Image-Based Visual Hulls

Wojciech Matusik*
Laboratory for Computer Science
Massachusetts Institute of Technology

Chris Buehler*
Laboratory for Computer Science
Massachusetts Institute of Technology

Ramesh Raskar‡
Department of Computer Science
University of North Carolina - Chapel Hill

Steven J. Gortler†
Division of Engineering and Applied Sciences
Harvard University

Leonard McMillan*
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

In this paper, we describe an efficient image-based approach to computing and shading visual hulls from silhouette image data. Our algorithm takes advantage of epipolar geometry and incremental computation to achieve a constant rendering cost per rendered pixel. It does not suffer from the computation complexity, limited resolution, or quantization artifacts of previous volumetric approaches. We demonstrate the use of this algorithm in a real-time virtualized reality application running off a small number of video streams.

Keywords: Computer Vision, Image-Based Rendering, Constructive Solid Geometry, Misc. Rendering Algorithms.

1 Introduction

Visualizing and navigating within virtual environments composed of both real and synthetic objects has been a long-standing goal of computer graphics. The term "Virtualized Reality™", as popularized by Kanade [23], describes a setting where a real-world scene is "captured" by a collection of cameras and then viewed through a virtual camera, as if the scene was a synthetic computer graphics environment. In practice, this goal has been difficult to achieve. Previous attempts have employed a wide range of computer vision algorithms to extract an explicit geometric model of the desired scene.

Unfortunately, many computer vision algorithms (e.g. stereo vision, optical flow, and shape from shading) are too slow for real-time use. Consequently, most virtualized reality systems employ off-line post-processing of acquired video sequences. Furthermore, many computer vision algorithms make unrealistic simplifying assumptions (e.g. all surfaces are diffuse) or impose impractical restrictions (e.g. objects must have sufficient non-periodic textures) for robust operation. We present a new algorithm for synthesizing virtual renderings of real-world scenes in real time. Not only is our technique fast, it also makes few simplifying assumptions and has few restrictions.

*(wojciech | cbuehler | mcmillan)@graphics.lcs.mit.edu
†sjg@cs.harvard.edu
‡raskar@cs.unc.edu

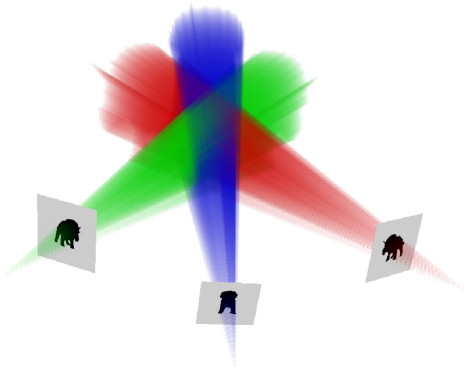

Figure 1 - The intersection of silhouette cones defines an approximate geometric representation of an object called the visual hull. A visual hull has several desirable properties: it contains the actual object, and it has consistent silhouettes.

Our algorithm is based on an approximate geometric representation of the depicted scene known as the visual hull (see Figure 1). A visual hull is constructed by using the visible silhouette information from a series of reference images to determine a conservative shell that progressively encloses the actual object. Based on the principle of *calculatus eliminatus* [28], the visual hull in some sense carves away regions of space where the object "is not".

The visual hull representation can be constructed by a series of 3D constructive solid geometry (CSG) intersections. Previous robust implementations of this algorithm have used fully enumerated volumetric representations or octrees. These methods typically have large memory requirements and thus, tend to be restricted to low-resolution representations.

In this paper, we show that one can efficiently render the exact visual hull without constructing an auxiliary geometric or volumetric representation. The algorithm we describe is "image based" in that all steps of the rendering process are computed in "image space" coordinates of the reference images.

We also use the reference images as textures when shading the visual hull. To determine reference images that can be used, we compute which reference cameras have an unoccluded view of each point on the visual hull. We present an image-based visibility algorithm based on epipolar geometry and McMillan's occlusion compatible ordering [18] that allows us to shade the visual hull in roughly constant time per output pixel.

Using our *image-based visual hull* (IBVH) algorithm, we have created a system that processes live video streams and renders the observed scene from a virtual camera's viewpoint in real time. The resulting representation can also be combined with traditional computer graphics objects.

2 Background and Previous Work

Kanade's virtualized reality system [20] [23] [13] is perhaps closest in spirit to the rendering system that we envision. Their initial implementations have used a collection of cameras in conjunction with multi-baseline stereo techniques to extract models of dynamic scenes. These methods require significant off-line processing, but they are exploring special-purpose hardware for this task. Recently, they have begun exploring volume-carving methods, which are closer to the approach that we use [26] [30].

Pollard's and Hayes' [21] immersive video objects allow rendering of real-time scenes by morphing live video streams to simulate three-dimensional camera motion. Their representation also uses silhouettes, but in a different manner. They match silhouette edges across pairs of views, and use these correspondences to compute morphs to novel views. This approach has some limitations, since silhouette edges are generally not consistent between views.

Visual Hull. Many researchers have used silhouette information to distinguish regions of 3D space where an object is and is not present [22] [8] [19]. The ultimate result of this carving is a shape called the object's *visual hull* [14]. A visual hull always contains the object. Moreover, it is an equal or tighter fit than the object's convex hull. Our algorithm computes a view-dependent, sampled version of an object's visual hull each rendered frame.

Suppose that some original 3D object is viewed from a set of reference views R. Each reference view r has the silhouette s_r with interior pixels covered by the object. For view r one creates the cone-like volume vh_r defined by all the rays starting at the image's point of view p_r and passing through these interior points on its image plane. It is guaranteed that the actual object must be contained in vh_r. This statement is true for all r; thus, the object must be contained in the volume $vh_R=\bigcap_{r\in R} vh_r$. As the size of R goes to infinity, and includes all possible views, vh_R converges to a shape known as the visual hull vh_∞ of the original geometry. The visual hull is not guaranteed to be the same as the original object since concave surface regions can never be distinguished using silhouette information alone.

In practice, one must construct approximate visual hulls using only a finite number of views. Given the set of views R, the approximation vh_R is the best conservative geometric description that one can achieve based on silhouette information alone (see Figure 1). If a conservative estimate is not required, then alternative representations are achievable by fitting higher order surface approximations to the observed data [2].

Volume Carving. Computing high-resolution visual hulls can be tricky matter. The intersection of the volumes vh_r requires some form of CSG. If the silhouettes are described with a polygonal mesh, then the CSG can be done using polyhedral CSG, but this is very hard to do in a robust manner.

A more common method used to convert silhouette contours into visual hulls is volume carving [22] [8] [29] [19] [5] [27]. This method removes unoccupied regions from an explicit volumetric representation. All voxels falling outside of the projected silhouette cone of a given view are eliminated from the volume. This process is repeated for each reference image. The resulting volume is a quantized representation of the visual hull according to the given volumetric grid. A major advantage of our view-dependent method is that it minimizes artifacts resulting from this quantization.

CSG Rendering. A number of algorithms have been developed for the fast rendering of CSG models, but most are ill suited for our task. The algorithm described by Rappoport [24], requires that each solid be first decomposed to a union of convex primitives. This decomposition can prove expensive for complicated silhouettes. Similarly, the algorithm described in [11] requires a rendering pass for each layer of depth complexity. Our method does not require preprocessing the silhouette cones. In fact, there is no explicit data structure used to represent the silhouette volumes other than the reference images.

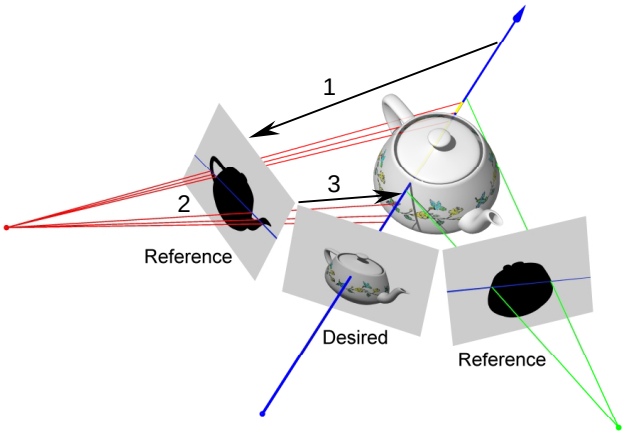

Figure 2 – Computing the IBVH involves three steps. First, the desired ray is projected onto a reference image. Next, the intervals where the projected ray crosses the silhouette are determined. Finally, these intervals are lifted back onto the desired ray where they can be intersected with intervals from other reference images.

Using ray tracing, one can render an object defined by a tree of CSG operations without explicitly computing the resulting solid [25]. This is done by considering each ray independently and computing the interval along the ray occupied by each object. The CSG operations can then be applied in 1D over the sets of intervals. This approach requires computing a 3D ray-solid intersection. In our system, the solids in question are a special class of cone-like shapes with a constant cross section in projection. This special form allows us to compute the equivalent of 3D ray intersections in 2D using the reference images.

Image-Based Rendering. Many different image-based rendering techniques have been proposed in recent years [3] [4] [15] [6] [12]. One advantage of image-based rendering techniques is their stunning realism, which is largely derived from the acquired images they use. However, a common limitation of these methods is an inability to model dynamic scenes. This is mainly due to data acquisition difficulties and preprocessing requirements. Our system generates image-based models in real-time, using the same images to construct the IBHV and to shade the final rendering.

3 Visual-Hull Computation

Our approach to computing the visual hull has two distinct characteristics: it is computed in the image space of the reference images and the resulting representation is viewpoint dependent. The advantage of performing geometric computations in image space is that it eliminates the resampling and quantization artifacts that plague volumetric approaches. We limit our sampling to the pixels of the desired image, resulting in a view-dependent visual-hull representation. In fact, our IBVH representation is equivalent to computing exact 3D silhouette cone intersections and rendering the result with traditional rendering methods.

Our technique for computing the visual hull is analogous to finding CSG intersections using a ray-casting approach [25].

Given a desired view, we compute each viewing ray's intersection with the visual hull. Since computing a visual hull involves only intersection operations, we can perform the CSG calculations in any order. Furthermore, in the visual hull context, every CSG primitive is a generalized cone (a projective extrusion of a 2D image silhouette). Because the cone has a fixed (scaled) cross section, the 3D ray intersections can be reduced to cheaper 2D ray intersections. As shown in Figure 2 we perform the following steps: 1) We project a 3D viewing ray into a reference image. 2) We perform the intersection of the projected ray with the 2D silhouette. These intersections result in a list of intervals along the ray that are interior to the cone's cross-section. 3) Each interval is then lifted back into 3D using a simple projective mapping, and then intersected with the results of the ray-cone intersections from other reference images. A naïve algorithm for computing these IBVH ray intersections follows:

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    for each scanline s in d
      for each pixel p in s
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        intervals int2D = calcIntervals(l2,r.silEdges)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

To analyze the efficiency of this algorithm, let n be the number of pixels in a scanline. The number of pixels in the image d is $O(n^2)$. Let k be the number of reference images. Then, the above algorithm has an asymptotic running time $O(ikn^2)$, where i is the time complexity of the `calcIntervals` routine. If we test for the intersection of each projected ray with each of the e edges of the silhouette, the running time of `calcIntervals` is $O(e)$. Given that l is the average number of times that a projected ray intersects the silhouette[1], the number of silhouette edges will be $O(ln)$. Thus, the running time of `IBVHisect` to compute all of the 2D intersections for a desired view is $O(lkn^3)$.

The performance of this naïve algorithm can be improved by taking advantage of incremental computations that are enabled by the epipolar geometry relating the reference and desired images. These improvements will allow us to reduce the amortized cost of 1D ray intersections to $O(l)$ per desired pixel, resulting in an implementation of `IBVHisect` that takes $O(lkn^2)$.

Given two camera views, a reference view r and a desired view d, we consider the set of planes that share the line connecting the cameras' centers. These planes are called *epipolar planes*. Each epipolar plane projects to a line in each of the two images, called an *epipolar line*. In each image, all such lines intersect at a common point, called the *epipole*, which is the projection of one of the camera's center onto the other camera's view plane [9].

As a scanline of the desired view is traversed, each pixel projects to an epipolar line segment in r. These line segments emanate from the epipole e_{dr}, the image of d's center of projection onto r's image plane (see Figure 3), and trace out a "pencil" of epipolar lines in r. The slopes of these epipolar line segments will either increase or decrease monotonically depending on the direction of traversal (Green arc in Figure 3). We take advantage of this monotonicity to compute silhouette intersections for the whole scanline incrementally.

[1] We assume reference images also have $O(n^2)$ pixels.

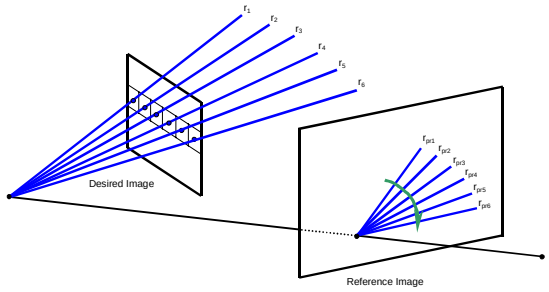

Figure 3 – The pixels of a scanline in the desired image trace out a pencil of line segments in the reference image. An ordered traversal of the scanline will sweep out these segments such that their slope about the epipole varies monotonically.

The silhouette contour of each reference view is represented as a list of edges enclosing the silhouette's boundary pixels. These edges are generated using a 2D variant of the marching cubes approach [16]. Next, we sort the $O(nl)$ contour vertices in increasing order by the slope of the line connecting each vertex to the epipole. These sorted vertex slopes divide the reference image domain into $O(nl)$ bins. Bin B_i has an extent spanning between the slopes of the ith and $i+1$st vertex in the sorted list. In each bin B_i we place all edges that are intersected by epipolar lines with a slope falling within the bin's extent[2]. During `IBVHisect` as we traverse the pixels along a scanline in the desired view, the projected corresponding view rays fan across the epipolar pencil in the reference view with either increasing or decreasing slope. Concurrently, we step through the list of bins. The appropriate bin for each epipolar line is found and it is intersected with the edges in that bin. This procedure is analogous to merging two sorted lists, which can be done in a time proportional to the length of the lists ($O(nl)$ in our case).

For each scanline in the desired image we evaluate n viewing rays. For each viewing ray we compute its intersection with edges in a single bin. Each bin contains on average $O(l)$ silhouette edges. Thus, this step takes $O(l)$ time per ray. Simultaneously we traverse the sorted set of $O(nl)$ bins as we traverse the scanline. Therefore, one scanline is computed in $O(nl)$ time. Over n scanlines of the desired image, and over k reference images, this gives a running time of $O(lkn^2)$. Pseudocode for the improved algorithm follows.

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    bins b = constructBins(r.caminfo, r.silEdges, d.caminfo)
    for each scanline s in d
      incDec order = traversalOrder(r.caminfo,d.caminfo,s)
      resetBinPositon(b)
      for each pixel p in s according to order
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        slope m = ComputeSlope(l2,r.caminfo,d.caminfo)
        updateBinPosition(b,m)
        intervals int2D = calcIntervals(l2,b.currentbin)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

[2] Sorting the contour vertices takes $O(nl \log(nl))$ and binning takes $O(nl^2)$. Sorting and binning over k reference views takes $O(knl \log(nl))$ and $O(knl^2)$ correspondingly. In our setting, $l << n$ so we view this preprocessing stage as negligible.

It is tempting to apply further optimizations to take greater advantage of epipolar constraints. In particular, one might consider rectifying each reference image with the desired image prior to the ray-silhouette intersections. This would eliminate the need to sort, bin, and traverse the silhouette edge lists. However, a call to `liftInterval` would still be required for each pixel, giving the same asymptotic performance as the algorithm presented. The disadvantage of rectification is the artifacts introduced by the two resampling stages that it requires. The first resampling is applied to the reference silhouette to map it to the rectified frame. The second is needed to unrectify the computed intervals of the desired view. In the typical stereo case, the artifacts of rectification are minimal because of the closeness of the cameras and the similarity of their pose. But, when computing visual hulls the reference cameras are positioned more freely. In fact, it is not unreasonable for the epipole of a reference camera to fall within the field of view of the desired camera. In such a configuration, rectification is degenerate.

4 Visual-Hull Shading

The IBVH is shaded using the reference images as textures. In order to capture as many view-dependent effects as possible a view-dependent texturing strategy is used. At each pixel, the reference-image textures are ranked from "best" to "worst" according to the angle between the desired viewing ray and rays to each of the reference images from the closest visual hull point along the desired ray. We prefer those reference views with the smallest angle [7]. However, we must avoid texturing surface points with an image whose line-of-sight is blocked by some other point on the visual hull, regardless of how well aligned that view might be to the desired line-of-sight. Therefore, visibility must be considered during the shading process.

When the visibility of an object is determined using its visual hull instead of its actual geometry, the resulting test is conservative– erring on the side of declaring potentially visible points as non-visible. We compute visibility using the visual hull, VH_R, as determined by IBVHisect. This visual hull is represented as intervals along rays of the desired image *d*. Pseudocode for our shading algorithm is given below.

```
IBVHshade(intervalImage &d, refImList R){
  for each pixel p in d do
    p.best = BIGNUM
  for each referenceImage r in R do
    for each pixel p in d do
       ray3D ry3 = compute3Dray(p,d.camInfo)
       point3 pt3 = front(p.intervals,ry3)
       double s  = angleSimilarity(pt3,ry3,r.camInfo)
       if isVisible(pt3,r,d)
         if (s < p.best)
           point2 pt2 = project(pt3,r.camInfo)
           p.color = sample_color(pt2,r)
           p.best = s
}
```

The `front` procedure finds the front most geometric point of the IBVH seen along the ray. The `IBVHshade` algorithm has time complexity $O(vkn^2)$, where *v* is the cost for computing visibility of a pixel.

Once more we can take advantage of the epipolar geometry in order to incrementally determine the visibility of points on the visual hull. This reduces the amortized cost of computing visibility to $O(l)$ per desired pixel, thus giving an implementation of `IBVHshade` that takes $O(lkn^2)$.

Consider the visibility problem in flatland as shown in Figure 4. For a pixel *p*, we wish to determine if the front-most point on the visual hull is occluded with respect to a particular reference image by any other pixel interval in *d*.

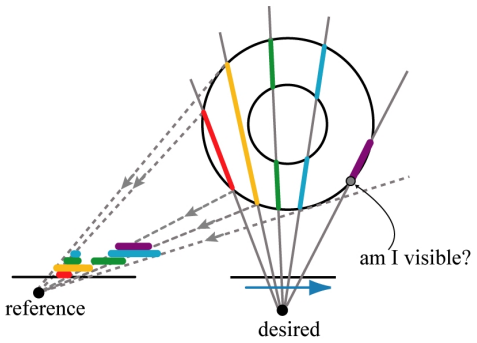

Figure 4 – In order to compute the visibility of an IBVH sample with respect to a given reference image, a series of IBVH intervals are projected back onto the reference image in an occlusion-compatible order. The front-most point of the interval is visible if it lies outside of the unions of all preceding intervals.

Efficient calculation can proceed as follows. For each reference view *r*, we traverse the desired-view pixels in front-to-back order with respect to *r* (left-to-right in Figure 4). During traversal, we accumulate coverage intervals by projecting the IBVH pixel intervals into the reference view, and forming their union. For each front most point, `pt3`, we check to see if its projection in the reference view is already covered by the coverage intervals computed thus far. If it is covered, then `pt3` is occluded from *r* by the IBVH. Otherwise, `pt3` is not occluded from *r* by either the IBVH or the actual (unknown) geometry.

```
visibility2D(intervalFlatlandImage &d, referenceImage r){
  intervals  coverage = <empty>
  for each pixel p in d do \\front to back in r
    ray2D ry2 = compute2Dray(p,d.camInfo)
    point2 pt2 = front(p.intervals,ry2);
    point1D p1 = project(pt2,r.camInfo)
    if contained(p1,coverage)
      p.visible[r] = false
    else
      p.visible[r] = true
    intervals tmp  =
                prjctIntrvls(p.intervals,ry2,r.camInfo)
    coverage = coverage UNION tmp
}
```

This algorithm runs in $O(nl)$, since each pixel is visited once, and containment test and unions can be computed in $O(l)$ time.

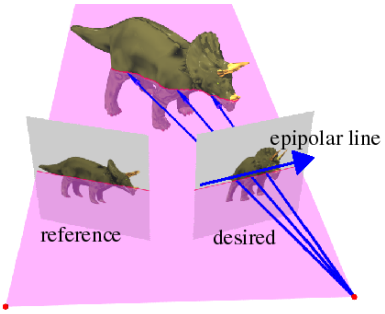

Figure 5 – Ideally, the visibility of points in 3D could be computed by applying the 2D algorithm along epipolar planes.

In the continuous case, 3D visibility calculations can be reduced to a set of 2D calculations within epipolar planes (Figure 5), since all visibility interactions occur within such planes. However, the extension of the discrete 2D algorithm to a complete discrete 3D solution is not trivial, as most of the discrete pixels in our images do not exactly share epipolar planes. Consequently, one must be careful in implementing conservative 3D visibility.

First, we consider each of the intervals stored in d as a solid frustum with square cross section. To determine visibility of a (square) pixel p correctly we consider $S_{p,}$ the set of all possible epipolar planes which touch p. There are at least two possible definitions for whether p is visible: (1) p is visible along **all** planes in S_p , (2) p is visible along **any** plane in $S_{p.}$ Clearly the first definition results in more pixels that are labeled not visible, therefore, it is better suited when using a large number of reference images. With a small number of reference images, the second definition is preferred. Implementing efficient exact algorithms for these visibility definitions is difficult, therefore, we use conservative algorithms; if the pixel is truly invisible we never label it as visible. However, the algorithms could label some pixel as invisible though it is in fact visible.

An algorithm that conservatively computes visibility according to the first definition is performed as follows. We define an epipolar wedge starting from the epipole e_{rd} in the desired view extending out to a one pixel-width interval on the image boundary. Depending on the relative camera views, we traverse the wedge either toward or away from the epipole [17]. For each pixel in this wedge, we compute visibility with respect to the pixels traversed earlier in the wedge using the 2D visibility algorithm. If a pixel is computed as *visible* then no geometry within the wedge could have occluded it in the reference view. We use a set of wedges whose union covers the whole image. A pixel may be touched by more than one wedge, in these cases its final visibility is computed as the *AND* of the results obtained from each wedge.

The algorithm for the second visibility definition works as follows. We do not consider all possible epipolar lines that touch pixel p but only some subset of them such that at least one line touches each pixel. One such subset is all the epipolar lines that pass through the centers of the image boundary pixels. This particular subset completely covers all the pixels in the desired image; denser subsets can also be chosen. The algorithm computes `visibility2D` for all epipolar lines in the subset. Visibility for a pixel might be computed more than once (e.g., the pixels near the epipole are traversed more often). We *OR* all obtained visibility results. Since we compute `visibility2D` for up to $4n$ epipolar lines in k reference images the total time complexity of this algorithm is $O(lkn^2)$. In our real-time system we use small number of reference images (typically four). Thus, we use the algorithm for the second definition of visibility.

The total time complexity of our IBVH algorithms is $O(lkn^2)$, which allows for efficient rendering of IBVH objects. These algorithms are well suited to distributed and parallel implementations. We have demonstrated this efficiency with a system that computes IBVHs in real time from live video sequences.

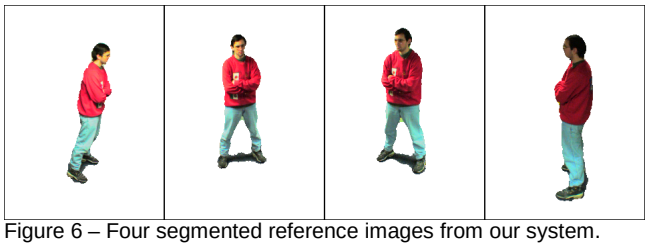

Figure 6 – Four segmented reference images from our system.

5 System Implementation

Our system uses four calibrated Sony DFW500 FireWire video cameras. We distribute the computation across five computers, four that process video and one that assembles the IBVH (see Figure 6). Each camera is attached to a 600 MHz desktop PC that captures the video frames and performs the following processing steps. First, it corrects for radial lens distortion using a lookup table. Then it segments out the foreground object using background-subtraction [1] [10]. Finally, the silhouette and texture information are compressed and sent over a 100Mb/s network to a central server for IBVH processing.

Our server is a quad-processor 550 MHz PC. We interleave the incoming frame information between the 4 processors to increase throughput. The server runs the IBVH intersection and shading algorithms. The resulting IBVH objects can be depth-buffer composited with an OpenGL background to produce a full scene. In the examples shown, a model of our graphics lab made with the Canoma modeling system was used as a background.

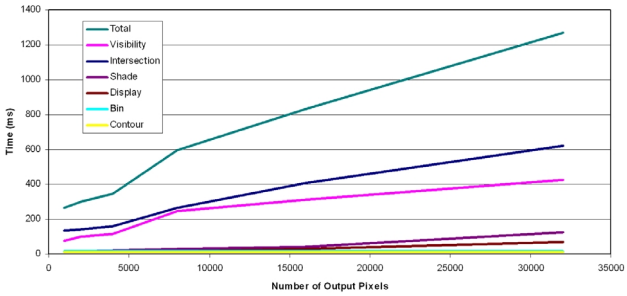

Figure 7 – A plot of the execution times for each step of the IBVH rendering algorithm on a single CPU. A typical IBVH might cover approximately 8000 pixels in a 640×480 image and it would execute at greater than 8 frames per second on our 4 CPU machine.

In Figure 7, the performances of the different stages in the IBVH algorithm are given. For these tests, 4 input images with resolutions of 256×256 were used. The average number of times that a projected ray crosses a silhouette is 6.5. Foreground segmentation (done on client) takes about 85 ms. We adjusted the field of view of the desired camera, to vary the number of pixels occupied by the object. This graph demonstrates the linear growth of our algorithm with respect to the number of output pixels.

6 Conclusions and Future Work

We have described a new image-based visual-hull rendering algorithm and a real-time system that uses it. The algorithm is efficient from both theoretical and practical standpoints, and the resulting system delivers promising results.

The choice of the visual hull for representing scene elements has some limitations. In general, the visual hull of an object does not match the object's exact geometry. In particular, it cannot represent concave surface regions. This shortcoming is often considered fatal when an accurate geometric model is the ultimate goal. In our applications, the visual hull is used largely as an imposter surface onto which textures are mapped. As such, the visual hull provides a useful model whose combination of accurate silhouettes and textures provides surprisingly effective renderings that are difficult to distinguish from a more exact model. Our system also requires accurate segmentations of each image into foreground and background elements. Methods for accomplishing such segmentations include chromakeying and image differencing. These techniques are subject to variations in cameras, lighting, and background materials.

We plan to investigate techniques for blending between textures to produce smoother transitions. Although we get impressive results using just 4 cameras, we plan to scale our system up to larger numbers of cameras. Much of the algorithm parallelizes in a straightforward manner. With k computers, we expect to achieve $O(n^2 l \log k)$ time using a binary-tree based structure.

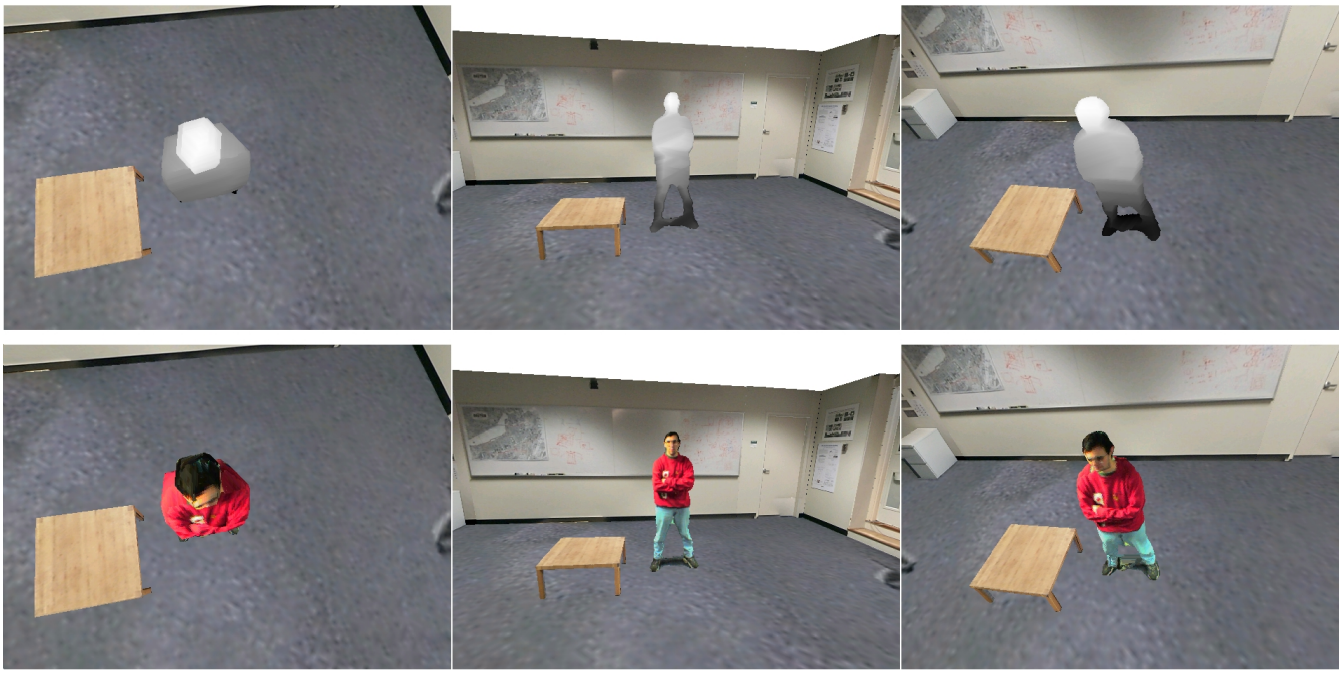

Figure 8 - Example IBVH images. The upper images show depth maps of the computed visual hulls. The lower images show shaded renderings from the same viewpoint. The hull segment connecting the two legs results from a segmentation error caused by a shadow.

7 Acknowledgements

We would like to thank Kari Anne Kjølaas, Annie Choi, Tom Buehler, and Ramy Sadek for their help with this project. We also thank DARPA and Intel for supporting this research effort. NSF Infrastructure and NSF CAREER grants provided further aid.

8 References

[1] Bichsel, M. "Segmenting Simply Connected Moving Objects in a Static Scene." *IEEE PAMI* 16, 11 (November 1994), 1138-1142.

[2] Boyer, E., and M. Berger. "3D Surface Reconstruction Using Occluding Contours." *IJCV* 22, 3 (1997), 219-233.

[3] Chen, S. E. and L. Williams. "View Interpolation for Image Synthesis." *SIGGRAPH 93*, 279-288.

[4] Chen, S. E. "Quicktime VR – An Image-Based Approach to Virtual Environment Navigation." *SIGGRAPH 95*, 29-38.

[5] Curless, B., and M. Levoy. "A Volumetric Method for Building Complex Models from Range Images." *SIGGRAPH 96*, 303-312.

[6] Debevec, P., C. Taylor, and J. Malik, "Modeling and Rendering Architecture from Photographs." *SIGGRAPH 96*, 11-20.

[7] Debevec, P.E., Y. Yu, and G. D. Borshukov, "Efficient View-Dependent Image-based Rendering with Projective Texture Mapping." *Proc. of EGRW* 1998 (June 1998).

[8] Debevec, P. *Modeling and Rendering Architecture from Photographs.* Ph.D. Thesis, University of California at Berkeley, Computer Science Division, Berkeley, CA, 1996.

[9] Faugeras, O. *Three-dimensional Computer Vision: A Geometric Viewpoint.* MIT Press, 1993.

[10] Friedman, N. and S. Russel. "Image Segmentation in Video Sequences." *Proc 13th Conference on Uncertainty in Artifical Intelligence* (1997).

[11] Goldfeather, J., J. Hultquist, and H. Fuchs. "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System." *SIGGRAPH 86*, 107-116.

[12] Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen. "The Lumigraph." *SIGGRAPH 96*, 43-54.

[13] Kanade, T., P. W. Rander, and P. J. Narayanan. "Virtualized Reality: Constructing Virtual Worlds from Real Scenes." *IEEE Multimedia* 4, 1 (March 1997), 34-47.

[14] Laurentini, A. "The Visual Hull Concept for Silhouette Based Image Understanding." *IEEE PAMI* 16,2 (1994), 150-162.

[15] Levoy, M. and P. Hanrahan. "Light Field Rendering." *SIGGRAPH 96*, 31-42.

[16] Lorensen, W.E., and H. E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." *SIGGRAPH 87*, 163-169.

[17] McMillan, L., and G. Bishop. "Plenoptic Modeling: An Image-Based Rendering System." *SIGGRAPH 95*, 39-46.

[18] McMillan, L. *An Image-Based Approach to Three-Dimensional Computer Graphics*, Ph.D. Thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 1997.

[19] Moezzi, S., D.Y. Kuramura, and R. Jain. "Reality Modeling and Visualization from Multiple Video Sequences." *IEEE CG&A* 16, 6 (November 1996), 58-63.

[20] Narayanan, P., P. Rander, and T. Kanade. "Constructing Virtual Worlds using Dense Stereo." *Proc. ICCV* 1998, 3-10.

[21] Pollard, S. and S. Hayes. "View Synthesis by Edge Transfer with Applications to the Generation of Immersive Video Objects." *Proc. of VRST*, November 1998, 91-98.

[22] Potmesil, M. "Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images." *CVGIP* 40 (1987), 1-29.

[23] Rander, P. W., P. J. Narayanan and T. Kanade, "Virtualized Reality: Constructing Time Varying Virtual Worlds from Real World Events." *Proc. IEEE Visualization* 1997, 277-552.

[24] Rappoport, A., and S. Spitz. "Interactive Boolean Operations for Conceptual Design of 3D solids." *SIGGRAPH 97*, 269-278.

[25] Roth, S. D. "Ray Casting for Modeling Solids." *Computer Graphics and Image Processing*, 18 (February 1982), 109-144.

[26] Saito, H. and T. Kanade. "Shape Reconstruction in Projective Grid Space from a Large Number of Images." *Proc. of CVPR*, (1999).

[27] Seitz, S. and C. R. Dyer. "Photorealistic Scene Reconstruction by Voxel Coloring." *Proc. of CVPR* (1997), 1067-1073.

[28] Seuss, D. "The Cat in the Hat," *CBS Television Special* (1971).

[29] Szeliski, R. "Rapid Octree Construction from Image Sequences." *CVGIP: Image Understanding* 58, 1 (July 1993), 23-32.

[30] Vedula, S., P. Rander, H. Saito, and T. Kanade. "Modeling, Combining, and Rendering Dynamic Real-World Events from Image Sequences." *Proc. 4th Intl. Conf. on Virtual Systems and Multimedia* (Nov 1998).