Computing Visibility from Folded Surfaces

A. James Stewart

Dynamic Graphics Project,
Department of Computer Science,
University of Toronto

Abstract

This paper presents an algorithm to estimate what parts of the surrounding environment are visible from each of many sample points on a surface, which may be highly folded and may contain holes. The results can be used to estimate how much of an extended light source is visible from each sample point and, thus, the shading of the surface. Examples are presented of shaded cloth. Unlike shading methods that use a local illumination model, the new method accurately estimates the darkness of deep folds in the cloth.

Key words: visibility, shading, cloth, rendering, illumination

1 Introduction

The goal of this work is to shade surfaces such as cloth which can contain holes and complex folds. The surface is assumed to be represented with a polygonal mesh, and shading will be computed at each mesh vertex. The **shading** of a point is the radiance leaving the point in the direction of the viewer.

Shading a mesh point, $\mathbf{x}$, requires that the incoming radiance be known for all directions above the surface at $\mathbf{x}$. These directions are typically represented by a hemisphere, $H(\mathbf{x})$, centered about $\mathbf{x}$'s surface normal. The radiance in the direction of the viewer is given by the rendering equation [1], which integrates incoming radiance over all incoming directions in $H(\mathbf{x})$.

The hemisphere of directions may be divided into two parts: those directions, $V(\mathbf{x})$, in which the outside environment is visible, and those directions, $H(x)\backslash V(x)$, in which another part of the surface is visible. For diffuse Lambertian surfaces of moderate albedo, the largest component of the shading comes from the outside environment, $V(\mathbf{x})$.

Algorithm Output and Applications

This paper describes an algorithm whose output is an approximation of $V(\mathbf{x})$ at each mesh point, $\mathbf{x}$. This approximation is called the **visibility cone at $\mathbf{x}$** and is represented by a polygonal cone whose vertex is $\mathbf{x}$.

Given $V(\mathbf{x})$, the shading of $\mathbf{x}$ can be computed with a local illumination model [2] that estimates the radiance arriving from directions in $H(x)\backslash V(x)$. This local illumination model has been shown by experiment to be fairly accurate for diffuse surfaces of low, constant albedo. After the vertex shading has been computed, OpenGL can be used to interpolate the shading of the surface between the vertices. Examples are shown in Section 4.

Other applications of the visibility cones include occlusion culling and view–dependent mesh simplification: Those vertices of the mesh that do not contain the current viewpoint in their visibility cone are necessarily occluded, and the mesh in the vicinity of the occluded vertices may be culled or simplified. The visibility cones might also be used for importance sampling in Monte Carlo ray tracing: If the surface is of low albedo, a greater contribution to the shading is expected from those directions inside the visibility cone.

Some Simple Approaches

It is possible to compute visibility cones with a simple but very slow algorithm. For example, one could project all mesh faces onto a small sphere surrounding each mesh point, $\mathbf{x}$. Then the visibility cone would correspond to the part of the sphere that is not covered by any mesh face. This approach would take time $O(n^3)$ in a mesh of size n. A plane–sweep technique [3] could be used to reduce this to $O(n^2 \log n)$, but plane–sweep is notoriously prone to robustness failure in the presence of small mesh features. Another approach might use ray shooting to sample visibility in various directions around $\mathbf{x}$. However, a straightforward implementation could take time $O(n^2)$, since each ray must be intersected with each face of the mesh. Hierarchical bounding boxes or an octtree could accelerate the ray shooting at the cost of a somewhat more complex algorithm. Section 5 contains a more detailed discussion.

The algorithm described in this paper uses a different approach, one in which the problem is reduced to several two dimensional visibility computations. The 2D visibility problem has been thoroughly studied in the computational geometry literature, and simple and efficient algorithms exist that solve it.

2 The Visibility Algorithm

A **planar slice** of a mesh is the intersection of the mesh with a plane. The intersection consists of vertices (which correspond to mesh edges) and edges (which correspond to mesh faces). If the mesh does not self–intersect the vertices and edges form a set of disjoint polygonal chains in the plane. The visibility algorithm described in this paper requires that the mesh not self–intersect.

In the remainder of the paper, a "vertex" will generally refer to a vertex in the planar slice. Vertices of the 3D surface mesh will be called "mesh vertices" to distinguish them.

The visibility algorithm takes many planar slices of the mesh, for various positions and orientations of the slicing plane. For each vertex, $\mathbf{v}$, in a slice, the **2D visibility cone** at $\mathbf{v}$ represents the set of directions in the plane in which the external environment is visible from $\mathbf{v}$. The visibility algorithm computes the 2D visibility cone of each vertex in the slice, and stores each visibility cone with the closest mesh vertex. After many slices have been taken, each mesh vertex stores many 2D cones at various orientations. These 2D cones are combined to form 3D cones.

2.1 Slicing the Surface

The set of slices chosen by the algorithm is dependent upon the light source that illuminates the surface. Here we consider the cases of uniform diffuse illumination (as on a cloudy day), point source illumination, and area source illumination.

- Under uniform diffuse illumination, light arrives from all directions and the set of slices should span many orientations. In the experiments with diffuse illumination presented in Section 4, six planes are used, with normal directions (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), and (0,1,1). In a particular orientation, adjacent slices are separated by the average size of one mesh quad.
- For a point light source, light arrives only from one point in space and each slicing planes should embed that point, since the 2D visibility cones are computed in the slice. Any line through the source may be chosen about which to rotate the slicing plane.
- For an polygonal area light source, slicing planes are rotated though lines that join pairs of opposite points on the convex hull of the area source.

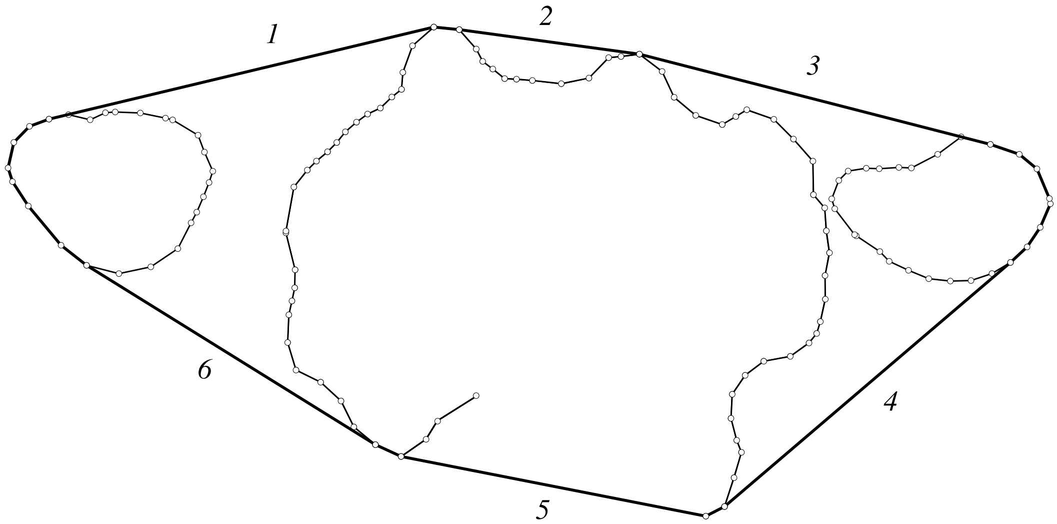

Fig. 1. A horizontal slice of a shirt mesh is shown. There are three separate polygonal chains corresponding to the body and two arms. These chains, along with the convex hull (shown with thicker lines) define six visibility polygons inside the hull. Each visibility polygon bounds a connected region of the plane. There are six window edges, labelled 1 through 6.

2.2 Computing Visibility Polygons

Each planar slice of the surface mesh contains a set of polygonal chains. Consider the convex hull of these polygonal chains. Define a **connected region** to be a maximal region of the plane such that, for all pairs of points in that region, there is a path between them that does not cross the convex hull or a polygonal chain. The exterior of the convex hull is one such region; there may be many such regions on the interior of the hull.

A **visibility polygon** is the boundary of a connected region, with the edges oriented to have the region interior to their left. Note that a visibility polygon may contains holes and may contain **dangling edge chains** which have the region interior on both sides. Edges on such chains appears twice on the visibility polygon: once in each direction. See Figure 1.

Of the edges of a connected region, there are two types: edges that are part of one of the polygonal chains, and edges that are not. An edge that is not part of a polygonal chain is necessarily on the convex hull, and is called a **window**. Intuitively, it is through these windows that the exterior environment is visible. Note that a visibility polygon may contain zero, one, or more windows.

After slicing the surface, the next step of the algorithm is to build the convex hull and to determine the set of visibility polygons.

The principal difficulty of this step lies in reliably building the convex hull. While many theoretically correct hull algorithms exist [4–7], they rarely succeed when the points are very close. This situation occurs frequently in the planar slices, particularly when the slicing plane intersects two mesh edges of the same mesh face very close to their common mesh vertex. Of several algorithms evaluated, the Qhull algorithm [7] was found to be most reliable. Another way to reduce the likelihood of robustness failure is to delete enough points so that the minimum distance between remaining points is above some threshold.

Given the convex hull, all window edges are identified and each is used as the starting point for a counterclockwise traversal of a visibility polygon. After this, any edge that has not been traversed necessarily belongs to a hole. These holes are identified and placed in the interior of the appropriate visibility polygons.

2.3 Computing the 2D Visibility Cones

In a given visibility polygon, the external environment can be seen only through a window of that polygon. Since windows lie on the convex hull, no point in one visibility polygon can see any point in another, and the visibility polygons may be treated independently.

This section describes the computation of 2D visibility cones at all vertices of a visibility polygon that contains a *single window*. Visibility polygons with multiple windows are processed once for each window; vertices in those polygons can have visibility cones that consists of several disjoint pieces, each piece directed toward a different window.

Note that a point that lies on the boundary between two visibility polygons will have two visibility cones, one in each of the adjacent polygons. These are called the **front–facing** and **back–facing** visibility cones at that point, the name depending upon the orientation of the surface at that point. Both cones are produced automatically by the algorithm. See Figure 2.

The problem of finding 2D visibility cones is well–studied in the computational geometry literature [8,9], and relies upon the idea of a "shortest path." Given two points, s and t, within a polygon the **shortest path**, $\pi(s,t)$, between s and t is the Euclidean shortest path that connects s and t while remaining in the polygon interior. The vertices of a shortest path are vertices of the polygon (since any path may be made shorter by "tightening" it until it lies against the polygon vertices). Thus $\pi(s,t) = v_1, v_2, \ldots, v_n$, where $v_1 = s$, $v_n = t$, and the remaining v_i are polygon vertices.

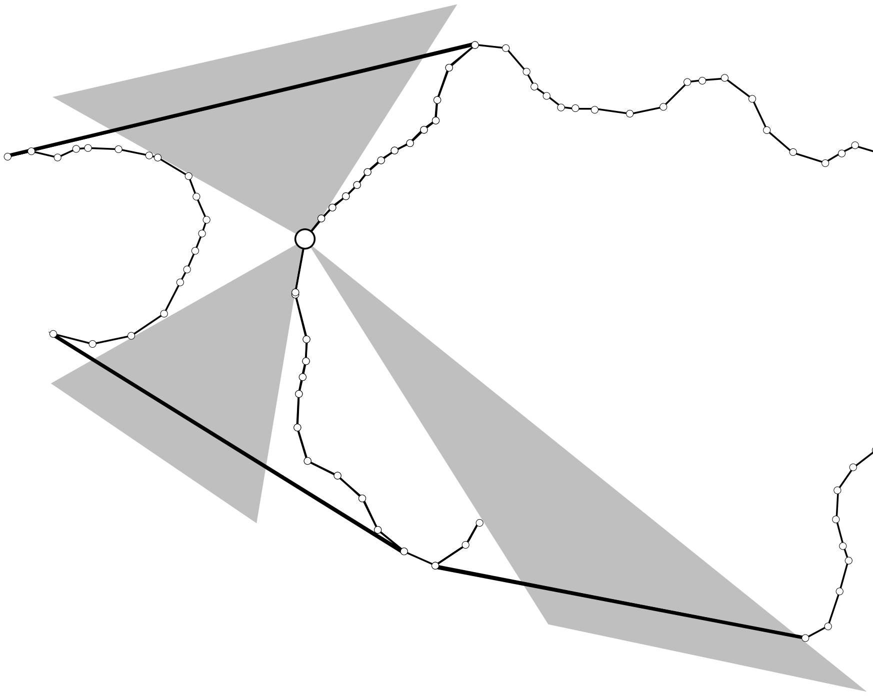

Fig. 2. A point on the boundary between two visibility polygons is shown. That point has a 2D visibility cone on each side of the boundary. On the left side of the boundary, the cone consists of two disjoint pieces.

Let v be a vertex of the visibility polygon and let w_ℓ and w_r be vertices of the window edge. Given the shortest paths $\pi(v, w_\ell)$ and $\pi(v, w_r)$, the 2D visibility cone at v is easily determined: It is bounded by the first segment on $\pi(v, w_\ell)$ and the first segment on $\pi(v, w_r)$. See Figure 3. Note that the visibility cone is empty if these two segments are collinear.

2.3.1 Computing Shortest Path Trees

An algorithm of Guibas *et al* [9] efficiently computes all shortest paths from a single "source" vertex to all other vertices within a simple polygon (polygons with holes are discussed later). This section describes that algorithm. To determine all 2D visibility cones in a visibility polygon, this algorithm is simply applied to each vertex of the window edge and the cones are immediately recovered.

The Guibas *et al* algorithm computes a "shortest path tree" rooted at the source vertex. The edges of the tree are segments joining polygon vertices, and each root–to–leaf path is the shortest path from the source vertex to one of the other polygon vertices. See Figure 4.

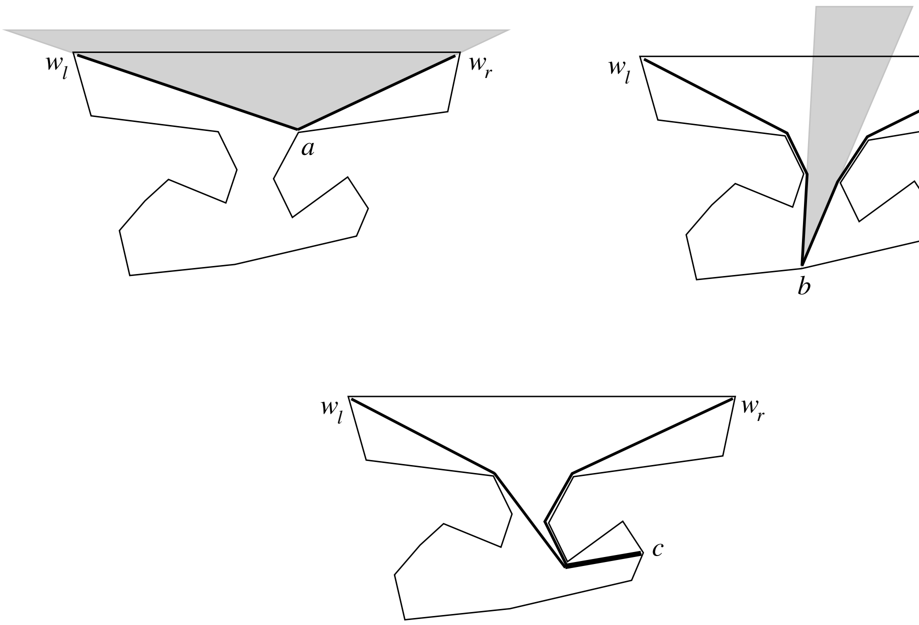

Fig. 3. The shortest paths from each end of the window to points a, b, and c are shown. Point a sees the whole window, point b sees only part, and point c sees none of the window.

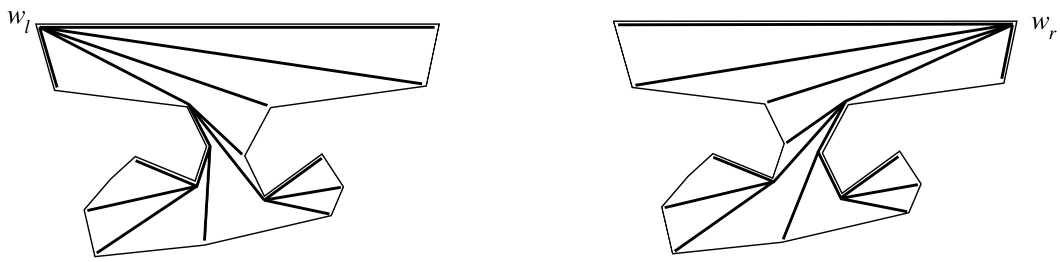

Fig. 4. The shortest path trees rooted at each endpoint of the window are shown.

First, the polygon is triangulated[1]. Each edge of this triangulation is either an original polygon edge or is a diagonal edge that joins two polygon vertices and crosses the interior of the polygon.

As with the convex hull computation, many theoretically correct triangulation algorithms exist [10–12], but few actually succeed in the presence of very small features. Of the publically–available implementations that were evaluated when implementing the cloth shading algorithm, that of Held [13] was capable of triangulating polygons with holes and dangling edge chains, and performed without fault in the presence of very small features.

Let s be the source vertex. Consider a diagonal edge, (u, v), of the triangu-

[1] Visibility polygons with multiple windows need only be triangulated once, since the same triagulation may be used in building each shortest path tree.

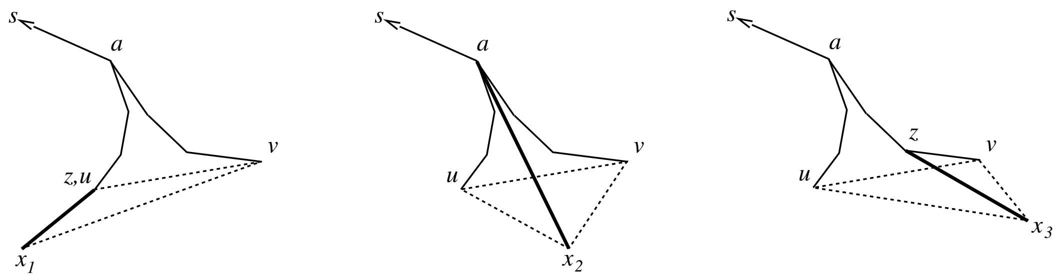

Fig. 5. The funnel, $F(u,v)$, with cusp a is shown. The shortest path, $\pi(s,x)$, to the triangle vertex, x, on the side of (u,v) opposite the funnel (i) may touch either u or v, (ii) may go directly from a, or (iii) may touch a point z inside the funnel at which the segment (z,x) is tangent.

lation. In the two shortest paths, $\pi(s,u)$ and $\pi(s,v)$, vertices u and v have a least common ancestor, a, at which the shortest paths from u and v first meet. The two shortest paths, $\pi(a,u)$ and $\pi(a,v)$, form a **funnel**, $F(u,v)$, which is convex inward. Vertex a is called the **cusp** of the funnel. See Figure 5.

Let (u,v,x) be the unique triangle that lies to the side of (u,v) opposite the funnel. Then the shortest path from s to x starts with the path $\pi(s,a)$ and is followed either (i) by the segment (a,x) if this segment does not intersect the funnel boundary, or (ii) by part of one side of the funnel, $\pi(a,z)$, followed by the segment (z,x), where z is the unique vertex of the funnel at which the segment (z,x) is tangent.

Thus, given a funnel, $F(u,v)$, and the triangle, (u,v,x), opposite the funnel, the shortest path $\pi(s,x)$ is quickly computed [2]. The path $\pi(a,x)$ divides $F(u,v)$ into two funnels, $F(u,x)$ and $F(x,v)$. (Note that one of these funnels will be empty if the tangent vertex, z, is identical to one of u or v.) If the edge (u,x) or the edge (x,v) is a diagonal in the triangulation, the same procedure can be applied recursively to that diagonal.

The algorithm of Guibas *et al* starts with a empty funnel, $F(s,t)$, (where t is a vertex adjacent to s) and applies the above procedure, continuing recursively until the shortest path tree can be extended no farther.

[2] Guibas *et al* use a data structure, the *finger tree*, to achieve an asymptotically efficient running time. The author of this paper found that finger trees were not necessary in the small polygons encountered in practice. It was sufficient to store at each vertex a pointer to its parent vertex in the shortest path tree.

2.3.2 Treating Holes in the Visibility Polygon

The algorithm of Guibas *et al* applies only to simple polygons that do not have holes. Where holes exist in a visibility polygon, they are ignored during the triangulation and shortest path computation, but must later be considered.

First, 2D visibility cones must be computed for each vertex of the hole. This is done in a brute–force manner: Let the 2D visibility cone of a hole vertex, v, initially be bounded by the directions toward the window endpoints. Consider each other vertex, u, of the visibility polygon that lies within the 2D visibility cone. If the two edges adjacent to u fall to the same side of the uv line, replace one bounding directions of the 2D visibility cone with the direction from v to u.

Second, the 2D visibility cones computed with the algorithm of Guibas *et al* must be corrected: The hole must be subtracted each such cone. This can be done efficiently: Take one vertex on the boundary of the visibility polygon and find the two tangents from that vertex to the boundary of the hole. The region between these two tangents is subtracted from the 2D visibility cone of that vertex (which might result in a visibility cone of two disjoint pieces, or an empty cone). Upon moving to an adjacent vertex on the boundary of the visibility polygon, the two tangent lines "slide around" the boundary of the hole and can be quickly updated.

2.3.3 Storing 2D Cones on Mesh Vertices

In a particular planar slice of the surface, the steps described above are performed to compute a 2D visibility cone at each vertex in the slice. Each slice vertex corresponds to some mesh edge, and the 2D cone is stored with the closest mesh vertex that is an endpoint of that mesh edge. When a mesh vertex is the recipient of a number of 2D cones, only one of the cones is stored (the cones are very similar, having all come from the same small region of the surface).

If, in a particular slice, a mesh vertex does not have a 2D cone, one is copied from the closest neighbour of the mesh vertex that does have a cone.

The number of slices is currently chosen manually so that the number of mesh vertices without cones is always below 5% of the mesh size. In the future, the number of slices should be chosen automatically. One method to do so is the following: The mesh vertices would be sorted in the direction perpendicular to the sweeping plane. The first slice would be placed to intersect the median vertex. After computing the 2D cones in that slice and placing them on mesh vertices, the mesh vertices that still do not have 2D cones would be placed in two groups: those above the slicing plane, and those below. The procedure

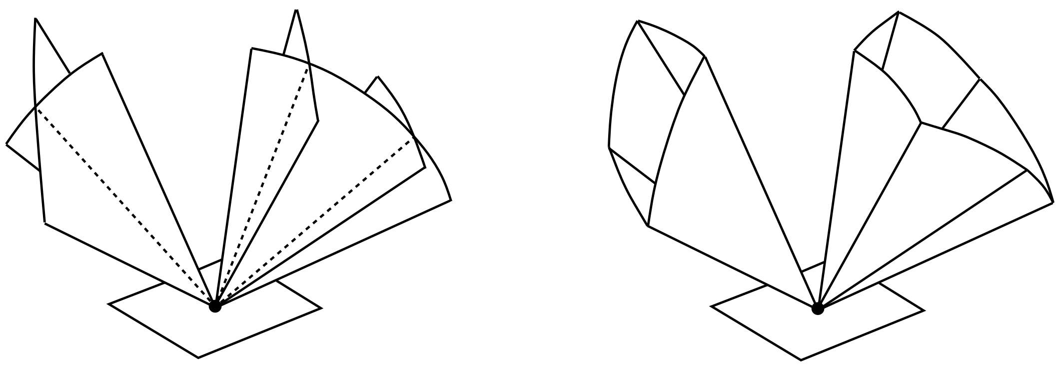

Fig. 6. A mesh vertex storing five 2D visibility cones is shown on the left. The 2D visibility cones fall into two equivalence classes. The two 3D visibility cones corresponding to these classes are shown on the right.

would be applied recursively to the two groups until the number of mesh vertices without cones falls below a user–defined threshold.

2.4 Building the 3D Visibility Cones

After many slices have been taken at many orientations, each mesh vertex will store many 2D visibility cones at various orientations. Finally, a 3D visibility cone is constructed for each mesh vertex by combining the 2D visibility cones stored at that vertex.

As the 2D visibility cones may consist of disjoint pieces, so may the 3D visibility cones consist of disjoint pieces. To identify these separate pieces, the 2D visibility cones stored at a mesh vertex, $\mathbf{v}$, are put into groups, as follows: Let $G(V, E)$ be a graph where the nodes, V, correspond to the 2D cones and where there is an edge between two nodes if the corresponding cones intersect at more than the cone vertex, $\mathbf{v}$. Then the groups of cones correspond to connected components of the graph: Two cones are in the same group if and only if there is a path in the graph between their corresponding nodes. See Figure 6.

For each group of connected 2D visibility cones, a 3D visibility cone is constructed as follows: The directions bounding the 2D cones of the group are normalized, resulting in a set of unit length vectors leaving the mesh vertex. Each endpoint of a unit length vector is projected onto the plane tangent to the surface at the mesh vertex. The convex hull of the projected points is computed and its centroid determined. The projected points (including those interior to the convex hull) are then sorted radially around the centroid. Finally, the 3D visibility cone is contructed: Its edges are the unit length vectors in the radial order of their projected endpoints. Note that the 3D visibility cone is not necessarily convex.

2.5 Algorithm Summary

In summary, the algorithm takes a number of planar slices of the surface at various positions and orientations. A 2D visibility cone is computed for each vertex within a slice, and that cone is stored on the closest mesh vertex. After all slices have been processed, the 2D cones stored at each mesh vertex are combined to form 3D visibility cones.

Memory requirements are fairly small, with the largest amount of memory being required to store the 2D visibility cones (the 3D cones later replace the 2D cones and take less total memory). For the shirt, each 2D cone had an average of 1.38 disjoint pieces and required six floats to store each piece. The 2964–point shirt mesh, using six orientations, required 1.1 Mb to store all of the 2D cones (for both the front and back faces of the mesh).

The running time of the algorithm depends linearly upon the number of slices and, in practice, linearly on the number of vertices in each slice: Over the 300 slices of the shirt (each slice having between zero and a few hundred vertices) the average time per vertex was 0.41 milliseconds. The exact time per vertex varied between 0.31 and 0.48 milliseconds. In theory, the triangulation of a slice of n vertices takes $\mathcal{O}(n \log n)$ time (or better, depending on the algorithm), while the shortest paths take $\mathcal{O}(n)$ time, and treating holes can take $\mathcal{O}(n^2)$ time (but occurs infrequently).

3 Shading the Cloth

The 3D visibility cone is used to determine the direct primary irradiance of a mesh point:

- If the illumination comes from a point light source, it is sufficient to test whether the point lies in the visibility cone.
- If a uniform diffuse area light source illuminates the surface, the area source is intersected with the visibility cone and a contour integral around the boundary of the part of the source inside the cone yields the direct primary irradiance [14,15].
- If the surface is illuminated by a uniform diffuse spherical source that surrounds the surface, a contour integral can be applied to the boundary of the visibility cone in the same manner as that of the area source.

The 3D visibility cone only permits an estimate of the direct primary irradiance from the environment beyond the surface; it does not permit an estimate of secondary, “multiple bounce” irradiance arriving from other parts of the surface.

A local illumination model described by Stewart and Langer [2] can be applied to estimate the secondary irradiance. The use of this model yields perceptually acceptable shading (see the images of Section 4) without resorting to an expensive global illumination step.

4 Experimental Results

The algorithm was applied to a shirt mesh containing 2964 points and 5752 faces. As mentioned in Section 2.3, visibility cones are computed for both sides of the surface, so the problem size is effectively doubled. (It would not have been possible to ignore the interior faces of the mesh, since those faces are visible and receive illumination.)

50 slices of the shirt were taken in each of six orientations: (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), and (0,1,1). The surface was surrounded by a uniform diffuse hemispheric source, which approximates the lighting of a cloudy day. The computed shading is shown in Figure 7.

Using six orientations of 50 slices each, the 3D visibility cones took 20.7 seconds to compute on a 300 MHz Pentium PC. Using those 3D visibility cones, the subsequent shading calculation took 1.5 seconds.

5 Comparison with Other Approaches

Ray shooting is a general method to determine surface shading: A set of rays is shot from the surface point and radiance is collected from the surfaces intersected by the rays [16–18]. This is a general technique that may be applied in almost any situation, and it is very straightforward to implement a basic version of ray shooting. However, this method is potentially expensive [1] since each ray might be tested against every face of the surface mesh (but a bounding volume hierarchy can reduce the cost substantially). An additional problem is that aliasing may occur where too few rays are used. However, the shading method of this paper may also suffer from aliasing if too few slicing planes are used.

The shirt mesh was also rendering with the POVray ray tracer [19]. The diffuse

Fig. 7. Three views of a 2964–point mesh surrounded by a diffuse hemispheric source. Note the gentle shading in the folds and under the arms, and how the interior face of the flared opening gradually becomes darker toward the interior.

hemispheric sky was simulated in POVray with 52 point sources scattered over a hemicube surrounding the shirt. The resulting image is shown in Figure 8. That image took 118 seconds to produce on a 300 MHz Pentium. However, if POVray were used to compute the shading at only the mesh vertices, it takes only 12.5 seconds, which is faster than the 22.2 seconds taken by the algorithm of this paper (which is an unoptimized prototype implementation).

The advantage of the algorithm of this paper over classical ray tracing is that the visibility cones need only to be computed once: After that, the surface can quickly be shaded under any form of illumination (in 1.5 seconds for the

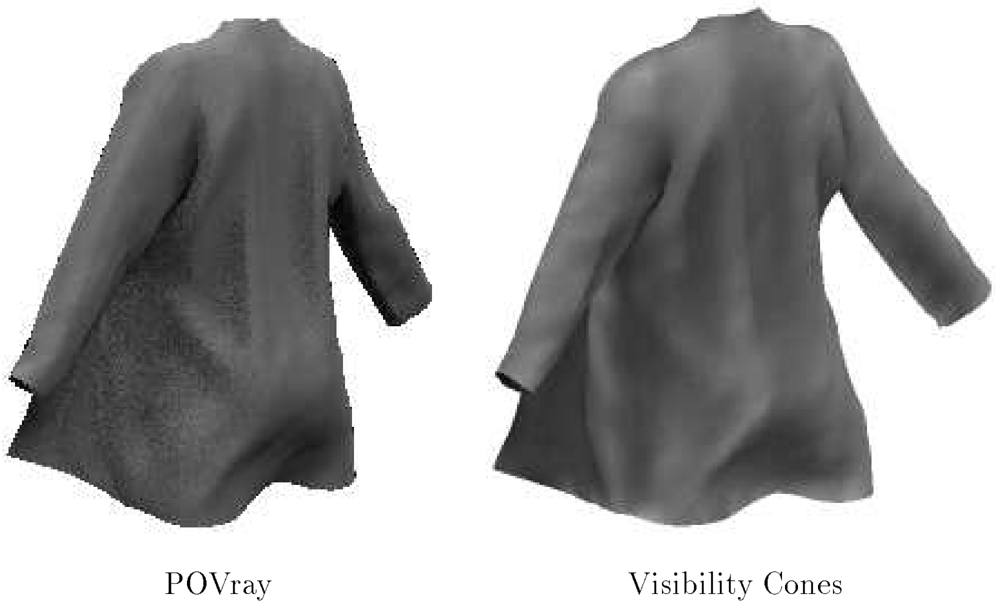

Fig. 8. POVray is compared to the visibility cone algorithm. The speckling due to undersampling in the POVray image would likely be diminished if samples were taken only on the mesh vertices (as is done by the visibility cone algorithm). There is a slight difference in viewpoints.

hemispheric source, for example). With ray tracing, a change in the illumination or in the viewpoint requires that the ray tracing be started anew. The visibility cone technique is superior when illumination and viewpoint change frequently, such as in lighting design applications.

Shadow volumes [20,21] may be used to determine surface shading from point light sources, or from an area light source if one is willing to sample at enough points on the area source. These limitations are not true of the method of this paper, which may be used with any distribution of illumination (at its most general, the illumination could be represented with an environment map).

There are several algorithms to compute visibility cones for a height fields, which are a restricted class of surfaces. These methods have all the benefits of the method described in this paper, but may not be applied to surfaces with folds, such as most cloth. Among the algorithms to compute visibility cones in height fields are those of Cohen-Or [22], Paglieroni [23], and Stewart [24].

6 Summary and Future Work

The key idea of the algorithm is to reduce the visibility computation to two dimensions and to exploit coherence within each planar visibility polygon by computing the tree of shortest paths. This avoids the potentially large cost of methods that would treat each mesh point independently.

The advantages of the algorithm include a quick computation of the 3D visibility cones (7 milliseconds per mesh point for the shirt) and quick shading under different lighting conditions (0.5 milliseconds per mesh point for the shirt).

In future work, coherence between adjacent parallel slices could be exploited. As the slicing plane is moved continuously through space, the surface cross–section within the slice usually varies continuously. One could consider the planar visibility polygons as continuous functions of the position of the slicing plane. At some initial position, the algorithm of this paper would be used to compute the initial trees of shortest paths. As the slicing plane moves, the edges of the shortest path tree change at discrete times that can be predicated and updated in order to continuously maintain the 2D visibility cones. However, upon a topological change in the slicing plane, it would be necessary to recompute many of the shortest path trees.

Also in future work, moving cloth will be treated. With moving cloth and a fixed slicing plane, the visibility polygons usually vary continuously with time, and the shortest path trees could be maintained as described above.

7 Acknowledgements

The author would like to thank several people and organizations that have contributed to this work.

The experiments would not have been possible without the kindness of Xavier Provot and Jeremy Denise of INRIA, who provided the high quality mesh model of the shirt. The mesh was particularly impressive in that it contained absolutely no self intersections.

The triangulation code, "Fast Industrial–Strength Triangulation," was provided by Martin Held of the Universität Salzburg. The code is robust in presence of very small features and correctly handles polygons with dangling edge chains and holes. This code succeeded where other implementations failed.

The convex hull code, "QHull," was implemented by Bradford Barber and Hannu Huhdanpaa and was provided by The Geometry Center at the University of Minnesota. Of the four convex hull implementations that the author tried, Qhull was the most robust and most often produced the correct convex hull in the presence of very close vertices.

The author's work is supported in part by NSERC, the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. T. Kajiya, "The rendering equation", *Computer Graphics*, vol. 20, no. 4, pp. 143–150, Aug. 1986.

[2] A. J. Stewart and M. S. Langer, "Towards accurate recovery of shape from shading under diffuse lighting", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 9, pp. 1020–1025, September 1997.

[3] M. I. Shamos and D. Hoey, "Geometric intersection problems", *7th Annual IEEE Symposium on Foundations of Comp. Sci.*, pp. 208–215, Oct. 1976.

[4] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters*, vol. 1, pp. 132–133, 1972.

[5] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane", *Information Processing Letters*, vol. 2, no. 1, pp. 18–21, Mar. 1973.

[6] D. G. Kirkpatrick and R. Seidel, "The ultimate planar convex hull algorithm?", *SIAM Journal on Computing*, vol. 15, no. 1, pp. 287–299, 1986.

[7] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls", *ACM Transactions on Mathematical Software*, vol. 22, no. 4, pp. 469–483, Dec. 1996.

[8] D. T. Lee and F. P. Preparata, "Euclidean Shortest Paths in the Presence of Rectilinear Barriers", *Networks*, vol. 14, no. 3, pp. 393–410, 1984.

[9] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, "Linear-Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons", *Algorithmica*, vol. 2, pp. 209–233, 1987.

[10] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, "Triangulating a simple polygon", *Information Processing Letters*, vol. 7, no. 4, pp. 175–179, June 1978.

[11] R. E. Tarjan and C. J. Van Wyk, "An $O(n \log\log n)$-time algorithm for triangulating a simple polygon", *SIAM Journal on Computing*, vol. 17, no. 1, pp. 143–178, Feb. 1988.

[12] B. Chazelle, "Triangulating a simple polygon in linear time", *Disc. and Comp. Geometry*, vol. 6, pp. 485–524, 1991.

[13] M. Held, "Efficient and reliable triangulation of polygons", in *Computer Graphics International*, 1998, `www.cosy.sbg.ac.at/~held`.

[14] C. Vedel, "Computing illumination from area light sources by approximate contour integration", in *Graphics Interface*, 1993, pp. 237–243.

[15] F. Sillion and C. Puech, *Radiosity and Global Illumination*, Morgan Kaufmann, San Francisco, CA, 1994.

[16] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing", *Computer Graphics*, vol. 18, no. 3, pp. 137–145, July 1984.

[17] G. J. Ward, "The RADIANCE lighting simulation and rendering system", *Computer Graphics*, vol. 28, no. Annual Conference Series, pp. 459–472, 1994.

[18] S. Coquillart and M. Gangnet, "Shaded display of digital maps", *IEEE Computer Graphics and Applications*, vol. 4, no. 7, pp. 35–42, July 1984.

[19] POVray team, "Persistence of vision ray tracer", `www.povray.org`.

[20] N. Chin, "Near real-time object-precision shadow generation using BSP trees—master thesis", Tech. Rep. CUCS-068-90, University of Columbia, 1990.

[21] K. Kaneda, F. Kato, E. Nakamae, T. Nishita, H. Tanaka, and T. Noguchi, "Three dimensional terrain modeling and display for environmental assessment", in *Computer Graphics (SIGGRAPH '89 Proceedings)*, July 1989, vol. 23, pp. 207–214.

[22] D. Cohen-Or and A. Shaked, "Visibility and dead-zones in digital terrain maps", *Computer Graphics Forum*, vol. 14, no. 3, pp. C/171–C/180, Sept. 1995.

[23] D. W. Paglieroni, "The directional parameter plane transform of a height field", *ACM Transactions on Graphics*, vol. 17, no. 1, pp. 50–70, Jan. 1998.

[24] A. J. Stewart, "Fast horizon computation at all points of a terrain with visibility and shading applications", *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 82–93, March 1998.