Polyhedral Visual Hulls for Real-Time Rendering

Wojciech Matusik Chris Buehler Leonard McMillan
MIT Laboratory for Computer Science

Abstract. We present new algorithms for creating and rendering visual hulls in real-time. Unlike voxel or sampled approaches, we compute an exact polyhedral representation for the visual hull directly from the silhouettes. This representation has a number of advantages: 1) it is a view-independent representation, 2) it is well-suited to rendering with graphics hardware, and 3) it can be computed very quickly. We render these visual hulls with a view-dependent texturing strategy, which takes into account visibility information that is computed during the creation of the visual hull. We demonstrate these algorithms in a system that asynchronously renders dynamically created visual hulls in real-time. Our system outperforms similar systems of comparable computational power.

1 Introduction

A classical approach for determining a three-dimensional model from a set of images is to compute shape-from-silhouettes. Most often, shape-from-silhouette methods employ discrete volumetric representations [12, 19]. The use of this discrete volumetric representation invariably introduces quantization and aliasing artifacts into the resulting model (i.e. the resulting model seldom projects back to the original silhouettes).

Recently, algorithms have been developed for sampling and texturing visual hulls along a discrete set of viewing rays [10]. These algorithms have been developed in the context of a real-time system for acquiring and rendering dynamic geometry. These techniques do not suffer from aliasing effects when the viewing rays correspond to the pixels in a desired output image. In addition, the algorithms address the rendering problem by view-dependently texturing the visual hull with proper visibility. However, these algorithms are only useful when a view-dependent representation of the visual hull is desired.

In this paper, we present algorithms for computing and rendering an exact polyhedral representation of the visual hull. This representation has a number of desirable properties. First, it is a view-independent representation, which implies that it only needs to be computed once for a given set of input silhouettes. Second, the representation is well-suited to rendering with graphics hardware, which is optimized for triangular mesh processing. Third, this representation can be computed and rendered just as quickly as sampled representations, and thus it is useful for real-time applications.

We demonstrate our visual hull construction and rendering algorithms in a real-time system. The system receives input from multiple video cameras and constructs visual hull meshes as quickly as possible. A separate rendering process asynchronously renders these meshes using a novel view-dependent texturing strategy with visibility.

1.1 Previous Work

Laurentini [8] introduced the *visual hull* concept to describe the maximal volume that reproduces the silhouettes of an object. Strictly, the visual hull is the maximal volume constructed from all possible silhouettes. In this paper (and in almost any practical setting) we compute the visual hull of an object with respect to a finite number of silhouettes. The silhouette seen by a pinhole camera determines a three-dimensional volume that originates from the camera's center of projection and extends infinitely while passing through the silhouette's contour on the image plane. We call this volume a silhouette cone. All silhouette cones exhibit the hull property in that they contain the actual geometry that produced the silhouette. For our purposes, a visual hull is defined as the three-dimensional intersection of silhouette cones from a set of pinhole silhouette images.

Visual hulls are most often computed using a discrete three-dimensional grid of volume elements (voxels). This technique, known as voxel carving [12, 19], proceeds by projecting each voxel onto each of the source image planes, and removing those voxels that fall completely outside of any silhouette. Octree-hierarchies are often used to accelerate this procedure. Related to voxel approaches, a recent algorithm computes discrete slices of the visual hull using graphics hardware for acceleration [9]. Other approaches improve the shape using splines [17] or color information [18].

If the primary purpose of a shape representation is to produce new renderings of that shape from different viewing conditions, then construction of an explicit model is not necessary. The image-based visual hull technique introduced in [10], renders unique views of the visual hull directly from the silhouette images, without constructing an intermediate volumetric or polyhedral model. This is accomplished by merging the cone intersection calculation with the rendering process, resulting in an algorithm similar in spirit to CSG ray casting [15].

However, sometimes an explicit three-dimensional model of the visual hull is desired. There has been work [4, 14] on general Boolean operations on 3D polyhedra. Most of these algorithms require decomposing the input polyhedra into convex polyhedra. Then, the operations are carried out on the convex polyhedra. By contrast, our algorithm makes no convexity assumptions; instead we exploit the fact that each of the intersection primitives (i.e., silhouette cones) are generalized cones with constant scaled cross-section. The algorithm in [16] also exploits the same property of silhouette cones, but exhibits performance unsuitable for real-time use.

View-dependent rendering is very popular for models that are acquired from real images (e.g., see [13]). The rendering algorithm that we use is closely related to view-dependent texture mapping (VDTM), introduced in [5] and implemented in real-time in [6]. The particular algorithm that we use is different from those two, and it is based on the unstructured lumigraph rendering (ULR) algorithm in [3]. In our implementation, we extend the ULR algorithm to handle visibility, which was not covered in the original paper.

Our real-time system is similar to previous systems. The system in [11] constructs visual hull models using voxels and uses view-dependent texture mapping for rendering, but the processing is done as an off-line process. The Virtualized Reality system [7] also constructs models of dynamic event using a variety of techniques including multi-baseline stereo.

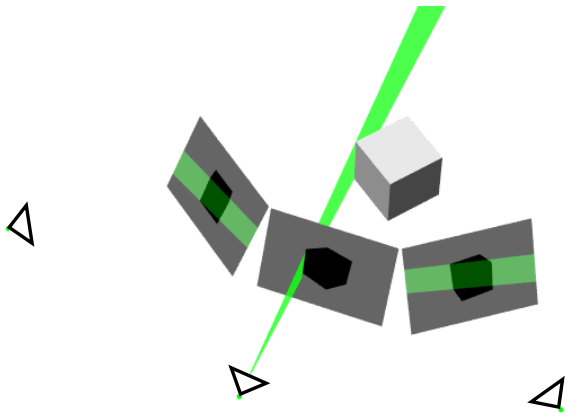


Fig. 1. A single silhouette cone face is shown, defined by the edge in the center silhouette. Its projection in two other silhouettes is also shown.

2 Polyhedral Visual Hull Construction

We assume that each silhouette s is specified by a set of convex or non-convex 2D polygons. These polygons can have holes. Each polygon consists of a set of edges joining consecutive vertices that define its (possibly multiple) contours. Moreover, for each silhouette s we know the projection matrix associated with the imaging device (e.g., video camera) that generated the silhouette. We use a 4×4 projection matrix that maps 3D coordinates to image (silhouette) coordinates, and whose inverse maps image coordinates to 3D directions.

2.1 Algorithm Outline

In order to compute the visual hull with respect to the input silhouettes, we need to compute the intersection of the cones defined by the input silhouettes. The resulting polyhedron is described by all of its faces. Note that the faces of this polyhedron can only lie on the faces of the original cones, and the faces of the original cones are defined by the projection matrices and the edges in the input silhouettes.

Thus, a simple algorithm for computing the visual hull might do the following: For each input silhouette s_i and for each edge e in the input silhouette s_i we compute the face of the cone. Then we intersect this face with the cones of all other input silhouettes. The result of these intersections is a set of polygons that define the surface of the visual hull.

2.2 Reduction to 2D

The intersection of a face of a cone with other cones is a 3D operation (these are polygon-polyhedron intersections). It was observed by [10, 16] that these intersections can be reduced to simpler intersections in 2D. This is because each of the silhouette cones has a fixed scaled cross-section; that is, it is defined by a 2D silhouette. Reduction to 2D also allows for less complex 2D data structures to accelerate the intersections.

To compute the intersection of a face f of a cone $cone(s_i)$ with a cone $cone(s_j)$, we project f onto the image plane of silhouette s_j (see Figure 1). Then we compute the intersection of projected face f with silhouette s_j. Finally, we project back the resulting polygons onto the plane of face f.

2.3 Efficient Intersection of Projected Cones and Silhouettes

In the previous section, we discussed intersecting a projected cone face f with a silhouette s_j. If we repeat this operation for every projected cone face in $cone(s_i)$, then we will have intersected the entire projected silhouette cone $cone(s_i)$ with silhouette s_j. In this section we show how to efficiently compute the intersection of the projected cone $cone(s_i)$ with the silhouette s_j. We accelerate the intersection process by pre-processing the silhouettes into Edge-Bin data structures as described in [10]. The Edge-Bin structure spatially partitions a silhouette so that we can quickly compute the set of edges that a projected cone face intersects. In the following, we abbreviate $cone(s_i)$ as c_i for simplicity.

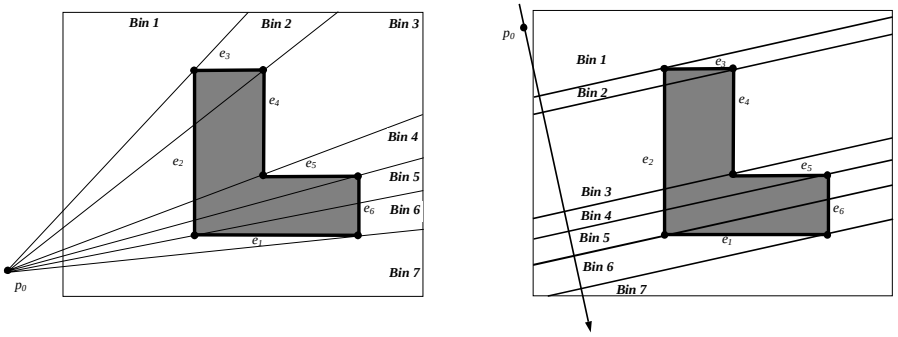
Construction of Edge-Bins. First, we observe that in case of perspective projection all rays on the surface of the cone c_i project to a pencil of lines sharing a common point p_0 (i.e., the epipole) in the image plane of s_j. We can parameterize all projected lines based on the slope α that these lines make with some reference line. Given this parameterization we partition the domain of $\alpha = (-\infty, \infty)$ into ranges such that any projected line with the slope falling inside of the given range always intersects the same set of edges of the silhouette s_j. We define a bin b_k to be a three-tuple: the start α_{start}, the end α_{end} of the range, and a corresponding set of edges S_k, $b_k = (\alpha_{start}, \alpha_{end}, S_k)$. We note that each silhouette vertex corresponds to a line that defines a range boundary.

In certain configurations, all rays project to a set of parallel epipolar lines in the image plane of s_j. When this case occurs, we use a line $p(\alpha) = p_0 + d\alpha$ to parameterize the lines, where p_0 is some arbitrary point on the line $p(\alpha)$ and d is a vector perpendicular to the direction of the projected rays. To define bins, we use the values of the parameter α at the intersection points of the line $p(\alpha)$ with the epipolar lines passing through the silhouette vertices. In this way we can describe the boundary of the bin using two values α_{start} and α_{end}, where α_{start}, α_{end} are the values of α for the lines passing through two silhouette vertices that define the region.

The Edge-Bin construction involves two steps. First, we sort the silhouette vertices based on the value of the parameter α. The lines that pass through the silhouette vertices define the bin boundaries.

Next, we observe that two consecutive slopes in the sorted list define α_{start} and α_{end} for each bin. To compute a set of edges assigned to each bin we traverse the sorted list of silhouette vertices. At the same time we maintain the list of edges in the current bin. When we visit a vertex of the silhouette we remove from the current bin an edge that ends at this vertex, and we add an edge that starts at the vertex. The start of an edge is defined as the edge endpoint that has a smaller value of parameter α. In Figure 2 we show a simple silhouette, bins, and corresponding edges for each bin.

The edges in each bin need to be sorted based on the increasing distance from the point p_0 (or the distance from parameterization line $p(\alpha)$ in case of the parallel lines). The efficient algorithm first performs a partial ordering on all the edges in the silhouette such that the edges closer to the point p_0 are first in the list. Then, when the bins are constructed the edges are inserted in the bins in the correct order.



Bin	1	2	3	4	5	6	7
Edges	$\emptyset$	e_2, e_3	e_2, e_4	e_2, e_5	e_2, e_6	e_1, e_6	$\emptyset$

Fig. 2. Two example silhouettes and their corresponding Edge-Bin data structures. Two cases are shown, one with convergent bins and one with parallel bins. The edges that are stored in the bins are listed in the accompanying table.

Efficient Intersection of the Projected Cone Faces with a Silhouette. Using the edge bin data structure, we can compute efficiently the intersection of the projected cone c_i with the silhouette s_j of some other cone c_j. In order to compute the intersection we process the faces of cone c_i in consecutive order. We start by projecting the first face f_1 onto the plane of silhouette s_j. The projected face f_1 is defined by its boundary lines with the values α_{p1} , α_{p2}. First, we need to find a bin $b = \{\alpha_{start}, \alpha_{end}, S\}$ such that $\alpha_{p1} \in (\alpha_{start}, \alpha_{end})$. Then, we intersect the line α_{p1} with all the edges in S. Since the edges in S are sorted based on the increasing distance from the projected vertex of cone c_i (or distance from line $p(\alpha)$ in case of parallel lines) we can immediately compute the edges of the resulting intersection that lie on line α_{p1}. Next, we traverse the bins in the direction of the value α_{p2}. As we move across the bins we build the intersection polygons by adding the vertices that define the bins. When we get to the bin $b' = \{\alpha'_{start}, \alpha'_{end}, S'\}$ such that $\alpha_{p2} \in (\alpha'_{start}, \alpha'_{end})$ we intersect the line α_{p2} with all edges in S' and compute the remaining edges of the resulting polygons. It is important to note that the next projected face f_2 is defined by the boundary lines α_{p2}, α_{p3}. Therefore, we do not have to search for the bin α_{p2} falls into. In this manner we compute the intersection of all projected faces of cone c_i with the silhouette s_j.

2.4 Calculating Visual Hull Faces

In the previous section we described how to perform the intersection of two cones efficiently. Performing the pairwise intersection on all pairs of cones results in $k-1$ polygon sets for each face of each cone, where k is the total number of silhouettes. The faces of the visual hull are the intersections of these polygon sets at each cone face. It is possible to perform the intersection of these polygon sets using standard algorithms for Boolean operations [1, 2], but we use a custom algorithm instead that is easy to implement and can output triangles directly.

Our polygon intersection routine works by decomposing arbitrary polygons into quadrilaterals and intersecting those. In Figure 3, we demonstrate the procedure with

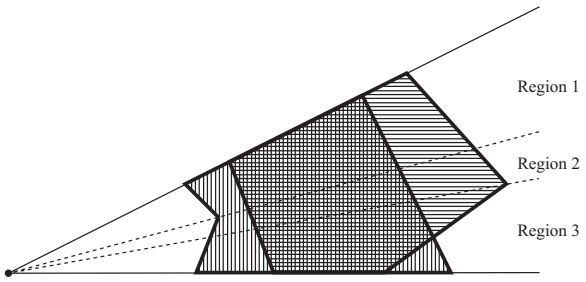


Fig. 3. Our polygon intersection routine subdivides polygons into quadrilaterals for intersection.

two 5-sided polygons, one with vertical hatching and the other with horizontal hatching. We first divide the space occupied by the polygons into triangular regions based on the polygons' vertices and the apex of the silhouette cone (similar to the Edge-Bin construction process). Note that within each triangular region, the polygon pieces are quadrilaterals. Then, we intersect the quadrilaterals in each region and combine all of the results into the final polygon, shown with both horizontal and vertical hatching.

The resulting polyhedral visual hull includes redundant copies of each vertex in the polyhedron (in fact, the number of copies of each vertex is equal to the degree of the vertex divided by 2). To optionally eliminate the redundant copies, we simply merge identical vertices. Ideally, our algorithm produces a watertight triangular mesh. However, because of our non-optimal face intersection routine, our meshes may contain T-junctions which violate the watertight property.

2.5 Visibility

In order to properly texture map the visual hull we need to determine which parts of the visual hull surface are visible from which cameras.

This visibility problem is equivalent to the shadow determination problem where one places a point light source at each reference camera position, and the goal is to determine which parts of the scene are illuminated by the light and which parts lie in a shadow. Standard graphics (hardware) algorithms are directly applicable since we have a mesh representation of the visual hull surface. However, they require rendering the scene from each input camera viewpoint and reading the z-buffer or the frame-buffer. These operations can be slow (reading the frame and z-buffer can be slow) and they can suffer from the quantization artifacts of the z-buffer.

We present an alternative novel software algorithm that computes the visible parts of the visual hull surface from each of the input cameras. The algorithm has the advantages that it is simple, and it can be computed virtually at no cost at the same time that we compute the visual hull polygons.

Let us assume that we want to compute whether the faces of the visual hull that lie on the extruded edge i in silhouette s_j are visible from image k.

We observe that these faces have to be visible from the camera k if the edge i is visible from the epipole p_0 (the projection of the center of projection of image k onto the image plane of camera j). This effectively reduces the 3D visibility computation to the 2D visibility computation. Moreover, we can perform the 2D visibility computation very efficiently using the edge-bin data structures that we already computed during the

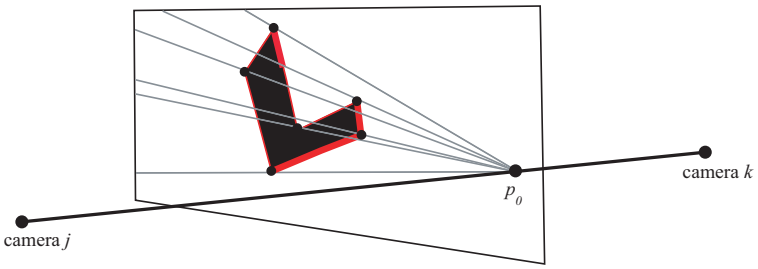


Fig. 4. We perform a conservative visibility test in 2D. In this example, the thick edges in the silhouette of camera c_j have been determined to be visible by camera c_k. These 2D edges correspond to 3D faces in the polyhedral visual hull.

visual hull computation.

First, we label all edges invisible. Then, to determine the visibility of edges in image j with respect to image k we traverse each bin in the Edge-Bin data structure. For each bin, we label the part of the first edge that lies in the bin as visible (see Figure 4). The edges in the bin are sorted in the increasing distance from the epipole; thus, the first edge in the bin corresponds to the front-most surface.

If the edge is visible in its full extent (if it is visible in all the bins in which it resides) then the edge is visible. If the edge is visible in some of its extent (if it is visible only in some bins in which it resides) then the edge is partially visible. The easiest solution in this case is to break it into the visible and invisible segments when computing the faces of the visual hull.

The visibility computation described in this section is conservative; we never label an edge visible if it is in fact invisible. However, it is often over-conservative, especially for objects whose silhouettes contains many holes.

3 View-Dependent Texturing

We have applied a novel view-dependent texturing strategy for rendering our polyhedral visual hull models in real-time. Our algorithm is based on the unstructured lumigraph rendering (ULR) algorithm detailed in [3], and we have added extensions to handle the visibility information computed during the visual hull construction.

The core idea of ULR is that the influence of a single image on the final rendering is a smoothly varying function across the desired image plane (or, equivalently, across the geometry representing the scene). These smooth weighting functions combine to form a image "blending field" that specifies how much contribution each input image makes to each pixel in the output image. The assumption of smoothness suggests an efficient rendering strategy: sparsely sample the image blending field and reconstruct it using simple basis functions (e.g., linear hat functions). The reconstructed blending field is then used to blend pixels from the input images to form the output image.

In the case of real-time rendering, the blending field can be efficiently reconstructed by triangulating the samples and using hardware alpha interpolation across the faces of

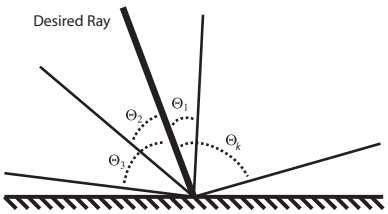


Fig. 5. Our k-nearest neighbor weighting is based on the k cameras with viewing rays closest in angle to the desired viewing ray. In this example, the desired ray is shown in bold in addition to four camera viewing rays. The angles of the k closest cameras are ordered such that $\Theta_1 \leq \Theta_2 \leq \Theta_3 \leq \ldots \leq \Theta_k$, and Θ_k is taken to be the threshold at which the weighting function falls to zero.

the triangles. The input image pixels are then blended together by projectively texturing mapping the triangles and accumulating the results in the frame buffer. The pseudocode for a multi-pass rendering version of the algorithm proceeds as follows:

```
Construct a list of blending field sample locations
for each input image i do
  for each blending field sample location do
    evaluate blending weight for image i and store in alpha channel
  end for
  Set current texture i
  Set current texture matrix P_i
  Draw triangulated samples using alpha channel blending weights
end for
```

The sample locations are simply 3D rays along which the blending field is evaluated. In the case when a reasonably dense model of the scene is available, sampling along the rays emanating from the desired viewpoint and passing through the vertices of the model is generally sufficient to capture the variation in the blending field. In this case, the triangles that are drawn are the actual triangles of the scene model. By contrast, in the general unstructured lumigraph case, one may sample rays randomly, and the triangles that are drawn may only roughly approximate the true scene geometry.

The texture matrix P_i is simply the projection matrix associated with camera i. It is rescaled to return texture coordinates between 0 and 1. In our real-time system, these matrices are obtained from a camera calibration process.

3.1 Evaluating the Blending Weights

Our view-dependent texturing algorithm evaluates the image blending field at each vertex of the visual hull model. The weight assigned to each image is calculated to favor those cameras whose view directions most closely match that of the desired view. The weighting that we use is the k-nearest neighbor weighting used in [3] and summarized here. For each vertex of the model, we find the k cameras whose viewings rays to that vertex are closest in angle to the desired viewing ray (see Figure 5). Consider the k^{th} ray with the largest viewing angle, Θ_k. We use this angle to define a local weighting function that maps the other angles into the range from 0 to 1: $weight(\Theta) = 1 - \frac{\Theta}{\Theta_k}$.

Applying this function to the k angles results (in general) in $k-1$ non-zero weights. We renormalize these weights to arrive at the final blending weights. In practice, we typically use $k = 3$ in our four camera system, which results in two non-zero weights at each vertex.

Although other weighting schemes are possible, this one is easy to implement and does not require any pre-processing such as in [6]. It results in a (mostly) smooth blending field except in degenerate cases, such as when k (or more) input rays are equidistant from the desired ray or when less than k nearest neighbors can be found (due to visibility or some other reason).

3.2 Handling Visibility

The algorithm in [3] does not explicitly handle the problem of visibility. In our case, we have visibility information available on a per-polygon basis. We can distinguish two possible approaches to incorporating this information: one that maintains a continuous blending field reconstruction and one that does not. A continuous blending field reconstruction is one in which the blending weights for the cameras on one side of a triangle edge are the same as on the other side of the edge. A continuous reconstruction generally has less apparent visual artifacts.

A simple rule for utilizing visibility while enforcing continuous reconstruction is the following: if vertex v belongs to *any* triangle t that is not visible from camera c, then do not consider c when calculating the blending weights for v. This rule causes camera c's influence to be zero across the face of triangle t, which is expected because t is not visible from c. It also forces c's influence to fall to zero along the other sides of the edges of t (assuming that the mesh is watertight) which results in a continuous blending function.

The assumption of a watertight mesh makes the continuous visibility rule unsuitable for our non-watertight visual hull meshes. Even with a watertight mesh, the mesh must be fairly densely tessellated, or the visibility boundaries may not be well-represented.

For these reasons, we relax the requirement of reconstruction continuity in our visibility treatment. When computing blending weights, we create a separate set of blending weights for each triangle. Each set of blending weights is computed considering only those cameras that see the triangle. When rendering, we replicate vertices so that we can specify different sets of blending weights per-triangle rather than per-vertex. Although this rendering algorithm is less elegant and more complex than the continuous algorithm, it works well enough in practice.

4 Real-Time System

The current system uses four calibrated Sony DFW-V500 IEEE-1394 video cameras. Each camera is attached to a separate client (600 MHz Athlon desktop PC). The cameras are synchronized to each other using an external trigger signal. Each client captures the video stream at 15 fps and performs the following processing steps: First, it segments out the foreground object using background subtraction. Then, the silhouette and texture information are compressed and sent over a 100Mb/s network to a central server. The system typically processes video at 320×240 resolution. It can optionally process 640×480 video at a reduced frame rate.

The central server (2x933MHz Pentium III PC) performs the majority of the computations. The server application has the following three threads:

- *Network Thread* - receives and decompresses the textures and silhouettes from the clients.
- *Construction Thread* - computes the silhouette simplification, volume intersection, and visibility.
- *Rendering Thread* - performs the view-dependent texturing and display.

Each thread runs independently of the others. This allows us to efficiently utilize the multiple processors of the server. It also enables us to render the visual hull at a faster rate than we compute it. As a result, end users perceive a higher frame rate than that at which the model is actually updated.

5 Results

Our system computes polyhedral visual hull models at a peak 15 frames per second, which is the frame rate at which our cameras run. The rendering algorithm is decoupled from the model construction, and it can run up to 30 frames per second depending on the model complexity. The actual frame rates of both components, especially rendering, are dependent on the model complexity, which in turn depends on the complexity of the input silhouette contours. In order to maintain a relatively constant frame rate, we simplify the input silhouettes with a coarser polygonal approximation. The amount of simplification is controlled by the current performance of the system.

In Figure 6, we show two flat-shaded renderings of a polyhedral visual hull that was captured in real-time from our system. These images demonstrate the typical models that our system produces. The main sources of error in creating these models is poor image segmentation and a small number of input images.

Figure 7 shows the same model view-dependently textured with four video images. In Figure 7a, the model is textured using information from our novel visibility algorithm. This results in a discontinuous reconstruction of the blending field, but it more accurately captures regions of the model that were not seen by the video cameras. In Figure 7b, the model is textured without visibility information. The resulting blending field is very smooth, although some visibility errors are made near occlusions.

Figure 8 shows visualizations of the blending fields of the previous two figures. Each of the four cameras is assigned a color (red, green, blue, and yellow), and the colors are blended together using the camera blending weights. It is clear from these images that the image produced using visibility information is discontinuous while the other image is not.

6 Future Work and Conclusions

In offline testing, our algorithms are sufficiently fast to run at full 30 frames per second on reasonable computer hardware. The maximum frame rate of our current live system is limited by the fact that our cameras can only capture images at 15 frames per second in synchronized mode. Clearly, it would improve the system to use better and more cameras that can run at 30 frames per second. Additional cameras would both improve the shape of the visual hulls and the quality of the view-dependent texturing.

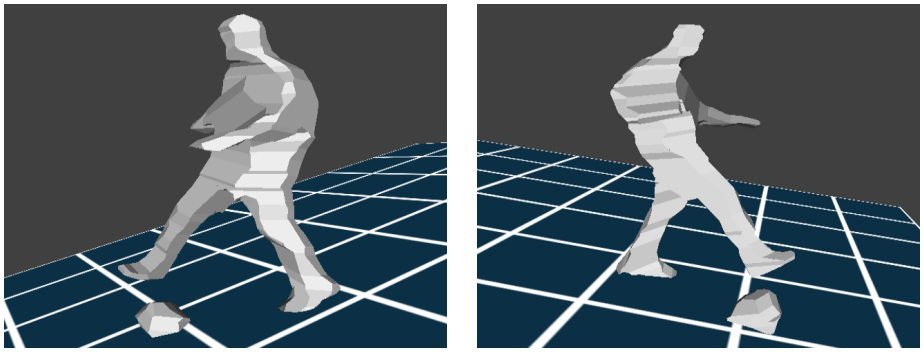
In the current system we compute and throw away a different mesh for each frame of video. For some applications it might be useful to derive the mesh of the next frame as a transformation of the mesh in the original frame and to store the original mesh plus the transformation function. Temporal processing such as this would also enable us to

accumulate the texture (radiance) of the model as it is seen from different viewpoints. Such accumulated texture information could be used to fill in parts that are invisible in one frame with information from other frames.

In this paper we have presented novel algorithms for efficiently computing and rendering polyhedral visual hulls directly from a set of images. We implemented and tested these algorithms in a real-time system. The speed of this system and the quality of the renderings are much better than previous systems using similar resources. The primary advantage of this system is that it produces polygonal meshes of the visual hull in each frame. As we demonstrated, these meshes can be rendered quickly using view-dependent texture mapping and graphics hardware.

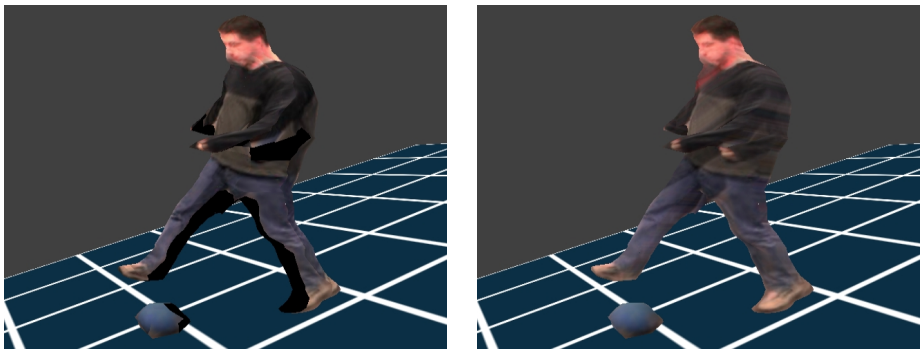
References

1. Balaban, I. J., "An Optimal Algorithm for Finding Segments Intersections," *Proc.* 11^{th} *Annual ACM Symposium on Computational Geometry*, (1995), pp. 211-219.
2. Bentley, J. and Ottmann, T., "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Trans. Comput.*, C-28, 9 (Sept. 1979), pp. 643-647.
3. Buehler, C., Bosse, M., Gortler, S., Cohen, M., McMillan, L., "Unstructured Lumigraph Rendering," To appear *SIGGRAPH 2001*.
4. Chazelle, B., "An Optimal Algorithm for Intersecting Three-Dimensional Convex Polyhedra," SIAM J. Computing, 21 (1992), pp. 671-696.
5. Debevec, P., Taylor, C., Malik. J., "Modeling and Rendering Architecture from Photographs," *SIGGRAPH 1996*, pp. 11-20.
6. Debevec, P., Yu, Y., Borshukov, G. D., "Efficient View-Dependent Image-Based Rendering with Projective Texture Mapping," *Eurographics Rendering Workshop*, (1998).
7. Kanade, T., P. W. Rander, P. J. Narayanan. "Virtualized Reality: Constructing Virtual Worlds from Real Scenes," *IEEE Multimedia*, 4, 1 (March 1997), pp. 34-47.
8. Laurentini, A., "The Visual Hull Concept for Silhouette Based Image Understanding," *IEEE PAMI*, 16, 2 (1994), pp. 150-162.
9. Lok, B., "Online Model Reconstruction for Interactive Virtual Environments," *I3D 2001*.
10. Matusik, W., Buehler, C., Raskar, R., Gortler, S., McMillan, L., "Image-Based Visual Hulls,"*SIGGRAPH 2000*, (July 2000), pp. 369-374.
11. Moezzi, S., D.Y. Kuramura, R. Jain. "Reality Modeling and Visualization from Multiple Video Sequences," *IEEE CG&A*, 16, 6 (Nov 1996), pp. 58-63.
12. Potmesil, M., "Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images," *CVGIP*, 40 (1987), pp. 1-29.
13. Pulli, K., Cohen, M., Duchamp, T., Hoppe, H., Shapiro, L., and Stuetzle, W., "View-based Rendering: Visualizing Real Objects from Scanned Range and Color Data," *8th Eurographics Workshop on Rendering*, 1997.
14. Rappoport, A. and Spitz, S., "Interactive Boolean Operations for Conceptual Design of 3D Solids," *SIGGRAPH 1997*, pp. 269-278.
15. Roth, S. D., "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, 18 (Feb 1982), pp. 109-144.
16. Rozenoer, M. and Shlyakhter, I., "Reconstruction of 3D Tree Models from Instrumented Photographs," M.Eng. Thesis, M.I.T., (1999).
17. Sullivan, S. and Ponce, J., "Automatic Model Construction, Pose Estimation, and Object Recognition from Photographs Using Triangular Splines," *ICCV '98*, pp. 510-516, 1998.
18. Seitz, S. and Dyer, C., "Photorealistic Scene Reconstruction by Voxel Coloring," *CVPR '97*, pp. 1067-1073, 1997.
19. Szeliski, R., "Rapid Octree Construction from Image Sequences," *CVGIP: Image Understanding*, 58, 1 (July 1993), pp. 23-32.



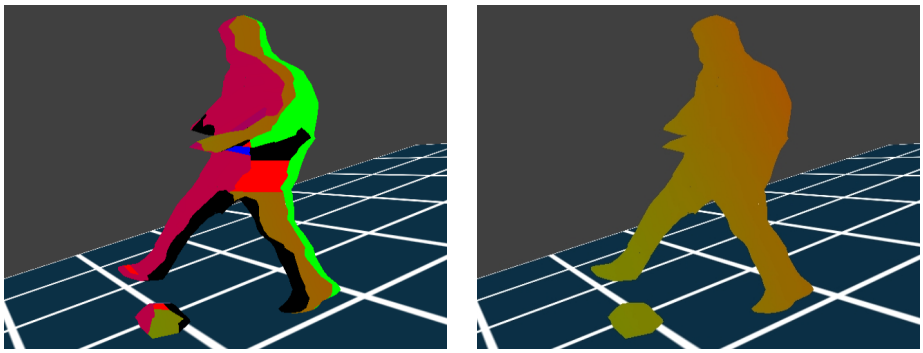
(a) (b)

Fig. 6. Two flat-shaded views of a polyhedral visual hull.



(a) (b)

Fig. 7. Two view-dependently textured views of the same visual hull model. The left rendering uses conservative visibility computed in real-time by our algorithm. The right view ignores visibility and blends the textures more smoothly but with potentially more errors.



(a) (b)

Fig. 8. Two visualizations of the camera blending field. The colors red, green, blue, and yellow correspond to the four cameras in our system. The blended colors demonstrate how each pixel is blended from each input image using both (a) visibility and (b) no visibility.