Creating and Rendering Image-Based Visual Hulls

Chris Buehler, Wojciech Matusik, Leonard McMillan
MIT, LCS Computer Graphics Group

Steven J. Gortler
Harvard University

Abstract

In this paper, we present efficient algorithms for creating and rendering image-based visual hulls. These algorithms are motivated by our desire to render real-time views of dynamic, real-world scenes. We first describe the visual hull, an abstract geometric entity useful for describing the volumes of objects as determined by their silhouettes. We then introduce the image-based visual hull, an efficient representation of an object's visual hull. We demonstrate two desirable properties of the image-based visual hull. First, it can be computed efficiently (i.e., in real-time) from multiple silhouette images. Second, it can be quickly rendered from novel viewpoints. These two properties motivate our use of the image-based visual hull in a real-time rendering system that we are currently developing .

Introduction

Computer graphics has long been concerned with the rendering of *static synthetic* scenes, or scenes composed of non-moving computer-created models. In time, attention turned to the rendering of *dynamic synthetic* scenes, as exemplified by virtual reality systems, most modern computer games, and the recent computer-animated movies. More recently, many researchers have embraced an image-based rendering approach in which scenes are represented by simple images that may be synthetic or acquired from the real world (say, with a digital camera). In this spirit, work has been done in rendering *static acquired* scenes, non-moving scenes acquired from real-world imagery (e.g., QuicktimeVR). However, relatively little work has been done in the case of *dynamic acquired* scenes. It is the goal of our work to develop an appropriate representation and rendering system for such scenes.

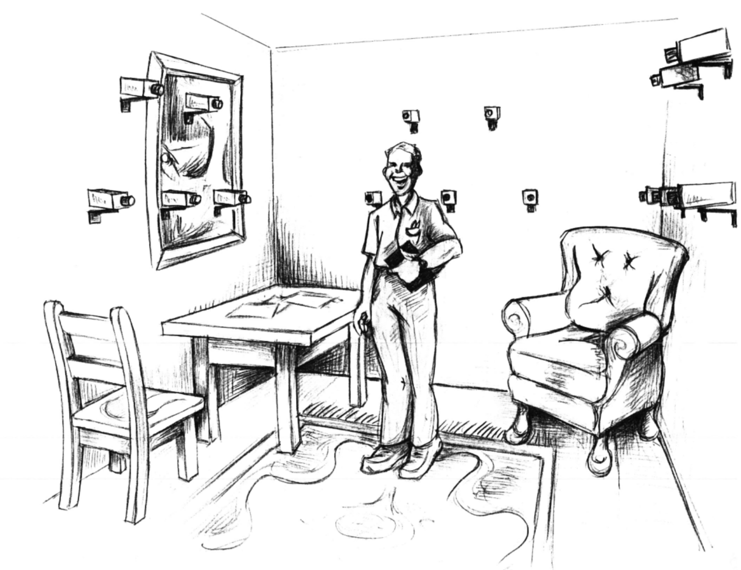

Figure 1. A hypothetical arrangement for acquiring dynamic scenes.

Using our system, a user can control a virtual camera within a moving scene that is acquired in real-time. Such a system has many potential uses. A commonly cited example is the virtual sports camera: users viewing a sporting event would be able to view the event from any angle, perhaps to focus on their favorite player or to see the action better. We are also targeting our current system at other tasks: teleconferencing and virtual sets. In a teleconferencing setting, our system would allow participants to navigate the virtual conference room or change their gaze while viewing the other participants moving in real-time. Applied to synthetic sets, our system would enable a director to see his actors perform in real-time in a dynamic three-dimensional virtual set.

A dynamic, acquired rendering system can be designed analogously to a static, acquired one. Static scenes (or objects) are typically acquired from many still photographs taken at different locations. Many photos are acquired, and often the same camera is used to take them. To extend this scenario to the dynamic case, we substitute video sequences for still photographs and place multiple, synchronized video cameras around the scene to acquire these sequences (see Figure 1). The dynamic setup is more restrictive than the static case: the number of input images is limited by the physical number of video cameras, and the cameras can only be placed in locations that do not impede the activity in the scene.

In both the static and dynamic cases, the acquired images are generally processed in some way—details vary from system to system—after which new images of the scene (or object) can be produced from arbitrary camera locations. In the dynamic case, a distinction can be made between *real-time* systems, those that process video and synthesize views at interactive rates, and *off-line* systems, those that require more extensive processing or rendering before viewing. In this paper, we are concerned with real-time systems.

There are a number of challenges inherent in real-time systems. The first is processing all the video frames at interactive rates. Obvious approaches for extracting useful information from multiple video streams, such as multi-baseline stereo algorithms, run too slow on current general-purpose hardware for a real-time system. The second challenge is rendering new views such that a virtual image exhibits as much visual fidelity as an image from one of the real cameras. For example, voxel-based systems often display noticeable artifacts in their images as a result of the low-resolution voxel data structure.

Our real-time system for rendering dynamic, acquired objects is designed to meet these challenges. We utilize between five and ten synchronized, digital video cameras to acquire continuous video streams. To achieve interactive rates, we process the video streams using efficient silhouette-based techniques to create a approximate on-the-fly models (called the *visual hull*) of the dynamic scene objects. We then create novel views of these dynamic objects using image-based rendering techniques, which are fast and preserve much of the detail of the original video sequences.

Related Work

Kanade's virtualized reality system [Kanade97] is perhaps closest in spirit to the dynamic acquired rendering system that we envision, although it is not currently a real-time system. They use a collection of cameras in conjunction with multi-baseline stereo techniques to extract models of dynamic scenes. Currently their method still requires significant off-line preprocessing time to perform the stereo correlation, but they are exploring special purpose hardware for this task, an option we wish to avoid. Recently, they have begun using silhouette methods such as the ones we use to improve the quality of their stereo reconstruction [Vedula98].

Pollard and Hayes [Pollard98] attempt to solve the problem of rendering real-time acquired data with their immersive video objects. Immersive video objects are annotated video streams that can be morphed in real-time to simulate three-dimensional camera motion. Their representation also utilizes object silhouettes, but in a different manner. They match silhouette edges across multiple views, and use these correspondences to compute a morph to a novel view. This approach has some problems, however, as silhouette edges are generally not consistent between views. These inconsistencies require their cameras to be placed close together, limiting the usefulness of the system.

Static Silhouette Methods

Silhouette contours have been used by computer vision researchers build approximate geometric models of static objects and scenes. These techniques are attractive because of the ease of extracting and working with silhouettes.

Typically these object models are computed by using silhouettes to “carve” away regions of empty space. Potmesil describes a method for computing a voxel representation of objects from sequences of silhouettes [Potmesil87]. He uses an octree data structure to represent a binary volume of space, and does not attack the problem of reconstructing novel views of his objects.

Szeliski has implemented a similar idea [Szeliski92]. He uses a turntable to rotate objects in front of a real camera. After automatically extracting object silhouettes, he computes an octree-based voxel representation of the object by projecting octree nodes into the silhouette images.

Laurentini, recognizing the interest in silhouette methods, has introduced a formalism for analyzing object reconstruction from silhouettes [Laurentini94]. Central to his theory is the concept of the visual hull, which, is the best approximation to an object’s shape that one can build from simple silhouettes. His framework is useful for understanding the limitations of silhouette methods, something that has not been quantified in earlier work.

Other volumetric carving methods, related to silhouette techniques, have also been suggested. These include volumetric reconstruction from active laser-range data [Curless96] and volumetric reconstruction based on photometric sample correspondences [Sietz97]. These techniques could be used to improve upon the approximate object models that are obtained from silhouettes. However, currently, they are not as well suited to real-time implementation.

Image-Based Rendering

Image-based rendering has been proposed as a practical alternative to the traditional modeling/rendering framework. In image based rendering, one starts with images and directly produces new images from this data. This avoids the traditional (i.e., polygonal) modeling process, and often leads to rendering algorithms whose running time is independent of the scene’s geometric and photometric complexity.

Chen’s QuicktimeVR [Chen95] is one of the first commercial static, acquired rendering systems. This system relies heavily on image-based rendering techniques to produce photo-realistic panoramic images of real scenes. Although successful, the system has some limitations: the scenes are static and the viewpoint is fixed.

McMillan’s plenoptic modeling system [McMillan95] is QuicktimeVR-like, although it does allow a translating viewpoint. The rendering engine is based on three-dimensional image warping, a now commonplace image-based rendering technique. Dynamic scenes are not supported as the panoramic input images require much more off-line processing than the simple QuicktimeVR images.

Light field methods [Gortler96, Levoy96] represent scenes as a large database of images. Processing requirements are modest making real-time implementation feasible, if not for the large number of cameras required (on the order of hundreds). The cameras must also be placed close together, resulting in a small effective navigation volume for the system.

Paper Organization

In the next section we describe the visual hull, the approximate geometric representation that we use in our system. We demonstrate how it is related to object silhouettes, and why silhouette-based analysis techniques are well suited to this sort of system. We also point out some of the problems with using the visual hull as an object approximation.

In the second section, we describe various algorithms for computing visual hulls using a image-based representation. The first algorithm is slow, but conceptually simple, while the second algorithm is faster and more sophisticated. We present advantages and disadvantages and runtime analyses.

The third section discusses various rendering algorithms for image-based visual hulls. We have investigated at least four algorithms, each with strengths and weaknesses. In this paper, we discuss three of the algorithms.

Silhouettes and the Visual Hull

Silhouette methods are well suited to real-time analysis of object shape. First, computing object silhouettes is fast and relatively robust. Second, multiple silhouettes of an object give a strong indication of that object's shape.

Computing Silhouettes

An object silhouette is essentially a binary segmentation of an image in which pixels are labeled "foreground" (belonging to the silhouette) or "background." In this paper, background pixels are typically drawn in white and foreground pixels non-white.

One common technique for computing silhouettes is chromakeying, or bluescreen matting [Smith96]. In this technique, the actual scene background is a single uniform color that is unlikely to appear in foreground objects. Foreground objects can then be segmented from the background by using color comparisons. Chromakey techniques are widely used in television weather forecasts and for cinematic special effects, which demonstrates their speed and quality. However, chromakey techniques do not admit arbitrary backgrounds, which is a severe limitation.

More general is a technique called background subtraction or image differencing [Bichsel94, Friedman97]. In background subtraction, a statistical model of a background scene is accumulated from many images. Changes in the scene, such as a figure walking into view, can then be detected by computing the difference between the new frame and the retained model. Differences that fall outside the allowed margins of the model are classified as foreground objects. There are many variations on the above two algorithms, but almost all of them are fast and robust enough to be used in a real-time system.

Shape from Silhouettes: The Visual Hull

It seems intuitive that the shape of an object can be recovered from many silhouettes. However, it is also clear that not all shapes can be recovered from silhouettes alone. For example, the concave region inside a bowl will never be evident in any silhouette, so any method based solely on silhouettes will fail to reconstruct it completely [Koenderink90].

Laurentini has introduced the concept of the visual hull for understanding the shapes of objects that can be reconstructed from their silhouettes [Laurentini94]. Loosely, the visual hull of an object is the closest approximation to that object that can be obtained from silhouettes alone.

The visual hull of an object depends both on the object itself and on a particular viewing region. A viewing region is a set of points in space from which silhouettes of an object are seen. The viewing region might be the set of all points enclosing the object, or, in a more practical case, a finite set of camera positions arranged around the object.

Formally, the visual hull of object S with respect to viewing region R, denoted $VH(S, R)$, is a volume in space such that for each point $P \in VH(S,R)$ and each viewpoint $V \in R$, the half-line from V through P contains at least one point of S [Laurentini94]. This definition simply states that the visual hull consists of all points in space whose images lie within all silhouettes viewed from the viewing region. Stated another way, the visual hull is the maximal object that has the same silhouettes as the original object, as viewed from the viewing region.

It is useful to think of an alternative, constructive definition of the visual hull with respect to a viewing region. Given a point V in the viewing region R, the silhouette of the object as seen from V defines a generalized cone in space with its apex at V (see Figure 2). The intersection of the cones from every point in R results in the visual hull with respect to R.

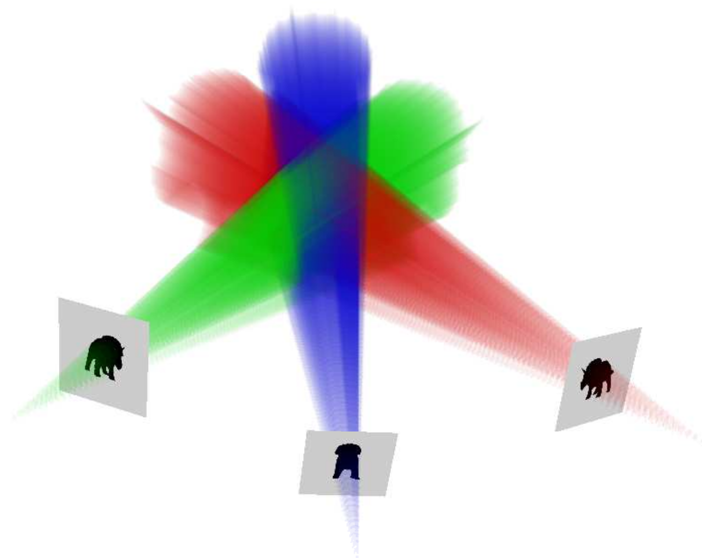

Figure 2. The intersection of the three silhouette cones defines the visual hull as seen from the viewing region. In this case, the viewing region contains only the apexes of the three silhouette cones.

This definition is useful because it implies a practical way to compute a visual hull. Almost all useful visual hull construction algorithms use some sort of volume intersection technique, as discussed in later sections.

Limitations of the Visual Hull

In the following discussion, we will assume that the viewing region for the visual hull is the set of all "reasonable" vantage points: those points outside the convex hull of the object. Using this special viewing region results in the closest possible approximation to the actual object. This viewing region is also assumed whenever reference is made to a visual hull whose viewing region is not implied by context.

The visual hull is a superset that contains the actual object's shape. It cannot represent concave surface regions (e.g., the inside of a bowl), in general, or even convex or hyperbolic points that are below the rim of a concavity (e.g., a marble inside a bowl). However, the visual hull is a tighter fit to the object than a convex hull, which only includes object regions that are globally convex. The visual hull of a convex object is the same as the object. However, the visual hull of an object composed of multiple, disjoint convex objects may not be the same as the actual objects, see Figure 3.

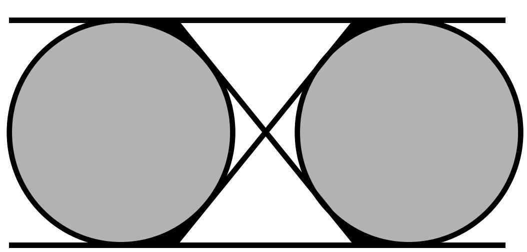

Figure 3. The visual hull of these two gray circles (black and gray regions) is slightly larger than the circles themselves. It is delimited by the bi-tangent lines drawn in the figure.

When the viewing region of the visual hull does not completely surround the object, the visual hull becomes a coarser approximation and may even be larger than the convex hull. The visual hull becomes even worse for finite viewing regions, and may exhibit undesirable artifacts such as phantom volumes (Figure 4).

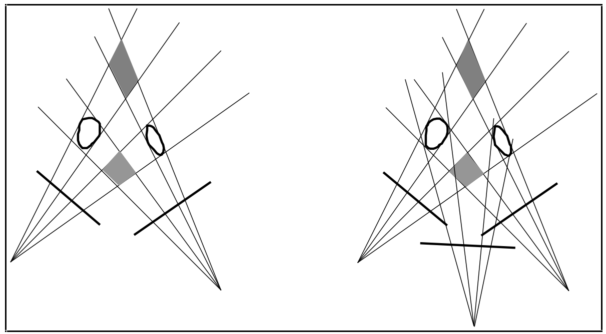

Figure 4. Intersecting the two silhouette cones results in "phantom" volumes, shown in gray on the left. A third silhouette can resolve the problem in this case (right).

In spite of these limitations, the visual hull is still a useful entity for approximating an object's shape in a dynamic rendering system. Object concavities can largely be camouflaged by object motion or hidden with surface texturing. Viewing regions that do not surround the object can be used as long as the virtual camera is confined to locations within the viewing region, as the visual hull is guaranteed to reproduce correctly all silhouettes seen from within the viewing region. Artifacts arising from using a finite viewing region (i.e., a finite amount of cameras) can be lessened by sampling a desired viewing region with appropriately placed viewpoints.

An Image-Based Visual Hull

One could attempt to compute a visual hull geometrically, but this approach, based on the intersection of multiple polytopes, is difficult to implement robustly and the resulting representation is composed of a great number of polygons if the silhouette contour is complex.

As a result, most visual hulls have been computed volumetrically by successively carving away all voxels outside of the projected silhouette cone. However, volumetric approaches suffer from problems with resolution. First, volumetric data structures are generally very memory intensive. This limitation is reduced somewhat by the fact that visual hull is a binary volume, and it is thus well suited to octree-type representations. However, it is still difficult to retain the full precision of the original silhouette images using a standard volumetric representation. If arbitrary configurations of input images are allowed then the intersection of the projected regions from them can have an arbitrarily high spatial frequency content. Thus no uniform spatial sampling is sufficient for exactly representing the final volume. Of course, reasonable approximations can be made by requiring the resulting volume to project to a silhouette contour that is within some error bound relative to the original.

In our approach, we prefer to use an image-based representation of the visual hull, which alleviates some of the problems with a standard voxel approach. In the graphics community, the term "image-based" has had many interpretations. In the strictest sense, an image-based representation consists solely of images (possibly along with matrices describing camera configurations). Along these lines, an image-based representation of the visual hull is simply the set of silhouettes themselves (along with the associated viewpoints). By definition, such a representation preserves the full resolution of the input images and contains no more or no less information than that provided by the silhouettes.

More generally, an "image-based" representation is often identified with a two-dimensional, sampled representation. For example, a standard color image is a rectangular grid of color samples, and a depth image is a grid of depth samples. Note that the samples are not considered connected in any way; they simply exist at regular intervals. The bulk of this paper is concerned with this second form of image-based visual hull.

This second type of image-based visual hull is constructed with respect to some viewpoint V in the viewing region of the visual hull. We can imagine that a camera at this viewpoint sees a silhouette image, which is discretized into a grid of pixels (i.e., samples). For each pixel in this silhouette image a list of occupancy intervals is stored. If a pixel does not belong to the silhouette (i.e., it is background), then the list is empty. Otherwise, the list contains intervals of space that are occupied by the visual hull of the object. These intervals, extruded over the solid angle subtended by the pixel, represent the region of the visual hull that projects to that pixel. The union of all such slices gives the visual hull as sampled from that viewpoint. In Figure 5, we show a slice of an image-based visual hull. The lines represent viewing rays along one column of the image, and the dark line segments denote occupied regions of space.

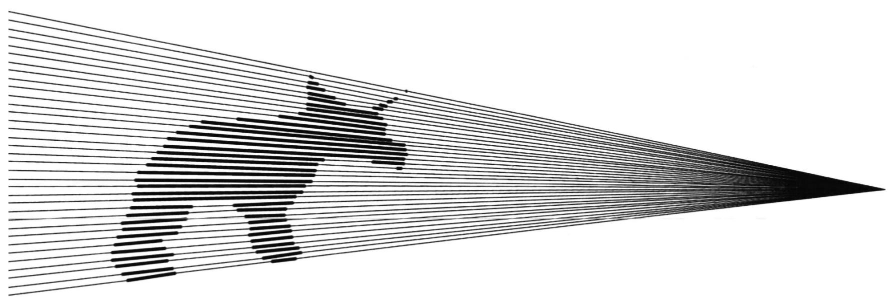

Figure 5. A single slice of an image-based visual hull. A full image-based visual hull contains many such slices, forming a volume in space.

Advantages of the Image-Based Representation

The image-based representation has a number of advantages in terms of storage requirements, computational efficiency, and ease of rendering.

The occupancy intervals can be stored as pairs of real numbers (where the numbers represent the minimum and maximum depths of the interval), similar to a run-length encoded volume. Thus, while the volume is discretized in two dimensions, the third dimension is continuous, allowing for higher resolution volumes than a voxel approach. Note also that this representation can be used for an arbitrary volume; it is not specialized for a visual hull. Similar data structures have been used by [VanHook86] and [Lacroute94] in traditional volume rendering settings.

Computing a visual hull using the image-based representation is much simpler than previous approaches.
As we will show in the next section, the three-dimensional generalized cone intersections and the volumetric carving operations of other methods are replaced with simple interval intersections in our method. These interval intersections are fast and robust, allowing for a real-time calculation of the visual hull.

Rendering the visual hull is also facilitated by the image-based representation. As we show in later sections, this representation can be rendered using only slight modifications to the standard three-dimensional image warp algorithm. This approach minimizes image resampling, as we only resample during rendering, and produces renderings of quality comparable to the input video images.

Mathematical Preliminaries

We first introduce the mathematical notation and concepts that we use in the rest of the paper. Dotted capital letters (e.g., $\dot{C}$) represent points in three-dimensional space, while lowercase over-bar letters (e.g., $\bar{x}$) represent homogeneous image (pixel) coordinates. Matrices (all are 3 x 3) are written in bold capital letters (e.g., **P**), while scalars are lowercase (e.g., t). We denote equality up to a scale factor with a dotted equals sign, $\doteq$.

One View

The basic quantity that we manipulate is a *view*, which is an image along with the viewpoint from which it was seen. We characterize a view [$\mathbf{P}$, $\dot{C}$] by a center of projection $\dot{C}$ (i.e., the viewpoint) and an inverse projection matrix $\mathbf{P}$ that transforms homogeneous image coordinates $\bar{x}$ to rays in three-dimensional world space according to the following equation:

$$\dot{X}(t) = \dot{C} + t\mathbf{P}\bar{x} \ ,$$

where $\dot{X}(t)$ represents three-dimensional world points parameterized by the distance (or range) t along a ray. Conceptually, these rays originate at $\dot{C}$ and pass through the pixel $\bar{x} = [u, v, 1]^{\mathrm{T}}$ in the imaging plane.

Often it is computationally more convenient to work with the reciprocal of the range parameter t. We call this quantity the *generalized disparity*, defined as

$$\delta = \frac{1}{t} \ .$$

Two Views

Two views [$\mathbf{P}_1$, $\dot{C}_1$] and [$\mathbf{P}_2$, $\dot{C}_2$] with different centers of projection (i.e., $\dot{C}_1 \neq \dot{C}_2$) are related by a so-called epipolar geometry. This geometry describes how a ray through a pixel in one view is seen as an *epipolar line* in the other view. Mathematically, this relationship between pixel coordinates in one view and epipolar lines in a second view is expressed by the *fundamental matrix* $\mathbf{F}_{21}$ between the two views [Faugeras93]. That is,

$$\bar{x}_2^{\,T}\mathbf{F}_{21}\bar{x}_1 = 0 \ ,$$

where the quantity $\mathbf{F}_{21}\bar{x}_1$ gives the coefficients of a line equation in the second image. Given two views [$\mathbf{P}_1$, $\dot{C}_1$] and [$\mathbf{P}_2$, $\dot{C}_2$], their fundamental matrix can be computed as

$$\mathbf{F}_{21} = \mathbf{E}_2\mathbf{P}_2^{-1}\mathbf{P}_1 \ .$$

Matrix $\mathbf{E}_2$ is a matrix representation of the cross product defined such that

$$\mathbf{E}_2 v = \bar{e}_2 \times v \ ,$$

where v is an arbitrary vector and vector $\bar{e}_2$ is the *epipole*, or the projection of the first view's center of projection onto the second view's image plane. This epipole is computed as

$$\bar{e}_2 = \mathbf{P}_2^{-1}(\dot{C}_1 - \dot{C}_2) \ ,$$

and the epipole of the first view with respect to the second is computed similarly.

Often we want to calculate a desired view from a known view. Given two views [$\mathbf{P}_1$, $\dot{C}_1$] and [$\mathbf{P}_2$, $\dot{C}_2$], where the first one is known and the second one is desired, we can transform pixels from the known view to pixels in the desired view using a three-dimensional image warping equation [McMillan96]:

$$\bar{x}_2 \doteq \mathbf{P}_2^{-1}\mathbf{P}_1\bar{x}_1 + \delta_1\mathbf{P}_2^{-1}(\dot{C}_1 - \dot{C}_2) \ . \qquad (1)$$

This equation gives pixel coordinates $\bar{x}_2$ in the desired view of the point defined by the pixel $\bar{x}_1$ and the disparity δ_1 in the first view. Thus, computing a desired view from a single known view requires auxiliary disparity information, which is often stored in the form of a depth image associated with the known view.

In computing image-based visual hulls, we are often interested in recovering the range (or disparity) parameter t_2 given corresponding image points in two views. We solve this problem by computing the range parameters of the points of closest approach of the two rays defined by the corresponding pixels in two images as follows:

$$t_1 = \frac{\det\left[\dot{C}_2 - \dot{C}_1 \quad \mathbf{P}_2\bar{x}_2 \quad \mathbf{P}_1\bar{x}_1 \times \mathbf{P}_2\bar{x}_2\right]}{\left\|\mathbf{P}_1\bar{x}_1 \times \mathbf{P}_2\bar{x}_2\right\|^2}.$$

The parameter t_2 can be computed similarly.

Three Views

It has been shown [Shashua97] that three views are related by a mathematical entity called the trilinear tensor. Similar to the fundamental matrix for two views, the trilinear tensor describes the relationship between points and lines in the three views. A complete description of the trilinear tensor is beyond the scope of this paper, however, we do present four equations derived from the tensor which relate the coordinates of a pixel $\bar{p}'' = [x'', y'', 1]$ in a third view to the coordinates of pixels in two other views ($\bar{p} = [x, y, 1]$ and $\bar{p}' = [x', y', 1]$):

$$\begin{aligned}
x''\alpha_i^{13}\bar{p}^i - x''x'\alpha_i^{33}\bar{p}^i + x'\alpha_i^{31}\bar{p}^i - \alpha_i^{11}\bar{p}^i &= 0\,, \\
y''\alpha_i^{13}\bar{p}^i - y''x'\alpha_i^{33}\bar{p}^i + x'\alpha_i^{32}\bar{p}^i - \alpha_i^{12}\bar{p}^i &= 0\,, \\
x''\alpha_i^{23}\bar{p}^i - x''y'\alpha_i^{33}\bar{p}^i + y'\alpha_i^{31}\bar{p}^i - \alpha_i^{21}\bar{p}^i &= 0, \\
y''\alpha_i^{23}\bar{p}^i - y''y'\alpha_i^{33}\bar{p}^i + y'\alpha_i^{32}\bar{p}^i - \alpha_i^{22}\bar{p}^i &= 0\,.
\end{aligned}$$

In the above equations, α_i^{jk} (i,j,k = 1,2,3) is the 27 element trilinear tensor, and the notation $\alpha_i^{nm}\bar{p}^i$ denotes a dot-product of a row of the tensor with $\bar{p}$. The elements of α_i^{jk} are obtained from the three views [$\mathbf{P}_1$, $\dot{C}_1$], [$\mathbf{P}_2$, $\dot{C}_2$], and [$\mathbf{P}_3$, $\dot{C}_3$] according to the formulas given in [Shashua97].

The important quality of these equations, with regard to image synthesis, is that the third pixel's location is completely constrained by the locations of the two other pixels; no auxiliary depth image is needed. As we will demonstrate, these equations can be exploited when rendering novel views given two or more known views.

Creating Image-Based Visual Hulls

In the following sections, we describe algorithms for computing image-based visual hulls from a finite number of silhouette images. In all of these algorithms, the input is assumed to be a set of k silhouettes (i.e., binary images), their associated viewpoints, and a viewpoint from which the visual hull is to be constructed. The algorithms output a sampled image of the visual hull, in which each pixel of the image contains a list of occupied intervals of space.

To ease algorithm analysis, the input silhouettes are assumed to be square m x m arrays of pixels. The output resolution of the image-based visual hull is n x n pixels.

The Basic Algorithm

We implement the same basic idea in all of our visual hull construction algorithms. We cast a ray into space for each pixel in the desired view of the visual hull. We intersect this ray with the k silhouette cones defined by the k silhouette views and record the intersections as pairs of enter/exit points (i.e., intervals). This process results in k lists of intervals, which are then intersected together to form a single list. This final list, representing the intersection of the viewing ray with the visual hull, is stored in our data structure.

The key aspect of all our algorithms is that all of the ray/cone intersection calculations are done in two dimensions rather than three. Recall that each silhouette cone is defined by a two-dimensional silhouette image and a center of projection. Instead of projecting these cones into three-dimensional space and then computing ray intersections, we can project the three-dimensional ray into the two-dimensional space of the silhouette image and perform intersections there. The ray simply projects to a line (in fact, the epipolar line as discussed in a previous section), and the resulting two-dimensional calculations are much more tractable.

The above observations lead directly to an algorithm for computing the image-based visual hull:

```
for each pixel p = [x,y,1] in VHULL
    initialize VHULL[x][y] = [depth_min, depth_max]

for each silhouette image SIL_i
    compute fundamental matrix F_i
    for each pixel p = [x,y,1] in VHULL
        compute epipolar line coefficients F_i p
        trace epipolar line in image SIL_i
        record list of silhouette contour intersection points [p_i,k]
        interval_list = []
        for each pair of intersection points p_i,2l and p_i,2l+1
            compute depth_i,l,min  and depth_i,l,max measured w.r.t. VHULL
            interval_list = interval_list ∪ [depth_i,l,min, depth_i,l,max]
        endfor
    VHULL[x][y] = VHULL[x][y] ∩ interval_list
    endfor
endfor
```

The algorithm is illustrated in Figure 6. Six silhouettes from a synthetic dinosaur model are shown, and the desired image-based visual hull is computed from the viewpoint of the upper left silhouette (the primary view). Three pixels are labeled in this primary view. The corresponding epipolar line for each pixel is shown in the remaining five (secondary) images. The algorithm processes one secondary image at a time. First it detects each interval where the line crosses through the silhouette of the object. At each of these silhouette contour crossings the length along the ray of the primary image is computed using the equation for the point of closest approach. A list of these intervals is computed for each secondary image. Finally, the interval lists are merged by computing their intersections across all secondary images. This process is repeated for every pixel in the primary image.

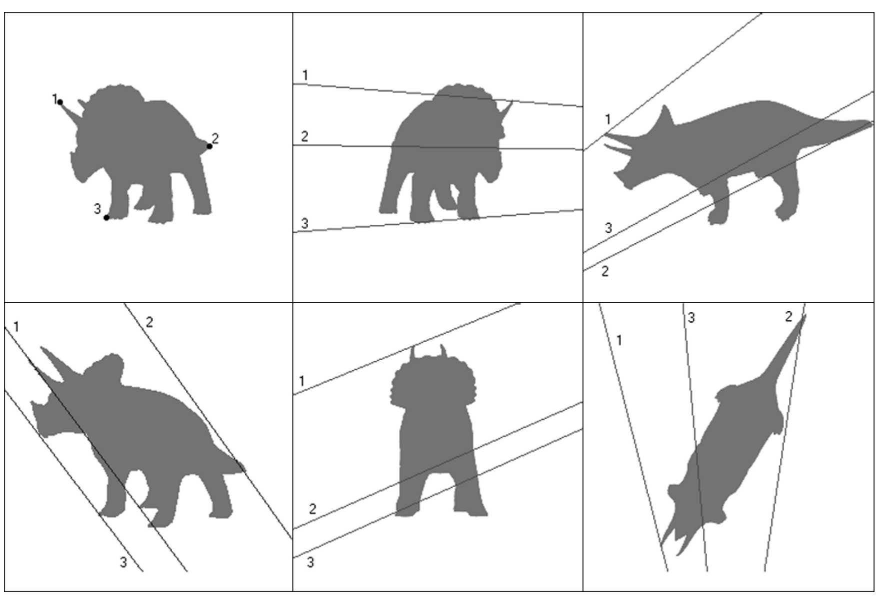

Figure 6. The image-based visual hull is computed from the viewpoint shown in the upper left. The epipolar lines corresponding to the three labeled pixels are shown in the five other silhouettes.

Analysis

The basic algorithm, while conceptually simple, is not a particularly efficient way to compute image-based visual hulls. The asymptotic running time is $O(km^2n)$, as the algorithm traces a line of length $O(n)$ in k images for each of m^2 pixels in the primary view. This analysis ignores the number of intervals traced and the cost of intersecting them. This omission is justified as there are typically far fewer intervals than the number of pixels in one dimension of a secondary image, and certainly not more than this number. When the primary and secondary images are of the same dimensions, a common case, then the running time is $O(kn^3)$. Thus, we generally consider this an n-cubed algorithm.

The algorithm also suffers from some quantization problems. The digital epipolar lines traced by the algorithm are generally not identical to the ideal epipolar lines. This discrepancy may cause the silhouette intersection points to be slightly off. In practice, such quantization problems have been largely unnoticeable.

Line-Cache Algorithm

The best running time we might expect from a visual hull construction algorithm is $O(km^2)$. This lower bound arises from the fact that we need to fill in interval lists for m^2 pixels, and we need to process k views. One might imagine a faster algorithm, based on a hierarchical decomposition (e.g., a quadtree) of the visual hull image, but here we will assume we want to create m^2 individual interval lists. A hierarchical decomposition, if desired, can then be applied to any of our algorithms.

The line-cache algorithm is an algorithm for computing the image-based visual hull that achieves the $O(km^2)$ running time. The increased efficiency is due to a simple observation: multiple three-dimensional rays from the primary image project to the same two dimensional line in the secondary images. This fact can be understood from the epipolar geometry between two views. A viewing ray from the primary image and the viewpoint of a secondary image are contained within a plane in space. This plane projects to an epipolar line in the secondary image. Any other viewing ray from the primary image which also lies in this plane projects to the same epipolar line in the secondary image.

The observation can also be demonstrated with a counting argument. It takes roughly $O(n)$ lines of length $O(n)$ to fill a discrete (pixelized) two-dimensional space of size $O(n^2)$. Thus, if we project $O(n^2)$ lines of length $O(n)$ into this space, we can expect that $O(n)$ lines will map to the same line. Of course, this argument is really only valid in a discrete setting, which is the setting in which we compute our image-based visual hulls.

Using the above observation, we amend our basic algorithm in the following way. When we attempt to compute the two dimensional line/silhouette intersection, we first check in an “epipolar line cache” data structure to see if the intersection intervals have already been computed. If so, we used the cached results. Otherwise, we compute the line intersections and store the resulting interval list in the line cache.

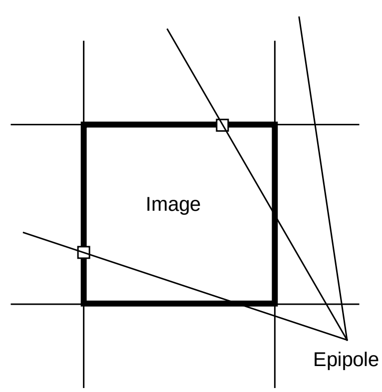

Figure 7. We determine line cache indices by the farthest intersection of the epipolar line with the image boundary. Lines that do not intersect this boundary need not be cached.

The only real issue to deal with in this algorithm is how to index the cache. That is, how do we determine that two lines are the same? There are many ways to do this; in our implementation we compute the intersection of the epipolar line with the farthest image boundary (see Figure 7). We use this intersection coordinate as the index to our cache. This indexing style allows us to vary the performance of our cache by changing the resolution of our coordinate system. For example, computing intersections to the nearest half-pixel gives a larger cache that better represents lines, but may result in fewer cache hits. Using the nearest double-pixel results in a smaller cache and more hits, but may group lines that are too dissimilar in the same cache location.

The line-cache algorithm is as follows:

```
for each pixel p = [x,y,1] in VHULL
    initialize VHULL[x][y] = [depth_min, depth_max]

for each silhouette image SIL_i
    for each cache index
        initialize CACHE_i[index] = EMPTY
    endfor
    compute fundamental matrix F_i
    for each pixel p = [x,y,1] in VHULL
        compute epipolar line coefficients F_i p
        compute line cache index = compute_index(F_i p)
        if(CACHE_i[index] = EMPTY)
            trace epipolar line in image SIL_i
            record list of silhouette contour intersection points [p_i,k]
            CACHE_i[index] = [p_i,k]
        else
            [p_i,k] = CACHE_i[index]
        endif
        interval_list = []
        for each pair of intersection points p_i,2l and p_i,2l+1
            compute depth_i,l,min  and depth_i,l,max measured w.r.t. VHULL
            interval_list = interval_list ∪ [[depth_i,l,min, depth_i,l,max]]
        endfor
    VHULL[x][y] = VHULL[x][y] ∩ interval_list
    endfor
endfor
```

Analysis

We will consider a worst case running time for the line-cache algorithm in which all cache lines are accessed. The size of each cache is O(n), and for each cache entry a line of length O(n) is traversed, leading to a total time of O(kn^2) spent computing all cache entries. The algorithm spends time O(km^2) retrieving interval lists from the caches. Thus, the runtime is O(kn^2) if $n > m$, and O(km^2) otherwise. In practice, we find that 90% of the cache entries are accessed, so this worst case analysis is applicable.

The line-cache algorithm gains its speed by making some tradeoffs in the quality of the resulting visual hull. In addition to the quantization errors from the basic algorithm, the line cache algorithm introduces errors by mapping slightly different epipolar lines to the same cache location. In practice, such errors are small, although they may be noticeable near depth discontinuity edges.

Rendering Image-Based Visual Hulls

The rendering problem is to produce a novel image of the *original object* as seen from some desired view, given an image-based visual hull of the object along with its original source views (i.e., the camera pose and images before segmentation). Since we have already shown that the visual hull is an approximation to the object's true shape, it will generally be impossible to create the exact image of the object from the new view. Thus, the goal of our rendering algorithms is to reproduce as closely as possible the true object's shape and color with information from the visual hull (shape) and the original camera images (color).

We are interested in a number of additional sub-goals for our rendering algorithms. First, they should be fast enough so that they will be applicable in our dynamic, real-time system. Second, they should offer high quality imagery in the sense that rendered images should be reasonably indistinguishable from the original camera images.

The inputs to each algorithm are assumed to be an image-based visual hull (*n* x *n* pixels), *k* original camera images (*n* x *n* pixels), and a desired view. The output is an *m* x *m* pixel image as seen from the desired view.

In all comparisons, we use the synthetic dinosaur images as inputs. The visual hull is computed from six 256 x 256 images. We generate novel renderings from three different viewpoints to exercise the strengths and weaknesses of the different algorithms. All six input dinosaur images are shown in Figure 8.

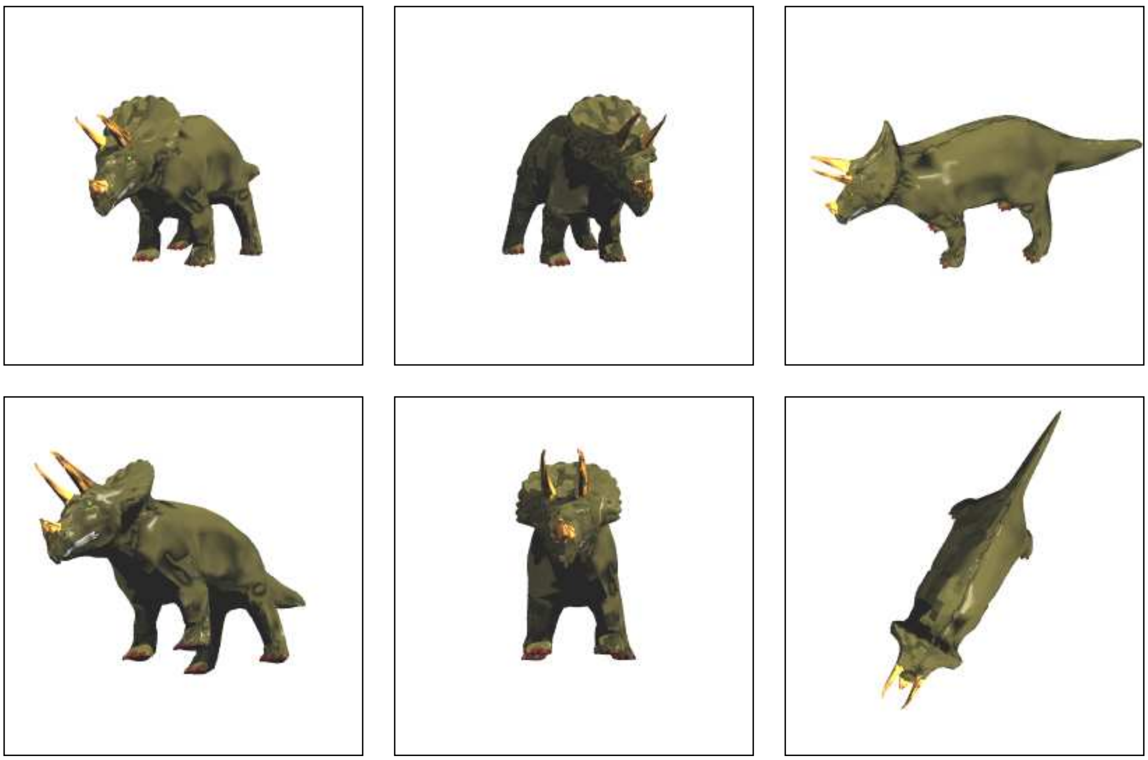

Figure 8. The six input dinosaur images (textures and silhouettes) used to create and render the image-based visual hull examples in this paper.

Texture Extrusion

The texture extrusion rendering method requires the image-based visual hull to be computed from the same viewpoint as one of the original camera images. In this special case, the pixels in the camera image are in one-to-one correspondence with the pixels in the visual hull image. In other words, each list of occupancy intervals in the visual hull image has a color assigned to it from the corresponding pixel in the camera image.

This special arrangement suggests a simple rendering technique: we can draw the occupancy intervals as seen from the new view, and we can color them with the colors assigned from the camera image. Such a rendering technique amounts to extruding the two-dimensional color image (or texture) along viewing rays to create a three-dimensional textured volume.

The basic requirement to use this technique is an ability to render a list of occupancy intervals from arbitrary viewpoints. The occupancy intervals are essentially long, thin cones in space. Calculating their projected shape exactly in the desired view would be prohibitively expensive for a real-time rendering algorithm. However, for viewpoints that are close to the viewpoint of the visual hull, the occupancy intervals can be approximated by simple line segments. Drawing these line segments can be done very quickly since it is possible calculate the end points of the line segments efficiently.

The line segment endpoints can be incrementally computed using the three-dimensional warping equation (Equation 1). Recall that the image-based visual hull data structure stores a list of disparity values $[\delta_{1,\min}, \delta_{1,\max}, \ldots, \delta_{k,\min}, \delta_{k,\max}]$ for each pixel $\bar{p} = [x, y, 1]$, much like a Layered Depth Image [Shade98]. As is done when rendering Layered Depth Images, we exploit the fact that the warping equation reduces to a simple function of disparity for a fixed pixel $\bar{p}$:

$$\bar{x}_2(\delta) \doteq \bar{a} + \delta\bar{e} \,, \qquad (2)$$

where $\bar{a} = \mathbf{P}_2^{-1}\mathbf{P}_1\bar{p}$ and $\bar{e} = \mathbf{P}_2^{-1}(\dot{C}_1 - \dot{C}_2)$, which are constant for a given $\bar{p}$.

While a Layered Depth Image only stores depth values for front-facing surfaces, we store pairs of depth values that delimit occupied regions of space. Thus, to calculate the endpoints for the line segments, we evaulate this simple expression for each disparity pair $(\delta_{\min}, \delta_{\max})$ in the occupancy interval list. Given the endpoints, we draw the line segments using a fast digital line drawing routine. The complete texture extrusion algorithm is as follows:

```
compute H = P2^-1P1
compute e = P2^-1(C1 – C2)
for each pixel p = [x,y,1] in VHULL
    compute a = Hp
    for each interval [dl,min, dl,max] in VHULL[x][y]
        compute line segment endpoints [xl,min, yl,min] = a + dl,mine
        and [xl,max, yl,max] = a + dl,maxe using the incremental
        three-dimensional warp equation (Equation 2)
        draw_line(xl,min, yl,min, xl,max, yl,max, VHULL[x][y].color)
    endfor
endfor
```

Analysis

The texture extrusion algorithm runs in time complexity O(n^2m), as it draws a line of length O(m) for each of n^2 interval lists in the visual hull data structure. Although this may not seem fast, in practice it is fast enough for real-time rendering (~ 20 frames/sec). Texture extrusion also produces reasonably good looking images for viewpoints close to the viewpoint of the visual hull. Figure 9a demonstrates a novel viewpoint close to the original one. The visual hull in this case was computed from the viewpoint of the upper left-hand image in Figure 8.

Texture extrusion fails, however, when the desired viewpoint is far from the viewpoint at which the visual hull was sampled. This failure is primarily due to two factors. First, when the viewpoint is moved too far to one side, the extruded colors no longer approximate the true color of the object (see Figure 9b). This problem is unavoidable, as a single camera image can not see the entire object at one time. Second, when the viewpoint is moved very close to the object, the approximation of drawing line segments for the occupancy intervals is no longer valid and the images "explode" (see Figure 9c).

(a) (b) (c)

Figure 9. Images rendered from three novel viewpoints using texture extrusion.

Texture Projection

The texture projection algorithm extends the texture extrusion algorithm to handle a wider range of viewpoints. It corrects the second viewpoint problem, that of incorrect colors for distant viewpoints, by combining colors from multiple textures into a single rendering.

Texture projection is a simple extension to the texture extrusion algorithm. In texture extrusion, a single texture is essentially projected through the volume of the visual hull. Regions of the visual hull that are seen from the texture's viewpoint are colored correctly, while other regions are colored incorrectly. In texture projection, we project multiple textures onto the surface of the visual hull. Regions of the visual hull that are not seen by one texture can be colored with information from another texture.

We implement texture projection by a small modification to the texture extrusion algorithm. Instead of drawing each line segment with a constant color, we projectively texture map the line segment with colors from another texture. The projective texture mapping is done using the trilinear tensor equations. The tensor between the three views—the visual hull's view, the texture's view, and the desired view—allow us to compute texture coordinates in the texture's view given coordinates in the visual hull's view and the desired view. Pseudocode for the algorithm is give below. In the pseudocode [$\mathbf{P}_1$, $\dot{C}_1$] refers to the visual hull's view, [$\mathbf{P}_2$, $\dot{C}_2$] denotes the desired view, and [$\mathbf{P}_k$, $\dot{C}_k$] is one of the texture views.

```
compute H = P2^-1 P1
compute e = P2^-1 (C1 – C2)
for each pixel p = [x,y,1] in VHULL
    compute a = Hp
    for each interval [dl,min, dl,max] in VHULL[x][y]
        compute line segment endpoints [xl,min, yl,min] = a + dl,min e
            and [xl,max, yl,max] = a + dl,max e using the incremental
            three-dimensional warp equation (Equation 1)
        k = select_texture(x, y, l)
        draw_line_proj_tex(x, y, xl,min, yl,min, xl,max, yl,max, P1, C1, P2, C2, Pk, Ck)
    endfor
endfor
```

The auxiliary function `draw_line_proj_tex` implements projective texture mapping using the trilinear tensor computed from [$\mathbf{P}_1$, $\dot{C}_1$], [$\mathbf{P}_2$, $\dot{C}_2$], and [$\mathbf{P}_k$, $\dot{C}_k$]. The function `select_texture` selects the texture to be mapped to the indicated visual hull interval. Many mappings are possible; we implemented a particularly simple strategy in our real-time implementation. We choose the texture with the minimum angle between the visual hull interval and the texture's viewpoint.

Analysis

The texture projection algorithm has the same asymptotic running time as the texture extrusion algorithm, $O(n^2m)$. However, because of the cost of the texture mapping, the hidden constant is much larger, which makes the algorithm slower in practice. The quality of the images is generally better, and the algorithm is useful for larger changes in the viewpoint (see Figures 10a and 10b). However, texture projection does suffer from the same zooming problem as the texture extrusion algorithm (see Figure 10c).

(a) (b) (c)

Figure 10. Images rendered from three novel viewpoints using texture projection.

Ray-Casting

Both the texture extrusion and the texture projection algorithms suffer from the same problem with viewpoints that are too close to the object: the image tends to break apart. This problem is directly related to the fact that both algorithms are *forward mapped*. They transform points from the visual hull to pixels in the desired view, and they may miss pixels along the way. Similar problems exist in other areas of computer graphics, and they are typically solved by using a *backward mapped* algorithm. In such an algorithm, pixels in the desired view are transformed to points in the visual hull. In this manner, every pixel in the desired view can be mapped to some point in the visual hull and colored appropriately.

To implement a backward mapped algorithm for rendering visual hulls, we would like to know for every pixel in the desired view whether or not the ray through that pixel intersects the visual hull. To compute this, we can cast a ray for every pixel in the desired view and test it for intersections with the k silhouette cones from the k cameras. Or, in other words, we can compute an image-based visual hull from the desired viewpoint.

An image-based visual hull computed from the desired viewpoint effectively gives the *shape* of the visual hull in the form of a depth image. However, we would like to have the proper colors along the with shape. We can compute the colors using a bit of additional computation to back project the visual hull to the k camera images and sample the colors. The complete algorithm is as follows:

```
compute VHULL_d from view [P_d, C_d]

for each pixel p = [x, y, 1] in VHULL_d
    extract depth_min from VHULL_d[x][y]
    for each camera image CAM_k
        backproject p to p_k = [x_k, y_k, 1] using Equation 1
        color_k = CAM_k[x_k][y_k]
    endfor
    VHULL_d[x][y] = weighted_avg(color_k)
endfor
```

The function `weighted_avg` simply computes some weighted average of the colors sampled from the k camera images. A color weight may be 0 if the camera makes no contribution to the color (e.g., it is occluded) or 1 if the camera contributes all the color (e.g., a winner-take-all strategy). In some cases, calculating the weights may be non-trivial. We use the winner-take-all approach in our implementation. That is, we assign a "best" camera a weight of 1 and assign all other cameras 0 weights. We define the best camera as the camera whose viewing ray is closest to that of the viewing direction. This strategy for assigning camera weights ignores the occlusion problem, and cameras may be selected which actually do not see the pixel to be colored.

Analysis

Due to its backward mapped nature, the ray-casting algorithm has a complexity fundamentally different than the previous two rendering algorithms. The running time is $O(km^2)$, as the visual hull calculation is $O(km^2)$, and the pixel coloring loop backprojects each of m^2 pixels k times. This running time is noteworthy as it is proportional to the size of the desired image and independent of the size of the camera images (for $m > n$). For $m = n$, the algorithm is n-squared, which compares favorably to the n-cubed forward mapped algorithms. However, the hidden constant is large, so this advantage is not realized at typical values of n.

This algorithm is slower than the forward mapped algorithms, but potentially produces images of higher quality (image quality and speed depend on the choice of color weighting). However, since the runtime of this algorithm includes the explicit visual hull calculation, the comparison is slightly unfair. Also, because it is backward mapped, problems with close range viewpoints are avoided. Ray-casting results are shown in Figures 11a, 11b, and 11c.

(a) (b) (c)

Figure 11. Images rendered from three novel viewpoints using ray-casting.

Conclusion

We have introduced the image-based visual hull as an approximate object representation for real-time dynamic acquired rendering systems. The needs of these systems require algorithms that allow for both the analysis of video inputs and the synthesis of rendered outputs to occur in real-time. Our algorithms for creating and rendering image-based visual hulls satisfy these requirements.

We have shown that the visual hull is a reasonable object representation to use in terms of accuracy and robustness. It provides a reasonable approximation to object shape in most cases, and requires only simple silhouette segmentation for acquisition.

We have demonstrated an efficient real-time algorithm for creating visual hulls. First, we exploit epipolar geometry to reduce three-dimensional volume intersections to simpler two-dimensional line intersections. Then, we use a line-caching approach to reuse previously computed results giving a further increase in performance.

Finally, we have presented a number of algorithms for rendering views of image-based visual hulls from novel viewpoints. The texture extrusion algorithm is fast but does not make use of all available color information. The texture projection algorithm, while slower, does utilize color information from all possible cameras. Both algorithms, however, suffer from a problem with viewpoints that are too close to the object. This problem is remedied by the ray-casting algorithm, which generates an image directly from the visual hull calculation.

Acknowledgements

Support for this research was provided by DARPA contract N30602-97-1-0283, and the Massachusetts Institute of Technology's Laboratory for Computer Science. We would also like to thank Anne McCarthy for providing artwork.

References

[Bichsel94] Bichsel, M., *"Segmenting Simply Connected Moving Objects in a Static Scene,"* **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. 16, No. 11, November 1994, pp. 1138-1142.

[Chen95] Chen, S.E., "*QuickTime VR - An Image-Based Approach to Virtual Environment Navigation*," **Computer Graphics** (SIGGRAPH '95 Conference Proceedings), August 6-11, 1995, pp. 29-38.

[Curless96] Curless, B., and M. Levoy. *"A Volumetric Method for Building Complex Models from Range Images,"* **Computer Graphics** (SIGGRAPH '96 Conference Proceedings), August 4-9, 1996, pp. 43-54.

[Faugeras93] Faugeras, O., **Three-dimensional Computer Vision: A Geometric Viewpoint**, The MIT Press, Cambridge, Massachusetts, 1993.

[Friedman97] Friedman, N., and Russell, S., *"Image Segmentation in Video Sequences,"* **Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence**, 1997.

[Gortler96] Gortler, S.J., Grzeszczuk, R., Szeliski, R, and Cohen, M.F., "*The Lumigraph*," **Computer Graphics** (SIGGRAPH'96 Conference Proceedings), August 4-9, 1996, pp. 43-54.

[Kanade97] Kanade, T., Rander, P. W., Marayanan, P. J., *"Virtualized Reality: Constructing Virtual Worlds from Real Scenes,"* **IEEE MultiMedia**, Vol.4, No.1, Jan. - Mar. 1997, pp.34-47.

[Koenderink90] Koenderink, J. J., **Solid Shape**, The MIT Press, Cambridge, Massachusetts, 1990.

[Lacroute94] Lacroute, P., Levoy, M., *"Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation,"* **Computer Graphics** (SIGGRAPH '94 Conference Proceedings), July 24-29, 1994, pp. 451-458.

[Laurentini94] Laurentini, A., *"The Visual Hull Concept for Silhouette-Based Image Understanding,"* **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. 16, No. 2, February 1994, pp. 150-162.

[Levoy96] Levoy, M. and P. Hanrahan, "*Light Field Rendering*," **Computer Graphics** (SIGGRAPH'96 Conference Proceedings), August 4-9, 1996, pp. 31-42.

[McMillan95] McMillan, L., and Bishop, G., "*Plenoptic Modeling: An Image-Based Rendering System*," **Computer Graphics** (SIGGRAPH '95 Conference Proceedings), August 6-11, 1995, pp. 39-46.

[McMillan96] McMillan, L., *"An Image-Based Approach to Three-Dimensional Computer Graphics,"* Ph.D. Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1996.

[Pollard98] Pollard, S., and Hayes, S.,*"View Synthesis by Edge Transfer with Applications to the Generation of Immersive Video Objects,"* **Proceedings of the ACM Symposium on Virtual Reality Software and Technology**, November 2-5, 1998, pp. 91-98.

[Potmesil87] Potmesil, M., *"Generating Octree Models of 3D Objects from Their Silhouettes in a Sequence of Images,"* **Computer Vision, Graphics, and Image Processing**, Vol. 40, 1987, pp. 1-29.

[Seitz97] Seitz, S. M., Dyer, C. R., *"Photorealistic Scene Reconstruction by Voxel Coloring,"* **Computer Vision and Pattern Recognition Conference**, 1997, pp. 1067-1073.

[Shade98] Shade, J., Gortler, S., He, L., and Szeliski, R., *"Layered Depth Images,"* **Computer Graphics** (SIGGRAPH '98) Conference Proceedings), July 19-24, 1998, pp. 231-242.

[Shashua97] Shashua, A., *"Trilinear Tensor: The Fundamental Construct of Multiple-view Geometry and its Applications,"* **International Workshop on Algebraic Frames For The Perception Action Cycle (AFPAC)**, Kiel Germany Sep. 8-9, 1997.

[Smith96] Smith, A. R., and Blinn, J. F., *"Blue Screen Matting,"* **Computer Graphics** (SIGGRAPH '96 Conference Proceedings), August 4-9, 1996, pp. 21-30.

[Szeliski92] Szeliski, R., *"Rapid Octree Construction from Image Sequences,"* **CVGIP: Image Understanding**, Vol. 58, No. 1, July 1993, pp. 23-32.

[VanHook86] Van Hook, T., *"Real-time shaded NC milling display,"* **Computer Graphics** (SIGGRAPH '86 Conference Proceedings), 1986, pp. 15-20.

[Vedula98] Vedula, S., Rander, P., Saito, H., Kanade, T., *"Modeling, Combining, and Rendering Dynamic Real-World Events from Image Sequences,"* **4th International Conference on Virtual Systems and Multimedia conference proceedings**, Nov. 1998.