Shadow Computation: A Unified Perspective

S. Ghali[†], E. Fiume[‡], and H.-P. Seidel[†]

Abstract

Methods for solving shadow problems by solving instances of visibility problems have long been known and exploited. There are, however, other potent uses of such a reduction of shadow problems, several of which we explore in this paper. Specifically, we describe algorithms that use a resolution–independent, or object–space, visibility structure for the computation of object–space shadows under point, linear, and area light sources.

The connection between object–space visibility and shadow computation is well–known in computer graphics. We show how that fundamental observation can be recast and generalized within an object–space visibility structure. The edges in such a structure contain exactly the information needed to determine shadow edges under a point light source. Also, the locations along a linear or an area light source at which visibility changes (termed critical points and critical lines) provide the necessary information for computing shadow edges resulting from linear and area light sources. Not only are instances of all shadow problems thus reduced to visibility problems, but instances of shadow problems under linear and area light sources are also reduced to instances of shadow generation under point and linear light sources, respectively.

Keywords: Shadows, Shadow Algorithms, Object-Space Visibility, Point, Linear, Area Light Sources, Critical Point, Critical Line, Radiosity.

1. Introduction

The connection between visibility algorithms and shadow algorithms has long been known, but has not been fully explored. The interest of the graphics community in algorithms geared to raster graphics devices has swelled as such devices became abundant. But even though non–raster based visibility algorithms can be avoided in most areas of computer graphics, one problem tenaciously resists solutions based on image–space visibility computation. In this paper, we consider the use of an object–space visibility data structure for shadow generation.

The inadequacy of image–space visibility for shadow generation has been made as early as 1977 by Crow[11] who writes: "Thus image–space algorithms which depend on the limited resolution of the display medium to ease the determination of hidden surfaces are inappropriate for [the shadow generation] application." The approach we take falls in the second category of Crow's classical taxonomy. Namely, the computation is performed in two passes, the first of which determines the shadow edges on the polygons in the scene. Under illumination by point, linear, and area light sources, we describe algorithms to determine the shadow edges and provide for each shadow edge the set of scene edges that define it.

In another classical paper,[1] Atherton, Weiler, and Greenberg describe an object–space shadow generation algorithm. The first pass in their algorithm uses polygon clipping[60] to determine the polygons visible from the light source.[67] In this paper, we revisit their algorithm and describe how an implementation that does not rely on polygon clipping can be achieved. Avoiding polygon clipping, such as the one originally used[60] or a more recent one,[66] is desirable for at least two reasons. First, as Devai[13] observes, visibility computation based on polygon clipping may take time cubic in the number of input edges. Such a case may arise if the size of the mask used for clipping from front to back reaches quadratic size after only half the polygons have been processed. A second reason to avoid polygon clipping is a systems one. Advanced geometric structures, such as the ones developed in the context of the CGAL library,[6] are likely to become standard tools in the years to come. An algo-

† Max Planck Institute for Computer Science, Saarbrücken
‡ Department of Computer Science, University of Toronto, Toronto

© The Eurographics Association 2000.

rithm that encapsulates the visibility step makes it easier to develop systems for object–space shadow computation by decoupling visibility and shadow generation. In particular, it would be possible to take advantage of future research in object–space visibility computation by reconnecting software modules. We see these two directions as synergistic: by showing uses of an object–space visibility structure, it is possible that the graphics community will have a practical reason for developing more robust and more efficient visibility map algorithms.

1.1. Previous Work

Since the output device in graphics applications is nearly always a raster device, a large proportion of visibility algorithms are developed to take advantage of the finite and discrete size of the output device. Algorithms such as Watkins, or scanline, and Z–buffer fall under the point sampling category of image–space algorithms in an early and highly influential taxonomy paper.[61] The abundant use of these and other image–space visibility algorithms and their implementation in hardware led to their use for the computation of visibility from a point light source. But even if the output device of the final image is discrete and finite, handling shadows is one problem in computer graphics that is hard to tackle using image–space techniques. Using a shadow buffer,[69, 28, 53, 70] which is a discrete image from the point of view of the light source, makes it possible to determine whether a point sample in the scene is in light or in shadow. The resulting image is prone to light buffer (or shadow buffer) aliasing, however, if the object casting the shadow is too close to the light source. Another persistent problem is that of coupling the sampling rate of the image to that of the light buffers used in rendering. This is a particularly acute problem in animated environments in which very high sampling rates must often be used for many frames so as to prevent artifacts that arise when varying buffer resolutions are employed.

Increasing the resolution of the light buffer will reduce artifacts, but computing visibility in object–space[67, 14] effectively increases the resolution of the light buffer to machine precision. Unfortunately, this approach has not been widely adopted, perhaps because of the difficulty of implementing, for example, a robust polygon clipping algorithm. In practice, it appears that the most popular technique is one that concentrates on the shadows as a visual cue giving the height above a single ground plane.[4] This technique, which is particularly popular in interactive graphics, does not suffer from aliasing, but is of course only an approximation which is why such shadows are dubbed "fake" shadows[4]: objects cannot cast shadows on each other but only on one or more prespecified "ground" planes. The additional geometry added to provide the visual cue of shadows may also be expensive to render. The resulting shadows are less prone to aliasing than shadow buffers since the projection on the ground plane occurs before the scanline conversion.

Methods for computing object–space visibility other than the one used by Weiler and Atherton,[67] which is based on polygon clipping,[60, 66] exist. For a scene consisting of n edges, a trivial lower bound for the time to compute object space visibility is $\Omega(n \log n)$. This holds for any of the three versions of the problem that may be addressed:

Visible Line Determination Compute the set of line segments visible. Algorithms for this problem were particularly suitable for vector–based displays, plotters, or other graphical output device that make it difficult or impossible to perform shading.

Visible Surface Determination Compute the set of polygons visible. The output is suitable for display on a device capable of shading such as a raster frame buffer.

Visibility Map Compute the set of cross–linked visible polygons, i.e. each pair of edges that appear on the image plane on the boundary of two polygons are cross–referenced. We are content with this definition for the visibility map at this point until we define it more carefully in Section 2.

The first problem, Visible Line Determination, is equivalent to the ones that Sutherland et al.[61] classify as object–space algorithms. VLD algorithms are not useful for shadow computation, however, as their output contains too little information. See Figure 1.

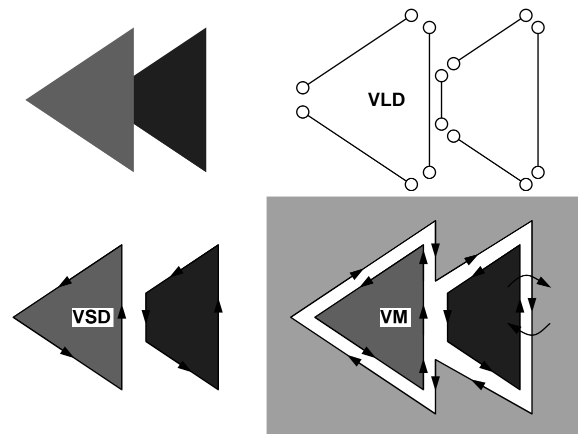

Figure 1: *An view of two triangles; the visible lines; the visible surfaces; the visibility map (only one edge adjacency is illustrated on the right).*

Weiler and Atherton's visible surface determination[67] is an example of an algorithm in the second category, that of Visible Surface Determination. Binary Space Partitioning[18, 51] also produces visible surfaces in object space. The output from a BSP algorithm is only a depth–sorted listed of polygon fragments, however, and so it does not fit in the classes of visibility problems above. Chin and Feiner use a algorithm based on binary space partitioning to determine object–space shadows under a point[7] and an

© The Eurographics Association 2000.

area[36] light source. The use of BSP precludes the generation of the conic shadows that arise in illumination under a linear or an area light source.

We point in this paper to applications of the visibility map for the computation of shadows. The visibility map is a rich data structure that contains sufficient information for the computation of shadows under point, linear, and area light sources using seamless approaches in the three cases.

Another problem that turns out to be central in this work is the off–line determination of the set of critical points encountered by a viewer when moving along a straight–line segment. The *off–line* version of this problem is one in which the viewpoint path is known as part of the input. This problem has been surprisingly under–studied, with only two algorithms known to date (to the knowledge of the authors): one by Mulmuley[47] and the other by Bern et al.[3]

The problem of computing critical points off–line is intimately related to the computation of aspect graphs. We only briefly discuss aspect graphs here as we return to them in Section 4.1.1. Three versions of the aspect graph problem can be stated:

Orthographic Projection from an Infinite Sphere In this problem, a viewpoint moves on a sphere at infinity and it is desired to partition the surface of the sphere into maximally contiguous regions such that the view of the scene is the same when the viewpoint remains inside that region.

Perspective Projection The problem is to partition the space into regions such that the view seen by an observer does not change as the observer moves inside a single region.

Perspective Projection from Polygons in the Scene This problem shares part of the two previous problems; the viewpoint space is (one side of) a polygon in the scene (so the observer can move with only two degrees of freedom) but the view seen by the observer is a perspective one.

The first two categories parallel the main branches in Bowyer and Dyer's taxonomy.[5] The third class of problems is the more interesting for shadow generation. Durand et al.[16] describe a data structure that partitions the polygons in a scene by constructing the adjacency relationships between *four–edge stabbers* (other work addresses the problem in the plane[52]). These stabbers have been introduced to the community by Teller[65] who called them *extremal lines* (see also Teller and Hohmeyer[63, 64] for an algorithm to compute the common stabber of four lines in space). To define a line in space, it is necessary and sufficient to specify either two points in space, a point and two lines, or four lines. Since a graphics system will often not define a point by itself in space but as a vertex of a polyhedron (or at least a polygon), it is convenient to assume that the first two cases above are only special cases of the four–edge stabber; in the first, two pairs of edges intersect at a vertex and in the second, one such pair arises. These four–edge stabbers are crucial for the algorithms we describe below as each endpoint of a shadow edge (under point, linear, or an area light source) is defined by a four–edge stabber. Here again, the visibility map provides the necessary information to define shadow edge endpoints *symbolically*. By that we mean that not only do we know the location of these endpoints in space, but we also know which scene edges gave rise to them. This information is crucial since the computation of the arrangement of shadow edges would not rely on the coordinates in space to match shadow edge endpoints, but would rely instead on the symbolic description of these endpoints.

We start the next section by describing the winged–edge data structure.[2] This data structure makes it possible to represent both the topology of the scene as well as the topology of the visibility map. By storing the connectivity information between the edges and the faces of the *scene*, the input consists of a collection of solids rather than a collection of polygons. This makes it possible, e.g., to traverse the surface of each solid and to easily determine whether two polygons on a solid are adjacent. Such an input consisting of solids does not forbid the representation of a polygon in space. A polygon can be represented as a lamina[41] with *two* faces. Such a scene representation is particularly crucial for visibility and shadow computation as it preserves the integrity of common heuristics such as backface culling. We discuss this point further in Section 5.

2. Generating Shadow Edges from a Point Light Source

2.1. Computation of the Visibility Map

We start this section by briefly surveying results from the literature to compute the visibility map before describing how the visibility map can be used to determine the set of shadow edges. The first phase of the research described here involved identifying which algorithms in the literature are amenable to an implementation. To be specific, we start by giving details of the data structure holding the topology of the visibility map and then briefly describe the algorithms of Schmitt,[55] of McKenna,[43] and of Goodrich[25] in the context of that data structure.

The data structure `PlanarMap` consists of a set of nodes, a set of directed arcs, and a set of faces. Each node holds a pointer to an arc outgoing from it (i.e., the node is the source of the arc). Each face also holds a pointer to an arc on its boundary. The `PlanarMap` data structure is centered on its arcs; each arc holds a reference to its source node, its target node, the face on its side, the edge succeeding it and the edge preceding it on the same face, and its dual edge, or the edge that is adjacent to both nodes but directed in the opposite direction. See Figure 2 for an illustration. Strictly speaking, this description contains two redundant pointers but it is convenient and more (time) efficient to include them. This data structure is essentially the same as the well–known winged–edge data structure pioneered by Baumgart[2] who used it to

© The Eurographics Association 2000.

represent polyhedra. We minimized the code development by using this data structure to represent both the scene and the visibility map. By using the C++ declaration

template⟨**class** V, **class** E, **class** P⟩
class PlanarMap {···};

for the `PlanarMap` data structure, it is possible to use the parameterization to both represent the scene topology and the visibility map topology. In this paper, we use the terms node, arc, and face to signify the topological relationships represented in `PlanarMap` and use the terms vertex, edge, and polygon to signify the geometric relationships in a particular instantiation of the generic `PlanarMap` structure either in the projection of the scene on the image plane or in the scene itself.

A variation of the winged–edge data structure has been reinvented by Muller and Preparata[46] who use it to determine efficiently the intersection of two polyhedra. Another variation on that structure was described by Guibas and Stolfi[27] who capitalize on the duality between nodes and faces; for example, just as the list of arcs outgoing from a node can be listed in order so the list of arcs adjacent to a face can also be determined in order. Kettner[37] dicusses the compromises between these representations.

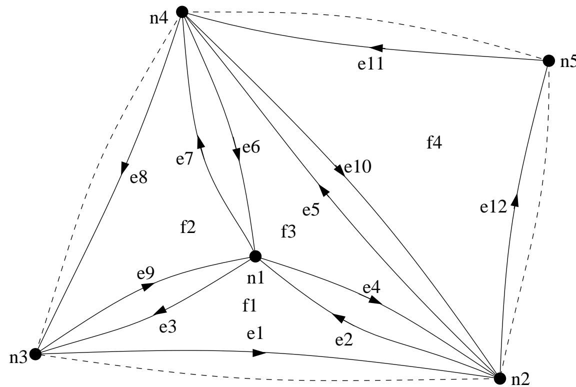

Figure 2: *The data structure* `PlanarMap` *is used to represent a planar map. The arcs are oriented counter–clockwise around a face. Each pair of arcs adjacent to the same pair of nodes but oriented in opposite directions are called duals (e.g. $e2$ and $e4$). For a node such as $n3$, it is possible to report one outgoing arc at it such as $e9$. For an arc such as $e2$, it is possible to report its* successor $e3$*, its* predecessor $e1$*, its* source $n2$*, its* target $n1$*, its* dual $e4$*, and its adjacent face $f1$. For a face such as $f1$, it is possible to report an edge defining it such as* e1 *(or* e2 *or* e3*). Since the sequence of edges $\langle e1, e2, e3 \rangle$ defines the face $f1$, the successor/predecessor relationships are uniquely defined for these edges.*

The algorithm of Schmitt[55] and that of Goodrich[25] start by computing the graph of line segment intersections on the image plane. That of McKenna,[43] on the other hand, computes the graph of intersections of the lines supporting the line segments on the image plane, thus taking guaranteed quadratic time for even simple scenes. As Dorward points out in a thorough survey,[14] starting the visibility map computation by either of these steps may be highly inefficient; a single polygon may hide a view of size quadratic in the size of the input. It is thus desirable to find an algorithm that computes the visibility map in time proportional to the size of the output without determining the intersection of invisible features. The best known algorithm for computing the visibility map is due to de Berg.[12] His algorithm runs in time proportional to the size of the output (up to a logarithmic factor). Unfortunately, his algorithm only proves the theoretical upper bound as it does not seem amenable to an implementation.

The algorithm by Goodrich,[24, 25] which builds on the work of Schmitt,[55, 56] proceeds in three passes. After the first pass that computes the graph of line segment intersections on the image plane, the second pass computes a depth order of the polygons in the scene. This is done by a step determining the "hide" relationship between pairs of polygons in the scene and is followed by a topological sort[39] to establish the depth order. The third pass performs a simulation of a painter–style algorithm[48] in object–space: as each polygon is "drawn," each edge inside it is marked invisible. If a cyclic depth ordering is discovered during the second pass, the user is notified to break the polygons and restart. By comparison, McKenna's algorithm is capable of handling cyclic overlap at the expense of taking guaranteed quadratic time even when the wire frame projection of the scene edges do not result in a quadratic number of intersections. Goodrich reports that, up to a logarithmic factor, his algorithm is proportional to the number of edges in the input, the number of *wireframe intersections* on the image plane prior to marking hidden edges, and to the number of polygon–polygon intersections on the image plane.

fIn a scene consisting of non–intersecting polyhedra, *real* vertices arise in the visibility map as a result of the projection of polyhedral vertices; *apparent* vertices also arise in the visibility map as a result of the apparent intersection of edges on the image plane. Defining these data structures in an object–oriented language such as C++ is easily achieved; define classes for real and apparent vertices as derived from a base vertex class and use the base class to replace **`class`** `V` at the time of instantiating the template `PlanarMap` class described above. Figure 3 illustrates the two types of vertices that arise in the visibility map from a point. A third type of vertex, which does not arise in the view but is inserted for convenience, is described below.

In the context of reusing the winged–edge data structure for both the scene and the visibility map, one detail is crucial. As discussed by Mäntylä,[41] nodes, arcs, and faces are not the only possible constituents to represent the topology

© The Eurographics Association 2000.

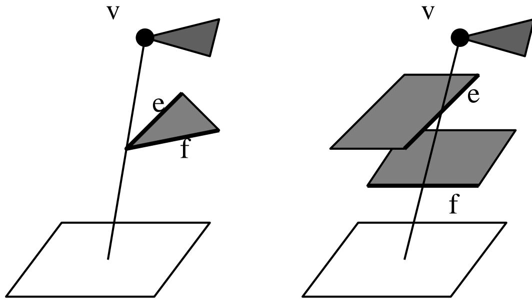

Figure 3: *The vertices in the visibility map seen by a viewer at v may be either real vertices as shown on the left or apparent vertices as shown on the right.*

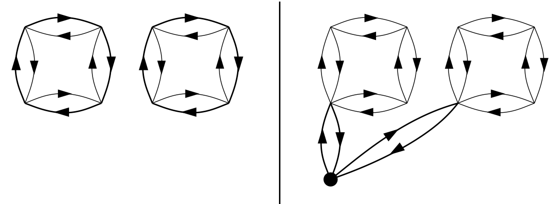

Figure 4: *To represent the outer face in the figure on the left, our data structure needs to handle polygons with holes. As shown on the right, this is remedied by using additional vertices and edges.*

in a solid modeler. A *ring* may be used to represent polygons with holes; a *shell* may be used to represent solids with voids not connected to the outside shell; and *solids* may be used to represent the collection of shells needed to define one solid. The reason that we have to consider these issues is that the topology of the visibility map can easily consist of a face with more than one sequence of edges. Such an example is shown on the left of Figure 4. The two polygons that appear in the visibility map result in three faces in that map: two to represent the polygons themselves and one to represent the outer or *background* face. This face is clearly defined by two cycles of four edges (the four edges dual to the ones defining the interior faces). The `PlanarMap` data structure described above is unable to represent the outer face. This is only a problem for the use of `PlanarMap` for visibility maps and not for solids. Indeed, as is well known,[41, 35] it is possible to represent a polygon with holes using a winged–edge data structure by simply adding extra edges connecting the outer–most face with the hole or holes. This idea has been used by Schmitt[55, 56] to represent visibility maps using a winged–edge structure. He introduces the notion of a *drain vertex* and a *drain edge*. In a visibility map, a drain edge is an extrinsic edge added between two sequences of edges describing the boundary of a single polygon. A drain vertex is an extrinsic vertex added for the necessary adjacency to a drain edge. One particular type of a drain vertex is the vertex at infinity.[55, 25] This vertex is conceptually at the point $(0, -\infty)$ and, as is shown on the right of Figure 4, is inserted to make it possible to use the `PlanarMap` structure to represent this configuration of a visibility map.

Figure 5 shows an example of a visibility map. Real vertices, which appear as solid dots in the figure, are the projection of polyhedral vertices on the image plane. The real vertex object contains a pointer referencing the polyhedral vertex object. Apparent vertices result from the apparent intersection of two polyhedral edges in the image plane. Each apparent vertex object stores a pointer to the pair of polyhedral edges that give rise to it.

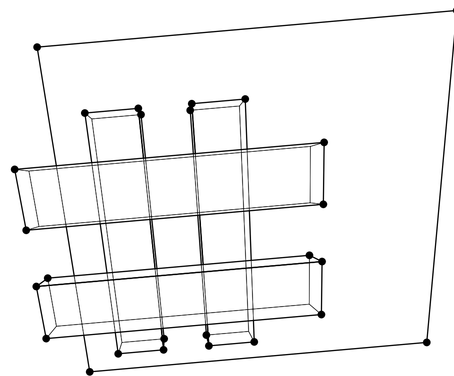

Figure 5: *The visibility map from a point light source. Real vertices are represented by solid circles.*

2.2. A Shadow Edge under a Point Light Source

A shadow edge is the locus of the points on a scene polygon such that the points on one side of the edge can see the point light source whereas those on the other side cannot. We show in the next section how the shadow edges can be determined following the computation of the visibility map. We start by arguing in this section that five scene edges (in addition to a polygon) are necessary and sufficient to describe a shadow edge. If we call the two points bounding a shadow edge on either side its endpoints, it is easy to see that two types of endpoint may arise. These two types directly parallel the two cases of vertices appearing in the visibility map shown in Figure 3. The key point is that each endpoint is defined by four scene edges and that for any one shadow edge, the two pair of quadruples will have three edges in common: two edges define the point light source and one edge define the edge casting the shadow. Figure 6 illustrates a shadow edge, its definition by three edges, and the definition of its endpoints by two four–edge stabbers.

If the two endpoints are defined by the quadruples e_1, e_2, e_3, λ_1 and e_1, e_2, e_3, λ_2, then the point at which the point light source is inserted is defined by two of the edges

© The Eurographics Association 2000.

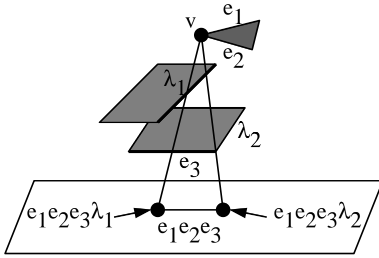

Figure 6: *The shadow edge shown is cast by edge f. We say that this edge is defined by a, b, f and that its endpoints are defined by the four–edge stabbers a, b, f, e and a, b, f, g.*

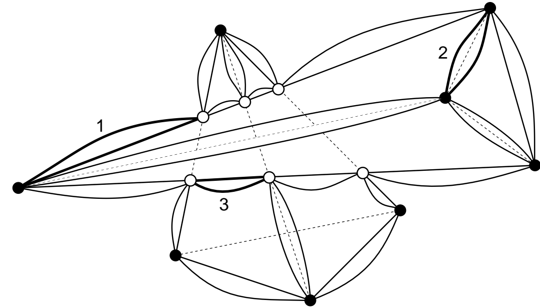

Figure 7: *The visibility map makes it possible to generate shadow edges and define them symbolically.*

e_1, e_2, e_3 and the third in that list is the polyhedral edge casting the shadow. There are two advantages in preventing the user from inserting point light sources *other than* at scene vertices: On one hand, this makes it possible to define all shadow edges under a point light source (as well as under a linear and an area light source as shall be described in Sections 3 and 4) by five scene edges. On the other hand, and as will be shown in Section 3, this makes determining the shadow edges under a linear light source simple by reducing the problem to a collection of problems of lower dimensionality. We now describe an algorithm to deduce the set of shadow edges resulting from illumination by a point light source after the computation of the visibility map.

2.3. Determining Shadow Edges under Illumination from a Point

To determine the shadow edges under illumination from a point light source, we start by computing the visibility map from a viewer located at the position of the point light source. The shadow edges can then be determined by the following algorithm. For each pair of edge duals in the visibility map, we consider the two faces adjacent to them in the visibility map. As shown in the projection of the two tetrahedra in Figure 7, three cases may arise:

1. If one of the two faces is the background plane, no shadow edge is generated.
2. If the two faces adjacent to the edge duals are also adjacent in the scene, no shadow edge is generated.
3. If neither of the two faces adjacent to the edge duals is the background face and the two faces are not adjacent in the scene, a shadow edge is generated.

The shadow edge is "deposited" on the polygon farther from the viewpoint by inserting it into a list of shadow edges. Even though the three edges e_1, e_2, and e_3 that define such a shadow edge can be defined by the edge casting the shadow and two of the edges adjacent to the point light source, it is useful in practice to cache the actual point in space at which the light source is inserted by explicitly storing it. To determine the two edges λ_1 and λ_2 delimiting the shadow edge, we define a method called `getOtherEdge` for the vertices in the visibility map. When called for a vertex v, this method takes as parameter a single edge e adjacent to v and returns another edge adjacent to v in the visibility map that is not the projection on the image plane of the same edge in the scene. If more than one edge is adjacent, any one may be chosen, though the calculation to follow of the coordinates of the two endpoints of a shadow edge will be more reliable if we choose the edge closest to perpendicular to the edge passed as parameter. This method is illustrated in Figure 8.

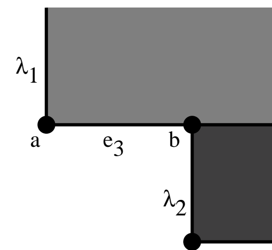

Figure 8: *The visibility map contains enough information to determine shadow edges symbolically by using a method* `getOtherEdge`. *For the case shown above,* a`.getOtherEdge(`e_3`)` *returns* λ_1 *and* b`.getOtherEdge(`e_3`)` *returns* λ_2.

Since the main loop in the procedure described above iterates over the edges of the visibility map and since it takes time proportional to each edge's adjacencies to determine whether each such pair defines a shadow edge, the shadow generation procedure takes time proportional to the size of the visibility map. The bottleneck remains the computation of the graph of line segment intersections and the "hide" relationships between polygon pairs. Figure 9 shows an example of the shadow edges generated following the computation of the visibility map shown in Figure 5.

© The Eurographics Association 2000.

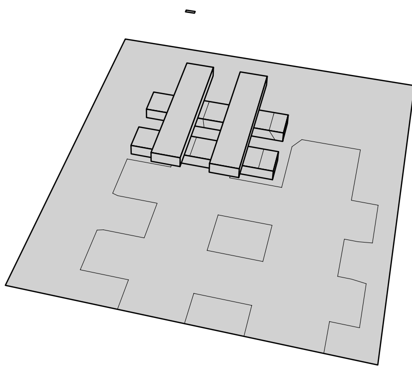

Figure 9: *Shadow edges are determined following the computation of the visibility map.*

3. Generating Shadow Edges from a Linear Light Source

We motivate the following discussion by showing an extension to illumination from a point light source. If a viewpoint moving on the linear light source sees the same visibility map, then the computation of the shadow edges is simple and, as we describe below, would consist of the projection of the edges in the visibility map from the endpoints and the projection of the vertices in the visibility map from the line segment defining the linear light source. Such an example is shown in Figure 10.

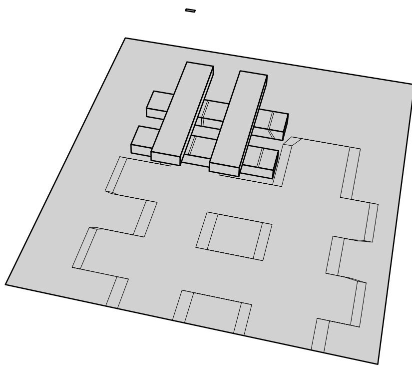

Figure 10: *The set of shadow edges resulting from illumination under a short, or critical point–free, linear light source consists of edges projected from endpoints and vertices projected from the edge defining the source.*

The problem becomes more difficult in the presence of critical points. Consider the simple scene shown in Figure 11. We would like to determine the shadow edges when this scene is illuminated by a linear light source inserted along the line segment shown at the top of the figure. Whereas a shadow edge in a scene illuminated by a point light source separates a region in light from a region in shadow, a shadow edge in a scene illuminated by a linear (or an area) light source separates two regions that see *qualitatively* different views of the light source. By "qualitatively," we mean to say that as our final objective is to compute the correct value of the illumination at any point on the surfaces of the scene, we would like to determine the regions from which a single evaluator is possible. In this context, an evaluator is a function well–known in illumination engineering (see, e.g., Higbie's text[34]) that makes it possible to determine the light energy falling on an arbitrary point by knowing the extent of the linear light source(s) that illuminate(s) it. The notion of shadow edges is further discussed elsewhere.[20]

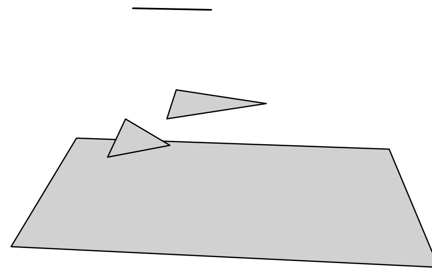

Figure 11: *Simple Scene illuminated by a Linear Light Source.*

With reference to the scene shown in Figure 11, we say that two points in the scene have different qualitative illumination if one of them sees the left endpoint of the linear light source and the other does not (of course this condition is sufficient but not necessary). This suggests that we compute the view of the scene from the left endpoint which results in the visibility map shown in Figure 12. The same algorithm described for illumination under a point light source now applies; a shadow edge is deposited on the farther of two polygons adjacent in the visibility map but not adjacent in the scene. Nishita and Nakamae[49] are generally credited by the community for making this important observation (see also Nishita et al.[50]). We simply add here a twist to show that the same idea used by Atherton et al. in other fundamental work[1] can be used to determine this subset of the shadow edges.

To introduce our next observation, we consider the visibility map seen by an observer located at the right endpoint of the linear light source shown in Figure 11. In that visibility map, which is shown in Figure 13, a portion of two scene edges are hidden (shown in thin lines). This also suggests

© The Eurographics Association 2000.

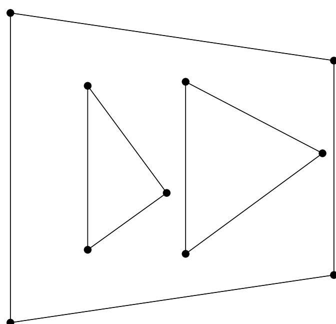

Figure 12: *Visibility map from a viewer located at the left endpoint of the linear light source shown in Figure 11.*

that the shadow edges arising from the right endpoint of the linear light source may be computed by reducing the problem by one dimension to an illumination from a point light source.

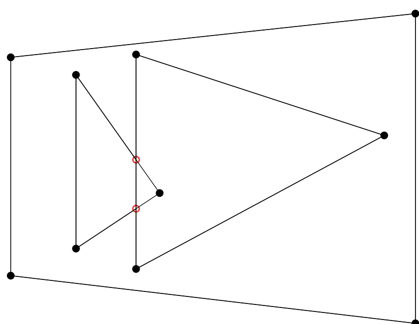

Figure 13: *Visibility map from a viewer located at the right endpoint of the linear light source shown in Figure 11. The edges shown in thin lines are hidden.*

As the viewer at the left endpoint moves gradually towards the right endpoint, the visibility map seen by the viewer eventually changes. Recall that the visibility map is a labeled planar graph where each vertex, edge, and polygon store pointers to the vertices, edges, and polygons in the scene. In other words, the visibility map does not contain information about the location of the viewer (or more precisely, this information is irrelevant). The point at which the visibility map changes has been called a *critical point* by the computer vision community in the context of the computation of the aspect graph (see Bowyer and Dyer for a survey on aspect graphs[5]).

We now state our next observation then argue for its correctness. Just as the *edges* in the visibility map define shadow edges under illumination from a point light source, the *vertices* in the visibility map define shadow edges under illumination by a linear light source. To see that this is true, consider two points a and b in close proximity on a polygon in the scene. Suppose that the portion of the linear light source that illuminates each of these two points is different because a *vertex* in the visibility map is crossed when we move from a to b. Indeed, the combination of an edge in the visibility map for the case of illumination under a point light source with the point light source itself as well as the combination of a vertex in the visibility map for the case of illumination under a linear light source each define a *critical surface*. As with critical points, critical surfaces are heavily studied by the vision community. Critical surfaces have been introduced to the graphics community by Heckbert[32, 30] and have been used in many other algorithms[40, 62, 58, 15, 59, 21] where they were called *discontinuity surfaces*. They were so called because they induce a discontinuity in the illumination function.[30] As we concentrate here on visibility and geometry rather than on illumination itself, we choose to adopt the older terms critical points and surfaces. This also reduces the gap between the graphics and vision communities who indeed seek similar understandings but with opposing objectives. We deviate in this and related work[20] from the previous approaches outlined above, however, in that we do not determine critical surfaces (enumeration of discontinuity surfaces) and therefore do not intersect critical surfaces with the scene geometry, the bottleneck step in the previous approaches.

3.1. Shadow Edges from an Endpoint of a Linear Light Source

As the definition of shadow edges resulting from the two endpoints of the linear light source are identical to the definition of shadow edges under a point light source, we only briefly mention here that computation. Each edge in the visibility map may induce a shadow edge in the scene. Such a shadow edge is defined by five scene edges: the edge holding the linear light source itself, another edge adjacent to the linear light source and to its endpoint, the edge generating the shadow, in addition to two delimiting edges resulting from a call to the method `getOtherEdge` described in Section 2.3. The set of shadow edges resulting from the two endpoints are shown in Figure 14.

3.2. Shadow Edges from a Portion between Critical Points of a Linear Light Source

As can be seen in Figure 15, when the viewer reaches the critical point on the linear light source, the view will change. This critical point is defined by a four–edge stabber, or an edge that simultaneously touches four edges in the scene. The shadow edges are generated independently for the two portions of the linear light source on either side of the critical point. For each of these two portions, imagine a line segment connecting a point moving on the linear light source while simultaneously touching the two edges adjacent to a vertex in the visibility map. This line leaves a trace on some

© The Eurographics Association 2000.

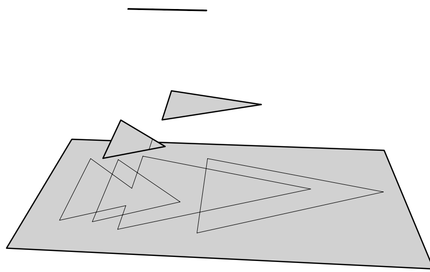

Figure 14: *The shadow edges resulting from illumination by two point light sources inserted at the endpoints of the linear light source are among those shadow edges that result when illumination is under the linear light source.*

polygon where the polygon can be determined from the labeled visibility map. For a real vertex in the visibility map, the shadow edge is a straight–line segment. For an apparent vertex in the visibility map, the shadow edge is defined by the intersection of a *swath*,[65] the locus of the points that are aligned with three skew edges, with the polygon on which the shadow edge falls. That polygon can also be easily determined from the labeled visibility map. Such a swath defines a ruled quadric surface in space and its intersection with a polygon is a conic. Such a conic shadow edge can degenerate into a straight–line segment even if the three edges defining it are truly skew. For that to occur, it suffices that the polygon and the swath intersect in a *directrix*, or a straight–line embedded in the surface of the ruled quadric. More details about the properties of these fascinating surfaces can be found in a standard treatise on classical geometry such as the one by Salmon.[54] Figure 15 shows the set of shadow edges that result in this example.

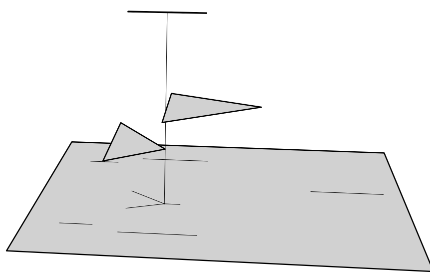

Figure 15: *The shadow edges shown are those that result from tracing the vertices in the visibility map as seen by a viewer moving along the linear light source.*

Just as not all edges that appear in the visibility map may generate shadow edges (such an edge must be a contour edge[11]), not all vertices that appear in the visibility map generate shadow edges. If the vertex is a real vertex, at least one of the faces adjacent to it in the visibility map must be not adjacent to it in the scene. In fact, a real vertex can have at most one such adjacent face (if we assume that the manifolds defining our polyhedra are homeomorphic to a disk everywhere[35] — which, intuitively, is satisfied if no two solids touch at a vertex). If the vertex is an apparent vertex, on the other hand, then both edges that define the vertex must be contour edges. In that case, the polygon that is *not* adjacent to either of the two edges is the polygon on which the shadow edge is cast. See Figure 16.

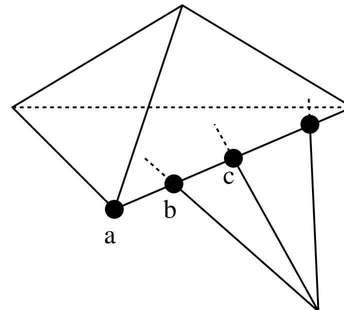

Figure 16: *Vertex a casts a shadow edge on the polygon adjacent to it in the map that is* not *adjacent to it in the scene. Vertex b, which is defined by the intersection of two scene edges, casts a shadow on the polygon adjacent to it in the map other than the two polygons adjacent to its two defining edges in the scene. Vertex c does not define a shadow edge and is the linear light source replication of edge 2 shown in Figure 7.*

These four sets of shadow edges together constitute the shadow edges in this scene when illuminated by an linear light source. Each shadow edge is defined by five scene edges $e_1, e_2, e_3, \lambda_1, \lambda_2$. The three edges e_1, e_2, e_3 define the critical surface giving rise to the shadow edge and the two delimiting edges λ_1, λ_2 define the endpoints. For a shadow edge thus defined, each of its two endpoints is described by a four–edge stabber[65]: e_1, e_2, e_3, λ_1 and e_1, e_2, e_3, λ_2. This information is essential for constructing the arrangement of the shadow edges: the vertices in the arrangement are not matched by their coordinates in space. They are matched instead by the symbolic definition of the four edges giving rise to their endpoints.

3.3. Shadow Edges from Face Critical Points

The reader would have noticed that we do not address degenerate inputs in this paper. For example, we do not treat the case of an input in which a critical point coincides with the extremities of a linear light source. In such a case, we simply assume, as is so often done in geometric computer graphics,[17] that a jittering argument suffices. In the case just

© The Eurographics Association 2000.

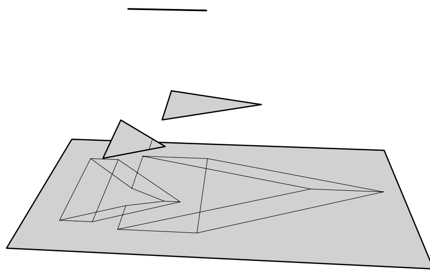

Figure 17: *The set of shadow edges resulting from illumination by the linear light source shown.*

described, for instance, we assume that shrinking the linear light source by an epsilon amount will break the degeneracy. In any case, and as Weiler points out in a recent Siggraph panel[68]: "Literally thousands of years of geometric theory and culture will not die easily, and in the meantime we have much work to do to put out current house, built from geometry and digital computer number representations, in better order."

There is one input, however, that cannot be swept under the rug by appealing to a standard perturbation argument. This input occurs when the plane carrying a polygon in the input intersects the linear light source. In this case, a viewer located on the linear light source but slightly on the positive side of that polygon will see some or all the edges defining that polygon. If the viewer is located slightly to the negative side, however, the face will be completely invisible. In Figure 18, a face critical point arises on the linear light source shown. In this case, in addition to the sets of shadow edges resulting from the two endpoints and from a viewer moving along the linear light source, shadow edges arise in the plane carrying the face. These shadow edges can be generated by computing the visibility map for a point on the linear light source and slightly to the positive side of the plane. The edges in this visibility map that cast shadow edges are *only* those edges that lie on the boundary of the polygon defining the face critical surface. These edges are generated by inserting a point light source at the location of the face critical point and restricting our attention to the edges adjacent to the face defining the critical point. It is interesting to note that even though the shadow edges previously described do not overlap (for a non–degenerate input), the shadow edges arising from face critical points *always* overlap. Indeed, this overlap is necessary to complete the labeling of the collection of shadow edges arising from a face critical point.[21] This shadow edge overlap makes it harder to compute the arrangement of shadow edges. Fortunately, help is on the way: as Mehlhorn and Näher show,[44] it is possible to construct robustly the arrangement of a set of line segments in the plane in the case of multiple edge overlap and in the case of line segments degenerating to a point (a case we are not aware may arise as a shadow edge from a non–degenerate input).

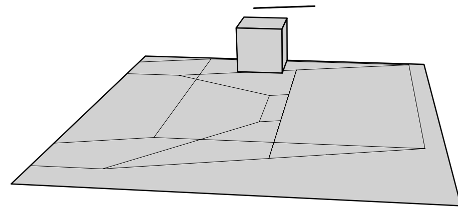

Figure 18: *One face critical point is found on the linear light source shown. The shadow edges arising from such a critical point arise in the plane carrying the face.*

3.4. Determining Critical Points

Determining the critical points along a viewpoint trajectory is an important problem in computer graphics. Given a trajectory, identifying the critical points would make it possible to compute the visibility map only once and then render it from multiple viewpoints between a pair of critical points. The rendering required would only be a two–dimensional painting of the visibility map on the output device. The case when the viewpoint trajectory is known interactively is a hard problem and only recently has there been some discussion of it in the literature.[10] Unfortunately, even if the viewpoint trajectory is known in advance, the problem is under–studied. Bern et al.[3] describe an algorithm to determine the critical points along a prespecified straight–line viewpoint trajectory. Their algorithm runs in time at least quadratic and at most cubic in the number of edges in the input scene (up to logarithmic factors).

The number of resulting shadow edges is the sum of the sizes of the three sets of shadow edges described above. The first is equal to the number of edges appearing in the two visibility maps at the two endpoints of the light source. The second is equal to the total number of vertices appearing in successive visibility maps on segments of the linear light source bounded by critical points. The third is equal to the number of fragments of the edges bounding faces that induce face critical points. The total number of shadow edges is thus directly related to the size of the different visibility maps that we compute.

4. Generating Shadow Edges from an Area Light Source

The last shadow generation problem we consider is illumination from an area light source. Standard techniques have long been known to tackle point light sources,[69, 1] but techniques that handle area light sources have emerged[30] only after the problem was separated from its closely related problem, radiosity computation.[26, 9] Despite these advances, a unifying

theory of shadow generation has been lagging. Understanding area light sources is important for understanding meshing for radiosity computation and, as in the previous sections, our objective in this section is to show that shadow computation from an area light source can be reduced to visibility problems from a polygon. Another objective is to show that just as shadow computation under linear light sources can be reduced to a collection of visibility and shadow computations under point light sources, shadow computation under an area light source can be reduced to a collection of visibility and shadow computations under point and linear light sources. Even though only the versions of shadow problems in three dimensions are interesting in computer graphics, in two dimensions as well shadow problems[31, 19] (as well as visibility problems[33]) are non–trivial and often elucidate the study of the problems in space.

We show that the illumination from a polygonal light source can be solved using the same tools and techniques for illumination by a point or a linear light source. As in the two previous sections, the computation of shadow boundaries is reduced to computing the set of different views from the area light source. We then show how a solution to this pure visibility problem is used to determine the shadow edges analytically.

An important advantage of handling shadow generation from point, linear, and area light sources seamlessly is that it is possible to insert multiple light sources of different types. We say that an API supporting such insertions is "aware" of the light sources in that the model can be updated to reflect the shadow boundaries that result from the cumulative effect of inserting a collection of light sources of different types.

4.1. Visibility from a Polygon

4.1.1. Background: Aspect Graphs

The problem of partitioning a polygon into cells such that the view of the scene from each cell is the same is related to the computation of aspect graphs. Aspect graphs are an important tool in computer vision. It is a method to categorize the different views of an object so that matching a view of that object to a category makes it possible to determine both the object seen and its orientation.

Given a three–dimensional object, computing the aspect graph asks to partition the *view space* into regions such that the view of the object seen from each region is the same. If the view space is not constrained and is all of three–dimensional space (second version of the problem discussed in Section 1.1, then the number of cells is in $O(n^9)$ for an object with n edges; although this bound is not tight for many objects. If the viewing direction is constrained to orthographic projection (first version described in Section 1.1), which is equivalent to having a viewer at infinity, then the cells partitioning the view space can be described on the surface of a sphere co–centered with the object. The number of cells in that case is in $O(n^6)$ but this bound is also not tight for many objects.

Given a polygon in the scene, our objective is to partition the polygon into cells such that the view of the scene from each cell is unique. Notice that this definition of the visibility from a polygon is distinct from both problems alluded to above. In the first, the viewer (seeing a projective view) moves in 3D whereas in the latter, the viewer (seeing an orthographic view) moves on a sphere at infinity. In the problem we discuss here, the viewer is moving on a polygon in 3D. This means that the view we are concerned with is a projective view but that the viewer has only two degrees of freedom. Thus this problem shares part of each of the two variants of aspect graphs. An example of the portion of the aspect graph as defined above is shown in Figure 19.

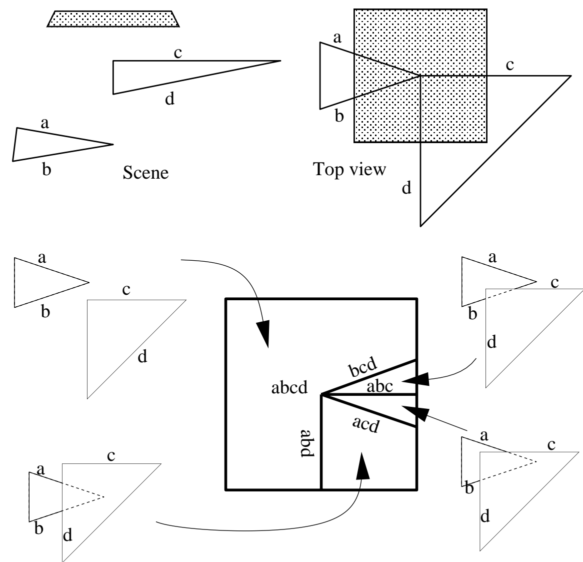

Figure 19: *The aspect graph.*

4.1.2. Incremental Computation

A viewer moving on the line segments defining the boundary of the polygon casting shadows will encounter critical points. These critical points are defined as in the case of a viewer moving along a line segment in space. In a non–degenerate input with no five edges stabbed by one line, the critical points on the boundary of the polygon are defined by the intersection of a critical edge with that boundary. This leads us to a simple heuristic: compute the location of the critical points and use the location of the critical edges thus defined to incrementally build the portion of the aspect graph inside the polygon. The approach described below is a heuristic because there are inputs on which it fails, namely, if the aspect graph consists of more than one connected component. In such cases, the approach we use of incrementally inserting critical edges will miss some visibility events.

© The Eurographics Association 2000.

We start by solving the following problem: Determine the location of the critical points as a viewer moves along the *boundary* of a polygon in space, which consists of several instances of the problem discussed in Section 3.4. Each of the resulting critical points along the boundary is defined by three edges e_1, e_2, e_3 in the scene — two of which may be adjacent. Define a *critical edge* as the intersection of the critical surface defined by e_1, e_2, e_3 with the plane of the polygon. Each critical point on the boundary of the polygon lies on a critical edge.

The problem is now reduced to a collection of problems of visibility on a line. If the critical edge is a line segment (i.e. two of the three edges are adjacent), then the critical edge is used as the viewpoint trajectory and the procedure used for the solution in Section 3.4 can be used directly. A slightly more interesting case arises for the conic critical edges (when the three edges are skew). Notice that the conic in that case is a section of a *ruled* quadric that is defined by a set of *generators* and a set of *directrices*.[54] In the context of visibility, we abuse the terminology and say that the three edges defining the critical surface are the three generators. There is, however, an infinite number of generators for the same quadric. Similarly, there is an infinite number of directrices. Given three generators, a directrix can be determined by choosing a point on one of the generators and computing the stabber to that point and the two other generators.

The discussion above points out to a simple solution to handling conics by reduction to visibility from a *straight*–line segment. The observation is that given a triple of edges, it is possible to determine a generator above the polygon (where the term "above" is used to signify "on the inside of the solid of which the area light source lies" or simply, on the negative side of the polygon defining the area light source). This generator is a straight line segment that we can use as the straight line trajectory along which to compute critical points. The generator is chosen such that it lies above the polygon. Since the visibility events in which the polygon is involved are only with the polygons on the positive side of its plane, the polygons on the negative side can safely be excluded during processing.

Given a line segment in space which is either a critical edge inside the polygon or a generator for the quadric, we determine the first critical points on the viewpoint trajectory. Two cases may arise:

1. The critical point is defined by a four–line stabber (defined by the three edges defining the critical surface and another edge in the scene). Suppose that this vertex is defined by the four edges e_1, e_2, e_3, e_4. If the critical edge is defined by e_1, e_2, e_3, we determine the edge e_4 and identify *three* instances of visibility from a line segment that need to be subsequently handled. These line segments are defined by the critical surfaces (e_1, e_2, e_4), (e_1, e_3, e_4), and (e_2, e_3, e_4) — or the other three permutations of the four edges. Four edges are thus adjacent to a four–edge stabber (this is discussed further in Section 5). Since we know that a vertex defined by a four–edge stabber must be adjacent to exactly four edges which correspond to the four permutations of edge–triples among the four edges defining the stabber, we know that there are exactly three critical edges that need to be followed.
2. The critical point is defined by three edges f_1, f_2, f_3 where $f_i \neq e_j$ for $i, j \in 1, 2, 3$. Intuitively, while an imaginary viewer is sliding on the critical line inside the area light source, a change of the visibility may arise from a visual event (disappearance of a vertex, for example) that is not related to the critical line currently being traversed. Such a critical point may be crossed twice during the traversal — once for each of the two intersecting critical lines. In this case the same critical line is considered again until a four–line stabber is found (possibly by reaching the boundary of the area light source).

Notice that one of the advantages of using dual edges[2] to represent a `PlanarMap` instance is that we can incrementally insert critical edges into the map. The structure is a legal planar map after inserting any one edge.[45]

As hinted above, the heuristic discussed above has a flaw; we assume that by proceeding in the direction of critical edges spawned from critical points we will eventually complete the structure of the portion of the aspect graph inside the polygon casting shadows. This is only true if the portion of the aspect graph is a connected component with the boundary of the polygon. This heuristic is intuitively more likely to fail when handling large area light sources. Since our main purpose from computing the portion of the aspect graph inside a polygon is to use the result to compute the shadow boundaries resulting from the illumination of the scene by an area light source, this restriction effectively prevents us from using polygonal light sources that are too large. The term "large" here is hard to qualify. It is related to the smallest features in the scene. In any case, the shadow edges resulting from the vertices and edges on the polygon's boundary are arguably more important visually than the boundaries resulting from edges inside the polygon. So if we are only interested in a scene partitioning as input to a radiosity system, then missing some of the shadow edges resulting from critical edges interior to the polygon is unlikely to be important. Anyhow, it is possible to confirm that we have not missed any visibility events by sampling on a regular grid, for example, inside the polygon. We compute the visibility at each point on the grid and confirm that all points inside the same cell share the same qualitative view. If two points inside the same cell have different views, we stop and report to the user that one of the polygons on which an area light source is too large and needs to be subdivided into two or more portions by the modeler before proceeding.

The heuristic we describe above for computing aspect graphs is reminiscent of the simplex method.[8] In our case, the feasible solution with which we start is the set of critical

© The Eurographics Association 2000.

points on the boundary of the area light source. Refining this partial solution is possible as long as the critical edges are reachable from a critical point on the boundary.

4.2. Determing Shadows Edges from an area Light Source

4.2.1. Determining Shadow Edges under a Single Occluder

Before discussing the general problem, we discuss the simpler problem of illuminating a scene with a single occluder from an area light source. The observations in this section have been made by Nishita and Nakamae.[49] The scene shown in Figure 20 consists of a triangular light source (top), a rectangular occluder (center), and a receiver (bottom). The shadow boundaries on the receiver can be determined analytically.

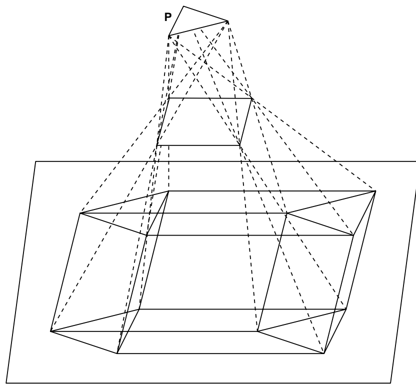

Figure 20: *The shadow edges generated by a triangular light source and a rectangular blocker. Notice that each vertex in the arrangement of the shadow edges is adjacent to exactly four edges. This holds for arbitrary occlusion under illumination by a polygonal light source.*

Each vertex on the source casts a set of shadow edges from the edges (in general, the silhouette) of the occluder. To determine these shadow edges, compute the visibility map at each *vertex* on the light source. Each *edge* in the view generates a shadow edge. In this case, each vertex of the source casts four shadow edges on the receiver, in addition to four shadow edges at infinity, which are ignored.

Also, each edge on the source casts a set of shadow edges from the vertices of the occluder. To determine these shadow edges, compute the visibility map from a viewpoint moving along each *edge* of the light source. Each *vertex* in the visibility map generates a shadow edge. Here also, each edge of the source casts four shadow edges on the receiver, in addition to four shadow edges at infinity, which are ignored.

We point out that the arrangement of the shadow edges can be computed by matching the signatures of the shadow

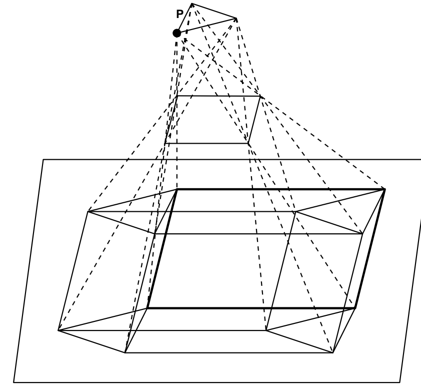

Figure 21: *The shadow edges generated by a vertex on the source.*

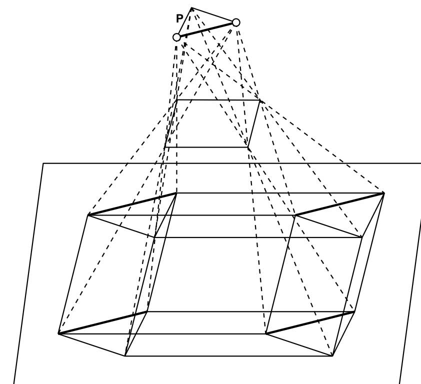

Figure 22: *The shadow edges generated by an edge on the source are shown.*

edge endpoints. Each endpoint of a shadow edge is defined by a four–edge stabber. In this scene, the number of four–edge stabbers is $3 \times 4 = 12$ as there are 3 vertices on the source and 4 vertices on the occluder. This way a pair of shadow edges are considered adjacent if the signature of their four–edge stabber is the same without relying on the accuracy of the computation of the coordinates of the endpoints.

4.2.2. Determining Shadow Edges under Multiple Occluders

Using the definition of the area light source, the critical points along its boundary, and the critical edges in its interior, we show in this section how to determine the resulting shadow edges. The signature of a shadow edge tells us whether it arises from a vertex of the lit polygon, an edge on its boundary, or a critical edge in its interior. The first two cases are identical to those discussed in Sections 2 and 3. If each resulting shadow edge is defined by $(e_1, e_2, e_3, \lambda_1, \lambda_2)$,

© The Eurographics Association 2000.

the set of shadow edges can be divided into the following three sets:

1. Shadow edges for which two of the edges (e_1, e_2, e_3) are edges belonging to the boundary of the polygon.
2. Shadow edges for which one of the edges (e_1, e_2, e_3) is an edge belonging to the boundary of the polygon.
3. Shadow edges for which none of the edges (e_1, e_2, e_3) belongs to the boundary of the polygon.

A collection of the shadow edges resulting from illumination by an area light source P is shown in Figure 23. The three sets of shadow edges can be respectively generated as follows:

1. For each vertex v on the boundary of P, generate shadow edges for each edge appearing in the visibility map for a viewer at v. This is identical to generating shadow edges for a scene illuminated by a point light source situated at v.
2. For each edge e on the boundary of P, generate shadow edges for each *vertex* appearing in the visibility map for a viewer moving along e. This is identical to generating shadow edges for a scene illuminated by a linear light source situated at e.
3. For each edge e inside P, generate a single corresponding shadow edge. If the critical edge is defined by the three edges (e_1, e_2, e_3) and is delimited by the edges $\lambda_1, \lambda_2)$, then the shadow edge is also defined by (e_1, e_2, e_3) and delimited by $\lambda_1, \lambda_2)$.

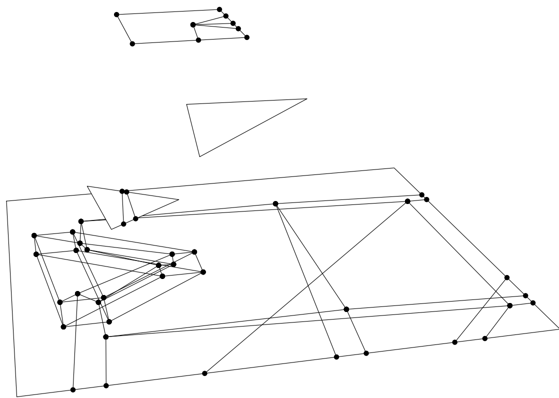

Figure 23: *The set of shadow edges in a scene illuminated by an area light source consists of three sets. To determine the first set, the visibility map at each vertex of the light source is computed and the edges of each visibility maps are projected from the vertex. The visibility map is also computed for a viewpoint moving along the boundary of the area light source between adjacent critical points. The* vertices *of these visibility maps along with the viewpoint trajectory trace the second set of shadow edges. Finally, each critical line lying inside the area light source contributes one additional shadow edge.*

5. Implementation

The visibility map and the resulting shadow generation algorithms described in this document have been implemented in C++. The data structures needed are simple and, in addition to the `PlanarMap` outlined above — which itself is implemented by no more than a collection of linked lists and heavy internal cross–referencing, only need standard structures such as search trees and priority queues. Also, since the problems we tackle here are all in object–space, we implemented a postscript driver to make it possible to visualize the shadow edges while delaying the rasterization step to the postscript device used for output.

Our implementation was in the context of a study to identify ways to maximize the internal reuse in a object–space computer graphics visibility and shadow system. We were interested in determining the set of reductions that we describe in this paper algorithmically: just as it is possible to reduce shadows from a point to visibility from a point, it is also possible to reduce shadows from a line to both visibility from a line and to shadows from a point. The class representing a scene in our system has behavior defined by the methods `insertPointLight`, `insertLinearLight`, and `insertAreaLight`. These methods in turn capture the shadows by issuing a set of lower level messages and the use of such methods provides flexibility to a system handling illumination problems. This is to be contrasted with lighting networks,[57] where the flexibility of the illumination system emanated from the ability to build systems by building networks of illumination algorithms.

The system was built in three layers. The bottom–level layer consists of classes for points, lines, segments, etc. A middle layer consists of the topology or generic structures such as linked lists, directed graphs, and the winged–edge data structure. The top level layer consists of the algorithms we describe in Sections 2, 3, and 4. Even though implementing the equivalent of our bottom–level layer is a mundane task for any graphics practitioner, we would like to add some details regarding the operations to be made available in that layer. Three operations are so pervasive in the system that we attempt here to convince the reader who might be building such a software layer to incorporate these operations. The first operation is a method which takes a point and two segments in space and returns the single line in space that is their common stabber. The fact that there is such a single stabber is a basic observation in classical geometry. Yet, it is surprising that graphics toolkits often lack such an operation. Another method, which is needed for illumination under a linear or an area light source but not under a point light source, is to determine the common stabber of four line segments in space. This problem can be solved directly in the affine space by generating a quadric surface from three segments and computing its intersection with the fourth line segment (see the appendix of either of Gigus et al.'s papers[23, 22] for details). Alternatively, Teller

© The Eurographics Association 2000.

and Hohmeyer's elegant solution[63, 64] based on Plücker coordinates may be used.

The final method that we suggest be incorporated is a method `isSilhouette`. This method is defined for edges in the scene and, given a viewpoint position, return `true` if the edge is adjacent to a polygon facing the viewer whereas its dual edge is adjacent to a polygon facing away from the viewer. Kettner and Welzl[38] study the properties of polyhedral contours that are thus defined. If our only interest were to compute the visibility map to determine shadows from a point light source, one may suspect that a heuristic that purges those edges that are not silhouette edges from consideration may be used. This would be not unlike the standard graphics heuristic of purging back–facing polygons before rendering. Unfortunately, attempts to use a heuristic based on `isSilhouette` were futile; the complete visibility map is necessary to determine (both symbolically and numerically) the extents of each shadow edge and the polygon on which it falls. Of the edges appearing in the visibility map after it is computed (the visible edges), it is still possible to reject a large number of the visible edges from candidacy to casting shadows based on the `isSilhouette` test.

To determine the set of critical points, we iterated over the faces and edges in the scene. This critical point determination step is likely to remain the bottleneck to generating shadow edges from a linear light source.

Another observation that is interesting to note follows. In the arrangement of the shadow edges under a point light source, every vertex in the arrangement must be adjacent to exactly *two* shadow edges (for non–degenerate inputs). The vertices appearing in the arrangement of shadow edges under a linear light source, however, must be adjacent to exactly *three* shadow edges the vertices appearing in the arrangement of shadow edges under an area light source must be adjacent to exactly *four* shadow edges. By vertices in these three statements we are refering to the endpoints of the shadow edges; indeed, under a linear or an area light source, two shadow edges may in general intersect at a vertex. This is easy to see if we notice that the permutations of the signature of a four line stabber defines *all* the shadow edges adjacent to it. In the case of illumination by a point light source, two of these four edges defining the stabber are fixed and there remains a choice of one out of two. In illumination under a linear light source, one of the four edges is the linear light source itself, leaving us with choosing two out of three. In illumination by an area light source, a choice of three out of four is possible since none of the four edges must arise in the definition of a shadow edge. These combinations lead to the numbers of adjacency of 2, 3, and 4 mentioned above.

Finally, we return to the point alluded to earlier. We attempt to argue that the designer of a visibility and shadow system has every advantage in forbidding the user from defining polygons in space but defining instead *solids* in space. We give the following arguments:

- The pervasive backface culling step becomes inconsistent if we define a polygon as having a single face. Consider a case where two points a and b are hidden from each other by such a single–faced polygon where a is to the front of the polygon. A viewer located at a would naturally not see b. A viewer located at b, however, may erroneously report that a is visible if the first step taken by the visibility step is to discard backfacing polygons.
- We propose that the operation of depositing shadow edges, or identifying the polygon on which the shadow edge falls, is an important operation in a shadow system. For such an operation to succeed, the *sidedness* of the polygon must be identified. This is easily done if polygons are defined as solids.

Indeed, rather than attempting to say that individual polygons (i.e. with no adjacency) should be forbidden, we suggest that polygons should only be inserted as bifaced. For example, a triangle in space would consist of three vertices, *six* edges, and *two* polygon sides. This primitive is a standard one in solid modeling where it is called a lamina,[41] as discussed in Section 1.1. Our current implementation allows for inserting only laminae and objects of cubic topology in the scene. Inserting such objects can quickly become prohibitive in the lines of code, in particular in a system that needs to construct both nodes, arcs, and faces for topology and vertices, edges, and polygons for the solids. Hierarchical methods for constructing solids[42, 29] possibly reduce the onus of using more complex solids.

6. Conclusion

We discussed in this paper that shadow problems can be systematically tackled as visibility problems and that an object–space visibility structure such as the visibility map is a versatile tool that has and will be highly exploited in a multitude of algorithms in computer graphics. The use of the visibility map for device–independent rendering is well–understood; however, this tool can also be used to replace Atherton et al.'s fundamental idea[1] in computing true shadows from point light sources and can be used in a unified manner to generate true shadows edges in illumination from a linear or an area light source. In this context, discovering a *practical* algorithm to compute the visibility map in time proportional to the size of the output (rather than proportional to the number of wire frame intersections on the image plane) is one of the most important open problems in computer graphics.

To conclude, it is interesting to point to the relative ease of generating shadows resulting from a small–sized linear or area light source. The generation of shadows from an area light source under a single occluder (which effectively results in a single view from the area light source) has long been known.[49] But, as discussed in Sections 3 and 4, the presence of critical points or critical lines inside the linear or the area light source, respectively, provides the richer setting that requires the treatment discussed in this paper.

© The Eurographics Association 2000.

References

1. P. Atherton, K. Weiler, and D. P. Greenberg. Polygon shadow generation. *Computer Graphics*, 12(3):275–281, 1978. Proc. SIGGRAPH '78.
2. B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pages 589–596, 1975.
3. M. Bern, D. Dobkin, D. Eppstein, and R. Grossman. Visibility with a moving point of view. *Algorithmica*, 11:360–378, 1994.
4. J. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics & Appl.*, 8(1):82–86, 1988.
5. K. W. Bowyer and C. R. Dyer. Aspect graphs: An introduction and survey of recent results. *Int. J. of Imaging Systems and Technology*, 2:315–328, 1990.
6. *CGAL Reference Manual*. http://www.cgal.org/.
7. N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. In *Proc. SIGGRAPH '89*, pages 99–106, New York, August 1989. ACM SIGGRAPH.
8. V. Chvátal. *Linear Programming*. W. H. Freeman, New York, NY, 1983.
9. M. F. Cohen and D. P. Greenberg. The hemicube: a radiosity solution for complex environments. *Computer Graphics*, 19(3):31–40, 1985. Proc. SIGGRAPH '85.
10. S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 78–87, 1996.
11. F. C. Crow. Shadow algorithms for computer graphics. *Computer Graphics*, 11(2):242–248, 1977.
12. M. de Berg. *Efficient algorithms for ray shooting and hidden surface removal*. Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1992.
13. F. Dévai. Quadratic bounds for hidden line elimination. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 269–275, 1986.
14. S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.
15. G. Drettakis and E. Fiume. A fast shadow algorithm for area light sources using backprojection. *Computer Graphics Proceedings, Annual Conference Series 1994*, 28:223–230, August 1994.
16. F. Durand, G. Drettakis, and C. Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 89–100. ACM SIGGRAPH, Addison-Wesley, 1997.
17. I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1–2):219–242, September 1997.
18. H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.
19. S. Ghali. Computation and maintenance of visibility and shadows in the plane. In *Sixth Int. Conf. in Central Europe on Computer Graphics and Visualization, WSCG '98*, pages 117–124, February 1998.
20. S. Ghali. *A Geometric Framework for Computer Graphics Addressing Modeling, Visibility, and Shadows*. PhD thesis, Department of Computer Science, University of Toronto, 1999.
21. S. Ghali and A. J. Stewart. A Complete Treatment of D1 Discontinuities in a Discontinuity Mesh. In *Proceedings of Graphics Interface '96*, pages 122–131, San Francisco, CA, May 1996. Morgan Kaufmann.
22. Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, June 1991.
23. Z. Gigus and J. Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(2):113–122, 1990.
24. M. T. Goodrich. A polygonal approach to hidden line elimination. In *Proc. 25th Allerton Conf. Commun. Control Comput.*, pages 849–858, 1987.
25. M. T. Goodrich. A polygonal approach to hidden-line and hidden-surface elimination. *CVGIP: Graph. Models Image Process.*, 54(1):1–12, 1992.
26. C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 213–222, July 1984.
27. L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, April 1985.
28. E. A. Haines and D. P. Greenberg. The light buffer: A shadow–testing accelerator. *IEEE Computer Graphics & Applications*, pages 6–16, September 1986.
29. P. Hanrahan. Creating volume models from edge-vertex graphs. *Computer Graphics*, 16(3):77–84, 1982. Proc. SIGGRAPH '82.
30. P. Heckbert. Discontinuity meshing for radiosity. *Third Eurographics Workshop on Rendering*, pages 203–215, May 1992.
31. P. Heckbert. Radiosity in flatland. *Eurographics*, 11(2), 1992.
32. P. S. Heckbert. *Simulating Global Illumination Using Adaptive Meshing*. PhD thesis, University of California, Berkeley, CA, January 1991.
33. P. J. Heffernan and J. S. B. Mitchell. Structured visibility profiles with applications to problems in simple polygons. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 6th Annual Symposium on Computational Geometry (SCG '90)*, pages 53–62, Berkeley, CA, June 1990. ACM Press.
34. H. Higbie. *Lighting Calculations*. John Wiley, 1934.
35. C. Hoffmann. *Geometric and Solid Modeling*. Morgan-Kaufmann, San Mateo, CA, 1989.

© The Eurographics Association 2000.

36. A. T. Campbell III and D. S. Fussell. Adaptive mesh generation for global diffuse illumination. *Computer Graphics*, 24:155–164, August 1990.

37. L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annual ACM Symp. Computational Geometry*, 1998.

38. L. Kettner and E. Welzl. Contour edge analysis for polyhedron projections. In *Geometric Modeling: Theory and Practice*. Springer–Verlag, 1997. (Proc. Int. Conf. Theory and Practice of Geometric Modeling in Blaubern, Germany, Oct. 1996).

39. D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1st edition, 1968.

40. D. Lischinski, F. Tampieri, and D. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics & Applications*, pages 25–39, November 1992.

41. M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

42. M. J. Mäntylä and R. Sulonen. GWB: A solid modeler with Euler operators. *IEEE Computer Graphics & Appl.*, 2(5):17–31, 1982.

43. M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

44. K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Report 94–160, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1994.

45. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999.

46. D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

47. K. Mulmuley. Hidden surface removal with respect to a moving point. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 512–522, 1991.

48. M. Newell, R. Newell, and T. Sancha. A new solution to the hidden surface problem. In *Proc. ACM Annual Conf.*, pages 443–448, 1972.

49. T. Nishita and E. Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 23–30, July 1985.

50. T. Nishita, I. Okamura, and E. Nakamae. Shading models for point and linear sources. *ACM Transactions on Graphics*, 4(2):124–146, April 1985.

51. M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

52. M. Pocchiola and G. Vegter. The visibility complex. *Internat. J. Comput. Geom. Appl.*, 6(3):279–308, 1996.

53. W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics*, 21(4):283–291, July 1987.

54. G. Salmon. *A treatise on the Analytical Geometry of Three Dimensions*. Longmans, Green and Co., 1912.

55. A. Schmitt. On the time and space complexity of certain exact hidden line algorithms. Report 24/81, Fakultät Inform., Univ. Karlsruhe, Karlsruhe, West Germany, 1981.

56. A. Schmitt. Time and space bounds for hidden line and hidden surface algorithms. In *Proc. Eurographics 81*, pages 43–56, Amsterdam, Netherlands, 1981. North-Holland.

57. Ph. Slusallek, M. Stamminger, W. Heidrich, J.-Ch. Popp, and H.-P. Seidel. Composite lighting simulations with lighting networks. *IEEE Computer Graphics & Applications*, 18(2):22–31, March–April 1998. ISSN 0272-1716.

58. A. J. Stewart and S. Ghali. An output sensitive algorithm for the computation of shadow boundaries. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 291–296, 1993.

59. A. J. Stewart and S. Ghali. Fast computation of shadow boundaries using spatial coherence and backprojections. *Computer Graphics Proceedings, Annual Conference Series 1994*, 28:231–238, August 1994.

60. I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17:32–42, 1974.

61. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.

62. F. Tampieri. *Discontinuity Meshing for Radiosity Image Synthesis*. PhD thesis, Cornell University, May 1993.

63. S. Teller and M. Hohmeyer. Computing the lines piercing four lines. url: http://graphics.lcs.mit.edu/~seth/pubs/-fourlines.ps.Z, 1993.

64. S. Teller and M. Hohmeyer. Determining the lines through four lines. *Journal of Graphics Tools*, 4(2):11–22, 1999.

65. S. J. Teller. Computing the antipenumbra of an area light source. *Computer Graphics*, 26(4):139–148, July 1992. Proc. SIGGRAPH '92.

66. B. R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, 1992.

67. K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics*, 11(2):214–222, 1977. Proc. SIGGRAPH '77.

68. K. Weiler, T. Duff, S. Fortune, C. Hoffmann, and T. Peters. Is robust geometry possible? In Celia Pearce, editor, *Conference Abstracts and Applications*, pages 217–219. Siggraph Panel session, Aug 1998.

69. L. Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, August 1978.

70. A. Woo, P. Poulin, and A. Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, November 1990.

© The Eurographics Association 2000.