Computing the Approximate Visibility Map, with Applications to Form Factors and Discontinuity Meshing

A. James Stewart and Tasso Karkanis
Dynamic Graphics Project
Department of Computer Science
University of Toronto

Abstract. This paper describes a robust, hardware–accelerated algorithm to compute an approximate visibility map, which describes the visible scene from a particular viewpoint. The user can control the degree of approximation, choosing more accuracy at the cost of increased execution time. The algorithm exploits item buffer hardware to coarsely determine visibility, which is later refined. The paper also describes a conceptually simple algorithm to compute a subset of the discontinuity mesh using the visibility map.

Key words:
approximate visibility, visibility map, discontinuity meshing, hardware assisted, occlusion culling, form factor, item buffer

1 Introduction

The visibility map is a data structure that describes the projection of the visible scene onto the image plane. It is a planar graph in which the vertices, edges, and faces are annotated with the corresponding vertices, edges, and faces of the scene. The visibility map provides more than just the set of visible surfaces; it provides their arrangement on the image plane. This paper describes a fast and robust algorithm to compute an approximation of the visibility map.

Previous algorithms have computed the exact visibility map. They are somewhat complicated to implement, due to the large number of "special case" input configurations that must be handled. They are also prone to robustness failure upon encountering geometric degeneraces and very small features. As most algorithm implementors will report, it is very difficult to make a geometric algorithm that is immune to this sort of failure.

This paper presents an algorithm to compute an approximate visibility map, which is approximate in the sense that some small features may be missing or distorted. The user can control the degree of approximation, choosing more accuracy at the cost of increased execution time.

The new algorithm can exploit graphics hardware to accelerate the computation, using an item buffer [1] to determine visibility at a coarse scale. The algorithm is more robust than previous object space algorithms because it operates on the discrete pixels of the item buffer, thus avoiding most of the small features and numerical uncertainty that tend to plague object space algorithms.

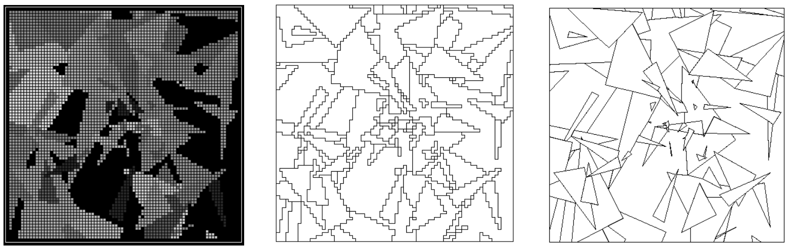
Many visibility algorithms have been developed. That of Watkins [2] was the first widely used visible surface algorithm in computer graphics, but it has been all but abandoned in favor of hardware Z–buffering which, although less elegant, is typically much faster. Various other approaches have been taken, including the "cookie cutter" approach of Atherton, Weiler, and Greenberg [3], the use of BSPs by Fuchs, Kedem, and Naylor [4], a randomized algorithm by Mulmuley [5], a plane sweep algorithm by McKenna [6], and the classic Z-buffer technique of Catmull [7].

The new approach differs in that it builds an *approximate* visibility map which contains a subset of the visible scene features. The visibility map is not discretized (as are the results of Watkins and Catmull); it is, rather, represented with floating–point coordinates on the image plane.

2 Visiblity Map Algorithm

The new algorithm to compute an approximate visibility map is conceptually simple:

1. The scene is rendered into an item buffer in which each pixel contains the unique identifier of the face visible in that pixel (Figure 1a).
2. The item buffer is traversed to build a coarse, rectilinear planar graph whose vertical and horizontal edges separate pixels containing different faces (Figure 1b).
3. The rectilinear graph is "relaxed" to produce straighter edges and a more correct positioning of its features. The topology of the graph can change to become closer to that of the exact visibility map.



(a) item buffer (b) rectilinear graph (c) relaxed graph

Fig. 1. The algorithm first renders the scene into an item buffer, then produces a coarse, rectilinear graph, and finally "relaxes" the rectilinear graph to produce a more accurate visibility map.

2.1 Step 1: Rendering into the item buffer

In the first step, each face of the scene is given a unique color and is drawn with the standard Z–buffer technique [7]. Upon completion, each pixel of the item buffer stores the color of the scene face visible through that pixel. An item buffer pixel is said to **contain** a scene face if it stores the color of that face. Each pixel also stores the depth of the corresponding face.

To distinguish pixels on a face boundary from those on the interior, the line segments bounding each face are also drawn. Each line segment of a particular face is assigned a unique bit and the logical "or" of these bits is accumulated in the alpha channel of each pixel on the face boundary. (If a face has more than eight edges, multiple passes are required.) This information is later used to help determine the correspondence between scene edges and rectilinear graph edges.

2.2 Step 2: Building a Rectilinear Graph

In the second step, a rectilinear planar graph is built. The item buffer is traversed and horizontal or vertical edges are placed between pixels that contain different faces.

The horizontal and vertical edges must be connected with vertices, which are of two types: A NORMAL vertex has exactly two adjacent edges, while a T vertex has three and is used where one edge becomes blocked by another on the image plane. The **blocked** edge corresponds to a scene edge that disappears from view. The other two **blocking** edges at the T vertex correspond to the single scene edge under which the blocked edge disappears. If fours edges meet at a vertex, they are decomposed into NORMAL and T vertices as described below.

Care must be taken to meaningfully connect the horizontal and vertical edges with the appropriate types of vertices. Figure 2a shows four pixels separated by three edges. Each pixel contains a face and is labelled with a depth, where '1' denotes the closest face and '3' denotes the most distant. The edge separating '2' and '3' bounds face 2 and becomes blocked when it passes below face 1, so the three edges must be joined by a T vertex.

Other cases are shown in Figure 2: If exactly two edges meet at a point, they are joined by a NORMAL vertex. If four edges meet at a point, they are either split into two pairs of joined edges, or two of the four edges become blocked at two T vertices. The algorithm simply iterates over all positions in the item buffer and, at each position, connects the edges according to the rule shown in the figure.

Once the graph is connected, more vertices are added which correspond to the visible scene vertices: A list is made of scene faces that appear in at least one pixel. For each such face, each of its scene vertices is projected onto the image plane. If the projection falls within a pixel containing that face then a graph vertex, which corresponds to the scene vertex, is added to the rectilinear graph.

As shown in Figure 3, vertices are projected outward from the centroid of the projected scene face until they meet a graph edge (this assumes that scene faces are convex). Each new vertex causes a graph edge to be split into two edges. More than one vertex of the face can project to the same edge, which would be split into multiple pieces. Each new vertex is labelled with the corresponding scene vertex, while the two graph edges adjacent to the vertex are labelled with the scene edges adjacent to the scene vertex. Let the notation "SCENE vertex" (as opposed to "scene vertex") denote a *graph* vertex created and labelled in this manner. There are now three types of graph vertices: NORMAL, T, and SCENE.

After all vertices have been projected, the known edge labels are propagated throughout the graph: The label on one edge can be propagated to an adjacent, unlabelled edge provided (a) they are not separated by a SCENE vertex and (b) they are not separated by a T vertex at which the edge being propagated is blocked.

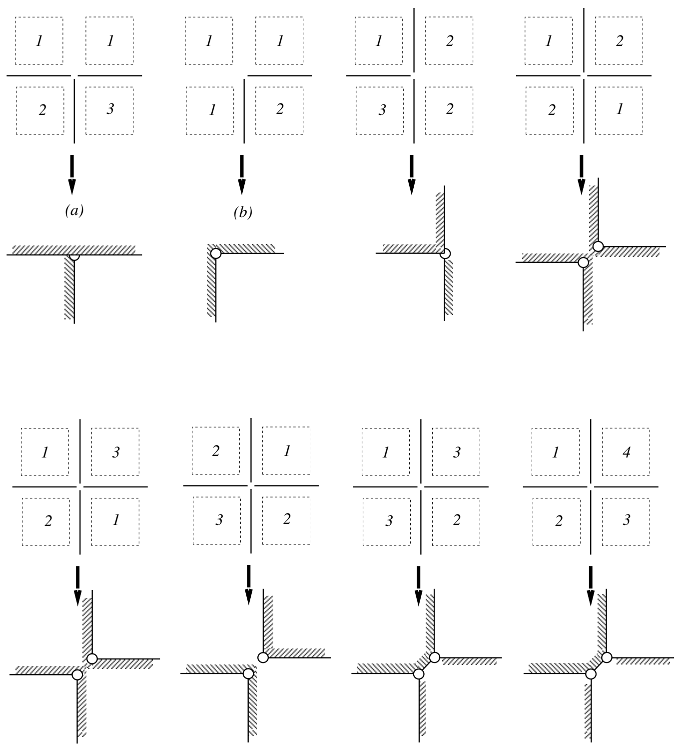


Fig. 2. The different ways in which edges are joined in the rectilinear graph. NORMAL vertices appear as open circles and T vertices appear as open semicircles, with the blocked edge on the semicircle. Symmetric cases are not shown. A depth number is repeated if and only if a particular face occupies more than one pixel. The hashing to one side of each edge indicates the closer scene face, which is bounded by that edge.

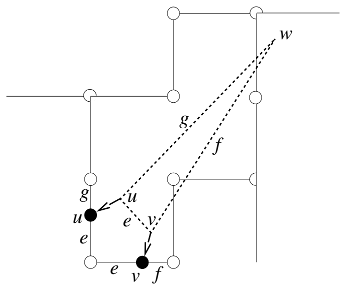


Fig. 3. The projection of a scene face, uvw, into the item buffer. Each scene vertex that appears in a pixel containing uvw is projected onto a graph edge on the pixel boundary. A new graph vertex is created and labelled with the corresponding scene vertex. The adjacent graph edges are labelled with the corresponding scene edges. SCENE vertices appear as solid circles (recall that NORMAL vertices appear as open circles).

Occasionally, a graph edge cannot be labelled in this manner. This occurs, for example, on a chain of edges whose ends are blocked by two T vertices; the SCENE vertices that would normally bound the chain are hidden. To label such a graph edge, consider that it separates two pixels, one of which contains a closer scene face. Two labelling strategies are used: If the pixel containing the closer face is identified with a unique scene edge in the item buffer, that scene edge labels the graph edge. Otherwise, each scene edge of the closer scene face is projected onto the image plane. A particular projected edge labels the graph edge if (a) it passes within one pixel of the graph edge, (b) no other projected edge passes within one pixel, and (c) it is oriented within 90 degrees of the graph edge. New labels are propagated as described above.

Upon completion of this step, the algorithm has produced a rectilinear graph that is connected and planar, and that has its faces, edges, and vertices labelled with the corresponding faces, edges, and vertices of the scene. Note that NORMAL vertices are not labelled, and T vertices are labelled with their blocked and blocking edges.

2.3 Step 3: Relaxing the Graph

The final steps consist of straightening the graph edges and moving the graph vertices to their correct locations on the image plane. A vertex of the graph is said to be **settled** if it is at its correct location (i.e. the location of its projection) on the image plane. A T vertex is correctly located at the intersection of the projections of its blocked and blocking scene edges.

Define a **multi–edge** to be a maximal–length chain of graph edges, each labelled by the same scene edge. A multi–edge represents one segment of the projection of a single scene edge. Its endpoints can be SCENE vertices or, if the scene edge disappears from view at some point, T vertices. See Figure 4.

The relaxation algorithm repeatedly cycles through the graph vertices, settling them whenever possible. Two conditions must be satisfied before a vertex may be settled:

- The interior vertices of a multi–edge cannot be settled until the endpoints have themselves been settled (otherwise, the straight line segment joining the unsettled endpoints would be undefined). This imposes a partial order on the multi–edges, in which one multi–edge is **deeper** than another if one of its endpoints is a T vertex on the other. Multi–edges must be processed in order of increasing depth.
- For a given multi–edge with settled endpoints, the straight line segment joining the endpoints in the image plane may be parameterized between 0 and 1. Each interior T vertex of the multi–edge corresponds to some parameter, which is that of the image plane intersection of the straight line segment and the projection of the T vertex's blocked scene edge. Interior T vertices cannot be settled until their parameters are **consistently ordered**: Each parameter must be in the range $[0, 1]$ and parameters must increase monotonically as the multiedge is traversed from the tail endpoint (with parameter 0) to the head endpoint (with parameter 1).

Relaxation. To relax the graph, the algorithm iterates through the following steps until nothing remains to be settled. Each step is explained below.

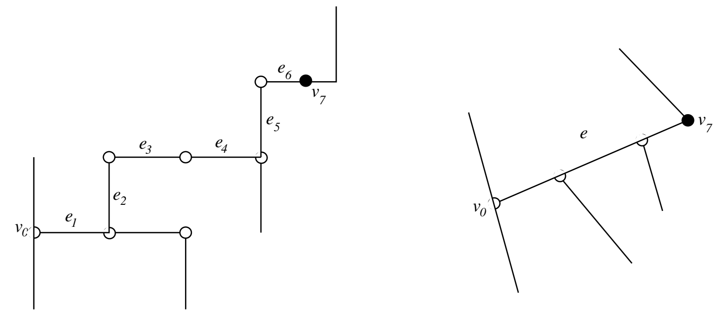


Fig. 4. *Left:* A multi–edge consisting of graph edges $e_1, e_2, \ldots e_6$, bounded by graph vertices v_0 (T) and v_7 (SCENE). Each e_i is labelled with the same scene edge. *Right:* The relaxed multiedge, e. Only after v_0 and v_7 are settled may the multi–edge be straightened and its five interior vertices be settled.

1. For each multiedge with settled endpoints but unsettled interior vertices, attempt to order the interior vertices consistently by their parameters. Consistently ordered vertices are said to be **viable**.
2. For each unsettled graph vertex, v, attempt to settle it. If v is a T vertex, it must be viable before being settled.

It is possible that the algorithm terminates with remaining **unsettlable** vertices, which cannot move to their correct positions without violating the planarity of the graph. At that point, these vertices are frozen in their (incorrect) positions. The quality of an approximate visibility map can be measured by its percentage of unsettlable vertices.

Settling vertices. To settle a vertex, the vertex must be moved to its correct location in the image plane. When a vertex moves, its attached edges are pulled along with it.

In attempting to settle a vertex, the algorithm first determines whether the moving vertex intersects any graph edge on the way to its correct location. The algorithm also determines whether any of the attached edges intersects a graph vertex as it is being pulled along. These tests need only to be performed in at most three graph faces immediately adjacent to the moving vertex.

If no intersection occurs, the vertex is moved to its correct location and is considered to be settled. If an intersection occurs, the movement would cause the graph to become non–planar, so the attempt fails and the vertex remains at its current position. The algorithm thus guarantees that the graph remains planar.

Consistent ordering on multi–edges. Interior vertices of a multi–edge are consistently ordered if their parameters increase monotonically from tail to head, and if all parameters lie in the range $[0, 1]$.

In attempting to settle a multi–edge, the algorithm searches for inconsistencies in the order of interior vertices. Where inconsistences are found, the topological structure of the graph is modified to eliminate them.

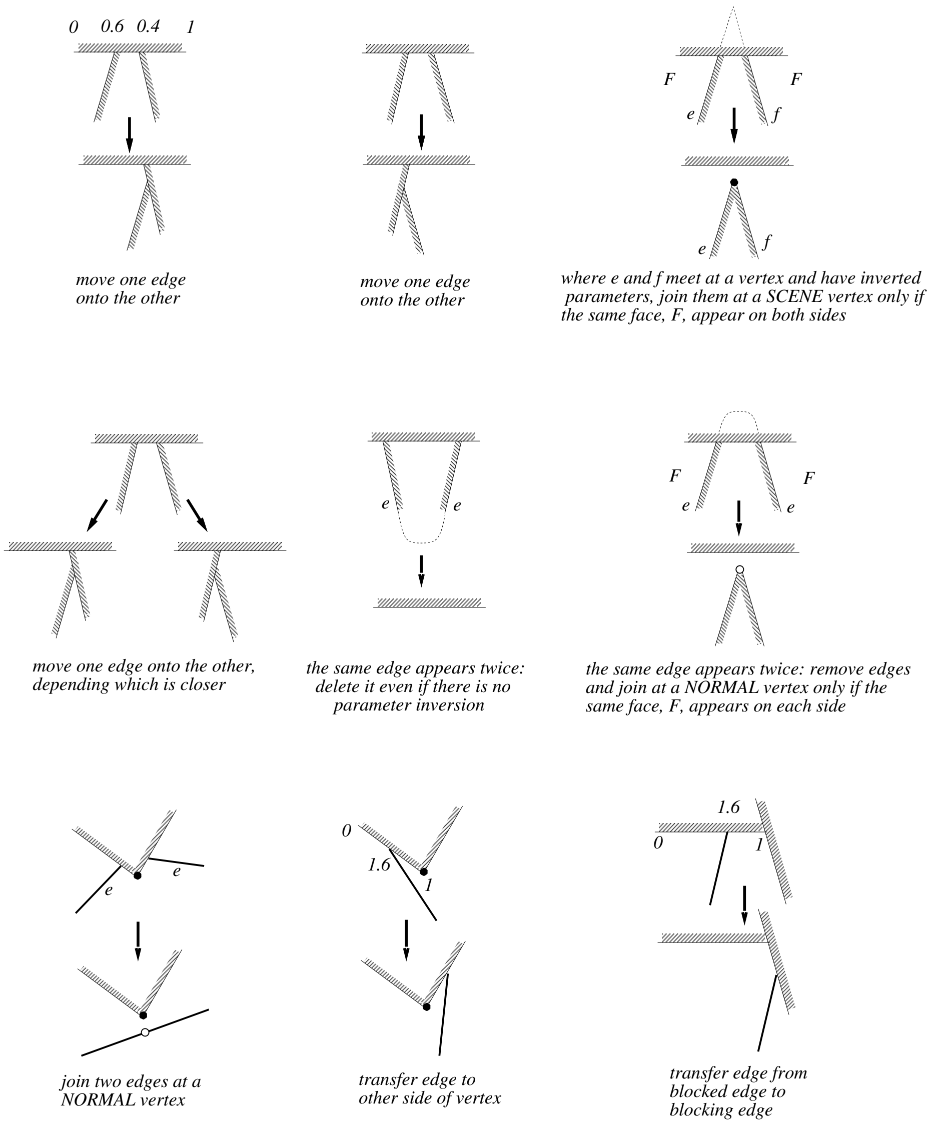


Fig. 5. Different actions are taken to resolve inconsistencies on a multi–edge. The diagrams show the adjacent scene faces with hash marks and sometimes show the parameter values at the T vertices. In all cases, there is either a parameter inversion (first and second rows) or a parameter is outside the range $[0, 1]$ (third row)

Figure 5 shows the different ways in which the graph may be changed. For example, the upper left diagram of Figure 5 shows two T vertices with inconsistently ordered parameters, 0.6 and 0.4. In this particular situation, each graph edge below one of the T vertices has a corresponding scene edge that bounds its scene face to the right, as shown by the hash marks on the right side of each edge. To resolve the inconsistency, the algorithm transfers the left edge to the interior of the right edge, since the scene face adjacent to the right edge is closer to the viewpoint than is the scene face adjacent to the left edge.

The algorithm matches each inconsistency with one of the cases shown in Figure 5 and resolves it as indicated in the Figure. Note that it is possible that not all inconsistencies are resolved. For example, a T vertex may have parameter 1.6 (outside the range $[0, 1]$), but be unable to move off the multi–edge because the head vertex (having parameter 1) is too convex. This is like the diagram in the center of the bottom row of Figure 5, but with a much sharper vertex.

2.4 Experimental Results

The algorithm has been implemented in C++. Figure 6 shows the visibility map generated for a set of 10,000 random, non–intersecting triangles. The item buffer step has eliminated all but 1453 of the triangles; this substantially reduces the required work of the subsequent rectilinear and relaxation steps. Figure 8 shows the results using different item buffer resolutions on a room scene that has been subdivided into 784 abutting polygons. The number of visually apparent defects and the percentage of unsettlable vertices generally decrease with increasing item buffer resolution. Execution times and the percentage of unsettlable vertices are shown in Table 1. We were surprised to discover that the software rendering performs better than the hardware rendering. Perhaps there's little opportunity to exploit the hardware because each face is rendered individually and there are frequent changes in the rendering mode (e.g. with the stencil buffer used to label edge pixels).

Table 1. Execution times in seconds for a 300 MHz Pentium II (software rendering) and an SGI 250 MHz High Impact (hardware rendering). The total time is divided into that used to fill the item buffer and that used to build the visibility map. A lower percentage of unsettlable vertices corresponds to a more accurate visibility map.

		unsettlable	Pentium times			SGI times		
scene	resolution	vertices	item	map	total	item	map	total
10,000 tri	400×400	8%	3.57	6.92	10.49	6.06	11.91	17.97
room	800×400	14%	1.53	3.94	5.47	1.12	7.35	8.47
room	400×200	18%	0.60	1.96	2.56	0.63	2.85	3.48
room	200×100	16%	0.34	1.01	1.35	0.49	1.21	1.70
room	100×75	25%	0.26	0.53	0.79	0.46	0.66	1.12

3 Application to Form Factors

The point–to–patch form factor describes the fraction of energy leaving a patch that arrives at a point, and is computed by integrating over the visible area of the patch. Determining visibility is the most expensive operation in computing a form factor. The hemicube method [8] is typically used to determine form factors, but can suffer aliasing problems because visibility is sampled on a

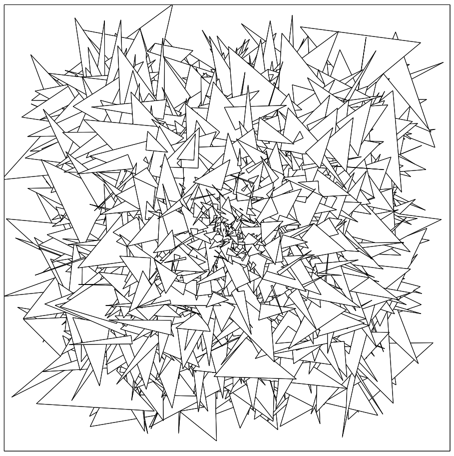


Fig. 6. The approximate visibility map of a scene of 10,000 randomly positioned triangles, using a 400×400 item buffer. Only 1453 triangles appear in the item buffer, which substantially reduces the subsequent processing time.

relatively coarse, regular grid. Other algorithms use ray tracing to sample the visibility [9, 10].

The approximate visibility map would seem ideally suited for form factor computation. After rendering into the item buffer, the relaxation step should produce a more accurate visibility map and better form factors.

However, approximate visibility performs poorly in computing form factors. It takes substantially longer and, in some cases, actually has a larger RMS normalized error than the hemicube method, as shown in Table 2. The large RMS error occurs in the approximate visibility map when small faces are "pulled larger" in order to maintain topological constraints.

Table 2. Comparison of the hemicube and the approximate visibility map in computing form factors, with hardware rendering.

resolution	method	mean error	RMS error	time (sec)
100×100	hemicube	0.011703	0.148934	0.08
100×100	visibility	0.017119	0.494695	1.71
400×400	hemicube	-0.000557	0.026423	0.50
400×400	visibility	-0.006068	0.037054	4.50

4 Application to Discontinuity Meshing

Discontinuity meshing is used to increase the accuracy and speed of radiosity algorithms. The work of Campbell and Fussell [11] and Heckbert [12] in this area has spurred a number of meshing algorithms, including those of Lischinski, Tampieri, and Greenberg [13, 14], Drettakis and Fiume [15], and others. A more general structure, the Visibility Skeleton [16], may also be used to compute the discontinuity mesh.

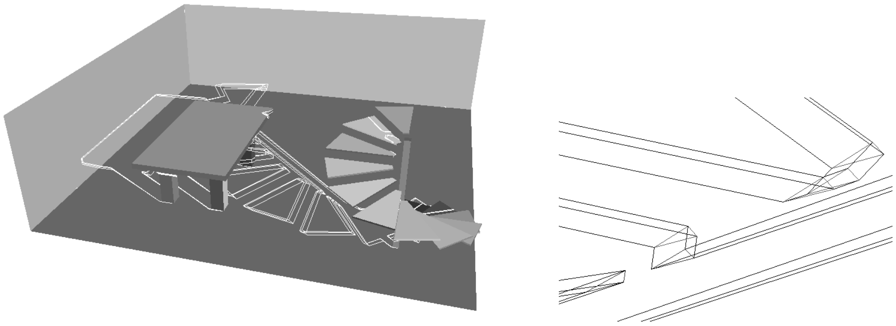


Fig. 7. *Left:* Discontinuity mesh. *Right:* Closer view of some segments.

Discontinuity meshing has received little attention recently, in part because of the unavailability and sheer complexity of discontinuity meshing programs. These programs must treat many special, complex geometric configurations and are prone to robustness failure in the presence of degenerate configurations or numerical uncertainty.

This section describes a very simple discontinuity meshing algorithm that computes *only a subset* of the segments of the discontinuity mesh: in particular, it determines those segments defined by a source vertex and scene edge (called EV_s) and some of the segments defined by a source edge and two scene edges (called EEE_s). It does not determine any segments that do not involve a source vertex or edge.

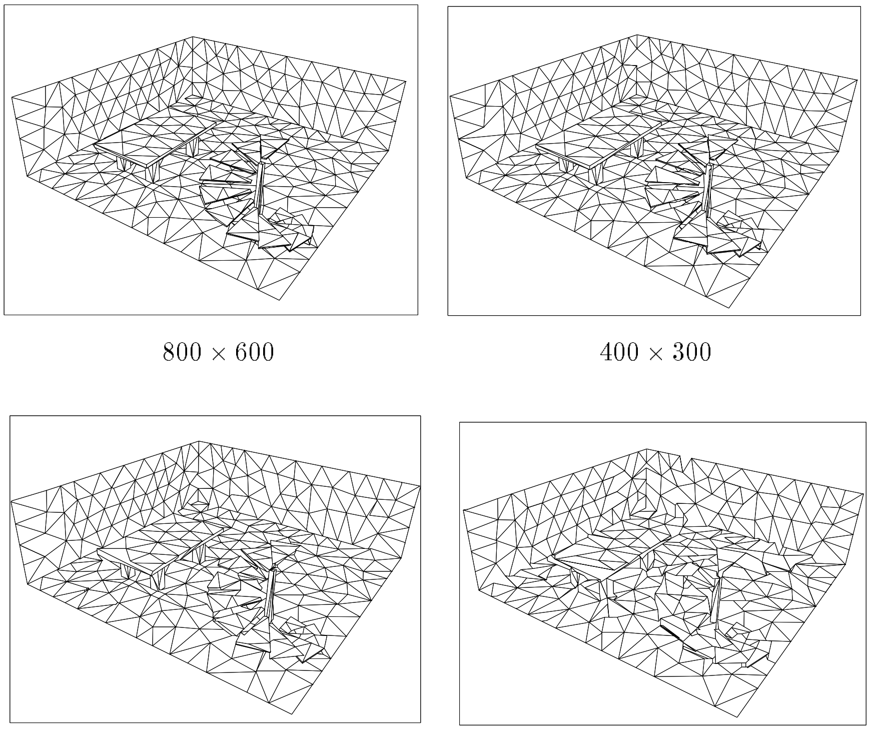
Two observations from the literature are exploited:

- The EV_s segments can be determined by computing the visibility map at each source vertex and projecting the edges of the map into the scene [12].
- The endpoints of the discontinuity segments can be tagged by the scene features that define them. Endpoints with matching tags can be joined with a discontinuity segment [17, 16].

The algorithm does exactly that. It computes the approximate visibility map from each source vertex and projects the edges of the map onto the faces of the scene to form EV_s segments. Each segment endpoint is tagged with the source vertex and two scene edges that define it. Where there are two vertices tagged by adjacent source vertices and identical scene edges, an EEE_s segment is identified. Each such segment is projected onto the appropriate faces with a two–dimensional visibility calculation [12, 15].

Figure 7 shows the computed mesh for a simple scene of 104 polygons and a triangular light source. Using software rendering on a 300 MHz Pentium, the mesh was computed in 9.5 seconds.

This simple meshing algorithm has some attractive properties: It avoids the slow, error–prone geometric computations of other meshing algorithms; it is very easy to implement, given the approximate visibility code; and the item buffer step very efficiently eliminates most occluded faces, which don't contribute to the mesh. Some potential problems exist with the algorithm: If the approximate visibility map does not contain an edge visible from the source, then the corresponding EV_s segment will not appear in the mesh. Using a higher–resolution



800 × 600   400 × 300

200 × 150   100 × 75

Fig. 8. Various item buffer resolutions are used for a 784–polygon scene.

item buffer helps to avoid this. Spurious segments can occur in the mesh if the approximate visibility map contains incorrect T vertices. This is avoided by ignoring unsettlable T vertices.

5 Summary

This paper has introduced an algorithm to compute the approximate visibility map. The user can control the accuracy of the approximate visibility map by choosing a higher resolution item buffer, at the cost of increased execution time.

There are two principal advantages to using the item buffer: It culls many of the small or hidden objects from consideration, and it provides the algorithm with a well defined graph as an initial approximation to the visibility map. Planarity of the final visibility map is guaranteed because no planarity–violating alterations are permitted during the relaxation phase.

The approximate visibility map is not useful for form factors, but it is useful for a limited form of discontinuity meshing. The authors hope that other applications, such as occlusion culling, may also find a use for approximate visiblity.

6 Acknowledgements

The authors thank Nasser Keshmirshekan for the creating the subdivided scenes, and thank the reviewers for their thoughtful and thorough reviews.

References

1. H. Weghorst, G. Cooper, and D. Greenberg, "Improved computational methods for ray tracing", *ACM Transactions on Graphics*, vol. 3, no. 1, pp. 52–69, Jan. 1984.
2. G. S. Watkins, "A real-time visible surface algorithm", Report UTEC-CS-70-101, Dept. Comput. Sci., Univ. Utah, Salt Lake City, UT, 1970.
3. P. Atherton, K. Weiler, and D. Greenberg, "Polygon shadow generation", in *Computer Graphics (SIGGRAPH)*, 1978, pp. 275–281.
4. H. Fuchs, Z. M. Kedem, and B. Naylor, "On visible surface generation by a priori tree structures", *Comput. Graph.*, vol. 14, no. 3, pp. 124–133, 1980.
5. K. Mulmuley, "An efficient algorithm for hidden surface removal", *Comput. Graph.*, vol. 23, no. 3, pp. 379–388, 1989.
6. M. McKenna, "Worst-case optimal hidden-surface removal", *ACM Trans. Graph.*, vol. 6, pp. 19–28, 1987.
7. E. Catmull, *A subdivision algorithm for computer display of curved surfaces*, Ph.d. thesis, Computer Science Department, University of Utah, 1974, Report UTEC-CSc-74-133.
8. M. F. Cohen and D. P. Greenberg, "The Hemi-Cube: A radiosity solution for complex environments", in *Computer Graphics (SIGGRAPH '85 Proceedings)*, B. A. Barsky, Ed., August 1985, vol. 19, pp. 31–40.
9. J. R. Wallace, K. A. Elmquist, and E. A. Haines, "A ray tracing algorithm for progressive radiosity", in *Computer Graphics (SIGGRAPH '89 Proceedings)*, Jeffrey Lane, Ed., July 1989, vol. 23, pp. 315–324.
10. P. Schroder, "Numerical integration for radiosity in the presence of singularities", in *Proceedings of the Fifth Eurographics Workshop on Rendering*, 1993, pp. 177–184.
11. A. T. Campbell and D. S. Fussell, "Adaptive mesh generation for global diffuse illumination", in *Computer Graphics (SIGGRAPH)*, 1990, pp. 155–164.
12. P. Heckbert, *Simulating Global Illumination Using Adaptive Meshing*, Ph.D. thesis, CS Division (EECS), Univ. of California, Berkeley, June 1991.
13. D. Lischinski, F. Tampieri, and D. P. Greenberg, "Discontinuity meshing for accurate radiosity", *IEEE Computer Graphics Graphics & Applications*, pp. 25–39, November 1992.
14. D. Lischinski, F. Tampieri, and D. P. Greenberg, "Combining hierarchical radiosity and discontinuity meshing", in *Computer Graphics (SIGGRAPH '93 Proceedings)*, 1993, pp. 199–208.
15. G. Drettakis and E. Fiume, "A fast shadow algorithm for area light sources using backprojection", in *Computer Graphics (SIGGRAPH)*, 1994, pp. 199–208.
16. F. Durand, G. Drettakis, and C. Puech, "The Visibility Skeleton: A powerful and efficient multi–purpose global visibility tool", in *Computer Graphics (SIGGRAPH)*, 1997, pp. 89–100.
17. G. Drettakis, *Structured sampling and reconstruction of illumination for image synthesis*, Ph.D. thesis, Computer Science Department, University of Toronto, January 1994.