these techniques can be applied to our algorithm.

Recall that in Section 2, we made the assumption that any occluder and any occludee should be separable by a plane. We can enforce this assumption at the stage of occluder selection, by "merging" any object which intersects the convex hull of an already selected occluder with that occluder. If the computation of convex hull is deemed to be too expensive, we can simply use a bounding volume instead.

6.2 Initialization

- **Initialize the set $\mathcal{S}$ of silhouette edges**:

  Given an object A, find the silhouette edges by trying all edges of A : if the one face incident on the edge is front facing while the other is back facing, then the edge is a silhouette edge.

- **Initialize the set $\mathcal{O}$ of outline edges**:

  Given the silhouette edges of A, we use a variant of the line-sweep algorithm to determine both the outline and the silhouette overlappings $C(e)$ for each silhouette edge e.

  Note that the silhouette of A is not necessarily separated from the view point by a plane, so there is no image plane that we can project onto. Instead, we project the silhouettes onto the image sphere, changing the coordinate system into an angular one. The projections of the silhouettes partition the image sphere into regions. In each region there is a unique number of layers of the object[1].

The algorithm sweeps a longitudinal line across the image sphere, determining the number of layers for each region. At the same time, it adds the edges which separate regions of layer 0 from regions of layer 1 to the actual outline. The algorithm is depicted in Figure 6.1. To avoid problems, we first make sure that both the north pole and the south pole do not intersect any silhouette edges and the initial longitude (at $\phi = 0$) do not pass through any silhouette vertices or the intersections of two silhouette edges. If they do, we can rotate the silhouette around.

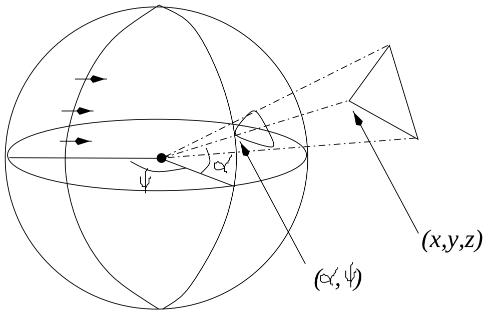

FIGURE 6.1 *The line sweep variant to determine the actual outline.*

Note that the silhouette itself does not provide enough information about the actual number of layers present at every point—consider a hollow spherical object with the view point in the center, this has no silhouettes nor outlines. However given the number of layers at a certain point, the silhouettes convey enough information to propagate this information to the whole image sphere. Hence to initialize the line-sweep, a ray is shot in a direction with $\phi = 0$ to determine the actual number of layers there. Then this information is propagated along the sweep line—e.g. if we pass from the inside to the outside of a convex silhouette edge, the number of layers is decremented by 1 (note that the *reverse* is true for *concave* edges).

[1]Given a point on the image sphere, if we shoot a ray from the viewpoint through that point, the ray is going to penetrate A a number of times. We call the number of penetrations the number of layers of A at that point.

At every point in time the sweep line stores the silhouette edges which currently overlap it and the number of layers at each region that the sweep line intersects.

As the line sweep proceeds it will cross silhouette vertices. This is depicted in Figure 6.2. The small arrows of each edge denotes the side of the edge that has one more layer (i.e. the inside of a convex edge and the outside of a concave edge). In (a), before crossing the vertex the sweep line keeps track of the number of layers at each region to the left of the vertex. After crossing, in (b), the silhouette edges incident on the vertex is visited in a clockwise manner. Consider visiting edge e. The region A before f has 0 layer and is on the side of f having less layers (the small arrow). Hence the next region, B, has $0 + 1 = 1$ layers. Now notice e is a silhouette edge separating a region of 0 layers from a region of 1 layer, so e is on the outline. The algorithm then proceeds to edge f.

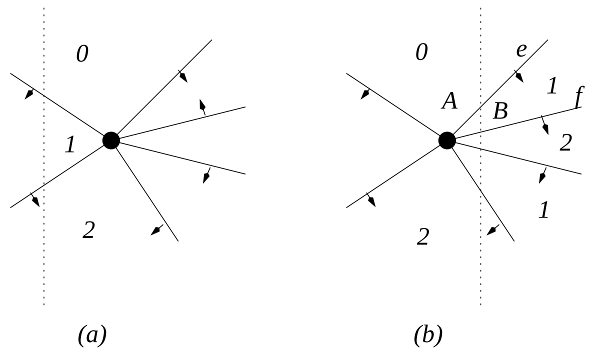

FIGURE 6.2 *The sweep line encounters a silhouette vertex.*

The sweep line will also encounter silhouette overlaps as in Figure 6.3. In this case we can determine the number of layers at the region to the right either from the region at the top, with 1 layer, and from the orientation of the silhouette edge e, which indicates that the region at the right has one more layer than the region at the top, i.e. 2.

After the algorithm, we would have determined both the actual outline of the object, and the silhouette overlappings. To get the conservative outline, we just remove appropriate edges from the actual outline until the property described in subsection 4.4 is satisfied, and then add in those edges in the interior which can potentially create holes in the actual outline after EEE events.

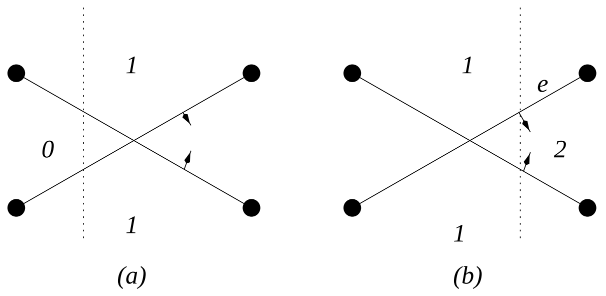

FIGURE 6.3 *The sweep line encounters a silhouette overlap.*

- **Initialize the set $\mathcal{T}$ of touch planes**:

  Given a plane $\langle V, e\rangle$, determining whether it is a touch plane is not so easy. We propose a *conservative* test which could overestimate the set of touch planes; however, no true touch plane will be missed by the test.

  For $\langle V, e\rangle$ to be a touch plane, the two faces incident to e must lie on the same side of $\langle V, e\rangle$. Now let $e_1, \ldots, e_k$ be the edges incident to V, in that order. If all these edges lie on the same side of $\langle V, e\rangle$, then $\langle V, e\rangle$ is surely a touch plane; this corresponds to the configuration shown in Figure 6.4(a). Otherwise, we traverse the edges $e_1, \ldots, e_k$ in order, recording on which side of $\langle V, e\rangle$ each edge e_i lies. If there are more than *two* "switching of sides", then we identify $\langle V, e\rangle$ as a candidate for touch planes. It is not hard to see that if $\langle V, e\rangle$ is indeed a touch plane of the type shown in Figure 6.4(b), then there should be at least two "switching of sides" in our traverse of the edges $e_1, \ldots, e_k$. The converse is not necessarily true.

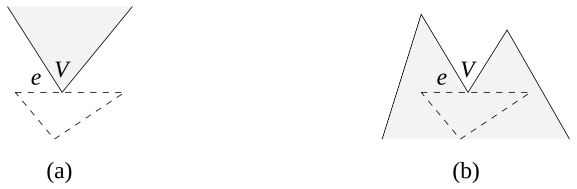

FIGURE 6.4 *Two possible configurations of a touch plane.*

All touch planes are predetermined in the initialization. A table look-up is used to locate the set of touch planes involving an outline (or silhouette) vertex or edge.

- **Initialize the set $\mathcal{R}$ of relevant planes**:

  The initial set of silhouette faces is the set of faces with an edge in the silhouette. This can be computed along with the initial silhouette computation.

  The saddle faces are computer by iterating over all faces of all the objects. If a face as a convex edge and a concave edge, it is a saddle face.

  The relevant touch faces between an occluder and occludee is simply the set of touch faces $\langle V, e \rangle$ with both V and e either in the silhouette of the occludee or the conservative outline of the occluder. The relevant touch faces on an occluder used to maintain the outline are the touch faces $\langle V, e \rangle$ with both V and e on the silhouette of the occluder.

  The relevant sliding events as well as the silhouette overlappings on a occluder can be performed along with the conservative outline computation. the relevant sliding events and silhouette overlappings between an occluder and occludee can be computed using a similar sphere-sweep algorithm as that used for initializing the outlines - we simply have to detect the silhouette overlappings involving an edge on the occluder and an edge on the occludee.

  As for the covering and piercing events, a brute force approach is required. We iterate over all edges and faces on the occludees and all outline edges and vertices on the occluders. If an outline vertex is in a face of an occludee, we add the relevant piercing events; if an outline edge crosses an edge of an occludee, we add the relevant covering events.

  A similar problem occurs when a hole suddenly appears on the outline, for example the first problem above. Figure 4.4. As the hole does not exist on the outline before, we cannot use an incremental approach to update the relevant planes (eg, covering and piercing planes). Hence we will have to use a brute force method to 'initialize' the relevant planes correctly.

6.3 Detection of plane crossings

After each incremental change of the viewpoint, we have to detect whether the viewpoint crossed a relevant plane. If the viewpoint crosses a plane then a number of planes are inserted or deleted from the relevant plane set. The relevant plane set can be quite large so maintain it simply as an unstructured set is unacceptable, since for every viewpoint change we need to check the new viewpoint against all relevant planes individually. On the other hand we can construct an arrangement of the relevant planes. Then detecting plane crossings will be efficient but insertions and deletions will incur large costs. Further, the arrangement requires storage space of the order of p^3 where p is the number of relevant planes.

Coorg and Teller [1] described two methods to maintain the relevant plane set :

- Near planes

  In addition to storing all the relevant planes, store a set of em near planes which intersect a sphere of radius r containing the viewpoint. Then while the viewpoint is still in the sphere, we only need to check whether the viewpoint crossed the near planes. On the other hand insertions and deletions will only take constant time. If the viewpoint goes out of the sphere, then a new set of near planes intersecting a sphere of radius r centered at the new viewpoint is generated from the set of all relevant planes and the viewpoint is checked against the new near planes. Assuming that the viewpoint moves at most a constant distance and the number of near planes is proportional to r, the algorithm takes $O(\sqrt{p})$ expected time.

- Hierarchical space partitioning

  A generalization of the previous algorithm is to maintain the planes in a hierarchy. Suppose we have a hierarchical space partitioning. Each node of the hierarchy corresponds to a portion of space and stores the set of planes intersecting that portion of space. When the viewpoint moves from V to W, it is checked against the set of planes stored at the lowest node in the hierarchy which contains both V and W. The expected runtime for checking for plane crossings is $O(\log p)$, while insertions and deletions are $O(1)$.

Unfortunately, both these algorithms have worst case runtime of $O(p)$. Furthermore, while the algorithms are fast most of the time, once in a while the algorithms will take $O(p)$ time to complete (when reconstructing the near planes, or when the lowest node is the root itself). This means that in an interactive walk-through, once in a while the computer will seem to be especially slow. This is more distracting to the user than an algorithm that performs moderately fast all the time.

6.4 Updating visibility and relevant planes

After we detect a relevant plane crossed by the viewpoint, a number of updates need to be performed depending on the type of the plane, as shown in section 2.6. In this section we shall only describe the algorithm when a piercing event occurs. The rest will be standard but technical.

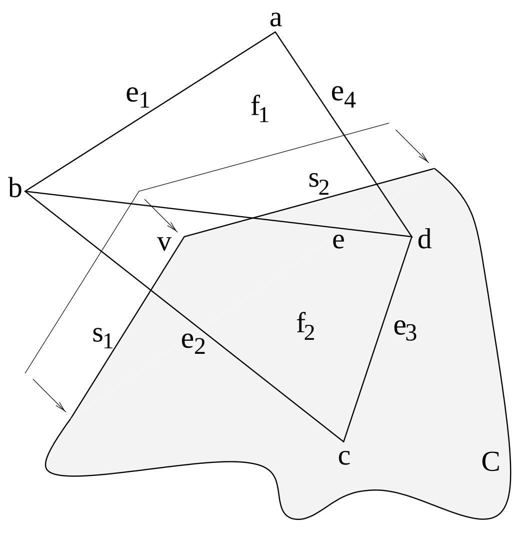

FIGURE 6.5 *A Piercing event occurs with silhouette vertex V crossing edge e.*

Suppose a piercing event occurs between edge e on the occludee and outline silhouette vertex V on the occluder C, as shown in figure 6.5. Since V has moved from face f_1 to face f_2, both outline edges s_1 and s_2 cross face f_1 before the event, and cross face f_2 after the event. The following changes occur :

- If e_1 is not on the silhouette of the occludee then remove $\langle e_1, V\rangle$ from the set of piercing events.
- If e_4 is not on the silhouette of the occludee then remove $\langle e_4, V\rangle$ from the set of piercing events.
- If e_2 is not on the silhouette of the occludee then add $\langle e_2, V\rangle$ to the set of piercing events.
- If e_3 is not on the silhouette of the occludee then add $\langle e_3, V\rangle$ to the set of piercing events.
- for $s \in \{s_1, s_2\}$,
  - Decrement $\mathcal{N}(f_1, C)$ by 1 and increment $\mathcal{N}(f_2, C)$ by 1.
  - if s does not intersect f_1 after the event, then decrement $\mathcal{N}(f_1, C)$ by 1 and remove $\langle s, a\rangle$ from the covering events.
  - if s did not intersect f_2 before the event, then increment $\mathcal{N}(f_2, C)$ by 1 and add $\langle s, c\rangle$ to the covering events.
- if after the event $\mathcal{N}(f_1, C) = 0$ and f_1 is on the inside of s_1, then increment the number of occluders totally occluding f_1 by 1. If f_1 was not occluded before then remove f_1 from the visible triangles.
- if before the event $\mathcal{N}(f_2, C) = 0$ and f_2 was on the inside of s_1, then decrement the number of occluders totally occluding f_2 by 1. If there are no more occluders occluding f_2 then add f_2 to $\mathcal{V}$.

6.5 Efficiency heuristics

We offer some efficiency heuristics which can speed up our visibility algorithm. Some of these heuristics are fairly standard.

6.5.1 Occludees as bounding volumes

When a group of small and close-by objects are *far away* from the view point, it makes sense to form a tight bounding volume, e.g., a cube, around them, and simply perform visibility computations on the bounding volume, instead of on the individual objects. This is because that such a group of objects is more likely to be either completely occluded by occluders close to the view point or completely visible. If the volume is deemed visible, all objects in the group are rendered; otherwise, all objects are culled away. This can be especially beneficial computationally when we are dealing with a dense scene consisting of many small objects.

There are several criteria for grouping the objects. First of all, objects to be grouped should be small. Secondly, we do not want a group to contain too many objects, since when the viewpoint moves closer to the group, the bounding volume should be "taken off", and the set of relevant planes corresponding to the objects in the group need to be computed; we do not want this computation to take too much time. And finally, the objects to be grouped should be close-by.

6.5.2 Hierarchy of bounding volumes

To further speed up our algorithm, we can extend the idea of using bounding volumes to constructing a hierarchy of bounding volumes. This is similar to the use of an *kd*-tree structure in Coorg and Teller [2], and to the approach used by Zhang et al. [7] in their visibility culling algorithm.

6.5.3 Grid partitioning of 3D space for overlapping tests

Quite often, our algorithm will need to perform overlapping tests between edges in the scene. For example, when a new occluder is added, or when new silhouette or outline edges emerge, we need to test for overlappings between the new silhouette or outline edges against the existing set of silhouette edges.

An exhaustive search may be too much of a slow-down. What we can do is to partition the 3D space into uniform grids. At each grid, we store the set of silhouette edges which intersect the grid. Knowing the grid containing the current viewpoint, and the set of adjacent grids containing a new silhouette edge e, we can locate those grids which can contain an edge overlapping e quickly. Then we need to test overlappings involving e against only the silhouette edges intersecting those grids.

7 Conclusion and future work

We have developed an incremental conservative visibility algorithm which works in an environment consisting of complex occluders. We take advantage of the spatial and temporal coherence between successive frames to achieve interactive update of visibility changes.

7.1 Compare to hierarchical occlusion map

Zhang et al. [7] develop a visibility culling algorithm combining the use of an object space bounding volume hierarchy and an image space *hierarchical occlusion maps*. An occlusion map is the combined image of a set of selected occluders. The hierarchical occlusion map is built by recursively filtering from the highest-resolution map down to some basic map. The filtering process is an *averaging* of the opacity values in adjacent rectangular blocks of pixels.

The visibility status of an occludee is determined by first doing an "overlapping" test of the occludee against the hierarchy of occlusion maps, and then a depth test against a *depth estimation buffer*. Their algorithm is also of a conservative nature. The efficiency of their algorithm is mainly attributed to the hierarchical representation of the occlusion map, which facilitates conservative and early termination of the overlapping tests.

It is worth noting that their algorithm does not exploit the temporal coherence between successive frames, except in the selection of occluders. At each frame, the occlusion map and its hierarchy are reconstructed, and the overlapping and depth tests are performed again for each occludee. So the number of primitives processed at each step is still large.

7.2 Multi-layered silhouette representation

Recall that we only consider visual events corresponding to vertices or edges on the outlines of the occluders. We made this choice solely out of efficiency concerns. Those silhouette edges not on the outline do cause occlusion, as shown in Figure 4.2. Even concave silhouette edges, which can never be visible, play an important role if we are to distinguish among various ways an occluder can occlude an occludee.

Consider the situation illustrated in Figure 7.1. Suppose that the viewpoint has just crossed the plane $\langle V, e\rangle$ from left to right. Before this occurs, the face f was occluded by just one *layer* of object A, meaning that any ray shot from the viewpoint into f intersects A exactly once. After the crossing however, part of f is occluded by two layers of A, since a ray shot from the viewpoint to vertex V intersects A twice.

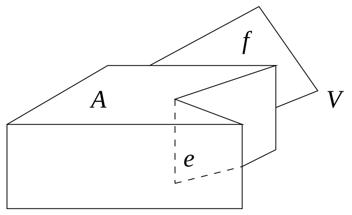

FIGURE 7.1 *Occlusion by multiple layers.*

For our algorithm to work, we made an assumption in section 4.1 that give any occluder and any occludee, there is a plane that separated the two. this is required because otherwise, the outline of the occluder is not sufficient in determining the occlusion information of the occludee. This is depicted in Figure 4.2, where the occludee G is occluded, although it lies in front of the outline edges on K.

A multi-layered approach which uses the silhouette directly to compute visibility can obviate the need for this assumption. Consider Figure 7.2. We can determine that G is totally occluded by a layer of the object whose silhouette is E, i.e. L. We can also determine that L partially occluded K since its silhouette E is in front of K.

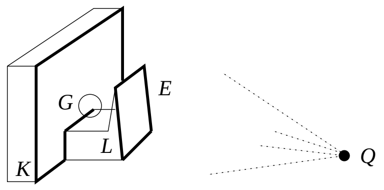

FIGURE 7.2 *Multi-layered occlusion. Thick edges are silhouette edges.*

An incremental visibility algorithm using multi-layered silhouettes will be very similar to the algorithm we presented. In fact we can view the outline approach as replacing each occluder by a potentially much simpler object—

one which has only one layer, and whose silhouette coincides with the outline of the original occluder. In the multi-layered approach, we can view each occluder as being decomposed into a number of distinct layers each of which functions as a distinct occluder which occludes objects individually. The question is how to perform this decomposition correctly. This is a question for future research.

7.3 Future work

Implementation.

References

[1] S. Coorg and S. Teller, "Temporally Coherent Conservative Visibility", In *Proc. 12th Annual ACM Symposium on Computational Geometry,* Philadelphia, PA., May 24-26, 1996.

[2] S. Coorg and S. Teller, "Real-time Occlusion Culling for Models with Large Occluders," *In Proc. 1997 ACM Symposium on Interactive 3D Graphics,* (1997), 83-90.

[3] Z. Gigus, J. Canny, and R. Seidel, "Efficiently Computing and Representing Aspect Graphs of Polyhedral Objects," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 13, 6 (1991), 542-551.

[4] H. Hubschman and S. W. Zucker, "Frame to Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World," *ACM Transactions on Graphics (USA),* 1, (1982), 129-162.

[5] L. Kettner and E. Welzl, "Contour Edge Analysis for Polyhedron Projections," Technical Report, ETH Zürich, Institute of Theoretical Computer Science, 1996.

[6] W. Plantinga and C. Dyer, "Visibility, Occlusion, and the Aspect Graph," *Int. J. Computer Vision,* 5, 2 (1990), 137-160.

[7] H. S. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III, "Visibility Culling using Hierarchical Occlusion Maps," *Computer Graphics Proceedings, SIGGRAPH Annual Conference,*, 1997.

8 Appendix

8.1 Proof of Theorem 1

Proof: Let f be the first face crossed by the view point. Let e_1, e_2, and e_3 be the three edges of f. Suppose that f is neither a silhouette face nor a saddle face. Then none of e_1, e_2, and e_3 is a silhouette edge.

1. f is forward-facing with respect to Q:

   Then e_1, e_2, and e_3 are all forward-facing with respect to Q. If e_1, e_2, and e_3 are all convex edges, then we must have something like what is shown in Figure 8.1(a), where f_1, f_2, and f_3 are the planes containing the face, other than f, incident to e_1, e_2, and e_3, respectively. Position Q must be inside the region bounded by planes f_1, f_2, and f_3, as shown. Clearly, the view point cannot cross f before crossing f_1, f_2, or f_3, contradicting our assumption that f is the first face crossed.

   If e_1, e_2, and e_3 are all concave edges, then we must have something like what is shown in Figure 8.1(b), with relevant planes labelled as before. Again, we see that Q must be inside the region bounded by planes f_1, f_2, f_3, and f, as shown. Now for the view point to cross f before crossing any of f_1, f_2, or f_3, it must have penetrated face f, which is not allowed.

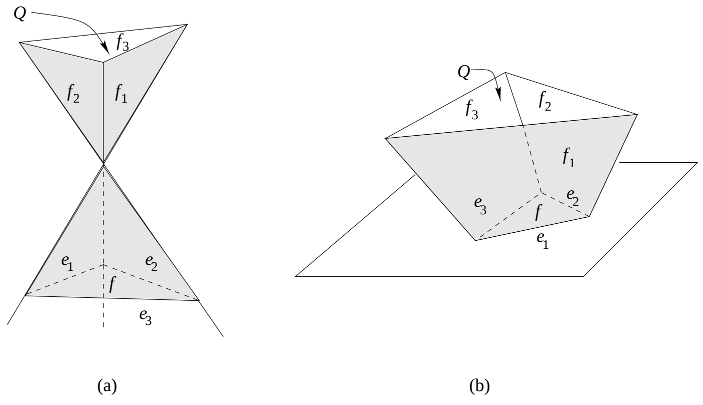

FIGURE 8.1 *All faces forward-facing, all edges convex or concave.*

2. The case where f is back-facing with respect to Q is symmetric. □