Image-Based Visual Hulls

by

Wojciech Matusik

B.S. Computer Science and Electrical Engineering
University of California at Berkeley (1997)

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2001

© Massachusetts Institute of Technology 2001. All rights reserved.

Author ....
Department of
Electrical Engineering and Computer Science
January 31, 2001

Certified by ....
Leonard McMillan
Assistant Professor
Thesis Supervisor

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

Image-Based Visual Hulls

by

Wojciech Matusik

Submitted to the Department of
Electrical Engineering and Computer Science
On February 1, 2001, in partial fulfilment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis addresses the problem of acquisition and rendering of dynamic real-world scenes. It describes an efficient image-based approach to computing and shading the scene geometry described by visual hulls – shapes obtained from silhouette image data. The novel algorithms take advantage of epipolar geometry and incremental computation to achieve a constant rendering cost per rendered pixel. They do not suffer from the computation complexity, limited resolution, or quantization artifacts of previous volumetric approaches. The use of these algorithms is demonstrated in a real-time virtualized reality application running off a small number of video streams.

Thesis Supervisor: Leonard McMillan
Title: Assistant Professor

Acknowledgements

First, I would like to thank my advisor Professor Leonard McMillan for letting me work on this exciting research topic. Professor McMillan has been a true mentor throughout the work on this thesis. I also would like to thank Professor Steven Gortler for co-advising me on this research project and always being enthusiastic about it. Many technical discussions with Professors McMillan and Gortler were vital in developing the ideas in this thesis.

I also would like to thank Chris Buehler for his original ideas and insights. Chris has been working with me on this project since the very beginning and he has been a great partner in developing this research.

Thanks also go to Ramesh Raskar for lots of help to get the initial version of the system to work, Kari Anne Kjølaas for the lab model, Tom Buehler for great videos, Ramy Sadek and Annie Choi for their help with the Siggraph paper. I also would like to thank the whole Computer Graphics Group at MIT for their support and understanding, especially when this project was taking over the whole lab space.

Finally, I would like to thank my family and my fiancé Gosia for always being there for me. Their support has been invaluable for me.

Contents

1 Introduction **15**

1.1 Contributions 16

1.2 Thesis Overview 16

2 Previous Work **19**

2.1 Systems 19

2.2 Concepts and Techniques 20

2.2.1 Visual Hull 20

2.2.2 Volume Carving 21

2.2.3 CSG Rendering 21

2.2.4 Image-Based Rendering 22

3 Visual Hull Sampling **25**

3.1 General Approach 25

3.3 Eliminating Polyhedra Intersections 26

3.3 Eliminating Polyhedron-Line Intersections 27

3.4 Projecting Rays and Lifting Back Intervals 28

3.5 Efficient Silhouette-Line Intersection 29

3.5.1 Edge-Bin Algorithm 30

3.5.1.1 Bin Construction 31

3.5.1.2 Efficient Bin Search 33

3.5.1.3 Bin Traversal Order 34

3.5.2 Wedge-Cache Algorithm 35

3.5.3 Selecting Valid Segments of Epipolar Lines 37

3.6 Coarse-to-Fine Sampling 40

3.7 Computation of the Visual Hull Surface Normals 41

3.8 Orthographic Camera 43

3.8.1 Orthographic Sampling Notations 43

3.8.2 Ray Projections in Orthographic Sampling 43
3.8.3 Edge-Bin and Wedge-Cache Algorithms in Orthographic Sampling 44
3.8.4 Selecting Valid Segments of Epipolar Lines 44
3.9 Time Complexity Analysis 47
3.9.1 Time Complexity Analysis of IBVH Sampling using Bin-Edge Algorithm ... 47
3.9.2 Time Complexity Analysis of IBVH Sampling using Wedge-Cache Algorithm 48

4 Visual Hull Shading **49**
4.1 View-Dependent Shading 49
4.2 Time Complexity Analysis of the Shading Algorithm 51
4.3 Visibility Algorithm 51
4.3.1 Naïve Approaches 52
4.3.2 Reduction to Visibility on Epipolar Planes 52
4.3.3 Front-to-Back Ordering 53
4.3.4 Efficient 2D Visibility Algorithm 54
4.3.5 Extension to the Discrete 3D Algorithm 57
4.3.5.1 Approximate Visibility Algorithm (Definition 1) 58
4.3.5.2 Approximate Visibility Algorithm (Definition 2) 59
4.3.6 Time Complexity Analysis of the Visibility Algorithm 60

5 Distributed Algorithms **61**
5.1 Node Types 61
5.2 Algorithm Description 61
5.3 Time Complexity Analysis 64

6 System Overview **65**
6.1 Hardware 65
6.2 System Tasks 65
6.3 Server Design 66
6.4 Image Segmentation 67

7 Results **71**

7.1 Input Images 71
7.2 Background Segmentation 71
7.3 Network Performance/Image CODEC 72
7.4 Output Images 72
7.5 Server Performance 73

8 Future Work & Conclusions **75**
8.1 Contours as Basic Geometric Primitives 76
8.2 Conclusions 77

References **79**

Appendix A **83**

List of Tables

Table 7.1 Running time of different parts of the system. 73

Table 7.2 Running time of different parts of the system when coarse-to-fine sampling and lower resolution visibility is used. 74

List of Figures

Figure 2.1 Three extruded silhouettes of a teapot and the resulting visual hull 21

Figure 3.1 Eliminating polyhedra intersections 26
Figure 3.2 Reduction to 2D intersections 27
Figure 3.3 Partitioning of the reference image space into bins – epipolar lines intersect at a point 32
Figure 3.4 Partitioning of the reference image space into bins – epipolar lines are parallel to each other 32
Figure 3.5 Monotonic property of epipolar lines for a scanline 33
Figure 3.6 Image boundaries used for wedge-cache indexing 36
Figure 3.7 Valid segments of epipolar lines 39
Figure 3.8 Valid segments of epipolar lines when the epipolar lines are parallel to each other 40
Figure 3.9 Coarse-to-Fine Sampling 41
Figure 3.10 Surface normal computation 42
Figure 3.11 Valid segments of epipolar lines in orthographic sampling 46
Figure 3.12 Valid segments of epipolar lines in orthographic sampling when the epipolar lines are parallel to each other 46

Figure 4.1 View-dependent texturing strategy 50
Figure 4.2 Reduction to visibility on epipolar planes 53
Figure 4.3 Front-to-back ordering on an epipolar plane 54
Figure 4.4 An example of the 2D visibility algorithm execution 55
Figure 4.5 Incorrect visibility determination 56
Figure 4.6 Approximate visibility algorithm (definition 1) 57
Figure 4.7 Approximate visibility algorithm (definition 2) 58
Figure 4.8 Desired image boundaries used to define wedges and lines 59

Figure 5.1 View propagation phase 62
Figure 5.2 Ray-cone intersection and ray intersection phases 62
Figure 5.3 IBVH propagation phase 63
Figure 5.4 Visibility computation and shade selection phases 64

Figure 7.1 Number of pixels vs. Time 74
Figure 7.2 Number of pixels vs. Time (coarse-to-fine sampling) 74

Figure A.1 Results of the stages of the background subtraction algorithm 83
Figure A.2 Sample images from four video streams used to compute IBVHs 83
Figure A.3 Sample output depth images from different viewpoints 84
Figure A.4 Sample output depth images computed using coarse-to-fine sampling 85
Figure A.5 Texture-mapped IBVH seen from different viewpoints 86
Figure A.6 Color-coded images of the IBVHs 87

Chapter 1

Introduction

One of the goals of the three-dimensional computer graphics is to create realistic images of dynamically changing scenes. Conventionally, these images are synthesized based on the geometric model, material properties, and lighting of the scene. Realistic images can be synthesized by acquiring these geometric description and properties from real scenes. Current acquisition methods can be divided into two groups: active and passive. Passive methods are the ones that use only passive devices, (for example digital cameras) to acquire the scene description. Active methods include structured light and laser based 3D range scanners. Typically passive methods are not very robust in capturing geometric scene description. Moreover, passive methods are frequently computationally expensive. Therefore, it is generally hard to obtain high-quality results at interactive frame rates using passive methods. In contrast, active methods usually produce high quality results, but they typically require expensive hardware. Furthermore, the acquisition process is time consuming and might require modifying the scene (painting objects white). Therefore, it is difficult to use these methods to acquire many scenes.

An ideal acquisition method would use inexpensive hardware and produce high quality models of the scene. Moreover, the whole acquisition process and scene reconstruction would be done at interactive frame rates. Such a method would have many potential applications. One of the most frequently cited applications is digitizing sporting events in real-time. This allows the viewer to observe the event from any position in the scene. Another potential application is a virtual meeting. The virtual meeting allows people at distant places to be digitized in real-time and placed in some virtual environment where they can interact with each other.

The method of acquisition and rendering of real scenes described in this thesis has many of the properties of an ideal method. It is a passive method, and it does not require any expensive hardware other than a small number of digital video cameras. The method is robust – it works well on all types of objects. It is also very efficient – it is able to reconstruct and render high-resolution images of the scene fast.

The proposed method for generation of the scene geometry can be classified as extracting structure-from-silhouettes. The method is robust since it relies only on the silhouettes of objects in the scene. Silhouettes of the object can be obtained using well-known and reliable techniques such as background-subtraction or chroma-keying. Furthermore, the method does not suffer from the computation complexity, limited resolution, or quantization artifacts of previous volumetric approaches.

1.1 Contributions

The contributions of this thesis include:

(1) an efficient algorithm for computing the geometry of the scene that takes advantage of the epipolar geometry and incremental computation to yield approximately constant rendering time per pixel per reference image.

(2) an efficient algorithm for determining the visibility and correctly shading the geometry of the scene. The visibility algorithm also takes advantage of the epipolar geometry and incremental computation to yield approximately constant rendering time per pixel per reference image.

(3) a real-time system that demonstrates the use of the above algorithms. The system allows constructing a view of the dynamic scene from an arbitrary camera position based on a set of input video streams.

1.2 Thesis Overview

In chapter 2, previous work in the area is discussed, including background information on the concept of the visual hull – the basic geometric primitive I use. In chapter 3, I describe how to construct the geometry of the scene, including a series of optimizations to make this operation efficient. I also introduce an efficient data structure

for storing the geometry of the scene. In chapter 4, I describe how to shade the scene based on the texture from the input video streams, and show how to efficiently determine which parts of the scene are visible from which camera. In chapter 5, I show how to parallelize both geometry construction and shading algorithms in a multiprocessor/distributed system. Chapter 6 describes the architecture of the real-time system. In chapter 7, I present and discuss the results obtained from the system. Chapter 8 discusses the extensions of the algorithms and possible directions for future work.

Chapter 2

Previous Work

The previous work discussion is divided into two categories: (1) the first category describes research with the same goal as mine – to build a system that digitizes dynamic scenes in real-time. This work is described in section 2.1. (2) The second category presents an overview of various techniques upon which the presented algorithms are based. These are described in section 2.2.

2.1 Systems

Moezzi's et al. Immersive Video system [19] is perhaps the closest in spirit to the rendering system I envision. Their system uses silhouette information obtained by background subtraction to construct the visual hulls of the dynamic objects in the scene. The model of a background is created off-line. Similarly, they use view-dependent texturing to shade their acquired models. However, their algorithms are less efficient; and therefore, their system is an off-line system. They construct sampled volumetric models using volume carving techniques and then convert them to the polygonal representations.

Kanade's virtualized reality system [20] [23] [11] is also similar to my system. Their initial implementations have used a collection of cameras in conjunction with multi-baseline stereo techniques to extract models of dynamic scenes. These methods require significant off-line processing, but they are exploring special-purpose hardware for this task. Recently, they have begun exploring volume-carving (voxel coloring) methods, which are closer to the approach that I use [12] [26] [29].

Pollard's and Hayes' [21] immersive video objects allow rendering of real-time scenes by morphing live video streams to simulate three-dimensional camera motion. Their representation also uses silhouettes, but in a different manner. They match silhouette edges across pairs of views, and use these correspondences to compute morphs

to novel views. This approach has some limitations, since silhouette edges are generally not consistent between views.

2.2 Concepts and Techniques

The concepts and techniques that I use can be broken down to four different subcategories: visual hull, volume carving, Constructive Solid Geometry (CSG) rendering, and image-based rendering.

2.2.1 Visual Hull

Many researchers have used silhouette information to distinguish regions of 3D space where an object is and is not present [22] [7] [19]. The ultimate result of this carving is a shape called the object's *visual hull*. The concept of the visual hull was introduced by Laurentini [14]. In computational geometry, the visual hull is sometimes related to as the line hull. A visual hull has the property that it always contains the object. Moreover, it is an equal or tighter fit than the object's convex hull.

Suppose that some original 3D object is viewed from a set of reference views R. Each reference view r has the silhouette s_r with interior pixels covered by the object's image. For view r one creates the cone-like volume vh_r defined by all the rays starting at the image's center of projection p_r and passing through these interior points on its image plane. It is guaranteed that the actual object must be contained in vh_r. This statement is true for all r; thus, the object must be contained in the volume $vh_R = \cap_{r \in R} vh_r$. As the size of R goes to infinity, and includes all possible views, vh_R converges to a shape known as the visual hull vh_∞ of the original geometry. The visual hull is not guaranteed to be the same as the original object since concave surface regions can never be distinguished using silhouette information alone [13]. Furthermore, one must in practice construct approximate visual hulls using only a finite number of views. Given the set of views R, the approximation vh_R is the best conservative geometric description that one can achieve based on silhouette information alone (see Figure 2.1). If a conservative estimate is not required, then alternative representations are achievable by fitting higher order surface approximations to the observed data [1].

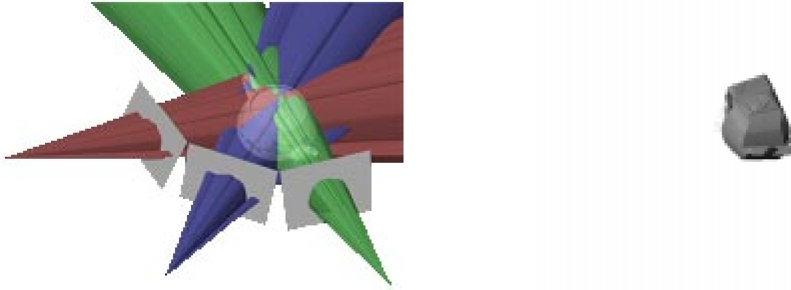

Figure 2.1 Three extruded silhouettes of a teapot and the resulting visual hull.

2.2.2 Volume Carving

Computing high-resolution visual hulls can be tricky matter. The intersection of the volumes requires some form of CSG. If the silhouettes are described with a polygonal mesh, then the CSG can be done using polyhedral CSG, but this is very hard to do in a robust manner. A more common method used to convert silhouette contours into visual hulls is volume carving [22] [7] [28] [19] [4] [26]. This method removes unoccupied regions from an explicit volumetric representation. All voxels falling outside of the projected silhouette cone of a given view are eliminated from the volume. This process is repeated for each reference image. The resulting volume is a quantized representation of the visual hull according to the given volumetric grid. A major advantage of the view-dependent method presented in this thesis is that it minimizes artifacts resulting from this quantization.

2.2.3 CSG Rendering

A number of algorithms have been developed for the fast rendering of CSG models, but most are ill suited for the task of real-time model extraction. The algorithm described by Rappoport [24], requires that each solid be first decomposed to a union of convex primitives. This decomposition can prove expensive for complicated silhouettes. Similarly, the algorithm described in [9] requires a rendering pass for each layer of depth complexity. The method presented in this thesis does not require pre-processing the

silhouette cones. In fact, there is no explicit data structure used to represent the silhouette volumes other than the reference images.

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

2.2.4 Image-Based Rendering

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

Chen's QuicktimeVR [3] is one of the first systems for rendering acquired static models. This system relies heavily on image-based rendering techniques to produce photo-realistic panoramic images of real scenes. Although successful, the system has some limitations: the scenes are static and the viewpoint is fixed.

McMillan's plenoptic modelling system [17] is QuicktimeVR-like, although it does allow a translating viewpoint. The rendering engine is based on three-dimensional image warping, a now commonplace image-based rendering technique. Dynamic scenes are not supported as the panoramic input images require much more off-line processing than the simple QuicktimeVR images.

Light field methods [15] [10] represent scenes as a large database of images. Processing requirements are modest making real-time implementation feasible, if not for

the large number of cameras required (on the order of hundreds). The cameras must also be placed close together, resulting in a small effective navigation volume for the system.

A system developed by Debevec [5] uses photographs and an approximate geometry to create and render realistic models of architecture. The system requires only a small number of photographs. However, a user is required to specify an approximate geometry and rough correspondences between the photographs. Therefore, the system can be only applied to static scenes.

One advantage of image-based rendering techniques is their stunning realism, which is largely derived from the acquired images they use. However, a common limitation of these methods is an inability to model dynamic scenes. This is mainly due to data acquisition difficulties and pre-processing requirements. This system generates image-based models in real-time, using the same images to construct the IBHV and to shade the final rendering.

Chapter 3

Visual Hull Sampling

This chapter explains how to efficiently compute the sampling of the visual hull from an arbitrary viewpoint. The algorithms that are described in this chapter take as input a set of silhouettes and their locations in space. Each silhouette is specified by a binary segmentation of the image. Image formation is modelled using a pinhole camera. The location of the input silhouettes is derived from the extrinsic and intrinsic parameters of the input cameras. The algorithm outputs a view-dependent, sampled version of an object's visual hull. The output visual hull samples are defined as the intervals along each viewing ray that are inside of the object's visual hull. Section 3.1 describes the naïve algorithm that computes the visual hull sampling. Sections 3.2-3.6 show a series of optimizations to obtain an efficient algorithm. Section 3.7 discusses how to compute surface normals for each point on the visual hull. Section 3.8 extends the algorithms to the orthographic sampling. Finally, section 3.9 analyzes the time complexity of the algorithms described in this chapter.

3.1 General Approach

The obvious way to compute an exact sampling of a visual hull from some specified viewpoint consists of two steps: first, compute a polyhedron that defines the surface of a visual hull; then sample this polyhedron along the viewing rays to produce the exact image of a visual hull. Since each of the extruded silhouettes is a generalized cone and a visual hull is an intersection of all these cones the first step is equivalent to a CSG intersection operation (or an intersection of 3D polyhedra). The second step is also an intersection operation – an intersection of 3D lines with the resulting visual hull polyhedron. The pseudo-code for the algorithm is shown below.

```
VHisect (intervalImage &d, refImList R)
(1)    Polyhedron vh(R) = BoundingBox
(1)    for each referenceImage r in R
(2)       Polyhedron cone(r) = computeExtrudedSilhouette(r)
(3)       vh(R) = vh(R) ∩ cone(r)
(4)    for each pixel p in desiredImage d do
(5)       p.intervals = {0..inf}
(6)       p.intervals = p.intervals ∩ vh(R)
```

3.3 Eliminating Polyhedra Intersections

The visual hull method presented in this thesis relies on the commutative properties of the intersection operation. The intersection of a ray with a visual hull polyhedron is described mathematically as follows:

$$vh(R) = \left(\bigcap_{r \in R} cone(r) \right) \cap ray3D \;.$$

This operation is equivalent to:

$$vh(R) = \bigcap_{r \in R} \left(cone(r) \cap ray3D \right) .$$

This means that one can first intersect each extruded silhouette with each 3D ray (3D line) separately. This results in a set of intervals along the ray. The intersection of all these sets of intervals for all silhouettes can then be computed. In this process one exchanges the polyhedra and polyhedron-line intersections for simpler polyhedron-line and line-line intersections (Figure 3.1). This simplifies the geometric calculations. The next sections explain how to further reduce the volume-line intersections.

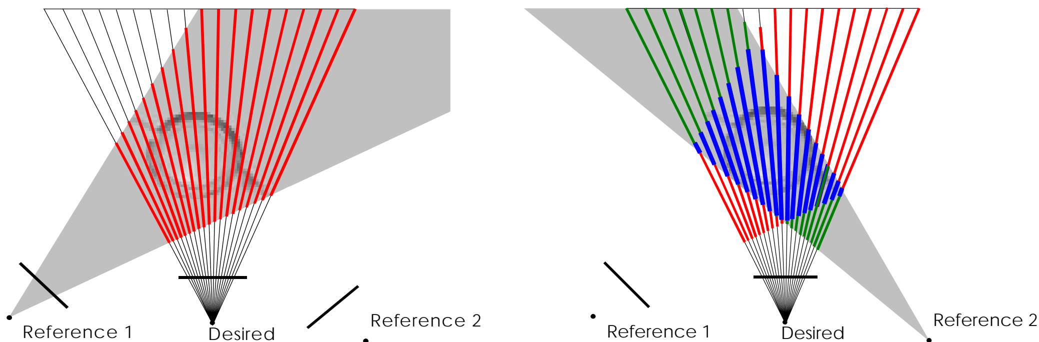

Figure 3.1 Eliminating polyhedra intersections. To obtain the samples along the rays in the desired image compute the intersection of the first silhouette cone with the desired rays (red line segments). Then compute the intersection of the second silhouette cone with the desired rays (green line segments). Finally, compute the intersection of both red and green line segments to obtain the visual hull samples with respect to these two reference images.

3.3 Eliminating Polyhedron-Line Intersections

The next optimization of visual hull computation relies on the fact that the absolute cross-section of the extruded silhouette remains fixed (this is because the extrusion is defined by the 2D silhouette). This observation implies that instead of computing the intersection of extruded silhouette with a ray directly, one can achieve the same result by performing the following steps. First, compute the intersection of the cone cross-section with a ray projected onto any plane. Then, reproject the results of the intersection onto the initial 3D ray (Figure 3.2). Effectively it reduces the polyhedron-line intersections to cheaper polygon-line intersections.

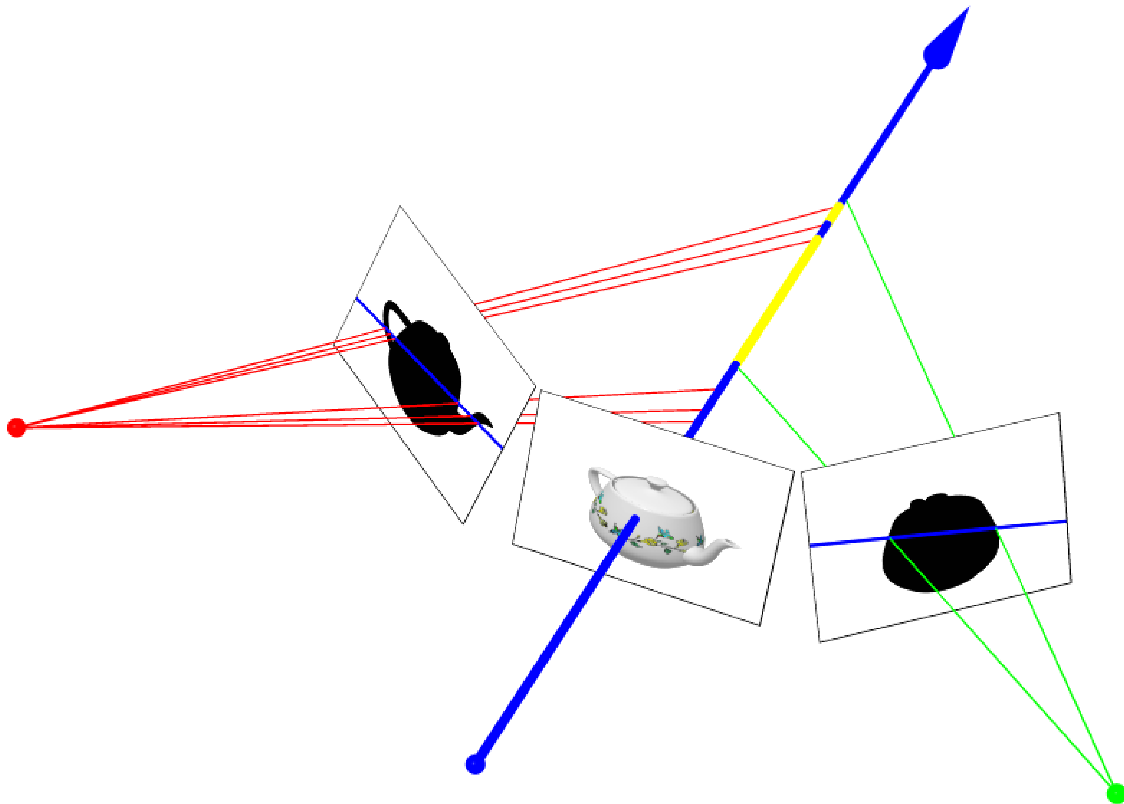

Figure 3.2 Reduction to 2D intersections. Instead of computing the intersection of the desired ray with extruded silhouette cones project the desired ray onto the reference images, compute the intersections of the projected rays with the silhouette, and lift back the resulting intervals to 3D.

One could pick any plane that intersects the entire silhouette cone to define the needed cross-section when performing the silhouette/line intersection. However, it is best

to perform the operation on the reference image plane because the cross-section is already defined for this plane (it is the silhouette), and one avoids any possible image resampling artifacts. The following pseudo-code outlines the new version of the intersection operation with the above optimizations.

```
IBVHisect (intervalImage &d, refImList R)
(1)   for each referenceImage r in R
(2)      computeSilhouetteEdges (r)
(3)   for each pixel p in desiredImage d do
(4)      p.intervals = {0..inf}
(5)   for each referenceImage r in R
(6)      for each pixel p in d
(7)         ray3D ry3 = compute3Dray(p,d.camInfo)
(8)         lineSegment2D l2 = project3Dray(ry3,r.camInfo)
(9)         intervals int2D = calc2Dintervals(l2,r.silEdges)
(10)        intervals int3D = lift2Dintervals(int2D,r.camInfo,ry3)
(11)        p.intervals = p.intervals ∩ int3D
```

3.4 Projecting Rays and Lifting Back Intervals

Before proceeding with further details I will discuss how to project rays onto planes and how to lift the resulting 2D intervals back to 3D. First, I describe the notation for the pinhole camera model used. The point C denotes the center of projection of the camera and P denotes an inverse projection matrix that transforms homogenous image coordinates $x = [u\ v\ 1]^T$ to rays in 3D. The pair $[P\ |C]$ specifies the location of the camera in the 3D space. The following equation specifies how to compute a ray in 3D:

$$X(t) = C + t\,Px.$$

The following equation is used to obtain point a, the projection of the 3D point A onto the camera's image plane specified by $[P\ |C]$:

$$a = P^{-1}\,(A-C).$$

According to the pseudo-code for the `IBVHisect` algorithm one first computes the ray in 3D for each pixel in the desired image (line 7) and then projects the ray onto the reference image plane (line 8). One could compute the ray projection by projecting two of its points onto the reference image and then determining the line through the points. Alternatively one could compute the fundamental matrix F [8]. The fundamental matrix F expresses the relationship between the pixel coordinates of the first (desired) view and the epipolar lines in the second (reference) view. That is, the fundamental matrix satisfies the following equation:

$$x_{ref}^{T} F_{ref\text{-}des}\, x_{des} = 0.$$

The quantity $F_{ref\text{-}des}\, x_{des}$ gives the coefficients of the line in the reference image. The fundamental matrix can be computed using the following formula:

$$F_{ref\text{-}des} = E_{ref}\, P_{ref}^{-1} P_{des},$$

where matrix E_{ref} is a skew-symmetric matrix defined such that

$$E_{ref}\, v = {}_{0}x_{ref\text{-}des} \times v,$$

where v is an arbitrary vector and vector ${}_{0}x_{ref\text{-}des}$ is the epipole, or the projection of the desired camera's center of projection onto the reference camera's image plane. The epipole ${}_{0}x_{ref\text{-}des}$ is given by the formula:

$${}_{0}x_{ref\text{-}des} = P_{ref}^{-1}\, (C_{des} - C_{ref}).$$

The subroutine `lift2Dintervals` computes the intervals along the desired 3D ray based on the intervals in the reference image. This is achieved by computing 3D locations of each startpoint and endpoint of the interval in 2D. The 3D location is just an intersection of two rays: the desired ray and the reference camera ray passing through the startpoint/endpoint of the interval (Figure 3.2). In fact, the point of the closest approach to these two rays is computed. The formula for the distance to the point of the closest approach along the desired viewing ray expressed in terms of the vector from the desired center of projection to the desired viewing plane is given below:

$$t = \frac{\det\left| C_{ref} - C_{des} \quad \mathbf{P}_{ref} x_{ref} \quad \mathbf{P}_{des} x_{des} \times \mathbf{P}_{ref} x_{ref} \right|}{\left\| \mathbf{P}_{des} x_{des} \times \mathbf{P}_{ref} x_{ref} \right\|^{2}}$$

3.5 Efficient Silhouette-Line Intersection

The naïve algorithm for computing intersections of epipolar lines with a silhouette traverses each epipolar line. When a line is traversed its intersections with the silhouette are computed. The running time of the naïve algorithm is too expensive. Therefore, this section describes how to compute line/silhouette intersections efficiently. Two alternative algorithms for this task are presented: Edge-Bin algorithm (Section 3.5.1) and Wedge-Cache algorithm (Section 3.5.2). Both of these algorithms exploit properties of epipolar geometry and employ incremental computation to achieve good performance.

In both algorithms a silhouette is represented as a set of contour edges. In order to extract this silhouette contour I have used a 2D version of the marching-cubes algorithm [16]. As a result a set of edges is found, where each edge is defined by its start-point and end-point. It is important to note that (1) there is no differentiation between edges of inner and outer contours; and (2) the edges in the set do not have to be the consecutive edges in the contour.

3.5.1 Edge-Bin Algorithm

Observe that all projected rays intersect at one common point, the epipole ${}_0x_{ref\text{-}des}$ (the projection of the desired camera's optical center onto the reference image plane). Thus, it is possible to partition the reference image into regions such that epipolar lines that lie entirely in the region intersect all the edges and only the edges in this region. The regions boundaries are determined by the lines passing through the epipole ${}_0x_{ref\text{-}des}$ and the silhouette vertices. All projected rays (epipolar lines) can be parameterized based on a slope m that they make with some reference epipolar line.

There is a special case, when the projected rays are parallel to each other. This special case arises when the epipole ${}_0x_{ref\text{-}des}$ is at infinity (${}_0x_{ref\text{-}des\ 3} = 0$). Therefore, it is not possible to use the slope m for parameterizing the epipolar lines since all lines have exactly the same slope. In this case a line given by the following equation is used:

$$p(t)=p_0 + d\, t.$$

In this notation point p_0 is some arbitrary point on the line p and vector d is a vector perpendicular to the direction of the projected rays. The projected rays can be parameterized based on the value of t at the intersection points of the line p with the projected ray.

The rest of the Edge-Bin algorithm does not differ whether the epipolar lines are parallel to each other or not. Thus, let parameter s denote parameter m if epipolar lines intersect at the epipole and let it denote parameter t if the epipolar lines are parallel to each other.

Given this parameterization the domain of $s = (-\infty, \infty)$ is partitioned into ranges such that any epipolar line with the value of parameter s falling inside of the given range

always intersects the same set of edges of the reference silhouette. The boundary of the range is described using two values s_{start} and s_{end}, where s_{start}, s_{end} are the values of s for lines passing through two silhouette vertices that define the region. Let a bin b_i be a three-tuple: the start s_{start}, the end s_{end} of the range, and a corresponding set of edges S_i, $b_i=(s_{start}, s_{end}, S_i)$. Section 3.5.1.1 describes how construct these bins efficiently.

The bin data structure allows to efficiently look up all the edges intersected by any epipolar line – for an epipolar line with the slope s one needs to find a bin such that $s_{start} \leq s < s_{end}$. The bins are sorted in the order of increasing slopes; therefore, the search for the corresponding bin can be performed quickly. A simple binary search could be used to find the corresponding bin. However, this operation can be done even more efficiently using incremental computation (Section 3.5.1.2). Once the bin corresponding to a given ray is found one can compute analytical intersections of the edges in the bin with a projected ray to determine the intervals inside the silhouette. These are simple line/line intersections in 2D.

3.5.1.1 Bin Construction

This section outlines how to compute bins efficiently. For each edge point of the silhouette the value of the parameter s of the epipolar line that passes through this vertex is computed. Then the list values of the parameter s is sorted in a non-decreasing order. Next, the redundant copies of the parameter s are removed. Two consecutive values in the sorted list define the start and the end of each bin. To compute the set of edges assigned to each bin one traverses the sorted list of silhouette vertices. At the same time one keeps the list of edges in the current bin. When a vertex of the silhouette is visited one removes from the current bin an edge that ends at this vertex and adds an edge that starts at the vertex. A start of an edge is defined as the edge endpoint that has a smaller value of the parameter s of the corresponding epipolar line. Figures 3.3 and 3.4 show a simple silhouette, bins, and corresponding edges for each bin.

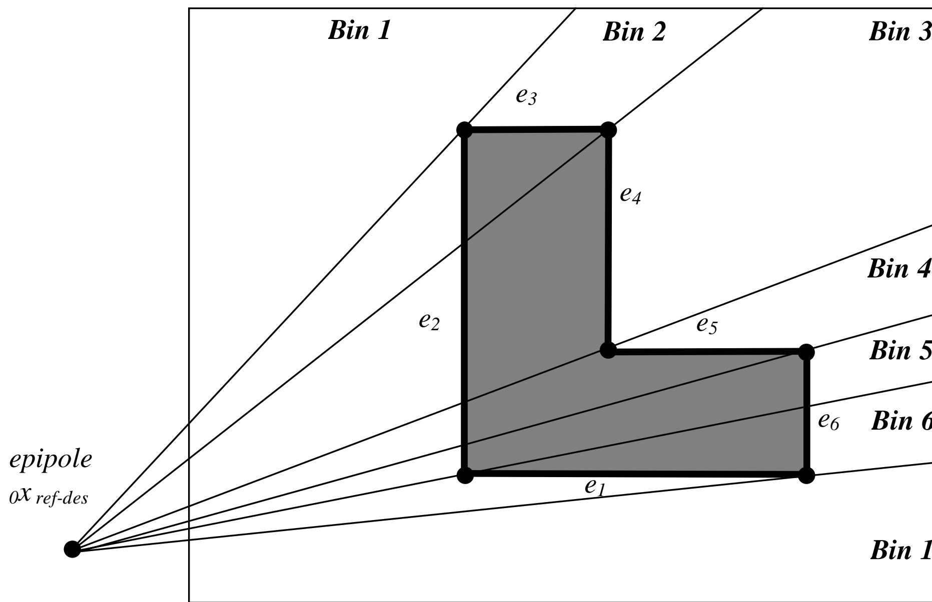

Figure 3.3 Partitioning of the reference image space into bins – epipolar lines intersect at a point. In this example reference image space is partitioned into six bins. Bin 1 contains no edges. Bin 2 contains edges e_2, e_3. Bin 3 contains edges e_2, e_4. Bin 4 contains edges *e2, e5*. Bin 5 contains edges e_2, e_6. Bin 6 contains edges e_1, e_6.

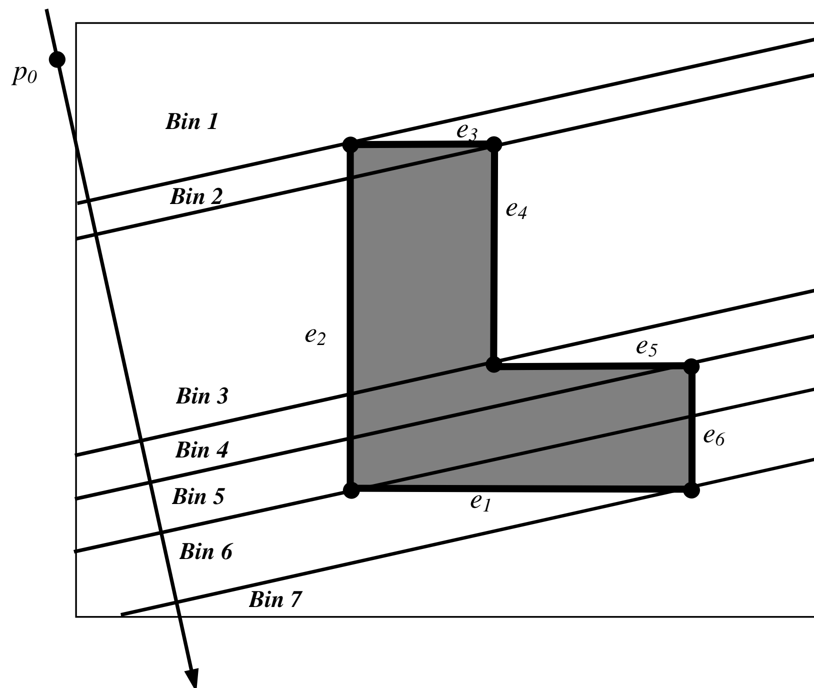

Figure 3.4 Partitioning of the reference image space into bins – epipolar lines are parallel to each other. In this example there are 7 bins. Bin 1 and Bin 7 do not contain any edges. Bin 2 contains edges e_2 and e_3. Bin 3 contains edges e_2 and e_4. Bin 4 contains e_2 and e_5. Bin 5 contains e_2 and e_6. Bin 6 contains edges e_1 and e_6.

3.5.1.2 Efficient Bin Search

Next, I describe how to take advantage of the epipolar geometry to perform the search for each bin incrementally. First, notice that the values of parameter *s* of the epipolar lines in the reference image corresponding to the consecutive pixels in the scanline of the desired image are monotonic – they are either increasing or decreasing. This property is illustrated in Figure 3.5. This implies that the efficient algorithm for one scanline of the desired image proceeds as follows: (1) determine the monotonicity type (increasing/decreasing) for the scanline; (2) find the corresponding bin for the first pixel in the scanline using binary search; (3) next, move forward (in case of increasing slopes) or backwards (in case of decreasing slopes) in the sorted list of bins to find the correct bin for each projected ray. Usually successive pixels in a scanline project to lines that are no more than a few bins away from the current position in the list.

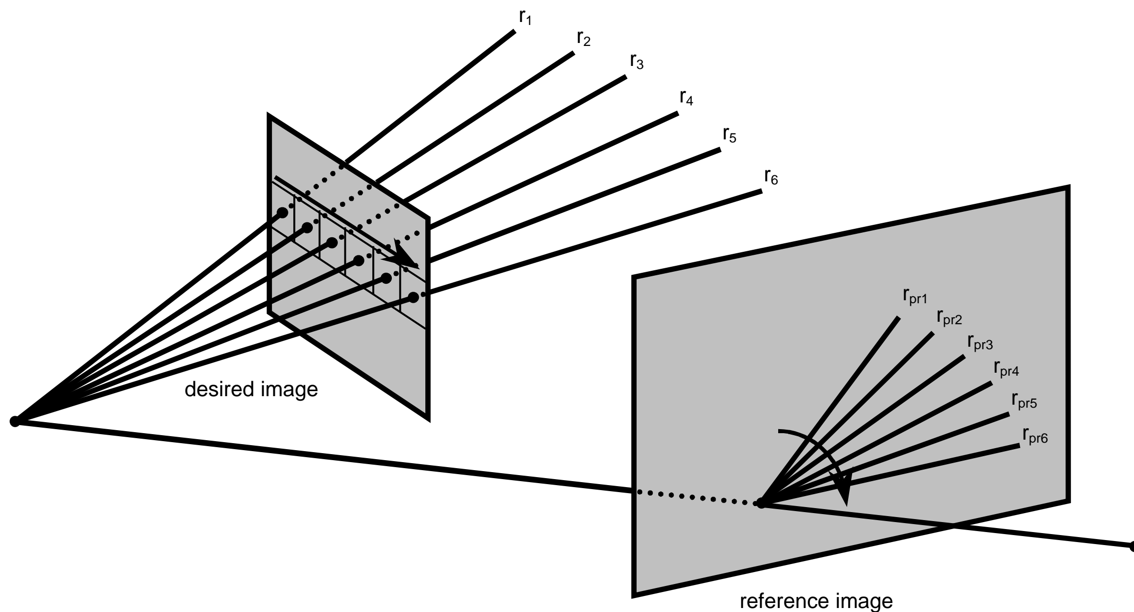

Figure 3.5 Monotonic property of epipolar lines for a scanline. As one moves across the scanline in the desired image the projected rays in the reference image trace out a pencil of epipolar lines. These lines have a monotonic slope with respect to some reference line passing through the epipole.

3.5.1.3 Bin Traversal Order

This section describes how to determine the monotonicity type of the projected rays in a scanline. First, project two points in the desired image onto the reference image plane. These points are the first P_{first} and the last P_{last} points in the scanline on the desired image plane. Use the following formulas to compute their projections p_{first} and p_{last}:

$$p_{first} = P_{ref}^{-1} (P_{first} - C_{ref}) \qquad p_{last} = P_{ref}^{-1}(P_{last} - C_{ref})$$

In the first case, when all projected rays intersect at the epipole ${}_0x_{ref\text{-}des}$, in order to determine monotonicity of the projected rays the angle between vectors ${}_0x_{ref\text{-}des}p_{first}$ and ${}_0x_{ref\text{-}des}p_{last}$ needs to be examined. If the angle is positive then the slopes are increasing. If the angle is negative then the slopes are decreasing. Instead of computing the angle it is easier to use coordinates of the two vectors directly. The pseudo-code for the function is given below:

```
BinTraversalOrderCase1 (point2D p_firtst, point2D p_last, point2D 0x_ref-des)
(1)    0x_ref-des p_first = p_firtst - 0x_ref-des
(2)    0x_ref-des p_last = p_last - 0x_ref-des
(3)    if (0x_ref-des p_first.x == 0  && 0x_ref-des p_last.x == 0)
(4)       // the same slope
(5)       return no_change
(6)    // first check if vector p_first p_last intersects relative y axis
(7)    if (p_first.x <= 0x_ref-des.x && p_last.x >= 0x_ref-des.x)
(8)       if (p_first.y >= 0x_ref-des.y)
(9)          return decreasing
(10)      else
(11)         return increasing
(12)   else if (p_last.x <= 0x_ref-des.x && p_first.x >= 0x_ref-des.x)
(13)      if (p_first.y >= 0x_ref-des.y)
(14)         return increasing
(15)      else
(16)         return decreasing
(17)   else
(18)      // vector p_first p_last does not intersect relative y axis
(19)      s1 =  0x_ref-des p_first.y / 0x_ref-des p_first.x
(20)      s2 = 0x_ref-des p_last.y / 0x_ref-des p_last.x
(21)      if (s1 > s2)
(22)         return decreasing
(23)      else
(24)         return increasing
```

In the second case, when all projected rays are parallel to each other, the monotonicity type depends on the angle between the vector $p_{firtst}p_{last}$ and the direction vector d of the parameterization line. If the angle is less then 90 degrees then the order is increasing; if the angle is more than 90 degrees then the order is decreasing. In practice, it

is easier to do this comparison based on the cosine of the angle. The pseudo-code for the whole function is given below.

```
BinTraversalOrderCase2 (point2D p_firtst, point2D p_last, vector2D d)
(1)    p_first p_last = p_last - p_first
(2)    p_first p_last = p_first p_last / | p_first p_last |
(3)    cos_angle = dotproduct (d , p_first p_last)
(4)    if ( cos_angle  > 0)
(5)          return increasing
(6)    else
(7)          return decreasing
```

Finally, the pseudo-code for the efficient `IBVHisect` that uses Edge-Bin algorithm is given:

```
IBVHisect (intervalImage &d, refImList R)
(1)    for each referenceImage r in R
(2)       computeSilhouetteEdges(r)
(3)    for each pixel p in desiredImage d do
(4)       p.intervals = {0..inf}
(5)    for each referenceImage r in R
(6)       Bins bins = constructBins(r.caminfo, r.silEdges, d.caminfo)
(7)    for each scanline e in d
(8)       IncDec order = binTraversalOrder(r.caminfo, d.caminfo, e)
(9)       for each pixel p in e
(10)         ray3D ry3 = compute3Dray(p,d.camInfo)
(11)         lineSegment2D l2 = project3Dray(ry3,r.camInfo)
(12)         int s = computeParameter(ry3,r.camInfo)
(13)         adjustBinPosition (s, bins, order)
(14)         intervals int2D = calc2Dintervals(l2,bins.currentbin)
(15)         intervals int3D = lift2Dintervals(int2D,r.camInfo,ry3)
(16)         p.intervals = p.intervals ∩ int3D
```

3.5.2 Wedge-Cache Algorithm

The Wedge-Cache algorithm is an alternative method for computing line/silhouette intersections. The main idea behind this algorithm is also based on the epipolar geometry of two views. An epipolar plane, a plane that passes through centers of projections of both cameras, intersects image planes of both cameras. It is easy to see that all rays that pass through line A, the intersection of the first image plane with the epipolar plane, project to the same line B, the intersection of the second image plane with the epipolar plane. Therefore, many rays in the desired view project to the same line in the reference view. One can take advantage of this fact to compute the fast line/silhouette intersection. First, project a desired ray onto the reference view and if the intersection of this epipolar line has not been computed yet, traverse the line (using for example Bresenham's algorithm) and compute all its intersections with the silhouette. Store all the

silhouette edges that intersect this epipolar line. When other desired rays project to the same epipolar line look up silhouette edges that this epipolar line intersects [2].

Discretization of the desired and reference images into pixels introduces some problems. The pixels in the desired image do not usually lie on the same line and therefore the corresponding projected desired rays do not have exactly the same slopes. Since one wants to reuse silhouette intersections with epipolar lines intersections of the silhouette with wedges rather than lines need to be computed. Therefore, the reference image space is divided into a set of wedges such that each wedge has exactly one pixel width at the image boundary where the edge leaves the image. Depending on the position of the epipole with respect to the reference image, there are nine possible cases of image boundary parts that need to be used. These cases are illustrated in Figure 3.6.

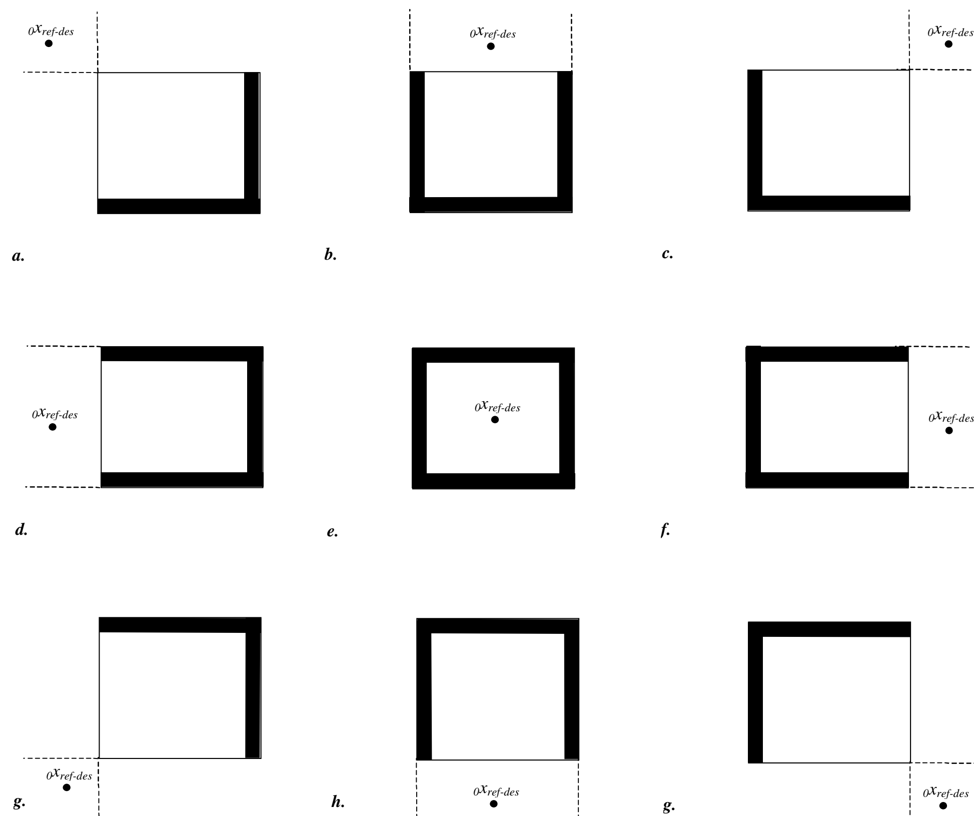

Figure 3.6 Image boundaries used for wedge-cache indexing. Depending on the position of the epipole with respect to the reference image boundary different parts of the reference image boundary (thick black lines) need to be used for wedge indexing.

The Wedge-Cache algorithm can be described as follows. For each desired ray compute its epipolar line. Then look up into which wedge this line falls. If the silhouette

edges of this wedge have not been computed yet then use Bresenham's-like algorithm to traverse all the pixels in the wedge to find these edges. Next, test which of the computed edges in the wedge actually intersect the given epipolar line. Later, when other epipolar lines fall into this wedge just look up the edges contained in the wedge and test for intersection with the epipolar line. The main advantage of the Wedge-Cache algorithm over the previous algorithm is that (1) it is less complex; and (2) it employs a lazy-computation strategy – pixels and edges in the wedge are processed only when there is an epipolar line which lies in this wedge. The pseudocode for `IBVHisect` that uses the Wedge-Cache algorithm is given below.

```
IBVHisect (intervalImage &d, refImList R)
(1)    for each referenceImage r in R
(2)       computesSilhouetteEdges(r)
(3)    for each pixel p in desiredImage d do
(4)       p.intervals = {0..inf}
(5)    for each referenceImage r in R
(6)       clear(Cache)
(7)       for each pixel p in desiredImage d
(8)          ray3D ry3 = compute3Dray(p,d.camInfo)
(9)          lineSegment2D l2 = project3Dray(ry3,r.camInfo)
(10)         int index = computeWedgeCacheIndex(l2)
(11)         if Cache[index] == EMPTY
(12)            silhouetteEdges S = traceEpipolarWedge(index,r)
(13)            Cache[index] = S
(14)         intervals int2D = calc2Dintervals(l2,Cache[index])
(15)         intervals int3D = lift2Dintervals(int2D,r.camInfo,ry3)
(16)         p.intervals = p.intervals ∩ int3D
```

3.5.3 Selecting Valid Segments of Epipolar Lines

When performing the intersection of the epipolar lines (projections of the rays of the desired image) with the silhouette in the reference image some extra care needs to be taken since some portions of the epipolar lines are invalid. This is because one is only interested in the portions of the epipolar line (a half line starting at desired center of projection) visible in the reference image. Before discussing how to determine valid portions of an epipolar line a concept of a maximum and a minimum image-space extent along the desired ray needs to be introduced. The maximum extent is given by the formula:

$$_{\infty}x_{ref\text{-}des} = P_{ref}^{-1} P_{des} x_{des}.$$

It is a point in the reference image that corresponds to the image of desired rays's vanishing point. The minimum extent is given by the following formula:

$$_0x_{ref\text{-}des} = P_{ref}^{-1} (C_{des} - C_{ref}).$$

It represents the image of the desired center of projection on reference image plane. A discussion of these extents can be found in McMillan [18].

According to McMillan [18], when the epipolar lines intersect at one point, there are four cases that need to be considered. These are based on:

(1) the position of the desired center of projection with respect to the reference image plane. It is determined by the third homogenous coordinate of $_0x_{ref\text{-}des}$. If $_0x_{ref\text{-}des\ 3}$ is positive then desired center of projection is in front of the reference image plane. If $_0x_{ref\text{-}des\ 3}$ is negative then desired center of projection is behind of the reference image plane.

(2) the direction of the desired ray with respect to the reference image plane

The four cases are given below:

(1) ($_0x_{ref\text{-}des\ 3} < 0$) and ($_\infty x_{ref\text{-}des\ 3} > 0$) (Figure 3.7 a)

The valid extent of the epipolar line is the half-line ending at $_\infty x_{ref\text{-}des}$ in the direction towards $_0x_{ref\text{-}des}$.

(2) ($_0x_{ref\text{-}des\ 3} > 0$) and ($_\infty x_{ref\text{-}des\ 3} < 0$) (Figure 3.7 b)

The valid extent of the epipolar line is the half-line from $_0x_{ref\text{-}des}$ in the direction opposite of $_\infty x_{ref\text{-}des}$.

(3) ($_0x_{ref\text{-}des\ 3} > 0$) and ($_\infty x_{ref\text{-}des\ 3} > 0$) (Figure 3.7 c)

The valid extent of the epipolar line is the segment connecting $_0x_{ref\text{-}des}$ to $_\infty x_{ref\text{-}des}$ in the reference image.

(4) ($_0x_{ref\text{-}des\ 3} < 0$) and ($_\infty x_{ref\text{-}des\ 3} < 0$) (Figure 3.7 d)

There is no valid extent of the epipolar line. No point is visible in the reference image that falls into extent of the desired ray.

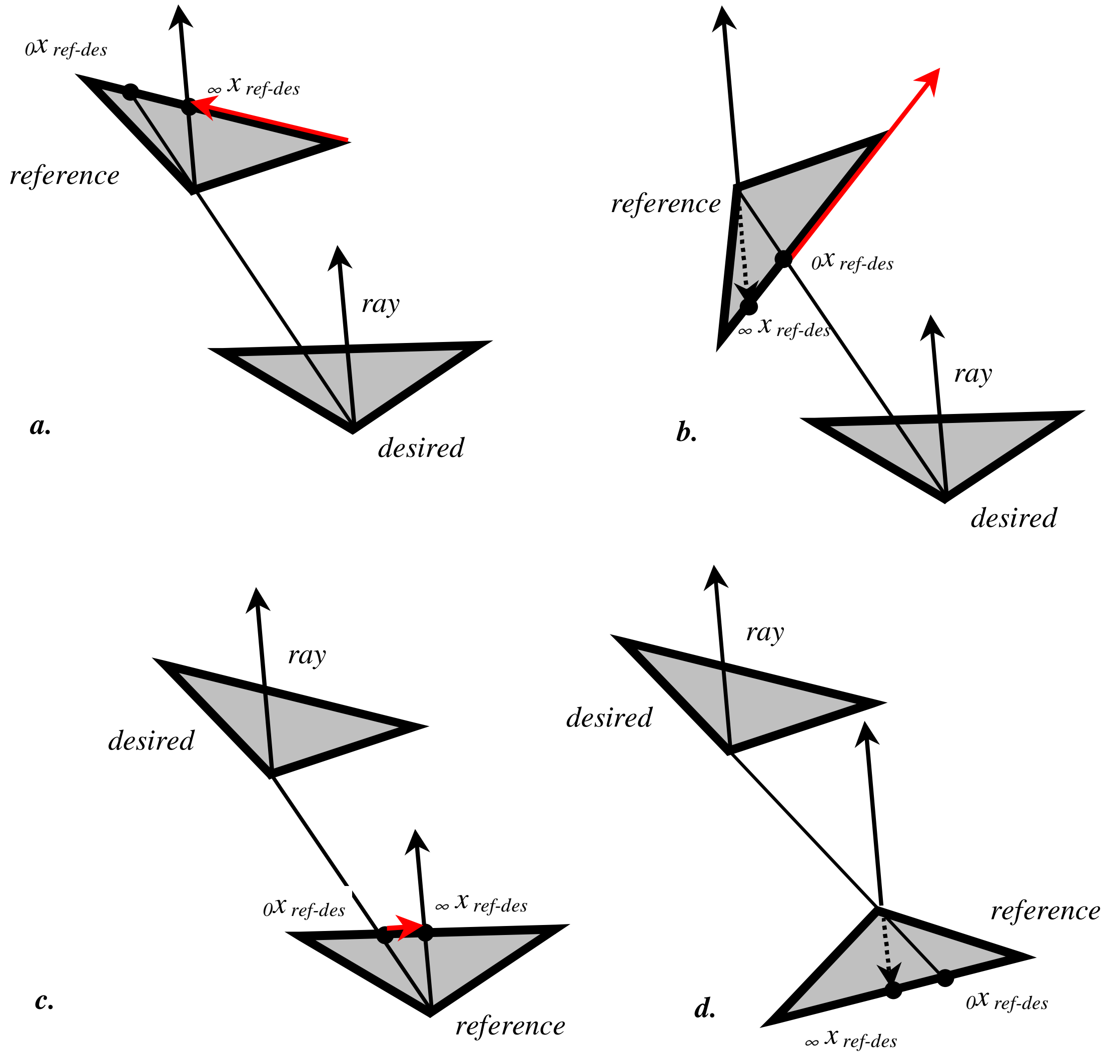

Figure 3.7 Valid segments of epipolar lines. The valid segments of the epipolar line in the reference image depend on the position of the desired camera center of projection with respect to the reference image and the direction of the desired ray. The valid segment for each case is shown in red.

In the case, when the projected rays are parallel to each other, there are two cases (Figure 3.8). These cases are based on the direction of the desired ray with respect to the reference image plane. If ${}_{\infty}x_{ref\text{-}des\ 3}$ is positive then the valid direction is from ${}_{0}x_{ref\text{-}des}$ (at infinity) to the ${}_{\infty}x_{ref\text{-}des}$. If ${}_{\infty}x_{ref\text{-}des\ 3}$ is negative then there is no valid extent of the epipolar line.

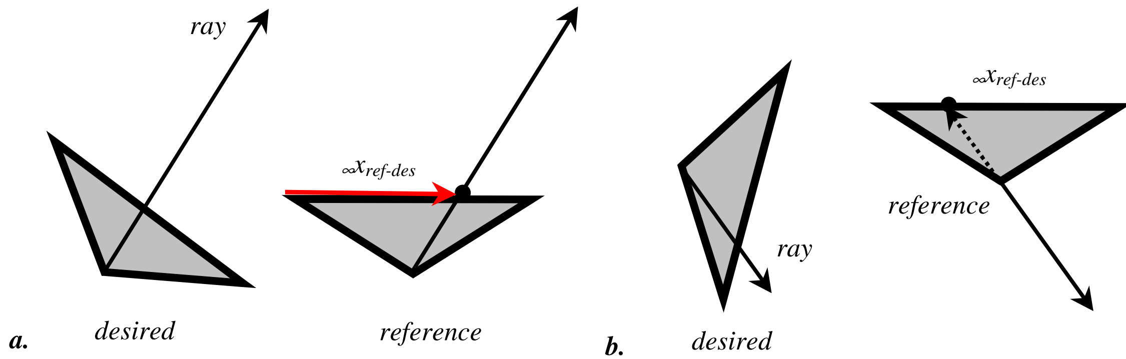

Figure 3.8 Valid segments of epipolar lines when the epipolar lines are parallel to each other.

3.6 Coarse-to-Fine Sampling

The efficiency of the visual hull sampling algorithms can be improved using standard ray casting optimizations. One of the most effective optimizations is a two pass coarse-to-fine strategy. In the first (coarse) pass the visual hull is sampled at low resolution. Then, in the second (fine) pass, the visual is sampled at higher resolution but only on the boundary of the visual hull (Figure 3.9). For the rays in the interior of the visual hull bilinear interpolation is used to obtain the front depth values and the remaining intervals are copied from the neighboring computed rays. Using this strategy only a fraction of the rays needs to be computed at high resolution.

Implementation of the coarse-to-fine sampling in the Wedge-Cache algorithm is straightforward. However, in case of the Edge-Bin algorithm some extra bookkeeping is required to take the advantage of incremental computation. During the first pass each ray also needs to store its bin position in each of the reference images. This allows to start the incremental search for the correct bin during the second pass.

In practice, in the first pass every k^{th} ray in the scanline and rays in every k^{th} scanline are sampled. The first pass divides the desired image into squares with width and height equal to k. Based on the values of the ray samples at each vertex of the square the second pass proceeds according to the three cases outlined below:

(1) If there are no visual hull samples along each of the four rays, assume that the space contained within this square is empty. Assign no sample values for all rays in the square. The assumption is not always correct e.g. there could be some small part of the visual hull contained entirely in this square.

(2) If there are visual hull samples along each of the four rays, then assume that the space contained within this square is full. Bilinearly interpolate the front values and copy the rest of the ray intervals. The described assumption is also not always valid e.g. there could be some small hole in visual hull in this square.

(3) If some of the four rays contain the sample values and some do not then assume that this square contains the visual hull boundary. Therefore, compute the visual hull samples for all the rays in this square.

Figure 3.9 Coarse-to-Fine Sampling. In the first pass one samples at low resolution (red dots). In the second pass one samples at high resolution but only on the visual hull boundary.

3.7 Computation of the Visual Hull Surface Normals

This section describes how to compute exact visual hull surface normals for the interval endpoints of the sampled visual hull representation. Surface normals are useful when reconstructing and properly shading surfaces. It is possible to estimate high quality

normals from a visual hull. This is due to the fact that an object's silhouette defines a set of tangent planes that graze the object's surface. At these surface points the normal of the tangent plane is also the surface normal. Thus, visual hulls provide a direct measurement of the surface normals for some set of points on the object's surface.

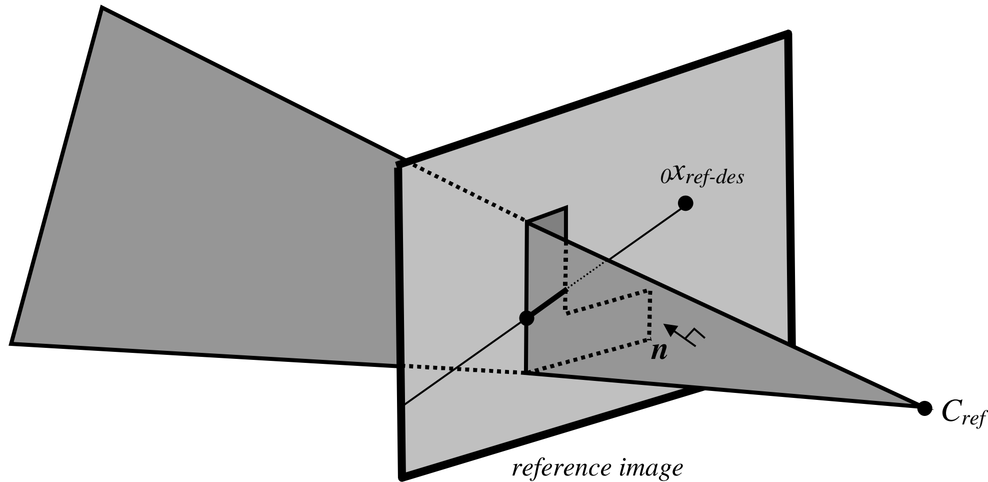

Figure 3.10 Surface normal computation. Each silhouette edge defines a wedge in 3D. An IBVH normal for some sample endpoint is the same as the normal of the 3D wedge that defines it.

In order to compute the normal some extra bookkeeping is needed. For each interval endpoint a pointer to the silhouette edge and the reference image that determine this interval endpoint is stored. Each interval is then defined as $\{ \{depth_{start}, edge_{i,m}\}, \{depth_{end}, edge_{j,n}\} \}$, where i, j are the indices of the reference images and m, n are the silhouette edge indices in the reference images. Each of the stored edges and the center of projection of the corresponding reference image determine a plane in 3D (three points, two endpoints of the edge and the center of projection, define a plane in 3D). The normal of this plane is the same as the surface normal of the point on the visual hull (Figure 3.10). The plane normal can be computed using the cross-product of the two vectors on this plane. These vectors are the vectors from the center of projection of the reference camera to the two endpoints of the silhouette edge. This leaves two choice of normals (differing in sign); the proper normal can be chosen based on the direction of the sampling ray and the knowledge of whether the sampling ray is entering or leaving the visual hull. If the ray is entering the visual hull pick the direction such that the angle

between the ray and the normal is greater than 90 degrees. If the ray is leaving the visual hull pick the direction such that the angle between the ray and the normal is less than 90 degrees.

3.8 Orthographic Camera

This section discusses how to extend the visual hull sampling algorithm to uniformly sampled grids. Uniform sampling of the visual hull corresponds to the use of the orthographic camera model instead of the pinhole (projective) camera model. This means that all rays are parallel to each other and the distance between them does not change.

3.8.1 Orthographic Sampling Notations

Orthographic sampling is defined using three vectors and a point: vectors u and v define the sampling plane. The vector u corresponds to the difference between two consecutive rays in the scanline. The vector v corresponds to the difference between rays in two consecutive scanlines. The point P defines the origin of the sampling plane. The vector d is the direction of the sampling rays (it is a cross product of vectors u and v).

3.8.2 Ray Projections in Orthographic Sampling

In case of the orthographic sampling all rays project to lines in the reference images. There are two cases how the projected lines are positioned relative to each other:

(1) if the direction of the sampling rays is not parallel to the reference image plane then all projected lines intersect at a single point on the reference image plane. This point is the so-called vanishing point.

(2) if the direction of the sampling rays is parallel to the reference image plane then all projected lines in the reference image are parallel to each other.

If d is the direction of the sampling rays then the vanishing point v is given by the following formula:

$$v_{ref\text{-}des} = P_{ref}^{-1}\, d.$$

3.8.3 Edge-Bin and Wedge-Cache Algorithms in Orthographic Sampling

There are almost no modifications to the Edge-Bin algorithm for the orthographic sampling. Bin construction is very similar to the projective sampling. In case, where all projected rays intersect at the vanishing point $v_{ref\text{-}des}$, the vanishing point takes the role of the epipole in defining the parameterization. In case, where all projected rays are parallel to each other, the parameterization is exactly the same as in the projective case – values of parameter s at intersection points of the epipolar lines with the line perpendicular to them. The efficient algorithm for one scanline of the desired sampling plane proceeds as follows: (1) determine the monotonicity type for the scanline; (2) find the corresponding bin for the first pixel in the scanline using binary search; (3) next, move forward (in case of increasing values of parameter s) or backwards (in case of decreasing values of parameter s) in the sorted list of bins to find the correct bin for each projected ray; (4) compute the analytical intersections of each projected ray with all edges in the corresponding bin.

Similarly, there are almost no changes to the Wedge-Cache algorithm. The only difference is that the vanishing point $v_{ref\text{-}des}$, takes the role of the epipole in defining the wedges in the reference image.

3.8.4 Selecting Valid Segments of Epipolar Lines

The segments of the sampling rays behind the sampling plane or the reference image planes are assumed to be invalid. Therefore, one needs to identify the projection of only valid segments of the sampling ray with respect to each reference image. There are two cases. In the first case, when the projected rays are not parallel to the reference image plane, there are four subcases. These subcases are based on:

(1) position of the start of the desired ray with respect to reference image plane that is determined by the third homogenous coordinate of p_{start}, where p_{start} is defined according to the formula:

$$p_{start} = P_{ref}^{-1} (P_{start} - C_{ref})$$

If $p_{start\ 3}$ is positive then the start of the desired ray is in front of the reference image plane. If $p_{start\ 3}$ is negative then the start of the desired ray is behind the reference image plane.

(2) sampling direction with respect to the reference image plane.

The four cases are given below:

(1) ($p_{start\ 3} < 0$) & ($v_{ref\text{-}des\ 3} > 0$) (Figure 3.11 a)

The valid extent of the epipolar line is the half-line ending at $v_{ref\text{-}des}$ in the direction towards p_{start}.

(2) ($p_{start\ 3} > 0$) & ($v_{ref\text{-}des\ 3} < 0$) (Figure 3.11 b)

The valid extent of the epipolar line is the half-line from p_{start} in the direction opposite of $v_{ref\text{-}des}$.

(3) ($p_{start\ 3} > 0$) & ($v_{ref\text{-}des\ 3} > 0$) (Figure 3.11 c)

The valid extent of the epipolar line is the segment connecting p_{start} to $v_{ref\text{-}des}$ in the reference image.

(4) ($p_{start\ 3} < 0$) & ($v_{ref\text{-}des\ 3} < 0$) (Figure 3.11 d)

There is no valid extent of the epipolar line. No point is visible in the reference image that falls into extent of the desired ray.

In the second case, when the projected rays are parallel to the reference image plane, there are two subcases. These subcases are based on the position of the ray with respect to the reference image plane. This can be determined by the third homogenous coordinate of p_{start}, projection of the start of the desired ray onto the reference image plane. The two cases are given below:

(1) If ${}_3p_{start}$ is negative then the ray is behind the reference image plane and there is no valid extent of the epipolar line (Figure 3.12 a).

(2) If ${}_3p_{start}$ is positive then the ray is in front of the reference image and the projection of the desired ray is a half-line starting at p_{start} (Figure 3.12 b). The only issue left is to determine the direction of the projected ray on the reference image plane. This is obtained by projecting the sampling direction vector onto the reference image plane.

$$x_{dir} = P_{ref}^{-1} d$$

The direction of the desired ray is $[{}_1x_{dir},\ {}_2x_{dir}]$ (the third homogenous coordinate ${}_3x_{dir}$ is zero).

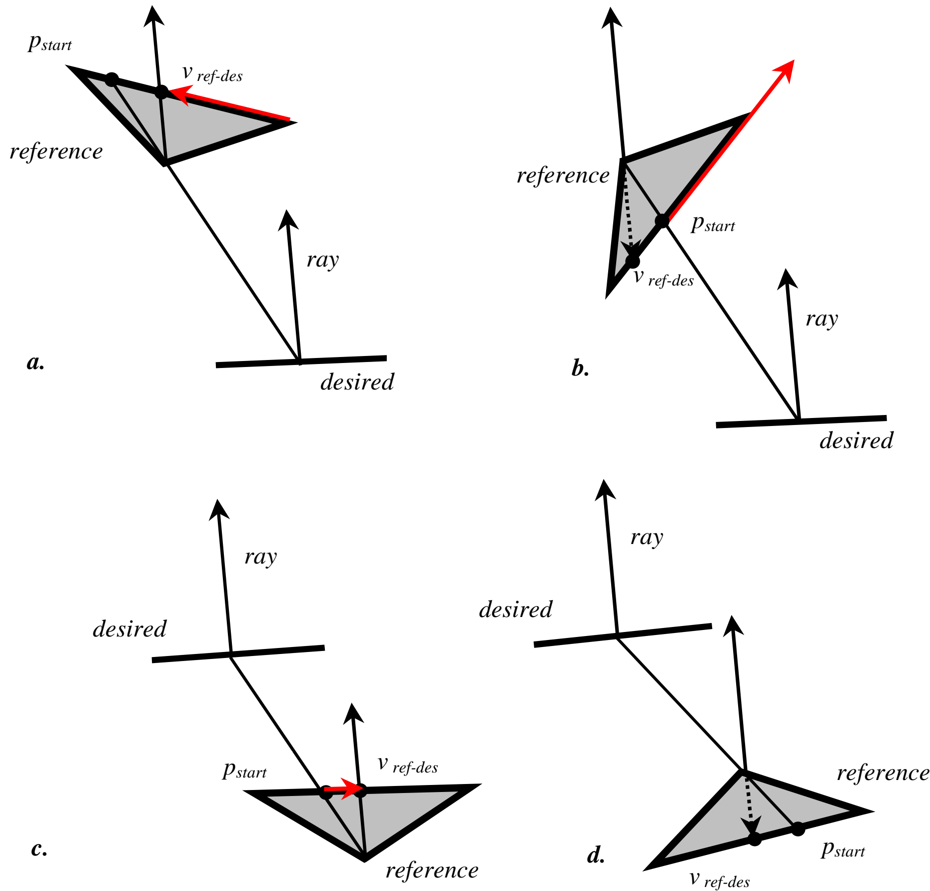

Figure 3.11 Valid segments of epipolar lines in orthographic sampling. In case of the orthographic sampling the valid segments of the epipolar line in the reference image depend on the position of the start of the desired ray with respect to reference image plane and the direction of the desired ray with respect to the reference image plane. The valid segments for each of the four cases are shown in red.

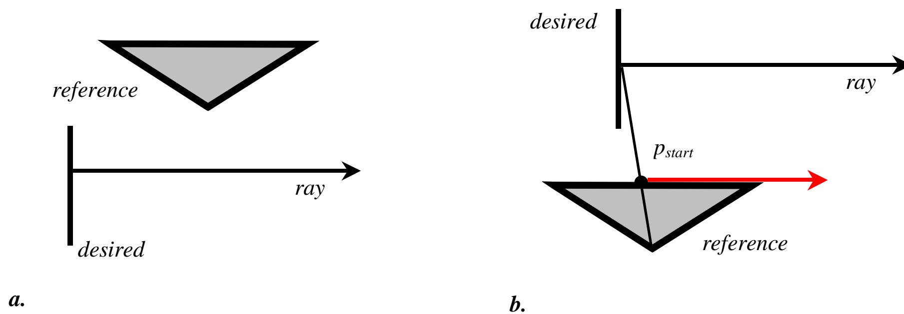

Figure 3.12 Valid segments of epipolar lines in orthographic sampling when the epipolar lines are parallel to each other. In case (a) the ray is behind the reference image; therefore, it is not visible. In case (b) the ray is in front of the reference image plane. The valid portion of the epipolar line is marked in red.

3.9 Time Complexity Analysis

The following symbols and notations are introduced to analyze the time complexity of the IBVH algorithms:

k – the number of reference images and silhouettes,

n – the dimensions of the reference images (for simplicity width and height are assumed to be the same),

m – the dimensions of the desired image (also width and height are assumed to be the same),

e – the number of silhouette contour edges in each reference image,

l – the average number of times a projected ray intersects silhouette contour.

3.9.1 Time Complexity Analysis of IBVH Sampling using Bin-Edge Algorithm

First, the time complexity of the efficient `IBVHisect` that uses Bin-Edge algorithm is analyzed. Computing contour in each reference image takes $O(n^2)$ operations. Constructing bins requires sorting slopes of e epipolar lines intersecting the vertices of the silhouette and putting edges in each bin. These steps take $O(e \log e)$ *and* $O(le)$ respectively. Adjusting bin position for each scanline traversal takes at most $O(e)$ since there are e bins. The amortized cost of each `calc2Dintervals` is $O(l)$ since there is on average l edges in each bin. Therefore, the total running time for each reference image is $O(n^2 + e \log e + le + lm^2 + me)$. Thus, for k reference images the time complexity is $O(kn^2 + ke \log e + kle + klm^2 + kme)$. Under the assumption that $m \sim n$ (the resolution of the desired and reference images is comparable) the expression reduces to $O(ke \log e + kle + kln^2 + kne)$. Furthermore, if $e = O(ml)$ (the number of edges in a silhouette is equal to the resolution of the image times the average number of times a projected ray intersects silhouette contour) then the time complexity is $O(knl \log nl + kl^2n + kln^2)$. In practice l is much less than m; therefore, the final time complexity is $O(kln^2)$.

3.9.2 Time Complexity Analysis of IBVH Sampling using Wedge-Cache Algorithm

The time complexity of the `IBVHisect` that uses Wedge-Cache algorithm is slightly different. The number of pixels in each wedge is $O(n)$. There are $O(n)$ wedges. Therefore, the total time for traversing all wedges in a single reference image is $O(n^2)$. The average number of silhouette edges in each wedge is l. Thus, the time to compute the intersection of the projected ray with one silhouette is $O(l)$. This gives the running time $O(n^2 + m^2 l)$ for a single reference image. Then for k reference images the time complexity is $O(kn^2 + km^2 l)$. Assuming that $m \sim n$ the expression reduces to $O(kn^2 l)$. This is the same as the complexity of the `IBVHisect` that uses Bin-Edge algorithm.

Chapter 4

Visual Hull Shading

4.1 View-Dependent Shading

In order to determine the proper color for the front most points of the sampled visual hull described in the previous chapter I use texture information from the input images. My texturing strategy is view-dependent – the color of the front most points depends on the position of the virtual camera. First, for each front-most point a list of reference images is determined where this point is visible. Then, the color from the reference image that is the most similar to this virtual view is chosen. The angle between the ray from the visual hull point to the desired camera center and the ray from the visual hull point to the desired camera center is used to measure the similarity [5] [6] [19] (Figure 4.1).

There are potentially other similarity measures. For example, one could prefer the views that see the surface point from a closer range. The reasoning behind this strategy is that the views from the closer range capture more texture detail than the views from the further distance. In my setup all cameras are at approximately the same distance from the object; therefore, I employ the first strategy. The strategies outlined above fall into the nearest-neighbor category. They are simple and, therefore, fast to compute. It is also possible to use more sophisticated techniques. One could interpolate/blend the color samples in the neighboring views or even reconstruct a radiance function for each surface point.

More elaborate reconstruction strategies will not work in my setup for the following reasons: (1) there is only a small number of reference views (typically 4); therefore, there are few color samples for each point on the visual hull surface. (2) The visual hull geometry might be not the same as the real surface of the object; therefore, the color samples of a point on the visual hull might come from different points on the

surface on the object. Interpolation/blending these samples usually results in the loss of the quality. The pseudocode for the visual hull shading algorithm is given below:

```
IBVHshade(intervalImage &d, refImList R)
(1)    for each pixel p in d do
(2)       p.best = BIGNUM
(3)    for each referenceImage r in R do
(4)       for each pixel p in d do
(5)          point3 pt3 = computeFrontMostPoint(p,d)
(6)          if isVisible(pt3,r,d)
(7)             double s = cosAngle(pt3,d.cop,r.cop)
(8)             if (s < p.best)
(9)                point2 pt2 = project(pt3,r.camInfo)
(10)               p.color = sampleColor(pt2,r)
(11)               p.best = s
```

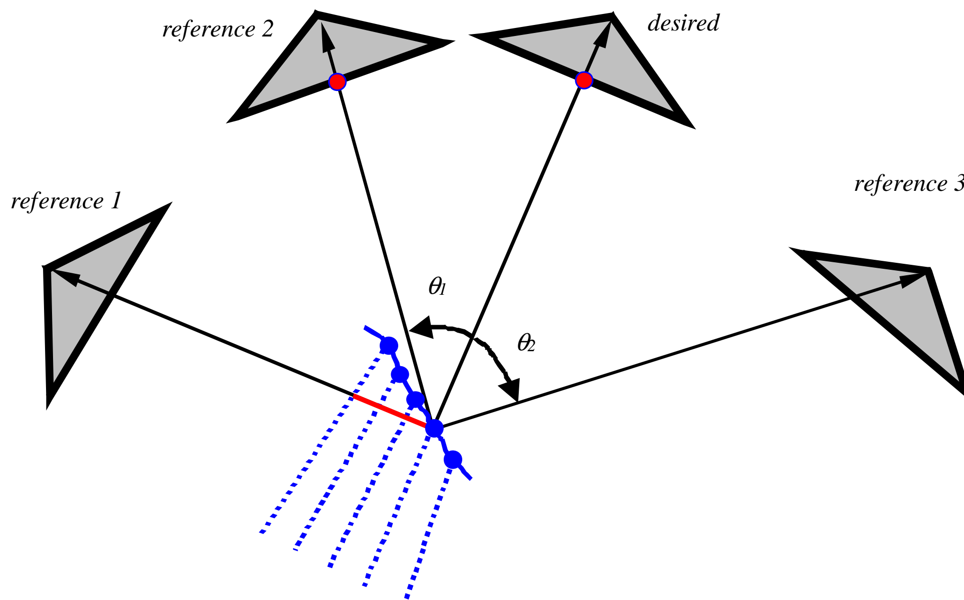

Figure 4.1 View-dependent texturing strategy. In the example there are three reference cameras. For each front most point on the visual hull (blue line segments) compute the rays to each camera center. Pick the image for which the angle between the ray to the desired camera and the ray to the reference camera is the smallest (in this case reference image 2). The images (reference image 1) that do not have a view of the visual hull point cannot be used.

The function `computeFrontMostPoint` determines the location of the front most point on the visual hull for the particular pixel (ray) in the desired image. It is computed according to the formula:

$$X = C_{des} + t\, P_{des}\, p$$

where, C_{des} is the center of projection for the desired camera, P_{des} is the projection matrix of the desired camera, p is the location of the desired pixel in homogenous coordinates, and t is the start of the first interval that is computed during the visual hull sampling. The details of the `isVisible` function that determines whether a given point is visible in a reference view are described in the next section. The function `cosAngle` computes two vectors: from the sample point to the reference camera's center of projection C_{ref} and from the sample point to the desired camera's center of projection C_{des}. Then, instead of computing the angle between these two vectors it is easier to compute its cosine. This is done by normalizing each vector and computing a dot product. The function `project` computes the coordinates of the projected 3D point on the reference image plane. The following formula is used:

$$x = P^{-1}_{ref} (X - C_{ref})$$

However, the coordinates of the projected point have floating point values. The function `sampleColor` determines the coordinates of the four closest pixels in the reference image to the projected point. Then, it bilinearly interpolates the color values of these pixels based on the location of the projected point.

4.2 Time Complexity Analysis of the Shading Algorithm

Let k be the number of reference images, and n^2 be the number of pixels in the desired image. If v denotes the running time of the `isVisible` subroutine then the worst case running time of the shading algorithm is $O(n^2 v)$ for one reference image. This yields $O(kn^2 v)$ running time for k reference images.

4.3 Visibility Algorithm

The following sections describe the novel algorithm that computes visibility of the front-most points of the visual hull with respect to the reference images. Before describing the image-based visibility algorithm a set of assumptions needs to be outlined. First, since the true geometry of the scene is not known the visibility is computed based on the sampled visual hull geometry. Second, the visual hull representation needs to be conceptually changed. The scene is represented as a set of line intervals in 3D. In this

representation one can see through the whole sampled volume; all endpoints (except the few that lie exactly behind the lines) are visible from any direction. Therefore, for the purpose of the visibility each of the intervals is considered to be a solid frustum with a square cross-section.

4.3.1 Naïve Approaches

This visibility problem is equivalent to the shadow determination problem where one places a point light source at each reference camera position and the goal is to determine which parts of the scene are illuminated by the light and which parts lie in a shadow. Standard graphics algorithms are not directly applicable to the visual hull sampled representation since they rely on the polygonal or solid representation of the scene. There are at least three standard algorithms for solving shadow determination problem: shadow maps, shadow volumes, and ray casting. In this case one would have to create one geometric primitive for each visual hull interval. Since the number of these intervals is large the running time of these algorithms is excessive and the computation cannot be done in real-time. In the next section I describe how to perform the visibility test efficiently using epipolar geometry.

4.3.2 Reduction to Visibility on Epipolar Planes

First, I describe how the problem of computing continuous 3D visibility can be reduced to a set of 2D visibility calculations within epipolar planes. An epipolar plane is a plane that contains both camera centers of projection (a desired and a reference camera center). Visibility relationships cannot interact between two such planes. Figure 4.2 shows one such a plane. The visual hull samples on rays residing on different epipolar planes cannot occlude each other. Therefore, one breaks the rays in the desired image into sets such that rays in each set are on the same epipolar plane. Then the visibility for the rays in each set is considered separately.

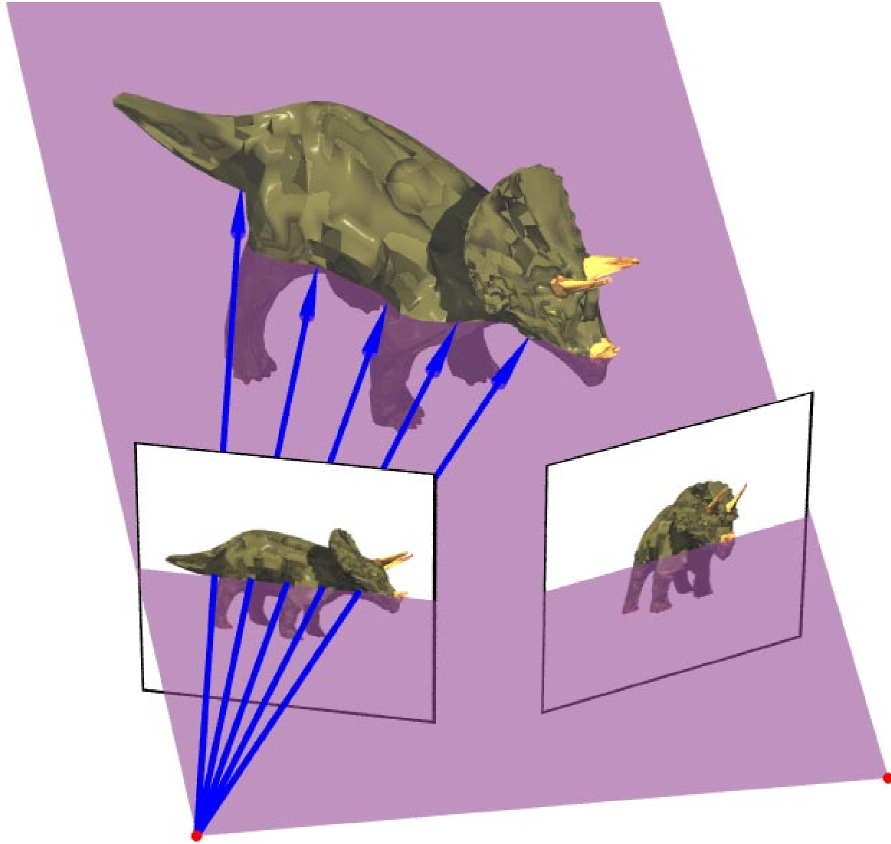

Figure 4.2 Reduction to visibility on epipolar planes. One can break the 3D visibility calculation into a set of 2D visibility calculations. Each 2D visibility computation corresponds to determining visibility of the IBVH samples on an epipolar plane, a plane that containing both the reference and desired image centers of projections.

4.3.3 Front-to-Back Ordering

Next, I show how to efficiently determine front-to-back ordering of the samples on the epipolar plane. Notice that the ordering of the visual hull samples on the epipolar plane with respect to the reference image can be determined based on an occlusion compatible ordering [18]. There are two cases that need to be considered. They depend on the position of the reference image center of projection with respect to the desired image plane.

(1) If the reference image's center of projection is in front of the desired image plane then the samples on the rays corresponding to the pixels further away from the epipole ${}_{0}x_{ref\text{-}des}$ (projection of reference center of projection onto desired image

plane) cannot occlude the samples on the rays corresponding to the pixels closer to the epipole ${}_0x_{ref\text{-}des}$ (Figure 4.3 a).

(2) If the reference image's center of projection is behind the desired image plane then the samples on the rays corresponding to the pixels closer to the epipole ${}_0x_{ref\text{-}des}$ cannot occlude the samples on the rays corresponding to the pixels further away from the epipole ${}_0x_{ref\text{-}des}$ (Figure 4.3 b).

If the epipole ${}_0x_{ref\text{-}des}$ is inside the desired image pixels on two sides of the epipole have to be considered separately. There are no visibility interactions between the samples on the rays corresponding to pixels on different sides of the epipole. Furthermore, observe that there are no visibility interactions between samples on the same ray except the case when the ray corresponds to the line joining the centers of projection of the desired and reference image.

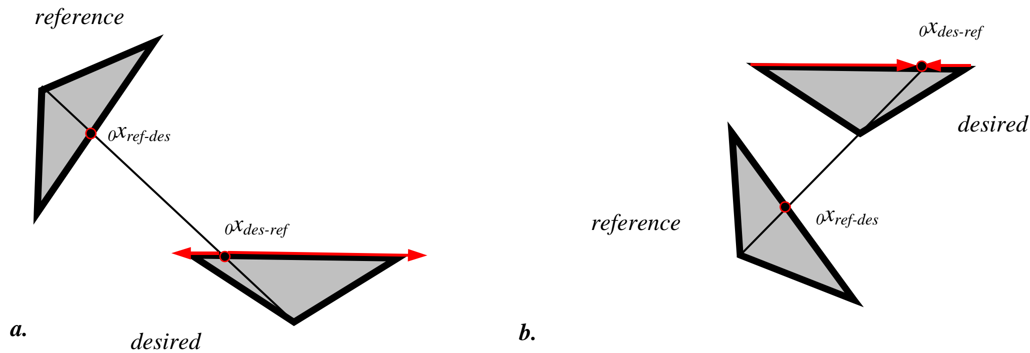

Figure 4.3 Front-to-back ordering on an epipolar plane. Front-to-back ordering of the rays in the desired image depends on the position of the reference image center with respect to the desired image. The correct ray traversal in a front-to-back order for each of the two cases is shown using red arrows.

4.3.4 Efficient 2D Visibility Algorithm

This section describes the efficient algorithm for determining visibility information for samples on one epipolar plane. First, one traverses the desired-view pixels in front-to-back order with respect to reference image *r*. During this traversal, the algorithm accumulates coverage intervals by projecting the IBVH pixel intervals into the reference view, and forming the union of them. For each front-most point, *pt3*, one checks to see if its projection in the reference view is already covered by the coverage

intervals computed thus far. If it is covered, then *pt3* is occluded from *r* by the IBVH. If it is unoccluded, then *pt3* is unoccluded from *r* by the IBVH (Figure 4.4).

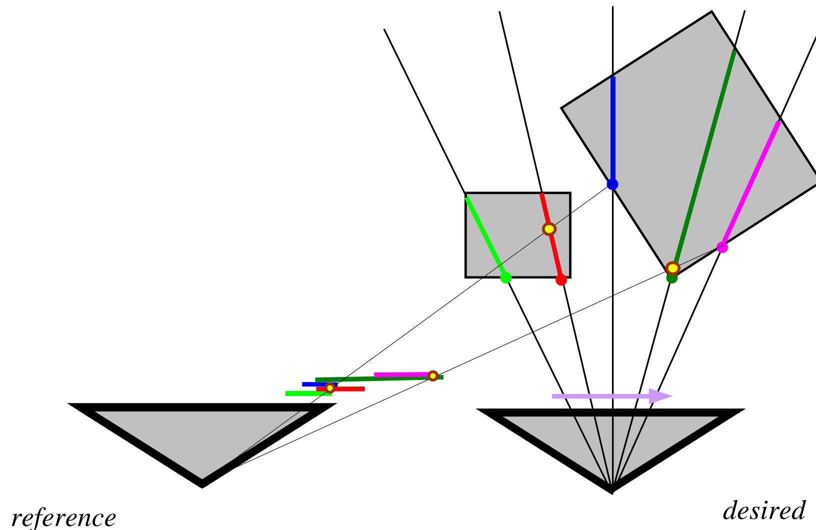

Figure 4.4 An example of the 2D visibility algorithm execution. Traverse the rays in the desired image from left to right. As the rays are traversed their intervals are projected onto the reference image and the coverage mask (the union of all projected intervals up to this point) is computed. The IBVH point is declared visible when its projection onto the reference image does not fall into the coverage mask of all rays processed earlier. In the example the front most point of the third and fifth ray are not visible in the reference image.

During the visual hull sampling one computes the normal at each interval endpoint point. Therefore, it is possible to perform backface culling of the points. It has to be true that the angle between the sample normal and the ray from the sample point to the reference camera center of projection has to be less than 90° in order for the sample to be visible. This test is usually redundant (if the containment test outputs visible then back-face test also outputs visible). However, there are a few cases where this is not true (the containment test outputs visible but the back-face test outputs not-visible). These cases arise when (1) visibility of the first sample is computed (according to the containment test the first sample is always visible) or (2) when the scene is not sampled densely enough. These cases are illustrated in Figure 4.5. The pseudo-code for the 2D visibility algorithm follows.

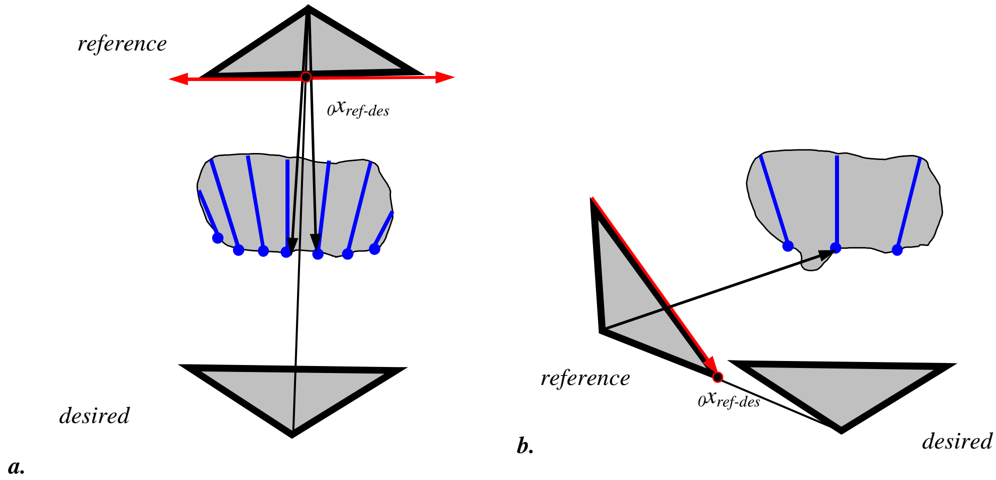

Figure 4.5 Incorrect visibility determination. In case (a) it is possible to see-through the sampled volume (the problem can be resolved using backface culling). In case (b) the sampling skipped some portion of the object (this problem can be reduced by decreasing distance between two consecutive rays).

```
visibility2D(IntervalImage &d, referenceImage r)
(1)   point1D  epipole = project(r.COP, d.caminfo)
(2)   intervals coverage = <empty>
(3)   \\ proceed front to back with respect to r
(4)   for each pixel p in d on the right side of the epipole
(5)      ray2D ry2 = compute2Dray(p,d.camInfo)
(6)      point2 pt2 = front(p.intervals,ry2)
(7)      vector2D n = pt2.normal
(8)      vector2D v = r.COP - pt2
(9)      angle a =  angle(n, v)
(10)     point1D p1 = project(pt2,r.camInfo)
(11)     if contained(p1,coverage) OR  angle ∉ <0° , 90)
(12)        p.visible[r] = false
(13)     else
(14)        p.visible[r] = true
(15)     intervals tmp = projectIntervals(p.intervals,ry2,r.camInfo)
(16)     coverage = coverage UNION tmp
(17)  \\ proceed front to back with respect to r
(18)  for each pixel p in d on the left side of the epipole
(19)     ray2D ry2 = compute2Dray(p,d.camInfo)
(20)     point2 pt2 = front(p.intervals,ry2)
(21)     vector2D n = pt2.normal
(22)     vector2D v = r.COP - pt2
(23)     angle a =  angle(n, v)
(24)     point1D p1 = project(pt2,r.camInfo)
(25)     if contained(p1,coverage) OR  angle ∉ <0° , 90)
(26)        p.visible[r] = false
(27)     else
(28)        p.visible[r] = true
(29)      intervals tmp = prjctIntrvls(p.intervals,ry2,r.camInfo)
(30)      coverage = coverage UNION tmp
```

4.3.5 Extension to the Discrete 3D Algorithm

So far I have described how the continuous 3D visibility can be reduced to a set of 2D visibility computations. The extension of the discrete 2D algorithm to a complete discrete 3D solution is non-trivial, as most of the discrete pixels in images do not exactly share epipolar planes. Consequently, one must be careful in implementing 3D visibility.

To determine visibility of a pixel p correctly one needs to consider a set S_p that contains all possible epipolar lines that touch p. There at least two possible criteria for determining whether p is visible:

(1) If p is visible along **all** lines in S_p then p is visible otherwise it is invisible.

(2) If p is visible along **any** line in S_p then p is visible otherwise it is invisible.

Other criteria could e.g. label a pixel p visible if p is visible along at least a fraction f of all epipolar lines in S_p. Clearly the first definition produces more pixels that are labeled not visible. Therefore, it is better suited for use with a large number of reference images because it is more likely that the pixel was visible in at least one of them. With a small number of reference images the second definition is preferred. In the current real-time system there are only four video cameras. Thus, the approximation algorithm for the second definition of visibility is used.

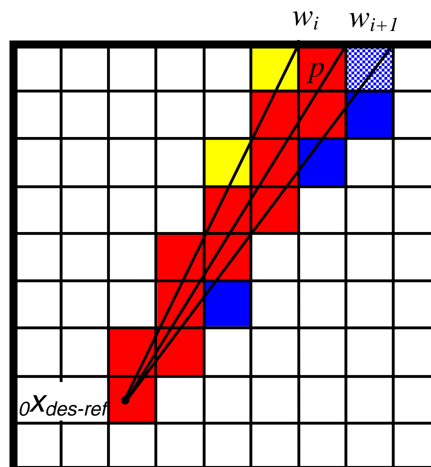

Figure 4.6 Approximate visibility algorithm (definition 1). Two wedges w_i and w_{i+1} are shown. Pixels that belong to both are colored red; pixels that belong only to w_i are colored yellow; pixels belong only to w_{i+1} are colored blue. To determine visibility of pixel p one needs to consider all solid red, yellow, and blue pixels in wedges w_i and w_{i+1}.

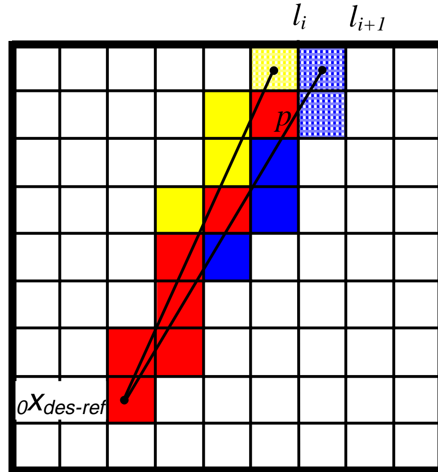

Figure 4.7 Approximate visibility algorithm (definition 2). Two lines l_i and l_{i+1} are shown. Pixels that belong to both are colored red; pixels that belong only to l_i are colored yellow; pixels that belong only to l_{i+1} are colored blue. To determine visibility of pixel *p* the algorithm considers all solid red, yellow, and blue pixels on lines l_i and l_{i+1}.

Implementing exact 3D visibility algorithms for these visibility definitions is difficult. Therefore, approximate algorithms are used. These approximations have the property that if the pixel is truly invisible it is never labeled visible. However, the algorithms could label some pixel invisible though it is in fact visible.

4.3.5.1 Approximate Visibility Algorithm (Definition 1)

An approximation algorithm that conservatively computes visibility according to the first definition operates as follows. Define an epipolar wedge starting from the epipole $_0x_{des\text{-}ref}$ in the desired view and extending to a one pixel-width interval on the image boundary. Depending on the position of the epipole in the desired image plane with respect to the image boundary there are nine possible cases of image boundary parts that should be used to define the wedges. These cases are illustrated in Figure 4.8.

Depending on the relative camera views, traverse the wedge in the front-to-back order (either toward or away from the epipole). For each pixel in this wedge, compute visibility with respect to the pixels traversed earlier in the wedge using the 2D visibility algorithm (Figure 4.6). If a pixel is computed as *visible* then no geometry within the wedge could have occluded this pixel in the reference view. Use a set of wedges whose

union covers the whole image. A pixel may be touched by more than one wedge. In these cases its final visibility is conservatively computed as the *AND* of the results obtained from each wedge.

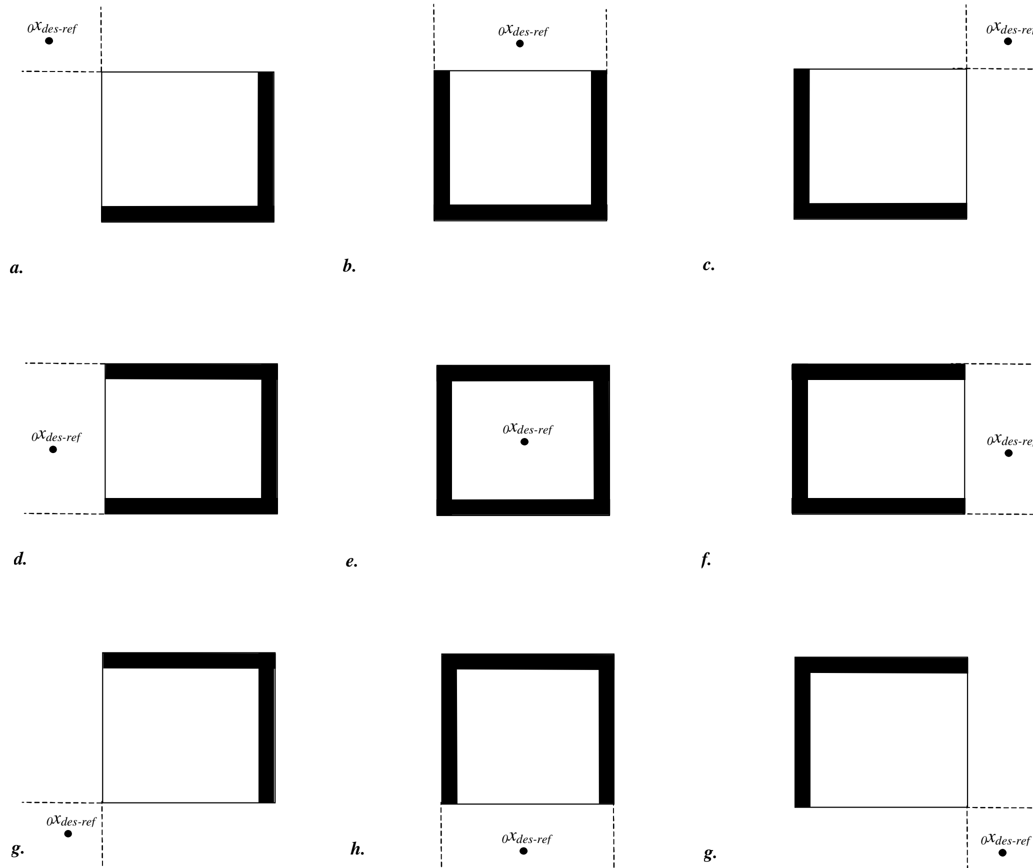

Figure 4.8 Desired image boundaries used to define wedges and lines. Depending on the position of the epipole in the desired image different parts of the desired image boundary (the black thick lines) are used to define wedges (Approximate Visibility Algorithm – Definition 1) and lines (Approximate Visibility Algorithm – Definition 2).

4.3.5.2 Approximate Visibility Algorithm (Definition 2)

An approximation algorithm for the second visibility definition operates as follows. From the set of all possible epipolar lines that touch pixel p only some subset of them is considered. I chose a subset such that at least one line touches each pixel. One subset with this property consists of all epipolar lines that pass through the centers of the image boundary pixels. Depending on the position of the epipole in the desired image there are nine cases of image boundary pixels that need to be used to define the epipolar

lines (Figure 4.8). The pixels touched by all the lines in the subset are all the pixels in the desired image; denser subsets can also be chosen. The approximation algorithm computes `visibility2D` for all epipolar lines in the subset. Visibility for a pixel might be computed more than once (e.g., the pixels near the epipole are traversed more often). The *OR* operation is performed on all obtained visibility results (Figure 4.7).

4.3.6 Time Complexity Analysis of the Visibility Algorithm

First, I analyze the time complexity of the `visibility2D` algorithm. Let m be the width and height of the desired image and let l be the average number of intervals along the desired ray. There are at most $O(m)$ pixels in a wedge or in an epipolar line in the desired image. Since one traverses each pixel once, and containment and unioning can be done in $O(l)$ time `visibility2D` runs in $O(ml)$. In the first version of the approximation algorithm there are at most $4m$ wedges. Similarly in the second version of the approximation algorithm there are up to $4m$ epipolar lines that have to be traversed. Therefore, the running time of the 3D visibility for each reference image is $O(lm^2)$ – this gives $O(l)$ running time for each desired ray. Since this operation needs to be performed for k reference images the total running time of the visibility algorithm is $O(klm^2)$. This is the same as the time complexity of the visual hull sampling algorithm.

Chapter 5

Distributed Algorithms

In this section I show that both the sampling and shading algorithms can be easily parallelized. The running time of the sampling, shading, and visibility algorithms described in chapters 3 and 4 is proportional to the number of reference images. I show that using distributed or multiprocessor system organized in a tree-structured network this factor in the IBVH computation can be reduced from linear to logarithmic time.

5.1 Node Types

In the distributed version of the algorithm there are the following different computational entities:

(1) an IBVH consumer node – this is the user computer that ultimately displays the shaded IBVH image,

(2) Image producer nodes – these are computers connected to the cameras that produce the segmented video stream,

(3) Intermediate nodes – these are the computers that carry out the IBVH computation.

5.2 Algorithm Description

The distributed IBVH computation algorithm can be divided into six phases. I describe all phases in order:

(1) View propagation

The IBVH is sampled from a specified viewpoint. Therefore, as the first step the desired view is propagated from the IBVH consumer node to all intermediate nodes.

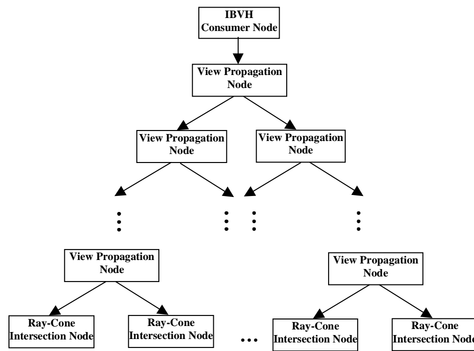

Figure 5.1 View propagation phase

(2) Ray-cone intersection

Each image producer node is connected to an intermediate node (a ray-cone intersection node). This intermediate node computes the intersection of the extruded silhouette generated by the image producer node with the rays in the desired image. Then it sends the generated IBVH to its parent intermediate node (a ray intersection node).

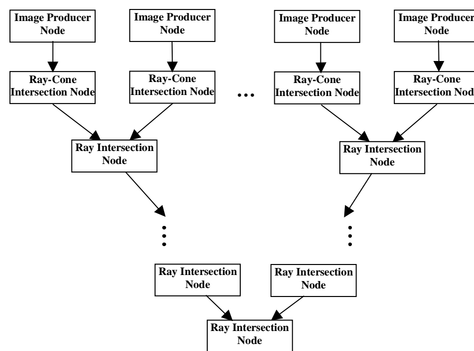

Figure 5.2 Ray-cone intersection and ray intersection phases

(3) Ray intersection

This phase uses a network of ray intersection nodes with a tree topology. Each of the ray intersection nodes is connected to a parent. Each parent node has 2 child nodes. Each leaf node of the tree is connected to the two ray-cone intersection nodes. Each ray intersection node receives two IBVH interval images. Then it computes the intersection of two IBVHs – this is simply a 1D intersection of intervals for the corresponding rays in two IBVHs. Finally, it sends the resulting IBVH to its parent node. At the end of this phase the root of the tree holds the

final IBVH image. At this point if one is only interested in sampled geometry the IBVH image can be sent to the IBVH consumer node.

(4) IBVH propagation

In order to compute shading and visibility efficiently the IBVH structure has to be propagated to *k* intermediate nodes (visibility computation nodes) where *k* is the number of input images. This operation is also performed using a tree-structured network, but this time IBVH interval image is propagated top-down – from the root node to the leaves. At the end of this phase each visibility computation node has a copy of the IBVH.

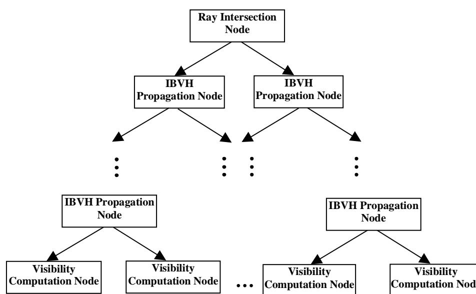

Figure 5.3 IBVH propagation phase

(5) Visibility computation

During this phase each visibility computation node determines the visibility for a specified input camera. Then it obtains the image from the corresponding image producer node. It shades each front-most IBVH pixel that is visible from the input camera. For each pixel it also stores the angle between the rays to the input and desired camera center. Finally, it sends the shaded front IBVH image to the shade selection node.

(6) Shade selection

In the final step the best shading information for the IBVH is selected. A tree-structured network of intermediate nodes – shade selection nodes is used for this step. In this tree each shade selection node receives shaded front IBVH images from its two children. For each IBVH pixel it selects the shading information from the input IBVH that has the smallest angle between the rays to the reference

and desired camera center. Finally, the new shaded IBVH image is passed to its parent node. At the very end the root of the tree obtains the final shaded image of the IBVH. This image is passed to the IBVH consumer node.

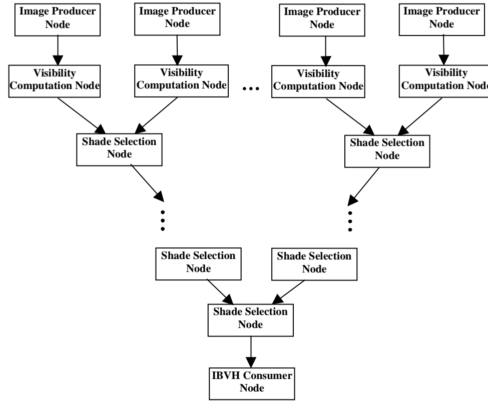

Figure 5.4 Visibility computation and shade selection phases.

The multiprocessor system is simpler than the distributed version I have just described. The steps (1) and (4) can be greatly simplified. In the view propagation phase the view information is placed in the system memory so that all processes can look it up. Similarly, the IBVH image can be also placed in the system memory in the IBVH propagation phase.

5.3 Time Complexity Analysis

In this section the time complexity of this algorithm is analyzed by considering each of the six phases. The definitions of all variables are the same as in Section 3.9. The view propagation phase takes $O(\log k)$ time steps. The ray-cone intersection phase takes $O(n^2 + m^2 l)$. The ray intersection phase takes $O(m^2 l \log k)$ since the height of the tree is $\log k$. The time complexity of the IBVH propagation phase is also $O(m^2 l \log k)$. The visibility computation phase takes $O(lm^2)$ and the shade selection phase takes $O(m^2 \log k)$. To conclude the total running time of the distributed algorithm is $O(l\, m^2 \log k + n^2)$.

Chapter 6

System Overview

In this chapter I explain the details of the real-time system that generates image based visual hulls. In section 6.1 I describe the equipment used to build the whole system. In section 6.2 I list the tasks that need to be performed to obtain the output IBVHs. Then in section 6.3 I describe the details of the server design and its implementation. Finally, in section 6.4 I specify the details of the image subtraction algorithm.

6.1 Hardware

The current system uses four Sony DFW-V500 IEEE-1394 video cameras. Each of the cameras is attached to a separate client (600 MHz desktop PC). There is also a central server (4x550MHz PC) that performs the majority of IBVH computations. The server is connected over the 100Mbit LAN network to all clients. The cameras are equipped with an external trigger mechanism. The server machine is also connected to the external trigger of the cameras.

6.2 System Tasks

Before the virtual video stream (the video sequence where an arbitrary position of the camera can be specified) can be obtained the following tasks need to be performed:

(1) External and internal parameters for all cameras have to be determined.
(2) All clients have to record and store the background scene.
(3) If one desires to composite the IBVH with some other background scene the two coordinate systems have to be calibrated with respect to each other.

In order to obtain each frame of the virtual video sequence the following steps have to take place in the specified order.

(1) The server issues an external trigger signal to all cameras.

(2) All cameras take pictures of the scene at the same time instant and send them to the clients over the IEEE-1394 connection.
(3) The clients correct the input images for radial lens distortion.
(4) The clients perform the background/foreground segmentation (using background subtraction) to obtain the foreground regions. The details of the background/foreground segmentation algorithm are described in section 6.4.
(5) The clients compress both the foreground texture and the foreground/background classification mask. JPEG compression is used for the texture and run length encoding for the classification mask.
(6) Each client sends the compressed information to the server over the LAN network.
(7) The server collects the reference frames from all cameras and decompresses them.
(8) The user selects a desired view.
(9) The server performs the visual hull sampling, shading, and visibility operations based on the current set of input images and the desired view.
(10) The server displays the IBVH image. Optionally, it can depth-composite the IBVH with some background scene.

6.3 Server Design

The server performs different tasks; therefore, it is designed as a multithreaded application. The server application has the following threads:

(1) The main network thread. This thread establishes initial connection with each client. It also creates client-server network threads.
(2) The client-server network threads. The application has one thread for each client to service the connection with this client. These threads perform the following tasks: receiving images from the client, decompressing these images, and putting them in the input-buffer queue.
(3) The IBVH computation threads. Each of these threads performs the sampling and shading operations for a single frame of the IBVH. The current server is a multiprocessor machine and to increase throughput of the system there are a few

(typically more than 4) of these threads. Each IBVH computation thread performs the following tasks. First, it checks if the element at the beginning of the input-buffer queue has images from all the clients. If it does, it removes the element from the queue. Then, it computes the image of the IBVH based on the current view. The computed image is placed at the appropriate place in the display queue. Then, the thread repeats the whole process.

(4) The display thread. This thread is responsible for displaying the images of the IBVH and depth-compositing them with the background scene. The thread constantly checks if there are any IBVH images at the beginning of the display-queue. The thread depth-composites the IBVH and background scene and displays the final image on the screen if there are no earlier IBVH images that are still being computed.

(5) The synchronization thread. The thread is responsible for issuing the trigger signal to all cameras. The trigger is issued at a constant rate (typically 10Hz).

6.4 Image Segmentation

Before it is possible to run image based visual hull algorithms their input has to be generated. The most important inputs to the image based visual hull algorithms are segmented images of the scene. There are at least two well-known techniques for segmenting objects from the background: chroma-keying (or blue-screening) and background subtraction. Chroma-keying relies on the fact that the background has some unique color (typically blue). In general, the algorithm assigns the background to the pixels in the image that have values within some threshold of the specified background color value. Chroma-keying has some significant drawbacks: (1) it requires setting up a controlled environment where the whole scene has some uniform color; (2) there is usually some amount of color bleeding from the background to the image of the object – this causes some artifacts on the texture and can confuse segmentation; (3) the objects in the scene cannot have colors similar to the background color.

Background subtraction relies on recording and storing a model of the background. Then the image with the foreground objects is compared to the stored

background image. The region is identified as the foreground where the images differ; the region is identified as the background where they are the same. This algorithm does not require setting up controlled environment. The amount of color bleeding from a typical background to the object is usually insignificant. However, the technique might identify some portion of the foreground as the background. This happens when the color of some portion of the object is very similar to the color of the background at that place. The technique might also identify some portions of the background as the foreground. This typically happens when the foreground object casts shadows onto the background. I discuss how to modify the background subtraction algorithm to minimize the described errors.

In the pre-processing stage the current background-subtraction algorithm records a sequence of *n* frames of the static background. Then for each pixel in the image its mean and standard deviation from the mean in these *n* frames are calculated. Then the actual algorithm proceeds in four stages. The outputs of each stage are the inputs to the next stage. The stages are described below:

(1) Individual difference test

In the first stage compare the value of each pixel *p* in the current frame with the mean value μ_p. If the absolute difference between the pixel value and the mean μ_p is greater than some multiple *k* (I use k =3) of the standard deviation σ_p then classify the pixel as not background ($type_{s1}$ (p) = not-background). Otherwise classify the pixel as foreground ($type_{s1}(p)$ = foreground).

(2) Shadow removal

Observe that the pixels of the background in the shadow of the object have the same brightness values relative to the neighboring pixels. Therefore, for each pixel previously classified as not-background ($type_{s1}$ (p) = not-background) a normalized correlation test can be used. If a neighborhood region centered at *p* in the current image correlates well with the corresponding region in the background image then classify the pixel *p* as background ($type_{s2}(p)$ = background) . If the region does not correlate well then classify the pixel as foreground ($type_{s2}(p)$ = foreground). All pixels classified as foreground in the stage 1 are still classified as

foreground ($type_{s2}(p)$ = foreground). For the correlation c at a pixel $p=(u_0,v_0)$ use the following formula:

$$c(u_0,v_0)=\frac{1}{K}\sum_{u_1=-N}^{N}\sum_{v_1=-N}^{N}(I_1(u_1+u_0,v_1+v_0)-I_1(u_0,v_0))^2(I_1(u_1+u_0,v_1+v_0)-I_1(u_0,v_0))^2$$

where,

$$K(u_0,v_0)=(2N+1)^2\sigma_1(u_0,v_0)\sigma_2(u_0,v_0)$$

$$\sigma_1^2(u_0,v_0)=\frac{1}{(2N+1)^2}\sum_{u_1=-N}^{N}\sum_{v_1=-N}^{N}(I_1(u_1+u_0,v_1+v_0)-\mu_1(u_0,v_0))^2$$

$$\mu_1(u_0,v_0)=\frac{1}{(2N+1)^2}\sum_{u_1=-N}^{N}\sum_{v_1=-N}^{N}I_1(u_1+u_0,v_1+v_0)$$

$I_1(u_0,v_0)$ and $I_2(u_0,v_0)$ denote intensity at pixel p in the static background image and the current frame.

(3) Hole removal

The third stage tries to minimize the effect of foreground pixels incorrectly classified as background. A majority filter (similar to the median filter) is used for each pixel in the image. If the majority of the pixels in the neighborhood of a pixel p are classified in the stage 2 as foreground pixels then the pixel p is classified as foreground ($type_{s3}(p)$=foreground). Otherwise the pixel p is classified as background ($type_{s3}(p)$=background). The third stage eliminates noise in the previous classification. It also coalesces the holes in the foreground object that result from similar color values of the background image and the foreground object.

(4) Contour refinement

The third stage of the algorithm degrades the contours of the foreground objects. Therefore, the last stage tries to reverse this effect. First, detect all pixels in the image within some distance d from the contours (boundaries between foreground and background) according to classification after the stage 3. The distance d is related to the size of the majority filter. Then for all these detected pixels revert to the classification determined by stage 2. For all other pixels (not within distance d from contour) the classification determined by stage 3 is used.

The described segmentation method works well in practice. The sample results are shown in Color Plate A.1.

Chapter 7

Results

In this chapter I describe the results obtained in various stages of the system. In section 7.1 I discuss the quality of the input video streams. Moreover, in section 7.2 I describe the performance of the background segmentation algorithm. In section 7.3 I examine the network performance and data compression/decompression used while transporting the data to the server. Finally, in sections 7.4 and 7.5 I describe the quality and speed of the described IBVH algorithms.

7.1 Input Images

The first set of pictures (Color Plate A.2) shows four input images – one from each camera. Each camera is able to capture images at the maximum 30fps at 640x480 resolution. The current system captures input images at 10 fps at 320x240 resolution. The raw input from the Sony DFW-V500 has the format YCrCb 4:2:2 (each pixel has an 8 bit luminance value, two consecutive pixels in a scanline have the same two 8-bit chrominance values). The same lens and settings are used for all cameras and the colors match well between the images from different cameras. Therefore, no special photometric calibration of the cameras was performed.

7.2 Background Segmentation

The background segmentation is the main bottleneck of the system. The video stream can be segmented at the maximum 11-12 fps. The segmentation algorithm is generally conservative – it produces more foreground pixels than necessary. There are two main problems:

(1) Small holes inside the object are filled even if they belong to the background;

(2) Very dark shadows are often classified as a foreground because there is often not enough texture information at these regions to distinguish background from foreground.

The quality of the output images heavily depends on the segmentation. In a controlled environment standard blue-screening techniques could be used that are substantially faster and they might yield better results.

7.3 Network Performance/Image CODEC

The Intel JPEG Library is used to compress/decompress input textures. The images can be compressed/decompressed at a rate of more than 100fps. The highest image quality is set for the compression. The lossy JPEG compression cannot be used to encode the foreground/background mask. The foreground/background classification of each pixel without any compression is encoded using 1 bit per pixel. Then the mask is compressed using run-length encoding. The compression/decompression time required for this task is insignificant. The average sizes of the JPEG compressed image and run-length encoded mask are 90 kB and 1.7kB , respectively. Since this data is sent from all four clients directly to the server at a rate of 10-11 frames per second, it requires at least 33Mbit bandwidth. The 100Mbit LAN network is sufficient for this task.

7.4 Output Images

In this section I discuss the quality of the resulting IBVH images. First, in Color Plate A.3 the output depth images from a few different views are shown. The images are computed at 400x400 resolution. The colors represent distance to the camera (white – close, black – far). The external contours of the computed model look smooth and do not exhibit quantization artifacts characteristic to the volumetric approaches. However, in case of the coarse-to-fine sampling (Color Plate A.4) it is visible that the internal depth discontinuities are blurred since the front depth values are interpolated.

The computed geometric model is crude; however, when the textures from the input images are mapped onto the model the apparent quality improves. In the next picture set (Color Plate A.5), the same IBVH with textures are shown. The green colors

denote the regions not visible in any of the input images. Another picture set (Color Plate A.6) shows which reference image texture corresponds to which area of the IBVH – different colors denote the different reference images. Observe that when the IBVH matches the geometry of the object well the seams between textures are barely noticeable. When the geometry is far off then some ghosting artifacts can be seen. It is important to note that the apparent quality of the textured IBVH model improves even more when its video sequence is viewed. The sample video sequences can be downloaded from http://graphics.lcs.mit.edu/~wojciech/vh/results.html.

7.5 Server Performance

The tables 7.1 and 7.2 show that the time to sample and shade the IBVH scales linearly with the number of pixels in the image. The input and output images are set to the same resolution. Observe that the running time to compute an image of IBVH is highly dependent on the number of rays that intersect the visual hull (this is because the intersection of a ray with a silhouette is not computed if the intersection with the previous silhouettes yields an empty interval set). In the shown example the IBVH occupies roughly ¼ of the output image. The running times both for the standard sampling algorithm and coarse-to-fine sampling algorithm are shown. The computation is divided into the following stages: contour computation, bin creation, ray intersection, visibility, shading, and display. Clearly ray intersection and visibility operations are the most expensive. They take roughly the same time. In the coarse-to-fine sampling one traverses every 4th pixel in a scanline and also every 4th scanline. Visibility is also computed at a lower resolution.

	Number of Pixels	Contour Computation [msec]	Bin Creation [msec]	Ray Intersection [msec]	Visibility [msec]	Shading [msec]	Display [msec]	Total [msec]
100x100	10000	2.0	6.0	54.0	30.0	6.0	12.0	110.0
150x150	22500	4.0	9.0	118.0	72.0	14.0	17.0	234.0
200x200	40000	7.0	12.0	203.0	132.0	25.0	25.0	404.0
250x250	62500	10.0	16.0	321.0	227.0	39.0	36.0	649.0
300x300	90000	14.0	22.0	460.0	296.0	53.0	48.0	893.0
350x350	122500	18.0	28.0	564.0	423.0	75.0	62.0	1170.0
400x400	160000	24.0	31.0	788.0	562.0	90.0	79.0	1574.0
450x450	202500	28.0	38.0	1096.0	700.0	118.0	97.0	2077.0

Table 7.1 Running time of different parts of the system.

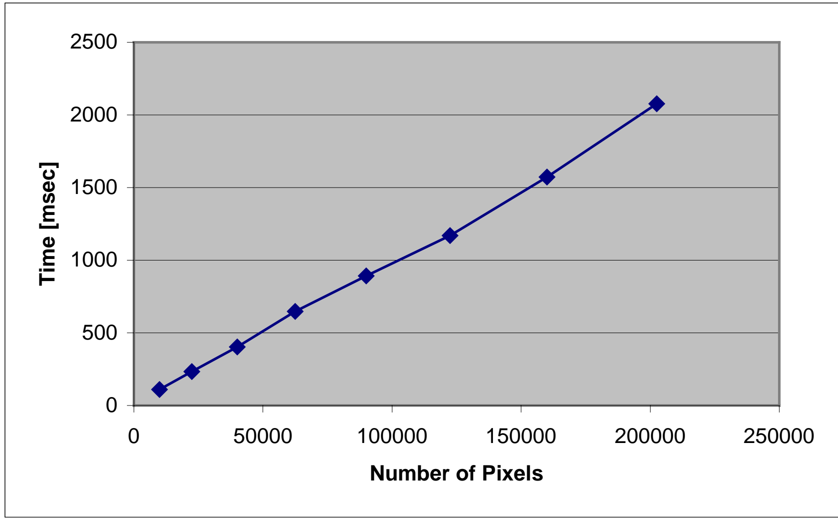

Figure 7.1 Number of pixels vs. Time.

	Number of Pixels	Contour Computation [msec]	Bin Creation [msec]	Ray Intersection [msec]	Visibility [msec]	Shading [msec]	Display [msec]	Total [msec]
100x100	10000	2.0	6.0	4.0	2.0	6.0	12.0	32.0
150x150	22500	4.0	9.0	10.0	6.0	14.0	17.0	60.0
200x200	40000	7.0	12.0	17.0	11.0	25.0	25.0	97.0
250x250	62500	10.0	16.0	30.0	17.0	39.0	36.0	148.0
300x300	90000	14.0	22.0	39.0	27.0	53.0	48.0	203.0
350x350	122500	18.0	28.0	61.0	35.0	75.0	62.0	279.0
400x400	160000	24.0	31.0	75.0	51.0	90.0	79.0	350.0
450x450	202500	28.0	38.0	98.0	64.0	118.0	97.0	443.0

Table 7.2 Running time of different parts of the system when coarse-to-fine sampling and lower resolution visibility is used.

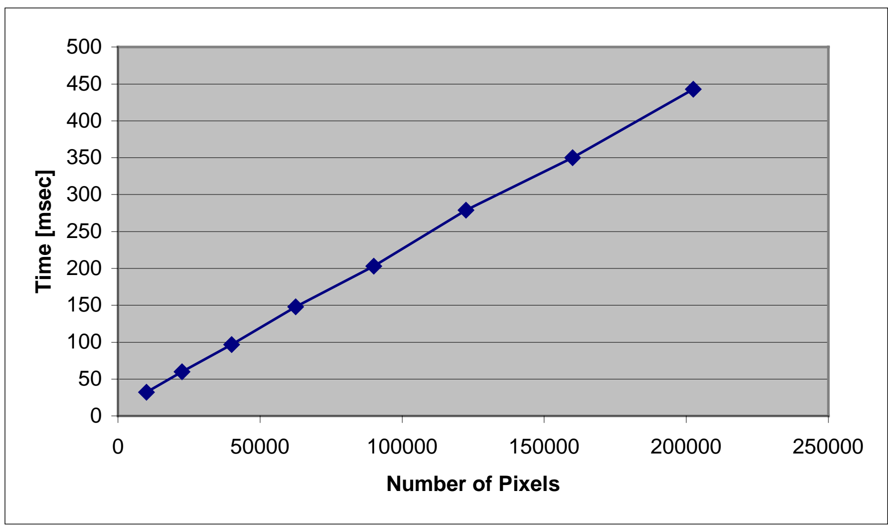

Figure 7.2 Number of pixels vs. Time (coarse-to-fine sampling).

Chapter 8

Future Work & Conclusions

There are many different avenues for the future work. First, I believe that the overall quality of the output video stream can be greatly improved. The 640x480 input video streams could be used. The frame rate could also be increased to 30fps. Moreover, the number of video streams could be increased. In order to improve the quality, a more controlled environment and blue-screening for the background/foreground segmentation could be used. All this can be achieved using newer and faster equipment and by employing further optimizations to my implementation.

The visual hull, in general, is a good conservative approximation to the true shape of the object. There are even points on the visual hull that also lie on the actual surface of the object. Another property of the visual hull is that it always contains the actual object. Therefore, the visual hull can be a good starting point for other vision algorithms (for example structure from shading, multi-baseline stereo) that could further refine the geometry. In particular, in case of the multi-baseline stereo method the search for the correspondences can be significantly narrowed down.

I believe that this work provides the infrastructure for a variety of applications. A few of them (virtual sport camera, virtual meeting) are described in the introduction chapter. Another interesting idea is the notion of the virtual set where the film director uses the system to capture the whole scene. Later the director can select the shots from any desired camera view. The system would also allow to produce some special effects for example freeze frame – where different shots of the scene of the same time instance can be obtained. Moreover, a pair of systems like this can be used for 3D teleconferencing. This application would enhance the teleconferencing experience since the users could view each other from arbitrary camera positions, and superimpose themselves onto some other background scenes.

The system described in this thesis can be also beneficial for solving various computer vision problems. The stream of IBVHs could be used as an input to the feature/object tracking algorithms. The tracking is simplified if it is performed in the 3D domain rather than in the 2D images. The system can be also used for real-time object identification.

8.1 Contours as Basic Geometric Primitives

In this section I describe how the concepts and algorithms described in this thesis can be further generalized to obtain a useful representation of the scenes. I describe how the sampling algorithm can be extended so that it can sample arbitrary geometry and not just the visual hull. In order to make the algorithm work two following modifications need to be made:

(1) Both internal and external contours have to be used. First, the algorithm is used to compute intersection operation for the shape defined by the external contours. Then, the algorithm is used to compute the intersection of all the generalized cones defined by the internal contours. In order to obtain the sampling of the true geometry the intervals corresponding to the concavities (the intersection of all internal contours) are subtracted from the visual hull samples (the intersection of all external silhouettes). For each desired ray this last step is a simple difference operation in 1D that can be implemented very efficiently.

(2) A much tighter approximation to the volume of the scene can be obtained if the scene is subdivided or segmented into parts. Then for each part intersection of its extruded silhouettes is computed separately. In the last step, for each ray the union of the samples of all parts is computed. This additional step is a union operation in 1D that can be implemented efficiently.

The pseudocode for the algorithm follows:

```
SampleScene(intervalImage& d, ExternalParts E, InternalParts I)
(1)   intervalImage ExternalImage
(2)   for each pixel p in ExternalImage do
(3)      p.intervals = EMPTY
(4)   for each external scene part e in E
(5)      refImgList R = getPartSilhouettes(e)
(6)      intervalImage exPartImage
(7)      exPartImage = IBVHisect (R)
```

```
(8)       ExternalImage = UNION (exPartimage, ExternalImage)
(9)    intervalImage InternalImage
(10)   for each pixel p in InternalImage do
(11)      p.intervals = EMPTY
(12)   for each internal scene part i in I
(13)      refImgList R = getPartSilhouettes(i)
(14)      intervalImage inPartImage
(15)      inPartImage = IBVHisect (R)
(16)      internalImage = UNION (inPartImage, InternalImage)
(17)   d = DIFF (ExternalImage, InternalImage)
```

Using the above algorithm the whole scene can be represented as a set of images where each image stores both internal and external contours for the parts of the scene. Representing the whole scene as a set of contours has many potential advantages. The main reason is that the primitives are 2D entities. Many useful operation are much more efficient in 2D. As an example two of them can be listed: (1) an efficient storage – each contour can be stored in a compact manner especially when it is stored as a set of edges; therefore, this representation might be useful for geometry compression. (2) the level of detail control – the number of edges in the silhouettes can be controlled (there are standard techniques for reducing the number of edges in the contour). If the number of edges in the reference images is reduced then it is possible to transition seamlessly to models with lower geometric complexity.

8.2 Conclusions

In this thesis I have presented a set of efficient algorithms for real-time acquisition, 3D reconstruction, and rendering of dynamic scenes. I also presented a real-time system that demonstrates all these ideas. The algorithms and the system have the following advantages: (1) Speed – the algorithms have one of the fastest running times of all current algorithms. The time is roughly constant per desired pixel per input camera. The algorithms scale very well both with size of the images and with the number of input cameras. (2) Robustness – the system works without problems for majority of different scenes since it relies mainly on the quality of foreground/background segmentation algorithm. (3) Minimal quantization artifacts – the algorithms produce the exact sampling (up to a floating point) of the scene visual hull. (4) Low cost – no active devices (for example 3D range scanners) are used but only a few digital video cameras. (5) Quality – although the visual hull gives only an approximate geometry, the apparent quality is good

when it is shaded with the view-dependent texture from the input images. I believe that these algorithms and the system are a framework for many future algorithms and applications. I believe they will be used during the next years.

References

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

[2] Buehler, Christopher, Wojciech Matusik, Steven Gortler, and Leonard McMillan, "Creating and Rendering Image-based Visual Hulls",
MIT Laboratory for Computer Science Technical Report MIT-LCS-TR-780, June 14, 1999.

[3] Chen, S. E. “Quicktime VR – An Image-Based Approach to Virtual Environment Navigation.” *SIGGRAPH 9*5, 29-38.

[4] Curless, B., and M. Levoy. “A Volumetric Method for Building Complex Models from Range Images.” *SIGGRAPH 9*6, 303-312.

[5] Debevec, P., C. Taylor, and J. Malik, “Modeling and Rendering Architecture from Photographs.” *SIGGRAPH 9*6, 11-20.

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

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

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

[9] Goldfeather, J., J. Hultquist, and H. Fuchs. “Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System.” *SIGGRAPH 8*6, 107-116.

[10] Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen. “The Lumigraph.” *SIGGRAPH 9*6, 43-54.

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

[12] Kimura, M., H. Saito, and T. Kanade. “3D Voxel Construction based on Epipolar Geometry.” ICIP99, 1999.

[13] Koenderink, J., J. *Solid Shape*. The MIT Press, 1990

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

[15] Levoy, M. and P. Hanrahan. "Light Field Rendering." *SIGGRAPH 9*6, 31-42.

[16] Lorensen, W.E., and H. E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." *SIGGRAPH 8*7, 163-169.

[17] McMillan, L., and G. Bishop. "Plenoptic Modeling: An Image-Based Rendering System." *SIGGRAPH 9*5, 39-46.

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

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

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

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

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

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

[24] Rappoport, A., and S. Spitz. "Interactive Boolean Operations for Conceptual Design of 3D solids." *SIGGRAPH 9*7, 269-278.

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

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

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

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

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

Appendix A

Color Plates

Figure A.1 Results of the stages of the background subtraction algorithm. An input image and four stages of the procedure.

Figure A.2 Sample images from four video streams used to compute IBVHs.

Figure A.3 Sample output depth images from different viewpoints.

Figure A.4 Sample output depth images computed using coarse-to-fine sampling.

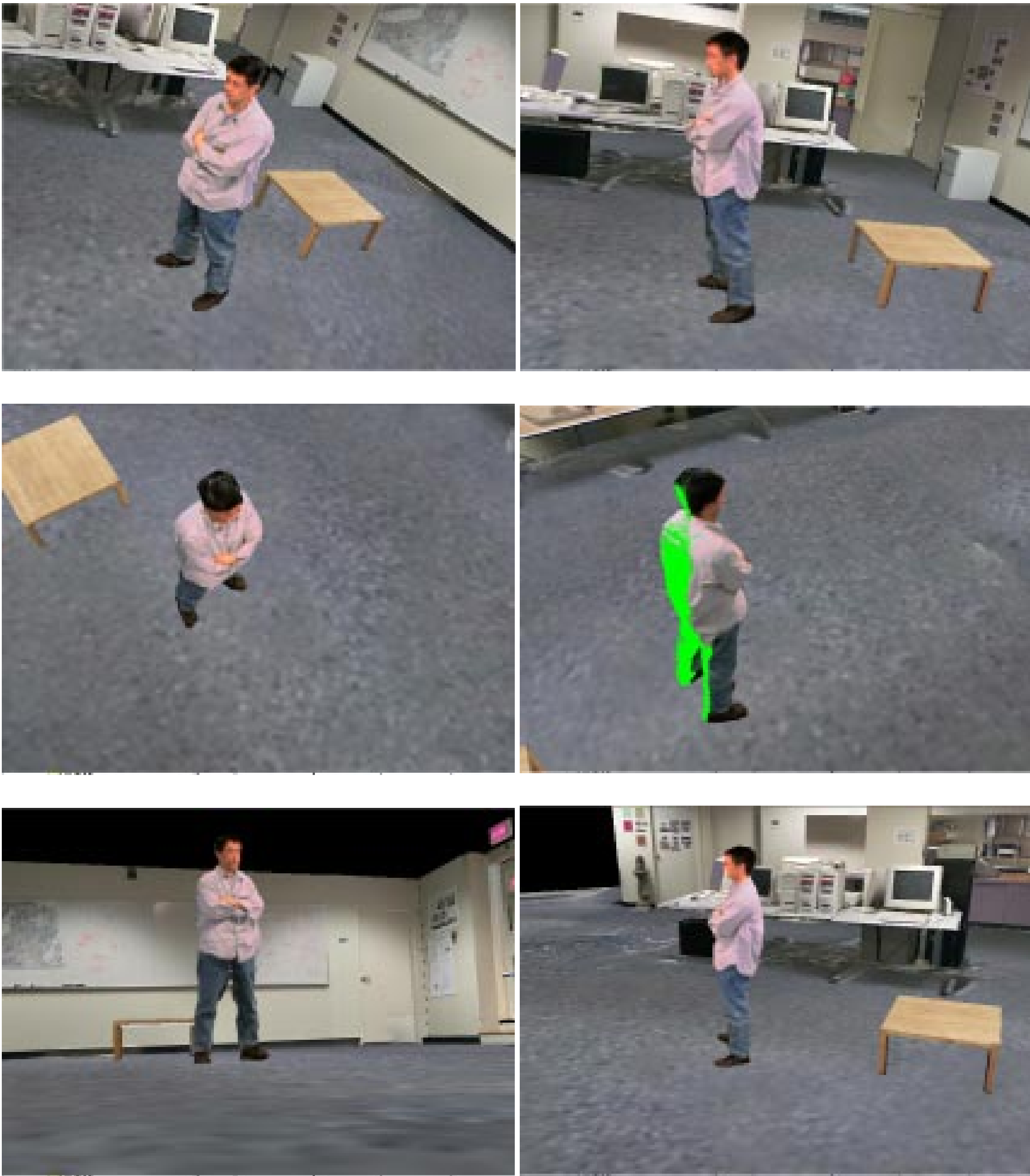

Figure A.5 Texture-mapped IBVH seen from different viewpoints.

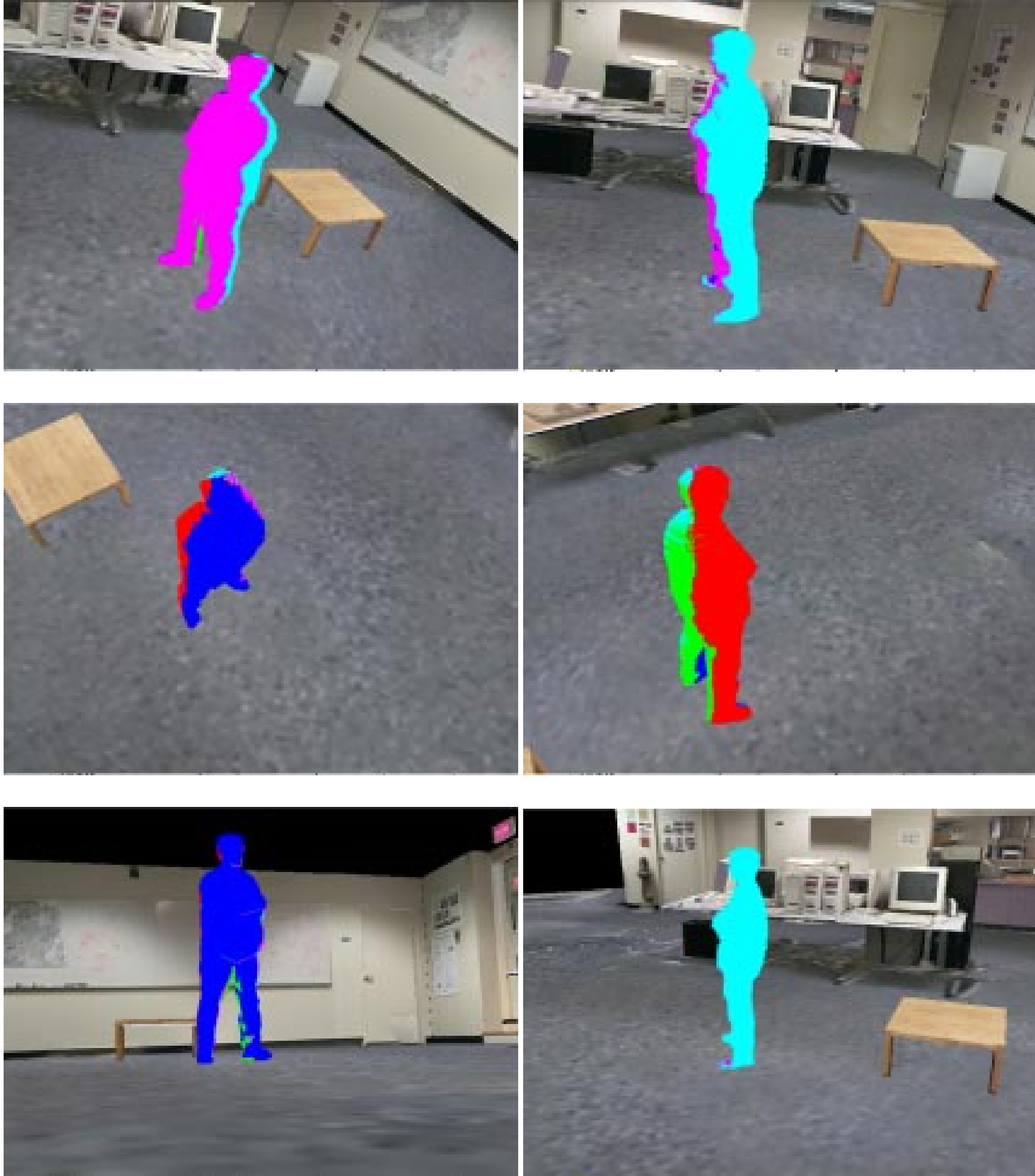

Figure A.6 Color-coded images of the IBVHs – different colors represent textures from different cameras.