Variable Viewpoint Reality: A Prototype for Realtime 3D Reconstruction

by

Owen W. Ozier

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Author .....
Department of Electrical Engineering and Computer Science
September 2, 1999

Certified by .. ..
Paul A. Viola
Associate Professor
Thesis Supervisor

Accepted by ... .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Variable Viewpoint Reality: A Prototype for Realtime 3D Reconstruction

by

Owen W. Ozier

Submitted to the Department of Electrical Engineering and Computer Science
on September 2, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Streams of 2D image information from a single real-world scene are now common – in movies, in sporting events, and in motion capture studios. With easy digital access to that information from multiple camera views, one would like to reconstruct real-time streams of 3D data. Most algorithms for producing 3D models of a scene from several camera views are very time consuming. In this thesis, I describe a system for producing 3D models in real-time from multiple cameras.

The greatest challenges in constructing this system are been managing bandwidth, calibrating cameras, and accurately segmenting foreground from background in camera images. Bandwidth is difficult to manage because there is not enough of it to transmit multiple image feeds over an Ethernet network. Instead of transmitting entire images, this system transmits only compressed silhouettes, resulting in a non-textured model. The compression consists of run-length encoding in non-standard directions. Camera calibration is difficult because this system is in an uncontrolled environment, and as such, identifying landmarks or fiducials is not always reliable. We use an advanced camera model that takes some camera imperfections into account, and calibrate to that model using point-correspondences. Segmentation is also made difficult by the uncontrolled background environment, and at present we use a simple background subtraction augmented with basic morphological operators to run segmentation. Calibration and segmentation techniques are still being developed in this project.

The system I describe is not yet running in real-time, so I discuss the components that have been completed and describe their present level of integration. However, the system has been tested in off-line runs, and I present results from those runs. Finally, I offer directions for future work, from user interfaces to the study of human kinematics.

Thesis Supervisor: Paul A. Viola
Title: Associate Professor

Acknowledgments

I extend the biggest thanks of all to my advisor, Paul Viola, for providing the resources and energy that made this thesis possible, and for helping me through my burned-out moments during research and writing. I would also like to thank Dan Snow, without whom there would be no system of which to speak and no giant calibration objects in the vision arena. Thanks also go to Mike Ross, for his Linux knowhow and for helping the laptop see the world, to John Winn, for letting me know precisely how deft or daft my ideas were, and to Nick Matsakis, for loads of LaTeX help.

Thanks to my parents, Linda and Lance Ozier, for giving me the freedom I had growing up to explore and discover. Special thanks to my brother, Drew Ozier, who would call me late at night to find out how the thesis was going when everyone else had gone to sleep.

I also thank my friends: Jon Zalesky for rounds two and three; Mary Obelnicki for reminding me of what was important; Abigail Vargus for her editorial prowess and for the rice chex; Brenton Phillips and Derek Bruening for putting up with my habit of sleeping late year after year; Justin Miller for the Matrix, the Force, and the art of kindness; Morgan McGuire and Aasia Saleemuddin for Combat Math; Dave Korka and Laughton Stanley for the new apartment (and its carpet); everybody else in Excalibur for the fellowship - it was a fair time that cannot be forgotten, and because it will not be forgotten, that fair time may come again; Carl Dietrich for the closing fanfare; Matt Norwood, for having seen Pete at Todd's car and for being on a roof at the right moment; Audra Parker for standing by me; and Dee Donahue for her smile. There is not room to name you all, but to those not mentioned here, thank you - you know who you are.

Gold Leader, wherever you are, that shot was one in a million – you know the song I hear. Green Team, it was fun.

Contents

1 Introduction **11**

1.1 Motivation 11

1.2 System Overview 12

1.2.1 Approach 13

1.2.2 Silhouette Intersection 14

1.2.3 Parallelization for Realtime 14

1.2.4 Calibration 15

1.2.5 Segmentation 16

1.2.6 My Contributions 16

1.3 Thesis Outline 18

1.3.1 Background 18

1.3.2 Imagelines and Worldlines 18

1.3.3 Parallelization for Real-time 18

1.3.4 Calibration and Segmentation 19

1.3.5 System Performance 19

1.3.6 Conclusions 19

1.3.7 Making Movies 19

2 Background **22**

2.1 Phenomenology of Reconstruction 22

2.2 Previous Work and Our Goal 23

2.2.1 Hidden Surfaces 25

2.3 Geometry of the Algorithm 26

2.3.1 Volume Intersection 27

3 Imagelines and Worldlines **30**

3.1 Saving Bandwidth 30

3.2 Algorithm Development 31

3.3 Saving Computation 34

3.4 Oversampling 34

4 Parallelization for Real-time **38**

4.1 Architecture 38

4.1.1 Cameras 39

4.1.2 Network Layer 40

4.2 Non-Runtime Preprocessing 40

5 Calibration and Segmentation **44**

5.1 Calibration 44

5.1.1 Basic Camera Model 45

5.1.2 Calibrating to the Basic Model 45

5.1.3 Advanced Camera Models: Corrections 46

5.1.4 Easier Calibration 46

5.2 Segmentation 47

6 System Performance **48**

6.1 Slave 48

6.2 Master 49

6.3 Display 49

6.4 Network 50

6.5 Results 50

7 Conclusions **53**

7.1 Difficulties 53

7.1.1 Efficient Use of Bandwidth 53

7.1.2 Calibration 54
7.1.3 Camera Parameters 54
7.1.4 Camera Quality 54
7.2 Improvements 55
7.2.1 Integration for Real-Time 55
7.2.2 Multiple-Camera Segmentation 55
7.2.3 Adding Limited Correspondence 56
7.3 The Future 56

A Glossary **58**

B Making Movies **62**
B.1 Volume Intersection: Cookie-Cutting 63
B.1.1 cameraspitter.c 63
B.1.2 silplane.c 64
B.1.3 silcaster.c 64
B.1.4 mergevect.c and off_boxer.c 65
B.1.5 Final volume intersection diagram 66
B.2 Imagelines and Worldlines 67
B.3 Imagelines and Worldlines 68
B.3.1 worldlinemaker.c 68
B.3.2 imagefromworld.c 68
B.3.3 sectImlTosectWrl.c and sectImlTosect.c 68
B.4 Future Work 69

List of Figures

1-1 System flow diagram 13

1-2 Clockwise from top left: Points on an image; Points on an image with segments of the corresponding lines through space; A region of an image and a section of the cone resulting from casting that image region. In this figure, cameras are represented by a small view frustum; the image is the rectangle. 20

1-3 Imageline directions for various camera angles. From left to right: (1) A level camera, rolled slightly clockwise about the optical axis; (2) a camera looking straight down (or up); (3) a camera looking about $45°$ up from the horizontal, or an upside-down camera looking about $45°$ down from the horizontal; (4) a camera looking slightly down; (5) a level camera 21

1-4 Left: two photos of a Tweety replica used for reconstruction; Right: two renderings of a crude model reconstructed from a sequence of 18 images, including the two at left. 21

2-1 A teapot from three views. 25

2-2 Parts of the teapot not visible from three views in figure 2-1. 26

2-3 On the left, a volume cast form a silhouette; On the right, the intersection of four such volumes (only two of which are shown). 27

2-4 A cube with a piece removed, causing a visual concavity. 28

3-1 A possible silhouette of a person and a frisbee. 31

3-2 Illustrations of how a spatial run-length encoding might be built from a collection of horizontally run-length encoded images. 33
3-3 From left to right: (1) A camera, with some imagelines, looking at a set of worldlines; (2)the same camera, oriented to show a worldline corresponding to the rightmost imageline; (3) the same camera, oriented to show a worldline corresponding to the next imageline. . 34
3-4 Top: three cameras' views of a person, with a worldline; the three cameras with just the worldline. Top middle: the worldline with the run from the right camera highlighted. Bottom middle: the worldline with the run from the center camera highlighted. Bottom: the worldline with the run from the left camera highlighted. 36
3-5 The three runs along a worldline from figure 3-4; at right, the intersection of those runs – the part that would remain after reconstruction. 37
3-6 Nearly best and worst case scenarios for oversampling imagelines. . 37

4-1 System flow diagram from chapter 1. 39
4-2 Pseudocode description of our volume intersection algorithm 43

6-1 Left: A reconstruction with four legs. Right: The reason for the four legs – only four cameras, at redundant positions, can see the legs; the other two cameras are too high. 51
6-2 Six frames from the frisbee reconstruction. In frames 4 and 5, the in-flight frisbee is actually distinguishable. 52

A-1 *get caption from the ImlWrl chapter* 59
A-2 A cube with a piece removed, causing a visual concavity. 60

B-1 An initial sketch of the reconstruction procedure 62
B-2 An initial sketch of volume intersection 63
B-3 Camera model produced by cameraspitter.c: top boundary line is RGB 77ffbb; all other boundaries are RGB 880044 64

B-4 From left: a result of the silplane.c routine, with a camera; another result of silplane.c, from close up; the same result, from a distance. . 65
B-5 Result of the silcaster.c routine, with a camera and silplane. 65
B-6 OFF model and a camera VECT model, before and after bounding box matching (mergevect.c, off_boxer.c) 67
B-7 The final volume intersection diagram 67
B-8 Initial sketches of the use of imagelines and worldlines 67
B-9 Worldlines, with a camera and associated imagelines. 68
B-10 Left: Run-length encoded worldlines. Right: Their intersection. . . . 69

Chapter 1

Introduction

1.1 Motivation

Artificial intelligence researchers have long sought the ability to recover 3D information about an object from one or more camera views. The human brain does well with only two "cameras" but uses a collection of prior models, pattern matching neurons, and other cognitive machinery that researchers don't yet fully understand. Solutions to the problem of reconstruction can be divided into two groups: those that imitate a human-like visual system, and those that use more cameras but less modeling. The work described in this thesis is of the latter type, using no prior models but many cameras. The system uses a volume-intersection approach to produce rough 3D models, with an architecture designed to run in real time. Our system lays groundwork for systems that might use prior information, such as articulated models or adjacent frames, to improve real-time 3D data streams. These systems could eventually result in realistic, dynamically updating models that can be viewed from any angle.

For this project, the larger goal for these 3D models is the capability to synthesize novel views of a scene – I refer to this as "variable viewpoint reality" (Grimson and Viola, 1998), though it has also been called "virtualized reality" (Kanade, Narayanan and Rander, 1995). This capability would have applications in motion capture for movies and video games, in the sports and entertainment industry, in

video conferencing, and in human kinematics research, to name a few.

Some approaches to this problem do not involve creating a 3D model at all – images are merely manipulated in a way that produces a new view, without going to and from a model (Seitz and Dyer, 1996; Seitz and Dyer, 1995; Gu et al., 1999). This project aims to actually produce 3D models because the models provide a few key benefits: First, if the object being reconstructed is partially self-occluding, we will have to rely on prior models to generate a texture for the partially hidden area. The easiest way to apply that knowledge is to use a 3D model rather than an image-warping technique. Second, errors in our reconstruction process can be corrected from the standpoint of likely versus unlikely 3D models. For example, if reconstruction results in a person with three legs, we can use prior models of people to easily dismiss the model, and we can use prior frames of reconstructed video to determine which two legs ought to be there. Finally, we serve broader research interests by producing databases of 3D data (gesture recognition, human behavior, kinematics, etc.).

A number of research groups are working on 3D reconstruction, but few are working towards producing a real-time system. Until computation speeds up by orders of magnitude, exhaustive correspondence won't be feaible in real-time. Also, until bandwidth grows larger by orders of magnitude, there will be no way to move the image data necessary to support such a correspondence algorithm. Until that time, this system demonstrates a straightforward architecture for producing reasonable 3D models quickly.

1.2 System Overview

In this thesis, I describe a system for reconstructing non-textured three-dimensional models from multiple camera views.

1.2.1 Approach

The general strategy is to use a reasonably large number of cameras (twelve) capturing images at frame-rate (30fps) continuously. Several computers sit near those cameras. Each computer is connected to one or two cameras, for which it processes frames. The goal of the processing is to reduce each frame to a silhouette: to segment the foreground from the background in the image and, from then on, work with only one bit per pixel.

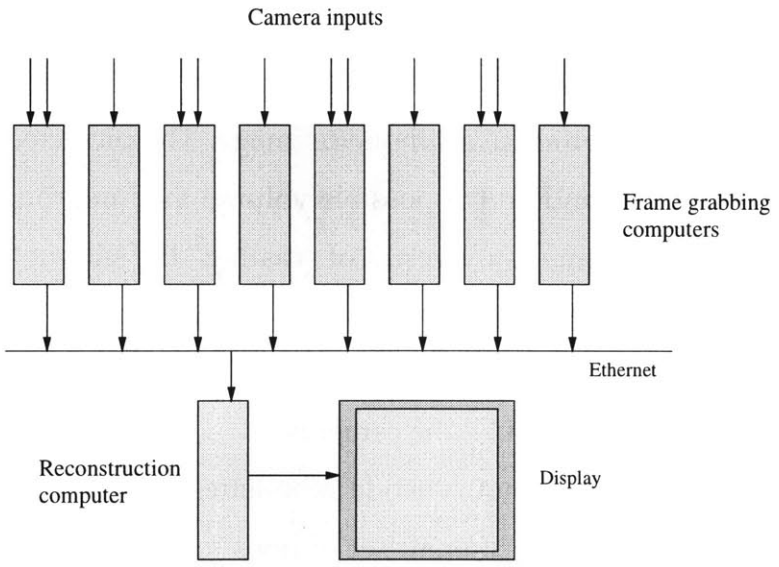

Figure 1-1: System flow diagram

After processing the frames, individual computers transmit processed images (silhouettes) to a central computer. Transmitting copious amounts of data would consume tremendous bandwidth if the images weren't compressed. Conversely, compression and decompression can be time-consuming tasks. Therefore, we compress the silhouettes in a quick, intelligent way. We take into account the geometry of the camera that captured that particular silhouette, and do a variant of run-length encoding based on that geometry. If used properly, run-length encoded silhouettes don't need to be decompressed in order to run volume intersection; we thereby save decompression time on the central computer by using a geometry-specific encoding, and save network bandwidth at the same time.

The central computer takes silhouettes from every camera and intersects the

volumes they represent to produce a rough 3D model – a visual hull (Laurentini, 1994). This entire networked procedure happens many times each second, as bandwidth and computation time allow.

1.2.2 Silhouette Intersection

Every point in a camera's image corresponds to a ray in the world; similarly, every bounded region in a camera's image corresponds to a bounded volume in the world – a cone with a ragged cross-section (see figure 1-2).

Once an image has been captured by a camera, we identify the foreground region in the image, and produce a silhouette image. The projected volume of the silhouette is an upper bound on the possible volume the foreground object could occupy. I refer to this volume as the result of "casting" the silhouette.

Silhouette intersection is based on the premise that the intersection of all silhouettes – as cast from their respective cameras – will produce a volume which bounds the object being viewed by the cameras.

For compression, we run-length encode silhouettes in directions parallel to vertical lines in space, yielding a different set of lines for encoding for each camera. Figure 1-3 shows several such possible encoding directions. That set of encoding directions, which I call "imagelines," can be determined in advance, and once computed, allows volume intersection to be done quickly at any chosen spatial resolution.

1.2.3 Parallelization for Realtime

The design for this system is aimed at producing a real-time system. Thus, computation is shared by several computers, each devoted to a specific task. In this architecture, only one computer has access to the segmented information from all cameras; the other computers only have information about the geometry and image content of the camera(s) for which they are grabbing frames.

This set-up trades bandwidth for the ability to add texture to models. To save

bandwidth, the actual content of the camera images is never transmitted over the network. The computer doing reconstruction therefore has no way to map a texture onto the generated models. It might be possible to send highly compressed image data over the network while allowing real-time computation, but this system is only designed to produce the 3D shape, not a fully textured model. Once we are consistently reconstructing, we may investigate texture mapping.

Since we use a geometry-specific run-length encoding, there are several mathematical operations that have to be performed in order to compress silhouettes properly. There are also 3D and 2D projection operations that have to be performed in order to intersect the volumes cast by the silhouettes. Knowledge of every camera's geometry allows many of those mathematical operations to be performed before using the system, so before reconstruction begins, we create lookup tables for those operations. This preprocessing saves computation time later, in exchange for a large chunk of memory on the central reconstruction computer.

1.2.4 Calibration

Our initial calibration scheme involved registering several points per camera whose real-world locations were known. Running a combination of gradient descent and other search algorithms, we arrived at a camera geometry, using a very basic camera model. The gradient descent found a minimum, but depending on the region of the image, the resultant calibration was sometimes several pixels off. When an arm or leg is only ten or twenty pixels wide, this becomes a serious problem.

Dan Snow began work with more complex camera models, involving radial distortion and other error correction parameters. Results from these models compare favorably to the results of the previous, simpler model. We continue to experiment with better and easier calibration methods.

1.2.5 Segmentation

Our segmentation strategy was essentially background subtraction, with morphological operators added to remove small errors. This is a minimally functional starting point, because we expect to eventually use some kind of modeling to improve the reconstructions, therefore high precision segmentation is a lower priority than getting *some* reconstruction running in real-time. A more complex strategy may be incorporated later.

1.2.6 My Contributions

For the bulk of this document, I use "we" to refer to some subset of Paul Viola, Dan Snow, and myself. Here, to clarify who did which work, I describe my specific contributions to the project over time.

Initial Experiments

When this project began, I started with a pre-existing dataset of images of an object (a replica of Tweety Bird) on a turntable, and undertook to reconstruct a three-dimensional model of Tweety. To do so, I converted the data to silhouettes by hand and, knowing the camera's positions, intersected the volumes cast by the silhouettes. This process produced a three-dimensional model. Tweety had been photographed in near-orthographic projection from 360 camera positions at one-degree increments along a circle. I used only 18 of those silhouettes, and obtained reasonable results; my goal was not to use a minimal or maximal number, but to use a reasonable number – to get a feeling for the kind of computation that is required.

I also used a very simple way of viewing the model: placing small cubes at all the boundary points. The result looks a little blocky, but the shape is fairly clear (see figure 1-4).

For that work, I used a voxel-based[1] representation of the data. I reconstructed the model one voxel at a time, checking whether each voxel corresponded to a

pixel that was "on" in all 18 silhouettes.

Improvements

To save memory, I run-length encoded the images in a horizontal direction before intersecting them. This technique negated the need to store each silhouette in its entirety; only the leftmost and rightmost bounds of the silhouette on every row of pixels were necessary. Now, as before, I looped through every voxel but then confirmed that the voxel's projection in every image fell between the stored left and right bounds. That algorithm was only slightly slower than before, but still the same order of growth, and the silhouettes consumed two orders of magnitude less memory.

One question that followed my work with the Tweety model is how best to encode camera images, in the more general cases of people and other objects, for both compression and reconstruction. A specific question I came up with for reconstruction was whether it could also undergo a drastic reduction in memory consumption. Our solution to both problems is to run-length encode the images in a direction that projects to vertical lines in the world – that way the reconstructed model can also be run-length encoded, achieving the desired memory savings.

Implementation

At that point, I developed a format and routine for quickly intersecting run-length encoded strips of volume, to be used at the reconstruction end of a system. Dan Snow joined the project, and together we devised a system architecture. We also divided the work of developing a system: Dan developed the parallelization and networking for the code, using MPI[2]; I focused on using the right encoding for each camera's image, given the camera's geometry. I also developed a camera

[1]A voxel is a (typically cube-shaped) element of space. Just as there are pixels in images, there are voxels in volumes. They can be iterated through, colored, given opacities, and otherwise manipulated.

[2]Message Passing Interface

model structure and a three-dimensional vector structure in C, along with useful function libraries to accompany them. After working on the parallelization, Dan began development of both camera calibration and image segmentation, with early help from me. When the system began coming together, I also worked on a graphic form of project documentation – movies.

Movies

To accompany our reconstruction system, I created a series of images, movies, and 3D models to explain how our system works. I used Geomview, a standard 3D model display program, to render the instructive models.

1.3 Thesis Outline

1.3.1 Background

In order to describe this system, I first discuss the general character of the problem in the context of research that others have done. In Chapter 2, I cover concepts that are fundamental to the decisions made in the system design our system, including voxels, volume intersection, multiple baseline stereo, and image morphing.

1.3.2 Imagelines and Worldlines

In Chapter 3, I describe our variant of volume intersection. I describe the run-length encoding we use for representing both silhouettes and 3D space, and I give a short analysis of the running time of such an algorithm.

1.3.3 Parallelization for Real-time

In Chapter 4, I discuss the architecture of cameras and computers we use to make real-time reconstruction possible. I also discuss the preprocessing we do in order to make the reconstruction itself less intensive.

1.3.4 Calibration and Segmentation

Camera calibration and image segmentation were handled primarily by Dan Snow. We encountered interesting issues in both areas, and both are integral to the system, so a brief discussion of each is included. Calibration is discussed in section 5.1, and segmentation is discussed in section 5.2.

1.3.5 System Performance

The system we built has yet to run in real-time as a whole, though parts of it have done so. In Chapter 6, I explain which pieces work, as of this writing, and how they will be integrated into a coherent whole. I also discuss the design of our network layer, and how we might be able to improve performance by changing it.

1.3.6 Conclusions

Finally, in Chapter 7, I take stock of all that we learned in this project, and look towards future directions of research. I discuss which avenues already have opened because of this project and which avenues are still too far off to approach.

1.3.7 Making Movies

In appendix B, I describe several of the programs written to produce demagogic aids, and I show some of the output from those programs. Illustrations that explain both volume intersection and the use of imagelines and worldlines are included: I start with the initial concept sketches, and I finish with images of the 3D models that replace those sketches.

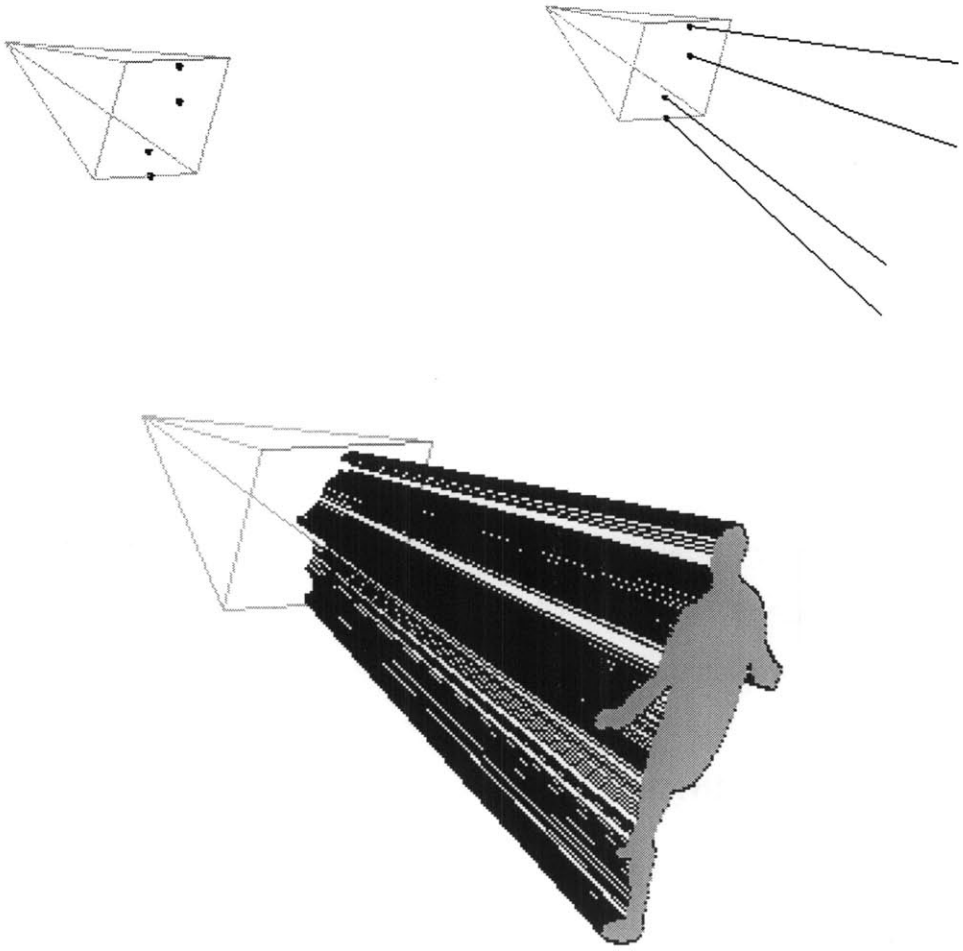

Figure 1-2: Clockwise from top left: Points on an image; Points on an image with segments of the corresponding lines through space; A region of an image and a section of the cone resulting from casting that image region. In this figure, cameras are represented by a small view frustum; the image is the rectangle.

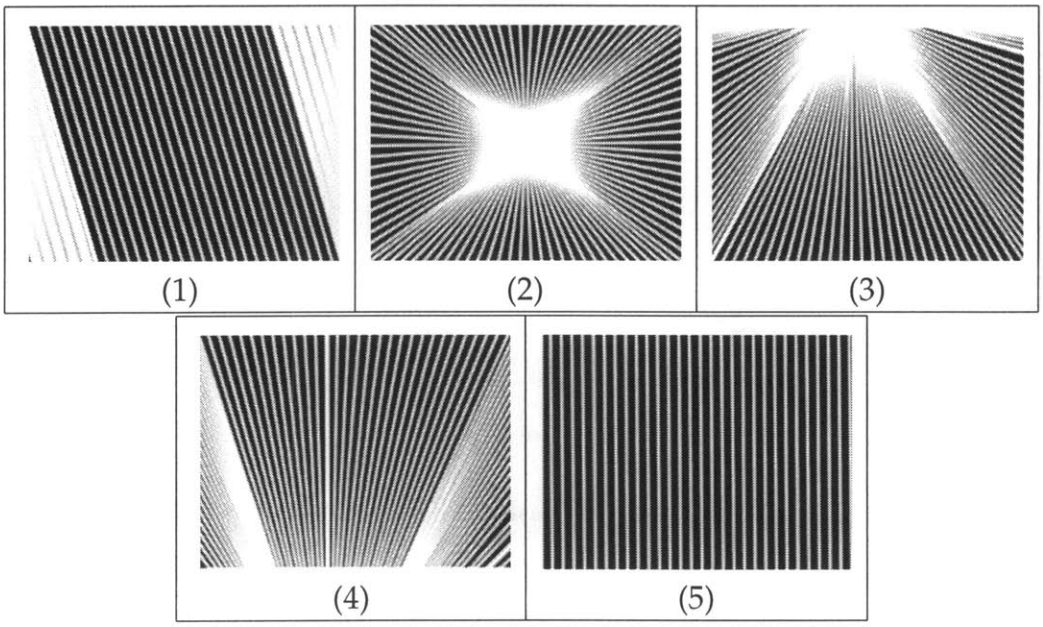

Figure 1-3: Imageline directions for various camera angles. From left to right: (1) A level camera, rolled slightly clockwise about the optical axis; (2) a camera looking straight down (or up); (3) a camera looking about $45°$ up from the horizontal, or an upside-down camera looking about $45°$ down from the horizontal; (4) a camera looking slightly down; (5) a level camera

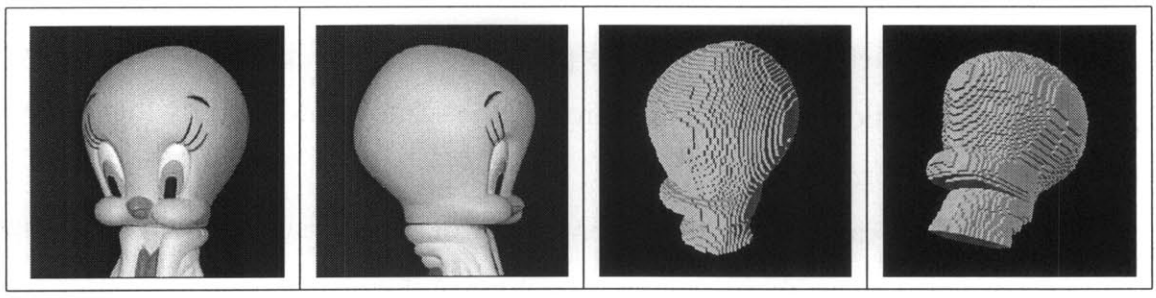

Figure 1-4: Left: two photos of a Tweety replica used for reconstruction; Right: two renderings of a crude model reconstructed from a sequence of 18 images, including the two at left.

Chapter 2

Background

In this chapter, I describe the character of the 3D reconstruction problem and analyze a few approaches to it, showing why we chose the one we did. I also discuss previous work in the field and its relation to this project.

2.1 Phenomenology of Reconstruction

Suppose we want to reconstruct an object in the world, using a discrete set of camera views. What do we know about that object? How can we relate the differences between camera images to the 3D properties of the object?

We will first make the assumption that an image grabbed by a camera is a perspective projection of a scene, or a known distortion of such a projection. We will also assume that each pixel's intensity is proportional to the light coming only from points along a particular optical ray through the scene, as defined by the perspective projection (Horn, 1986).

From different camera angles, the boundary of object's silhouette may change. The changes give clues about the outer contour of the object (Vijayakumar, Kriegman and Ponce, 1996; Zheng, 1994). We may use these silhouettes to obtain an upper bound on the volume occupied by the object, or we may use slight changes in the silhouette from camera to camera to determine the direction of the surface normal along the boundary (Vaillant and Faugeras, 1992). Given a possible set of

3D points occupied by the object, we could try to match the object to a 3D model (Besl and McKay, 1992; Dickinson, Pentland and Rosenfeld, 1992), or we could try to match a model to the silhouettes without ever rebuilding the 3D model (Kriegman and Ponce, 1990). A 3D model could be based on an analytic solution to the intersection of silhouettes, it could be based on voxels, or it could be based on splines (Sullivan and Ponce, 1996), to name a few possibilities.

From different camera angles, the colors of the object may also change, due to different colors on different parts of the object, or due to changes in lighting from different perspectives.

If we use changes due to the coloring of the object, we may be able to use a point-correspondence scheme to determine disparities of points in sets of images, and from that infer the distances from the cameras to points on the surface of the object (Rander, Narayanan and Kanade, 1996).

If we use changes due to the variation in illumination on the object, we may be able to find points of high and low curvature by studying how much specular refections linger on a particular part of the object (Horn, 1986; Koenderink, 1991); we may also use knowledge about the location of lights to infer surface normals, and from these, work towards determining the shape of the object. We could also use shadows of other objects (Bouguet and Perona, 1998), or special lights (lasers, etc.) to gather essentially topographic data, which can be assembled into a shape.

All of the methods mentioned so far assume the object is opaque; the possibility of transparency or translucency introduces more complexity in explaining the reasons for color variation from different views; the result is a problem in tomography (DeBonet and Viola, 1999).

2.2 Previous Work and Our Goal

This system is a single step on a path to achieving several larger goals. One goal is to study human motion; another is to build a system capable of reconstructing a real-world scene, such as a sporting event, in real time, while displaying a dynam-

ically updating model to users. In addition, users ought to be able to select any viewpoint they want, independent of other users, and independent of whether there is *physically* a camera where they want one in the real world (Grimson and Viola, 1998).

Real-time, brodcast quality camera calibration and planar patch rendering is already here (Crain, 1978), as demonstrated by the team at Princeton Video Imaging. They can place advertisements in video streams, as though they are part of the scene, for instance on the road or on a wall. This works even when the cameras move. However, it only works for a limited set of objects, and the 3D model is not transmitted over television; it is rendered first. The technology is on the way, though, so our goal is to work in that direction.

Kanade, et al. (Kanade, Narayanan and Rander, 1995; Rander, Narayanan and Kanade, 1996) at Carnegie Mellon University have developed a *multiple baseline stereo* approach, which reproduces good three-dimensional models of a scene from a collection of about fifty cameras. Their long-term goals are the same - reconstructing things like live sports events. The drawback is the processing time: multiple-baseline stereo is essentally a very large correspondence algorithm, which takes a very large amount of time to run. The setup used by Kanade, et al. is also not currently aimed at realtime, because all the cameras capture and save to files before any image processing is done; there is currently no network to support a real-time aspect of their reconstruction project.

With the end goal of creating new views of a real-world scene, it isn't always necessary to go to a three-dimensional model first; Seitz and Dyer (Seitz and Dyer, 1995; Seitz and Dyer, 1996) have modified traditional image morphing algorithms with a warping step, to produce physically valid views. The drawback is that while objects will follow physically valid paths (presuming correspondence is right), there is no such guarantee for light reflecting around the scene, especially in the case of specular highlights. Their approach is also dependent on correspondence, which can be a bottleneck for processing.

Gu, et al., (Gu et al., 1999) have tried taking a large number of silhouettes and

reducing them to a smaller number of representative silhouettes, from which the original set can be reproduced to a high fidelity. Culling silhouettes isn't particularly fast, and Gu typically started with hundreds of silhouettes; at least an order of magnitude more than we're prepared to handle right now (especially in light of the real-time goal).

2.2.1 Hidden Surfaces

In synthesizing a novel view, there are usually regions of the new view that one can be fairly confident about, and regions of almost no confidence.

In figure 2-1[1], there are three views of a teapot. From the views, one knows something about the texture over most of the exterior of the teapot. There are a few spots that remain mysterious, though, on the surfaces of the spout and handle that face towards the body of the teapot (see figure 2-2). These are surfaces that are hidden from all three views, and hence, some assumption would have to be made about their textures in order to synthesize some novel views of the teapot.

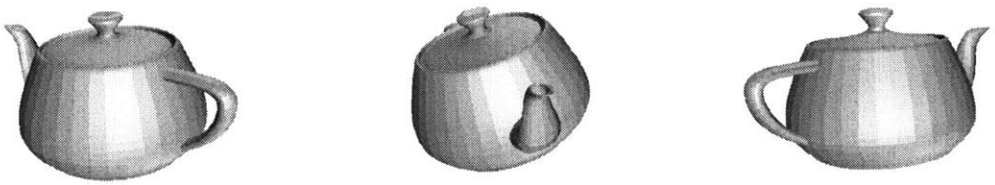

Figure 2-1: A teapot from three views.

Most algorithms, such as the one being developed by Gu, et al., try to render such areas by looking to a current image for help - but if we know that no image has that information (as might frequently be the case in a scene involving moving, self-occluding people[2]), it might be acceptable to refer to a prior model of the

[1]Teapot 3D model created by members of the Geomview team at the University of Minnesota.
[2]This scenario will arise much more for deformable objects than for rigid ones. For rigid objects,

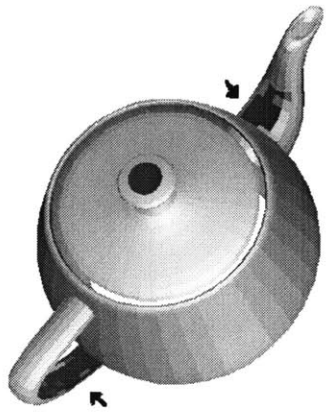

Figure 2-2: Parts of the teapot not visible from three views in figure 2-1.

situation for help in texturing this area. This sort of scenario leads us to try to produce a three-dimensional model of the object in the scene, so that priors can be constructed for the surface of that object. Priors would not be so straightforwardly applied if a view-morphing procedure were at work.

2.3 Geometry of the Algorithm

Having established that we want a 3D reconstruction, as opposed to a view morphing variant, We recognize that correspondence will be a time-consuming algorithm, no matter what the conditions; therefore, we adopted an approach that relies only on silhouettes. This could be modified to be corrected with correspondence information (Zheng, 1994), but the bulk of the processing will not be correspondence.

The calibration of cameras and segmentation of silhouettes I will leave for separate discussions in sections 5.1 and 5.2, respectively; for now, I will assume calibrated cameras (having solved for each camera's internal and external parame-

visible parts can't be self-occluded if there are enough cameras.

ters), and correct silhouettes.

2.3.1 Volume Intersection

As described in Chapter 1, casting a silhouette from a camera results in a volume, in particular a cone, where the object in the scene could be. By intersecting these volumes, we can find a rough volume occupied by the object.

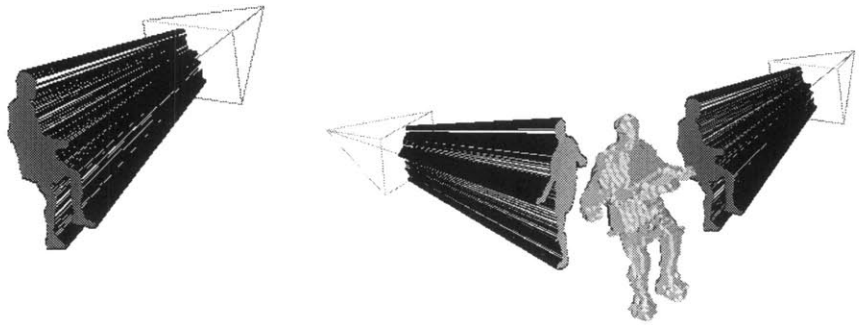

Figure 2-3: On the left, a volume cast form a silhouette; On the right, the intersection of four such volumes (only two of which are shown).

As we use more cameras, we will approach a limit of silhouette intersection, called the "visual hull." Laurentini (Laurentini, 1994) defines the visual hull relative to a set of views in the following way:

The visual hull $VH(S, R)$ of an object S relative to a viewing region[3] *R is a region of E^3 such that, for each point $P \in VH(S, R)$ and each viewpoint $V \in R$, the half-line starting at V and passing through P contains at least a point of S.*

I will also introduce the term "visual concavity," meaning a set of points that are on the surface of an object, but not on the surface of its visual hull. For example, in the left half of figure 2-4, the notch in the cube is a visual concavity. This is not a new concept; Zheng (Zheng, 1994) uses the term "occluding contour" to mean roughly the same thing.

[3] *A viewing region is an area from which an object can be viewed: a continuous set of possible views.*

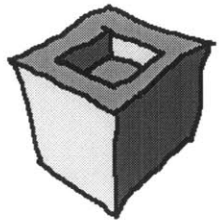

Figure 2-4: A cube with a piece removed, causing a visual concavity.

Repairing the Visual Hull

Using pixel-by-pixel correspondence, we could make iprovements to the visual hull method, because such an algorithm could "see" into many visual concavities, knowing to carve them out because of disparity information. Such a reconstruction algorithm would become more complicated than simple volume intersection, and processing time would greatly increase. In this system, we accept the visual hull as a reasonable model of the situation, since the objects we are interested in modeling are people, and people do not have too many visual concavities - people, for the most part, aren't concave.

If we used correspondence only for small regions of uncertainty, or used prior models from other frames in the video stream, that might be computationally efficient enough to still be real-time, while solving most concavity problems introduced by using volume intersection.

Volume Intersection: Analytic vs. Voxelated

Analytically intersecting volumes would mean that every planar slice cast from a camera (along the boundary of the cone cast from the camera), must be checked for intersection with all the planes cast from other cameras.

The efficiency of these intersections depends on the kind of object in the scene, as well as the spatial partitioning and culling schemes in use for example octrees. For this analysis, I assume octrees and similar culling schemes could be applied equally to all solutions, and so do not check their impact on algorithm running

time.

For this analysis, I also assume a relatively solid object in the scene, occupying a large part of the camera's field of view. For camera images x pixels by y pixels, this means a boundary on the order of $x + y$ pixels long[4]. Thus, $x + y$ boundary points yield $x + y$ planes being cast from each camera. With n cameras, this means order n^2 camera pairs, with order $(x + y)^2$ plane intersection checks per pair, for a net order of growth $O(n^2(x + y)^2)$.

As an alternative to the analytical approach, The voxel approach, at its simplest, checks, for every voxel, whether it is in the "object" region in every camera image. To do this at some spatial resolution, s (in voxels per dimension of a bounded volume), there will be $n \cdot s^3$ projections back to camera planes.

If a very high spatial resolution is the goal, the analytic solution is more appropriate. If only a modest spatial resolution is required, the voxel approach becomes reasonable. The voxel approach also has the advantage that the relevant constant-time operation, projecting back to the camera image, is computationally cheaper than the analytic constant-time operation, plane intersection.

Since we want only rough 3D models, for use in more sophisticated algorithms later in a pipeline, we chose the voxel approach, and found a way to increase its speed: essentially run-length encode the voxels in some direction. This speed increase brings the order of growth down to $O(n(s^2))$, better than the analytic $O(n^2(x + y)^2)$. In the next chapter, I describe that encoding strategy in detail.

[4]If the object weren't very solid (holes, wrinkles, noise, etc.), the number of boundary pixels could go up from $O(x + y)$ to $O(xy)$. For clarification of $O()$ notation, see (Cormen, Leiserson and Rivest, 1991).

Chapter 3

Imagelines and Worldlines

The key to making our system realtime is saving as much bandwidth and realtime computation as possible. To save bandwidth, we compress the images grabbed from the cameras before sending them over the network; I discuss this briefly in section 3.1. To save computation, we use a compression scheme that is efficient to decompress, and is immediately useful for 3D reconstruction. I discuss the development of this scheme in section 3.2, and I discuss the details of the scheme in section 3.3. Our encoding scheme leads to oversampling of silhouettes, but in most cases that oversampling is within reason. I discuss this oversampling in section 3.4.

3.1 Saving Bandwidth

Under available architectures, the cameras can't all be directly connected to one computer. As such, we need to transmit some of the information gathered by the cameras from computer to computer. This could be done by transmitting every pixel individually, but bandwidth is limited, so a more efficient scheme is preferable.

Because we make the assumption that the silhouettes will be of a largely solid (and largely convex) object, run length encoding is a reasonably efficient compression scheme. In figure 3-1, the image shown is at 320×240 resolution, for a total of 76800 pixels. By using a run-length encoding, this can be reduced to 1196 transi-

tions if encoded vertically, or 1100 transitions if encoded horizontally. Those transitions are usually one byte apiece, but implementations vary. Run-length encoding is a very effective compression: JPEG compression at 75% quality takes 4532 bytes, and GIF87 compression takes 1983 bytes. If the image were larger by a factor of n, the pixel count would grow as n^2, but the run-length code would only grow as n.

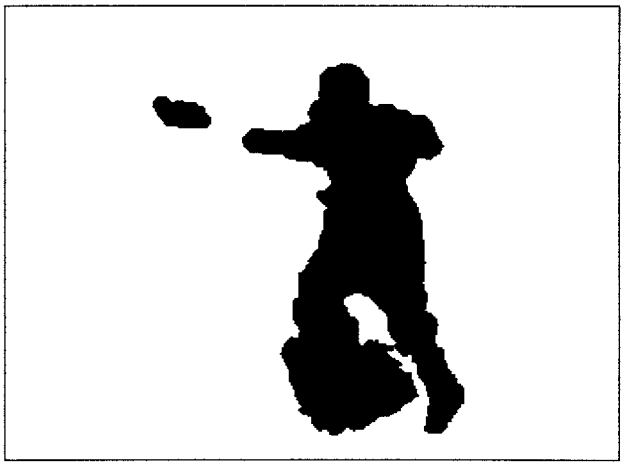

Figure 3-1: A possible silhouette of a person and a frisbee.

Given that we are using a run-length encoding, the question then arises, what direction to encode in. As stated, in this case horizontal was better than vertical, mostly because there were fewer horizontal runs than vertical runs. Here, I suggest we can make a computational savings, in addition to a bandwidth savings: rather than run length encode in a direction fixed relative to the images, why not run length encode in a direction fixed relative to the world?

3.2 Algorithm Development

I needed to save space when I was doing preliminary work on the Tweety model (see section 1.2.6), so I run-length encoded the Tweety silhouettes horizontally. I also began to think of ways to speed up the program, in addition to saving space. Rather than iterating through every voxel, I decided to run-length encode the voxels in addition, then just intersect run-length encodings to produce the 3D model. The question arose: What direction should I run-length encode the model? Assum-

ing the images are all in an orthographic projection, and all cameras are horizontal and level,[1] a horizontal run in each image would correspond to a planar slice of the object in the world.[2] The planar slice could be run-length encoded around a central point, thus run-length encoding the entire volume around a central axis.

In figure 3-2, the procedure is outlined. For each horizontal plane, every camera has a row of pixels; among those pixels, only some are on. The upper left of figure 3-2 illustrates one such plane; the upper right shows how, geometrically, images' runs would be intersected; the lower left displays the result of intersecting the runs; the lower right shows what a stack of such run-length encoded planes might look like: a 3D object.

Though it was never implemented, the cylindrical coordinate system in figure 3-2 would have been a reasonable way to encode the 3D model, at least in the case of the Tweety replica.

I also considered a spherical system, with all runs going through a central point in space; cameras' images would no longer be run-length encoded horizontally. Instead, the images would be encoded in a redundant way that makes it easy to intersect the runs in three dimensions. This new idea does not require all cameras to be level, and allows for perspective projection, so it seems much better for eventual use in real-world reconstruction, except for the oversampling both in the resultant 3D model and in every image. While having the finest possible resolution at the point of interest would be useful, oversampling does not yield a better model unless the cameras themselves are foveated or are somehow undersampled[3].

Since we weren't using any special variety of camera, and spherical coordi-

[1]Cameras could be horizontal and not level; they could be rolled about the optical axis.

[2]Actually, to get horizontal runs in the images to correspond to a planar slice of the object in the world, orthographic (or near-orthographic) projection is required only in the vertical direction; perspective projection is completely permissible in the horizontal direction, as seen in figure 3-2. Such a situation might arise in a panoramic slice of a wide angle lens, spanning many degrees horizontally but only a few degrees vertically.

[3]If we had cameras that could be more finely sampled at a moment's notice, i.e., dynamically foveating, this representation could be more interesting, because we could move around the point of oversampling to make it coincide with the region of interest in the scene. Utilizing this ability would be an interesting project in itself, allocating attention to the right area, but this project used only traditional cameras.

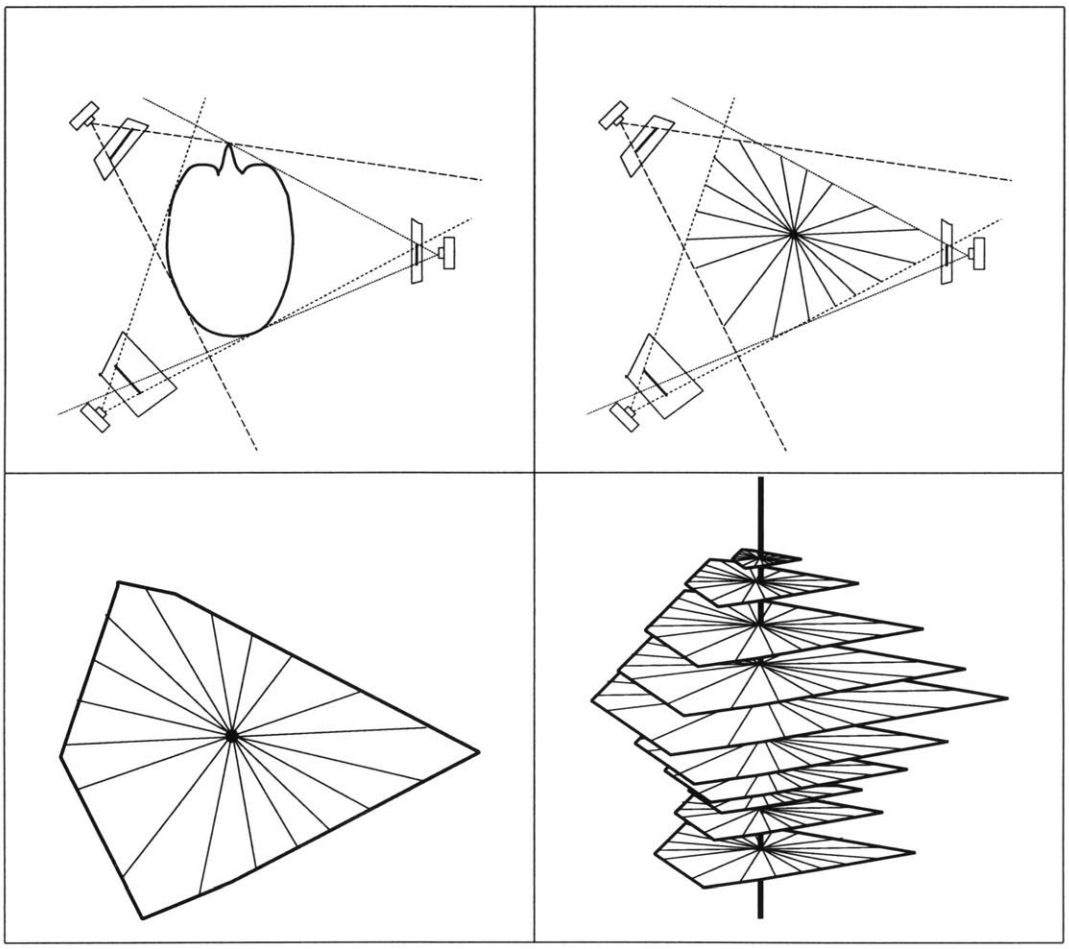

Figure 3-2: Illustrations of how a spatial run-length encoding might be built from a collection of horizontally run-length encoded images.

nates aren't the standard, another encoding scheme was preferable. A couple of discussions yielded the idea of encoding along vertical lines in the world: with this scheme the reconstruction is more position-independent than rotation-independent, and we use the more common Cartesian coordinate system.

For our actual reconstruction system, we chose run-length encoding in a direction on the image that would project to vertical lines in the world. In doing this, we encode in a set of directions that may vary across the image. If the direction the camera is looking has no vertical component (the camera is level), the directions will be parallel. If the camera is tilted, the directions will vary in a way that depends on the tilt (See figure 3-3). I refer to those lines of encoding, parallel to vertical planes in the world, as "imagelines."

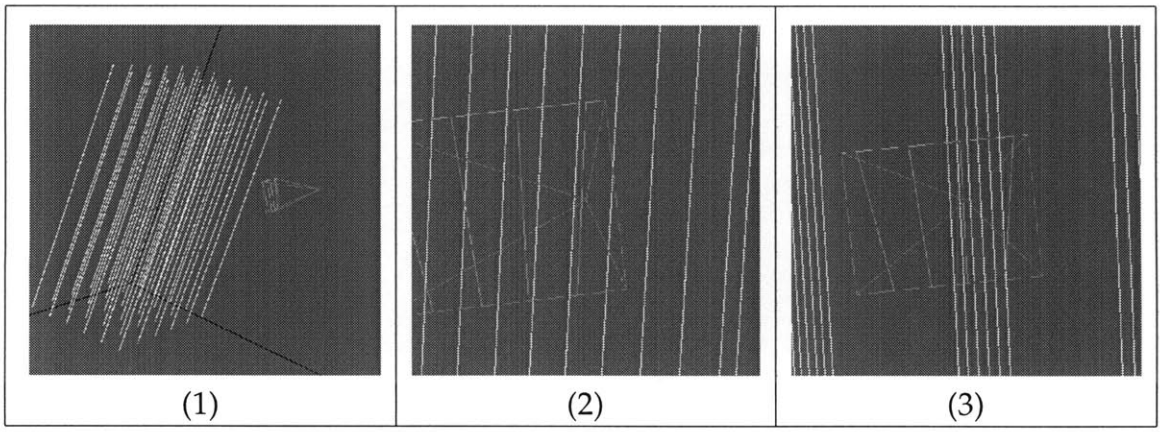

Figure 3-3: From left to right: (1) A camera, with some imagelines, looking at a set of worldlines; (2)the same camera, oriented to show a worldline corresponding to the rightmost imageline; (3) the same camera, oriented to show a worldline corresponding to the next imageline.

3.3 Saving Computation

If we encoded in a fixed direction relative to the image, as in usual run-length encoding schemes, the only easy way to access the silhouette data for reconstruction purposes would be to first write it out to an image: 76800 writes to memory, immediately. If the images were encoded in a fixed direction relative to the world, such as vertically in space, they wouldn't ever need to be decompressed from the run length encoding – the runs in each image could be reinterpreted as runs through the solid cone cast by the silhouette. This would be a run-length encoding of volume, which could be intersected with other cameras' run-length encodings of that same volume, at a low computational cost. I refer to vertical lines in space of run-length encoding as "worldlines." In figure 3-4, I show several cameras' view of a worldline, and in figure 3-5, I show the intersection of those views of the worldline – volume intersection along a worldline run.

3.4 Oversampling

Imagelines are not as efficient as a normal run length encoding, however. For cameras pointing in a mostly horizontal direction, the imageline encoding is nearly the

same as doing a vertical run-length encoding. For cameras pointing in a significantly vertical direction, however, there can be great inefficiency. Regions of the image that are near a vanishing point get oversampled; the closer to a vanishing point, the worse the oversampling.

In the best case, the imagelines run parallel to the longest dimension of the camera image; in the worst case, the worldlines' vanishing point is in the image, yielding a tremendous oversampling of the vanishing point. This is unlikely; it would take a camera pointed nearly straight up or down. Even in the worst case, most silhouettes would compress reasonably using imagelines. In figure 3-6, I show example good and bad cases for encoding with imagelines.

Though it has its drawbacks, imagelines seemed the right decision for this system; the oversampling problem was outweighed by the computational advantage of being able to run-length encode volumes. In Chapter 4, I discuss the details of implementing this system, using imagelines and worldlines as the reconstruction machinery.

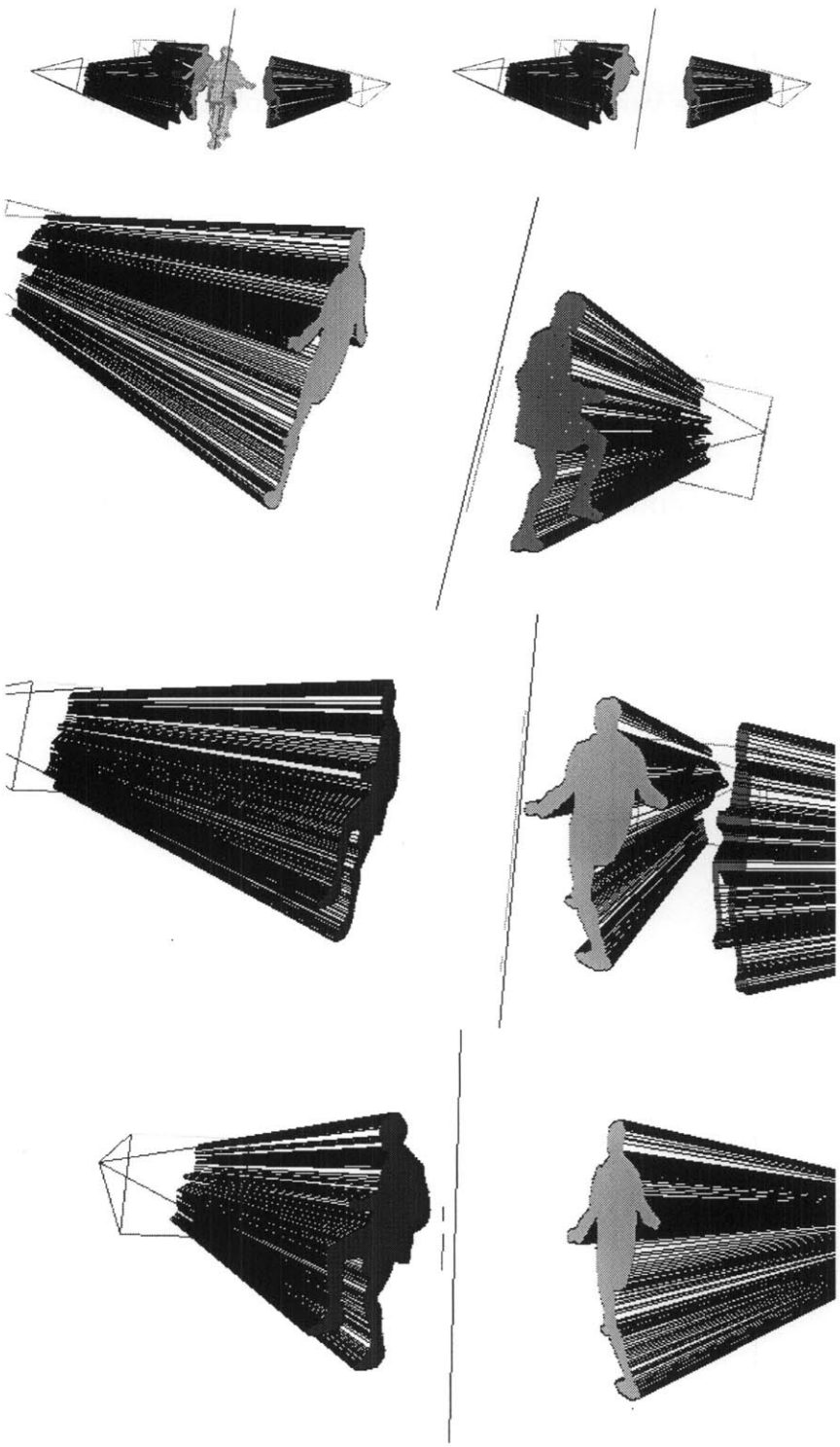

Figure 3-4: Top: three cameras' views of a person, with a worldline; the three cameras with just the worldline. Top middle: the worldline with the run from the right camera highlighted. Bottom middle: the worldline with the run from the center camera highlighted. Bottom: the worldline with the run from the left camera highlighted.

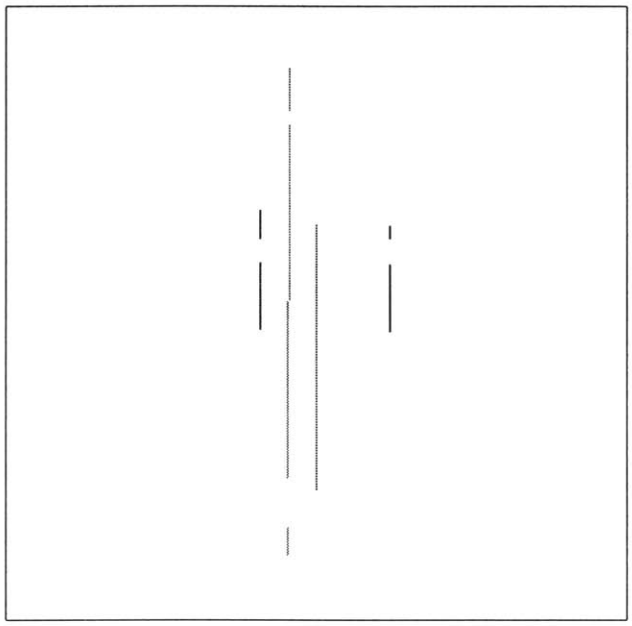

Figure 3-5: The three runs along a worldline from figure 3-4; at right, the intersection of those runs – the part that would remain after reconstruction.

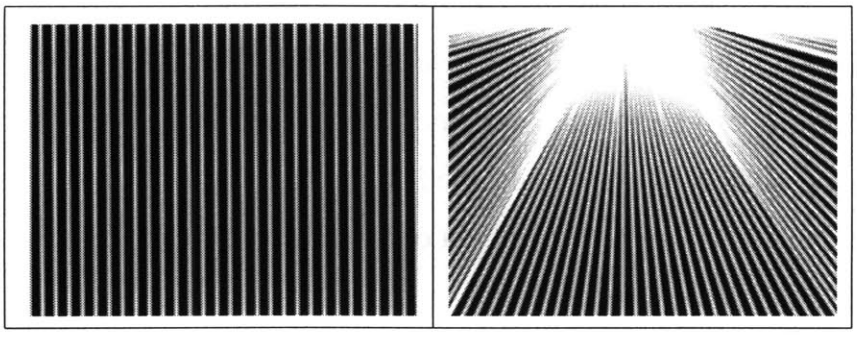

Figure 3-6: Nearly best and worst case scenarios for oversampling imagelines.

Chapter 4

Parallelization for Real-time

This project required designing a system architecture capable of supporting real-time reconstruction, taking into account the constraints provided by the available hardware. In section 4.1, I present the bulk of that architecture. In section 4.2, I discuss the devised accelerations for reducing computation time in the imageline/worldline volume intersection algorithm. The accelerations are based on caching information to convert mathematical operations to lookup operations.

4.1 Architecture

Most 3D reconstruction systems run off-line, opting for a highly accurate reconstruction at the expense of computation time. Kanade, et al., use the fifty cameras to store data synchronously, before doing any reconstruction (off-line, multiple-baseline stereo) (Rander, Narayanan and Kanade, 1996). They explicitly analyzed the problems associated with real-time 3D reconstruction and concluded that the bandwidth and memory requirements on such a system are beyond the realm of possibility given current technology.

From the beginning, however, our system has been designed to operate in real-time. We acknowledged that not all the video data could be put across a network

in real-time[1], so we aimed to reconstruct only non-textured models. By omitting that step, the actual image information doesn't need to be broadcast. Therefore, computers of some kind must sit between the cameras and the network. In addition, one more computer must remain on the receiving end, collecting the data from all cameras and intersecting it to produce 3D models.

4.1.1 Cameras

For our cameras, we chose an inexpensive solution, so we could purchase, position, and connect a lot of cameras to a network quickly. We used Intel's *Create & Share*[2] camera pack, which includes a frame grabber board with a Brooktree BT848 chip, and a small camera. The camera produces video at a 640×480 resolution, but the board has options to use either a 640×480 or 320×240 mode, so we captured using the latter to save bandwidth.

We found that we could reliably install and use two frame grabber boards in a Linux box[3]. With eight Linux boxes available for frame grabbing, we installed two boards in half of the computers and only one board in the other half, for a total of twelve cameras and boards ready to capture.

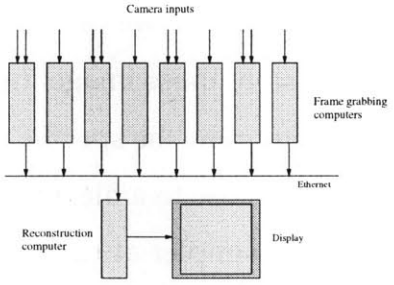

Figure 4-1: System flow diagram from chapter 1.

[1]At 640×480 resolution, a frame (in 32 bit mode) is 1,228,800 bytes, or 9,830,400 bits. At thirty frames per second, that's just under 3×10^8 bits per second. With several cameras, this requires more bandwidth than can be currently obtained over Ethernet (usually a maximum of 10^8 bps).

[2]*Create & Share* is a registered trademark of Intel corporation.

[3]Four boards have worked simultaneously, but that result has not been reliablly duplicated

4.1.2 Network Layer

With the hardware settled, we then had to transmit the data from the frame grabbing computers to the central reconstruction computer. We chose a protocol to use on top of the network layer, MPI (Message Passing Interface). We used an implementation called LAM (Local Area Multicomputer), developed at the Ohio Supercomputer Center and subsequently at Notre Dame. The interface is convenient, and provides what we need for moving data from computer to computer.

4.2 Non-Runtime Preprocessing

With the architecture in place, several aspects of the geometry could be cached in appropriate places around the system. Here, I describe exactly which pieces of information we cached in the system. We characterize that caching as non-runtime preprocessing because it can all be computed with only the cameras' parameters; it requires no silhouettes.

Imagelines

Given only the parameters of the cameras, we can determine where the imagelines should run for each camera. We store those imagelines in a file by their beginning and ending pixel. We can also determine the specific pixels that will be intersected on those imagelines; we also store those to a file, indexed by imageline. That way, stepping through the imageline is simpler at runtime. This can be used by both the Slave process, running on the individual frame grabbing computers, and the Master process, running on the central reconstruction computer.

Reconstruction Method

The reconstruction side of our algorithm entails iterating through a set of worldlines (instead of voxels) and, for each worldline, projecting all relevant[4] cameras' information to the worldline (see figures 3-4 and 3-5). Only one imageline in each relevant camera will be important, so only that one correct imageline must be projected. The projected imagelines are intersected along the worldline, which yields the strip of remaining volume along that worldline after volume intersection.

Reference Worldlines

In the absence of caching, our algorithm would entail projecting a pixel from the camera towards the worldline in question and taking the closest point on that worldline. This would have to be done for every transition along every imageline, for every frame of video. We didn't want to do projection and closest-point-on-a-worldline math for every frame, though, because we would be doing the same operations repeatedly, and those operations are involved.

Instead, we wanted to just look up the Z-coordinate along the worldline, given the worldline's coordinates and the pixel on the imageline. Such a lookup would completely bypass any math at runtime. This first plan involved create an array to cache all this information – something like **float ZCoordFromPixel [WC][WR][CAM][IML][PIX]**, holding a floating point number, given a worldline row, worldline column, camera, imageline, and pixel. That could be redundant (since some imagelines hit the same pixels), so the array was refined to be more efficient: **float ZCoordFromPixel [WC][WR][CAM][X][Y]**. The size of this array, however, would be unmanageable: **Y** from 0 to 239, **X** from 0 to 319, **CAM** from 0 to 11, **WR** and **WC** both from 0 to some desired spatial resolution, s. Not counting the worldlines, this array would have $921,600$ entries; for even a modest spatial resolution, for instance s = 30, the **ZCoordFromPixel** array would have more than 800 million entries – a

[4] A camera is relevant if it can see the worldline in question. If it faces the wrong way, it is not relevant for that iteration.

prohibitively large number.

Instead of this giant array, we decided to cache the same information in two steps. First, each imageline would be projected to a reference worldline.[5]Second, we would store the Z-coordinates of pixel-by-pixel projections along that reference line, but no others; this process yields an approximately $921,600$-entry array, as mentioned earlier. Unlike the earlier scheme, this is not necessary for every worldline. The only other array this method requires is one with a mapping from each worldline to its reference worldlines, relative to each camera. The following arrays were used in this project.

float ImagePixel[CAM][IML][PIX] is an array that stores the Z-coordinate along the reference worldlines of the projections of the pixels along the imagelines.

float RefDist[CAM][IML] is an array that stores the distance from a camera's optical center to the reference worldline for every imageline.

int WorldToImage[WR][WC][CAM] is an array that stores the imageline corresponding to each worldline for every camera. The integer stored is an index into an array of pre-stored imagelines.

float WorldDist[WR][WC][CAM] is an array of distances from every worldline to every camera's optical center. For a given worldline-camera pair, this distance is compared to an imageline-camera pair in **RefDist**, and the Z-coordinate information from **ImagePixel** is translated and scaled appropriately.

Once all this information is computed, it is stored in files, to be loaded at runtime. At runtime, once the system has loaded the above files, the iteration looks something like the pseudocode in figure 4-2.

[5]Which way we define a reference worldline is not key, but for our system, Dan Snow chose to use the worldline running through the beginning point of the imageline in the camera model.

```
For all worldlines WR, WC {
    For all relevant cameras CAM {
        Find imageline IML with WorldToImage[WR][WC][CAM];
        For the pixels of the run-length encoding along IML {
            Convert pixels to Z-coords with ImagePixel[IML][CAM][PIX];
            Adjust Z with RefDist[CAM][IML] and WorldDist[WR][WC][CAM];
        } /* run-length encoding */
        Store that set of Z-coordinates;
    } /* all relevant cameras */
    Intersect all sets of Z-coordinates;
} /* all worldlines */
Output 3D model, consisting of run-lenth encodings along worldlines.
```

Figure 4-2: Pseudocode description of our volume intersection algorithm

Chapter 5

Calibration and Segmentation

My work on this project did not focus on either camera calibration or image segmentation. However, they were integral parts of the project, so I discuss them briefly here.

5.1 Calibration

To achieve good reconstruction, the data from all cameras must be combined with a precise knowledge of the relationships among cameras[1]– what slice of space they provide. That knowledge can be absolute, relative, or even probabilistic. In the absolute case, the camera locations in space would be a certainty. In the relative case, one camera could be fixed and all the others would be calibrated relative to it. In the probabilistic case, one might generate joint probability distributions for the location of an object in space and the locations and parameters of cameras. That joint distribution could be used to make a probabilistic assessment of the scene, using prior probabilities from earlier frames in the video stream, and any knowledge about the object in the scene.

In section 5.1.1, I will briefly discuss the camera model we use; In section 5.1.2,

[1]Some systems, such as the one in (Vijayakumar, Kriegman and Ponce, 1996), don't initially have the camera's parameters. They do need them, though – they just use the same data for camera calibration that they use for 3D reconstruction, whilw we solve for the cameras' parameters ahead of time, using separate data

I will discuss specifics of calibrating to this model in our project. Cameras aren't perfect, though. There are several types of distortion that can be considered; I will discuss those in section 5.1.3. We continue to develop our calibration algorithms; I briefly discuss our current work in section 5.1.4.

5.1.1 Basic Camera Model

In addition to the location of the camera (three coordinates), there are four other parameters in the simplest camera model: The focal distance, two coordinates for viewing direction, and one coordinate for roll about that viewing direction. That gives the basic camera model seven parameters.

We used a redundant camera model for most of the code, though, because it had more immediate access to more of the valuable information. That camera model has five parameters, four of which are actually 3D vectors. The parameters are: position, direction of view (with focal distance), the "up" direction, the "right" direction, and the ratio of real world meters to camera pixels at the specified focal length.

This assumes the camera is running at some fixed resolution. Cameras may also have different numbers of pixels along various dimensions, or may have different resolution modes. We used cameras at a 320×240 frame size, but could have used varied resolutions at the frame grabber level. That would have to be accounted for in our camera model, so as yet we don't support varying resolutions, but we may eventually modify our model to handle that option.

5.1.2 Calibrating to the Basic Model

I wrote an early program that would calibrate a camera using a version of gradient descent based on *Numerical Recipies in C*'s "amoeba" algorithm (Press et al., 1992). For each camera, we would identify several (usually six) points in the world whose 3D location was known, and find correspondences with points in the image. We would use an initial estimate of the camera's position, based on triangulation using

a laser rangefinder, and use my modified gradient descent to go from there.

My algorithm was very slow, but achieved reasonable results. Usually after calibration, points were between zero and five pixels away from where they were predicted, an error that we attibuted to human measurement mistakes and slight distortions in the camera's lens (making perfect perspective projection a good approximation, but an invalid model).

5.1.3 Advanced Camera Models: Corrections

There are several well-known ways cameras typically differ from the ideal perspective-projection model (Stein, 1993). Dan Snow investigated these: in particular, uneven aspect ratios, off-center principal point[2], and radial distortion. He found our Intel cameras to have nearly centered principal points and nearly even aspect ratios; the most significant correction he made was for radial distortion.

After correcting for radial distortion, he modified my gradient descent code, accelerating it drastically, and got even better results for calibration.

5.1.4 Easier Calibration

The calibration scheme described above relied on correspondence, and was performed by hand. We didn't have different looking calibration points (fiducials), so correspondence was necessarily by hand. Dan Snow created a number of different calibration objects that we could set in the vision arena for calibrating cameras in a more automatic way, but we would like to improve on them. We are currently exploring easier ways of setting up calibration points and having the cameras reliably, automatically calibrate. Our arena is in the open, so there are frequently people walking by and monitors flickering; this makes it hard to do background subtraction. We also lean away from permanent calibration points because of our

[2]The principal point is the point on the imaging plane of the camera (or in the image produced by the camera) for which the corresponding optical ray is perpendicular to the imaging plane. If the principal point is not in the center of the image, the perspective projection model will have to be recentered; otherwise the world will appear skewed.

eventual goal: to be able to deploy this system at a sports arena or in a conference room, without having to build fiducials into the arena or room.

We are currently exploring movable controlled lights for calibration, making no permament calibration marks, and making background subtraction more reliable.

5.2 Segmentation

Our current segmentation scheme is simply intensity-based background subtraction, not even using the colors in the images. We take a background averaged over several frames, and use that for the duration of a reconstruction.

After background subtraction, there are typically speckles, in both predominately foreground and background areas. To clean this, we dilate and erode our model by several pixels, thereby removing holes in the foreground, but leaving most of the speckles on the background. We view this as acceptable, because if the speckles on the background are erroneous, then when they are intersected with the other camera views, they ought to diappear. We want to get an upper bound on the volume occupied by the object being viewed, and a dilation followed by an erosion satisfies those properties.

Chapter 6

System Performance

The system we constructed has several components, many of which now work in real-time, but the system as a whole has yet to be integrated. In this chapter, I describe exactly what is working, and what remains to be integrated. In sections 6.1, 6.2, and 6.3, I describe the three primary components of the system: slave, master, and display. For each component, I discuss their operation both as it is and as it eventually ought to be. I also discuss the network layer and simple improvements we might make to it in section 6.4. Finally, in section 6.5, I show some results from a non-real-time run of the system. I discuss the implications of those results in Chapter 7.

6.1 Slave

The final implementation of the slave ought to capture images, segment them, and transmit the resultant silhouettes, encoded along the imagelines, to the central reconstruction computer.

At present, we have not included the imageline encoding in the slave process; the current slave merely captures images when the master sends the signal, and writes silhouettes to a file. Those files can later be intersected.

6.2 Master

The final implementation of the master ought to signal the slaves to acquire silhouettes, receive their imageline-encoded data, and intersect the imagelines along worldlines, producing a continuously updating 3D model that is sent to the display.

At present, the master is broken into a capture phase and an intersection phase. In the capture phase, the master signals the slaves to capture video; it keeps the slaves synchronized, at least within 0.1 second. In the intersection phase, the master can be pointed at a set of files, and reads them in as silhouettes. It intersects those silhouettes, one voxel at a time; the current master does not use any variety of imagelines or worldlines to speed up reconstruction. As a consequence, it is fairly slow.

The current master also saves successive 3D models to successive files, rather than sending them to a display program. As it turns out, all that is required for continuously updating display (using Geomview[1]) is writing to a UNIX pipe, so that component of the master is done.

6.3 Display

The master and slave components are not yet integrated into a real-time system, so there has been no demand for a viewer that can continuously update models for display. In a brief investigation of display options, we found that Geomview can read input from a UNIX pipe. A pipe would allow one process to write geometric models as frequently as they were generated, while another process (Geomview) read the models out, and updated the display.

I experimented with this on a single machine, with one program iterating through 3D models and writing them to a pipe, while Geomview read from the pipe. It worked well, though the updating model was a relatively simple one. This could

[1]Geomview is described in the Glossary.

stand further testing, but it is my present belief that as soon as master and slave are able to continuously generate 3D models, the display will be trivial. We have only to pipe master's output to a locally mounted pipe, and have Geomview, running on the same machine, read it.

I make the distinction of a locally mounted pipe because most files are hosted through NFS. We wouldn't want to go back and forth between file servers in order to display our reconstructions, though; hence a local pipe.

6.4 Network

Instead of using the lab-wide Ethernet network, we are considering using a switched router. A switched router would allow us all the functionality we currently have, only we would not be competing with completely unrelated computers in the lab for bandwidth.

We haven't run our system in real-time yet, but when we do, it will consume a lot of the available bandwidth. For our sake, and for the sake of others in the lab, a switched router is the most likely course of action, but as yet, there has been no need for it.

6.5 Results

Over the course of an evening, we used the components of the system to reconstruct a person throwing a frisbee. We used the non-real-time slave processes to capture the images, segment them, and save the silhouettes; we then intersected the silhouettes with the non-real-time master process, without imagelines or worldlines, and rendered movies of the dynamically updating models using Geomview.

We only used six of the cameras, because we still had calibration problems, and for the sake of time constraints, it was easier to calibrate only six of the cameras. The reconstruction is reasonable, though it reflects the positioning of those

six cameras (see figure 6-2).

The four legs in figure 6-1 arise from the positioning of the cameras, as shown in the right half of the figure. The reconstruction is accurate, though. We used a spatial resolution of one voxel per inch: eighty inches tall, forty on a side. Dan produced the 3D model from those $256,000$ voxels using a Marching Cubes algorithm (Lorensen and Cline, 1987). Because the reconstruction volume was only forty inches on a side, the body in figure 6-2 is truncated, but that is not a limitation of the system, only an artifact of that particular run.

On the whole, the results are promising. They show that this system is viable, and that the components all do what they ought to; all that remains is to integrate the components and streamline them for real-time.

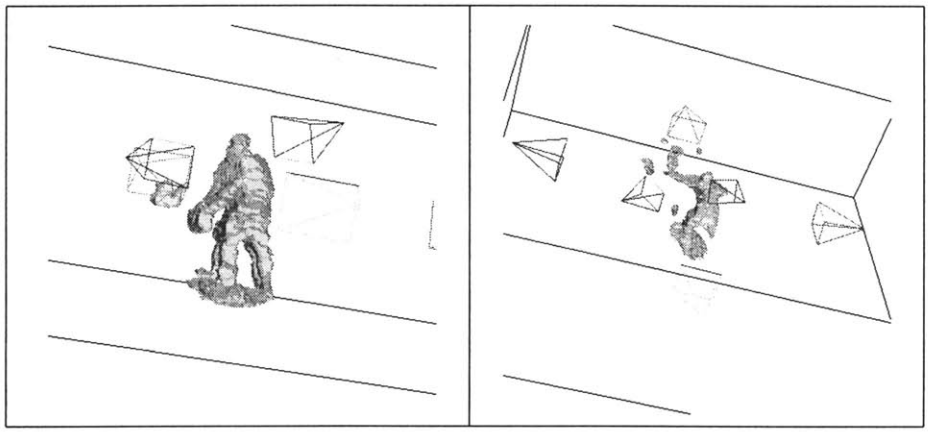

Figure 6-1: Left: A reconstruction with four legs. Right: The reason for the four legs – only four cameras, at redundant positions, can see the legs; the other two cameras are too high.

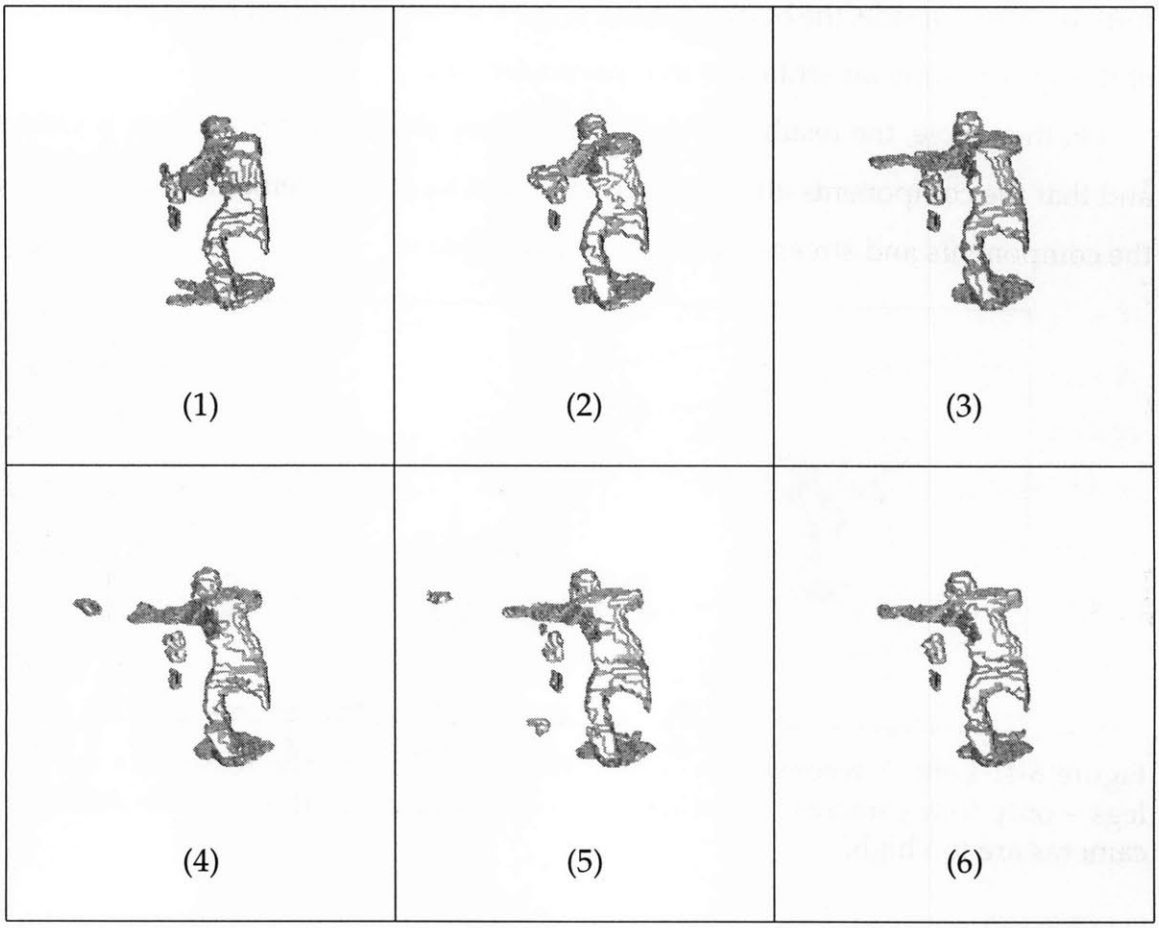

Figure 6-2: Six frames from the frisbee reconstruction. In frames 4 and 5, the in-flight frisbee is actually distinguishable.

Chapter 7

Conclusions

The system described in this thesis has performed well in a non-real-time run, as shown in section 6.5. During that run, data was captured at about eight frames per second, and the reconstruction looked good. In the process of developing that system, we learned a fair amount firsthand about the problems involved in reconstruction; I discuss this in section 7.1. We also found opportunities to improve the system that are research projects unto themselves; I discuss these in section 7.2. Finally, this work opens up opportunities in a variety of different areas; in section 7.3, I conclude by discussing the possibilities for continuing research and engineering beyond this project.

7.1 Difficulties

7.1.1 Efficient Use of Bandwidth

We knew we couldn't transmit the entire color images captured by the cameras across the network all the time; the trick was to send as much as possible, for the best return. Right now, we send no color information at all over the net, opting only to send silhouettes. For real-time texture mapping, we would probably have to send some color information (prior models would need to be corrected); a good way to do that has yet to be determined.

7.1.2 Calibration

Camera calibration remains an interesting yet problematic branch of our project. We hoped that calibration would be relatively painless, but we eventually came to realize that we had set ourselves with a difficult calibration task: We have an uncontrolled environment, we aren't making many permanent markings in the room, and we demand speed from our calibration system. We set ourselves with a challenge, and we continue to work on answers to that challenge.

7.1.3 Camera Parameters

We initially used a very basic camera model, in the hopes that any small deviation from that model would be too insignificant to matter for reconstruction of large objects. However, if a feature is only ten or twenty pixels wide, every pixel's precision counts. We learned which additional camera parameters were important, and focused on radial distortion; that exploration was instructive.

7.1.4 Camera Quality

For high speed motion, even and odd rows of an image were not aligned. This is because NTSC video is interlaced, so even and odd rows are usually sampled $\frac{1}{30}$ of a second apart (Rander, Narayanan and Kanade, 1996). For high speed motion, this will create jagged reconstructions. Another possible problem with our current cameras is automatic gain control (AGC). This will cause cameras to slightly rebalance the intensity level of an image if a particularly bright or dark object comes into view.

Since continuous, real-time reconstruction entails an image with objects coming in and out of view, this may turn out to be a problem. At the least, we would have to adjust our segmentation algorithm to look for AGC effects, and compensate for them during background subtraction.

A final issue with the cameras we currently use is that they are not precisely synchronized. This, too, will be a problem for reconstruction of high-speed motion.

They are within $\frac{1}{10}$ second now, based solely on the MPI calls, so we can reconstruct most slow-moving things just fine; even a slow enough frisbee (as in figure ??) works alright.

Eventually, we may need to use more precisely controllable cameras, but for now, we are making do with what we already have.

7.2 Improvements

7.2.1 Integration for Real-Time

Since the system is not yet fully assembled, as detailed in Chapter 6, the first improvement to be made to the system will be completing the integration of the components, and running the system continuously. Streamlining and generally cleaning up the code would also come immediately after we achieve continuous reconstruction.

7.2.2 Multiple-Camera Segmentation

Our segmentation scheme, at present, relies on information from only one camera at a time. Another camera's information might be useful in an example like the following[1]:

If a person with a white shirt stood in front of a background with a white mark on it, then the camera seeing that view might think there was a hole in the person's silhouette. If second camera was able to say that the shirt was supposed to be white (the same color as that piece of the background, then the first camera's hole-in-the-shirt theory could be disregarded, thereby repairing the reconstructed model.

To implement this in our current system, we would have to think carefully about how to manage bandwidth, and still negotiate enough between cameras to get a benefit.

[1]This idea was first brought up in a discussion with John Winn, Dan Snow, and myself.

7.2.3 Adding Limited Correspondence

A different use for color information would be to identify areas where there are likely to be visual concavities, and use correspondence (stereo of some kind) within those areas to determine a more precise object shape. By limiting the correspondence to a small area, the computational intensity and bandwidth requirements of the problem might be reduced enough to become tractable in real-time.

Another option would be to do a complete correspondence on some initial frames of video, along with a high quality texture mapping. The result could be used as a prior model for succesive frames of video; it could be articulated or distorted to try and match the colume information coming from those succeeding frames.[2]This idea originated in a discussion with Dan Snow.

7.3 The Future

Once we begin generating streams of real-time data, a number of options will be open. Here, I list a few:

We will be able to explore texture mapping, as a general topic, once we have long temporal sequences of image and 3D data that can be compared. Adjacent frames of 2D and 3D data may provide useful information for texture, and with such new data sets, such a possibility will be easier to investigate.

We may also study patterns in the dynamics of human motion. We could track limbs, center-of-mass and other characteristics, in order to identify individual habits, or form concise descriptions of frequent behavior modes.

We could use the 3D data as the equivalent of mouse information, to control some application, making a more natural human-computer interface.

Finally, we could use the reconstructions as exactly what they are, and use them to record sports events or video conferences, replaying them from any viewing angle imaginable.

The possibilities are not limited to those I mention here, but these are some of

the options we are likely to pursue, as we further the development of our real-time reconstruction system.

Appendix A

Glossary

AGC Automatic Gain Control, a feature on CCD some cameras that automatically adjusts to the brightness levels in the environment.

Bps Bits per second.

Fps Frames per second.

Frame-rate The standard rate at which frames are captured for full-motion video. In this thesis, I refer to the NTSC standard rate, 30 interlaced frames per second.

Frustum A section of a pyramid. See also *View frustum.*

Geomview A program developed at the University of Minnesota's Geometry Center. The program renders 3D models from several file formats, including "OFF," a format for polyhedral models, and "VECT," a format for vector graphics.

Imageline Lines of encoding an image that lie parallel to the projections, in that image, of vertical lines in the world. A camera's imagelines depend on the orientation of the camera relative to the vertical. See figure A-1.

LAM Local Area Multicomputer. An implementation of the MPI standard, developed first at the Ohio Supercomputer Center (http://www.osc.edu), now

the three cameras figure from ImlWrl would be good here.

Figure A-1: *get caption from the ImlWrl chapter*

furthered at Notre Dame's Laboratory for Scientific Computing (http://www.lsc.nd.edu).

MPI Message Passing Interface. A standard for message passing libraries developed between 1992 and 1994; We used an implementation of this standard, called LAM.

Occluded Not viewable from one or more cameras. I use the term loosely; *partially occluded* I usually use to mean that part of an object is not visible from any camera angle.

Optical axis The optical axis is an imaginary line passing horizontally through the center of a compound lens system. It goes the direction the camera is "looking."

Optical center The place in a camera where all light rays cross. In a pinhole camera, the optical center is the pinhole; in a CCD camera, it is usually in the center of the lens.

Optical ray The ray extending from the optical center of a camera through a particular point in the camera's field of view. Not the same as the optical axis, though the optical axis is one possible optical ray.

Principal point The point on the imaging plane of the camera (or in the image produced by the camera) for which the corresponding optical ray is perpendicular to the imaging plane.

Run-length encoding A method of compression that stores data consisting of repeating values by encoding the value, with the number of times it is repeated – the length of the run.

Segmentation The process of deciding whether pixels in an image are foreground or background; deciding whether they are a part of the object we care about.

The result of segmentation is a binary image – each pixel is either in or out. For this reason, segmentation is sometimes called *binarization*.

Tomography A method of producing a 3D model of interior structures of an object (such as the human body or the earth). Tomography uses differences in the transmission of energy waves (such as X-rays) through the object from different angles to reocnstruct the interior structure of the object.

View frustum Volume of space visible to a camera, forming a pyramid. The frustum is that pyrimidal volume constrained by front and back clipping planes.

Visual concavity A set of points that are on the surface of an object, but not on the surface of its visual hull. For example, the notch in figure A-2 is a visual concavity; no silhouette will reveal its presence. Essentially the same as what Zheng (Zheng, 1994) refers to as an "occluding contour." See also section 2.3.1.

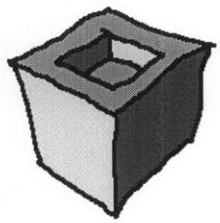

Figure A-2: A cube with a piece removed, causing a visual concavity.

Visual hull The maximal volume that can be substituted for an object without affecting any silhouette of that object. Also, the closest approximation of the object that may be obtained through volume intersection. (adapted from (Laurentini, 1994).) Defined by Laurentini (Laurentini, 1994), *The visual hull $VH(S, R)$ of an object S relative to a viewing region*[1] *R is a region of E^3 such that, for each point $P \in VH(S, R)$ and each viewpoint $V \in R$, the half-line starting at V and passing through P contains at least a point of S.* See also section 2.3.1.

[1] *A viewing region is an area from which an object can be viewed: a continuous set of possible views.*

Voxel A voxel is a (typically cube-shaped) element of space. Just as there are pixels in images, there are voxels in volumes. They can be iterated through, colored, given opacities, etc.

Worldline Vertical lines in space for run-length encoding of volume. For every camera, each worldline corresponds to only one imageline; for each imageline, however, there are many corresponding worldlines.

Appendix B

Making Movies

In the course of building the prototype for the 3D reconstruction system, we needed to create visual documentation of our algorithm for demonstrations and talks. This initially took the form of sketches on paper or markerboard (see figure B-1) but eventually required more precise diagrams.

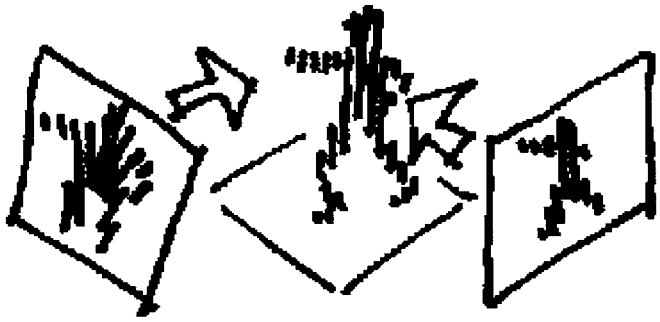

Figure B-1: An initial sketch of the reconstruction procedure

With the exception of a system diagram (as in Chapter 4), this kind of documentation was solely 3D modeling – sometimes images of 3D models, sometimes movies of those models changing. Our platform for displaying models was Geomview, so I worked within that framework to create a set of programs that would produce various instructive 3D models. Here, I outline the basic visual demonstrations we deemed useful, and I describe the programs that produced those demon-

strations.

B.1 Volume Intersection: Cookie-Cutting

The first diagram I produced explained the concept of volume intersection: By taking the intersection of silhouettes from multiple camera views, one can form a rough 3D model – the more cameras, the better constrained the model. At the time of producing this diagram, 3D reconstruction was already working, and as a consequence we had already produced a sequence of camera models, captured images, associated silhouettes, and resultant 3D models. The next step was to integrate this real data into a format that explained the relationships between the different kinds of data in a visual way (see figure B-2 for the initial sketch of this diagram).

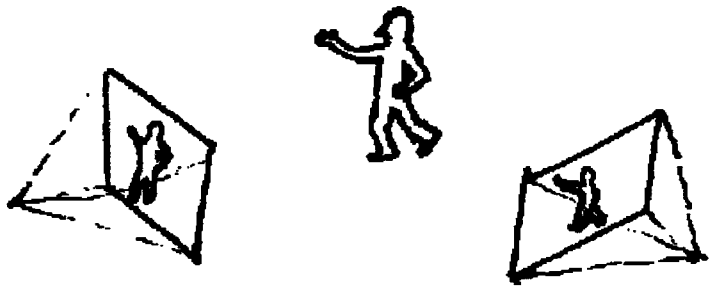

Figure B-2: An initial sketch of volume intersection

B.1.1 cameraspitter.c

I had to create a way to easily visualize a camera; I chose to use a small *view frustum*: the camera's optical center, with a small projection screen in front of it. The screen was the correct distance away, size, and orientation to convey the camera's field of view and orientation. The different color of the top boundary of the view frustum indicated the up direction in the camera's view;

That single boundaty was the inverse of the color of the other boundar lines

of the frustum. For example, if the other lines were RGB ff2299, the top would be 00dd66. See figure B-3 for an example of such a camera model.

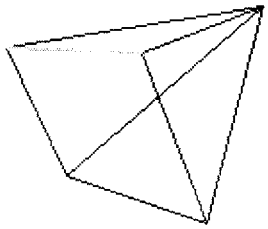

Figure B-3: Camera model produced by cameraspitter.c: top boundary line is RGB 77ffbb; all other boundaries are RGB 880044

B.1.2 silplane.c

Silhouettes from a particular camera's perspective also had to be available, so that they could be positioned in space to show their different perspectives on the object being reconstructed. In silplane.c, silhouettes can be color-coded, and the user can determine how far to place the silhouette from the camera model's optical center. The silhouette is drawn as a set of points in space, so that, if it is viewed from a distance, it appears solid, but if viewed closely, appears translucent (see figure B-4). A set of points was also a very easy representation for me to use when converting from an image file to a silhouette; no texture mapping or complicated geometry was necessary.

B.1.3 silcaster.c

I felt it would help to have lines showing the projection direction as a silhouette is cast out to fill the appropriate cone in space. I wrote a routine to do this, which fits well with silplane.c (figure B-5). The routine goes through the silhouette image, deletes all but the boundary points on the silhouette, and creates optical rays

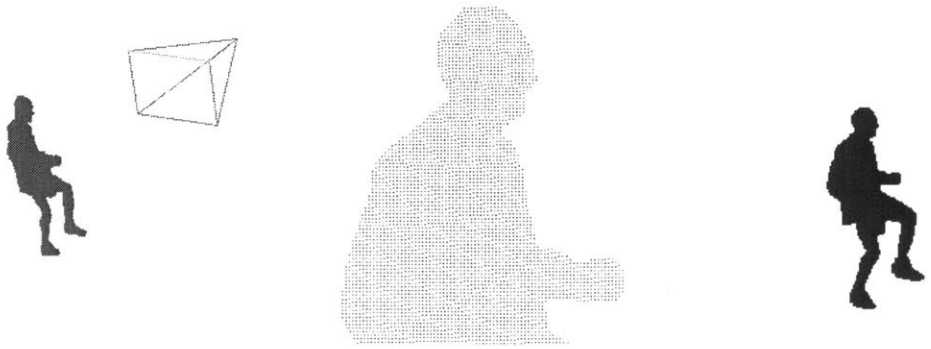

Figure B-4: From left: a result of the silplane.c routine, with a camera; another result of silplane.c, from close up; the same result, from a distance.

corresponding to those boundary points. The user is given options of selecting a starting and finishing distance from the camera's optical center, in addition to the preferred color of the rays.

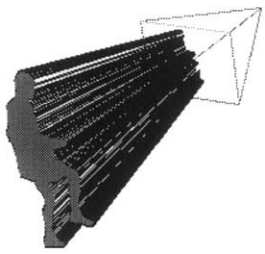

Figure B-5: Result of the silcaster.c routine, with a camera and silplane.

B.1.4 mergevect.c and off_boxer.c

Finally, to combine the elements (the camera, the model, the silhouette, etc.), they needed to have the same bounding boxes, so that Geomview would place them correctly relative to one another in the viewer. Geomview's default is to take any object description, find a bounding box along X, Y, and Z axes, and rescale that box so it fills the screen. This is a problem if the models are actually *supposed* to be different sizes and different locations; the solution I use is to add two inconspicuous

points to every model, so that all models have the same bounding box and thus appear in the correct positions relative to one another.

Most of the files I generated, as with the above routines, were VECT files. The VECT format is used by Geomview strictly for vector graphics – lines and points in space, but no polygons. I wrote mergevect.c to take any two VECT files and combine them. That way, if I created a file with only the desired bounding box points, I could merge all other files with it, and they would align properly.

As an alternative to creating a universal bounding box, all the objects (cameras, silhouettes, etc.) could have been merged together into one file, thus correctly aligning them with respect to one another. That approach has two disadvantages, however: First, the properties of any particular object in the scene cannot be altered independent of other objects, since the individual objects would now appear to Geomview as one big object. Second, this merging only works within a file format, like VECT. To view a VECT file and an OFF[1] file at the same time, in the same frame of reference, the objects described by those files need to be given the same bounding box. See figure B-6 to see the results of displaying objects simultaneously with and without the same bounding boxes.

I wrote off_boxer.c to put bounding boxes into OFF files, giving final 3D models the same bounding boxes as camera models and silhouettes. In this way, it complements the function mergevect.c provides for VECT files, at least for bounding boxes. I did not write a *mergeoff.c* program, because we never needed to view multiple reconstructed models at the same time. This program may be added to the suite in the future, though.

B.1.5 Final volume intersection diagram

When all elements were combined, the result was, as anticipated, instructive and accurate (figure B-7).

[1]The OFF format is used by Geomview for polyhedral objects. This is especially useful for the 3D models that result from reconstruction.

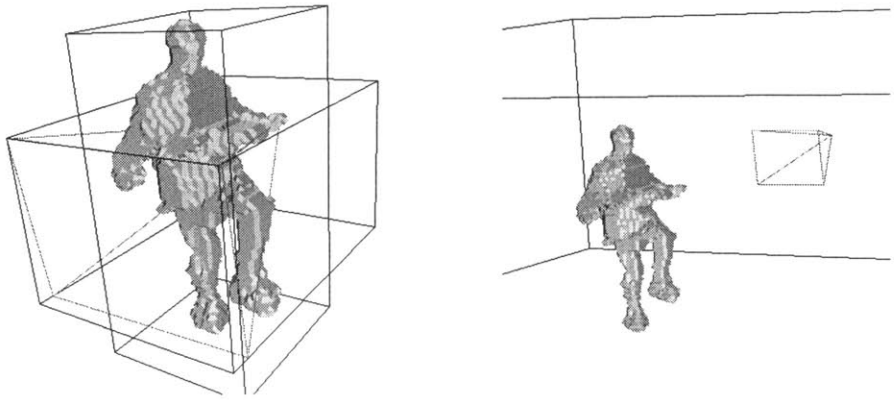

Figure B-6: OFF model and a camera VECT model, before and after bounding box matching (mergevect.c, off_boxer.c)

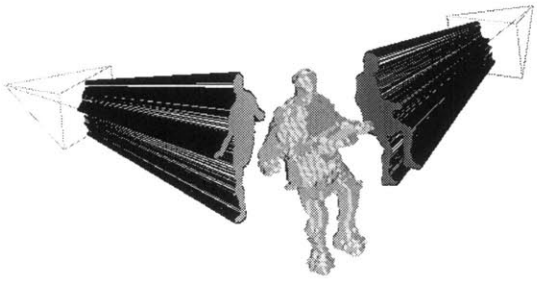

Figure B-7: The final volume intersection diagram

B.2 Imagelines and Worldlines

Figure B-7 shows the general idea of volume intersection. We also needed to illustrate the details of using imagelines and worldlines to carry out that volume intersection, as described in Chapter 3.

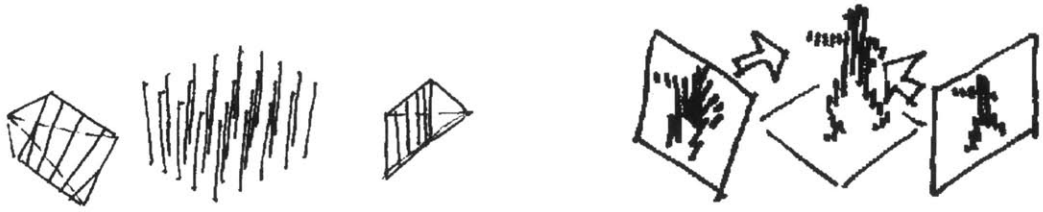

Figure B-8: Initial sketches of the use of imagelines and worldlines

B.3 Imagelines and Worldlines

In order to show the relationships between imagelines and worldlines, I wrote several programs to render worldlines, produce corresponding imagelines, and do volume intersection using worldlines. Here, I briefly describe those programs.

B.3.1 worldlinemaker.c

From user input of corner grid coordinates and a spatial resolution, this program generates a set of worldlines in space, as shown in figure B-9.

B.3.2 imagefromworld.c

From user input of a camera model and worldline end coodinates, this program generates the associated imageline. When coupled with **solidline.c**, a line drawing program, this can put imagelines on a camera plane, as shown in figure B-9.

B.3.3 sectImlTosectWrl.c and sectImlTosect.c

From user input of an imageline, a camera model, and a silhouette to encode, these programs encode the silhouette along the imageline, and project that encoding onto a user-specified worldline. Using **sect.c**, such projections can be intersected, producing a visual deomstration of run-length encoded volume intersection, as in figure B-10.

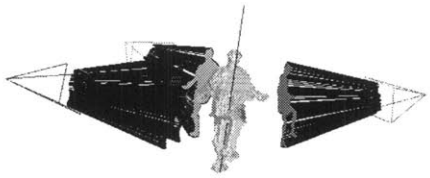

Figure B-9: Worldlines, with a camera and associated imagelines.

Figure B-10: Left: Run-length encoded worldlines. Right: Their intersection.

B.4 Future Work

The suite of programs I developed has been useful, but right now file manipulation is an essential and tedious part of the process. If this suite is developed further, a high-level utility for tracking, naming, and manipulating these files would save users time, and make the process of movie making easier.

References

Besl, P. J. and McKay, N. D. (1992). A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Machine Intell.*, 14:239–256.

Bouguet, J.-Y. and Perona, P. (1998). 3d photography on your desk. In *ICCV 98 Proceedings*.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1991). *Introduction to Algorithms*. MIT Press, Cambridge, Mass.

Crain, D. W. (1978). Us4084184: Tv object locator and image identifier. U.S. Patent US4084184, U.S. Patent and Technology Office.

DeBonet, J. S. and Viola, P. (1999). Roxels: Responsibility weighted 3d volume reconstruction. In *Proceedings of ICCV*.

Dickinson, S. J., Pentland, A. P., and Rosenfeld, A. (1992). 3-d shape recovery using distributed aspect matching. *IEEE Trans. Pattern Anal. Machine Intell.*, 14:174–198.

Dyer, C. R. (1998). Image-based visualization from widely-separated views. In *Proc. Image Understanding Workshop*, pages 101–105.

Grimson, W. E. L. and Viola, P. (1998). Immersive sporting events. Grant proposal to n.t.t., Massachusetts Institute of Technology.

Gu, X., Gortler, S., Hoppe, H., McMillan, L., Brown, B., and Stone, A. (1999). Silhouette mapping. Computer Science Technical Report TR-1-99, Harvard University.

Horn, B. K. P. (1986). *Robot Vision*. MIT Press, Cambridge, Mass.

Kanade, T., Narayanan, P. J., and Rander, P. W. (1995). Virtualized reality: Concepts and early results. In *IEEE Workshop on the Representation of Visual Scenes*, Boston, MA. IEEE.

Koenderink, J. (1991). *Solid Shape.* MIT Press, Cambridge, Mass.

Kriegman, D. J. and Ponce, J. (1990). On recognizing and positioning curved 3-d objects from image contours. *IEEE Trans. Pattern Anal. Machine Intell.*, 12:1127–1137.

Laurentini, A. (1994). The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Machine Intell.*, 16:150–162.

Lorensen, W. and Cline, H. (1987). Marching cubes: a high resolution 3d surface construction algorithm. In *Computer Graphics SIGGRAPH '87*, pages 163–170.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipies in C.* Cambridge University Press, Cambridge, England.

Prock, A. C. and Dyer, C. R. (1998). Towards real-time voxel coloring. In *Proc. Image Understanding Workshop*, pages 315–321.

Rander, P. W., Narayanan, P. J., and Kanade, T. (1996). Recovery of dynamic scene structure from multiple image sequences. In *1996 Int'l Conf. on Multisensor Fusion and Integration for Intelligent Systems*, pages 305–312, Washington, D.C.

Seitz, S. M. and Dyer, C. R. (1995). Physically-valid view synthesis by image interpolation. In *Proc. Workshop on Representations of Visual Scenes*, Cambridge, MA.

Seitz, S. M. and Dyer, C. R. (1996). View morphing. In *SIGGRAPH 96.*

Stein, G. P. (1993). Internal camera calibration using rotation and geometric shapes. Master's Thesis AITR-1426, Massachusetts Institute of Technology.

Sullivan, S. and Ponce, J. (1996). Automatic model construction, pose estimation, and object recognition from photographs using triangular splines. Technical report, Beckman Institute, University of Illinois.

Vaillant, R. and Faugeras, O. (1992). Using extremal boundaries for 3-d object modeling. *IEEE Trans. Pattern Anal. Machine Intell.*, 14:157–173.

Vijayakumar, B., Kriegman, D. J., and Ponce, J. (1996). Structure and motion of curved 3d objects from monocular silhouettes. In *CVPR '96*.

Zheng, J. Y. (1994). Acquiring 3-d models from sequences of contours. *IEEE Trans. Pattern Anal. Machine Intell.*, 16:163–178.