Machine Vision and Applications manuscript No.
(will be inserted by the editor)

Real-time Shape-from-silhouette

Dan E. Small, Lance R. Williams

Department of Computer Science, University of New Mexico, Albuquerque, NM 87110, USA

15 June 2002

Abstract The computer vision field has undergone a revolution of sorts in the past five years. Moore's law has driven real-time image processing from the domain of dedicated, expensive hardware, to the domain of commercial off-the-shelf computers. This paper describes our work on the design, analysis and implementation of a Real-Time Shape from Silhouette Sensor (RTS^3). The system produces time-varying volumetric data at real-time rates (10–30Hz). The data is in the form of binary volumetric images. Until recently, using this technique in a real-time system was impractical due to the computational burden. In this paper we review the previous work in the field, and derive the mathematics behind volumetric calibration, silhouette extraction, and shape-from-silhouette. For our sensor implementation, we use four color camera/framegrabber pairs and a single high-end Pentium III computer. The color cameras were configured to observe a common volume. This hardware uses our RTS^3 software to track volumetric motion. Two types of shape-from-silhouette algorithms were implemented and their relative performance was compared. Lastly, an application of our sensor to the problem of generating synthetic views was explored.

Key words: Shape-from-silhouette, volumetric reconstruction, multiple cameras, voxel.

1 Introduction

Applications such as virtual reality, telepresence, smart rooms, human robot interaction, automatic surveillance, gesture analysis, movement analysis for sports (and many others) require real-time motion-tracking. Because of this enormous number of applications requiring human computer interaction, real-time 3D motion-tracking is an important problem. Much work has already been done in this field, but has used systems where images are acquired in real-time and processed later [3,11]. However, offline processing does not permit real-time interaction. Off-line system provide greater accuracy, but we are willing to sacrifice accuracy in order to achieve real-time performance.

There are a wide range of real-time motion tracking sensors currently in use. The most common ones require the object or person being tracked to be instrumented in some fashion, either through magnetic field sensors [26], wired optical emmiters [31] or passive optical targets [30]. As 3D user interfaces become commonplace, it becomes imperative to eliminate encumbering wires, markers or sensors. We propose a technique that, until quite recently, had computational and memory requirements that exceeded the capabilities of commonly available computing systems. Our approach is called "Real-Time Shape-from-Silhouette."

1.1 Shape-from-silhouette Techniques

There has been much work in the area of shape-from-silhouette. One of the goals of this paper is to thoroughly investigate all aspects of using this method for real-time data acquisition. The implementation of a Real-Time Shape- from Silhouette Sensor (RTS^3) is contingent upon three main concepts:

- camera calibration,
- silhouette extraction
- silhouette projection and intersection.

The problem of mapping 3D points in the world to a pixel in the image plane of the camera is addressed by camera calibration. There are two sets of parameters that must be estimated. The first are the intrinsic parameters, which model the optics of the camera. The second set of parameters are the extrinsic parameters, which model the position and orientation of the camera in three-space, relative to an external coordinate system. To accomplish the multi-camera implementation of a shape-from-silhouette sensor, all cameras must be calibrated to a common coordinate system.

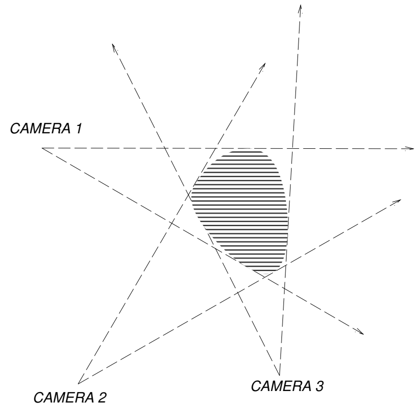

Fig. 1 The intersection of three views (in a 2D plane) results in a reconstruction that includes 1) the original object; 2) points in concavities that are not visible from any viewpoint outside the convex hull of the object; and 3) points that are not visible from at least one of the three given views. Variations in the estimated geometry can occur when approximating the shape with a visual hull. In general, the visual hull is a super-set of the actual geometry, and is not equivalent to the true shape, due to the inability of the shape-from-silhouette technique to resolve concavities.

Silhouette extraction is the process of identifying, in each image, those pixels which have changed with respect to a background image. This process is also called *background subtraction* or *background differencing*. We will describe commonly used methods, as well as some novel techniques.

Silhouette projection and intersection uses camera models to project the extracted silhouettes into a shared volumetric space, and compute their mutual intersection. Figure 1 shows the problems inherent in approximating a shape by its visual hull. The visual hull is generally a superset of the actual shape due to the inability of the shape-from-silhouette technique to resolve concavities. Silhouette projection and intersection can be accomplished using a variety of techniques. We will review some of these techniques and will present an execution time analysis of two algorithms which we implemented and tested.

1.2 Why Do This in Real-Time?

There are many reasons to pursue a real-time implementation of shape-from-silhouette. First, with the advent of ubiquitous computing, new user interfaces will be needed. Human gesture is a natural interface that can be exploited to allow people to interact with these ubiquitous computer systems. However, human gesture is inherently three dimensional. Therefore, we need a system that is capable of measuring gestural data in three dimensions. A second reason for creating a system that works in real-time is the commercial demand for markerless motion tracking. Such a system would represent a substantial savings in time and money and be more convenient for applications that currently use commercially available tracking systems (e.g. Ascension Technologies' "Flock of Birds" [31], the Polhemus Fastrak [29], or Northern Digital's Optotrak sensor [28]). These areas include virtual reality, human and animal motion tracking for special effects work, and innovative video game interfaces.

To the best of our knowledge, only one other group has attempted to implement a shape-from-silhouette sensor that functions in real-time. Using five cameras and six computers, Cheung *et al.*[6] achieved a frame-rate of about 15 Hz for a voxel volume of size $64 \times 64 \times 64$. In this paper, we describe a shape-from-silhouette sensor which is significantly faster, yet runs on a single (albeit faster) computer.[1] Using four cameras, we have achieved a frame-rate of 23 Hz for a voxel volume of size comparable to that of Cheung *et al.*[6], and comparable frame rates (15 Hz) for a volume three times as large.

The remainder of this paper is organized as follows: In Section 2 we provide an overview of past work in the area of shape-from-silhouette. Technical details and early work in this area will also be presented. Section 3 describes the implementation and architecture of RTS^3, and presents results from experiments in silhouette extraction. Finally, Section 4 gives a run-time analysis of two algorithms that were implemented on our system, and a brief overview of a novel application.

2 History and Background

The background and history of shape-from-silhouette has its roots in three separate areas: camera calibration, background subtraction (silhouette extraction) and volume intersection.

2.1 Camera Calibration

Camera calibration has been well studied in computer vision [1,10,13,24,22,23,25]. Because we want to cover as large a volume as possible with the fewest number of cameras, the RTS^3 system uses cameras with extremely wide angle lenses. However, the 180° field of view of fish-eye lenses is too large because key features are projected onto a very small number of pixels. Typically, lenses in the systems we have reviewed have a field of view anywhere between 60° and 90°. This requires a calibration technique that includes estimating the coefficients of radial distortion. Tsai [22] showed how to analytically determine radial distortion parameters for machine vision systems. If a camera with a narrow field of view is used it may not be necessary to model radial distortion, as the camera will be well approximated by a pinhole. The

[1] Cheung *et al.* used six computers with seven Intel Pentium II processors clocked at 400 MHz while we used a single computer with an Intel Pentium III processor clocked at 866 MHz.

pinhole model is commonly used in systems that do not use wide angle lenses.

Ganpathy [10] explores calibration for cameras on a robot. However, this method is limited because there is no way to achieve the matrix decomposition of the calibration if radial distortion is considered. This is due to the nonlinearities inherent in the modeling of the radial distortion. It is also desirable to use a calibration technique that relies on planar calibration targets rather than 3D calibration targets. This allows for simple calibration rigs (paper targets on glass or hard wood) that are easier to set up than 3D calibration rigs. Such methods were found first in Wei *et al.* [23] and further developed in Zhang's work on EasyCalib [25]. We used the camera calibration functions implemented in the OpenCV computer vision library [2], which are based on EasyCalib. Zhang demonstrated that the EasyCalib algorithm was robust and easy to use. As a result, the implementation used in the OpenCV library is also robust. A calibration filter application implemented in the library uses feature extraction algorithms to locate the black/white intersections of a checkerboard in real-time, which speeds the process of collecting the data used in calibration.

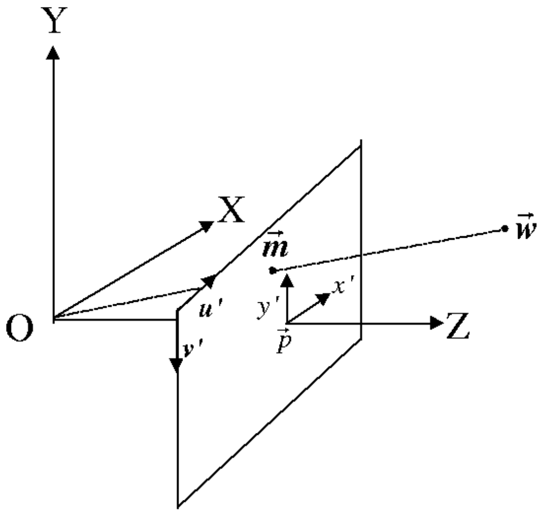

Fig. 2 Projection of 3D point $\mathbf{w}=[X, Y, Z]^T$ onto image plane at $\mathbf{m}=[x, y]^T$.

Recovering three dimensional structure from images is very difficult without calibrated images. For our purposes, a camera is said to be calibrated if the mapping between image coordinates and directions relative to the camera center are known. However, the position and orientation of the camera in space is not necessarily known. For an ideal pinhole camera delivering a true perspective image, this mapping can be characterized completely by just four numbers, called the *intrinsic parameters* of the camera. In contrast, a camera's *extrinsic* parameters represent its location and rotation in space relative to the world coordinate system. We can represent these two sets of parameters with three coordinate systems:

- image coordinate system
- camera coordinate system
- world coordinate system.

The camera coordinate system can be represented by a perspective projection, which can be written as a linear mapping between two points in projective coordinates:

$$\begin{bmatrix} x \\ y \\ s \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & -f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \qquad \text{where } x' = \frac{x}{s}, \quad y' = \frac{y}{s} \tag{1}$$

and f is the *focal length.* This 3×4 projection matrix represents a map from 3D to 2D in projective coordinates.

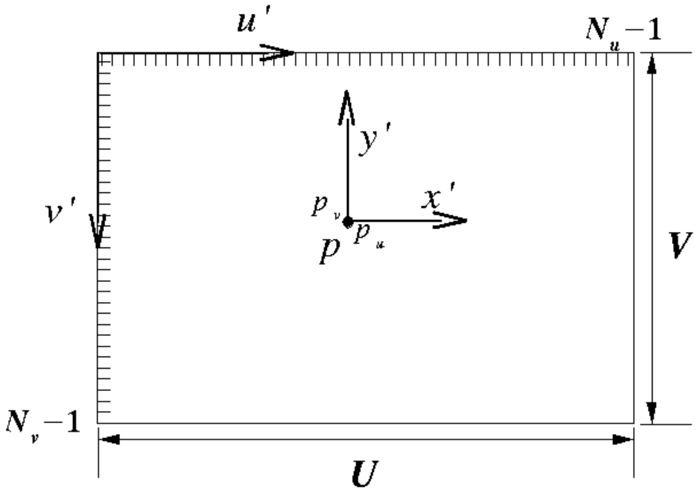

Fig. 3 p is the point where the optical axis intersects the camera plane, and is often referred to as the *principal point.* This is the origin of the projective coordinate system. The upper left corner is the origin of the image coordinate system

As shown in Figure 3, the two coordinate systems, (x', y') and (u', v'), are related by a translation and a reflection in one axis. We need to translate and invert the camera coordinate system, (x', y') so that the origin of the image coordinate system (u', v') is transformed to the upper left corner. The image coordinate system needs to be scaled such that maximum values in the u and v axes correspond to the maximum pixel coordinates that can be accessed in the video framegrabber. To accomplish this we modify the parameters of the matrix in Equation 1. To derive a new offset and scale for the image coordinate system, we need to define the following terms:

N_u - number of pixels along the u axis

N_v - number of pixels along the v axis
U - physical width of the CCD sensor element
V - physical height of the CCD sensor element.

These parameters can be combined into two parameters, (k_u, k_v), which are scale factors which take into account the size and resolution of the CCD:

$$(k_u, k_v) \;=\; \left(\frac{N_u}{U}, \; \frac{N_v}{V}\right).$$

Using these scale factors, we can relate the origin of the camera coordinate system, $[x_0, y_0]^T$, to the image coordinate system, $[u_0, v_0]^T$, using the following relationship:

$$(u', v') \;=\; (k_u x' + p_u, \quad k_v y' + p_v) \tag{2}$$

The optical axis intersects the camera plane at p. The point (p_u, p_v) is the coordinate location of p in image coordinates, and is often referred to as the *principal point.* We now have enough information to define the projection in terms of the image plane with an origin at $[u, v]^T$ and a maximum coordinate value of (N_u, N_v). We define four parameters to model the projection:

(p_u, p_v) - center of proj. in image coord. w/origin (u_0, v_0)
fk_u - scaled focal length for u in pixels
fk_v - scaled focal length for v in pixels.

Once these four parameters (i.e the *intrinsic calibration parameters*) are known, the camera is said to be *calibrated.* A calibrated camera can map pixels in the image to rays in three dimensional space and three dimensional points to pixels in the image. From these values we can form the intrinsic projection matrix, $\mathbf{A}$, which is often called the *camera calibration matrix*:

$$\mathbf{A} = \begin{bmatrix} fk_u & 0 & p_u & 0 \\ 0 & -fk_v & p_v & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3}$$

The last coordinate transformation relates the camera and world coordinate systems. The extrinsic parameters can be represented by a rotation matrix and a translation vector which can be combined into a homogeneous 4×4 transform,

$$\mathbf{T} = \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4}$$

We can now represent the projection of an arbitrary three dimensional point in the world coordinates, $\mathbf{w}$, to a two dimensional point in camera coordinates, $\mathbf{m}$, using the following equation:

$$\underbrace{\begin{bmatrix} u \\ v \\ s \end{bmatrix}}_{\mathbf{m}} = \underbrace{\begin{bmatrix} fk_u & 0 & p_u & 0 \\ 0 & -fk_v & p_v & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\mathbf{w}} \tag{5}$$

where $(u', v') = (u/s, \; v/s)$. To derive the pixel coordinates on the image plane, u and v must be scaled by dividing by s, the third element of the vector $\mathbf{m}$. This is a scale factor which is derived from the distance to the point, $\mathbf{w}$, in camera coordinates.

The calibration of real cameras is often approximated with this four-parameter mapping, though rarely is such an approximation accurate to within one pixel [25]. For most images taken with standard lenses, the center of projection is at (or near) the coordinate center of the image. However, small but significant shifts are often introduced in the digitizing process. Such an image shift has the greatest impact on camera calibration for lenses with shorter focal lengths (such as the ones we are using in our system). The focal lengths of the camera in pixels can be estimated by dividing the marked focal length of the camera lens by the width of the image on the imaging surface (the film or CCD array), and then multiplying by the width and height of the final image in pixels. This scaling is modeled in Equation 2.

Real cameras deviate from the pinhole model in several respects. The most significant effect is radial distortion caused by the lens. Because of constraints in the lens manufacturing process, straight lines in the world imaged through real lenses generally become somewhat curved on the image plane. However, since each lens element is radially symmetric, and the elements are typically centered with high precision on the optical axis, this distortion is almost always radially symmetric. As a result, the center of radial distortion also corresponds to the intersection of the center of optical axis with the image plane. This point is the principal point, (p_u, p_v).

We can model the distortion with a nonlinear geometric transform. Let $(\hat{u}, \hat{v})$ be the true position of pixel, (u, v), in the distorted image. Let (p_u, p_v) be the center of the radial distortion. Zhang [25] demonstrated that representing the radial distortion using a second degree polynomial is good enough for most purposes and that using a higher order approximation actually causes numerical instability. Therefore, we use a second order approximation here. We can represent the geometric transform between the distorted and undistorted image spaces as follows:

$$\hat{u} = u + (u - p_u)\left[k_1(u^2 + v^2) + k_2(u^2 + v^2)^2\right] \tag{6}$$
$$\hat{v} = v + (v - p_v)\left[k_1(u^2 + v^2) + k_2(u^2 + v^2)^2\right]. \tag{7}$$

The coefficients k_1 and k_2 can be determined by measuring the curvature of straight lines in sample images. Because our system uses very wide angle lenses, we have found intrinsic camera calibration and estimation of radial distortion parameters to be a straightforward process that considerably simplifies the problem of three dimensional reconstruction. By using Equations 6 and 7 in conjunction with Equation 5, we can get a pinhole projection model that is corrected for radial distortion.

The calibration method presented by Zhang[25] estimates the intrinsic matrix and radial distortion parameters by using a series of checkerboard patterns as shown in Figure 4. The checkerboard represents a three dimensional rigid body that lies on the plane, $Z = 0$. The interior corners of the checkerboard are located to sub pixel accuracy in a series of images. Given the dimensions and the size of the checkerboard, the intrinsic parameters are solved for using an iterative non-linear least squares method that successively estimates the intrinsic matrix and the relative position and orientation of the checkerboards with respect to the camera. During this process the extrinsic parameters of the camera are fixed at zero.

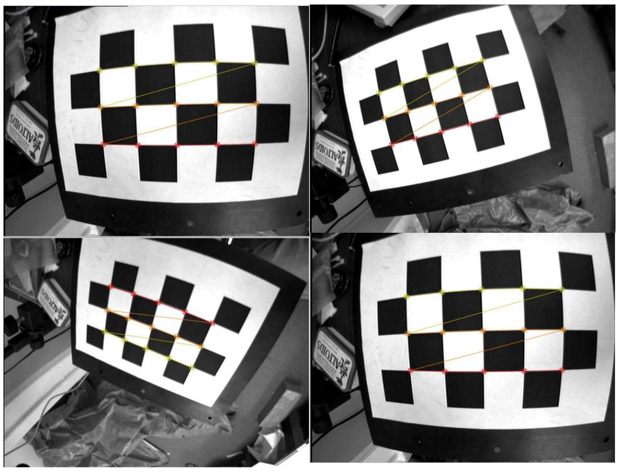

Fig. 4 This figure shows the data that is used to estimate the intrinsic parameters. These are four consecutive images of the checkerboard calibration object. The intersections are automatically located to sub-pixel accuracy. This is the output from the calibration filter, which is implemented in the Open Source Computer Vision library[2].

Once the intrinsic parameters have been determined, the same process is used to solve for the extrinsic parameters. The intrinsic parameters are fixed at their best estimates, and the extrinsic parameters are solved for by observing a common set of planar targets and associating each 2D point with its 3D world coordinate. The number of points in the planar calibration target must exceed the number of parameters that are being solved for. In our case this is six: $X, Y, Z, \Omega_x, \Omega_y$ and Ω_z. This yields a set of linear equations where the number of equations is greater than the number of unknowns. The targets we used are the square black and white 2×2 checkerboards

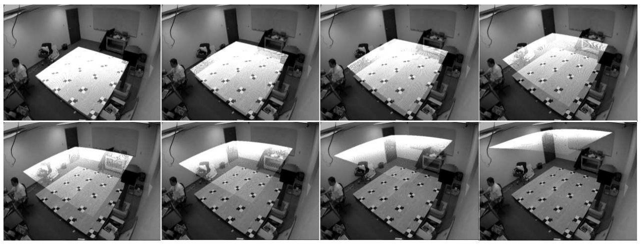

Fig. 5 This series of images shows a set of points in the xy plane (separated at a $\frac{1}{4}$ inch resolution) which are back-projected into the image at increasing heights through the volume. You can see the radial distortion present in the images, as the grid appears to bow heavily at the top of the volume.

on the wooden platform in Figure 5. This series of images shows a set of points in the xy plane (separated at a $\frac{1}{4}$ inch resolution) which are back-projected into the image at increasing heights through the volume. You can see the radial distortion present in the images, as the grid appears to bow heavily at the top of the volume. This type of back-projection can be accomplished only when the extrinsic parameters have been determined. The extrinsic calibration process requires only one image per camera. If all of the cameras extrinsic parameters are estimated by observing the same planar points, we say the the cameras are calibrated with respect to the same *world coordinate system*. If we use a checkerboard to calibrate the extrinsic parameters, then the checkerboard must remain in the same static position while images from all of the cameras are acquired. However, the targets should span a large portion of the field of view for each camera [25], so the calibration checkerboards make poor targets for calibration of multiple wide-angle cameras, unless the baselines between the cameras are small enough that the checkerboard can occupy a large portion of all the camera images from a single static location.

The only drawback to this calibration technique is that the numerical method for computing the intrinsic parameters requires that there be no two images in the data set where the checkerboards are related by a pure translation. In cases where this happens, the iterative process is numerically unstable and does not converge. A complete description of this process can be found in Zhang [25], and an implementation can be found in the Open Source Computer Vision Library [2], with optimized shared libraries for the Win32 and Linux operating systems.

2.2 Background Subtraction

Background subtraction is a commonly used method for motion detection and silhouette extraction. We describe a method that combines the best features of existing techniques which are compatible with real-time performance.

Carlson [4,5] and Davis [8] used only monochrome cameras in their systems. Depending on the environment, monochrome cameras can be adequate. In the first two papers by Carlson [4,5], the background subtraction algorithm first filters all images with a low-pass filter, and saves the first image as a reference image. A threshold image, consisting of a separate threshold value for each pixel, is initialized to a constant value. Both the reference image and the threshold image are updated slowly over time. The reference image incorporates new image data by averaging new images into the reference image at a rate specified by the user. The threshold image is updated by incrementing or decrementing the initial threshold pixel values based on their rate of change with respect to the current image. After a period of time, the slowly updating threshold image will approximate the standard deviation of the video images over time. In Davis' paper [8], the image statistics are calculated during a training period during which a mean background reference image and a standard deviation image are computed directly.

In Davis' work [8], because of the use of back-lighting, the problem of shadows was negligible. Carlson [4,5] did not address the problem of shadows. More specifically, their Volumetric Video Motion Detector (VVMD) did not attempt to construct volumetric data near the floor, where the problem of cast shadows is most severe.

The goal of Cheung *et al.*[6] was to accurately reconstruct human motion, so shadows were a much larger concern. In addition to intensity they used color for extra discrimination power. Rather than using image statistics such as the mean and standard deviation, they used a complicated scheme which involved manually partitioning each image into separate regions, which were then assigned different upper and lower intensity thresholds. They also used color as a secondary means of differentiating foreground and background.

In Davis and Bradski [9], the algorithms are simple enough to run in real-time, but are still interesting in the sense that they adapted a new technique to solve an old problem. Like Carlson [4,5], they used simple image statistics as reference images, but they also used a combination of morphological processing, contour extraction and filling algorithms to fill the small holes that are pervasive in silhouette images.

To summarize, we examined different techniques for background subtraction, adopted those that fit our goals, and adapted those that did not. We used a hybrid algorithm that combined statistical image differencing, hue angle differencing, and contour analysis. This will be described in Section 3.3.

2.3 Shape-from-silhouette

Reconstructing an object's 3D shape from a set of images collected using multiple cameras is a classic computer vision problem. In the last few years, this problem has generated considerable interest, partly due to a number of new applications (e.g. Rander and Kanade [20]) that require good volumetric reconstructions. Shape from silhouette is a well-known approach to this problem, dating back at least to the early 1980's [18]. In the volumetric approach to shape-from-silhouette, the scene is represented as a set of three dimensional voxels, and the task is to label the individual voxels as occupied or empty. These methods work by first computing the object's silhouettes from each camera image, typically using background subtraction. In the geometric approach to shape-from-silhouette, every silhouette pixel from every camera image is projected into 3D space as a conic solid. After these solids are intersected, what is left is a geometric representation of the visual hull.

It is important to note that the views of an object from multiple cameras do not uniquely determine the shape of a non-convex object. This is true even with an infinite number of cameras. The visual hull of an object [15] is defined to be the maximal object that is consistent with the object's silhouette from any viewpoint. It is impossible to distinguish two different objects with the same visual hull purely by their silhouettes.

Many researchers have been experimenting with various algorithms to implement shape-from silhouette. Matusik *et al.* [19] used a geometric approach to silhouette intersection which required the use of a distributed computer system to solve for the intersections in real-time. They show that their intersection algorithm parallelizes linearly using a divide-and-conquer approach. However, this process is computationally intensive. Cheung *et al.*[6] use a volumetric method on one computer, but distributes the silhouette extraction among five others. The volumetric methods tend to be simpler, but use more memory and do not scale as well as the geometric methods. In Cheung's more recent work, he uses a high resolution volumetric approach. This is performed entirely off-line, and uses a marching cubes algorithm to extract an isosurface [7].

Recently, researchers have begun to experiment with systems that can both generate and interpret volumetric data in real-time. The system developed by Cheung *et al.* [6] uses six computers and five color cameras. It supports a voxel resolution of $64 \times 64 \times 64 \times 1\text{in}^3$. The main purpose of their system was 3D reconstruction of human motion. Simple fitting of ellipses was performed to track extremities. Carlson *et al.*[4,5] used a real-time sensor with a lower voxel volume of about 40^3 voxels and a voxel size of 3in^3. The main purpose of their system was real-time 3D interaction with a commercial robotic manipulator. A key difference between these two systems is that Cheung's was implemented on a distributed system that used color cameras, while Carlson's system was implemented using monochrome cameras and a single computer. Such a system is also used by Luck *et al.*[17], where the goal is also human motion tracking. However, because the Luck's system used monochrome video frame grabbers, the volumetric data computed by Cheung looks qualitatively better. One of the goals of our work is to combine the two approaches by using four color video framegrabbers in a single computer. In this way we produce a system with qualitatively better volumetric data than that computed by Luck *et al.*[17] without the major hardware requirements of Cheung *et al.*[6].

3 Implementation of RTS^3

3.1 System Architecture

RTS^3 is the system we built to acquire volumetric data. This sensor uses a combination of industry standard components including a high-end PC, four analog color video cameras, and four PCI-bus color frame grabber cards. Using this hardware, we create a time-varying volumetric image of the visual hull of whatever object is moving in the space observed by all four cameras. Figure 6 shows the positions of the four cameras from above. An additional goal was to design the software with a standard application programming interface (API) to the cameras, so that new types of cameras could be added with relative ease. To date, we have used the system with analog color framegrabbers, RS-170 monochrome framegrabbers, and IEEE 1394 (FireWire) cameras.

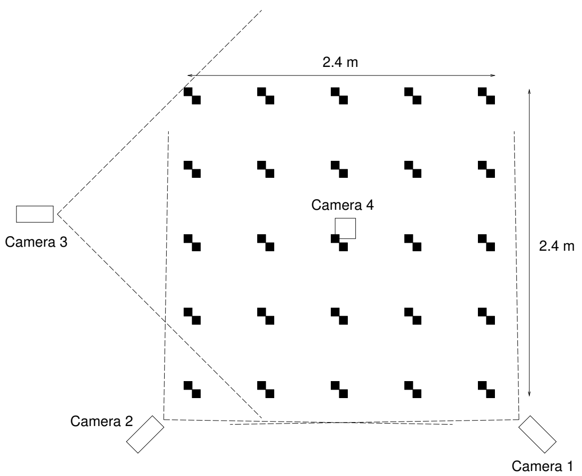

Fig. 6 Layout of the $\mathbf{RTS^3}$ lab at the University of New Mexico. This is a view from above. The cross-hatches are the extrinsic calibration targets. The dashed lines represent the approximate fields of view of the cameras.

We implemented RTS^3 using a software framework developed at Sandia National Labs [21] called Umbra. It is a cross-platform development environment (Windows, Irix, and Linux operating systems) written in C++, Tcl/Tk [32], OpenGL [27] and the Standard Template Library (STL). To better illustrate the 3D nature of the volumetric data, the voxels are rendered as spheres. Spheres of a single color occupy planes of constant Z value. This method is only used for illustration, as it creates a graphics bottleneck at higher resolutions. Normally the voxels are displayed as simple points. A second pop-up window displays the real-time silhouette information.

3.2 Camera Calibration Methodology

Our implementation of camera calibration involved two stages. As mentioned in Section 2.1, a series of checkerboard images was used to estimate the intrinsic parameters. The checkerboard image yields a set of 2D points that corresponds to points on a 3D rigid body. Next, each camera was calibrated relative to a common world coordinate system. The camera views have intrinsic calibration parameters associated with them. The manual identification of the targets is performed by the user. Inaccuracies are minimized by searching a 5×5 pixel neighborhood of the points around the selected coordinate. This provides for an optimal sub-pixel match for the center of the target. Once all targets are selected, the user invokes a function to find the extrinsic parameters. This GUI allows for multiple cameras to be selected and calibrated to the same world coordinate system.

Once the intrinsic and extrinsic parameters of each camera have been determined, the next step is to calibrate the common volume observed by all the cameras. Our system uses lookup tables to store precalculated index information. In summary, each voxel centroid position is back-projected into each of the camera images using the extrinsic and intrinsic calibration parameters. The algorithm to create the individual lookup tables is shown in Algorithm 1, where *BackProject()* takes a point in 3D, (X, Y, Z), and a set of camera calibration parameters, and returns the pixel position, $(\hat{u}, \hat{v})$, that the point maps to using the intrinsic and extrinsic calibration parameters for that camera. In our system, the precomputation allows us to perform the silhouette intersection in real-time. Each voxel holds a single pixel position for each camera. In the lookup table file, this is represented as an ordered pair, (u, v). At run time, a pixel position is translated into a pointer to the specific memory location representing that pixel in the silhouette image. On 32-bit operating systems pointers use four bytes of memory. This leads to a voxel lookup table whose total size is $4 \times \#Cameras \times N_x \times N_y \times N_z$ bytes (where N_x, N_y, and N_z are the number of voxels along their respective axes). The pseudocode for *BackProject()* is shown in Algorithm 2. These calibration operations are all performed offline. The precomputation of the 3D to 2D associations allows the system to perform in real-time.

Algorithm 1: Create 3D to 2D Look Up Tables

input : Volume origin, voxel sizes, volume resolution, and camera calibration parameters
output : A look-up table that associates 3D points in the volume to 2D points in the camera image planes

CreateLookup($x_0, y_0, z_0, \Delta x, \Delta y, \Delta z, N_x, N_y, N_z$)
$voxel \leftarrow 0$
begin
  for $k \leftarrow 1$ **to** N_x **do**
    for $j \leftarrow 1$ **to** N_y **do**
      for $i \leftarrow 1$ **to** N_z **do**
        //X, Y, Z are the actual pts. in 3D
        $X \leftarrow x_0 + k\Delta x$
        $Y \leftarrow y_0 + j\Delta y$
        $Z \leftarrow z_0 + i\Delta z$
        for $camera \leftarrow 1$**to**$\#cameras$ **do**
          $lookup[camera][voxel] \leftarrow$
          $BackProject(X, Y, Z, camera)$
        end
        $voxel \leftarrow voxel + 1$
      end
    end
  end
  return($lookup$)
end

Algorithm 2: Back Project 3D Point to Camera

BackProject($X, Y, Z, camera$)
begin
  $\mathbf{w} \leftarrow [X, Y, Z, 1]^{\mathrm{T}}$
  $T \leftarrow camera.T$
  $A \leftarrow camera.A$
  $k_1 \leftarrow camera.k_1$
  $k_2 \leftarrow camera.k_2$
  //put w into camera coord. frame
  $\mathbf{w'} \leftarrow T\mathbf{w}$
  // project $\mathbf{w'}$ onto the image plane
  $\mathbf{m} \leftarrow A\mathbf{w'}$
  //scale by m_2
  $x \leftarrow \frac{m_0}{m_2}$
  $y \leftarrow \frac{m_1}{m_2}$
  //correct for radial distortion
  $r \leftarrow \sqrt{x^2 + y^2}$
  $\hat{x} \leftarrow x + (x - p_x)[k_1(r) + k_2(r^2)]$
  $\hat{y} \leftarrow y + (y - p_y)[k_1(r) + k_2(r^2)]$
  return $(\hat{x}, \hat{y})$
end

The algorithms to construct the volume calibration are $O(n)$, where n is the number of voxels in the volume. The function *BackProject()* is $O(1)$. This assumes the number of cameras is a constant, and is *far* less than the number of voxels.

3.3 Background Differencing and Silhouette Extraction

The initial background subtraction algorithm was the simplest part of the sensor to implement, but creating an algorithm that performs robustly in many different lighting conditions is partly art. There were many problems to overcome, including noisy cameras, shadows, and separating silhouette pixels from background pixels.

The first problem we addressed was the elimination of shadows, especially those cast on the floor near the feet. In Cheung *et al.*[6] and Davis and Bradski[9] color cameras and framegrabbers were used to acquire 24-bit color data. Although these authors used slightly different techniques, their results were similar. Cheung *et al.*[6] calculated a hue angle difference when the intensity difference values were within a specified range. Davis and Bradski[9] estimated a mean and a standard deviation image (over a series of initial images) where they differenced the mean and the current image and compared it to a threshold which was proportional to the standard deviation of the background at that pixel.

We encountered a similar problem with cast shadows. The first version of RTS^3 was implemented using monochrome cameras. An example of the problem of cast shadows is illustrated in Figure 7 and in Figure 8. Figure 8 shows how shadows in the image produce errors in the volumetric reconstruction.

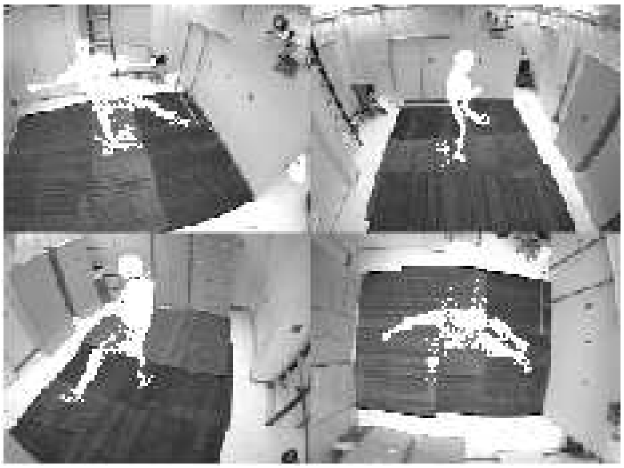

Fig. 7 This shows how using only monochrome images can lead to poor silhouette segmentation caused by shadows.

The key assumption used for eliminating shadows by color image differencing is that the area where a shadow is cast generally shows a large change in intensity but a small change in hue. Hue is an angular representation of color. Therefore, the angular difference between a pixel in the background and a shadowed pixel from the foreground should be relatively small. Let $\mathbf{c_1}$ equal the current RGB pixel. Let $\mathbf{c_0}$ equal the background RGB pixel. The change in hue between these two values is defined as:

$$\Delta H = \cos^{-1}\left[\frac{\mathbf{c_1} \cdot \mathbf{c_0}}{||\mathbf{c_1}||\,||\mathbf{c_0}||}\right]. \tag{8}$$

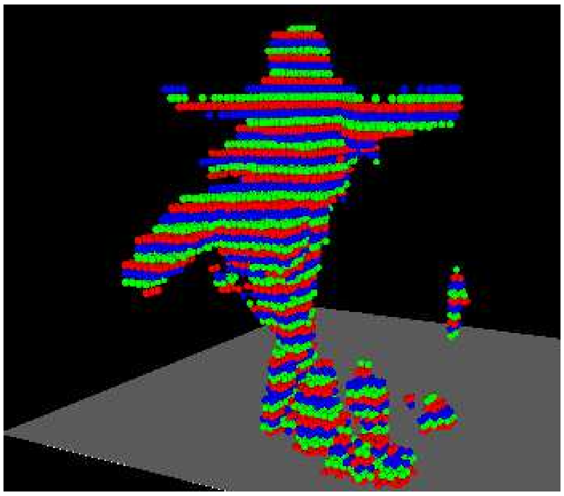

Fig. 8 This shows how cast shadows can introduce noise in the volumetric reconstruction. The shadows cast on the floor produce errors near the legs and feet in the 3D data.

We used hue difference as an additional check against classifying shadow pixels as silhouette. Cheung's system[6] was implemented in a distributed fashion. Specifically, the image processing for each camera was handled by a single computer. Since our goal was to implement the entire system on a single computer, this calculation was optimized by using look-up tables for the square root and inverse cosine functions. These look-up-tables are accurate to five significant digits. The addition of optimized hue differencing allowed us to raise the intensity thresholds above the level typically produced by shadows cast on the floor.

The next problem we addressed was the noise in individual cameras. Ambient noise levels can vary widely between cameras, which makes it impossible to use the same threshold values across a set of cameras that possibly have different noise properties.

One way to evaluate the noise in a digital camera is to take the standard deviation of a series of images. This is part of the method for statistically estimating a background image as discussed in Davis *et al.* [9]. The standard deviation image approximates the noise of the image sensor. An example of the standard deviation image (estimated using a set of 75 color images) is shown in Figure 9. The standard deviation is estimated across the red, green, and blue color planes individually. Brighter colors correspond to higher standard deviations. Notice that the images from cameras one and two have a much lower standard deviation than the images from cameras three and four. This illustrates that different cameras can have very different noise characteristics.

We merged these two methods (intensity differencing and hue differencing) into a single procedure that is both fast and flexible. The pseudocode for our procedure is shown in Algorithm 3

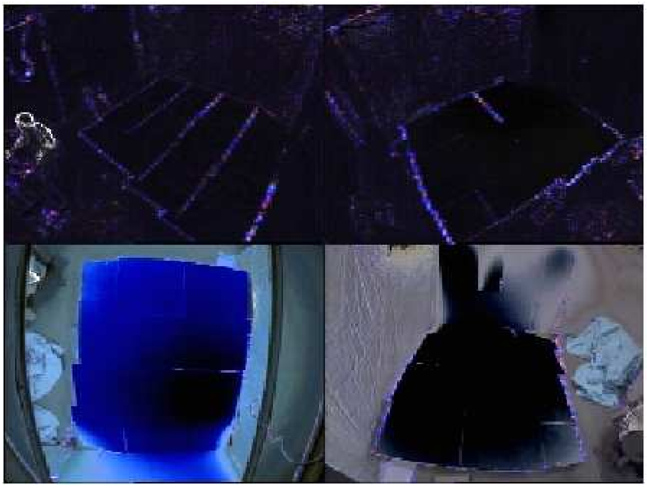

Fig. 9 This figure shows the per pixel standard deviation image estimated from 75 color images. The standard deviation image can be used for per-pixel thresholding and can compensate for differences in gain, exposure, and noise between cameras. The majority of the noise appears in the blue pixels. This is caused by the physical format of the CCD element, which gives more area to red and green pixels. This means that the blue elements on the CCD are being lit with a smaller number of photons, and must therefore use a higher gain.

The results of the background differencing algorithm are shown in Figure 10. The shadows have been successfully removed. The colored pixels have been classified as silhouette using the rules described in Algorithm 3. Pixels classified due to the intensity test are displayed as blue. Pixels classified due to the standard deviation test are displayed as red. Pixels classified due to the hue angle test are displayed in green. In Figure 11, the volumetric data computed using color images exhibits none of the noise that was present in the volumetric data computed using monochrome images.

The last problem we addressed was filling the small holes in the silhouettes. Typically, for this purpose, some kind of local morphological operator is used after thresholding. Unfortunately, the use of morphological operators can introduce errors in the 2D silhouette, which, when projected, produce errors in the reconstructed 3D shape. Our approach was to search the image for contours. We used a function in the OpenCV library (see [2]) that decomposes a binary image into a two level hierarchy of contours: exterior contours and their interior holes.

The next step was to fill the small interior regions with the value one, and to fill the small exterior regions with the value zero. This has the combined effect of removing holes from the large silhouettes, and removing speckle from the background. Fortunately, the function in the OpenCV library is fast enough to run in real-time.

Fig. 10 This picture shows the improvement in the silhouettes which is achieved by using color images and differencing with hue and standard deviation. Pixels classified due to the intensity test are displayed as blue. Pixels classified due to the standard deviation test are displayed as red. Pixels classified due to the hue angle test are displayed in green. The corresponding improvement in the 3D data is seen in Figure 11.

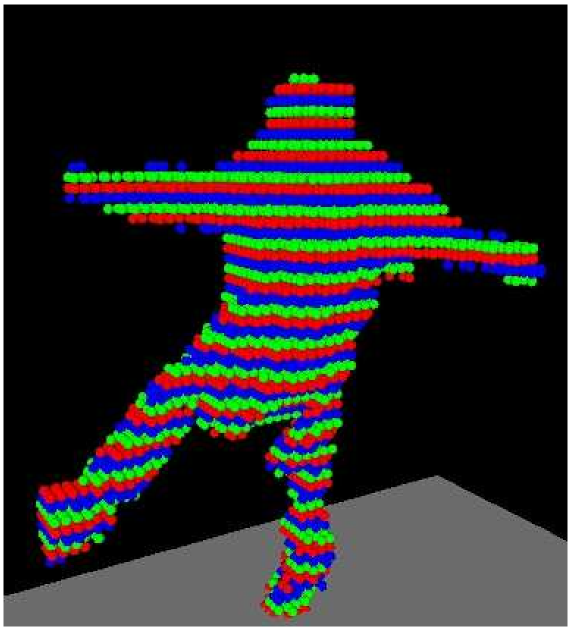

Fig. 11 This picture shows the improvement in the 3D data which results from using a color image processing technique to extract the silhouettes, as shown in Figure 10.

Algorithm 3: Extract Silhouette from Background

input : Color value, mean, and std. dev. images.
output : A binary silhouette.
ExtractSilhouette(R, R_σ, R_μ, ..., B_μ)
begin
  for $i \leftarrow 1$ **to** $\mathrm{N_v}$ **do**
    for $j \leftarrow 1$ **to** $\mathrm{N_u}$ **do**
      $silhouette[i,j] \leftarrow 0$
      $\Delta r \leftarrow R[i,j] - R_\mu[i,j]$
      $\Delta g \leftarrow G[i,j] - G_\mu[i,j]$
      $\Delta b \leftarrow B[i,j] - B_\mu[i,j]$
      if (not ($|\Delta r| < I_{lower}$ or $|\Delta g| < I_{lower}$ or $|\Delta b| < I_{lower}$)) **then**
        if ($|\Delta r| > I_{upper}$ or $|\Delta g| > I_{upper}$ or $|\Delta b| > I_{upper}$) **then**
          $silhouette[i,j] \leftarrow 1$
        else
          $\mathbf{c_1} \leftarrow [R[i,j], G[i,j], B[i,j]]^{\mathrm{T}}$
          $\mathbf{c_0} \leftarrow [R_\mu[i,j], G_\mu[i,j], B_\mu[i,j]]^{\mathrm{T}}$
          $\Delta H \leftarrow \cos^{-1}\left[\frac{\mathbf{c_1}\cdot\mathbf{c_0}}{\|\mathbf{c_1}\|\|\mathbf{c_0}\|}\right]$
          if ($\Delta H > \Delta H_{max}$) **then**
            $silhouette[i,j] \leftarrow 1$
          else
            if ($|\Delta r| > R_\sigma[i,j]$ or $|\Delta g| > G_\sigma[i,j]$ or $|\Delta b| > B_\sigma[i,j]$) **then**
              $silhouette[i,j] \leftarrow 1$
            end
          end
        end
      end
    end
  end
end

Figure 12 shows the silhouette images before contour processing and Figure 13 shows the silhouette images after contour processing.

3.4 Implementation of Shape-from-silhouette

We implemented two different shape-from-silhouette algorithms, which we have termed: 1) PixelVolume; and 2) HybridVolume. Both algorithms rely on table look up to speed the process of back-projection. The PixelVolume algorithm is shown in Algorithm 4. It works by traversing every voxel in the volume, and computing the (x, y) location in each image that the voxel centroid projects to. It examines the appropriate pixels in each silhouette image and if they contained within the silhouette, that voxel is marked as occupied. If a single pixel is outside the silhouette, then that voxel is marked as empty. Because we wanted to increase the frame-rate for tracking a human figure, a key optimization was to restrict the processing to voxels in the volume around the centroid of the human figure. The centroid, (x_0, y_0, z_0), and maximum height in the Z axis, Z_{max}, are tracked from frame to frame. The volumetric window is a cube that is Z_{max} wide, Z_{max} long and Z_{max} high. This cube is centered in the xy plane on (x_0, y_0). Depending on the full size

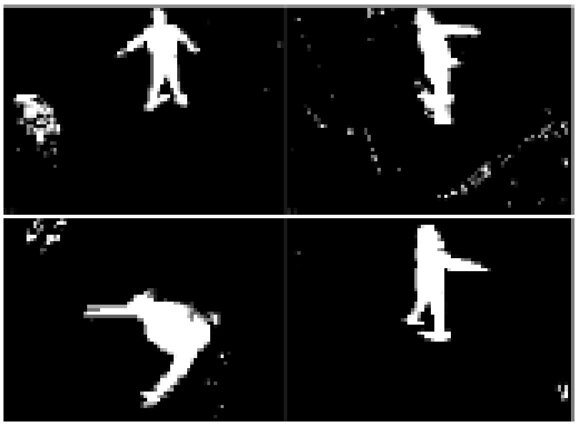

Fig. 12 Silhouette images from all four cameras before contour processing.

Fig. 13 Silhouette images from all four cameras after contour processing. The noise in the silhouette images has been successfully reduced without substantially changing the shape of the silhouettes.

Algorithm 4: PixelVolume Shape from Silhouette

```
input  : An array of silhouette images.
output : A binary 3D volumetric image.
PixelVolumeSFS(silhouettes)
begin
    for k ← 1 to N_x do
        for j ← 1 to N_y do
            for i ← 1 to N_z do
                for camera ← 1to #cameras do
                    voxel ← voxel[i, j, k]
                    (u, v) ← lookup[camera][voxel]
                    if (silhouettes[camera, u, v] = 0)
                    then
                        // If a voxel does not project
                        to a silhouette, break.
                        break
                    end
                end
                // If for-loop exits normally, mark
                voxel.
                if (camera = #cameras) then
                    volume[i, j, k] ← 1
                else
                    volume[i, j, k] ← 0
                end
            end
        end
    end
    return(volume)
end
```

of the monitored volume, this can result in significant, constant time savings.

The HybridVolume algorithm is a bit more complex. It is shown in Algorithm 5. A single camera is designated the *dominant* camera. For this single camera, we generate look-up table different from those used in the PixelVolume algorithm. Rather than looking up pixels from voxels, we invert the problem, and for each pixel, we store the list of voxels which that pixel projects to. For the dominant camera, we traverse all of the pixels, and we push voxels associated with pixels belonging to the silhouette onto a stack. After all pixels in the dominant camera have been examined, we begin to pop the voxels from the stack. At this point the stack contains a subset of the voxels in the volume. For each of the voxels on the stack, we use the PixelVolume algorithm to check that voxel in the images from the remaining cameras. If the voxel projects to silhouette pixels in the images from the remaining cameras, we mark it as occupied. If we encounter even a single non-silhouette pixel, then we mark that voxel as empty. The major difference between this algorithm and the PixelVolume algorithm is that HybridVolume drastically reduces the amount of time it takes to compute a volumetric frame. In practice, we have found that the camera which is most suitable to be the dominant camera is the one mounted on the ceiling looking down. Of course, this depends on the geometry of the monitored volume and the relative positions of the cameras. In general, a downward looking camera on the ceiling observes a smaller silhouette than do the other cameras. This has the effect of sampling a smaller amount of the volume for further testing than say, a side-looking camera.

If cameras noise occurs around specific features common to all camera views (e.g. shadows cast on the floor), then that noise becomes spatially correlated and produces false voxels. One nice thing about this approach is that it is reasonably tolerant of individual camera noise. This is because there is little spatial correlation between the noise of individual cameras. An example of this can be seen in Figure 14. Here, one of the cameras is producing a large amount of noise, while the others are not. Surprisingly, this results in a 3D reconstruction that does not contain large amounts of 3D noise, as shown in Figure 15. Volume intersection algorithms are tolerant to

Algorithm 5: HybridVolume Shape from Silhouette

input : An array of silhouette images.
output : A 3D binary volumetric image.

```
HybridVolumeSFS(silhouettes)
begin
    for j ← 1 to N_v do
        for i ← 1 to N_u do
            if silhouettes[DominantCamera, i, j] =
            1 then
                for ℓ ← 1 to VoxelList[i, j].size() do
                    stack.push(VoxelList[i, j].voxel[ℓ])
                end
            end
        end
    end
    for i ← 1 to stack.size() do
        voxel ← stack.pop()
        //Use the PixelVolume method for the re-
        maining cameras.
        for camera ← 1 to #cameras − 1 do
            (u, v) ← lookup[camera][voxel]
            if (silhouettes[camera, u, v] = 0) then
                break
            end
        end
        // If for-loop exits normally, mark voxel as
        occupied.
        if (camera = #cameras − 1) then
            voxel.MarkAsFilled()
        else
            voxel.MarkAsEmpty()
        end
    end
end
```

this kind of noise because a pixel must be classified as silhouette in images of *all* of the cameras in the system to create an occupied voxel.

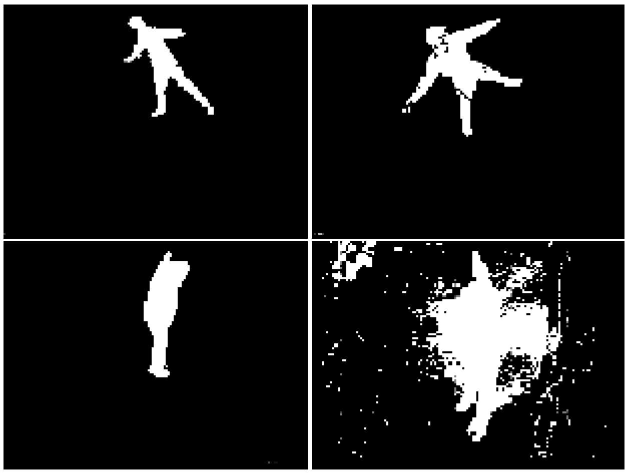

Fig. 14 A demonstration of the robustness of the system to noise in one camera. The 3D reconstruction resulting from this is shown in Figure 15.

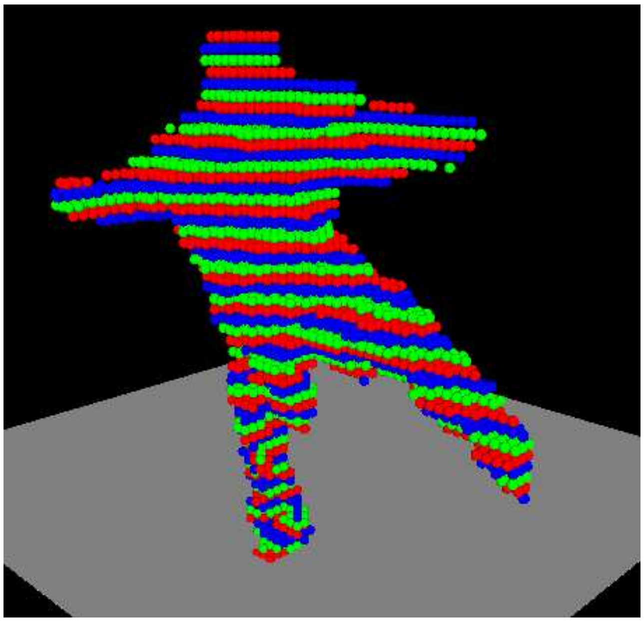

Fig. 15 An example of the effect of noise in one camera. One noisy image does not produce significant noise in the volumetric reconstruction. Consequently, the system is very robust to small changes in any of the cameras.

4 Experiments and Results

4.1 Resolution vs. Frame-rate

We experimentally compared the run-time behavior of the two different volume traversal algorithms: PixelVolume and HybridVolume. We tested the frame-rate of the RTS^3 system at 24 levels of volumetric resolution. We started with a Look Up Table (LUT) resolution of 5×10^6 total voxels, and reduced the resolution of each successive set of LUT's by 2×10^5 voxels. The final LUT in the set had approximately 8×10^5 voxels. For each LUT, we presented the same set of 600 color images to the system. The first 75 images were used to compute the mean and standard deviation images used for background subtraction, and the remaining 525 images showed a subject entering the volume, performing a series of motions, and exiting. To perform these tests, the RTS^3 system was modified to work in batch mode, using "write-images" and "read-images" functions. These functions work on the raw image data. The only sense in which batch mode differs from online mode is that the images are read off of the hard disk in batch mode instead of being read directly from the framegrabbers.

The results were somewhat surprising. The performance of the two algorithms is plotted in Figure 16. The frame rates were averaged over the 600 image frames at each LUT resolution.

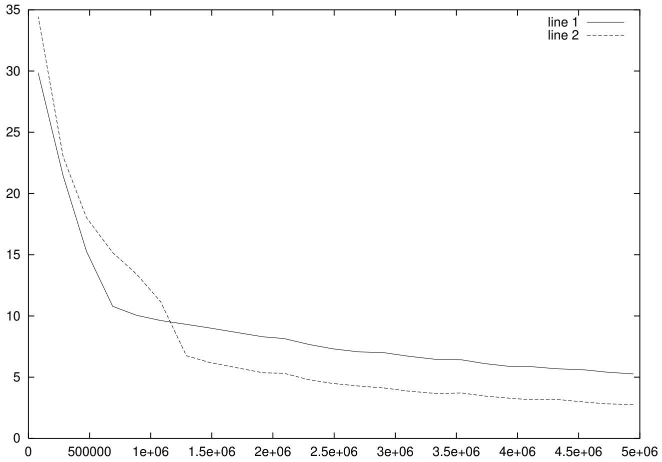

Fig. 16 Average frame-rate for the PixelVolume algorithm (dashed) and the HybridVolume algorithm (solid). These timing tests were performed using the same set of 600 images over a set of 24 linearly decreasing volumetric resolutions.

In both approaches, the frame rate goes up dramatically as the number of voxels falls below 10^6. This is due to cache locality. At higher volumetric resolutions, the number of context switches between cache and main memory is the main bottleneck in the algorithm. At lower resolutions, the volume is represented by substantially less memory, and thus it takes fewer and fewer main memory swaps into cache to traverse the entire volume.

The image resolution used in the tests was 160×120 pixels. Four images of this size requires 76 kilobytes of memory. The amount of Level 2 cache on the Pentium 3 CPU is 256 kilobytes. At this image resolution, we can fit all four silhouettes into cache. The remaining cache can be used to process the volume data.

The HybridVolume approach had frame rates that were twice as fast for the high resolution volumes. This is because the number of voxels that are checked is roughly one half the number checked at the same resolutions using the PixelVolume approach. The PixelVolume approach uses a rectangular window around the subject that is as wide and long as it is high. The HybridVolume approach projects the silhouette from the dominant camera from the top of the volume to the bottom. This results in a smaller number of voxels that are checked for occupancy. As the distance between the dominant camera and the volume is increased, the percentage of voxels that is pushed on the stack for testing is decreased.

4.2 Real-Time Synthetic 3D View

For the final experiment, we implemented a system for creating real-time synthetic 3D views. The synthetic 3D view problem has been studied widely by many researchers [20,16,12]. Rander *et al.*[20] used dense stereo to resolve the 3D figure from 50 cameras, which yielded good results, but was not real-time. Levoy *et al.* [16] and Gortler *et al.* [12] explored the area of light field rendering, which is a purely image based technique that does no 3D reconstruction. Results from these systems are impressive, but are expensive to compute. Consequently, all computation is done off-line.

With the advent of high-end commodity graphics cards, real-time texture mapping has become a realistic way to approach the problem of generating synthetic 3D views. Our approach to the problem was to use the RTS^3 system to generate a 3D volumetric model. In normal operating mode, the rendering of the volume data was accomplished using simple points (as opposed to surfaces or polygons). These points are defined by a position and size, and have no other geometric attributes associated with them. For this reason they are rendered as flat squares of constant height and width. In modern graphics APIs (OpenGL, DirectX, etc.), texture cannot be applied to points using the standard image and texture-coordinate associations. In order to take advantage of fast texture-mapping hardware, we need to represent the volumetric data with polygons instead of points. There are many approaches that have been investigated. Cheung's recent work [7] uses a marching cubes algorithm to extract an isosurface. Because this is expensive to compute, it was not considered for our real-time system.

To solve this problem, we employed a special kind of polygon: a single square that is always perpendicular to the optical axis of the synthetic view generated by the 3D graphics environment. These special polygons are known as *billboard polygons*. If we did not use billboards, it would be possible to look at the polygons on edge in which case the polygons would seem to disappear. Instead of shearing the polygon we warp the texture using the accelerated graphics hardware. Implementation of billboard polygons is relatively straight forward. The graphics API provides a method for extracting the current position and orientation of a "virtual camera" which is represented by the 4×4 homogeneous transform T (see Equation 4). The upper 3×3 matrix of $\mathbf{T}$ is $\mathbf{R}$, the rotation matrix. The 3 column vectors of $\mathbf{R}$ are the unit vectors of the rotation: $\mathbf{c_1}, \mathbf{c_2}$ and $\mathbf{c_3}$. The size of the billboard needs to be the width of a voxel (assuming cubic voxels) in order to avoid aliasing. Let a be a scale factor equal to half the voxel size. To form a billboard that is appropriately oriented at a new position, $\mathbf{r} = [\mathbf{X}, \mathbf{Y}, \mathbf{Z}]^T$, we need to calculate the four new corners of the billboard during each render cycle . The following formulas are used to derive the new corner positions in terms of the position, $\mathbf{r}$, the scale factor, a, and the unit vectors of rotation, $\mathbf{c_1}$, $\mathbf{c_2}$ and $\mathbf{c_3}$:

$$\mathbf{b_1} = \mathbf{r} + a(-\mathbf{c_1} - \mathbf{c_2})$$
$$\mathbf{b_2} = \mathbf{r} + a(\mathbf{c_1} - \mathbf{c_2})$$
$$\mathbf{b_3} = \mathbf{r} + a(\mathbf{c_1} + \mathbf{c_2})$$
$$\mathbf{b_4} = \mathbf{r} + a(-\mathbf{c_1} + \mathbf{c_2})$$

where $\mathbf{b_1}, \mathbf{b_2}, \mathbf{b_3}$ and $\mathbf{b_4}$ are the corners of the billboard. See Figure 17.

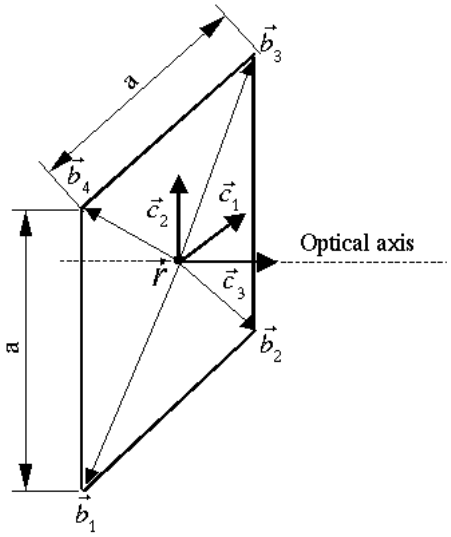

Fig. 17 This figure illustrates how a billboard polygon is computed. The corners of the billboard are updated during each rendering cycle, and repositioned so that the billboard is normal to the optical axis of the virtual camera.

The next step is to choose the appropriate camera for each new position that the "virtual camera" is in. This is accomplished by examining the dot product between each of the physical cameras normal vectors (the cosine vector of the optical axis), and the normal vector for the billboards, $-\mathbf{c_3}$. The camera with the minimum dot product between these two values is the one whose texture is chosen to map all of the billboards. The texture coordinates for each billboard are computed in exactly the same manner as we derive the pixel locations for individual voxels. We use the same function, *BackProject()*, given in Algorithm 2. For every billboard we project its four corners into the camera with the best view of that polygon.

Experimental results are shown in Figure 18. These results do not appear as realistic as work that can be done with fifty cameras offline [7]. However, these results were achieved in real-time with just four cameras. While navigating through the 3D environment, the best live camera is automatically chosen for the current view. Another advantage of this approach is that we can increase the number of cameras, and still maintain real-time rates. This can be accomplished by using only a small subset of cameras for volumetric data generation (three or four), and using the rest for generating real-time calibrated texture maps. The more live cameras that are used as sources for calibrated texture, the better the interpolation between views.

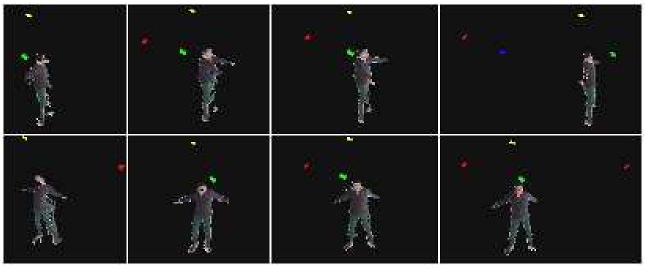

Fig. 18 This is a montage of the volumetric data and image data being used to derive synthetic 3D views. As more cameras are added to the system, the view interpolation will improve significantly

5 Conclusions

In this paper, we presented work on the design, analysis and implementation of RTS^3. The system produces time-varying volumetric data at real-time rates (10–30Hz). The data is in the form of binary volumetric images. The techniques needed to implement this system were discussed in detail: calibration, silhouette extraction, and volumetric intersection. We produced qualitative results comparable to other systems described in the literature, minimized the system's hardware requirements, and still achieved real-time performance. In this paper we reviewed the previous work in the field, and derive the mathematics behind volumetric calibration, silhouette extraction, and shape-from-silhouette. We reviewed the topics of camera calibration and volumetric intersection. For our sensor implementation, four color cameras were installed in a lab and used with our software to track volumetric motion. An analysis of resolution vs. frame rate was performed between two different implementations of the shape-from-silhouette technique. This showed that the HybridVolume approach is superior to the simpler PixelVolume approach. An application of the system to the problem of generating synthetic 3D views was also demonstrated.

Because the RTS^3 system works without encumbering wires and photo-reflective tape, is is a natural choice as the base technology for a new generation of motion capture devices. Our system is currently being used by the Colorado School of Mines as a data acquisition system for the purposes of markerless human motion tracking. In our most recent collaborations we are tracking a nineteen degree of freedom human kinematic model.

As Moore's law continues to increase computational power and reduce the size of the electronic components, this type of 3D computer vision system will soon be seen in the home, workplace, and community.

References

1. Bougnoux S (1998) From Projective to Euclidean Space Under Any Practical Situation, A Criticism of Self-Calibration. *Proc. 6th International Conference on Computer Vision (ICCV '98)*, pp. 790-796.
2. Bradski G, Editor. (2001, May 15) The Open Source Computer Vision Library. [WWW Document]. URL http://www.intel.com/research/mrl/research/opencv
3. Bregler C and Malik J (1998) Tracking People with Twists and Exponential Maps. *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR '98).*
4. Carlson JJ, Stansfield SA, Shawver D, Flachs GM, Jordan JB, and Bao Z (1997) Real-Time 3D Visualization of Volumetric Video Motion Sensor. *Proc. SPIE, Surveillance and Assessment Technologies for Law Enforcement*, **2935**, pp. 69-79.
5. Carson JJ, Little CQ, and Nelson, CL (1998) Advanced 3D Sensing and Visualization System for Unattended Monitoring. Sandia National Laboratories Technical Report SAND98-2803.
6. Cheung G, Kanade T, Bouguet J, and Holler M (2000) A Real-Time System for Robust 3D Voxel Reconstruction of Human Motions. *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR '00)*, pp. 714-720.
7. Cheung G, Vedula S, and Kanade T (2001, May 14). Spanish Dancing (Flamenco) with Music. [WWW Document]. URL http://www.cs.cmu.edu/~virtualized-reality/dance.html
8. Davis J and Bobick A (1997) Virtual PAT: A Virtual Personal Aerobics Trainer. MIT Media Lab Perceptual Computing Group Technical Report No. 436.
9. Davis J and Bradski G (1999) Real-Time Motion Template Gradients using Intel CVLib. *Proc. IEEE ICCV Workshop on Frame-Rate Vision.*
10. Ganapathy S (1984) Decomposition of Transformation Matrices for Robot Vision. *Pattern Recognition Letters*, **2**(4), pp. 401-412.
11. Gravilla DM (1999) The Visual Analysis of Human Movement: A Survey. *Computer Vision and Image Understanding* **73**(1), pp. 82-98.
12. Gortler SJ, Grzeszczuk R, Szeliski R, and Cohen MF (1996) The Lumigraph. *SIGGRAPH 1996*
13. Hartley, RI (1994) An Algorithm For Self Calibration from Several Views. *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR '94)*, Seatle, WA, pp. 908-912.
14. Laurentini A (1997) How Many 2D Silhouettes Does It Take To Reconstruct a 3D Object?. *Computer Vision and Image Understanding*, **67**(1), pp. 81-87.
15. Laurentini A (1994) The Visual Hull Concept for Silhouette-Based Image Understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **16**(2), pp. 150-162.
16. Levoy M and Hanrahan P (1996) Light Field Rendering. *SIGGRAPH 1996.*
17. Luck JP, Small DE, and Little CQ (2001) Real-Time Tracking of Articulated Human Models using a 3D Shape from Silhouette Method. *Proc. Robot Vision 2001* **6**(2), pp. 78-86.
18. Martin WN and Garwal JK (1983) Volumetric Descriptions of Objects from Multiple Views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **5**(2), pp. 150-158.
19. Matusik W, Buehler C, Raskar, R, Gortler SJ, and McMillan, L (2000) Image-Based Visual Hulls. *SIGGRAPH 2000*, ACM Press/Addison Wesley Longman, Kurt Akeley, Editor, pp. 369-374.
20. Rander P, Narayanan PJ, and Kanade T (1997) Virtualized Reality, Constructing Time Varying Virtual Worlds from Real Events. *IEEE Visualization*, Volume 552, pp. 277-283.
21. Small DE, Gottlieb EJ, Edlund K, and Slutter C (2000) A Design Patterns Analysis of the Umbra Simulation Framework. Sandia National Laboratories, Albuquerque, NM, Technical Report SAND2000-2380.
22. Tsai, R (1986) An Efficient and Accurate Camera Calibration Technique for 3D Machine Vision. *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR '86)*, Miami Beach, Florida, pp. 364-374.
23. Wei G and Ma S (1994) Implicit and Explicit Camera Calibration: Theory and Experiments. *IEEE Pattern Analysis and Machine Intelligence*, **16**(5), pp. 469-480.
24. Weng J, Cohen P, and Herniou M (1992) Camera Calibration with Distortion Models and Accuracy Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **14**(10), pp. 965-980.
25. Zhang, Z (1999) A Flexible New Technique for Camera Calibration. Technical Report MSR-TR98-71, Microsoft Research.
26. MotionStar Product Details. (2001, May 14) [WWW Document]. URL http://www.ascension-tech.com/products/motionstar
27. OpenGL Graphics Developer Web Site. (2001, May 16) [WWW Document]. URL http://www.opengl.org
28. Northern Digital's Optotrak Sensor. (2001, May 14) [WWW Document]. URL http://www.ndigital.com/optotrak.html
29. Polhemus Web Site. (2001, May 14) [WWW Document]. URL http://www.polhemus.com
30. Peak Performance Technologies. (2001, May 14) [WWW Document]. URL http://www.peakperform.com/biomech_prods.html
31. Reactor Product Details. (2001, May 13), [WWW Document]. URL http://www.ascension-tech.com/products/reactor/details.html
32. Tcl/Tk Developer Exchange, (2001, May 14) [WWW Document]. URL http://www.scriptics.com