Visibility with a Moving Point of View

Marshall Bern [*]   David Dobkin [†]   David Eppstein [‡]   Robert Grossman [§]

Abstract

We investigate 3-d visibility problems in which the viewing position moves along a straight flightpath. Specifically we focus on two problems: determining the points along the flightpath at which the topology of the viewed scene changes, and answering ray-shooting queries for rays with origin on the flightpath. Three progressively more specialized problems are considered: general scenes, terrains, and terrains with vertical flightpaths.

1.   Introduction

In recent years computer-generated images have grown commonplace, but computer-generated animations—sequences of images—are still prohibitively expensive for all but a few uses. For the most part, this disparity is inherent: high-quality animation uses at least 12 distinct images per second. On the other hand, this disparity is partially due to a lack of algorithms. Successive images are typically treated independently, even though they may differ only slightly.

In this paper we investigate a very simple type of animation: a fixed three-dimensional scene is viewed from a sequence of different points of view. More specifically, successive images correspond to perspective views of a polygonal scene from sample points along a straight trajectory, or *flightpath*. Though this problem is quite basic, it is also widely applicable in flight simulation and data visualization.

We assume that scenes are to be computed in *object-space*, that is, output is given as device-independent 2-d coordinates, rather than pixel-by-pixel [27]. The currently practical solutions to this problem are image-space solutions: either z-buffers, or the priority method with priority orderings computed using binary space partitions [8, 20]. Image space solutions, however, suffer from aliasing and hence tend to produce lower quality images.

In a sequence of views of a static scene, transitions between viewpoints will typically be smooth, rapidly-computable transformations. However, at certain points along the flightpath *topology changes* occur—for example, when an object first peeks around the edge of a closer object—and the visible scene is not so easily computed. We develop algorithms

for discovering topology changes, meaning the critical flightpath points as well as the corresponding changes to the topology of the visible scene. We also describe data structures that answer *ray-shooting queries*, that is, given a ray r with origin on the flightpath and arbitrary direction, return the first polygon struck by r. This type of query is fundamental to the technique of ray-tracing.

The running times of our algorithms depend on three parameters: n, the total number of edges in all objects; ℓ, the number of *transparent* topology changes (that is, the number of different scene topologies visible along the flightpath, assuming that all objects are transparent); and k, the number of opaque topology changes. A major open problem in this area is to replace dependence on ℓ by dependence on k, which is typically much smaller. In general, $0 \leq k \leq \ell < n^3/3$. We obtain the following results for finding topology changes. In the first case we find all transparent—including opaque—topology changes; in the other two we discover only opaque topology changes.

- For general polygonal scenes, a simple algorithm with running time $O((n^2 + \ell) \log n)$ and a more complicated algorithm with time $O(n^2 + \ell \log n)$.

- For terrains, an algorithm with time $O((n + k)\lambda_3(n) \log n)$. A *terrain* is a polyhedral surface intersected at most once by any line parallel to the z-axis. The functions $\lambda_i(n)$ are slightly superlinear for each i [26].

- For terrains with vertical flightpaths, an algorithm with time $O(n\lambda_4(n) \log n)$, matching an earlier result of Cole and Sharir [5]. The two algorithms are similar, but our explanation is more geometric and theirs more algebraic.

Techniques used in our algorithms include geometric sweeps and transforms similar to skewed projection [12]. There are relationships between finding topology changes and two planar problems: the well-studied problem of line segment intersection and the problem of finding the external contour of a union of polygons.

The ray-shooting problem is, in a sense, a special case of point location in a 3-d subdivision (the visible scene cross time). For this problem we obtain the following results.

- For the general problem, a data structure of size $O(n^2 + k)$ with query time $O(\log^2 n)$. Space improvement is possible if queries are ordered by time.

- For terrains with vertical flightpaths, a data structure of size $O(n\lambda_4(n))$ with query time $O(\log n)$, improving upon a known $O(\log^2 n)$ [5] and giving the first $O(\log n)$ point-location method for a transforming subdivision.

There has been surprisingly little work on these two problems directly, though there has been a fair amount of related work. Cole and Sharir [5] solve a number of visibility problems on terrains, including finding topology changes and ray-shooting for the special case of vertical flightpaths. Hubschman and Zucker [11] treat convex objects. Swart [28] considers the problem of viewing independently and linearly moving objects with trajectories that can be dynamically changed. His running times, however, depend on events such as changes in x-coordinate order of vertices in a projection of the scene. Plantinga [21, 22] and others give algorithms that compute "aspect graphs" and "aspect representations" for orthographic views of an object. These data structures have vertices or regions for each of the $O(n^4)$ topologically distinct views of an object. Translating our results into their

Figure 1. Skewed projection of a polygonal scene.

terminology, we show that to determine all views along a given flightpath, only a small portion of the (perspective) aspect representation need be computed.

2. Preliminaries

Assume we have a set $\mathcal{S}$ of polygons, nonintersecting except along boundaries, and an oriented line segment f, the flightpath, in 3-space. Let f be parametrized by "time" t, running from 0 to 1. The point on f with parameter value t will be denoted $p(t)$.

We imagine projecting all polygons in $\mathcal{S}$ from a given point $p(t)$ on f onto a sphere centered at $p(t)$ that is large enough to contain $\mathcal{S}$. One can view this projection as an embedding of a planar graph G_t, which has vertex set containing all intersection points of edges and the obvious edge set. Vertices of G_t are labeled, perhaps with the "names" of the intersecting edges. A point q along an edge of $\mathcal{S}$ is *visible* at time t if the line segment $qp(t)$ does not pass through the interior of a polygon of $\mathcal{S}$. The projection from $p(t)$ of all visible points of $\mathcal{S}$ defines a labeled, embedded subgraph of G_t called H_t. The edges of G_t that are not in H_t are called *hidden lines*. The *visible scene* at time t is the embedded graph H_t with each face labeled by the name of the polygon of $\mathcal{S}$ visible within that face.

We say G_t and G'_t are isomorphic if they are isomorphic as embedded, labeled graphs; that is, the mapping must preserve the embedding and the vertex labels. A *transparent (opaque) topology change* occurs at t if G_t (respectively, H_t) changes, that is, for each small $\epsilon > 0$, $G_{t-\epsilon}$ and G_t are nonisomorphic.

The problem of "finding all topology changes" is the following: given $\mathcal{S}$ and f, compute a list of the *critical values* of t at which a topology change occurs. This list should be in order of increasing t, and each entry in the list should include a description (of length $O(1)$) of the changes to the visible scene. The following lemma is immediate.

Lemma 1. *A transparent topology change occurs at time t if and only if there are three edges e_1, e_2, and e_3 of (not necessarily distinct) polygons in $\mathcal{S}$ such that there is a line that intersects $p(t)$, e_1, e_2, and e_3. An opaque topology change occurs at time t if, in addition, there is a line segment with one endpoint at $p(t)$ that intersects e_1, e_2, and e_3 and passes through no polygon interiors.* ■

Now let e be a line segment, not lying on the same line as the fixed flightpath f, and parametrized by u running between 0 and 1. Let T be the interior of the tetrahedron defined by all line segments with one endpoint on e and one on f. We define a mapping

3

$sp_e : T \rightarrow [0,1] \times [0,1]$ as follows: a point $p \in T$ maps to (u,t), where $p(u)$ and $p(t)$ are the points on e and f with parameter values u and t, and are the endpoints of the (unique) line segment l passing through p with endpoints on e and f. If e_1 is a line segment in T, then it is not hard to confirm that $sp_e(e_1)$ is either a line segment or a connected piece of a hyperbola in $[0,1] \times [0,1]$.

If e and f were complete lines rather than segments, sp_e could be extended to a map from R^3 to $\mathrm{R}^2 \cup \{\infty\}$. This extension is essentially the same as the *skewed projection* introduced by Jaromczyk and Kowaluk [12].

It is not hard to see that a transparent topology change involving edge e of a polygon in $\mathcal{S}$ corresponds exactly to the intersection of two curves $sp_e(e_1)$ and $sp_e(e_2)$ in the skewed projection of $\mathcal{S} \cap T$. The next lemma relates opaque topology changes to the external contour of a union of skewed projections of polygons. See Figure 1.

Lemma 2. *An opaque topology change occurs at time t if and only if there exists an edge e of $\mathcal{S}$ and $u \in [0,1]$ such that (u,t) is a vertex of the boundary of $sp_e(\mathcal{S})$.*

Proof: If (u,t) is a vertex of the boundary of $sp_e(\mathcal{S})$, then the line segment with endpoints $p(t)$ on f and $p(u)$ on e intersects 3 edges and the flightpath, but no polygon interior. Conversely, if there is a line segment that intersects the points $p(t)$ and $p(u)$ and two edges in T, then (u,t) must be the intersection of two curves in the skewed projection. If in addition this segment intersects no polygon interiors, then (u,t) must be a boundary vertex. ∎

3. General Scenes

We first give a simple, practical algorithm and then a more complicated, but asymptotically faster, algorithm.

Theorem 1. *All topology changes for a general scene with a line segment flightpath can be computed in time $O((n^2 + \ell) \log n)$ and space $O(n)$, where n is the total number of edges in all polygons and ℓ is the number of transparent topology changes.*

Proof: Below we shall describe an algorithm that computes ℓ 4-tuples, each consisting of a critical value of t and three edges that are coincident as viewed from $p(t)$. In all our algorithms for the topology-change problem, we consider a point $p(t)$ to be the site of more than one topology change if more than three edges are coincident as viewed from $p(t)$. In such a case, more than one 4-tuple would share the same t. An example is the case of a vertex of a polyhedron coming into view from behind a closer object.

After computing all 4-tuples, we sort them by increasing t. We then compute the graphs G_0 and H_0. Each face of H_0 is labeled with the polygon of $\mathcal{S}$ visible within that face; this gives the first visible scene. Each vertex of G_0 is labeled with the first polygon "below" that vertex; that is, the vertex at the intersection of the projections of edges e and f is labeled with the first polygon after both e and f along the viewing ray from $p(0)$ through e and f. The label "background" means that the viewing ray continues forever. The computation and labeling of G_0 and H_0 can be accomplished in time $O(n^2)$ using McKenna's hidden surface removal algorithm [15].

We then run through the sequence of 4-tuples while updating the labeled graphs G_t and H_t. Each update takes time $O(1)$. Notice that labels change only at transparent topology changes. A newly-visible face in H_t is either bounded by an edge of the polygon visible

Figure 2. Sweep-plane algorithm for general scenes.

within that face, or it is a "window" formed by 3 (or more) polygons through which a more distant polygon is visible. In the latter case, the face's label is computed using an appropriate vertex label from G_t; indeed, windows are the only reason to maintain these labels.

We now describe how to compute the list of 4-tuples. For each edge e of a polygon in $\mathcal{S}$, we perform a rotational sweep around e, similar to Bentley and Ottmann's line segment intersection algorithm [4]. Let T_t be the triangle with base equal to flightpath e and apex at point $p(t)$ on f. A *pierce point* of T_t is the intersection of T_t and an edge of a polygon in $\mathcal{S}$.

The sweep proceeds from $t = 0$ to $t = 1$ as shown in Figure 2. During the sweep, a balanced binary tree maintains the pierce points of T_t sorted by angle around $p(t)$. A priority queue maintains future events by increasing t. The events to be handled are: (1) An endpoint of a edge is reached, (2) a polygon edge intersects an edge of T_t, thereby entering or leaving the sweep tetrahedron, and (3) two adjacent pierce points exchange position in the angular order. There are at most $2n$ events of types (1) and (2); scheduling these events is straightforward. The lines passing through e, f, and any other segment define a quadratic surface S (see [12]). A fourth segment can intersect S in at most two points; thus the number of events of type (3) for a fixed edge e is at most $n(n-1)$. Scheduling an event of type (3) amounts to finding the minimum future t at which e, f, and two other given segments are colinear. This computation—straightforward analytic geometry that we omit—takes time $O(1)$. After an event of any of the three types, at most two future events of type (3)—the upcoming colinearities of the newly adjacent pairs—must be scheduled and inserted into the priority queue. After events of types (1) or (2), at most two future events that have already been scheduled must be deleted.

A priority queue with $O(\log n)$ update times results in $O((n+\ell_e)\log n)$ time for a sweep around edge e, where ℓ_e is the number of transparent topology changes discovered. The sum of all ℓ_e values is ℓ. ∎

Theorem 2. *All topology changes for a general scene with a line segment flightpath can be computed in time $O(n^2 + \ell \log n)$ and space $O(n^2)$.*

Proof: We perform a rotational sweep around f in order to discover critical values of t; the remainder of the algorithm after the computation of the 4-tuples is the same as in the first algorithm.

The configuration of pierce points of segments of $\mathcal{S}$ can be represented by its dual arrangement of lines, a data structure of size $O(n^2)$. Events are (1) the appearance or

disappearance of a line (corresponding to reaching a vertex of $\mathcal{S}$), or (2) three lines becoming coincident (corresponding to transparent topology changes).

The arrangment is represented as a graph with a node for each border segment of a face and edges between borders that share an endpoint. Each intersection of lines in the arrangement is the meeting of 8 border segments; the edges between their corresponding nodes are augmented with directional information so that faces may be traced either clockwise or counterclockwise. We also provide pointers so that the border segments incident to an intersection can be found in $O(1)$ time given the identifiers of the two intersecting lines. Events of type (1) necessitate $O(n)$ work in updating this data structure, corresponding to the total complexity of all faces bordering the line that is inserted or deleted [7]. Events of type (2) necessitate $O(1)$ work as only $O(1)$ border incidences are changed.

A priority queue (implemented as a heap) holds a schedule of possible future events, including the times at which each triangular cell in the arrangement degenerates to a point. Notice that the initial $O(n^2)$ possible events can be formed into a heap in $O(n^2)$ time. As triangles "invert", future events are inserted or deleted, resulting in the $O(\ell \log n)$ part of the running time. ∎

It is possible to compute an unsorted list of critical values of t in time $O(n^2 \log n + \ell)$, faster than the algorithm above for large ℓ. We perform the following steps for each polygon edge e. We compute the projection $sp_e(Q)$ of each polygon Q intersecting tetrahedron T. Each $sp_e(Q)$ will be a "curved polygon", one with sides that are portions of hyperbolas. Next we use Mulmuley's randomized segment or curve intersection algorithm [17] to compute all intersections in $sp_e(\mathcal{S})$ in expected time $O(n \log n + \ell_e)$, where ℓ_e is the number of intersections. The expectation is over the randomization used in the algorithm, not over a distribution of inputs. The sum of all ℓ_e is ℓ.

This algorithm explicitly computes the points of intersection of a set of curved polygons. By Lemma 2, the computation of the 4-tuples for all opaque topology changes can be reduced to n computations of the external contour of a union of curved polygons. We expect that an improved algorithm to compute the external contour of a union of ordinary polygons should also have implications for the case of curved polygons, and hence for the problem of finding topology changes.

4. Terrains

A *terrain* is a polyhedral surface that is intersected at most once by any line parallel to the z-axis [5, 25]. Thus the projection of a terrain onto the xy-plane is a planar subdivision. In this section $\mathcal{S}$ denotes a terrain with n edges. The advantage of a terrain is given by the following lemma, in which a *forward* ray with origin on flightpath f is one that has positive dotproduct with f oriented in the direction of increasing t.

Lemma 4. *In time $O(n \log n)$, the edges of $\mathcal{S}$ can be ordered $e_1, e_2, \ldots e_n$, such that if there is a forward ray from a point on f that intersects first e_i and next e_j, then $i < j$.*

Proof: Let $\mathcal{S}^*$ (respectively f^*) denote the projection of $\mathcal{S}$ (f) onto the xy-plane. As in Lee and Preparata's point location algorithm [14, 23], the edges of $\mathcal{S}^*$ can be assigned to polygonal chains, monotone with respect to lines perpendicular to f^*. (A *polygonal chain* is a path of line segments connected only at successive endpoints; it is *monotone* with respect to a line l if its intersection with any line perpendicular to l is at most one point [23].)

Figure 3. Events in the view of a terrain

Chains can be ordered front to back with respect to f^*, where front is the direction of decreasing t. Within chains edges may be ordered arbitrarily. It is easy to confirm that this ordering has the desired property. ∎

We define the *i-th silhouette* $S_t(i)$ to be the "horizon line" at flightpath point t, considering only the first i edges. That is, $S_t(i)$ is an ordered set of segments, each of which is a piece of an edge of index at most i, such that no line of sight through a segment of $S_t(i)$ passes below an edge e_j, $j \le i$. $S_t(i)$ is monotone with respect to a horizontal line in the viewed scene. For each t and i the silhouette $S_t(i)$ has at most $\lambda_3(i)$ vertices [5]. The function $\lambda_3(n)$ is known to be $\Theta(n\alpha(n))$, where $\alpha(n)$ is the very slowly growing inverse Ackermann function [9]. The function $\lambda_4(n)$ is known to be $\Theta(n2^{\alpha(n)})$ [2].

Theorem 3. *All k opaque topology changes for a terrain with an arbitrary flightpath can be computed in time $O((n+k)\lambda_3(n)\log n)$ and space $O(n\lambda_3(n))$.*

Proof: We show how to discover topology changes that are visible along forward rays in order along the flightpath as t increases. Running the algorithm twice, once with time reversed, computes all topology changes. Updating the visible scene is especially straightforward for terrains, as each face must be bounded by an edge of the polygon visible within that face; that is, there are no "windows". Thus in order to label H_t, we need not maintain G_t as in the previous section.

The first step is to compute all silhouettes for $p(0)$ using a standard hidden surface algorithm [15]. Our algorithm will maintain an unordered set $E_t(i)$ of all polygon edges that contribute at least once to silhouette $S_t(i)$ and an ordered list $V_t(i)$ of all vertices of the silhouette, implemented as a binary search tree. The edges can be specified simply by index, while the vertices are specified by ordered pairs of indices with the order implying the segments of $S_t(i)$. (As a practical matter, these edges and lists can be maintained by a similar lists or persistent data structure [6], though this is not necessary for the bounds of the theorem.)

As in the algorithm of Theorem 1, a priority queue maintains future events. The priority queue contains future events of two kinds sorted by increasing t: (1) some future point $p(t)$ on f, an endpoint of edge e_i, and a point on some edge in $E_t(i-1)$ are colinear, and (2) some future $p(t)$ on f, some point on edge e_i, and a vertex in $V_t(i-1)$ are colinear. See Figure 3. Given a vertex (either an endpoint of an edge e_i or a vertex of $V_t(i-1)$) and an edge, it is possible to determine their next colinearity in $O(1)$ time, since in the viewed scene vertices have either linear or quadratic apparent motion.

Notice that for each endpoint of edge e_i, each colinearity, not just the one that occurs first, with an edge of $E_t(i-1)$ is queued. Similarly, for each vertex of $V_t(i-1)$ each colinearity is queued. Thus throughout the algorithm, the priority queue contains $O(n\lambda_3(n))$ events.

An event of type (1) may not actually be an opaque topology change, as the edge of $E_t(i-1)$ involved in the colinearity may not be part of $S_t(i-1)$ at that intersection point. An event of type (2) always will be an opaque topology change, and all opaque topology changes will be of one type or the other. Events of type (1) are each reported at least twice, once for each of the edges of $\mathcal{S}$ sharing the endpoint. A minor modification avoids this redundancy. When an event of type (1) occurs, it is tested to see whether it is an opaque topology change. An event involving an endpoint of edge e_i and edge $e_j \in E_t(i-1)$ can be tested in time $O(\log n)$ by searching within the list $V_t(i-1)$ and checking whether the endpoint of e_i lies on $S_t(i-1)$ at the current time t. There are at most $O(n^2)$ events of type (1) that are not opaque topology changes, since each vertex and edge combine to produce at most one.

In the case of an event involving e_i that is also an opaque topology change, we update each $E_t(j)$ and $V_t(j)$, $j \geq i$, along with the priority queue. For each j, the list $E_t(j)$ (respectively $V_t(j)$) is updated by inserting or deleting $O(1)$ affected edges (vertices). The priority queue is updated by deleting all events involving a vertex of $V_t(j)$ (respectively, edge of $E_t(j)$) that no longer exists and inserting all events involving a new vertex of $V_t(j)$ (new edge of $E_t(j)$). In order to find the events that must be deleted, a dictionary into the priority queue to look up events by vertex (edge) must be provided.

The number of events of type (1) scheduled at $t = 0$ is bounded by $2n^2$ since (in the absence of degeneracies) each endpoint and edge uniquely specifies a future event time. The number of initial events of type (2) is bounded by $2n\lambda_3(n)$ since each vertex in $V_t(i-1)$ may combine with e_i to produce at most two events. Events that are also opaque topology changes incur extra work of time $O(\lambda_3(n)\log n)$ in inserting and deleting $O(\lambda_3(n))$ events from the priority queue. ∎

It may be possible to improve Theorem 3 with a sweep algorithm that, for each edge e_i, queues only its next event, rather than all future events with the current silhouette $S_t(i-1)$. A difficult data structure problem arises in attempting such an improvement: a query asks for the earliest intersection of a line segment, each endpoint of which has linear motion, and a polygonal chain, each vertex of which has quadratic motion. The solution should be dynamic, allowing fairly rapid updates of the polygonal chain.

Cole and Sharir [5] give an example in which k is $\Theta(n^3)$: flying past $\Omega(n)$ tall peaks with a scene of complexity $\Omega(n^2)$ (such as a mesh of tall peaks and broad valleys) in the distant background. Thus the algorithms of Section 3 are preferable in the case of large k.

5. Terrains with Vertical Flightpaths

In this section $\mathcal{S}$ is a terrain and f is a flightpath parallel to the z-axis. Let e be a line segment, not lying on the same line as f.

Lemma 5. *Each vertical (constant u) line in $[0,1] \times [0,1]$ intersects the boundary of $sp_e(\mathcal{S})$ at most once.*

Proof: Assume two points (u, t_1) and (u, t_3) both lie outside $sp_e(\mathcal{S})$, but some point (u, t_2) with $t_1 < t_2 < t_3$ lies inside. Then the interior of the vertical triangle in 3-space with vertices at point u on e and points t_1 and t_3 on f intersects $\mathcal{S}$, but the lower edge of this triangle does not intersect $\mathcal{S}$. This contradicts the fact that $\mathcal{S}$ is a terrain. ∎

Theorem 4. *All $O(n\lambda_4(n))$ opaque topology changes for a terrain with a vertical line segment flightpath can be computed in time $O(n\lambda_4(n)\log n)$ and space $O(\lambda_4(n))$.*

Proof: We first compute 4-tuples of critical times and edges as follows. For each edge e of $\mathcal{S}$, we repeat the following steps. We compute the image $sp_e(g)$ of each edge g of $\mathcal{S}$. By Lemmas 2 and 5 topology changes occur at exactly the vertices of the pointwise maximum of the curved segments $sp_e(g)$. To compute the pointwise maximum, one can use a divide-and-conquer method [3, 10]: recursively compute the pointwise maximum of two halves of the set of curved segments and then merge these maxima. The pointwise maximum has complexity $O(\lambda_4(n))$ and the divide-and-conquer algorithm takes time $O(\lambda_4(n)\log n)$.

It takes time $O(n\lambda_4(n)\log n)$ to merge the lists of 4-tuples for all edges e. Adding the descriptions of the scene changes to the 4-tuples is straightforward. ∎

Cole and Sharir adapt Wiernik and Sharir's arrangement of line segments with super-linear lower-envelope complexity [29] to show that the number of opaque topology changes for terrains with vertical flight paths may be $\Omega(n\lambda_3(n))$. It is unknown whether the number of topology changes may be as high as $\Theta(n\lambda_4(n))$.

6. Ray-shooting for General Scenes

In this section we sketch a data structure to answer ray-shooting queries for a general polygonal scene with an arbitrary flightpath. In the next section, we specialize to the case of terrains with vertical flightpaths. In the first case, we use a direct approach, that is, we maintain the visible scene as a subdivision of a 2-sphere and treat ray-shooting queries as point location queries. In the second case we use the dual approach of Cole and Sharir [5].

An interesting feature of this problem is that the subdivision is dynamic in two senses. At topology changes edges must be inserted or deleted; between topology changes the subdivision transforms continuously. Preparata and Tamassia [24] have recently considered the problem of monotone planar subdivisions dynamic in the first sense; we make use of their results. Very briefly, their method uses two total orders on the union of the sets of vertices, edges, and faces. These orders induce a unique decomposition of the subdivision edges into polygonal chains and guide the restructuring of these chains during an update.

We also make use of *persistent* data structures, specifically persistent search trees of various kinds. A persistent data structure is a data structure that in effect includes all its own old versions. A query to a persistent search tree includes a look-up key, as usual, along with a "time", that specifies which old version to search. The usual method of providing persistence is to copy the root-to-leaf access path of a newly-inserted or deleted node, so as to preserve both old and new versions. An initial search into a list of roots at various "times" then allows access to all old versions of the data structure. Path-copying requires $O(m\log m)$ space, where m is the total number of data items over all time. Driscoll et al. [6] showed that by adding a few extra pointers to each node and copying a node only when all its extra pointers are in use, the space requirement can be reduced to only $O(m)$.

Assume without loss of generality that line segment f lies along the z-axis. Sphere S_t will be centered at point $p(t)$ on f; each S_t is the same size and large enough that it contains all of $\mathcal{S}$. Assume that S_t is parametrized by spherical coordinates ϕ (latitude) and θ (longitude) with f lying along its polar axis. Thus lines parallel to the z-axis project to constant-θ lines (meridians).

Figure 4. Making a subdivision monotone.

The first step is to compute the projection of $\mathcal{S}$ onto the initial sphere S_0. Next, hidden lines are removed, giving an initial view of the scene that may be considered as a planar graph H_0 or as a polygonal subdivision of S_0. The polygonal subdivision can be made monotone with respect to latitude lines (that is, the intersection of any cell with a meridian is a single segment) by adding some *artificial edges* that extend latitudinally (along constant-ϕ lines) from *interior cusps*, as shown in Figure 4. We then compute Lee and Preparata's *chain tree* in order to answer point location queries in this subdivision [14, 23]. A chain tree stores a monotone polygonal chain at each node. Each edge of the subdivision is explicitly listed in only one chain, though we may think of each chain as completely dividing the subdivision into higher-latitude and lower-latitude parts. Because we have fixed the orientation of the scene by choosing f to lie along the polar axis, some "monotone" chains may include meridial segments; this degeneracy does not cause any real difficulties. (We call a line segment *meridial* if it lies along a meridian.)

Notice that there is a one-to-one correspondence between point location queries in the subdivision and ray-shooting queries with origin at $p(0)$. The following lemma assures us that a chain that is monotone with respect to latitude remains monotone as we vary t, so long as its topology remains unchanged. Notice that under a smooth transformation an edge must become meridial before it "bends backwards".

Lemma 6. *If edge e projects to a meridial segment from some point along f, then e projects to a meridial segment from every point along f.*

Proof: If edge e projects to a meridial segment from some point $p(t)$ along f, then e is contained in a plane containing f. ∎

Notice that the chain tree, unlike other planar point location data structures, does not need to change as the subdivision transforms smoothly while remaining monotone. That is, comparing a query ray (given by time t and spherical coordinates ϕ and θ) against a chain $\mathcal{C}$ still takes only $O(\log n)$ time, since the spherical coordinates of a given vertex or edge of $\mathcal{C}$ at time t can be computed in $O(1)$ time.

Each topology change necessitates the addition or deletion of $O(1)$ edges and vertices from the polygonal subdivision. When an interior cusp first comes into view an artificial edge must also be added. Each addition or deletion is an update that can be handled by the methods of Preparata and Tamassia [24]; in fact, our updates are local, special cases. Thus we can update the chain tree in time $O(\log^2 n)$. By using the persistence methods of Driscoll et al. [6] to maintain "old versions" of the chain tree, we can answer ray-shooting

queries with arbitrary origins on f. If ray-shooting queries are ordered by time, then we may update the chain tree nonpersistently instead.

In addition to handling topology changes, however, we must also handle *artificial topology changes*, that is, points along f at which graph H_t changes because an artificial edge a of H_t intersects a vertex v not previously on a. At artificial topology changes we must add a new vertex v' to the subdivision (at first coincident with v) and redefine the artificial edge to lie between v' and the interior cusp. The next lemma shows that the number of artificial topology changes is not excessive.

Lemma 7. *There are $O(n^2)$ artificial topology changes along f.*

Proof: Assume artificial edge a lies within a polygonal face F in the embedding of H_t and that a intersects a vertex v of H_t at time t but not at any prior time after the last topology change. Then v must be a vertex of the boundary of F at which the interior angle is reflex; hence v must be the projection of a vertex of a polygon of $\mathcal{S}$. Thus at time t, two vertices of $\mathcal{S}$—the one that induces artificial edge a and the one corresponding to v—project to the same ϕ-coordinate, and these vertices do not project to the same ϕ-coordinate at all times. There are $O(n^2)$ such t. ■

Theorem 5. *For general scenes with arbitrary flightpaths, a data structure of space $O(n^2 + k)$ that answers ray-shooting queries in time $O(\log^2 n)$ can be built in preprocessing time $O((n^2 + k)\log^2 n + p\log n)$. If queries are ordered by time, then the space can be reduced to the maximum complexity of a visible scene along f.*

Proof: We first run the algorithm of Theorem 1 and remember all opaque topology changes. We also compute all artificial topology changes in time $O(n^2)$ by testing each pair of vertices of $\mathcal{S}$. We then follow the method given above: compute the initial scene with hidden lines removed, build a chain tree, and persistently update the chain tree through topology changes. The preprocessing time follows from Theorem 1, the query time from the chain method [14, 24], and the space bound for unordered queries from the space-saving methods of Driscoll et al. [6]. ■

7. Ray-shooting for Terrains with Vertical Flightpaths

Assume $\mathcal{S}$ is a terrain and f is a segment along the z-axis. For simplicity, assume f is the entire z-axis. Below we describe a data structure that answers ray-shooting queries for rays with origin on f in time $O(\log n)$. As above, a ray is given by a triple (t, θ, ϕ), where $t = z$ is a parameter running along the flightpath, θ is longitude around sphere S_t, and ϕ is latitude.

We briefly describe the method of Cole and Sharir [5]. Consider the intersection of $\mathcal{S}$ with the vertical half-plane with boundary f and a fixed longitude θ_0. The intersection is a polygonal chain $\mathcal{C}$ as shown in Figure 5(a). If points in the vertical half-plane are given by cylindrical coordinates (r, z), then a ray with origin $p(t)$ on f and longitude θ_0 can be specified by an equation $z = ar + t$, $r \geq 0$. A duality mapping takes such a ray to a point $(-a, t)$. Each polygon P_i in Figure 5(b) consists of exactly those points that are dual to rays that first strike a given segment of the chain in 5(a). Polygons in 5(b) are unbounded, since one can see the entire terrain from a sufficiently high viewpoint. (Think of the horizontal axis as ϕ, though ϕ varies nonlinearly with horizontal distance.) Furthermore, each edge

Figure 5. (a) Cross-section of $\mathcal{S}$ at θ_0. (b) Dual subdivision $D(\theta_0)$. (c) A topology change in $D(\theta)$.

of the polygonal subdivision $D(\theta_0)$ in 5(b) lies on a ray r_i formed by the union of edges of $D(\theta_0)$. (Rays r_i are the duals of viewing rays through a vertex of $\mathcal{C}$.)

Point location on $D(\theta_0)$ answers ray-shooting queries with longitude θ_0. What happens to this polygonal subdivision as θ varies? Between two successive critical longitudes, the topology of subdivision $D(\theta)$ remains constant. There are two types of critical longitudes: (C1) the longitudes of vertices of $\mathcal{S}$, and (C2) longitudes at which 3 vertices of $\mathcal{C}$ and flightpath f can be connected by a straight segment that passes through no interiors of edges of $\mathcal{C}$. There are at most n critical longitudes of type (C1) and $O(n\lambda_4(n))$ of type (C2) [5]. At a critical longitude of type (C2), two vertices v_i of $D(\theta)$ pass through each other as shown in Figure 5(c). Below we view such a topology change as a rotation in a binary tree.

The crux of the ray-shooting problem is to give a planar point location method that works for varying θ. Cole and Sharir use chain trees. In the proof below we describe a faster method that exploits the fact that for each θ the edges of $D(\theta)$ form a tree.

Theorem 6. *For terrains with vertical flightpaths, a data structure with space complexity $O(n\lambda_4(n))$ that answers ray-shooting queries in time $O(\log n)$ can be built in preprocessing time $O(n\lambda_4(n)\log n)$.*

Proof: We first divide $\mathcal{S}$ into wedge-shaped *strips* by cutting outwards from f along a plane of constant θ through each vertex of $\mathcal{S}$. We shall build a separate search structure for each strip. Building an initial search structure for a strip can be accomplished in time $O(n\log n)$ and finding the strip for a given ray-shooting query takes time $O(\log n)$, so we may treat strips separately. (A unified structure, however, should be an improvement in practice.)

Now consider the polygonal subdivison $D(\theta_0)$ in the dual space of rays for the minimum longitude θ_0 in a strip as in Figure 5(b). $D(\theta_0)$ gives an unbalanced binary search tree T_{θ_0} by defining a node for each vertex of $D(\theta_0)$ and adding edges between nodes that correspond to adjacent vertices, as shown in Figure 6(a). Each node of T_{θ_0} then corresponds to a ray of $D(\theta_0)$, namely the one with origin at the corresponding vertex. In searching tree T_{θ_0}, an $O(1)$-time test at each node determines whether a query point (t, ϕ) lies above or below the line through the ray corresponding to the node. Notice that such a search tree remains invariant as $D(\theta)$ transforms smoothly.

We now show how to create a balanced search tree using "parallel tree contraction", a technique used in the design of parallel algorithms. Following Miller and Reif [16], we define

12

Figure 6. (a) Polygonal subdivision tree T_{θ_0}. (b) As merged by *Rake* and *Compress*. (c) Balanced search tree R_{θ_0}.

an operation *Rake* on rooted trees that merges each leaf with its parent. Call a connected set of degree-2 nodes in a tree a *path*; a node is called a path node if it lies on a path. Define an operation *Compress* that merges adjacent pairs of path nodes simultaneously all over the tree. Any set of adjacent pairs may be chosen, so long as any set of 4 successive vertices along a path contains a pair that merge. This is a nondeterministic, generalized version of *Compress*; the ordinary version merges successive pairs. The proof of Miller and Reif [16] immediately generalizes to show that any n-node tree is reduced to a single node after at most $c \cdot \log n$ alternating applications of *Rake* and *Compress*, where c is a constant.

We alternately apply *Rake* and *Compress*, starting with *Rake*, to T_{θ_0} until we obtain a single supernode, as shown in Figure 6(b). Here a dashed oval represents a merging due to *Rake* and a solid oval a merging due to *Compress*; numbers indicate the order in which supernodes merge. For simplicity, the *Rake* operation numbered 1 is not shown; *Compress* operations 6 and 8 do nothing.

We can define a new search tree level-by-level by considering each combined supernode as the parent of the combining supernodes. Each internal node in the new search tree R_{θ_0} results from the merger of two supernodes along an edge of T_{θ_0} or from the merger of two leaves and their parent. Thus at least one of the child supernodes corresponds to a proper subtree of T_{θ_0}. A proper subtree of T_{θ_0} corresponds, in turn, with a roughly wedge-shaped unbounded polygon in D_{θ_0}. This polygon has a lower boundary that is a ray and an upper boundary that is a convex chain. For example, the root of R_{θ_0} in Figure 6(c) corresponds to merger 9 in Figure 6(b), which is along the edge between the nodes labeled r_1 and r_8 in Figure 6(a), which in turn corresponds to the edge between v_1 and v_8 in Figure 5(b). The associated wedge has vertex v_8 and an upper boundary formed by r_8 and r_{11}.

We augment each internal node of R_{θ_0} with the following information:

(I1) the coordinates (t, ϕ) of the leftmost vertex v_i of the corresponding wedge-shaped polygon in $D(\theta_0)$ (as named in Figure 6(c));

(I2) the slope of the polygon's lower boundary; and

(I3) the largest slope of a boundary segment of the wedge-shaped polygon.

Notice that (I1), (I2), and (I3) will vary predictably with θ once longitude is unfixed. This information allows an $O(1)$-time "within-wedge" test to determine whether a given point

13

Figure 7. Before and after a rotation in T_θ.

query (t, ϕ) lies in the left or right subtree of a node in R_{θ_0}. Points in the polygon P_i immediately above the wedge-shaped polygon may go either way in this test. For example, a point just above the line segment between v_6 and v_7 in Figure 5(b) may go either way when tested at the node marked v_3, depending on whether it falls to the right or left of a line through v_3 with the same slope as r_7. Say this point tests inside v_3's and v_5's wedges, outside v_7's wedge, and finally inside v_6's wedge; then a single, final test determines whether the point lies in P_6 or P_8. These extra tests are indicated at the leaves in Figure 6(c); thus the number of tests needed for point location may be one more than the height of R_{θ_0}. In Figure 6(c), i marks the direction to take if a point tests *in* the wedge. Altogether point location for queries at longitude θ_0 can be accomplished using tree R_{θ_0} in $O(\log n)$ time.

Search tree R_{θ_0} is actually valid for all θ until the next critical longitude. At a critical longitude, either the strip ends or a rotation occurs in tree T_θ. We now show that by changing only $O(\log n)$ nodes and edges of R_θ at a rotation of T_θ, we can maintain the invariant that R_θ is a tree that could have resulted from T_θ by an alternating sequence of *Rake* and *Compress* operations.

A generic rotation is shown in Figure 7, with the T_θ trees shown before and after a critical longitude. (Of course, before and after could be reversed.) After an alternating sequence of *Rake* and *Compress* operations, call a supernode in the left tree *clean* if it contains neither y nor z and is not the parent of a supernode containing y. After any number of *Rake* and *Compress* operations, there are at most 3 unclean supernodes, and they induce a path in the left tree.

Assume inductively that each clean supernode on the left, except at most one, has a *counterpart* on the right, that is, a supernode containing exactly the same set of original nodes of T_θ. This condition certainly holds before any *Rake* and *Compress* operations have been performed. Now consider applying *Rake* to both the left and right trees. The counterparts of each pair of clean supernodes that merge on the left will merge on the right, since the adjacencies of clean supernodes and their counterparts are identical. A supernode on the left that results from a merger including an unclean supernode is itself unclean. And finally, a supernode on the left that results from a merger including a clean supernode without a counterpart, will reproduce the one allowable clean supernode without a counterpart.

Now consider applying *Compress* to the left. We assert that there exists a valid *Compress* for the right tree that maintains counterparts for each clean supernode. We join the counterparts of each merging pair of clean supernodes in this *Compress*. The pairing of other supernodes on the right is then dictated by this earlier pairing. For example, if A and B are both single nodes in Figure 7, then the first *Compress* on the left may combine

14

x and y but x' may have to remain unchanged on the right. The next merger above x and y, however, can be mimicked on the right. As in this example, the pairing on the right may leave gaps, that is, the merging pairs may be nonsuccessive along a path, but gaps of one are legal in our nondeterministic version of *Compress*.

There is also the case that the *Compress* on the right must merge the counterparts of a pair that did not merge on the left. Thus a single clean supernode on the left can lose its counterpart on the right. This loss cannot be repeated, however, until it has been reversed (i.e., until every clean supernode on the left has regained a counterpart), since the forced merger on the right only occurs when the length of the path from the root on the right to the supernode containing x' is one more than the length of the path from the root on the left to x. Thus after any number of *Rake* and *Compress* operations, there is a one-to-one mapping that takes all but one clean supernode on the left to a counterpart on the right; except for $O(1)$ nodes on the right this mapping is onto.

Altogether we conclude that only $O(1)$ supernodes in each level of search tree R_θ must change at a critical longitude. Furthermore, information (I1), (I2), and (I3) can be updated in time $O(1)$ per changed supernode by consulting that information at children of the changing supernode.

All changes to R_θ at a critical longitude lie along $O(1)$ root-leaf paths. Thus these changes can be performed persistently [6] to give a data structure that can answer ray queries for arbitrary θ within the strip. Altogether we obtain an $O(\log n)$-time search structure for each strip of the scene. ∎

Remark. An anonymous referee pointed out that parallel tree contraction methods that do not use *Compress* [1, 13] should give somewhat simpler proofs of Theorem 6. We are not sure which parallel tree contraction method gives the most satisfactory data structure, and we leave this question to interested readers.

8. Conclusions

We have given algorithms for some natural computer graphics problems that have not received sufficient attention. There are numerous possibilities for improvements to our algorithms. We list some specific open questions that we find intriguing.

- Can an unsorted list of points at which transparent topology changes occur be computed in time $O(n^2 + \ell)$?

- Can the external contour of a union of triangles (or curved triangles) be found in time faster than the total number of intersections of sides? (It appears that Mulmuley's randomized methods give a positive answer to these questions, with running time proportional to a sum in which each intersection contributes the reciprocal of one more than the number of polygons strictly containing it [18]. This would improve the running time of the algorithm given after Theorem 2.)

- Can all opaque topology changes for general scenes be found in time sensitive to k? (The analogous question for static viewpoints is the longstanding, largely open, question of finding an output-sensitive hidden line removal algorithm [19].)

- Can the "sensitivity"—i.e., the term involving k—of our algorithm for terrains with arbitrary flightpaths be improved?

- Can ray-shooting queries for general scenes be answered in time $O(\log n)$? Even in the special case of no opaque topology changes along f?

- Can our results be generalized to linearly moving objects?

References

[1] K. Abrahamson, N. Dadoun, D. K. Kirkpatrick, and T. Przytycka, A simple parallel tree contraction algorithm, *Proc. 25th Annual Allerton Conf. on Comm. Control, and Computing*, 1987, 624–633.

[2] P. K. Agarwal, *Intersection and Decomposition Algorithms for Planar Arrangements*, Cambridge University Press, 1991.

[3] M. J. Atallah, Some dynamic computational geometry problems, *Computers and Math. with Applications* 11 (1985), 1171–1181.

[4] J. L. Bentley and T. A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. on Computers* 28 (1979), 643–647.

[5] R. Cole and M. Sharir, Visibility problems for polyhedral terrains, *J. Symbolic Computation* 7 (1989), 11–30.

[6] J. R. Driscoll, N. Sarnak, D. Sleator, and R. E. Tarjan, Making Data Structures Persistent, *J. Computer and Systems Sciences* 38 (1989), 86–124.

[7] H. Edelsbrunner, J. O'Rourke, and R. Seidel, Constructing arrangements of lines and hyperplanes with applications, *SIAM J. Computing* 15 (1986), 341–363.

[8] H. Fuchs, Z. M. Kedem, and B. F. Naylor, On visible surface generation by a priori tree structures, *Computer Graphics* 14 (1980), 124–133.

[9] S. Hart and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, *Combinatorica* 6 (1986), 151–177.

[10] J. Hershberger, Finding the upper envelope of n line segments in $O(n \log n)$ time, *Inform. Proc. Letters* 33, 1989, 169–174.

[11] H. Hubschman and S. Zucker, Frame-to-frame coherence and the hidden surface computation: constraints for a convex world, *Computer Graphics* 15 (August 1981), 45–54.

[12] J. W. Jaromczyk and M. Kowaluk, Skewed projections with an application to line stabbing in R^3, *Proc. 4th ACM Symp. on Comp. Geometry*, 1988, 362–370.

[13] S. R. Kosaraju and A. L. Delcher, Optimal parallel evaluation of tree-structured computation by ranking, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing*, 1988, 101–110.

[14] D. T. Lee and F. P. Preparata, *Location of a point in a planar subdivision and its applications*, *SIAM J. Computing* 6 (1977), 594–606.

[15] M. McKenna, Worst-case optimal hidden surface removal, *ACM Trans. Graphics* 6 (1987), 19–28.

[16] G. L. Miller and J. H. Reif, Parallel tree contraction and its applications, *Proc. 26th IEEE Foundations of Comp. Science*, 1985, 478–489.

[17] K. Mulmuley, A fast planar partition algorithm, I, *Proc. 29th IEEE Foundations of Comp. Science*, 1988, 580–589.

[18] K. Mulmuley, On obstructions in relation to a fixed viewpoint, *Proc. 30th IEEE Foundations of Comp. Science*, 1989, 592–597.

[19] M. Overmars and M. Sharir, Output-sensitive hidden surface removal, *Proc. 30th IEEE Foundations of Comp. Science*, 1989, 598–603.

[20] M. Paterson and F. F. Yao, Binary partitions with applications to hidden surface removal and solid modelling, *Discrete Comput. Geometry* 5 (1990), 485–504.

[21] W. H. Plantinga and C. R. Dyer, An algorithm for constructing the aspect graph, *Proc. 27th IEEE Foundations of Comp. Science*, 1986, 123–131.

[22] W. H. Plantinga, C. R. Dyer, and B. Seales, Real-time hidden-Line elimination for a rotating polyhedral scene using the aspect representation, manuscript, 1988.

[23] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[24] F. P. Preparata and R. Tamassia, Fully dynamic point location in a monotone subdivision, *SIAM J. Computing* 18 (1989), 811–830.

[25] J. H. Reif and S. Sen, An efficient output-sensitive hidden-surface removal algorithm and its parallelization, *Proc. 4th ACM Symp. on Comp. Geometry*, 1988, 194-200.

[26] M. Sharir, Almost linear upper bounds on the length of general Davenport-Schinzel sequences, *Combinatorica* 7 (1987), 131–143.

[27] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden-surface algorithms, *Computing Surveys* 6 (1974), 1-25.

[28] G. R. Swart, A schema for real time hidden line removal, Tech. Report, Dept. of Computer Science, U. of Washington, 1984.

[29] A. Wiernik and M. Sharir, Planar realization of nonlinear Davenport-Schinzel sequences by segments, *Discrete Comput. Geometry* 3 (1988), 15–47.