Real-Time Hidden-Line Elimination for a Rotating Scene*

Harry Plantinga
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
planting@cs.pitt.edu

W. Brent Seales
Department of Computer Science
University of Kentucky

Charles R. Dyer
Department of Computer Science
University of Wisconsin

* The support of the National Science Foundation under Grants IRI-8802436 and CCR-9007612 and the University of Pittsburgh Central Research Development Fund is gratefully acknowledged.

1. Introduction

The three-dimensional structure of an object is easier to perceive when the object is rotating or when the viewer moves to see it from a range of viewpoints. Thus, CAD systems are usually able to display an object as it rotates. Topologists may more easily perceive the structure of a polyhedral model of a knot and biologists the structure of a molecule using such rotations. In problems such as these, real-time motion and hidden-line or hidden-surface removal for polyhedral scenes are both desirable goals, but achieving both simultaneously for large models on inexpensive hardware is difficult.

In this paper, we discuss the problem of displaying from a moving viewpoint a series of line-drawing images of a polyhedral object or scene with hidden lines removed. We will refer to this as the problem of animating rotation. We present an algorithm for animating the rotation of a polyhedral scene that displays frames at a rate roughly equivalent to the rate at which the same hardware could display line-drawings of the scene without hidden-line removal. We consider here only the case of a rotation about one axis of the coordinate system under orthographic projection.

The naive approach to animating rotation is to treat frames independently. For each frame, hidden lines are removed and the frame is displayed. The real-time goal is to do this at video rates, such as 15 or 30 frames per second. However, since the viewpoints are closely spaced, there will be little difference between two successive frames and repeated hidden-line removal may perform much redundant work. This *frame coherence* makes it possible to devise a more efficient algorithm.

The algorithm presented here takes advantage of frame coherence by computing the initial appearance of the scene in the first frame and the viewpoints at which the topological structure of the image changes, which are called *events*. The event viewpoints are computed through the construction of the *aspect representation* for the scene, a representation that makes explicit the set of vertices, edges, and faces visible from every viewpoint. The algorithm has two phases: a preprocessing phase, in which the initial appearance of the polyhedron and the events are computed and a list of visible edges is constructed; and an on-line phase, in which a sequence of frames is displayed in real time.

The approach presented here is an object-space, vector-based approach and does not use raster-based algorithms such as Z-buffer. This is a disadvantage where realistic rendering is desired and appropriate hardware is available. However, it also presents certain advantages. The display of vectors is often faster than the display of raster-based images, and the algorithm presented here does not use a Z-buffer, so this method may be used to achieve real-time animations with inexpensive hardware. Also, there are no existing algorithms using raster-based methods for precomputing a relatively small amount of data—much less than the sum of the sizes of all the views—that represent each view in an animation sequence exactly (although this algorithm may also be used for that purpose). It should also be pointed out that since the edges bounding polygons are available, this technique may also be used in conjunction with more realistic lighting and shading techniques, but in that case the real-time capability is likely to be lost.

The algorithm presented here is useful for some applications, such as the visualization of knots or molecules or other small geometric objects on inexpensive hardware. Also, we believe that the technique is interesting in its own right. It may be generalized to accommodate other types of viewer paths or greater freedom of motion for the viewer, and it may even be possible to modify this technique to improve the performance of animations when a Z-buffer is available. Finally, since processor speeds and memory capacities are growing at a much faster rate than memory bandwidth, and since memory bandwidth is a major bottleneck in raster graphics performance [1], object-space techniques that improve rendering efficiency are increasing in applicability and importance.

The aspect representation was introduced by Plantinga and Dyer [2] in the context of constructing the aspect graph for a polyhedron. It also appears in other work [3-5]. This approach to animating rotation was first introduced by Plantinga [3] and Plantinga *et al.* [5]. Section 2 of this

paper discusses frame coherence and the sorts of events that occur. Section 3 shows how to compute and store these events, and Section 4 describes how to use this information to animate rotation. Section 5 describes the implementation and results, and Section 6 discusses related work.

2. Coherence and Events

We restrict our discussion to scenes consisting of non-intersecting faces (except at their boundaries) and to the rotation of the scene about the world coordinate system *y*-axis (see Figure 1), viewed under orthographic projection. Other rotations are handled by a coordinate transformation. Perspective projection can be handled with relatively minor modifications [3,4]. We will say that a vertex and an edge *appear to intersect* from a given viewpoint when their projections are visible and intersect in an image from that viewpoint. We will refer to the apparent intersection point of two edges in an image (i.e. the image point at which they appear to intersect) as a *T-junction.* Thus, edges that appear to intersect actually intersect in the projection but do not necessarily intersect in 3-space.

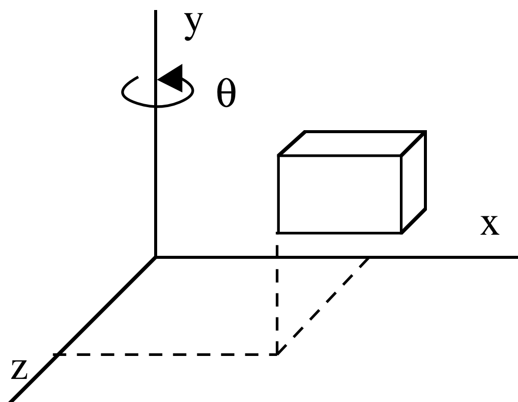

Figure 1. The viewing model.

Frame coherence means that images from nearby viewpoints are "similar." We exploit this similarity by representing the topology of the image with a graph that represents the planar subdivision induced by the edges in the image, so that the image can be rendered from the information in the graph. Then, to progress from frame to frame, we must discover the changes in the topology of the image and modify the graph appropriately. This graph will be called the *Edge Structure Graph* (ESG) of an image from a given viewpoint. The ESG from a viewpoint is the graph with edges and vertices corresponding to edge segments and intersection points of edge segments, respectively, visible in an image from that viewpoint. In addition, for each end of an edge in the ESG, we store a pointer to the (3D) endpoint of the corresponding object edge if the image edge does not end in a T-junction. If it does end in a T-junction, we store a pointer to the occluding edge (see Figure 2).

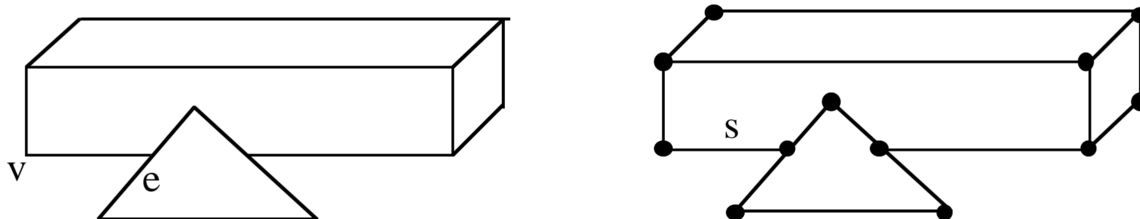

Figure 2. The image on the left is represented by the graph on the right. The endpoints of the image segment s are represented by pointers to vertex v and occluding edge e of the scene.

As the viewpoint changes between two values or, more generally, traces out a path in viewpoint space, the appearance of the object changes. If the change in viewpoint is small

enough, the topology of the image (and hence the ESG) may remain constant, even though the actual images rendered will differ due to the different viewpoint. However, some changes in viewpoint will change the topology of the image—when, for example, a face disappears from view. In that case the ESG changes. We must discover the viewpoints that have the property that an arbitrarily small change in viewpoint along the path will result in a change in the ESG. We call these viewpoints *events* after the visual events of Koenderink and van Doorn [6].

In order for a viewpoint to be an event, one of two situations must obtain: from that viewpoint either a scene vertex and a non-adjacent edge must appear to intersect (*EV-event*) or three scene edges must appear to intersect (*EEE-event*). To see that this is true, suppose that a given viewpoint is an event and that no non-adjacent vertex-edge pair appears to intersect. There is by definition a change in the ESG at that viewpoint, and it must occur at an image vertex. Image vertices are a result of object vertices or T-junctions, and if the T-junction is the apparent intersection of only two object edges, the structure doesn't change with a sufficiently small change in viewpoint. Since the image structure does change, at least three object edges must appear to intersect somewhere in the image.

Now suppose that an object vertex and non-adjacent edge appear to intersect from a given viewpoint. Unless the object edge is parallel to the plane of rotation, rotating slightly in some direction will separate the edge and the vertex, resulting in one of the following cases. If the vertex is in front of the edge from the given viewpoint, one of the events shown in Figure 3 occurs. If the vertex is behind the edge from the given viewpoint, one of the events shown in Figure 4 occurs. Suppose that three object edges appear to intersect. Let the first and second be arbitrary. Depending on the location and orientation of the third, one of the events shown in Figure 5 occurs. If more edges or vertices meet at a viewpoint, the event can be treated as two more events.

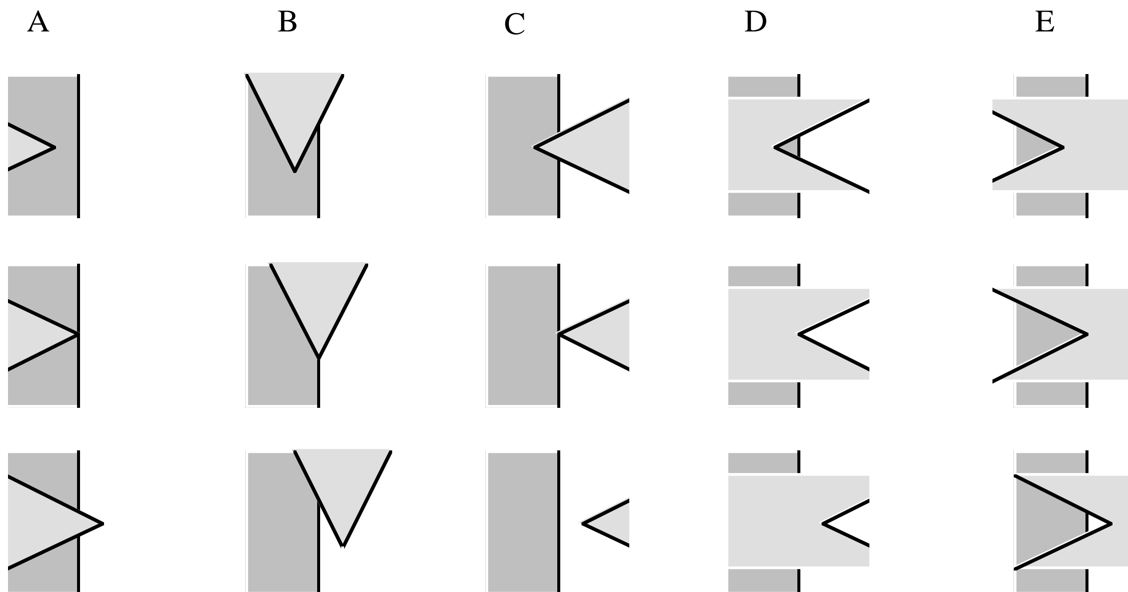

Figure 3. EV events with the vertex in front of the edge. In each column, the middle figure shows the image from the event viewpoint and the top and bottom figures show the appearance from just to the right and to the left.

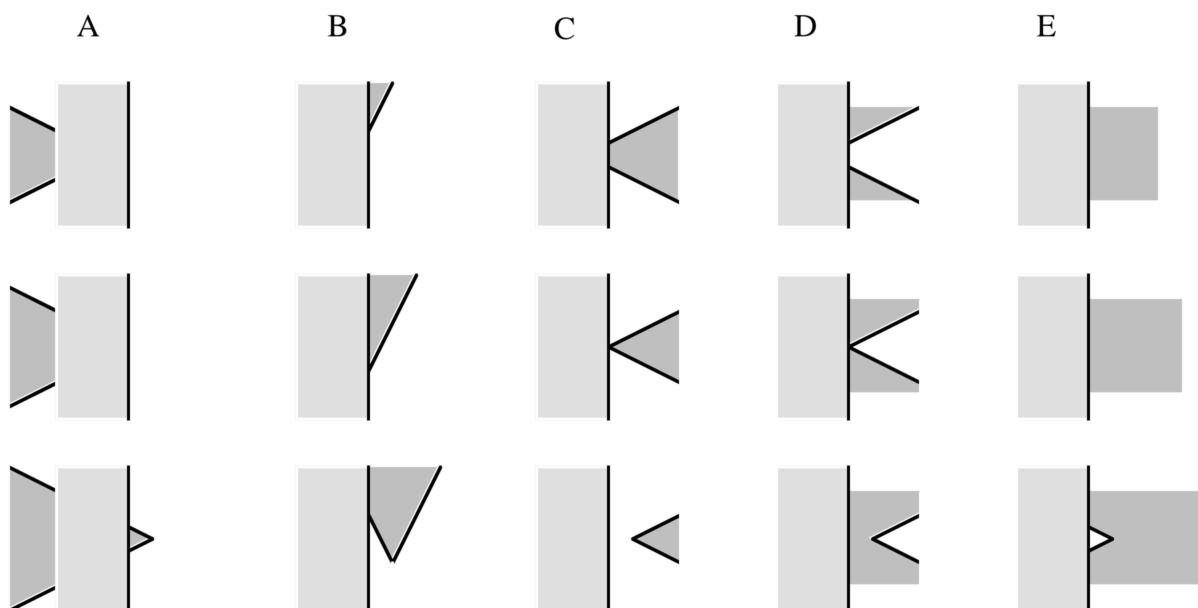

Figure 4. EV events with the vertex behind the edge.

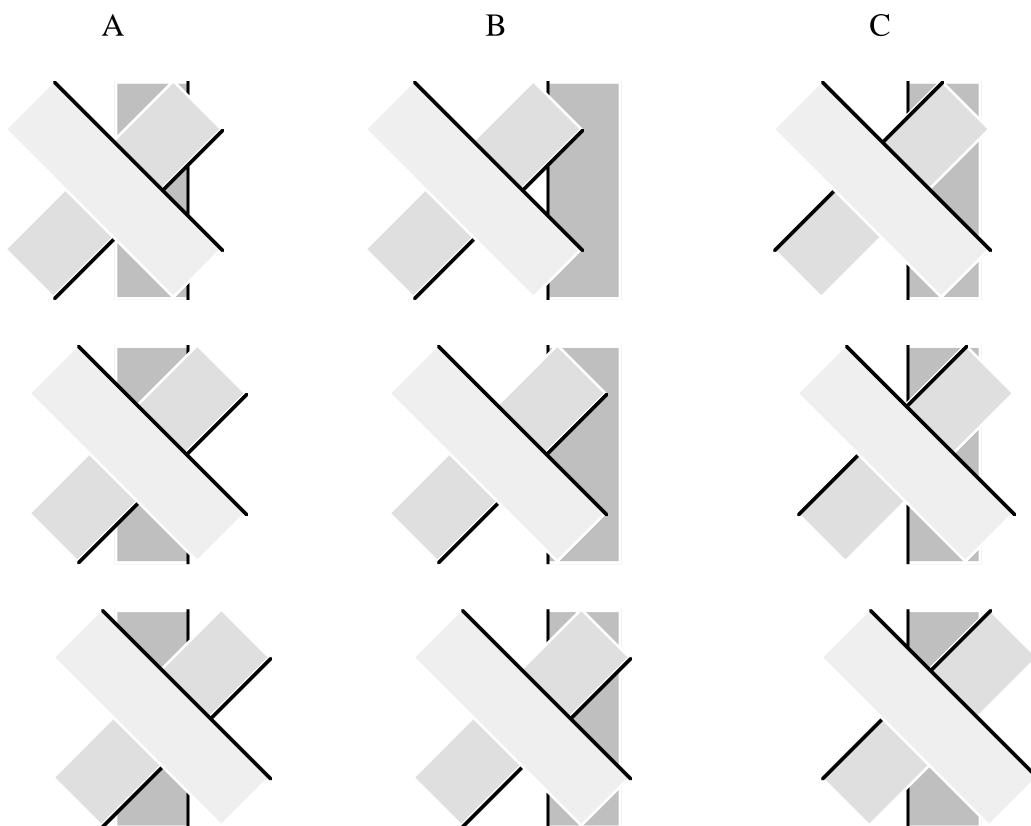

Figure 5. EEE events.

3. Computing Events

In order to animate rotation, the viewpoints at which topological changes in the ESG occur must be computed, and the change that occurs must be represented. Since events occur at viewpoints from which an edge and a vertex or three edges appear to intersect, it suffices to take

every edge-vertex pair and every set of three edges from the model and find the viewpoints from which they appear to intersect. If any of these viewpoints is on the viewer's path, then that viewpoint may be an event. Thus, it would be sufficient to find the locus of points from which all edge-vertex pairs and all sets of three edges appear to intersect. In the case of an edge and a vertex, the locus of apparent intersection points is a polygonal region of a plane, and in the case of three edges, it is a not a plane but a warped surface, as we shall see below. We call the viewpoints at which the edge and vertex or three edges appear to intersect in the absence of other occluding faces *potential events*; they are events if the associated change is visible in the image. It remains to determine whether the apparent intersection point is visible in the image. This could be computed for each potential event individually. However, it is usually true that most of the potential events are not visible; thus, this is an inefficient approach. We now describe a more efficient approach.

In order to represent what is visible from each viewpoint, a representation based only on viewpoint space is insufficient—it is necessary in some sense to represent all viewing directions from each viewpoint. The approach we take is to define *aspect space* to be image space × viewpoint space. In the present case, in which the rotation is about a single axis, we can consider viewpoint space to be $\mathbf{S}^1$ (a circle). Since image space is a plane, aspect space is $\mathbf{R}^2 \times \mathbf{S}^1$. We can represent this as a subspace of $\mathbf{R}^3$, specifically $\mathbf{R}^2 \times (-\pi,\pi]$. A point in aspect space then corresponds to a point in the image plane from a particular viewpoint. Thus, a point (x_0, y_0, z_0) in object space is represented by a 1-D locus of points in aspect space since it is visible from all viewpoints. This locus can be computed from the equations for the location of the point in image space after a rotation by θ about the y-axis. Denoting coordinates in the image plane (u,v), the result is

$$u = x_0 \cos\theta - z_0 \sin\theta \tag{1}$$

$$v = y_0 \tag{2}$$

Thus, a point in object space is represented in aspect space by the curve segment $(u(\theta),v)$, $-\pi < \theta \leq \pi$.

An edge of the scene connecting vertices $\mathbf{p}_1 = (x_1, y_1, z_1)$ and $\mathbf{p}_1 + \mathbf{a}_1 = (x_1 + a_1, y_1 + b_1, z_1 + c_1)$ can be represented parametrically as $\mathbf{p}(s) = \mathbf{p}_1 + s\,\mathbf{a}_1$, $0 \leq s \leq 1$. It appears in the image at the points

$$u = (x_1 + a_1 s) \cos\theta - (z_1 + c_1 s) \sin\theta \tag{3}$$

$$v = y_1 + b_1 s \tag{4}$$

Thus, an edge in object space is represented in aspect space by the 2-D locus of points $(u(s,\theta),v(s))$.

We are interested in computing the appearance of an object face from all viewpoints. If the face is not occluded from any viewpoint, it is represented as a volume of aspect space. We call that volume the *aspect representation* or *asp* for the face. The asp for a polygon is a volume bounded by surfaces of the form of Eqs. (3) and (4) and edges of the form of Eqs. (1) and (2). Figure 6 shows a part of the asp for a triangle in the xy-plane. Note that a cross section of the asp with $\theta = c$ is the appearance of the polygon from viewpoint $\theta = c$.

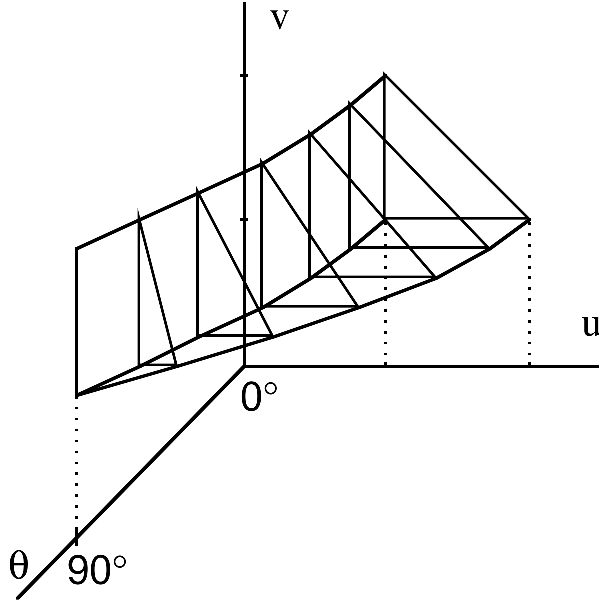

Figure 6. The asp for a triangle.

The asp for a face is represented much as a polyhedron would be represented. The difference is that surfaces are not planar. Therefore the parametric equations for each surface and curve bounding the volume are also represented.

If the triangle is occluded by other faces from some viewpoints, its appearance from all viewpoints may be represented as some subset of the volume shown in Figure 6. We would like to determine the appropriate volumes to subtract for each occluding face. To do so, we use the following property of occlusion. We assume that the faces of the scene are oriented, with the *front* of the face visible and the *rear* of the face invisible. We will say that a face occludes another face when there is a line of sight that passes through the front of the occluding face and hits the front of the occluded face. Let **f** be an oriented face of a scene. Let $\mathbf{P_f}$ be the plane containing **f**, and let the halfspace with respect to $\mathbf{P_f}$ on the front side of **f** be $\mathbf{P_f}^+$ and the other halfspace $\mathbf{P_f}^-$. The faces that **f** occludes from some viewpoint are precisely the faces (or parts of faces) in $\mathbf{P_f}^-$, and no face or part of a face in $\mathbf{P_f}^-$ occludes **f** from any viewpoint.

The algorithm for computing the events rests on the following observation: the appearance of a face **f** *as occluded by* the faces in $\mathbf{P_f}^+$ is characterized by the subtraction of the asps for the occluding faces from the asp for **f**. This is true since a point in aspect space that is a part of the asp for **f** and the asp for an occluding face corresponds to a viewpoint and an image point such that both faces are visible at that image point from that viewpoint in the absence of occlusion. Since the occluding face is in front of **f** in object space, it occludes **f** at that viewpoint.

We initially compute the asp for each face of the scene in the absence of occlusion. Then, to construct the asp for a face **f** as occluded by the other faces, the asp for each face in front of **f** is subtracted from the asp for **f**. (If a face intersects $\mathbf{P_f}$**,** the asp for the part of the face in $\mathbf{P_f}^+$ is subtracted.) The remaining volume of the asp for **f** consists of points corresponding to image points and viewpoints at which the face **f** is not occluded, i.e. appears in the image. In order to find the intersection of these volumes bounded by non-planar surfaces, we must know how to find the intersection of the surfaces. Then finding the intersections of asps will be similar to finding the intersections of polyhedra. The intersection of two of these surfaces is solved for in closed form below.

A volume of aspect space thus corresponds to the appearance of a polygon in the image from a range of viewpoints; the cross-section of that volume for a given value of θ is the appearance of the polygon from viewpoint θ. Similarly, such a volume is bounded by surfaces corresponding to edges in the image from a range of viewpoints. The surfaces are of the form given by Eqs. (3) and (4) above. The intersection of two of these surfaces corresponds to the apparent

intersection of two edges in an image, that is, a T-junction. If the two edges are $\mathbf{p}_1 + s_1\mathbf{a}_1$ and $\mathbf{p}_2 + s_2\,\mathbf{a}_2$, $0 \le s_1, s_2 \le 1$, solving for s_2 as a function of θ and for s_1 as a function of s_2 yields

$$s_1 = \frac{\langle \tan\theta \quad 0 \quad 1 \rangle \bullet \left((\mathbf{p}_2 - \mathbf{p}_1) \times \mathbf{a}_2 \right)}{\langle \tan\theta \quad 0 \quad 1 \rangle \bullet (\mathbf{a}_1 \times \mathbf{a}_2)} \tag{5}$$

Thus, the parameter s_1 is the ratio of linear functions of $\tan\theta$. Note that $\langle \tan\theta\ 0\ 1 \rangle$ is a vector in the viewing direction. Since then length of that vector does not affect the result, it may be replaced by the viewing vector, yielding an expression that is also defined at $\theta = 90°$ and $\theta = -90°$. Calling the viewpoint $\mathbf{v}$, we get

$$s_1 = \frac{\mathbf{v} \bullet ((\mathbf{p}_2 - \mathbf{p}_1) \times \mathbf{a}_2)}{\mathbf{v} \bullet (\mathbf{a}_1 \times \mathbf{a}_2)} \tag{5a}$$

Substituting this expression into Eqs. (3) and (4) yields the image point at which the edges appear to intersect as a function of θ. Thus, $(u(\theta), v(\theta))$ as given by Eqs. 3-5a is a parametric representation of the curve in aspect space corresponding to the intersection of two asp surfaces, and it is the general form for the curved "edges" or 1-D boundaries of the asp.

EEE visual events result from the apparent intersection of three object edges. Three edges can appear to intersect in a single point from at most two viewpoints and their polar opposites along the path. The viewpoints can be found by noting that if the line of sight passes through the point $\mathbf{p}_1 + s_1\mathbf{a}_1$ and the other two edges, it must lie in the plane containing that point and the second edge, and it must also lie in the plane containing that point and the third edge. Therefore it lies in the intersection of those two planes (see Figure 7).

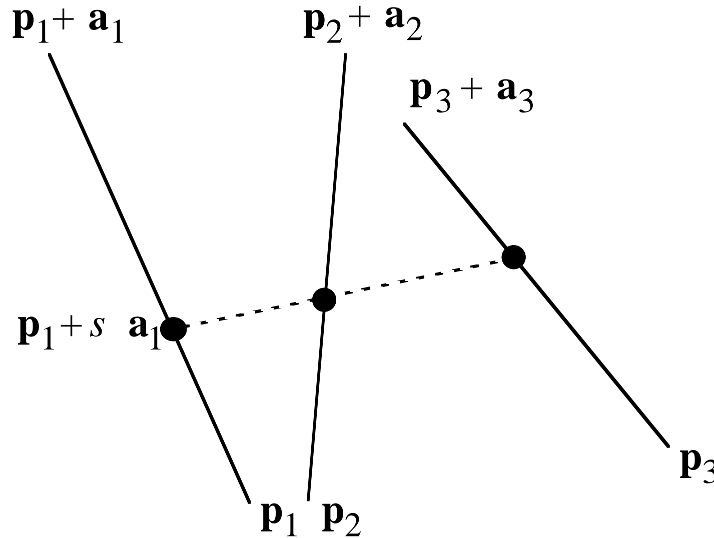

Figure 7. The viewpoints from which three object edges appear to intersect in a single image point.

Therefore, we can find the direction of the line of sight from which these edges appear to intersect by finding a vector parallel to the intersection of those two planes:

$$\mathbf{d} = ((\mathbf{p}_1 + s\,\mathbf{a}_1 - \mathbf{p}_2) \times \mathbf{a}_2) \times ((\mathbf{p}_1 + s\,\mathbf{a}_1 - \mathbf{p}_3) \times \mathbf{a}_3) \tag{6}$$

This is a quadratic equation in s. Since lines of sight in this problem are parallel to the xz-plane, the viewing direction is constrained to lie in the xz-plane. Thus, we can set the y-component of Eq. (6) to 0 and solve for s to get the viewpoints from which three lines appear to intersect. Substituting s into Eqs. 1-3 yields the asp vertices resulting from the intersection of three general asp surfaces.

EV visual events result from the apparent intersection of an object edge and a non-adjacent object vertex. Since the plane containing the vertex and the edge intersects the plane of rotation in a line, there is a single viewpoint (and its polar opposite) from which the vertex and edge appear to intersect. Since they intersect in a single image point, the event is represented by a point in aspect space. This kind of event is generated by the intersection in aspect space of the surface corresponding to the edge and the curve corresponding to the point. Solving Eqs. (1)-(4) for s and θ we see that the point is given by

$$s = \frac{y_0 - y_1}{y_2 - y_1} \tag{7}$$

$$\tan\theta = \frac{x_0 - (x_1 + a_1\, s)}{z_0 - (z_1 + c_1\, s)} \tag{8}$$

together with Eq. (1). This event is a special case of EEE events in which two of the edges intersect.

Thus, the asp for a face unoccluded by other faces is a volume of aspect space such as that pictured in Figure 6. When the face is occluded by other faces, other asps are subtracted from this volume, and the result is a number of pieces of the original volume. Any cross-section of these pieces for a particular θ is an image of the visible parts of the face from the corresponding viewpoint.

Since there is a constant number of asp vertices for every set of 3 object edges, the number of vertices is bounded by $O(n^3)$ where n is the number of edges in the scene. These are the vertices of volumes of a 3-dimensional space, so the number of edges, faces, and cells is also bounded by $O(n^3)$ by Euler's formula. Let q be the size of the asp for a face **f** and note that the asp for a face of size r by itself is $O(r)$. The time to subtract the asp for a face from the asp for **f** is $O(qr)$ using the analog of a standard $O(m_1 m_2)$-time algorithm to find the intersection of two polyhedra of sizes m_1 and m_2. Since the sum of the sizes of the faces is n, the time to construct the asp for the face is $O(qn)$, and the time to construct the asp for all faces is $O(n^4)$. These are worst case times; for many polyhedra the times are much better. For example, the convex case requires less time: there are no faces in front of any other face, so there is no subtraction of cells to be done. If the polyhedron is not known in advance to be convex, the naive algorithm for finding all of the faces in front of each face takes time $\Theta(n^2)$; thus, the asp construction algorithm takes time $\Theta(n^2)$.

4. Animating Rotation

The procedure for animating rotation involves computing and representing all events as well as the changes in the ESG that occur at each event. For example, suppose that the scene on the left side of Figure 8 is to be rotated. To display the first frame the ESG must be computed from the initial viewpoint using a standard hidden-line removal algorithm. The algorithm must be modified slightly to report occluding edges at T-junctions. Subsequent images are then displayed by determining whether any events are crossed by the change in viewpoint, and if so, updating the ESG. In Figure 8, when the viewpoint crosses event E_1, there is a topological change in the image: region R_1 gets an additional edge and region R_2 appears. When the viewpoint crosses event E_2, region R_1 again becomes 3-sided and region R_2 becomes 4-sided.

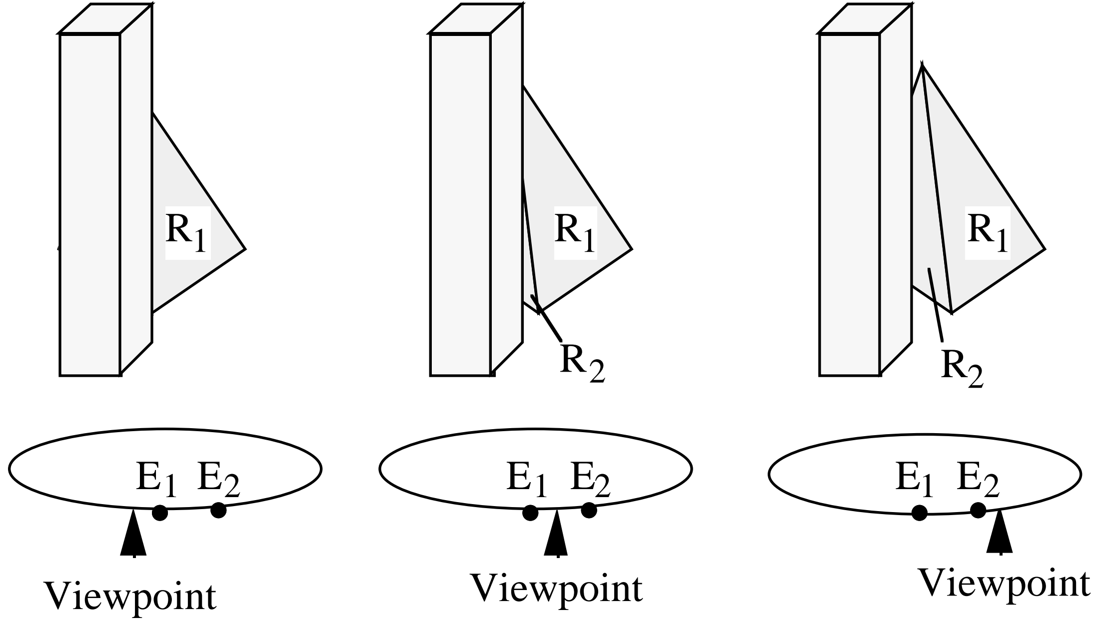

Figure 8. Three views of a scene from different viewpoints.

In order maintain the ESG, an additional data structure called the event list is maintained. The event list is a list of event viewpoints sorted by θ along the viewpoint path. With each event is stored a list of pointers to edges that appear in the ESG and a list of pointers to edges that disappear. The ovals of Figure 8 represent the event list; at event E_1 two new edges appear and one edge's endpoint becomes a vertex instead of a T-junction. At event E_2 one new edge appears and an endpoint of each of two edges changes from a T-junctions to a vertex. To construct the event list, we first construct the asp for the polyhedral scene. Vertices of the asp correspond to events, so for each vertex in the asp we add an event for the viewpoint of that vertex and add appropriate lists of edges to be removed and added from the ESG. When all of the events are found, they are sorted by viewpoint along the viewpath and put into a list.

The on-line phase of animating rotation involves keeping track of the current ESG as the viewpoint changes. Each time the viewpoint crosses an event, the ESG is updated by deleting and adding the specified edges. Each update requires linear time in the number of edges added and removed, which is usually a small number, insignificant compared to the number of edges in the scene. Displaying a frame is then about as fast as displaying a wire-frame object; the only added complexity is that the image coordinates of T-junctions must be computed by finding the intersection point of the two edges in image space. However, there are usually fewer visible edges to display for typical scenes, many of the edges may be hidden and are not drawn (although for very complex scenes there may be of course be more line segments to display than edges in the object). Thus, if the number of events is of the order of the number of frames to be displayed (as is the case for scenes of moderate complexity and moderate rotation rates), the display time is roughly equal to the time required to display a series of wire-frame views of the scene. Furthermore, the algorithm can take full advantage of 2-D and 3-D vector drawing and geometry hardware.

The preprocessing time required is dominated by the time to construct the asp for the scene. Constructing the asp takes time $\Omega(n^2)$ to $O(n^4)$, depending on the number of visual events in the scene, where n is the number of faces in the scene. Scenes that have a large number of visual events are scenes that have a large number of visible T-junctions from most viewpoints, such as scenes consisting of grids or highly non-convex objects. Probably most real-world scenes have $O(n^2)$ visual events, and the asp in this case is constructed in $\Omega(n^2)$ to $O(n^3)$ time.

This algorithm requires storage for every visual event along the path of viewpoints. Thus, the amount of storage required for events will be between $\Omega(n)$ and $O(n^3)$; for many scenes

it is $O(n^2)$. The amount required varies according to the visual complexity of the scene. For the worst case of visual complexity, the storage requirements may become impractical before the number of faces reaches 1000; as few as 600 faces may result in about 1,000,000 visual events for scenes of the highest-possible visual complexity [3]. In practice, common objects have a visual complexity much closer to the convex case than to the worst case, unless the scene modeled is visually very complex. The results of the implementation below suggest that the storage requirements are reasonable for scenes of small to moderate size and moderate visual complexity. Also, some of the data may be stored in slower secondary storage because the data in active use at one time is the event graph and the events close to the current viewpoint.

5. Results

A prototype of both the preprocessing and the on-line portions of this algorithm has been implemented on an IBM RS6000 model 320 workstation. The algorithm was tested on several scenes; six examples are shown in Figure 9.

Figure 9. Six polyhedral scenes on which the algorithm was tested.

Timing values and sizes for the asp construction phase of the algorithm for the six models are shown in Table 1. The numbers of vertices, edges, and faces of the original models are shown in columns 2-4. Columns 5-8 contain data about the events computed for the object in the selected rotation; column 6 is the size of the data file used to store the event data structure. Column 9 contains the preprocessing time required. Note that this is an inefficient prototype implementation; we believe that with more efficient program coding, the construction times could be greatly decreased.

Model	**Model Data**			**Asp Data**				
	Vertices	Edges	Faces	Events	Event data (KB)	Avg. events per edge	Max events per edge	Construction time (sec)
Squares	192	192	48	2365	28	12.3	25	7
Grid	82	105	41	1383	36	16.3	60	7
Objects	388	800	416	5269	120	6.6	30	299
Bars	451	880	449	9140	204	10.4	112	927
Tori	512	1024	512	5134	120	5.0	18	440
Spring	909	2709	1800	39953	868	14.7	52	3551

Table 1. Asp construction information.

The models were then subjected to 360-frame rotations (see Table 2). The timing results in columns 2-4 include the time to update the event structure as well as the time spent computing the image coordinates of each of the vectors to be displayed. The results in columns 5 and 6 include the time to update the event data structure and the time to compute 3-D coordinates of the vectors to be drawn. Because the experiments were run under Unix, the time required for a 360-frame rotation of a particular scene varied slightly; the average result of 10 trials is shown in Table 2. Since the time required to draw the vectors on the screen is not included, the actual frame rates will be somewhat slower if vector-drawing hardware is not available and nearly the same if fast vector-drawing hardware is available. If 3-D geometry hardware capable of comput-

ing T-junctions quickly is available, vectors should be drawn at essentially the full vector-drawing speed of the workstation since the time spent updating the display list for events is minimal.

Model	Animation Rate (frames per second) with 2-D vector drawing hardware			Animation Rate with 3-D vector drawing hardware	
	2° increment	6° increment	30° increment	2° increment	30° increment
Squares	409	346	321	907	896
Grid	456	414	404	976	957
Objects	83	83	82	159	159
Bars	67	60	58	122	121
Tori	65	63	63	123	122
Spring	19.7	19.7	19.6	40.4	40.4

Table 2. Animation speed results.

Several conclusions can be drawn from the data in Tables 1 and 2. First, asp construction time is reasonable for objects with thousands of edges, even with inefficient prototype program coding and significant occlusion, as is the case in Model 6. Second, the number of events that must be stored in the models tested is less than 16.3 times the number of edges. These models contain significant occlusion, so the number of events to be stored for models of approximately this size can be considered to be a small constant times the number of edges. Thus, the storage requirements appear to be practical for scenes of moderate size. However, note that since the number of events may be worse than linear in scene size, the storage required will not scale linearly with the number of edges in the scene for visually complex scenes.

The on-line results show that this technique is sufficiently fast that on a workstation such as the one used for tests, models with up to 1,000-10,000 faces may be shown rotating in real time, depending on the amount of occlusion in the scene and the speed with which the workstation can draw vectors. Second, the results show that a change in the number of degrees between frames in the animation sequence does not greatly affect the display rate. This indicates that the time needed to compute the image coordinates of the segments to be displayed greatly dominates the time needed to update the event list, so that processing a larger number of events between frames has little effect on the frame rate. Since the on-line computational cost of processing visual events is small and the storage requirements are reasonable, the animation of much larger scenes can be achieved when sufficient preprocessing time, storage, and vector-drawing speed are available.

6. Related Work

In computing successive frames of an animation sequence, one technique is to compute each frame independently and store the results. Animated displays of moderate complexity may thus be computed in advance and displayed in real time. Denber and Turner described a method of compressing the data in an animated sequence and increasing the speed of their replay [7]. The technique involves storing and displaying only the difference between successive images. The technique is raster-based, and it does not involve a faster method for computing the successive images. Also, approaches that involve the rapid display of pre-computed frames from secondary storage place high demands on the operating system and input-output hardware.

Glassner described a method for faster ray-tracing of a sequence of images in an animation [8]. The technique uses a space subdivision algorithm for decreasing the time required for ray tracing. The novel part of the method is that it uses the subdivision techniques in 4-D *space-time* rather than 3-D space, achieving approximately a 50% decrease in ray-tracing runtime for the examples given. This approach is superficially similar to the work presented here in that a higher-dimensional representation of the problem is used.

Sutherland, *et al*. [9] suggest that frame coherence is a potential source of speed improvements in hidden-surface computations. Hubschman and Zucker [10] use the idea of frame coherence to decrease the time required for hidden-line removal; they work in a world with one or two stationary convex polyhedra, and they find a number of frame coherence constraints. The result is a partition of the scene such that "the movement of the viewing position across a partition boundary results in an occlusion relationship becoming active or inactive." The scene is updated when one of these "change boundaries" is crossed. As a result, the storage requirements are $\Theta(n^3)$ even in the case of a single convex polyhedron, since the scene is partitioned by $\Theta(n)$ planes. Unfortunately, a generalization of this technique to multiple non-convex polyhedra would result in worst-case storage requirements of $\Theta(n^9)$ for a scene with n faces [4].

Shelley and Greenberg use the idea of frame coherence for animation with a viewpoint moving along a path of viewpoints in viewpoint space [11]. They use a number of culling and sorting rules to reduce the work involved. For example, they find the box bounding the path of viewpoints; any faces that point away from all viewpoints in the box can be removed from consideration for all viewpoints along the path.

Fuchs *et al*. [12, 13] address the problem of displaying a set of polygons from an arbitrary viewpoint in near-real time, with an approach that involves constructing the BSP-tree (Binary Space Partition tree) for a scene in an off-line preprocessing stage. Then the display of a frame from some viewpoint with hidden surfaces removed involves traversing the tree to get a list of faces in an approximation of back-to-front order. The faces are drawn on the screen in that order, and the result is an image with hidden surfaces removed. In this approach, displaying an image involves a tree-traversal and the display of all of the faces in the tree, which may be more than the number of faces in the scene since some faces are split in constructing the tree. This approach works well for the problems for which it applies. However, the BSP-tree approach cannot take advantage of the greater speed available in rendering scenes with vectors. In cases where hidden-line removal is a desirable or acceptable alternative, our approach may be used to achieve greater frame-rates since hidden-line removal makes it possible to draw the outline of polygons rather than filling them, which is usually slower. And even assuming that filling a polygon and drawing its outline require the same amount of time, our algorithm may have better on-line performance since it only draws the visible polygons in a scene, usually less than the total number of polygons in the scene. The BSP-tree approach requires drawing all of the polygons in the BSP-tree, which is more than the number of polygons in the scene. In more closely related work, Bern *et al.* [14] and Mulmuley [15] show how to determine the points along a straight flight path at which appearance changes topologically.

7. Concluding Remarks

This paper presents an algorithm for efficiently animating rotation by taking advantage of the frame coherence inherent in an animated sequence. The appearance from all viewpoints along a given path is computed in a preprocessing phase that works by constructing the aspect representation for the scene. The preprocessing phase also involves computing the initial appearance of the scene with a standard hidden-line removal algorithm. The on-line phase involves the display of views of the scene with hidden lines removed at a real-time rate. It is about as fast as displaying a wire-frame model of a polyhedral scene as it rotates, without removing hidden lines.

Another approach to this problem is to precompute all of the frames of the animation sequence. This approach requires considerably more storage than the method presented above, since each view is stored rather than one view and the differences between views. In our test im-

ages above, the number of events between frames was a small constant. In addition, pre-computation time is likely to be greater than that for the method presented above, although general comparisons are difficult to make. However, this simple approach may have the potential for slightly higher replay speed since the representation is in 2-D rather than 3-D vectors and T-junctions, and 2-D vectors may require less time to display. The asp approach is more flexible in that the number of degrees of rotation per frame is not determined in advance.

The only previous algorithm for efficiently animating rotation of general polyhedra with hidden lines or surfaces removed and without special-purpose graphics hardware is that of Fuchs *et al*. [12]. After some pre-computation time, it is efficient at computing the faces of the scene in an order that approximates back-to-front. This is sufficient for hidden-surface removal by drawing all of the (shaded) faces from back to front. However, when hidden-line removal is acceptable and when pre-computation time is available per viewpoint path (rather than per scene), our algorithm may provide better performance for a class of scenes, at the expense of greater storage requirements.

The algorithm presented here is practical only for a certain class of scenes. When the number of faces in the scene becomes large and the scene is visually complex, the number of visual events will eventually become too large to store. However, the prototype implementation shows that for polyhedral scenes of small-to-moderate size, the number of visual events is a relatively small constant times the number of edges in the scene. In practice, depending on the visual complexity of the scene, we estimate that a typical workstation has enough memory to store the events for polyhedral scenes containing up to approximately 1,000 to 100,000 edges.

In addition, there must be sufficient preprocessing time per rotation and sufficient processing speed to handle the on-line phase. Perhaps a good rule of thumb is that a workstation will be able to animate rotation for a polyhedral scene in real time if it can display a rotating wire-frame model of the scene in real time. Since only a small portion of the on-line display time is used in updating the ESG this rule applies even if 3-D rotation and vector-drawing hardware is available.

References

1 J Foley, A van Dam, S Feiner, and J Hughes, *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley (1990).

2 H Plantinga and C R Dyer, An algorithm for constructing the aspect graph, *Proc. 27th IEEE Symp. Foundations of Computer Sci*., 123-131 (1986).

3 H Plantinga, *The Asp: A Continuous, Viewer-Centered Object Representation for Computer Vision*, Ph.D. dissertation, University of Wisconsin – Madison, (1988).

4 H Plantinga and C R Dyer, Visibility, occlusion, and the aspect graph, *International Journal of Computer Vision* **5**(2), 137-160 (1990).

5 H Plantinga, C R Dyer, and W B Seales, Real-time hidden-line elimination for a rotating polyhedral scene using the aspect representation, *Proc. Graphics Interface '90*, 9-16 (1990).

6 J Koenderink and A van Doorn, The internal representation of solid shape with respect to vision, *Biol. Cybernetics* **32**, 211-216 (1979).

7 M Denber and P Turner, A differential compiler for computer animation, *ACM Computer Graphics* **20**(4), 21-27 (1986).

8 A Glassner, Spacetime ray tracing for animation, *IEEE Computer Graphics and Applications* **8**(2), 60-70 (1988).

9 I E Sutherland, R F Sproull, and R A Schumacker, A characterization of ten hidden-surface algorithms, *Computing Surveys* **6**(1), 1-55 (1974).

10 H Hubschman and S Zucker, Frame-to-frame coherence and the hidden surface computation: constraints for a convex world, *ACM Computer Graphics* **15**(3), 45-54 (1981).

11 K Shelley and D. Greenberg, Path specification and path coherence, *ACM Computer Graphics* **16**(3), 157-166 (1982).

12 H Fuchs, Z M Kedem, and B F Naylor, On visible surface generation by a priori tree structures, *ACM Computer Graphics* **14**(3), 124-133 (1980).

13 H Fuchs, G Abram, and E Grant, Near real-time shaded display of rigid objects, *ACM Computer Graphics* **17**(3), 65-72 (1983).

14 M Bern, D Dobkin, D Eppstein, and R Grossman, Visibility with a moving point of view, *Proc. Symposium on Discrete Algorithms*, 107-117 (1990).

15 K Mulmuley, Hidden surface removal with respect to a moving view point (Extended abstract), *Proceedings of the 23rd. Annual Symp. on Theory of Computing,* 512-522 (1991).