POLYHEDRAL HULL ONLINE COMPOSITING SYSTEM: RECONSTRUCTION AND SHADOWING

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Henry Hobday Letteron

August 2004

ABSTRACT

A fundamental limitation of traditional two-dimensional compositing is that it lacks the spatial information necessary for realistically merging live video with simulated environments. We introduce a novel three-dimensional compositing system that leverages multiple camera viewpoints to generate a geometric model of the foreground object. The foreground object can then be merged with a virtual background environment in three-space. Using the three-dimensional definitions of the foreground and background geometries, we correctly handle occlusions and generate physically-based global illumination effects, including shadows and interreflections. Knowledge of the scene structure also removes the fixed camera constraint of 2D systems, and we allow the viewer to specify an arbitrary camera location.

To demonstrate the feasibility of interactive 3D compositing, we have built a prototype system. Our system consists of four video cameras, four client computers to perform the image processing operations, and a server that executes the reconstruction algorithm, computes the global illumination effects, and generates the final rendered image. Both the reconstruction of the foreground geometry and the scene compositing occur in real-time, allowing our hardware and software system to merge live interactive video content with virtual worlds.

This thesis focuses on the geometric reconstruction and shadowing algorithms used in our system. We compute the intersection of the silhouette cones from multiple cameras to calculate a polyhedral hull of the foreground geometry. We implement a shadow generation technique based on the penumbra map algorithm, and by leveraging the computational power of the GPU, we simulate soft shadows in hardware. Our system is capable of reconstructing geometry and casting believable shadows onto the surrounding environment at interactive frame rates.

Biographical Sketch

The author was born in upstate New York on May 7th, 1980. In the spring of 2002, he earned his Bachelor of Science degree from Rensselaer Polytechnic Institute in Computer Science and Computer Systems Engineering. In the fall, he enrolled in Cornell University's Program of Computer Graphics. Over the next two years, he consumed approximately 109,572 cubic feet of oxygen while working on his Masters of Science degree.

To my family and friends, past, present and future.

Acknowledgments

I would like to start by thanking my adviser, Donald Greenberg, for his confidence and support and for his never-relenting enthusiasm for computer graphics and its applications. He has served as a true inspiration during my stay at Cornell, and is a testament to all the great things that can be accomplished when one follows their dreams.

I would also like to extend my thanks to Kavita Bala for her ongoing help over the past two years, and to Adam Kravetz for all the man hours contributed. I need to thank both of my officemates, Will Stokes and Vikash Goel, for making 590 Rhodes Hall a lively and interesting place in which to reside, and for their constant willingness to share ideas and offer assistance. Similarly, I want to thank Jacky Bibliowicz for all the math lessons he provided free of charge, and for our late-night conversations on lab politics. I owe thanks to Mike Donikian for being a valiant competitor in hallway baseball, and to Sebastian Fernandez for his patience in explaining programming concepts to me (often multiple times) in both C++ and Java.

If not for Hurf Sheldon and Martin Berggren, our hardware system would never have moved past the design stage. They were always ready to lend a hand in order to ensure things ran smoothly, and their assistance was invaluable. For their kindness and for their help in a thousand different ways, I thank Peggy Andersen and Linda Stephenson. They made my stay at the Program of Computer Graphics flow smoothly, and it was truly appreciated.

In addition, I want to thank Bruce Walter for answering any and all RTGI related questions that I had, Steve Westin for allowing us to borrow the sturdiest checkerboard I've ever run across (and probably ever will), Ryan Ismert for our discussions on camera geometry, and Jeremy Selan for sharing his chromakeying and video expertise. For all of our "photon" and "deep space" discussions, I thank Andrew Butts, and for their interest in graphics and video games, I thank both Alex Liberman and Eugene Lee. Randy Fernando deserves thanks for taking it upon himself to help me improve my ping-pong grip and swing, as does John Mollis for being a truly laid-back guy who was always easy to talk to. Finally, I would like to thank Jeremiah Fairbank and Maureen Dugan for allowing me to partake (as an ignorant observer) in their perceptual experiments in color science.

Table of Contents

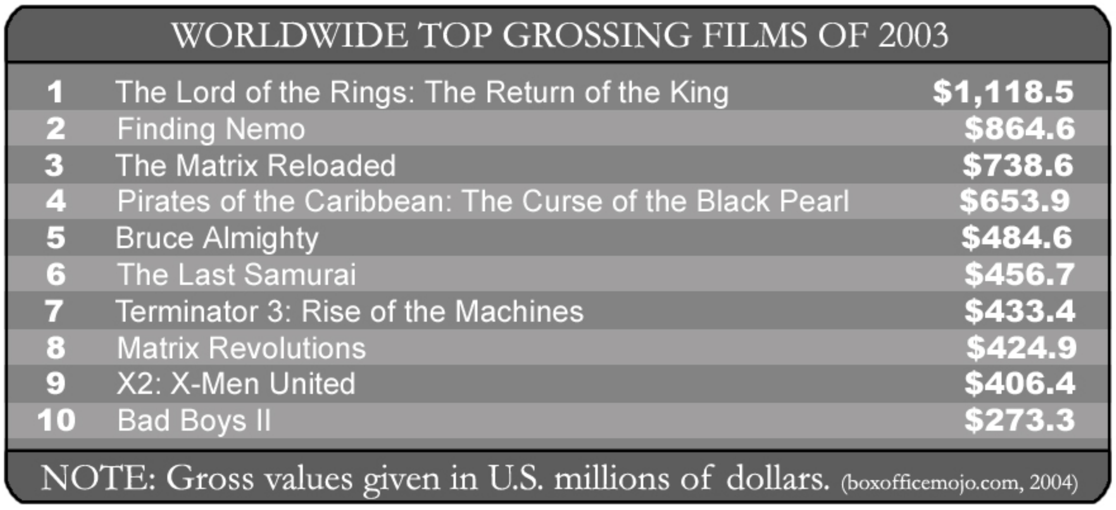
List of Figures

Chapter 1

Introduction

Over the past several years, the merging of live video footage with synthetic imagery has grown in popularity. Primarily driven by the entertainment industry, and more specifically the field of visual effects, it is becoming increasingly common to substitute some or all of a complicated movie sequence with computer-rendered alternatives. Aided by recent advancements in graphics hardware, lighting and material models, and development tools, it is now possible to create realistic looking images of items and places that do not exist outside the digital realm. In many instances, using a computer replaces the tedious process of physically building sets and costumes for filming exotic environments. This affords artists a broader range of possibilities, as they are no longer constrained by the pragmatic limitations of physical construction.

The introduction of visual effects has also caused a dramatic shift in the financial aspect of the entertainment industry. Evidence of this is shown in Figure 1.1, whereby we see that nine of the top-ten grossing films of 2003 are "effects" films that make use of compositing live action footage with synthetic imagery. The tenth movie, Finding Nemo, was entirely computer generated, and has no live video.



WORLDWIDE TOP GROSSING FILMS OF 2003

1	The Lord of the Rings: The Return of the King	$1,118.5
2	Finding Nemo	$864.6
3	The Matrix Reloaded	$738.6
4	Pirates of the Caribbean: The Curse of the Black Pearl	$653.9
5	Bruce Almighty	$484.6
6	The Last Samurai	$456.7
7	Terminator 3: Rise of the Machines	$433.4
8	Matrix Revolutions	$424.9
9	X2: X-Men United	$406.4
10	Bad Boys II	$273.3

NOTE: Gross values given in U.S. millions of dollars. (boxofficemojo.com, 2004)

Figure 1.1: *Nine of the top-ten grossing films in 2003 [Box04] combine live action footage with synthetic imagery. The remaining film, Finding Nemo, consists entirely of computer animation, and has no live video.*

Given current compositing techniques, it is not a simple task to incorporate high resolution film or video footage and synthetic digital content in the same scene. For example, say one wanted to create an image of a person standing on an extraterrestrial surface. Breaking the task into its components, it is trivial to capture an image of a person standing, and it is equally trivial (assuming a sufficiently talented modeler) to create a representation of the simulated terrain in any number of commercially available software packages. The challenge arises in finding a way to combine the two images such that the final result is believable. A large body of work has been published in the areas of image matting and compositing, however, there are still significant limitations that need to be addressed.

A serious drawback of traditional compositing techniques is that after matting out the foreground, one is left with only a two dimensional representation of the desired subject. This makes it difficult to create a convincing composite, as crucial spatial information is inherently missing. Without having some understanding as to the shape of the object being composited, it is impossible to recreate the lighting effects and physical interactions that would naturally arise if the foreground and background were in proximity to each other. For example, casting shadows from an arbitrary light source, or determining if the foreground object has "bumped" into a part in the background model, remains a daunting task without having a 3D definition of the inserted object. In Figure 1.2 we see an example of a traditional compositing approach. After the subject has been extracted from the background, the 2D representation is inserted into a background image. Scene interactions, including shadows and other GI effects, must either be added manually by an artist or generated by duplicating and skewing the subject's 2D silhouette. This approach only works if the light has a similar view of the object as the original camera, making the technique limited in its applications.

Another deterrent to the use of existing techniques is the time it takes to perform the rendering and compositing processes. In the motion picture industry, it is not uncommon for computer-generated frames to takes hours to render. For example, in the movie Shrek 2 (2004), each frame in the crowd scenes took between 30 and 40 hours to render. In total, it took approximately ten million computer hours to render the one-hour and 45-minute film. The work was performed on a render farm of 3,000 Pentium 4 processors [SFG]. When working with film sequences that are composites of live action recordings and computer imagery, each frame is often the result of a dozen or more sub-layers. An artist traditionally performs the compositing by hand, an off-line operation that can take hours to complete. This is a painstaking process, during which the artist attempts to recreate the lighting effects that would have existed in the synthetic environment. The task

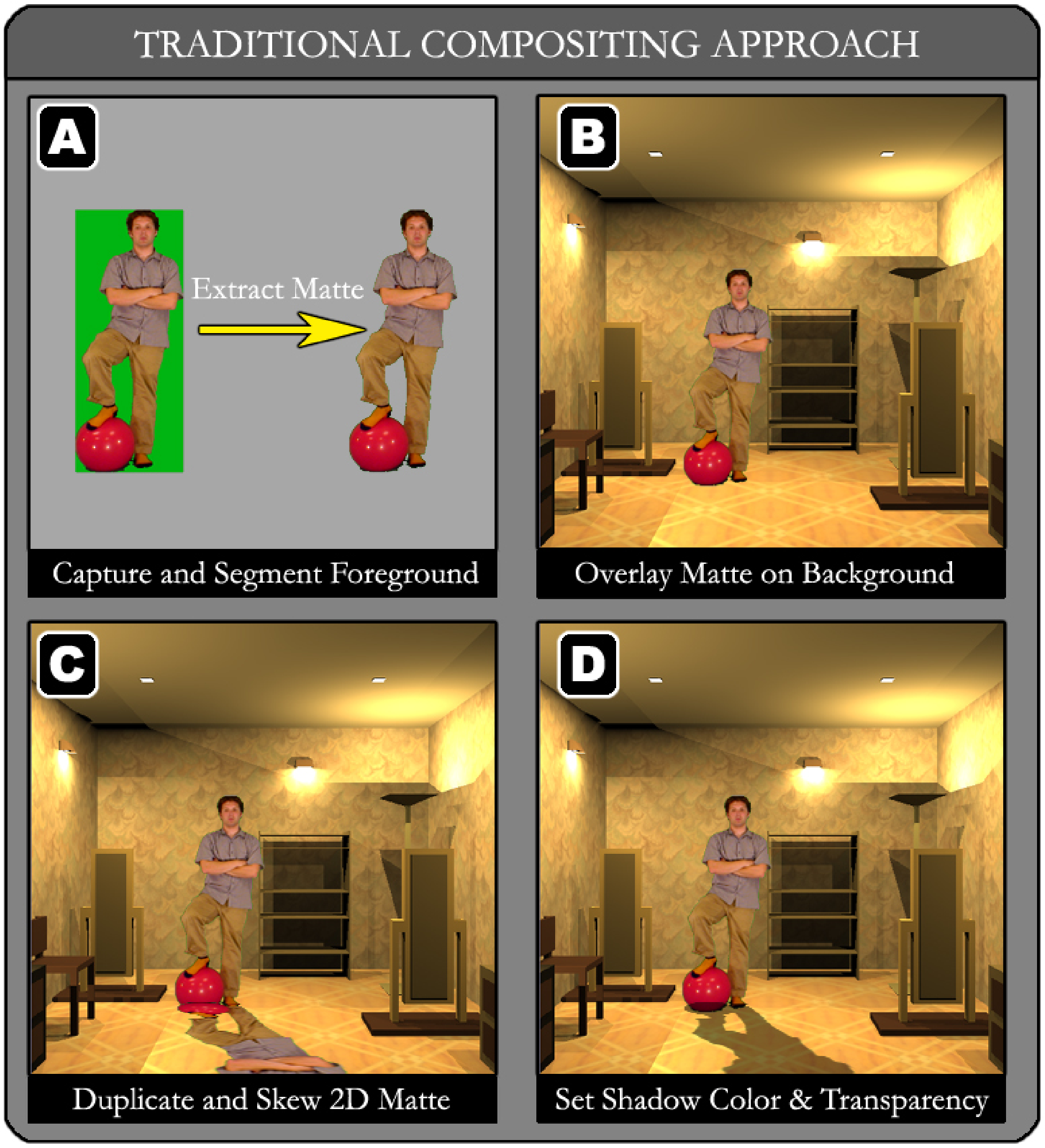

TRADITIONAL COMPOSITING APPROACH
A
Extract Matte
Capture and Segment Foreground
B
Overlay Matte on Background
C
Duplicate and Skew 2D Matte
D
Set Shadow Color & Transparency

Figure 1.2: *When only a single camera is used, the matte operation results in a 2D "cut-out" of the foreground (A). The matte is then composited with the background image (B). The shadow's shape can be approximated by duplicating and skewing the foreground matte (C). In (D), the shadow's color and opacity have been adjusted. This technique does not handle arbitrary light source locations or easily casting shadows onto the foreground, as the occluder and receiver geometry is unknown.*

Figure 1.3: *Shown above are several images from PDI Dreamworks' Shrek 2 (2004). Although most of the frames in Shrek 2 took less than 12 hours to render, the crowd scenes took between 30 and 40 hours per frame [SFG].*

is made particularly difficult because the only information available to the artist is 2D imagery; crucial spatial information concerning the shape of the object is missing. Furthermore, the user receives no immediate feedback about the quality of the work until hours or days after the filming has concluded.

We are attempting to build a system that can composite live action footage with synthetic imagery in an interactive environment. Unlike the movie industry, real-time compositing is important for live broadcasts such as the news or sporting events, which often use virtual content to augment recorded media. The system we envision could potentially serve as a direct substitute for current interactive compositing systems, providing additional information in the form of 3D foreground geometry. On the other end of the spectrum, the system could be used to add entirely new functionality, allowing the viewer to move the camera around the scene and observe the broadcast from an arbitrary viewpoint. Given the current trend in advancing television and broadcast technology, it may be possible in the near future to transmit scene geometry and thus permit view mobility. Knowing the foreground and background scene geometry allows the system to simulate global illumination (GI) effects such as shadows and inter-reflections. These GI effects lend to the believability of the composite images and result in a more immersive experience for the viewer.

In this thesis, we present a novel approach to the problem of compositing live video footage with synthetic imagery. Our method, which builds on recent research in the field of geometric reconstruction, employs the use of several video cameras to capture the shape of a dynamic foreground object. By having a three dimensional representation of the object, we are able to produce more realistic composites than is possible using prior two-dimensional matting techniques. We present a unique approach for casting shadows between the foreground and background environments, and we discuss how this work can be further extended to produce more accurate simulations. This thesis will specifically focus on the aspects of geometric reconstruction and shadow generation. For details concerning advanced texturing methods and reflections, refer to [Kra04]. The foreground reconstruction and compositing processes operate interactively, and we have been able to achieve several frames per second with our current system configuration. Figure 1.4 shows an overview of the basic steps used in our compositing system.

Perhaps even more so than film or television, our system is well suited for applications in virtual reality and interactive gaming. Within these contexts, the

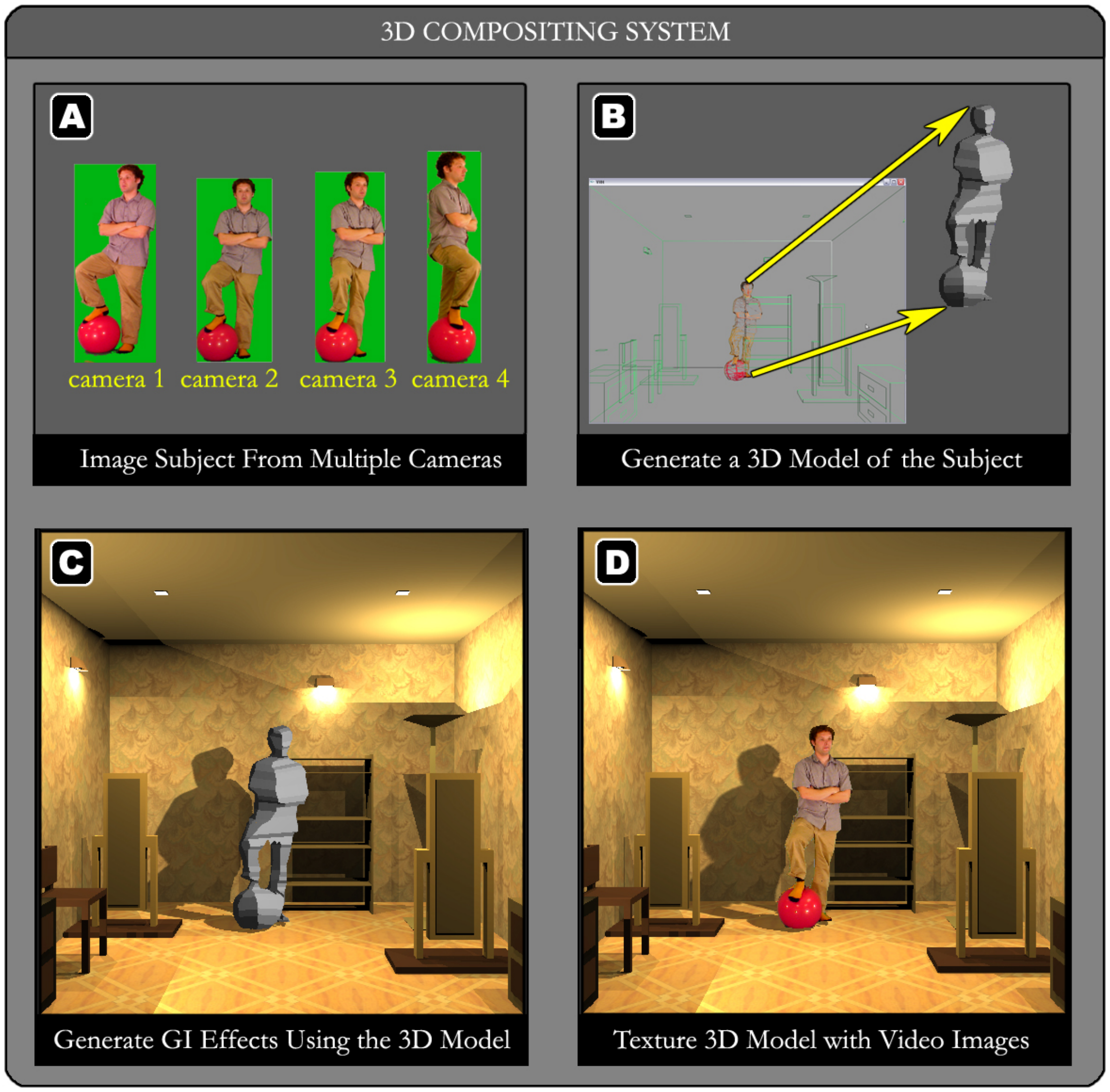

3D COMPOSITING SYSTEM
A
camera 1
camera 2
camera 3
camera 4
Image Subject From Multiple Cameras
B
Generate a 3D Model of the Subject
C
Generate GI Effects Using the 3D Model
D
Texture 3D Model with Video Images

Figure 1.4: *The figure above illustrates our compositing approach. We use multiple views (A) of the subject to reconstruct a three dimensional approximation of its shape (B). We use this model to cast shadows (C) and simulate other GI effects such as inter-reflections [Kra04]. Having a model of the object allows for the creation of more physically accurate composites (D).*

users are typically permitted free range to move the camera anywhere they desire. This type of immersive environment plays to our system's strengths, because our 3D representation of the foreground and background geometries allows for complete viewing freedom. Unlike traditional compositing systems, we do not require that the virtual camera's pose match that of any of the original cameras filming the scene. Instead, a user is able to select an arbitrary viewing location, and our system will render the generated model and the background from the specified position and orientation.

Chapter Two presents a literature review of previous geometric reconstruction and shadow algorithm research. Chapter Three covers the basic implementation details of our system's hardware and software configuration. The polyhedral visual hull algorithm [MBM01], used for the creation of the foreground models, is addressed in Chapter Four. Chapter Five introduces our novel technique for shadow generation, and Chapter Six outlines the results of our system's capabilities. Finally, in Chapter Seven, we conclude with a summary of our contributions, and suggestions for future work in the area of compositing live video with synthetic imagery.

Chapter 2

Previous Work

There are several distinct bodies of research which have contributed to our work in the area of merging captured geometry with synthetic environments. These fields include geometric reconstruction, shadowing algorithms, and scene compositing techniques. In the remainder of this chapter, the first two areas will be reviewed in depth, as they were the primary focus of our research and have directly impacted the work that was done.

2.1 Geometric Reconstruction

In order to create a convincing composite of several disparate entities, it is essential that you reproduce the light interactions that would exist if they were in natural proximity to each other. Shadowing phenomena are probably the single most important visual cue that allow a viewer to gauge the relative positioning of objects within a scene as well as determine where the light sources are located. In order to accurately simulate these effects within a virtual environment, you need to know something about the shape of the objects with which you are dealing.

How to reproduce the geometry of an arbitrary body is a question that has received a lot of attention over the past few years. Traditionally it has been a painstaking process, whereby you take measurements of an object in the physical world and then reconstruct its geometry with the aid of modeling software. Another alternative is to use complex hardware such as laser range scanners to produce extensive point sets that define the object's surface. Both of these methods are very time consuming however, and thus are unsuited for interactive applications.

Some of the earliest work in the area of generating geometry from images was done by Paul Debevec at the University of California at Berkeley. In his PhD thesis [Deb96] and a related paper [DTM96] Debevec presents a method for using several photographs of a large architectural environment to generate a photo-realistic model of the scene. His software system, referred to as Facade, requires the user to drive the modeling process by inputing some initial rough geometry in the form of cubes, prisms and surfaces of revolution. The user also identifies edges in the input photographs, and makes the correspondences between those edges and the edges of the primitive geometry. The program then has enough information to determine the pose of each of the cameras and the geometric parameters which govern the model. The problem with this approach is that it still requires a fair amount of user input, although much less than traditional modeling, and it takes on the order of minutes to solve for the reconstruction parameters (depending on the complexity of the scene).

At around the same time, work was being done by the group at Carnegie Mellon University in the area of new view synthesis. Though not using an explicit model for rendering, [SBK$^+$99] describes a hybrid approach that uses knowledge of the scene's 3D structure to help avoid occlusion errors when performing 2D view morphing. This cross between model based rendering and image based rendering consisted of creating a mesh, using multiple-baseline stereo [OK93], to determine pixel correspondences when interpolating between two existing views. Unfortunately this approach does not make any attempt to verify that the final image, as seen from the virtual viewpoint, is geometrically correct.

Another approach to geometric reconstruction is voxel occupancy. A voxel is defined as the smallest distinguishable box-shaped portion of a three-dimensional space. In voxel occupancy, the goal is to be able to determine whether these discretized spatial areas are occupied or empty. In 1997, Seitz and Dyer [SD97] presented a novel method called "voxel coloring" that formulates the problem as one of color reconstruction. They treat each voxel as having a single opaque color, and their goal is to traverse the scene in depth order, coloring the voxels in a manner that maximizes photo integrity. They present a visibility constraint that facilitates the depth traversal of the scene, thus allowing their model to explicitly handle occlusions. Through the use of many input images, their method is capable of producing a dense reconstruction. This permits the final model to be accurately viewed over a wide range of virtual viewpoints. Although they place limitations on the locations of the cameras within the scene, such that no scene point is contained within the convex hull of the camera centers, they claim that there are many environments in which the system is amenable. In order to generate a representation of sufficient quality for achieving photo realism, the scene has to be highly discretized and there has to be a large number of input images. Unfortunately, this results in long running times, often upwards of seconds or minutes.

In 2000, Snow, Viola, and Zabih [SVZ00] presented a method for representing the voxel occupancy problem with an energy minimization formulation. Their approach stands out in that it doesn't require the explicit computation of silhouettes, and it has a term for incorporating spatial smoothness. Instead of having hard silhouette constraints, they deal with the soft constraints of an energy function that allows pixels to take on their neighbors' values. Their system uses the video input from sixteen cameras which surround an image acquisition area. They attempt to rebuild the shape of the foreground object by marking those voxels which contain the subject as filled, and claiming the rest are empty. Unlike voxel coloring, a technique which assigns each voxel a color and a transparency and then prunes away empty voxels based on color mis-matches between camera views, voxel occupancy is a much easier problem to solve. Voxel occupancy treats the voxels within a scene as nodes in a graph, where each node is connected to its six neighbors. The problem can then be treated as a form of energy minimization. The goal is to partition the vertices into two disjoint groups, foreground and background, by making the minimum number of cuts through the graph. Unfortunately this method is unable to synthesize images from novel viewpoints in color, and the computational time to perform the minimum cut is on the order of seconds. These qualities do not lend themselves to interactive applications.

In 2000 and 2001, Matusik et. al. at MIT published several papers presenting algorithms for rendering visual hulls at interactive frame rates. The visual hull [Lau94] of an object is an upper bound on its volume, and thus is guaranteed to contain the object. In their approach, they surround the object with an array of cameras, and then treat the silhouettes from each of the cameras as a projected cone, starting at the camera origin and extending through the image plane to infinity. The next step is to intersect all of the cones to form an upper limit on the object's volume. This upper limit, which is the visual hull, is highly dependent on the number of cameras: the higher the camera count, the more refined the hull becomes in its approximation. Visual hulls do have their short-comings, however. Because of the fact that they are created through the intersection of extruded silhouettes, this approach performs best when dealing with convex objects. If an object has concavities which are internal to its silhouette contours, then it becomes impossible to reconstruct the exact geometry regardless of the number of viewpoints considered.

Matusik's first paper [MBR$^+$00] operated solely in image space, reducing a classic three dimensional intersection problem to two dimensions through the use of epipolar geometry to generate image correspondences. With this technique they are able to produce a view-dependent visual hull that can be rendered from novel view points. Using the video streams as textures during the shading stage, this approach is able to produce renderings with a high degree of realism, despite the limited geometric accuracy. However, since the algorithm does not explicitly compute a volumetric representation, the types of scene interactions that can be performed are limited.

In a related paper a year later [MBM01], Matusik extended the algorithm to reconstruct a polyhedral representation of the object. Like its predecessor, this algorithm also falls into the realm of deriving shapes from silhouettes, by analyzing images of an object's surface from multiple viewpoints. However, instead of only dealing with the intersection of projected rays in two dimensions, it has the ability to generate polygons when projecting the silhouette edges on to the other image planes. Using a novel intersection algorithm, they can then find the overlapping area of all such polygons that exist for a given edge. The mesh which defines the object's surface is consequently formed by taking the aggregate of all the intersection polygons. In 2002, [MBM02] presents a more efficient algorithm for computing the silhouette cone intersection which exploits the special structure of generalized cone polyhedra.

Addressing the capture and reconstruction of objects with complex surface characteristics, such as highly specular, transparent or refractive surfaces, or those with fuzzy geometry (fur, hair, etc), Matusik et. al. published a series of work [MPZ+02] [MPN+02] in 2002. Unlike his previous IBVH and PVH work, these algorithms require a high degree of pre-processing, taking several hours to capture all the necessary images, and then often require upwards of a day or more to process the data. The goal in these cases is to be able to capture as much detail as possible, so that when the model is re-rendered from an arbitrary location a high degree of realism results.

In 2002, Li, Schirmacher, and Seidel at the Max-Planck-Institut [LSS02] discovered that by combining the polyhedral visual hull algorithm with a depth from stereo approach, it is possible to determine the location of concavities within the hull. Depth from stereo is a technique that takes advantage of the fact that when all the objects in a scene are shifted by the same amount, objects which are closer to the viewer appear to have moved farther than objects which are far away. Keeping this in mind, pairs of images can be used to determine the relative depths of neighboring pixels by observing how far each pixels' location has changed from the left image to the right image. If a region of pixels appears to have moved less than its surroundings, then those pixels represent a surface which is slightly farther away, and thus implies a concavity. Although this technique has the added benefit of being able to determine concave regions, it adds extra complexity to the original algorithm, particularly in how to merge the depth changes back into the original polyhedral hull. It also requires more hardware, as now two cameras are required at each acquisition location instead of just one (a necessary component for stereo-based reconstruction).

That same year, Slabaugh [SS02] presented an algorithm for rendering novel views of an object's photo hull. A photo hull is defined as the largest 3D shape that is photo-consistent with photographs taken of a scene from multiple viewpoints. A point in space is photo-consistent if it doesn't project to the background and, when visible, its radiance in the direction of each reference view is equal to the observed color in the photograph. The algorithm, which is based on Matusik's IBVH algorithm [MBR$^+$00], operates completely in image space and makes use of color information in the form of additional constraints during the reconstruction process. This results is more refined geometry than would be possibly using only the IBVH algorithm, and it leads to better synthesized views of the captured scene. The method does have its weaknesses, however, since surfaces that are predominantly one color make color-based reconstruction difficult. Such surfaces can lead to many inconsistent pixels being registered as consistent. The photo-consistent validation process also increases the reconstruction time above and beyond that of the IBVH algorithm by roughly a factor of four.

A hardware accelerated approach to visual hull reconstruction was published by Li in 2003 [LMS03]. They use standard commodity graphics hardware to generate and render visual hulls at speeds of up to 80 frames per second. This is done by using projective texturing in conjunction with OpenGL's alpha test to determine the overlapping regions for both the face-cone and polygon-polygon intersections. One disadvantage of this approach is the fact that it combines both the visual hull generation and the final rendering into a single pass. Although this is a beneficial speed optimization, as it reduces latency in the system, it limits the use of the hull for such purposes as casting shadows. The polyhedral hull is only generated as seen by the virtual viewpoint, and thus its geometry can not be accessed.

In 2003, Franco and Boyer [FB03] revealed an algorithm for generating an exact polyhedral visual hull in real-time. Unlike the original PVH algorithm [MBM01], the exact hull is guaranteed to be both manifold and watertight. Their algorithm consists of three steps. First they back-project each vertex in the image plane contours to form world space viewing lines. The viewing lines are successively intersected with each additional silhouette, and the overlapping regions of intersection are recorded. The result is a set of intervals along the viewing lines, called “viewing edges”, which form an unconnected subnet of the final visual hull. In the second step, the missing surface points are identified. This is accomplished by intersecting the planes which generated the viewing edges in order to determine the search directions for the missing points. Each vertex is connected to exactly three edges, so in addition to the viewing edge, a left and a right edge must be generated for each viewing vertex. The search directions are then projected on to each image plane and intersected with the silhouette contours. If the projected search segment intersects the silhouette, then a “triple point” has been found, otherwise the new edge leads to an existing viewing vertex. Triple points are where three viewing cones intersect. After all the triple points have been located, the visual hull can be reconstructed. The third and final step is to walk through the graph of edges, identifying each face and converting the polygons to triangles.

2.2 Shadowing Algorithms

2.2.1 Hard Shadows

The first shadowing algorithms were developed at IBM by Appel in the mid 1960s, and since then shadows have remained a heavily studied topic within the field of computer graphics. Shadows play a crucial role in the generation of realistic images. They make clear the spatial relationships which exist between objects in a scene, and they allow for easy identification of the light sources present. The only exceptions are scenes where either the light's position matches that of the camera or the light sources are entirely diffuse.

The shadow volume algorithm was first introduced in the 1977 SIGGRAPH proceedings by Franklin Crow [Cro77]. His method makes use of the boundaries which exist between surfaces that face toward the light and those which face away from the light in order to define a shadowed region. This volume is constructed by extruding the light/dark silhouette edges to infinity in the direction away from the light source. When rasterizing an image, one tracks the number of shadow polygons crossed in the process of traversing the scene from the camera center to the first object at any given pixel. If an odd number of shadow volume polygons are crossed to reach the first object, then the pixel is in shadow. Otherwise, the pixel is visible to the light source. Shadow volume algorithms tend to be fill rate limited, as the shadow polygons extend to infinity and thus are quite large.

A year later, in 1978, Atherton, Weiler and Greenberg [AWG78] published work that generated shadows using a polygon clipping hidden surface removal algorithm. This particular technique consists of rendering the scene from the light's point of view, and thus determining the regions of complete visibility. The unoccluded areas of the scene, which are subject to direct lighting, are then added to the environment as surface details on their original source polygons.

Also in 1978, Williams [Wil78] proposed a new shadow algorithm, called shadow mapping, that involves rendering the scene twice, from the point of view of the light, and then from the point of view of the observer. After rendering the scene from the light's point of view, a shadow map is created, each value of which represents the distance from the light to the nearest object along a given path. When rendering with respect to the observer, each of the world coordinate points is transformed into the light's local space. If the projected point is closer to the light than the value stored in the shadow map, then that point is unoccluded and should be rendered as lit. However, if the transformed location is farther away from the light than the distance stored in the shadow map, then there exists a surface between the light and the point in question and that point should be rendered in shadow. Shadow maps tend to have aliasing problems when large numbers of pixels in the eye view project into the same pixel location in the shadow map. Shadow mapping is also subject to numerical imprecision in the depth buffer, which can lead to noticeable artifacts near light/dark boundaries. Biasing the depth values can often take care of this problem, as can rendering the back facing polygons when creating a shadow map (false positives on the back face of an object are not a problem, as the light's contribution to the surface will be zero).

Another shadowing technique involves projecting the occluding geometry onto a flat ground plane. This projected version defines the shadow region on the surface of which it was cast. Blinn [Bli88] proposed such an algorithm in 1988. This technique can be extended such that one renders the shadow to a texture. This texture can later be projected on to an arbitrary receiving surface [SKvW$^+$92].

Finally there is the brute force ray tracing approach to shadow generation. A typical ray tracer casts rays from the viewpoint through each of the pixels in the image plane to determine the closest object. Starting at these intersection points, rays are then shot to each of the lights in the scene. Using the most primitive form of visibility testing, if the ray intersects any other objects before it reaches the light, then the point is considered to be completely occluded, aka in shadow. If no other object is hit before the light, then the point is visible and deemed lit.

In 2000, Michael McCool [McC00] from the University of Waterloo published a paper which reviewed the pros and cons of each of the four general types of shadowing algorithms (ray casting, shadow volumes, shadow mapping, and projection). He also proposed a novel hybrid technique that combined aspects of both the shadow mapping and the shadow volume algorithms. His method involved first rendering the scene with respect to the light, and then running a contour finding algorithm on the resulting shadow map. The contours which are found in the depth image represent large depth disparities within the scene, and are consequently treated as silhouette edges between light and dark facing surfaces. Using the contours as silhouette edges, shadow polygons which bound the shadow volume can then be generated. This technique has benefits in that the models do not need to be closed (2-manifold), as the boundaries between light and dark are found from the shadow map. In addition, it enables the shadow volume algorithm to be performed in hardware with only a single bit stencil buffer. Because there can be no overlapping of the shadow polygons, the stencil buffer can be toggled as you pass from regions of light to dark and vice versa.

A technique for eliminating the aliasing problems associated with shadow mapping was published in 2001 [FFBG01] by Randima Fernando, Sebastian Fernandez, Kavita Bala and Donald Greenberg from Cornell University. This algorithm operates by using mip-mapping in hardware to determine the projected area of an eye-view pixel in the shadow map (light view). When there is a large disparity between the two views, a situation which has traditionally resulted in aliasing, the program progressively subdivides and refines the shadow map in that particular region. The shadow map is stored in a hierarchical grid structure than can be updated as the user's viewpoint changes, and the user can specify the amount of memory allocated to the shadow map, so that it does not continue to grow unchecked.

In 2002, Stamminger and Drettakis [SD02] presented a "perspective shadow mapping" technique that also serves to reduce the aliasing problem associated with standard shadow maps. Their approach generates the shadow map in normalized device coordinates, and is especially beneficial for scenes with wide depth ranges, where nearby shadows require much higher resolution than distant shadows. Using this perspective method, the shadow map is view dependent, and provides variable resolution so that objects close to the camera receive more detail. The shadow map projection can still be represented by a 4x4 projection matrix, and thus is amenable to graphics hardware. Due to its view dependent nature, the perspective shadow map must be recomputed each frame if the camera is allowed to move. This, however, is standard practice anyway in applications such as video games, where both the shadow occluders and receivers are permitted to move.

2.2.2 Soft Shadows

Hard shadows are primarily the result of infinitely small light sources, where the source begins to approximate a point light, or environments where the occluder is very close to the receiver. In the real world hard shadows are commonly observed at contact points, where an occluder is physically touching the receiver surface. However, most shadows that we are accustomed to seeing are soft shadows. Shadows consist of multiple regions. The umbra, or the hard shadow, is the portion of the shadow that is completely hidden from the light. The penumbra is a partially shadowed region inside which the scene is transitioning from completely lit to completely in shadow. Soft shadows add a high degree of realism to a scene, and in cases such as a dispersing contact shadow, can provide information concerning the spatial layout of the objects.

In 1987, Reeves, Salesin and Cook [RSC87] introduced a new sampling method called "percentage closer filtering" that can be used to reduce the aliasing artifacts which arise in shadow mapping. Their technique has the added benefit that it is capable of generating soft shadow edges that resemble penumbrae. Like the standard shadow map algorithm, their approach projects a pixel into the light's view, and then determines its location within the shadow map. However, instead of just performing a direct binary comparison to the stored depth value, they then compare the projected pixel depth to all the values within a specified region. This results in an array of binary values, one for each comparison made. The percentage of comparisons that returned "1", indicating shadow, becomes the pixel's shadow intensity. If half the comparisons returned shadow, then the pixel has a 0.5 shadow intensity, placing it in the penumbra. By adjusting the filter size, aka the size of the region of comparison, one can alter the characteristics of the penumbra. This approach requires both additional time and resources to compute, however, as the number of samples per region is constant, the additional cost is bounded by a constant factor as well.

Another technique that resulted in more accurate soft shadow simulation was developed in 1992 by both Lischinski et. al. [LTG92] and Heckbert [Hec92] independently. This procedure, called “discontinuity meshing”, was built on top of radiosity, a popular algorithm for modeling global illumination effects in a diffuse environment due to area light sources. They explain the relationship which exists between discontinuities in the radiance function and its derivatives and the umbra and penumbra boundaries within a scene. The algorithm maps out the radiance function across a surface in object space, using a piecewise linear interpolant to preserve the discontinuity edges. By explicitly modeling the important illumination boundaries, it is possible to generate highly accurate soft shadows while using a much courser mesh of the scene.

In a ray tracing context, soft shadows can be generated by casting multiple samples to an area light and then calculating the visibility based on the percentage of rays that were occluded. With a high sampling rate (casting many shadow rays per light source), this technique yields very accurate results. Unfortunately, shooting many rays per pixel is extremely costly in terms of the computation time required per frame. This makes interactive applications implausible unless vast amounts of computing resources are available.

One algorithm that addresses the issue of computational complexity associated with shooting many shadow rays per light was published in 1998 by Steven Parker et. al. from the University of Utah [PSS98]. Their method, which is geared toward generating perceptually acceptable shadows quickly, involves using only a single shadow ray per light to simulate the effects of a spherical area light. They accomplish this by creating a semi-opaque outer shell which surrounds each object in the scene and whose transparency varies from being completely opaque where it touches the inner object to being completely transparent at the outer edge. When shooting a shadow ray, if the original object is intersected, then the point is completely in shadow. If, however, the shadow ray hits the encompassing outer geometry, then the visibility is computed based on the tangential distance to the inner object. In this manner, smoothly varying penumbra regions between the umbra and the completely lit portions of the scene are achieved. The radius of the encompassing outer object is based on both the radius of the area light as well as the ratio of the distance from the occluding surface to the receiver surface versus the distance from the light to the receiver surface. The main drawback of this method is that the umbra region never diminishes in size regardless of the position or radius of the area light source. This is most noticeable when dealing with small geometries where, in the limit, the umbra region can disappear entirely.

When dealing with soft shadow generation in hardware, a standard approach has been to sample multiple locations on an area light source, rendering hard shadows for each one, and then averaging the results together. This method was presented in 1996 by Heckbert and Herf [HH96] [HH97] who use a texture map to represent the radiance at each point on a receiver's surface. In their algorithm they select multiple light samples, which are spread across the lights in the scene, from which to render the receiver and from which to project the occluding geometry on to the receiver's surface. The projected geometry, which is rendered in black, serves to represent the shadows cast from that particular sample location. The multiple passes are then averaged together using an accumulation buffer to generate a final radiance texture for the receiver surface. This can produce high quality results, but only with a large number of samples (often as high as 256), where each sample means an additional rendering pass. If there are too few samples taken, the result will not smoothly diminish to accurately simulate a soft shadow. Instead a visible set of discrete hard shadows will remain in the form of a disconcerting artifact. Because of the large number of rendering passes needed to produce suitable results, this technique is prohibitive for anything other than offline applications or applications where the textures can be precomputed. In the latter case, this places the restriction that both the scene geometry and the light locations are static.

Another hardware-based soft shadow approach was revealed by Agrawala et al [ARHM00] in 2000. They present two novel soft shadow algorithms that are built on top of shadow mapping, one of which is designed for interactive applications, and the other is for off-line use only. These techniques, termed “layered attenuation maps” and “coherence-based raytracing” respectively, are image-based approaches and thus are relatively independent of geometric scene complexity. The layered attenuation map algorithm renders the scene from multiple points on the surface of the light as a preprocessing step, and combines the generated depth maps by warping them to the light’s center. This produces a modified, “layered depth image” (LDI), which is indexed during the final rendering pass. Due to the fact that the light samples are correlated for each surface location, this method is susceptible to banding in the final image. Higher quality images are generated with the coherence-based raytracing method, as the sampling limitation is no longer a factor. Instead, shadow rays are traced through the multiple shadow maps, as opposed to scene geometry, in order to shade a surface point. To reduce the high cost associated with this operation, several novel acceleration structures for handling the shadow ray computations are presented. Due to the expensive nature of computing the modified LDI, which takes on the order of seconds to generate, even their interactive algorithm is not well suited for applications where the objects are constantly moving.

In 2001, Haines presented another projective soft shadowing technique [Hai01]. His algorithm produces perceptually acceptable penumbrae in a single hardware rendering pass. The end product is a texture of the shadow that can then be mapped on to any type of receiving surface; however, the further the surface is from a plane, the more unnatural the shadow appears.

One trend that is worth noting is the recent shift toward hardware-based approaches and programming for the graphics processing unit, or GPU. Inspired by the latest round of graphics cards, which add increased flexibility over the traditional fixed-function pipeline, many of the current shadowing algorithms are being designed with the GPU in mind. By performing the shadowing calculations in vertex and fragment shaders instead of on the CPU, it frees up the CPU to direct its attention to others areas of the code. This can result in dramatic performance increases. As a result, it has become common to see shadow algorithms adopted so that they map to the current level of graphics hardware available. The penumbra wedge, smoothies, and penumbra map algorithms which we will talk about next are three recent examples of methods that generate soft shadows by leveraging the GPU.

In 2002, Akenine-Moller and Assarsson [AMA02] extend the standard shadow volume algorithm so that it is capable of casting soft shadows on arbitrary receiver surfaces. Their method replaces the hard-edged shadow polygon with a primitive called the penumbra wedge. This primitive is used to depict the penumbra volume created by a silhouette edge. Within the wedge, the shadow intensity varies linearly across its span, from complete shadow on the inside edge to completely lit on the outside edge. By defining the penumbra region as being the volume inside the wedges, this algorithm implicitly models the umbra region as well. The umbra is the area contained inside the back-facing surfaces of all the wedges. A wedge is created by extending two planes through a silhouette edge, one passing through each side of the spherical light source, and then connecting those planes with triangular side panels. As the depth of a wedge (the distance between its front and back planes) is determined by the radius of the light source, along with the distance between the light center and the silhouette edge, the algorithm has the added benefit that it correctly models the fluctuating size of umbrae. This holds true even to the point that the umbra region will disappear completely should the light source be sufficiently large. The shadow wedge algorithm inherits several shortcomings from it's shadow volume predecessor, including the inability to handle non-polygonal shadow casting geometry. It also has some unique limitations associated with the robustness of wedge generation. If a silhouette edge is nearly parallel to the incoming light direction, then the generation of the wedge side planes will begin to degrade and shadow artifacts will arise. In addition, errors can arise when two objects overlap as seen by the light. In this scenario, it is highly probable that their penumbra wedges will also overlap, and the algorithm's proviso for handling this case, which involves subtracting the light from both wedges, is not always correct.

A year later, Assarsson and Akenine-Moller [AAM03] refined their approach, augmenting both the robustness and performance of their shadow wedge algorithm and adding additional features such as the use of video-textured light sources. The new algorithm involves generating a "visibility mask" to store the shadow intensity throughout the scene. The visibility mask is used to modulate the scene's specular and diffuse components, before the final ambient term is added in. In order to compute the visibility mask, it is necessary to determine the percentage of the light that can be seen at any given surface point. This is accomplished by discretizing the light source into 32x32 regions and generating a lookup table in the form of a 4D texture. Each point in the scene then indexes into the 4D texture to determine the fraction of light it receives. By maintaining a sequence of lookup textures, such that a different one is indexed each frame, it is possible to simulate dynamic lighting effects, such as the flickering of fire.

Assarsson [ADMAM03] later improves on the algorithm even more, eliminating artifacts and further optimizing the algorithm to take advantage of current graphics hardware. Though the shadow volume and wedge generation is still done on the CPU, the rendering is now all performed in pixel shaders, allowing frame rates which are commonly around 50 for relatively complex scenes.

Just as the penumbra wedges algorithm was an extension of shadow volumes, similarly Chan and Durand [CD03] proposed an extension to shadow mapping that would permit the generation of soft shadows in 2003. Their method does not focus on producing physically accurate shadows, and in fact it completely ignores the shape and orientation of the light source. Instead it focuses on the generation of perceptually “pleasing” shadows, using the distances between the light, occluder, and receiver, to estimate believable penumbrae widths. They generate a set of geometries, called “smoothies”, which surrounds the objects’ silhouette as seen by the light, and is rendered into a “smoothie buffer”. Then, when rendering the scene from an observer’s point of view, if a point should fail to fall into shadow from the shadow map, a smoothie buffer lookup is performed to determine if the point is in the penumbra. Because it has its roots in shadow mapping, the smoothies algorithm does not model the size of the umbra accurately. It also does not handle non-polygonal shadow casting geometry, as it is necessary to find the silhouette of the object as seen by the light in order to calculate the smoothie geometries. In addition, the performance of the algorithm is directly correlated to the width of the penumbra casting geometry. As a result, in environments where you would expect to find large penumbrae, this method tends to perform poorly.

Another hardware-based approach to soft shadow generation built around the shadow mapping algorithm was introduced by Wyman and Hansen [WH03] in 2003. They note that if you assume the shadow map approximates the umbra region, then when looking at a scene from the point of view of the light, the entire penumbra region is visible. Using this knowledge, they create a separate texture to store the penumbral intensity throughout the scene as observed by the light. This mapping of the penumbra intensities can be used in conjunction with the shadow map to produce realistic soft shadows. They build on Haines' plateau work [Hai01] to construct the relevant penumbra geometry from a series of cones and connecting planes. This algorithm has the ability to generate hardware-accelerated soft shadows at frame rates roughly half that of standard shadow mapping. The increased frame time is largely due to the computation of the silhouette boundaries, a necessary step for projecting the penumbra geometry. This approach suffers from the limitation that the umbra size does not shrink as the light radius grows. This is because it relies on shadow mapping to determine the umbra boundaries, and shadow mapping treats every light source as a point light. The method also makes the assumption that silhouette boundaries are constant across the surface area of the light.

Chapter 3

System Layout

The goal of our system is to merge live objects captured on video with synthetic imagery, in real-time. We create a high level pipeline that is capable of delivering, theoretically, 15 frames per second. This systems provides the user with an interactive experience as they work with our software.

Given the high level of computational complexity involved with geometric reconstruction, along with the goal of real-time results, the performance of our system is a high priority. Throughout this chapter, we focus on the design and implementation of our hardware and software system. We describe the difficulties which arise in trying to engineer a system that both captures geometry interactively and also merges that geometry in a believable manner with a synthetic scene. We justify the design choices that were made to assist in the future development of similar systems.

3.1 System Overview

The chain of events that our system performs can be broken down into eight broad stages, each of which has its own set of design difficulties that need to be addressed. A graphical illustration of the stages is shown in Figure 3.1. In Figure 3.2 we list some of the challenges associated with each of the respective stages. In stage one, each of the video cameras in our system must capture an image of the physical object that we want composited. Then, in stage two, the frames must be segmented into their foreground and background components. The next step, stage three, is to blur the segmented images. This removes small holes and smooths out the noise along the foreground/background boundaries. In stage four, a contour finding algorithm extracts the silhouettes from the segmented image. The foreground silhouettes later serve as input to the hull reconstruction algorithm. The frame and contour data is passed across the network from the client machines to the central server in stage five. This data is used in stage six to construct the polyhedral hull. In stage seven, the hull is textured with the images captured from the cameras. Finally, in stage eight, the composite of the virtual environment and the reconstructed hull is rendered to the screen. Using this approach, we can implement effects such as shadows and surface reflections between the real object and the virtual background.

The remainder of this chapter will be organized as follows. First, in Section 3.2, we will discuss our hardware setup, as well as the computing paradigm used within our system. Then we will briefly describe each of the major system modules referenced in Figure 3.1, with the exception of step six. Step six, the geometric reconstruction algorithm, will be covered in detail in Chapter 4.

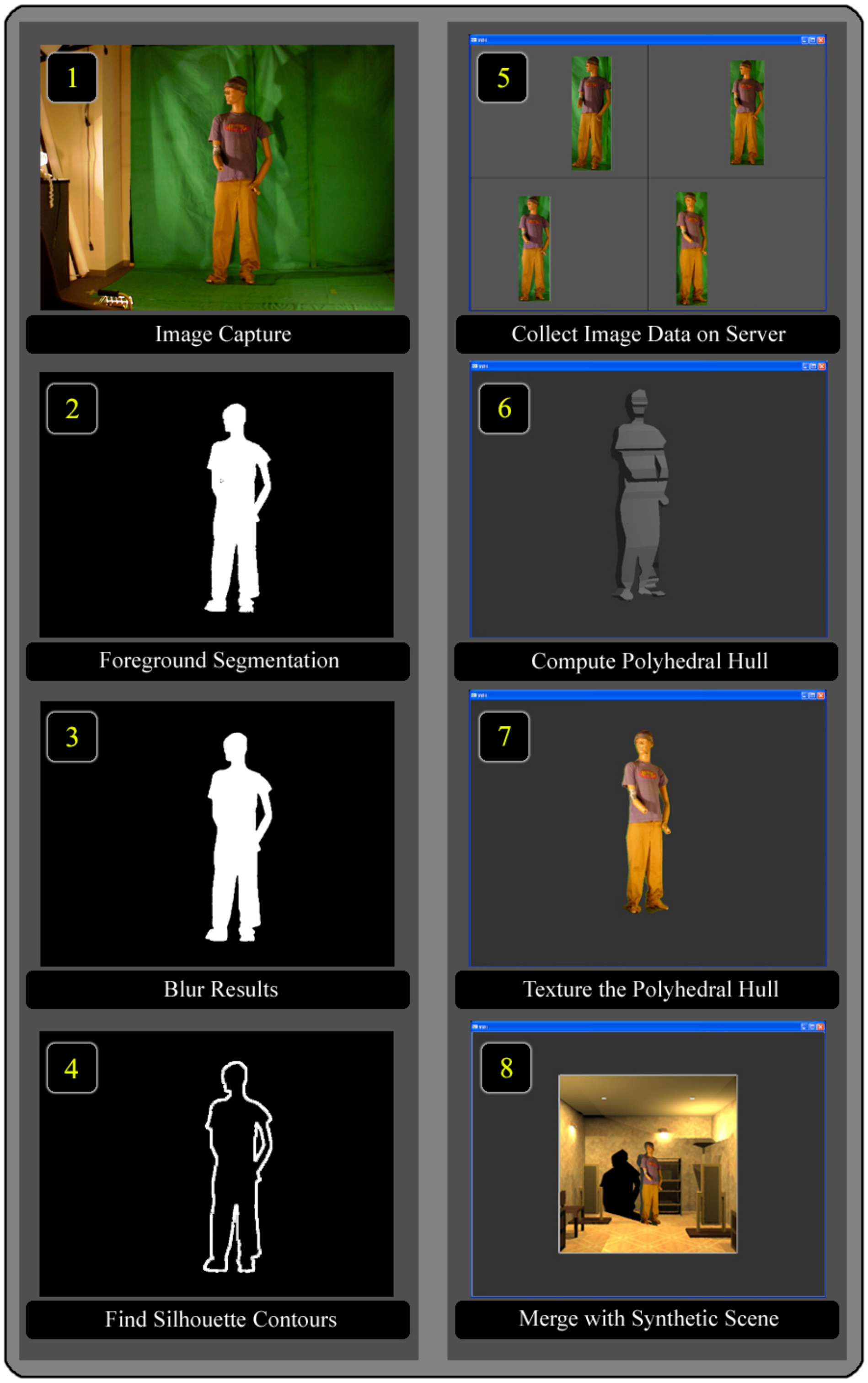


Figure 3.1: *The diagram above depicts each stage of the image processing, geometric reconstruction and compositing process.*

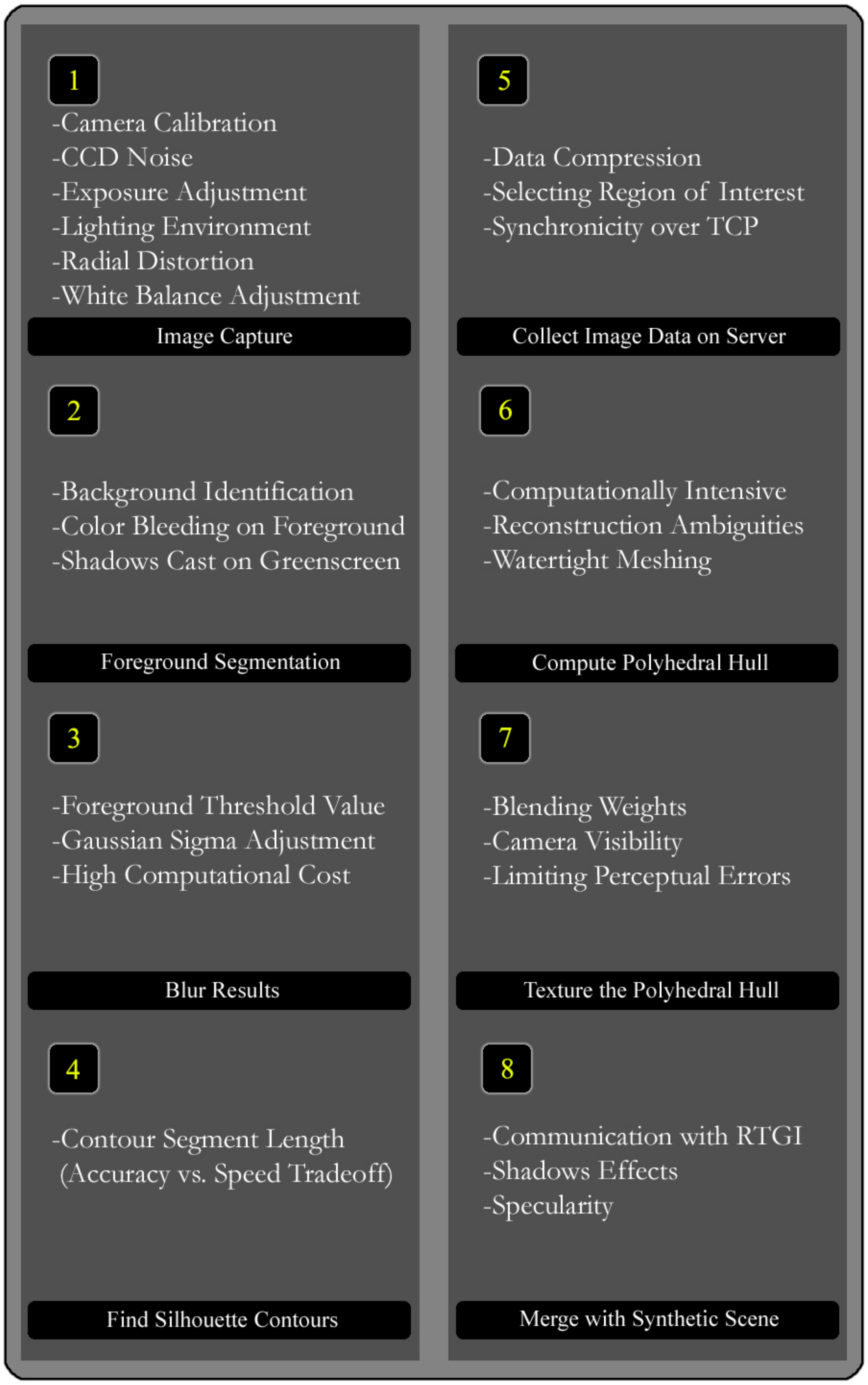


Figure 3.2: *The diagram above displays some of the challenges associated with each stage in the algorithm.*

3.2 Hardware Setup

In this section, we describe the hardware infrastructure of which our system is comprised. Further, we will discuss the client-server paradigm as selected for our computing model, and how this architecture has affected our system.

3.2.1 Computer and Camera Specifications

We use four Sony DFW-X700 cameras to capture the foreground object. Figure 3.3 shows one of our cameras. The cameras have a half-inch CCD, and provide progressive scan output at 15 frames per second up to a resolution of 1024x768. Each camera is coupled with a 7.5mm fixed focal length c-mount lens. At this focal length, we are able to capture a subject of approximately six feet in height from a distance of four meters. Each camera is linked to a separate client computer using the IEEE 1394 (FireWire) interface, supporting transfer speeds of up to 400Mbps. At 15 frames per second and a resolution of 1024x768 (three channel output), the maximum required bandwidth is 33.75 MB/sec, well below the upper limit of 50 MB/sec, even in the most demanding scenario. Each camera has an external trigger which allows for capture synchronization.

As previously mentioned, each camera is tethered to a separate client computer that performs the necessary image processing operations on the captured video frames. The computers are single processor 3.0GHz Pentium 4 machines with 512MB of RAM and Gigabit Ethernet cards. These machines perform one of the parallelizable steps in our processing pipeline. Specifically they extract the foreground object from the image and find the line segments which define its silhouette. These tasks are highly computational in nature, thus making the CPU

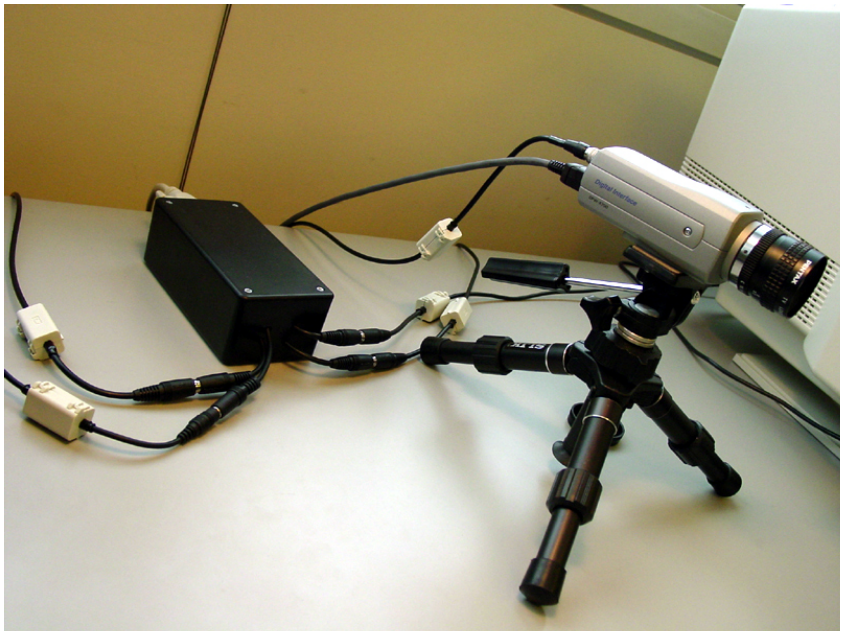


Figure 3.3: *The image above shows a DFW-X700 camera and the trigger circuit box.*

the most critical component.

The client computers are networked to the server using Cat 5E crossover cables which support Gigabit transfer speeds. The server has two two-port Intel 1000MT Gigabit server cards which allow the four client machines to each have a direct connection. The central server consists of dual 3.2GHz Pentium 4 processors with 2GB of RAM and an NVIDIA Quadro FX3000 graphics card. The server is responsible for the hull reconstruction, as well as the final hardware rendering and compositing, including shadow generation using advanced vertex and fragment shaders. In order to perform these tasks interactively, the server is required to have a high performance graphics board in addition to raw processing power. A representative image of our hardware configuration can be seen in Figure 3.4.

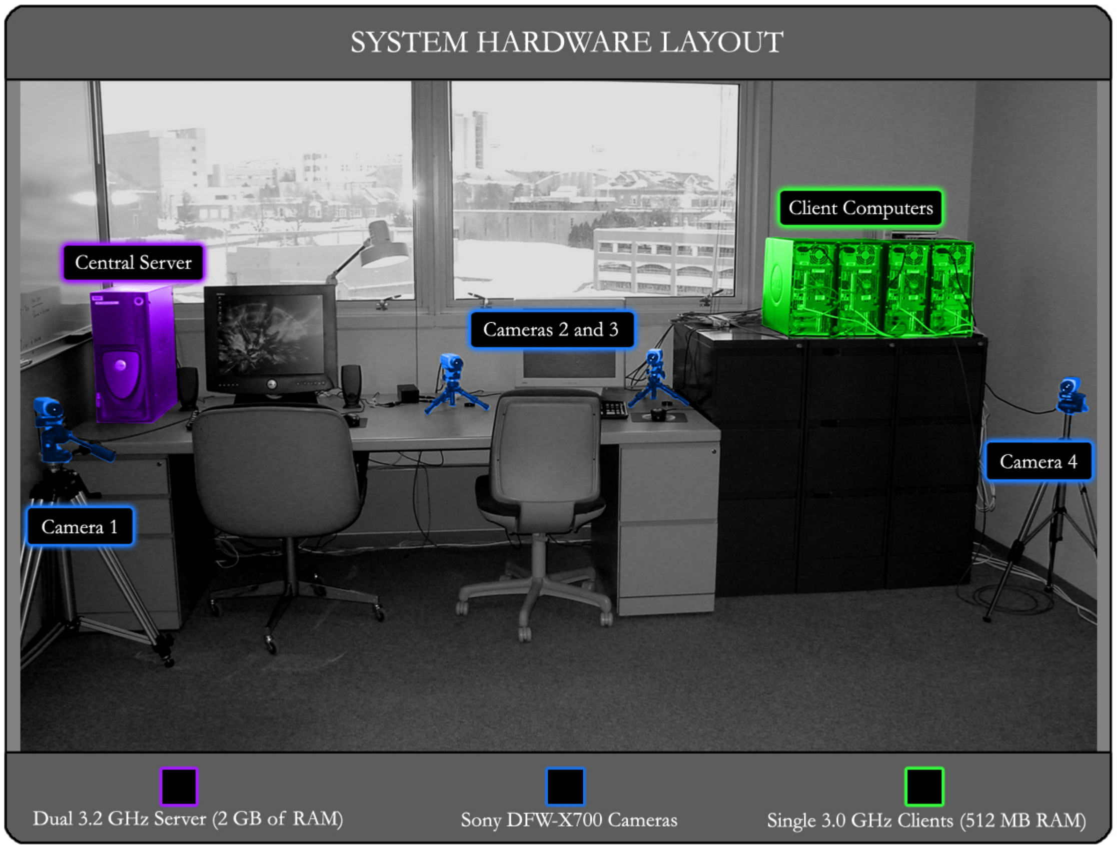


Figure 3.4: *The diagram above shows the hardware configuration as set up in one of our test environments. The diagram also highlights each of the individual components in our system.*

3.2.2 Client-Server Model

Many of the fundamental algorithms associated with geometric reconstruction and image synthesis are highly computational in nature. The goal of our project is to integrate these two areas into one encompassing system which is able to maintain interactive frame rates. This requires that we collimate as much of the work as possible. In order to achieve this parallelization, we implement a client-server model that distributes the workload across a networked array of computers. Each of the video cameras is attached to a separate client computer that performs the necessary image segmentation and contour finding operations. After this processing is performed, the client computers send the silhouette data and video textures across the network to the server. The geometric reconstruction occurs on the server. At the same time as the client computers are sampling the cameras and matting out the foreground object from the greenscreen, the Real Time Global Illumination (RTGI) system is running on a separate cluster of computers for background image generation, and the server is generating the polyhedral model from the previous frame. By pipelining these three disparate actions, we attempt to maximize the throughput in our system, and minimize the time spent waiting for other processes to complete their task. Since this is a pipelined system, there is an associated startup cost in processing, which results in a latency of one frame.

3.3 Image Capture

As shown in Figure 3.1, the first step in our system is to capture an image of the foreground object from each of the video cameras. This section covers camera calibration, the synchronization of the cameras, and the control of the cameras' internal and external parameters to achieve the best image quality and cleanest foreground segmentation. We also describe the techniques used to maintain a consistent lighting environment for capturing foreground geometry.

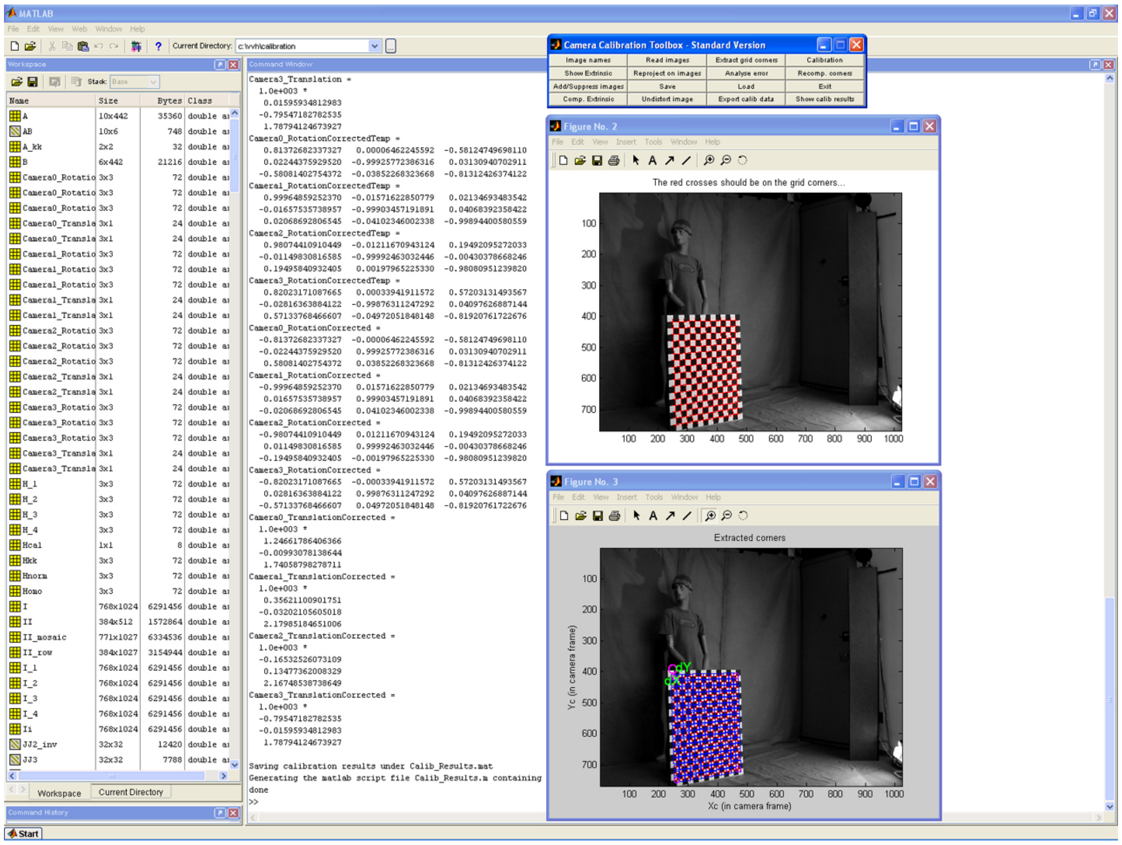
3.3.1 Camera Calibration

A fundamental requirement of the reconstruction algorithm is knowing both the position and the orientation of the cameras in relation to each other within a globally defined reference frame. These are known as the extrinsic parameters of the camera. The intrinsic parameters, namely the principal point and focal length, must be discovered through calibration as well. The principal point is the location where the principal axis intersects the image plane. The principal axis is the line passing through the camera center that is perpendicular to the image plane. To calibrate the cameras, we used the "Camera Calibration Toolbox for MATLAB" [1]. This is illustrated in Figure 3.5, which shows both our captured images and the MATLAB interface used to process them. To begin the calibration procedure, we placed a checkerboard pattern within the scene such that each of the four cameras had a clear view of the entire surface. We then captured a frame from each camera, and used those as input to the calibration routine. The calibration process, which uses Zhengyou Zhang's technique [Zha00], returns the grid reference frame with respect to each of the four cameras' reference frames. Since we desire the relationship that exists between the cameras, we convert each camera frame to being dependent on the grid's frame, so that the grid contains our global orthonormal basis. More details on how this was done can be found in the camera calibration section of the appendix, Appendix A.

3.3.2 Trigger Synchronization

It is imperative that each of the input images used for hull generation is captured at the exact same instant in time because the reconstruction algorithm operates by taking the intersection of the "silhouette cones". A silhouette cone is defined by an apex, which is located at the camera center, and the extrusion of the polygonal object silhouette away from the camera center to infinity. If the images are

[1]The Camera Calibration Toolbox for MATLAB can be downloaded at the URL: http://www.vision.caltech.edu/bouguetj/calib_doc/



```
Name              Size        Bytes  Class
A                 10x442      35360  double ar
AB                10x6          748  double ar
A_kk              2x2            32  double ar
B                 6x442       21216  double ar
Camera0_Rotatio   3x3            72  double ar
Camera0_Rotatio   3x3            72  double ar
Camera0_Rotatio   3x3            72  double ar
Camera0_Transla   3x1            24  double ar
Camera0_Transla   3x1            24  double ar
Camera1_Rotatio   3x3            72  double ar
Camera1_Rotatio   3x3            72  double ar
Camera1_Rotatio   3x3            72  double ar
Camera1_Transla   3x1            24  double ar
Camera1_Transla   3x1            24  double ar
Camera2_Rotatio   3x3            72  double ar
Camera2_Rotatio   3x3            72  double ar
Camera2_Rotatio   3x3            72  double ar
Camera2_Transla   3x1            24  double ar
Camera2_Transla   3x1            24  double ar
Camera3_Rotatio   3x3            72  double ar
Camera3_Rotatio   3x3            72  double ar
Camera3_Rotatio   3x3            72  double ar
Camera3_Transla   3x1            24  double ar
Camera3_Transla   3x1            24  double ar
H_1               3x3            72  double ar
H_2               3x3            72  double ar
H_3               3x3            72  double ar
H_4               3x3            72  double ar
Hcal              1x1             8  double ar
Hkk               3x3            72  double ar
Hnorm             3x3            72  double ar
Homo              3x3            72  double ar
I                 768x1024  6291456  double ar
II                384x512   1572864  double ar
II_mosaic         771x1027  6334536  double ar
II_row            384x1027  3154944  double ar
I_1               768x1024  6291456  double ar
I_2               768x1024  6291456  double ar
I_3               768x1024  6291456  double ar
I_4               768x1024  6291456  double ar
Ii                768x1024  6291456  double ar
JJ2_inv           32x32       12420  double ar
JJ3               32x32        7788  double ar
```

```
Camera3_Translation =
  1.0e+003 *
   0.01595934812983
  -0.79547182782535
   1.78794124673927
Camera0_RotationCorrectedTemp =
   0.81372682337327   0.00006462245592  -0.58124749698110
   0.02244375929520  -0.99925772386316   0.03130940702911
  -0.58081402754372  -0.03852268323668  -0.81312426374122
Camera1_RotationCorrectedTemp =
   0.99964859252370  -0.01571622850779   0.02134693483542
  -0.01657535738957  -0.99903457191891   0.04068392358422
   0.02068692806545  -0.04102346002338  -0.99894400580559
Camera2_RotationCorrectedTemp =
   0.98074410910449  -0.01211670943124   0.19492095272033
  -0.01149830816585  -0.99992463032446  -0.00430378668246
   0.19495840932405   0.00197965225330  -0.98080951239820
Camera3_RotationCorrectedTemp =
   0.82023171087665   0.00033941911572   0.57203131493567
  -0.02816363884122  -0.99876311247292   0.04097626887144
   0.57133768466607  -0.04972051848148  -0.81920761722676
Camera0_RotationCorrected =
  -0.81372682337327  -0.00006462245592  -0.58124749698110
  -0.02244375929520   0.99925772386316   0.03130940702911
   0.58081402754372   0.03852268323668  -0.81312426374122
Camera1_RotationCorrected =
  -0.99964859252370   0.01571622850779   0.02134693483542
   0.01657535738957   0.99903457191891   0.04068392358422
  -0.02068692806545   0.04102346002338  -0.99894400580559
Camera2_RotationCorrected =
  -0.98074410910449   0.01211670943124   0.19492095272033
   0.01149830816585   0.99992463032446  -0.00430378668246
  -0.19495840932405  -0.00197965225330  -0.98080951239820
Camera3_RotationCorrected =
  -0.82023171087665  -0.00033941911572   0.57203131493567
   0.02816363884122   0.99876311247292   0.04097626887144
  -0.57133768466607   0.04972051848148  -0.81920761722676
Camera0_TranslationCorrected =
  1.0e+003 *
   1.24661786406366
  -0.00993078138644
   1.74058798278711
Camera1_TranslationCorrected =
  1.0e+003 *
   0.35621100901751
  -0.03202105605018
   2.17985184651006
Camera2_TranslationCorrected =
  1.0e+003 *
  -0.16532526073109
   0.13477362008329
   2.16748538738649
Camera3_TranslationCorrected =
  1.0e+003 *
  -0.79547182782535
  -0.01595934812983
   1.78794124673927

Saving calibration results under Calib_Results.mat
Generating the matlab script file Calib_Results.m containing
done
>>
```

Figure 3.5: *The image above displays an example screenshot of the MATLAB camera calibration toolbox interface.*

captured at even slightly varying instances in time, then the silhouettes could potentially represent very different poses of the foreground object (assuming the object is moving). The intersection of these extruded silhouette cones would not accurately reproduce the original model at any of the different time steps, but would most likely appear as indiscernible noise, containing geometry only where the poses happened to overlap in space. To prevent this degenerate scenario from occurring, the server is responsible for triggering each of the cameras simultaneously in the main loop of the program, ensuring that the sampling is consistent at each time step. The server drives the trigger circuit by outputting signals on the serial port, COM1. For implementation details concerning our use of the serial port as well as our trigger circuit design, refer to Appendix B.

3.3.3 Use of DirectShow for Camera Control

In order to establish software communication with the video cameras, we took advantage of the DirectShow interface, a component of Microsoft's DirectX 9 multimedia suite. DirectShow allows for the configuration of a filter graph to manage the flow of data from the camera capture device to the user's program. DirectShow also provides an interface for setting such camera parameters as white balance, exposure, video capture format, and the active state of the external trigger. Our camera control software, which runs on each of the four client computers, uses the DirectShow interface to adjust the color balance of the incoming image in order to achieve optimal greenscreen results.

3.3.4 Whitebalance Control

To improve the results of the image segmentation process, the whitebalance on the camera should be adjusted. By slightly strengthening the red and blue channels it is possible to remove the excess color bleeding from the greenscreen that occurs on light-colored surfaces. Alternately, by decreasing the red and blue channels it is possible to compensate for shadows on the backdrop which leave parts of the greenscreen marked as foreground. Decreasing the red and blue channels will make the greenscreen surface appear more green, and thus can lead to less noise and better results. Figure 3.6 illustrates these effects. Making this trade-off in color balance is a subtle art, as it is important that the captured image appear as natural as possible while remaining easy to segment cleanly. To maintain consistency,

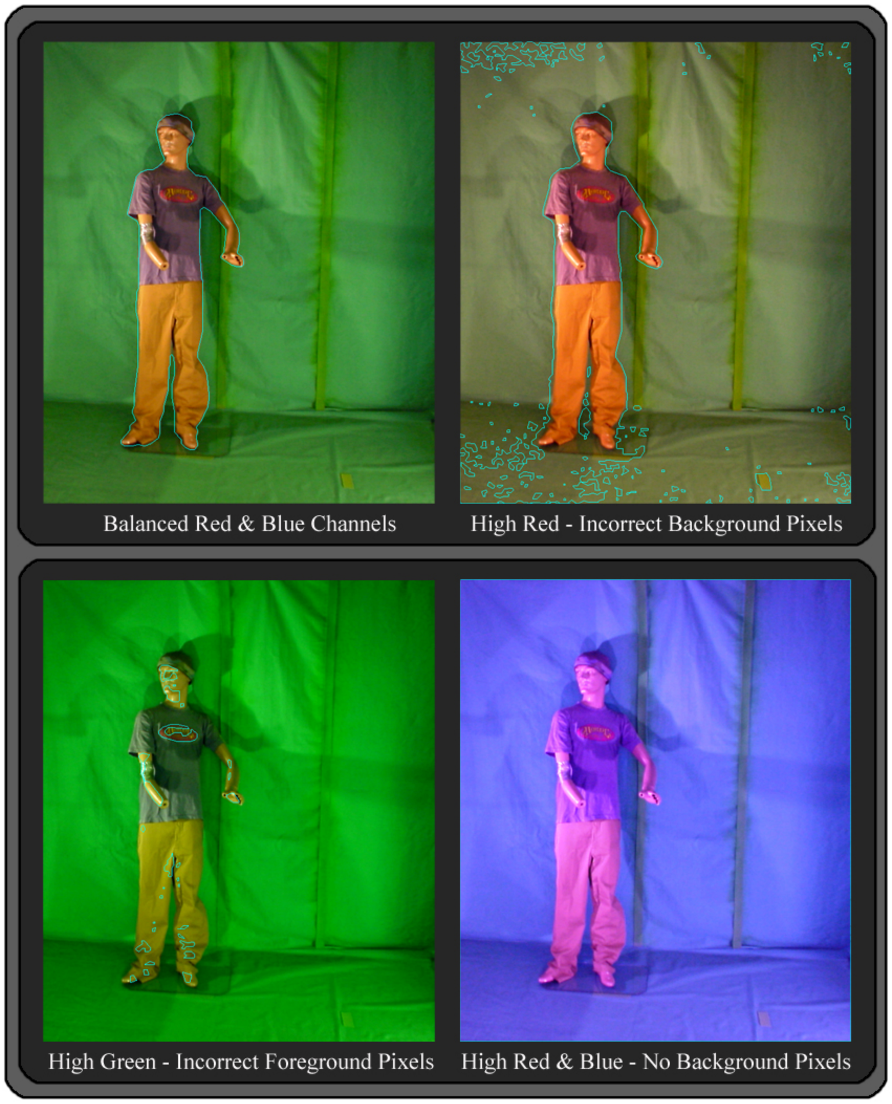

Balanced Red & Blue Channels
High Red - Incorrect Background Pixels
High Green - Incorrect Foreground Pixels
High Red & Blue - No Background Pixels

Figure 3.6: *The figure above shows a series of different color-balance adjustments, and the effect that each setting has on the segmented image.*

the program saves a configuration file that documents the last used whitebalance values. These settings are then re-loaded the next time the program is started. Similarly, values for the matte strength, blur threshold, and other camera-related parameters are also stored and loaded.

3.3.5 Scene Lighting

The lighting of the foreground object and the greenscreen is an important factor in determining the quality of the image segmentation and the resulting silhouette contours. Ideally the greenscreen would be placed at infinity, so that there would be no interactions between it and the foreground subject. Allowing the subject to be too close to the greenscreen often results in color bleeding, where parts of the foreground object, most notable the edges, take on a greenish tint due to light that is first been reflected off the screen and then bounced off the subject. Another situation that can lead to problems is when the foreground object casts shadows on to the surface of the greenscreen. This can be problematic because the darker shadowed regions of the background are often misinterpreted as foreground. The way we addressed these issues was to place several 500 watt halogen lamps in a semicircle configuration around the object acquisition area. Our goal was to simulate a diffuse environment where light was coming from all directions, and thus eliminate any hard shadows.

3.4 Image Segmentation

After an image has been captured from each of the video cameras, the next step is to segment the foreground from the background. In this section we will cover the initial foreground matting, as well as the subsequent Gaussian filtering operation that we use to eliminate high frequency noise.

3.4.1 Foreground Matting

After the cameras have been triggered and a frame returned, the next step is to segment the image into its foreground and background components. We borrow the method employed by Selan [Sel03] for performing this operation. If a pixel's green channel value is higher than the maximum of the red and blue channels by a preset margin then the pixel is marked background. This amount, termed the "matte strength", is a variable that can be adjusted in the software. If, conversely, the green channel is not significantly larger than both the red and blue channels, or if the pixel is extremely bright, with a combined channel value over some predefined threshold, then it is marked as foreground. When performing segmentation, we are only concerned with the region of the image that corresponds to the greenscreen (a subset of the entire image). The bounding box which defines this region of interest, or ROI, can be set in our software. Any pixel which falls outside this area is automatically marked as background and set to black in the segmented image. Figure 3.7 shows the greenscreen backdrop for our image acquisition area.

3.4.2 Gaussian Blurring and Thresholding

The segmented foreground image is often plagued by noise, due to the imprecise nature of the scene lighting and the image sensing device, as well as the non-homogeneous color of the greenscreen. As discussed in Section 3.3.5, areas of the foreground that suffer from heavy color bleeding often have a greenish tint, and are therefore erroneously marked as background. Conversely, areas of the background that receive heavy shadowing are often not recognizably green, and are thus deemed foreground pixels. Inaccurately labeled pixels are also common where the greenscreen meets the floor, as the corner is usually dark, and around perturbations

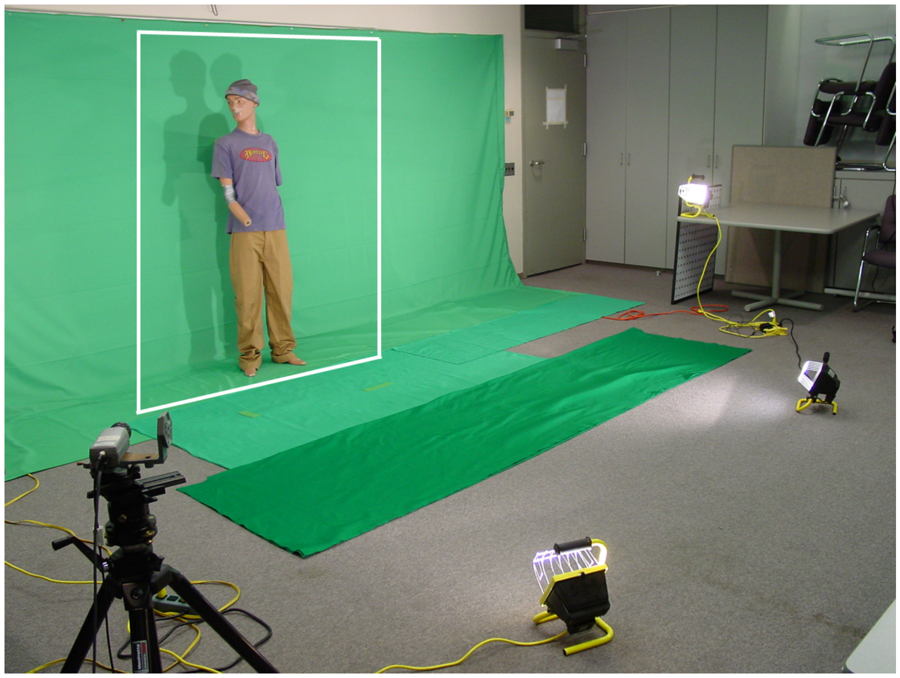


Figure 3.7: *The figure above shows the greenscreen that was used for segmenting out foreground geometry. The white box outlines the image acquisition area, where the four cameras' fields of view overlap.*

in the greenscreen fabric or areas where the lighting undergoes sharp transitions. We attempt to minimize these problem areas through physical means, however, it is difficult to eliminate them completely. We also rely on software techniques to aid in the image segmentation process. One way we alleviate noise is by convolving the segmented image with a Gaussian filter. Our Gaussian filter implementation, which takes advantage of dynamic programming techniques, makes multiple passes over the image in both the horizontal and vertical directions, each time convolving the image with a box filter. We have found empirically that a Gaussian sigma value between two and four works relatively well for removing the noise in our captured images.

After the image has been blurred, each pixel has a grayscale value between

0 (black) and 255 (white). The next step is to perform a threshold operation, such that pixels which contain a value above the threshold are marked foreground, or white, and those pixels with a value below the threshold are differentiated as background, or black. This threshold value is a variable in software, and can be adjusted as necessary for optimal segmentation.

3.5 Contour Finding

The fourth stage in our system is to find the contours that approximate the object's shape. This section will discuss how we recover the silhouettes from the segmented images, and a noise elimination approach that we use to prune away unwanted contours.

3.5.1 Polygon Approximation

After the image has been segmented into foreground and background regions, the silhouette contours that define the foreground object are extracted. This is done using Intel's open source computer vision library, OpenCV. We use the routine *cvFindContours()* , which uses an algorithm similar to that of marching squares in order to find and return the contours in a binary image. The contours that the algorithm recovers are initially very fine in resolution, such that each edge only spans neighboring pixels. To adjust the granularity of the contours, stringing together short edges in order to produce more representative line segments, we leverage the function *cvApproxPoly()*, which takes as input the original contours and a constant that denotes the desired level of contour resolution. The higher the constant, the more coarse the final contours will be. A value of zero returns the original contours with no change in granularity.

There is a trade-off to be made when using this routine. The higher the value, the fewer the line segments there will be. This will result in less data, lower network transfer times, and a quicker hull reconstruction process. However, the disadvantage is that the geometry is less refined. For example, when finding the contours of a hand, if the polygon approximation factor is set too high, the individual fingers will be lost and the hand will appear as a single polygon. If the approximation factor is set too low, then the contour definitions will be more detailed but the running time of the hull intersection routine will be much higher. This is a direct result of the fact that there are now many more surfaces on which to perform polygon intersections. The goal is to find a balance such that the contours are of a high enough resolution to generate visually pleasing models and yet low enough to maintain interactive frame rates. One proposed solution, employed by Matusik et al [MBM01], is to have the approximation factor adjusted on the fly in software. The contour resolution is decreased automatically when the program slows down, and increased when there are extra cycles to devote to the mesh generation process. In our system, the variable is user driven, as opposed to software controlled, and can be adjusted on the fly to match the user's current desire.

3.5.2 Noise Elimination

Despite our effort to eliminate noise at the segmentation stage, there are still occasionally incorrectly marked pixels that bleed into the contour finding routine. To counter this, we automatically throw away any contour that consists of three or less edges. No reasonable foreground object would be so nondescript, and thus we have found this to be a relatively successful way of removing background noise from the contour detection process.

3.6 Data Transfer to Server

In order for the server to reconstruct, and later texture, a model of the foreground object, it must acquire the necessary contour and image data from each of the client machines. This is the fifth step in our system overview diagram, Figure 3.1, and will be the topic of this section.

3.6.1 Network Protocol and Region of Interest

After the silhouettes have been extracted from the segmented image, a buffer is constructed that stores all the relevant data the server will need to compute the final polyhedral hull. Instead of sending the entire video frame to the server, which is later used for texturing the constructed hull, we only send the region that contains the foreground object. By minimizing the amount of data transferred between the client and server, the transfer times and the bandwidth consumed are also optimized. To determine the minimum bounding extent of the foreground object, we iterate through the line segments which define the contours and keep track of the minimum and maximum (x,y) points. These two points constitute a bounding box that encompasses any pixels potentially required during the texturing stage. The pixels inside this region of interest, along with the contour definitions, are then sent across the network. All our network communication is done over sockets, using the winsock library that comes standard with the Windows operating system. Figure 3.8 shows the hardware configuration of our system, and Figure 3.9 shows both the utilized and theoretical maximum bandwidth between each component.

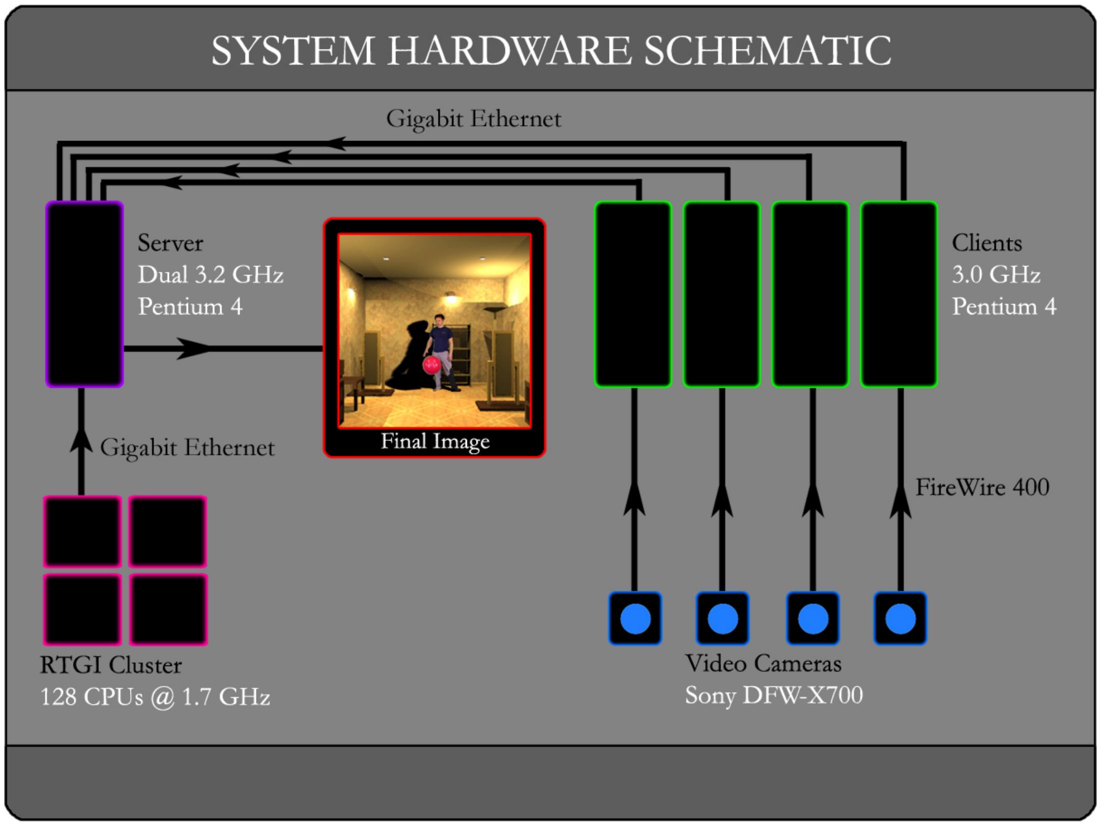


Figure 3.8: *The diagram above shows the hardware configuration in our system.*

3.6.2 Use of Compression

In order to reduce the transfer times between the client computers and the server, we experimented with compressing the data before sending it over the socket. We used the zlib compression library for our testing [2]. Unfortunately, the time required to compress and decompress the data was more than the time saved in transmission. This may not be true for all forms of compression (for example the JPEG algorithm might prove faster), or for all data set sizes. However, we have currently settled on sending the data in its raw form.

[2]The zlib library can be downloaded at the URL: http://www.gzip.org/zlib/

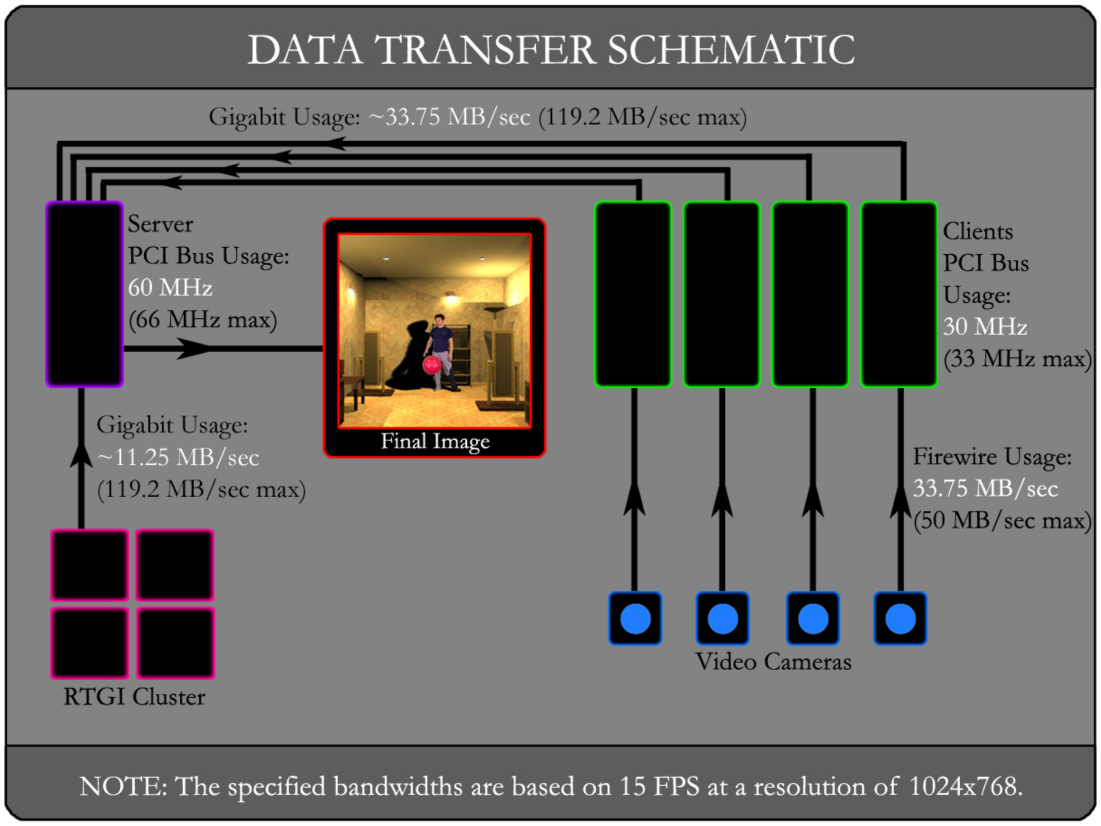


Figure 3.9: *The diagram above displays our system's utilized and theoretical data transfer rates.*

3.7 Texturing the Polyhedral Hull

After the hull has been reconstructed, as covered in Chapter 4, the next step is to texture the model using the original video frames. In this section, we review our initial naive texturing approach. For a discussion of the advanced techniques implemented, we point you to the thesis of Adam Kravetz [Kra04].

3.7.1 Naive Texuring Method

The source textures for the polyhedral hull are always comprised of the original video streams. In our naive texturing implementation, the process of selecting the optimal video image for texturing a hull surface reduces to a question of visibility and orientation. In the best case scenario, each vertex to be textured would correspond directly to a pixel from the set of video images. It is unlikely, however, that this will be the case. Therefore, we attempt to find the nearest match. The correctness of a match is quantified by the dot product between the viewing vector and the surface normal. The dot product indicates the cosine of the angle between the the two vectors, and thus we seek the smallest dot product for each surface. Ideally, the vertex has an exact match, and the vectors are directly opposing, resulting in a dot product of negative one. Using the aforementioned technique, we compute the visibility between each reference camera and hull surface pair. We then use the video frame from the camera with the best viewing angle for texturing that particular surface. Figure 3.10 shows an example hull surface and the camera that our naive algorithm would select for texturing the face.

3.8 Foreground and Background Compositing

The final stage in our system is to merge the reconstructed foreground object with the background environment. It is at this point that we add in global illumination effects, such as shadows and reflections, to enhance the plausibility of the composite. This section will focus on the method by which we generate our background images, as well as the compositing process. The shadowing technique that we developed for our system is discussed in detail in Chapter 5, and our implementation of reflections is covered in [Kra04].

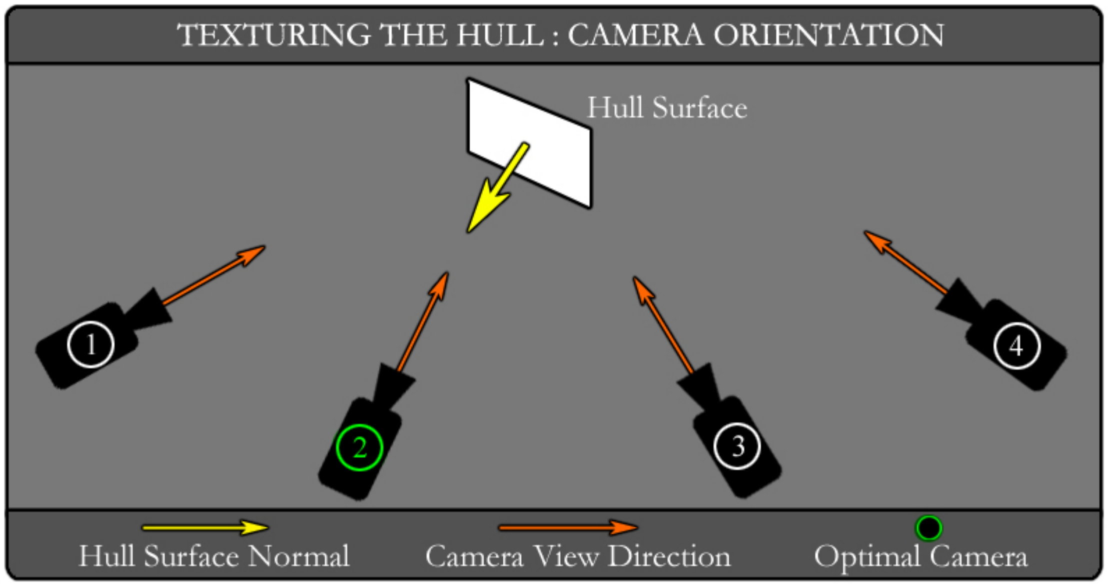


Figure 3.10: *The optimal camera for texturing a hull surface is the one whose viewing vector has the most opposing angle to the surface normal. In the diagram above, Camera 2 would be the best choice, as it has the most direct view of the face.*

3.8.1 Background Image Generation

RTGI is the Program of Computer Graphics at Cornell's "Real Time Global Illumination" system. It is a ray tracer that is designed for both interactive walkthroughs and the rendering of static scenes. The RTGI system can render in many different modes, each having varying levels of quality and performance; however, these options are all transparent to our system. As far as our software is concerned, RTGI is a "black box" that feeds it background images for compositing with our captured geometry. RTGI can be run in one of two modes, on a single client computer or in walkthrough mode, which takes advantage of the PCG's 128 CPU cluster [3]. When running in walkthrough mode, the pixels are distributed among each of the

[3]The cluster consists of 64 computers, each with dual 1.7GHz processors and 1GB of RAM.

machines, so that you can add secondary effects such as indirect lighting while still achieving interactivity. Although the RTGI system can handle an arbitrary number of light sources, our compositing software currently only supports hardware shadow generation for the primary light source in the scene. This allows us to maintain user interactivity, as implementing hardware shadows from multiple light sources requires additional rendering passes and texture lookups during the shadow generation stage. As a result, we primarily deal with background environments that have a relatively small number of lights.

3.8.2 Information Sharing

There are several parameters which need to be synchronized between the RTGI system and our software. These include the camera position and orientation, the primary light source location, and the current background model used for compositing. It is important to keep the cameras synchronized between the two systems, so that if the user moves the camera within the RTGI framework, thus changing the orientation of the background model with respect to the viewer, the captured foreground geometry will be viewed from an identical pose. If this consistency is not maintained, the alignment of the foreground object with respect to the background scene will be constantly changing – a highly disconcerting visual effect. Therefore, in order to guarantee compatibility, the RTGI camera position, view direction, up vector, field of view, and aspect ratio are transmitted to our software each frame. Similarly, the main light source in the scene must also be in the same position on both ends of the system. The RTGI environment allows the user to modify the light positions on the fly, and consequently our software needs to be made aware of such changes. This ensures that the hardware shadows cast by the foreground object will match the rest of the shadows in the background image. Unfortunately, as it is beyond the scope of this project to attempt to re-light the captured video frames, any changes to the scene lighting will not be incorporated in the textures used to shade the foreground geometry. Thus the user must maintain a plausible lighting environment in which to insert the captured model. RTGI is also responsible for which background model is currently in use, and it needs to alert our software when a model transition is being made.

3.8.3 Hardware Compositing Process

When a background image is received from the RTGI system, it is loaded into the color buffer on the graphics board. Our software stores a local copy of the scene model, and the next step is to render this geometry into the depth buffer. While the scene geometry is being rendered, the color buffer on the graphics card is disabled for writing. This prevents the ray-traced background image, which we previously loaded into the color buffer, from being overwritten with a hardware generated version. Then the color buffer is re-enabled for writing, and the captured foreground geometry is drawn. Because the depth buffer already has the scene geometry at this point, the depths are composited correctly.

In lieu of having our program render the scene into the depth buffer, we also experimented with RTGI forwarding along the depth data for the background image. Since the ray tracer has already generated a depth value for each pixel in the scene, it would seem logical to try and reuse this information by passing it over the network to our reconstruction server. However, we found that the time required to transfer the depth data across the network and load it into the z-buffer on the graphics board was significantly longer than it took to render the scene in hardware. Furthermore, a copy of the scene model is required on the reconstruction server for rendering shadows.

When performing shadow generation in our system, one needs to be able to render the scene from the point of view of the light source, thus requiring a local copy of the scene geometry. This might prove to be computationally prohibitive when dealing with highly complex scenes, for example when there are multiple polygons that exist within the space of a single pixel. Should this scenario ever arise, it may prove advantageous to revert back to a system where RTGI passes over the depth data for the generated background image, and then use a different approach for shadowing. This would eliminate the need for the compositing software to maintain a local copy of the scene model and would achieve a greater separation of complexity.

3.8.4 Object Positioning

The software which we have developed allows the user to rotate, translate, and scale the captured geometry so that it fits correctly within the scene. Exact positioning is necessary in order to create a convincing composite. Furthermore, the scenes in which we are inserting the subject are often modeled using arbitrary scales, uncorrelated to any physical units. In order to scale the model without also affecting its positioning, we first determine the center of the object. We implemented two different methods for computing the center of the object. The first approach is to use the average position of all the points which define the object's hull, which approximates the centroid of the object. The second approach is to use the center of the bounding box which surrounds the geometry. The later method has proven to be the better of the two, as the exact positioning of the vertices which define the hull are constantly changing due to slight variations in image intensity and numerical imprecision during the reconstruction processes. Using the former method, the centroid shifts slightly each frame, and consequently the model skips around within the scene. This motion is highly disconcerting, and detracts from the realism. The bounding box method is less affected by the inherent noise which plagues the individual vertex positions. The temporal variation is much less, and consequently the jittering of the object within the scene is also drastically reduced.

Finally, to position the foreground object within the scene, it is first translated such that its center is situated at the origin. The object is then scaled up or down to the user-specified size and then translated back to its original location.

Chapter 4

Reconstruction

4.1 Geometry Reconstruction

At the very core of our system, around which the entire compositing infrastructure is built, is the geometric reconstruction algorithm. In order to create the illusion that a physically disparate foreground object is an integral part of a synthetic environment, it is highly advantageous to know the object's approximate shape. Even with limited geometry, there exists the potential to create much more accurate composites than would be possibly with only a 2D representation. With a 3D model, occlusion testing can be performed between the foreground and background, and it becomes possible to cast physically-based shadows both from and onto the reconstructed hull. The remainder of this chapter describes the algorithmic approach that was used to reconstruct the shape of an object based on its silhouette contours as seen by several reference cameras.

4.1.1 Epipolar Geometry

The polyhedral visual hull reconstruction algorithm, developed at MIT by Matusik et al [MBM01], has its foundations in epipolar geometry. Epipolar geometry is a way of relating the views of two cameras at different locations, and is useful for making correspondences between these pairs of images. The only information necessary to begin realizing these relationships is the intrinsic parameters of the cameras as well as their pose in world space. When dealing with a single camera, there is no way to determine exactly where in world space a point on the image plane lies. A point on the image plane could represent an infinite number of locations in world space, all of which lie along the ray formed by back projecting the image point through the camera center (see Figure 4.1). This information proves useful when making correspondences between multiple cameras, as it can be seen that a point in the first camera's image plane is restricted to a location along the back projected ray as seen by the second camera. Thus the search space for making image correspondences is drastically reduced. Instead of having to search the entire second image for the corresponding point, the search can be limited to a single line of pixels on the image plane. This relationship between points in one image and lines in a second image is expressed through the fundamental matrix.

$$l' = Fx$$

F = Fundamental matrix relating the cameras

x = Point in the first camera's image plane

l' = Line in the second camera's image plane

The search line generated above is referred to as an "epipolar line" because it is guaranteed to run through the epipole. The epipole is the image of the first

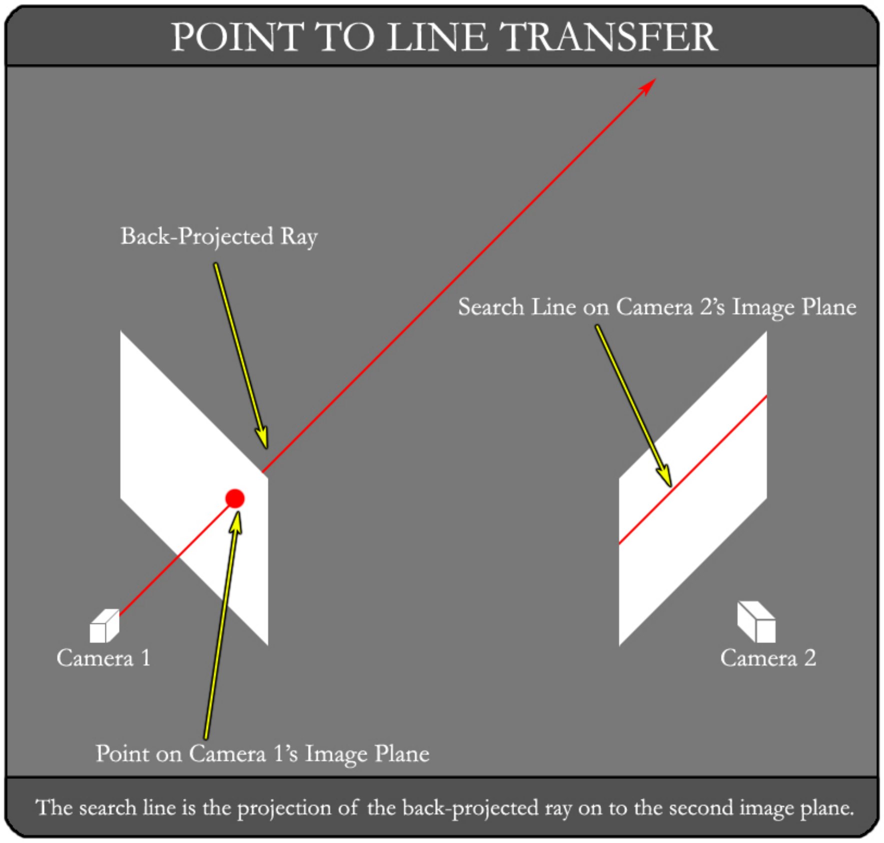

POINT TO LINE TRANSFER
Back-Projected Ray
Search Line on Camera 2's Image Plane
Camera 1
Camera 2
Point on Camera 1's Image Plane
The search line is the projection of the back-projected ray on to the second image plane.

Figure 4.1: *In the figure above, we see that a point in the first image plane becomes a line when projected on to the surface of the second image plane.*

camera in the second camera's image plane. One way to think of the epipole is the point where the baseline, the line connecting the two camera origins, intersects the image plane.

When configuring the pose of the cameras in a scene, it is critical that the baseline pass through the image plane in the proper location. An example of a valid configuration is shown in Figure 4.2A. There is a degenerate case which can arise when the cameras are not facing each other that results in the epipole inaccurately describing the remote camera's location. This is illustrated in Figure 4.2B, where the second camera is led to believe that the first camera is on its right, when in reality it is positioned on its left. Because of the fact that all the projected epipolar lines run through the epipole, lines on the image plane can be classified based on the angle they make against a reference line. This plays an important part in the hull reconstruction algorithm discussed in Section 4.1.3.

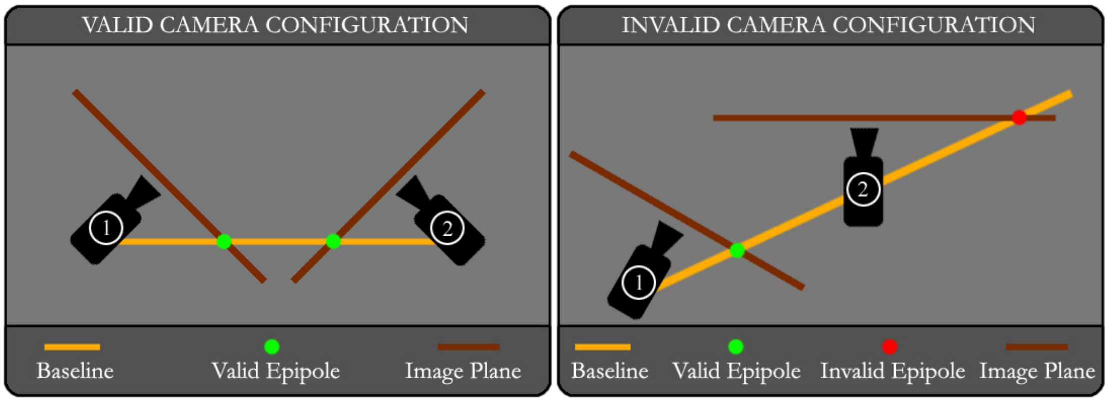


Figure 4.2: *(A) A valid camera configuration with the epipoles correctly identifying each camera's location. (B) An invalid camera configuration whereby the second camera believes the first camera to be off to its right.*

4.1.2 Visual Hulls

When looking at an image of a single object, there is no way to determine the size of that object. It could be infinitely small and positioned directly in front of the lens, or it could infinitely large and located a great distance from the lens. What is known is the fact that if the silhouette of the object in the image plane is extruded from a point at the camera center out to infinity, the object would lie completely within that space. The cone whose apex is the camera center, whose shape is defined by the silhouette of the object, and whose direction of extrusion is the vector along which the camera is looking, forms the visual hull of an object when dealing with a single camera. This cone is the minimum bounding volume which guarantees that the object is completely inside of it.

Now consider the case where there are two cameras. The result is two bounding volumes, each of which completely encapsulates the object. Each bounding volume is a cone starting at its respective camera center and extruding through the silhouette associated with that camera. Given that the two cameras are looking at the same object, the object must be contained within the intersection of the two extruded cones. As is expected, the region of intersection of the two cones is going to be a much smaller volume than that of either of the two original cone volumes, assuming the cameras have sufficiently varying poses within the scene. Each additional camera/cone pair acts as a "cookie cutter", confirming those regions in space that it agrees are valid portions of the object, and pruning away those areas in which it can definitively say the object does not exist.

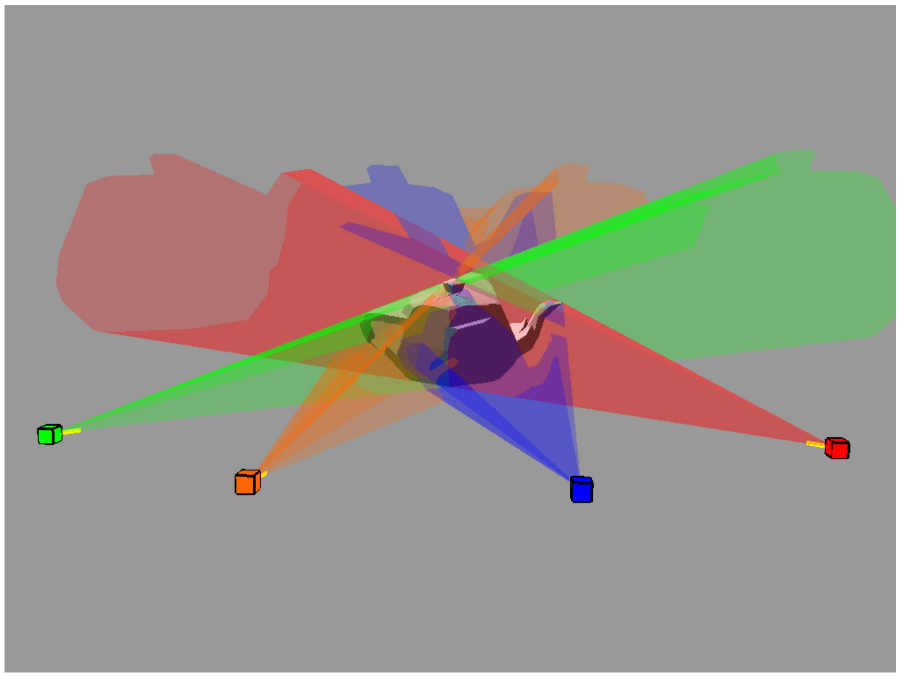


Figure 4.3: *The figure above shows the intersection of four silhouette cones that were used to generate the polyhedral visual hull of a teapot.*

When intersecting the silhouette cones of multiple cameras, ambiguities can arise. For example, compare the original and reconstructed objects in Figure 4.4 to those in Figure 4.5. In the first case, the intersection results in the proper reconstruction of the single cylinder. In the second case, however, ambiguities arise concerning where two cylinders are located within the scene. There are multiple solutions that would satisfy the silhouette cone constraints, and result in the reconstruction of four regions instead of two. These ambiguous cases are reduced with each additional camera added to the system.

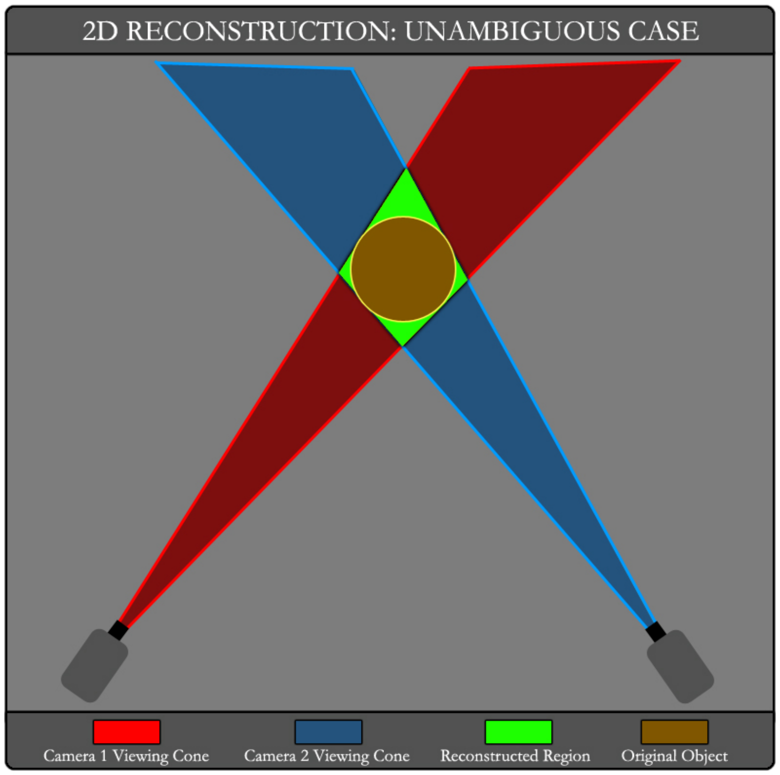


Figure 4.4: *The figure above illustrates the unambiguous reconstruction of a single cylinder from two silhouette cones.*

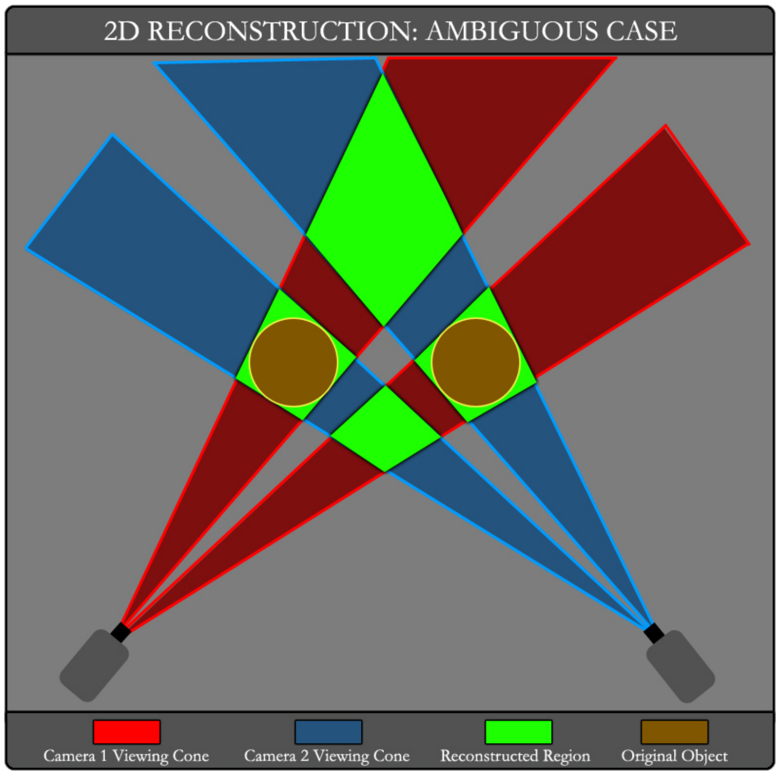


Figure 4.5: *In the figure above, there are multiple solutions that satisfy the silhouette cone intersections. These ambiguities can be eliminated by adding additional cameras to the system.*

4.1.3 Polyhedral Visual Hull Overview

Matusik's polyhedral visual hull algorithm [MBM01] is a way of creating an explicit three dimensional model by performing the silhouette cone intersections in two dimensions. This technique eliminates the complexity of performing the intersections in three-space (using CSG, for example) by leveraging the image correspondences that epipolar geometry provides.

Each edge in each silhouette image represents a plane in world space. This can be seen by recalling that a back projected point on the image plane produces a ray in world space. Thus the two image plane points which define a silhouette edge will result in two world space rays that define the boundaries of a world space plane. The projection of this plane on to another camera's image plane can be achieved through the use of the fundamental matrix. Using the epipolar geometry previously discussed, the two vertices which define the silhouette edge on the first camera's image plane can be multiplied by the fundamental matrix which relates the two cameras to produce two lines which run across the second image. These lines represent the projection of the edges of the world space plane, formed by back-projecting the silhouette edge, on the second camera's image plane. When the projected lines are intersected with the second image's silhouette, the resulting two dimensional region on the image plane corresponds to a surface of intersection on the world space plane formed by the first camera's back-projected edge. This can be seen in Figure 4.6. By projecting the region into world space, such that it lies on the first camera's edge plane, and intersecting it with the other regions created by projecting the same silhouette edge on to each of the other cameras' image planes, one is left with a surface on the final hull. The hull consists of the summation of all such surfaces that are created by projecting each silhouette edge in each image on to each other image plane. An example hull is displayed in Figure 4.7, where the polyhedral hull surfaces are shaded according to which camera in our four camera configuration generated that face.

4.1.4 Edgebin Creation

The "edgebin" data structure [MBM01] is the means by which the polyhedral visual hull algorithm quickly intersects the projected lines with the current image's silhouette. Each vertex in the object's silhouette is assigned an index value based on the slope of the line connecting that silhouette point to the appropriate epipole.

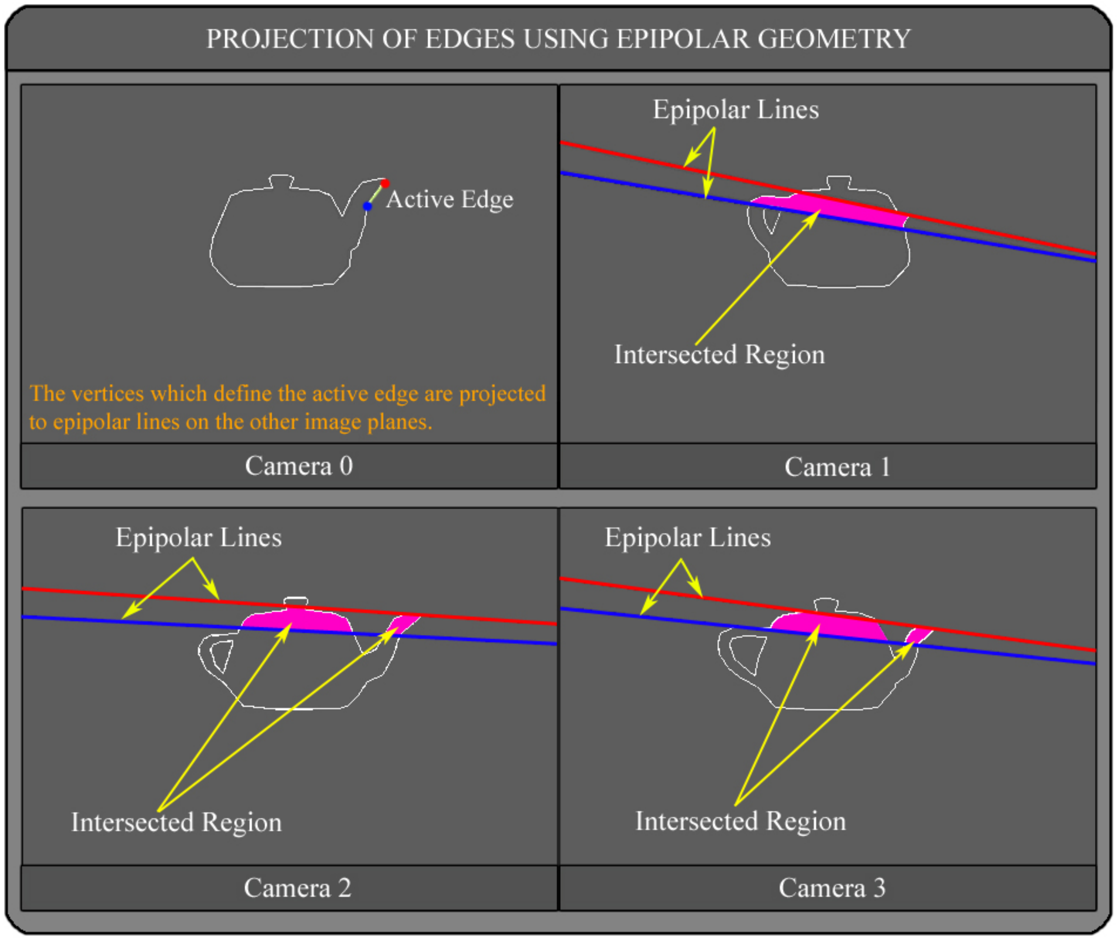

PROJECTION OF EDGES USING EPIPOLAR GEOMETRY
Active Edge
The vertices which define the active edge are projected to epipolar lines on the other image planes.
Camera 0
Epipolar Lines
Intersected Region
Camera 1
Epipolar Lines
Intersected Region
Camera 2
Epipolar Lines
Intersected Region
Camera 3

Figure 4.6: *In the figure above, a silhouette edge in Camera 0 is projected to epipolar lines in Cameras 1-3. These lines are intersected with the foreground silhouette from each camera, forming the shaded regions shown in pink. The intersection of these shaded regions in world space forms a surface on the polyhedral visual hull.*

The vertices are then sorted in ascending order based on their index values. The next step is to create a series of bins that span the index values from negative infinity to positive infinity. The first bin is defined as spanning from negative infinity to the first index value, the second bin spans from the first index value to the second, continuing in a similar fashion until the last bin is reached which spans from the highest index value to positive infinity. Each bin then stores a list of all

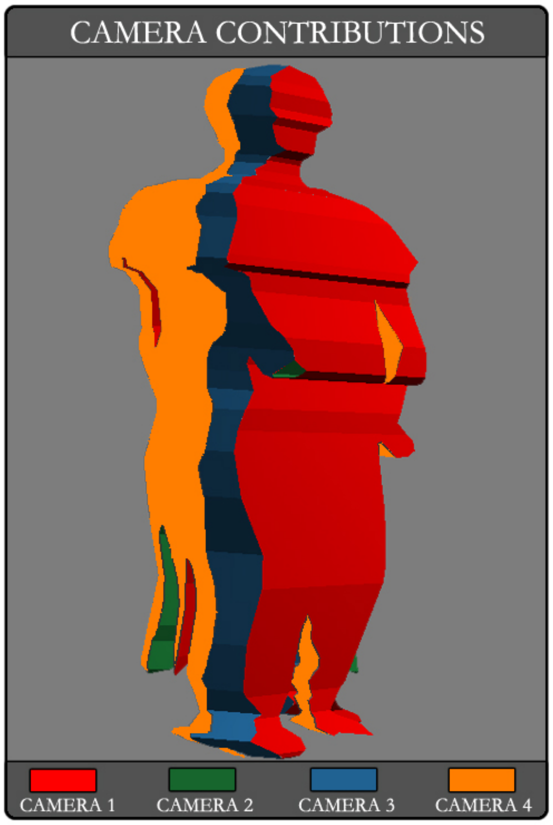


Figure 4.7: *In the figure above, we see the surface contributions from each of the four reference cameras used to produce the polyhedral visual hull.*

the edges whose start and end indices overlap the values that define the bin. Once the edges have been placed in the appropriate bins, the edges within each bin are sorted based on their distance from the epipole. This permits the traversal of the edges in a sequential fashion, moving from the epipole outwards across the image plane.

4.1.5 Polygon Generation

After the edgebins have been established, the edges in the first camera's silhouette are projected on to the second camera's image plane. Each projected edge creates two lines on the second image plane, and the index values for those two lines are computed. The next step is to select the line with the lower index value and determine which edgebin the line's index would fall in. Once the appropriate edgebin is found, the line is intersected with each edge in the bin by taking the cross product of the two lines in homogeneous form. The intersection points are added in turn to the intersection polygon. Then the algorithm iterates through the bins in sequential order, adding each bin's starting vertex to the intersected polygon, until the bin that spans the second projected line's index value is reached. At that point the second projected line is intersected with each of the edges in the bin, and those points are added to the intersection polygon as well. Recall that each two dimensional intersection polygon on the second camera's image plane represents a region in world space on the plane formed by back-projecting the first camera's active silhouette edge. To calculate that world space region, the points which define the intersection region on the second camera's image plane are back-projected to rays which can then be intersected with the the relevant plane. A plane can be defined by three points. In this case, the three points which constitute the plane are the first camera's origin and the two points on its image plane which define the active edge. For example, the plane might be defined as shown in pseudo-code below:

```
// First point defining plane
planePoint1 = camera1.origin;

// Second point defining plane
tempPoint = (silhouette[edge].point1.x, silhouette[edge].point1.y, camera1.focalLength);
planePoint2 = (camera1.rotationMatrix * tempPoint) + camera1.origin;
```

```
// Third point defining plane.
tempPoint = (silhouette[edge].point2.x, silhouette[edge].point2.y, camera1.focalLength);
planePoint3 = (camera1.rotationMatrix * tempPoint) + camera1.origin;
```

Likewise, each of the rays that are to be intersected with the plane can be defined as passing through the second camera's origin and the corresponding intersection point on the image plane. Note that in both the case of creating a plane or creating a ray, we are using the focal length of the camera as the z-value for the image point in world space. This is appropriate because the units in this situation are irrelevant – it is the ratio that's important, and our focal length is specified in pixels, as are the edge and intersection coordinates. Each of the rays can be formed as such:

```
// First point defining ray
rayPoint1 = camera2.origin;

// Second point defining ray
tempPoint = (intersectionPt.x, intersectionPt.y, camera2.focalLength);
rayPoint2 = (camera2.rotationMatrix * tempPoint) + camera2.origin;
```

It is important to realize that when using image plane coordinates as world space values, such as we do above, they need to first be adjusted so that the coordinate (0,0) is at the center of the image plane. To take this into account, simply subtract the principal point from the intersection coordinate before performing the world space conversion.

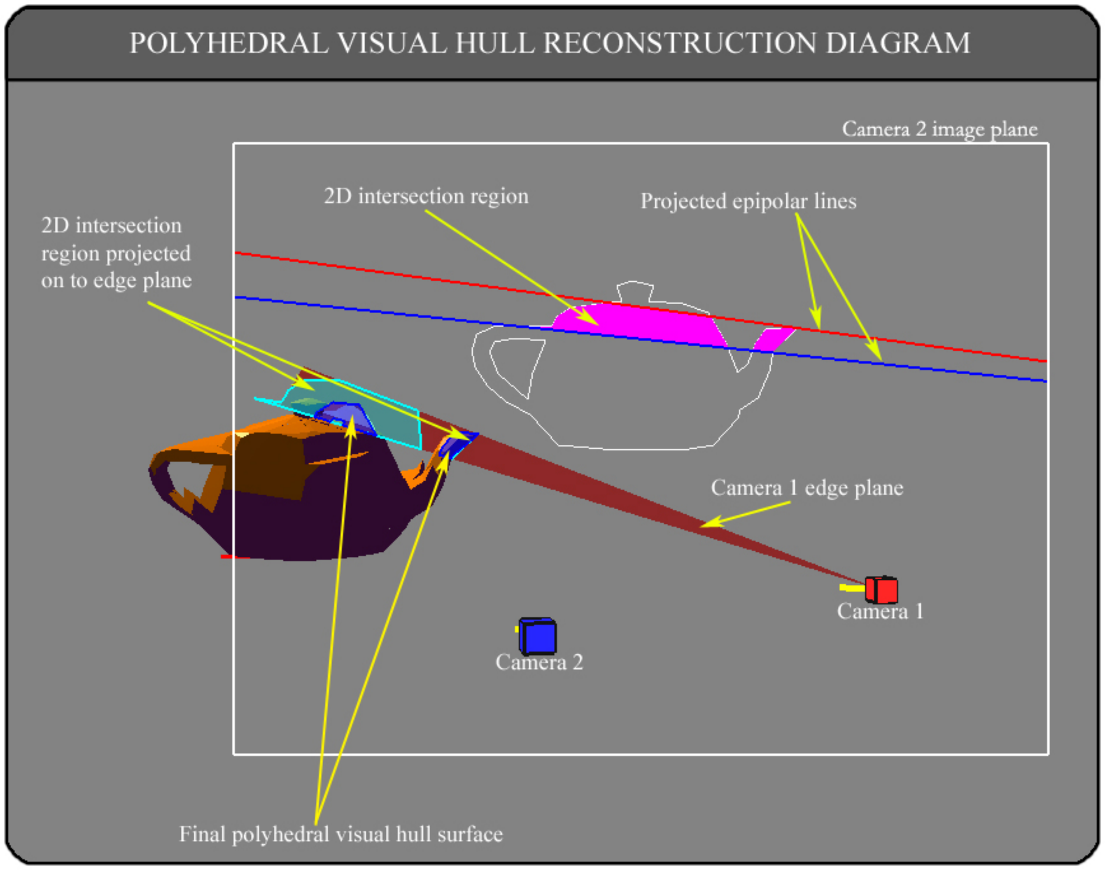

POLYHEDRAL VISUAL HULL RECONSTRUCTION DIAGRAM
Camera 2 image plane
2D intersection region
Projected epipolar lines
2D intersection region projected on to edge plane
Camera 1 edge plane
Camera 1
Camera 2
Final polyhedral visual hull surface

Figure 4.8: *The figure above shows two of the four cameras used to reconstruct the teapot model. An edge in Camera 1's silhouette is being projected on to Camera 2's image plane. The regions of intersection are shown in pink. On the left, the intersected regions have been back-projected on to Camera 1's edge plane (light blue). These regions are then intersected with the regions from each of the other reference cameras (not shown) to produce the final polyhedral hull surfaces (dark blue).*

4.1.6 Polygon Intersection

After each edge from each silhouette has been projected on to the image plane of all of the other cameras, and the 2D regions of intersection have been reprojected on to the edge planes in world space, what remains is a collection of polygons, of size one less than the number of cameras, associated with each of the edges in each of the silhouettes. The intersection of each of these sets of polygons defines a single surface on the object's polyhedral visual hull. To find the intersection of these polygon sets, we use the General Polygon Clipping (GPC) library [1]. This library can take two arbitrary polygons as input and return the polygon that defines their region of intersection. To handle the case of intersecting more than two polygons, we intersect the first two, and then use the resultant as one of the input polygons for a subsequent iteration. The library also contains routines for converting polygons to triangle strips, which is how we have chosen to render the final mesh.

4.1.7 Watertight Meshing

In the original polyhedral visual hull algorithm as presented by Matusik [MBM01], the generated meshes are not guaranteed to be free of holes, or "watertight". Similarly, in our implementation, the polygon intersection is done on a per face basis, and as a result, there is no connectivity information maintained between adjoining surfaces on the visual hull. Due to inaccuracies in the camera calibration process and the inherent numerical imprecision of the polygon intersection routine, vertices which define the coincident edges of adjacent triangle strips are not always colin-

[1]The GPC library is an open source intersection library that can be downloaded at the URL: http://www.cs.man.ac.uk/aig/staff/alan/software/gpc.html

ear. As a result, holes arise in the object's suface where the polygon boundaries are not perfectly aligned. When compositing the captured geometry with a synthetic background, these holes on the model surface become highly disconcerting visual artifacts that allow the viewer to see the background through the object and detract from the image's realism. In order to counter these effects, we attempted to implement Franco and Boyer's "Exact Polyhedral Visual Hull" algorithm [FB03], which uses knowledge of local edge orientation to traverse the viewing edges and build an exact hull which is both manifold and watertight. Unfortunately this approach was problematic, as it proved to be highly susceptible to numerical imprecision when computing line-line and line-plane intersections. We were unable to implement a robust version of the algorithm, and were forced to consider other alternatives.

The polyhedral visual hull is composed of the set of polygons formed by back-projecting the contour edges in each image plane and intersecting them with the other silhouettes. Thus two neighboring surfaces on the visual hull share the same world space line, called a viewing line, formed by back-projecting the shared contour vertex on the original image plane. In order to reduce the artifacts that arise from our non-watertight mesh construction process, we threshold the final polygon vertices based on their distance from this viewing line. If a polygon vertex is found to be within a specified distance from the viewing line, then the point is moved tangentially such that it lays on the line. This ensures that neighboring faces of the final hull share the same boundary edges, without adding much overhead in the form of computational complexity. The viewing lines have previously been computed and stored, and the point-line distance calculation can be performed relatively cheaply. To further reduce the complexity, we compute the distance

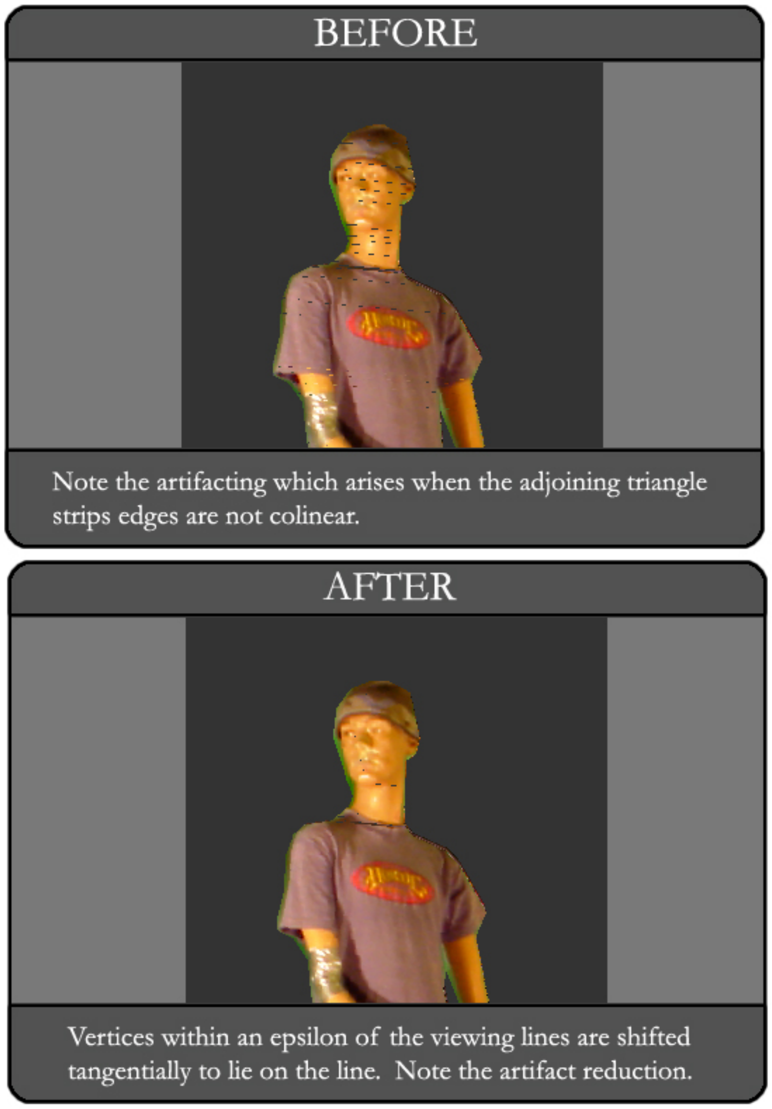

BEFORE
Note the artifacting which arises when the adjoining triangle strips edges are not colinear.
AFTER
Vertices within an epsilon of the viewing lines are shifted tangentially to lie on the line. Note the artifact reduction.

Figure 4.9: *The figure above illustrates the image quality that can be expected before and after our vertex thresholding operation. After the thresholding has been performed, the number of visible artifacts has been drastically reduced.*

squared as opposed to the actual distance. This eliminates the need for taking a square root. The result of our thresholding operation can be seen in Figure 4.9. Although this method does not completely eliminate the problem, it does drastically reduce the artifacts. The downside to this approach is that fine geometry has the potential to be collapsed to a single point if the thresholding distance is not properly set. However, for a given camera configuration, with known scene dimensions and camera resolution, the thresholding value can easily be adjusted such that it does not erode the geometry. Figure 4.10 shows the reconstructed hull of a person standing with one foot on a ball.

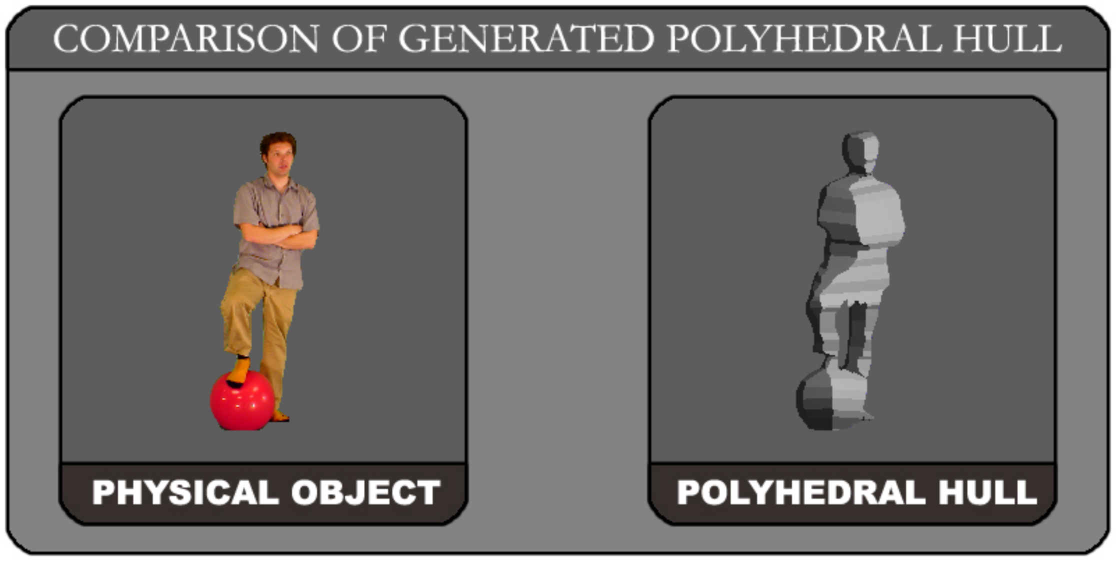


Figure 4.10: *The figure above shows a side-by-side comparison of a person standing with one foot on a ball and their reconstructed polyhedral visual hull.*

Chapter 5

Shadow Effects

Our system was designed with the intent of merging physical objects with computer generated imagery. We want this merging process to include realistic effects, such as shadows and environment inter-reflections, both of which are important for believability. Our two primary criteria are that the reconstruction and compositing processes happen quickly, to facilitate an interactive real-time application, and that the final output images are visually plausible. In this chapter we present a novel approach to shadow generation that meets the unique demands of our compositing environment while still abiding by the aforementioned principles. In keeping with our goal of performance, the algorithm is designed to be executed on graphics hardware, and our implementation only handles a single light source. To ensure a high degree of realism, we use the generated hull to cast believable shadows. We have opted for the production of soft shadows, acknowledging the fact that this approach requires significantly more processing time than hard shadows. Our approach is unique in that it prevents the foreground object from casting shadows on to itself, and in doing so, it also minimizes the processing required to compute the penumbra simulation.

5.1 Unique Shadow Environment

In compositing the captured geometry with a globally illuminated background scene, there are several unique considerations that our system must take into account. A primary consideration is that the captured video frames are subsequently used to texture the reconstructed hull during the image generation process. These textures were captured in a studio environment and inherently have effects of the studio lighting contained within the image capture of the foreground's surface reflections. The obvious difficulty which arises is that if the foreground object is placed in a completely different lighting environment, the shading will not match. One solution would be to attempt to remove these illumination effects. In order to accomplish this feat, one would first need to determine both the lighting in the scene and the material properties of the foreground object. Then a computationally complex inverse global illumination algorithm would need to be run on the video textures. Finally, the textures would have to be re-lit according to the the new lighting environment. Because this does not lend itself well to an interactive application, we have instead chosen to assume that there will be matched lighting between the studio capture environment and the virtual background scene. Enforcing this consistency ensures that the video textures already contain similar lighting for their new environment. A result of this observation is that when developing a shadowing algorithm for our system, we had to ensure that the foreground object did not cast shadows onto itself a second time. The textures already include self-shadowing effects, and thus if we were to compute them again, we would compound the shadows and make them darker than they should be.

We do, however, want the captured geometry to cast shadows onto the scene in which it is being placed. These shadows are extremely critical in making clear the spatial relationships that exist between the foreground and background, and how the lighting in the scene is positioned relative to both. It is also important to reproduce the shadows that the scene would cast onto the inserted geometry. We do not need to compute the shadows that the scene casts onto itself, as this information is native to the background image that the RTGI system has generated.

Another constraint that is unique to our particular system is the fact that we are lacking mesh connectivity information for our generated model. The reconstructed hull is not a closed manifold, and not every edge in the mesh is shared by exactly two triangles. Without this underlying structure, the process of determining the shadow boundaries is a more complicated operation. Traditionally when silhouette boundaries are required, for example when generating shadow volume geometry, it is efficient to traverse the light/dark boundaries of the model and record those edges as the shadow casting edges. However, this optimization is not possible in our case. The mesh could potentially be cleaned up with a post-processing routine, but the extra computation required to make these adjustments is prohibitive in an interactive application. Instead we developed a novel approach for robustly computing the outer shadow boundaries of an object as seen by the light source. The technique does not capture internal shadow boundaries, and thus is well suited for our application where we want to avoid self-shadowing. In the following sections we will discuss our shadow-boundary detection algorithm, and how we interface this with the Penumbra Map work done at the University of Utah [WH03].

5.2 Shadow Mapping

Shadow mapping was first introduced as a flexible shadow-generation algorithm in 1978 by Williams [Wil78]. When using shadow mapping, the scene is first rendered from the point of view of the light, and the depth value at each pixel location is recorded. This depth image is referred to as the "shadow map". In Figure 5.1, we see the shadow map generated for a teapot model.

After the production of the shadow map, the scene is rendered from the viewpoint of the virtual camera. As each fragment is rendered, its virtual camera eye-space location is projected into the light's clip-space. The necessary steps for the space conversion are displayed in Figure 5.2. Here we can see that in order to project the fragment into the light's clip-space, we need to first multiply each eye-space location by the inverse of the camera's view matrix, and then multiply it by the light's view and projection matrices. The depth value of the projected fragment in the light's eye-space is then compared to the depth value stored in the shadow map for that pixel location. If the projected depth value is less than or equal to the value stored in the shadow map, then the fragment is the closest surface to the light along that particular viewing ray. As a result, the fragment is rendered as lit. If, however, the projected fragment is further from the light than the shadow map value, then there must be an occluding surface between the fragment and the light. The fragment is therefore in shadow, and is rendered accordingly. The shadow map depth comparison is illustrated in Figure 5.3. On modern graphics cards, this comparison can be accomplished in hardware as the fragment passes through the graphics pipeline. In OpenGL, this functionality is accessed via vendor-specific extensions. These extensions are described in Section 5.3.2. The shadow mapping algorithm is designed to handle point lights, and

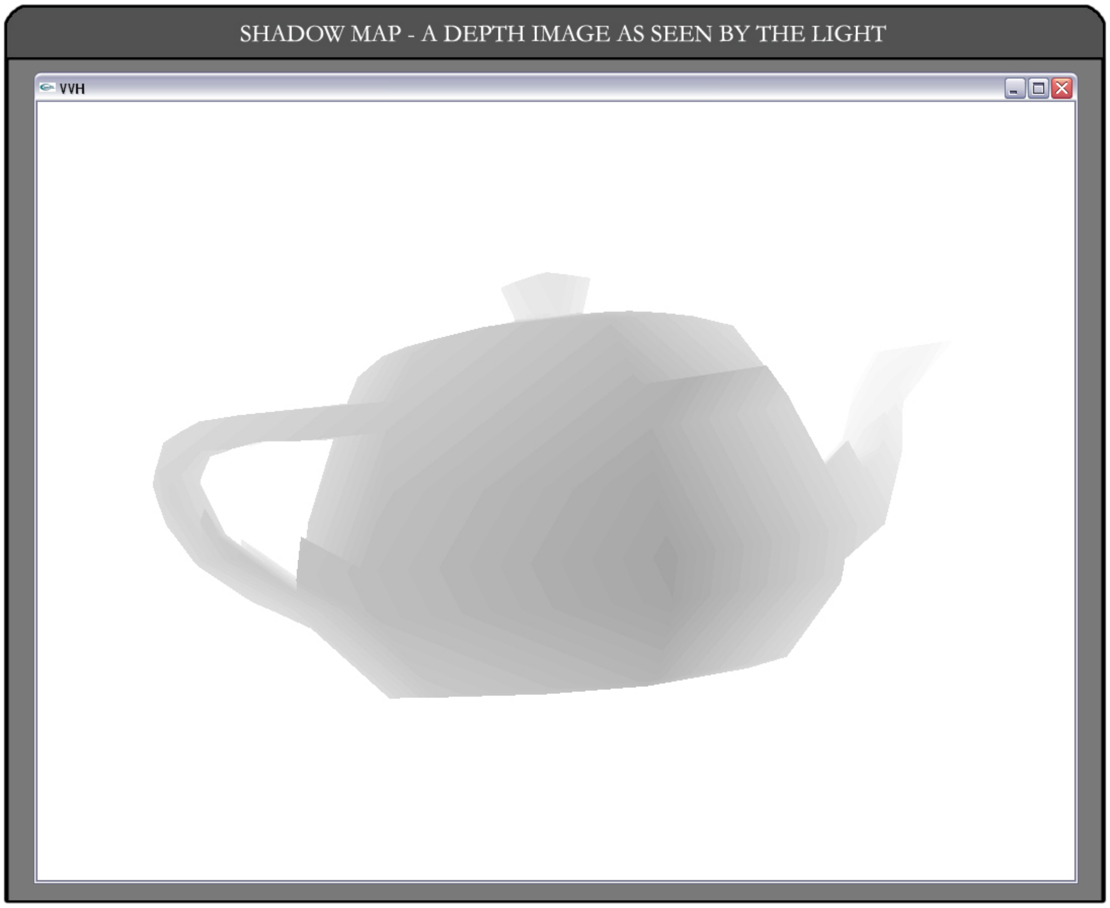


Figure 5.1: *In the figure above, a teapot has been rendered from the point of view of the scene's light. By using intensity values to represent distance, the figure shows the generated depth image (shadow map), which will be used during shadow computation. Darker regions of the image represent smaller depth values, where the object is closer to the camera. Conversely, lighter areas indicate larger depth values where the object is a greater distance from the camera.*

as a result, only hard shadows can be generated. In our system, we require that the foreground object is able to cast soft shadows onto the surrounding environment, so we use a variant of shadow mapping, called penumbra mapping.

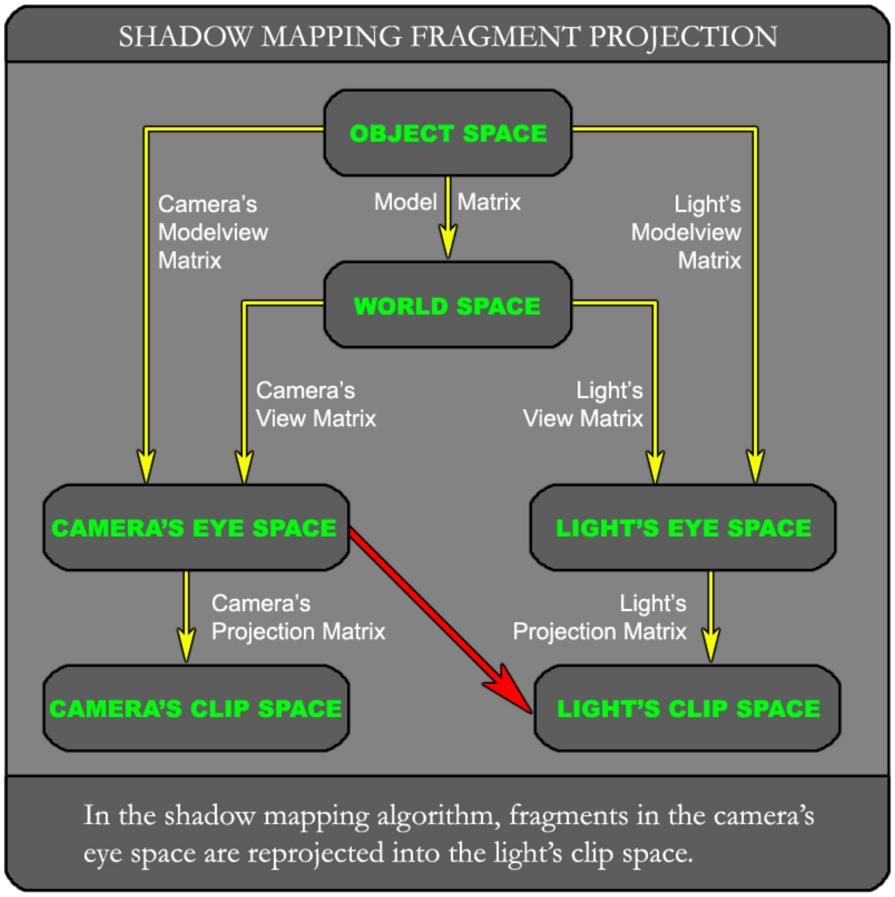

SHADOW MAPPING FRAGMENT PROJECTION
OBJECT SPACE
Camera's Modelview Matrix
Model Matrix
Light's Modelview Matrix
WORLD SPACE
Camera's View Matrix
Light's View Matrix
CAMERA'S EYE SPACE
LIGHT'S EYE SPACE
Camera's Projection Matrix
Light's Projection Matrix
CAMERA'S CLIP SPACE
LIGHT'S CLIP SPACE
In the shadow mapping algorithm, fragments in the camera's eye space are reprojected into the light's clip space.

Figure 5.2: *The red arrow in the figure above represents the reprojection of pixels in the camera's eye space to the light's clip space. This transformation is used during rendering to compare fragment depths to those stored in the shadow map.*

5.3 Penumbra Mapping

In order to generate visually plausible soft shadows at interactive frame rates, our system makes use of the penumbra map algorithm [WH03]. Penumbra mapping, a

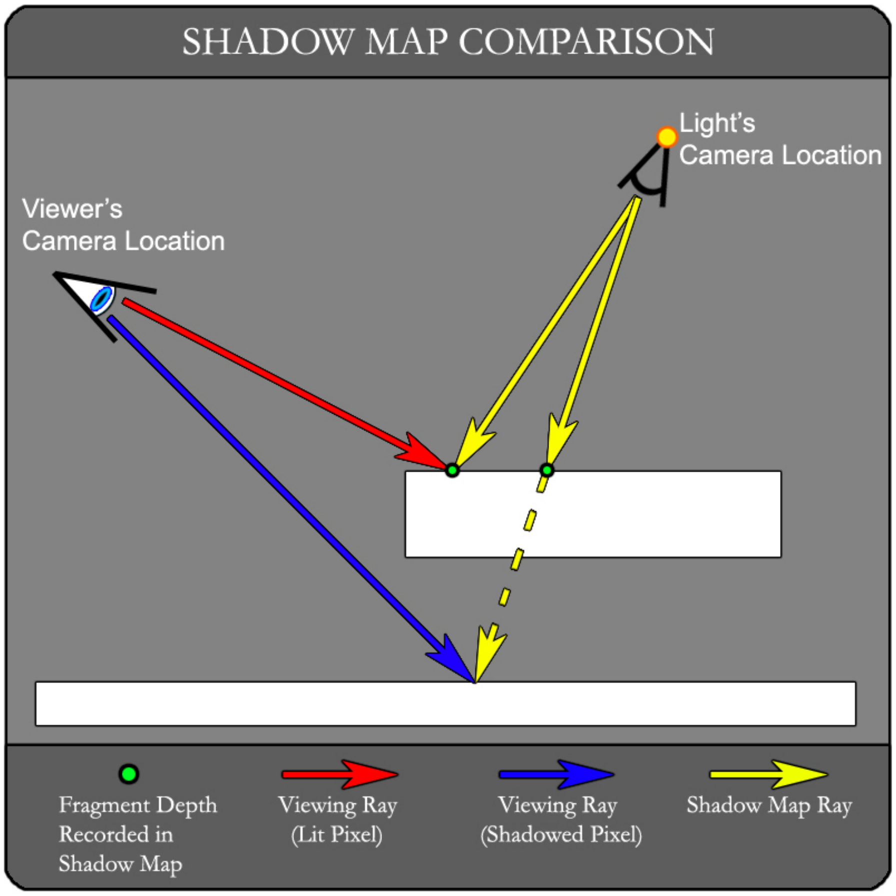


Figure 5.3: *The figure above shows two viewing rays. The fragment generated by the first ray (red), is the same surface location stored in the shadow map. Therefore, the pixel is lit. The second viewing ray (blue) hits a surface that is further from the light than the fragment stored in the shadow map. As a result, this pixel is in shadow.*

method dependent upon shadow mapping, allows for the fast calculation of believable shadows cast from area light sources. The algorithm relies on the principal fact that when treating an area light like a point light, the entire penumbra region cast by an occluder can be seen from the light's point of view. Therefore a shadow map can be used for calculating those portions of the environment that are in the umbra, and a penumbra map can be used to determine penumbral intensity at a given surface location. A penumbra map is generated by constructing a set of geometries around the shadow-casting edges in a scene, and rendering those geometries into a buffer as seen by the light source. Figures 5.4 and 5.5 illustrate the construction of the penumbra map cone and sheet geometry. This process will be described in detail in subsequent paragraphs.

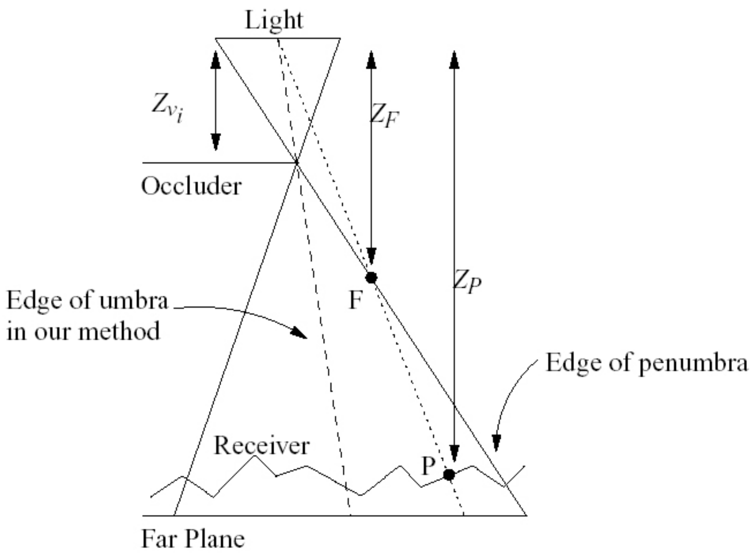


Figure 5.4: *Relevant dimensions used to compute the intensity stored in the penumbra map [WH03]*

The first step in the penumbra map algorithm is to generate a shadow map by rendering the scene from the light's point of view into a depth texture. The shadow map serves a dual purpose, being used in both the computation of the penumbra map as well as during the final rendering pass. For the final rendering, however, we do not want the background environment to be able to cast shadows on to itself. The environment's self shadowing has already been computed by the global illumination algorithms, and consequently we need to avoid compounding the shadowed regions by subtracting the illumination twice. To handle this, after we have used the shadow map to generate the penumbra map, we clear the shadow

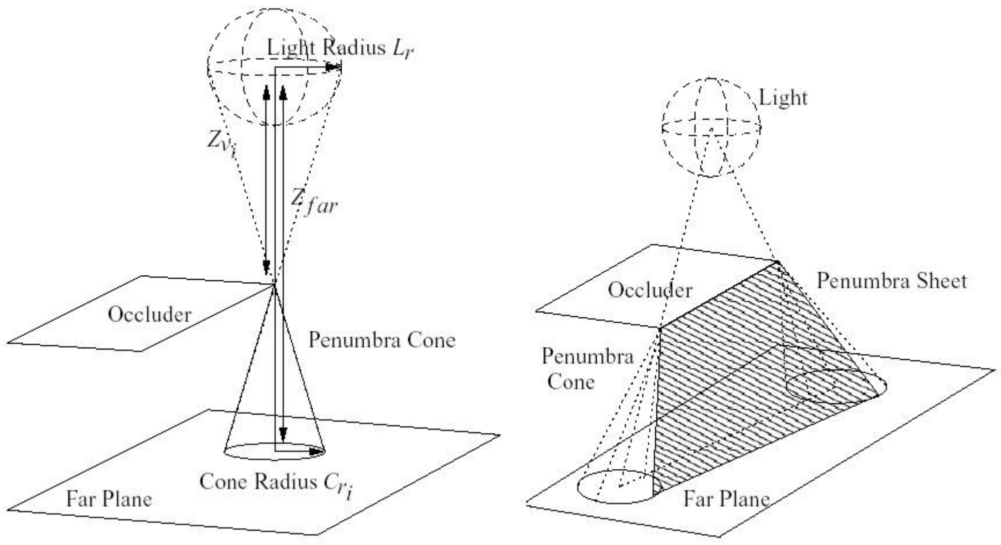


Figure 5.5: *The figure above shows the penumbra map cone and sheet geometry. [WH03]*

map's values to white and then re-render the foreground geometry into it. This prevents objects in the virtual camera's view from projecting to behind the background geometry during the shadow map lookup. Since the shadow map is empty except for the foreground geometry, during the final rendering only objects behind the foreground model will be marked as in umbra.

After the shadow map has been generated, the next step is to determine the contour of the foreground object as seen by the light. This topic will be covered extensively in Section 5.3.3. Once we have the object's contour, we can compute the penumbra map geometry. The penumbra map geometry consists of a cone at each silhouette vertex and a plane along each silhouette edge. The apex of each cone is located at its corresponding silhouette vertex, and the base of the cone is located on the camera's far plane. The radius of the cone base is dependent on both the radius of the light source as well as the ratio of the distance between the light and the silhouette vertex to the distance between the silhouette vertex and the far plane. This mathematical relationship for computing the cone radius can be expressed in the form:

$$C_{r_i} = \frac{(Z_{far} - Z_{v_i})L_r}{Z_{v_i}} \tag{5.1}$$

where Z_{far} is the distance from the light center to the far plane, Z_{v_i} is the distance from the light center to the silhouette vertex, and L_r is the light radius. This relationship can be seen in Figure 5.4. In addition, Figure 5.7 displays example penumbra map geometry as generated by our software for casting shadows from a teapot model. Figure 5.8 shows the geometry in relation to the light's view frustum.

In order to render the cones using standard hardware techniques, the cone geometry is decomposed into a triangle fan. The extent to which the cone geometry is tessellated presents a trade off between the smoothness of the shadows cast and the speed at which the algorithm runs. High tessellation leads to smoother shadow transitions between adjacent planes at the silhouette vertices, however it also results in increased fill rate and thus slower rendering times. Low tessellation has the inverse effect: shadows render quicker, but at the cost of smoothness in the penumbra.

The plane geometry is constructed based on each pair of adjacent cones. The top two vertices for a given plane are located at neighboring cone apexes. Similarly, the bottom two points which define the plane are also based on the cone dimensions. The point on a cone base that is furthest from the light, along the direction vector created by subtracting the light origin from the cone apex, is where a bottom vertex on the plane is defined. These points on two neighboring cone bases provide the

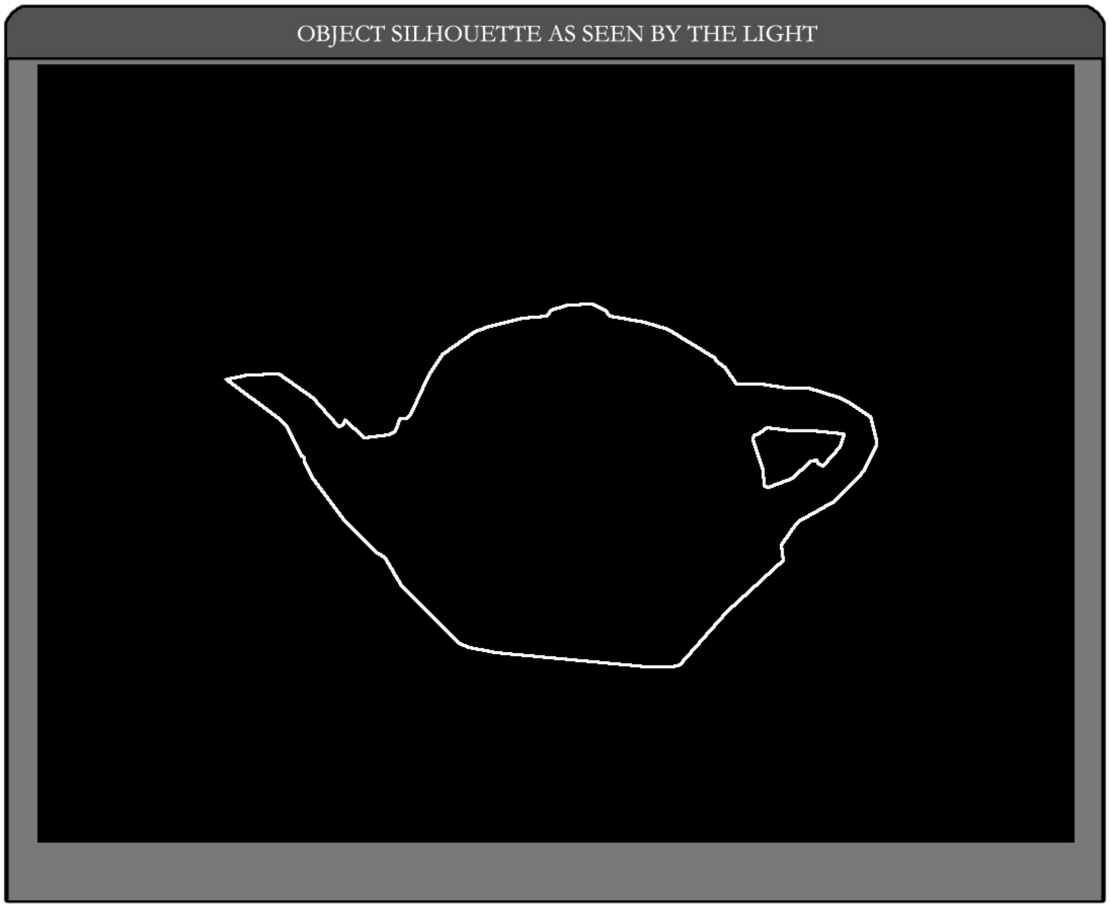


Figure 5.6: *The figure image above shows the outer contour of a teapot as seen by the light.*

bottom edge of the connecting plane structure. See Figure 5.5 for an illustration of this relationship.

5.3.1 Intensity Computation Using Cg

After the plane and sheet geometry has been generated, it is rendered into the penumbra map. The penumbra map stores the penumbra intensity in a scene, as viewed by the light. We implement Cg vertex and fragment shaders which are run on the GPU to perform this intensity calculation. As the cone and sheet

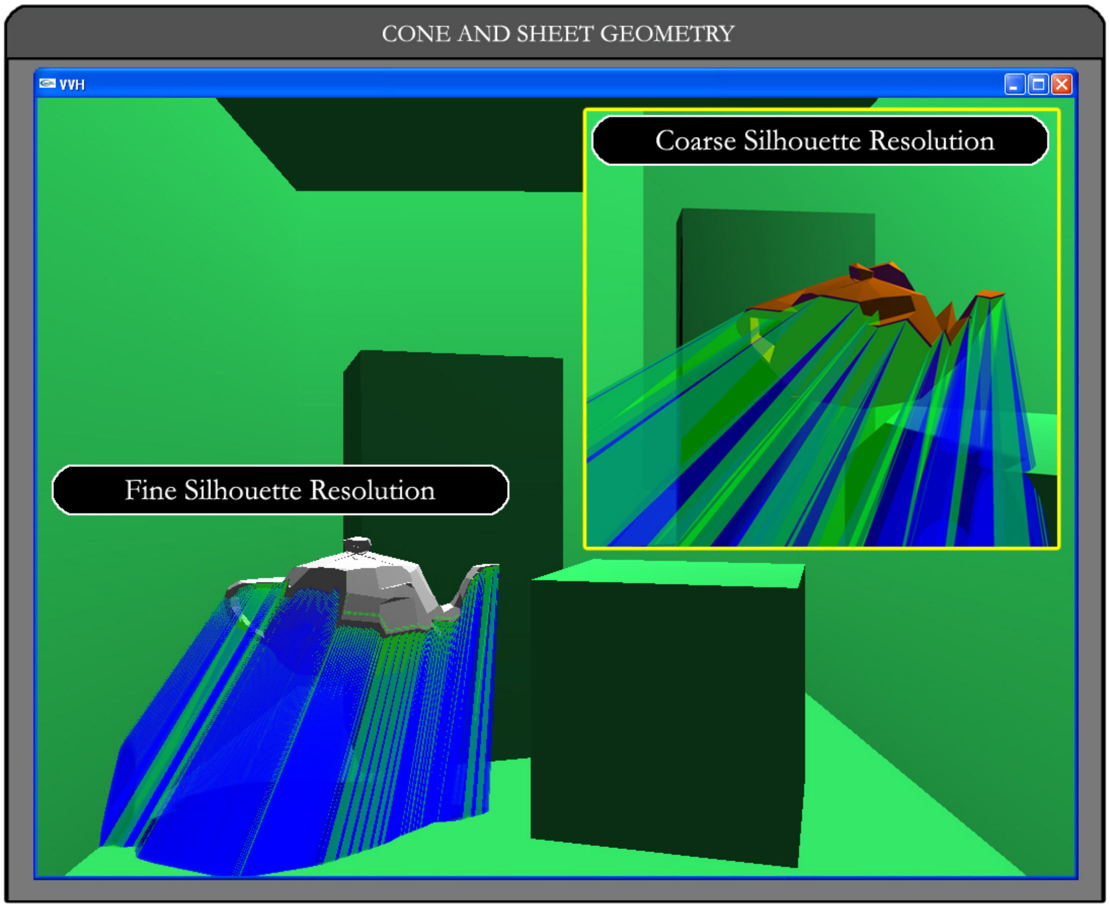


Figure 5.7: *The penumbra map geometry consists of cones at each silhouette vertex (blue) and planes along each silhouette edge (green). The figure above shows an example of both coarse and fine silhouette resolution.*

vertices pass through the graphics pipeline, the vertex shader multiplies each vertex by the model, view and projection matrices to position the vertex in the light's clip space. It also passes along the distance from the light center to the cone apex. The fragment shader is granted access to the shadow map, the screen width and height, and the inverted texture matrix. When a fragment is received, the shader calculates the penumbra intensity at that surface point. The equation for

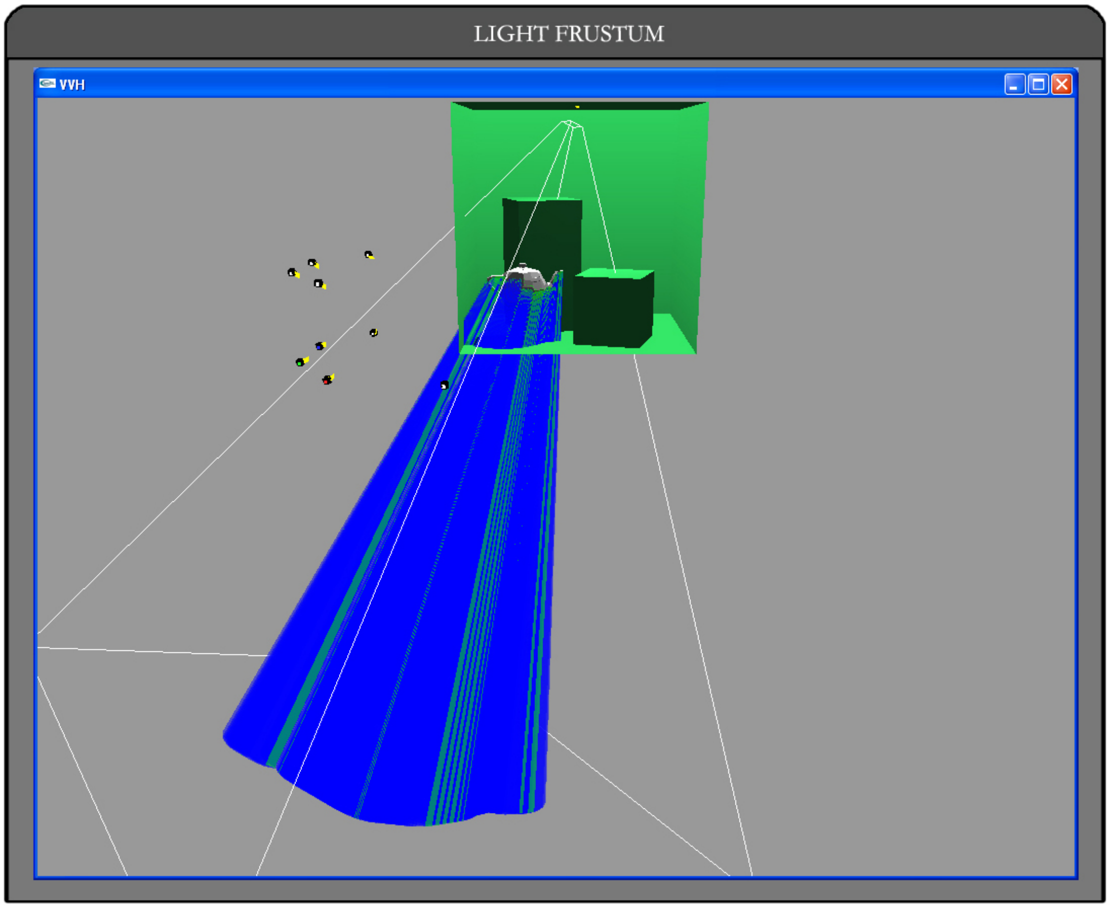


Figure 5.8: *In the figure above, the penumbra geometry is shown to extend from the foreground object contour to the far clipping plane.*

computing the intensity is expressed as follows:

$$Intensity = \frac{Z_F - Z_{v_i}}{Z_P - Z_{v_i}} \tag{5.2}$$

where Z_F is the distance from the light center to the current fragment being processed, Z_{v_i} is the distance from the light center to the silhouette vertex, and Z_P is the distance from the light to the received surface. This relationship can be seen in Figure 5.4. To generate Z_F, the fragment shader takes the incoming fragment depth (in the range [0,1]) and converts it back to the light's eye space by multiplying it by the inverted texture matrix. To compute Z_P, the depth of the receiver surface, a lookup is performed on the shadow map texture, and then the returned value is similarly multiplied by the inverse texture matrix to generate the depth value in the light's eye space. Once Z_F and Z_P have been calculated, it is trivial to compute the intensity value. The generated intensity value is a linear approximation of the shadow intensity between the umbra and fully lit regions of the scene. In order to approximate a sinusoidal falloff, as is standard for a spherical light source, the linear intensity value is run through the Bernstein interpolant:

$$s = 3\tau^2 - 2\tau^3 \tag{5.3}$$

After the sinusoidal intensity value is computed, the Cg program stores the fragment intensity value in the penumbra map texture for use during the final scene rendering. Figure 5.9 displays an example penumbra map.

5.3.2 Use of OpenGL Extensions

Many of the operations related to shadow generation have been implemented in hardware on modern graphics boards, primarily due to their recent popularity in video games. When available, it is typically advantageous to make use of these hardware features, often implemented as extensions to the standard API, as they offer increased speed and efficiency over the software alternative. In some instances, they even overcome severe limitations of trying to perform the operations manually in software. In our system we rely upon several OpenGL extensions to facilitate real-time performance.

In order to perform off-screen rendering, we make use of pixel buffers, or pbuffers, on the graphics card. Pbuffers are additional non-visible rendering buffers for an OpenGL renderer. Using pbuffers has several key advantages. First, it permits images to be rendered at a resolution higher than that of the frame buffer

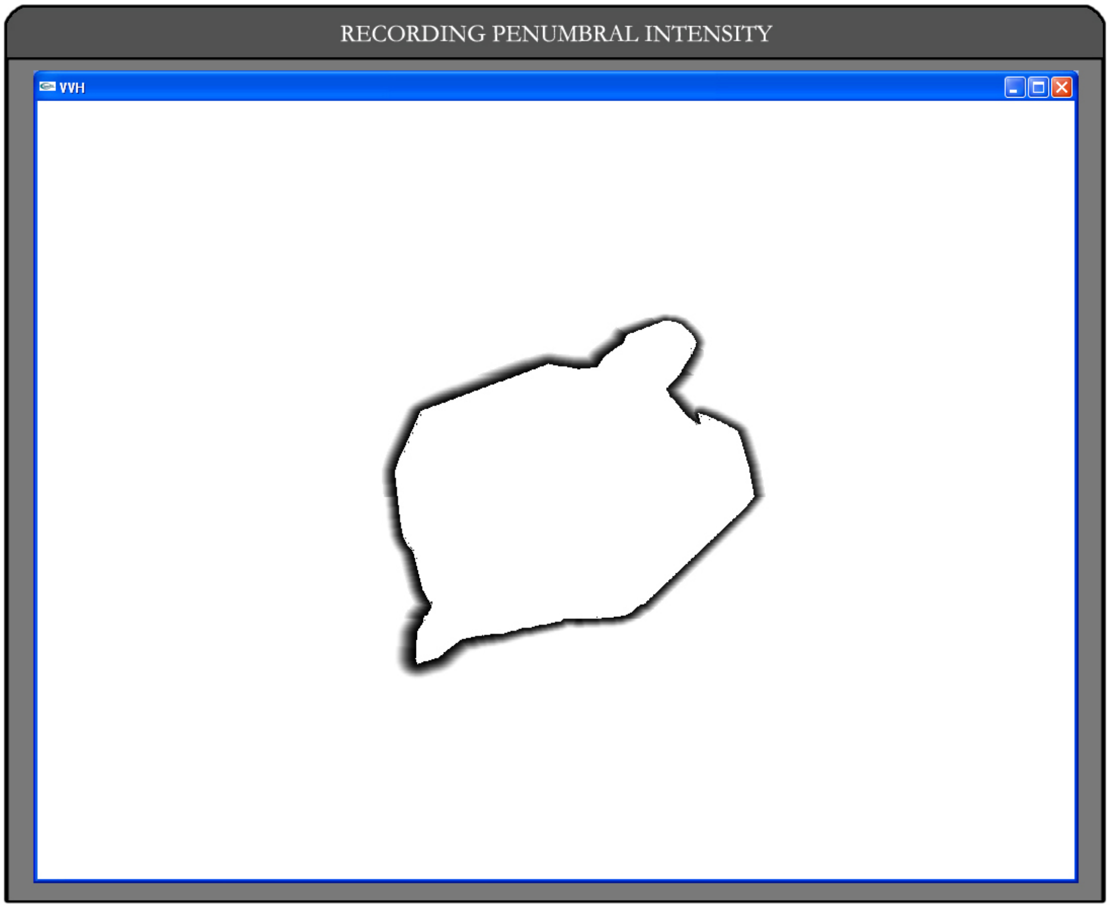


Figure 5.9: *The figure above displays a sample penumbra map. A penumbra map, which displays the shadow intensity as seen by the light, is white outside of the penumbra region and black at the contact points.*

(and potentially even higher than the monitor will support), and secondly, it allows one to render without having to worry about the pixel ownership test. In OpenGL, if a window is not the foremost window on the screen, then that window does not own the pixels inside of it, and rendering to those pixels is undefined. This is referred to as the pixel ownership test, and it can lead to problems when an occluded portion of a window has the potential to affect the rest of the screen. Such is the case during shadow generation, where a partially occluded window will lead to an incomplete shadow map. The missing shadow map regions can later lead to artifacts in the un-occluded portions of the screen, as pixels outside the occluded area may well project to the missing locations during the shadow map lookup. Pbuffers are also beneficial because they eliminate the need to render everything into the frame buffer first and then perform an expensive read-back to store the data in a different location. The extension that we use to implement our pbuffers is "WGL_ARB_pbuffer".

We also use "WGL_ARB_render_texture", OpenGL's render to texture extension, to render the shadow map and penumbra map directly to a texture. This eliminates the overhead associated with copying the image from either the frame buffer or a pixel buffer to a texture. In order to create a texture that can store depth values, as is necessary for the shadow map, we use the depth texture extension, "GL_SGIX_depth_texture". Using these extensions in conjunction with each other, it becomes very easy to quickly render the scene into a depth texture which can later be indexed during the shadow map lookup.

5.3.3 Computing Light/Dark Polygon Boundaries

In many shadow algorithms, including shadow volumes and penumbra mapping, it is necessary to identify the boundaries between light and dark facing polygons. If connectivity information is known, this can be achieved by iterating through a mesh and comparing the visibility of neighboring surfaces. However, in our system, the polyhedral visual hull is comprised of a set of disparate surfaces, each of which is stored as a series of triangle strips for fast hardware rendering. There is essentially no connectivity information known, and consequently we need to use an alternative method.

We have chosen to render the reconstructed object from the point of view of the light, and then run a contour finding algorithm on the image. The silhouette of the object as seen from the light defines the global shadow boundary that the object casts on to its surrounding environment. This technique will not capture self shadowing of the object; however, this is not an effect we desire, as the video frame textures inherently have self shadowing effects built in. If we were to capture these local shadows with our shadowing algorithm, then the shadowed regions would become too dark, as they would be compensated for twice. The effects of self shadowing are shown in Figure 5.10. Another benefit of only using the object's outer contour boundaries is that the minimum set of penumbra geometries is created, and this reduces the computation time for generating the penumbra map.

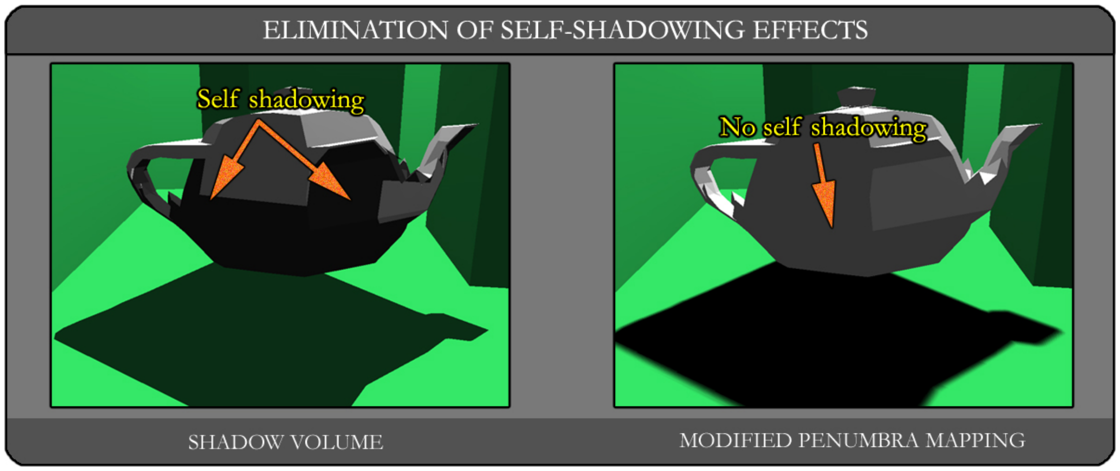

ELIMINATION OF SELF-SHADOWING EFFECTS
Self shadowing
No self shadowing
SHADOW VOLUME
MODIFIED PENUMBRA MAPPING

Figure 5.10: *The figure above compares the output of an algorithm that generates self-shadowing effects and our approach, which does not.*

Our technique is similar to that proposed by McCool [McC00]. He renders an object into the depth buffer and then uses the depth map as input to an edge detection algorithm. By finding the edges in the depth image, he is able to recover

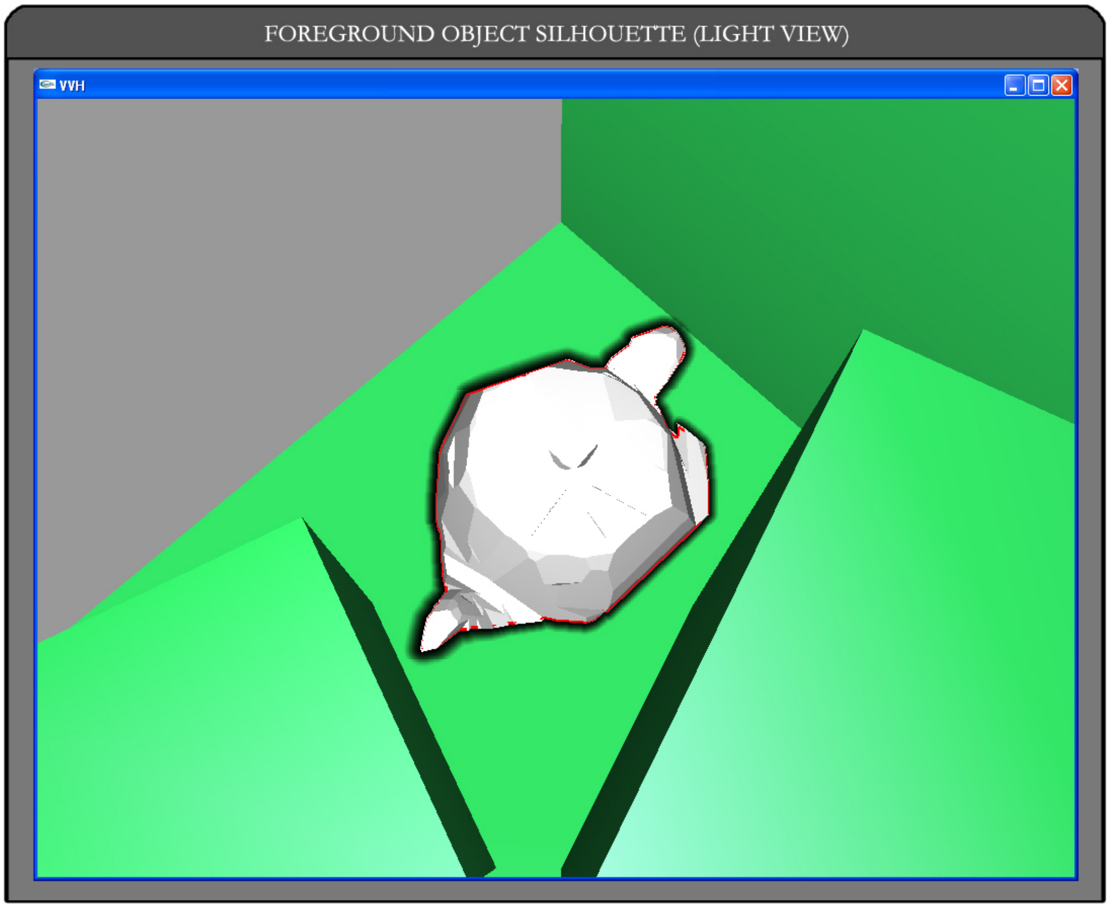


Figure 5.11: *Above, the silhouette of the foreground object (red) is shown as it is seen by the light.*

both the internal and external light/dark boundaries. This is required for his algorithm, as he is concerned with self shadowing. Because we are only interested in computing the extreme object boundaries, we perform the edge detection routine on the stencil buffer as opposed to the depth buffer. The stencil buffer is like the color or depth buffer except that the stencil pixels have application-specific meanings. In our system, the entire stencil buffer is initially set to zero. Subsequently, each time a fragment is written to the frame buffer, we increment the value in the stencil buffer at that pixel location. In this way, after rendering the foreground

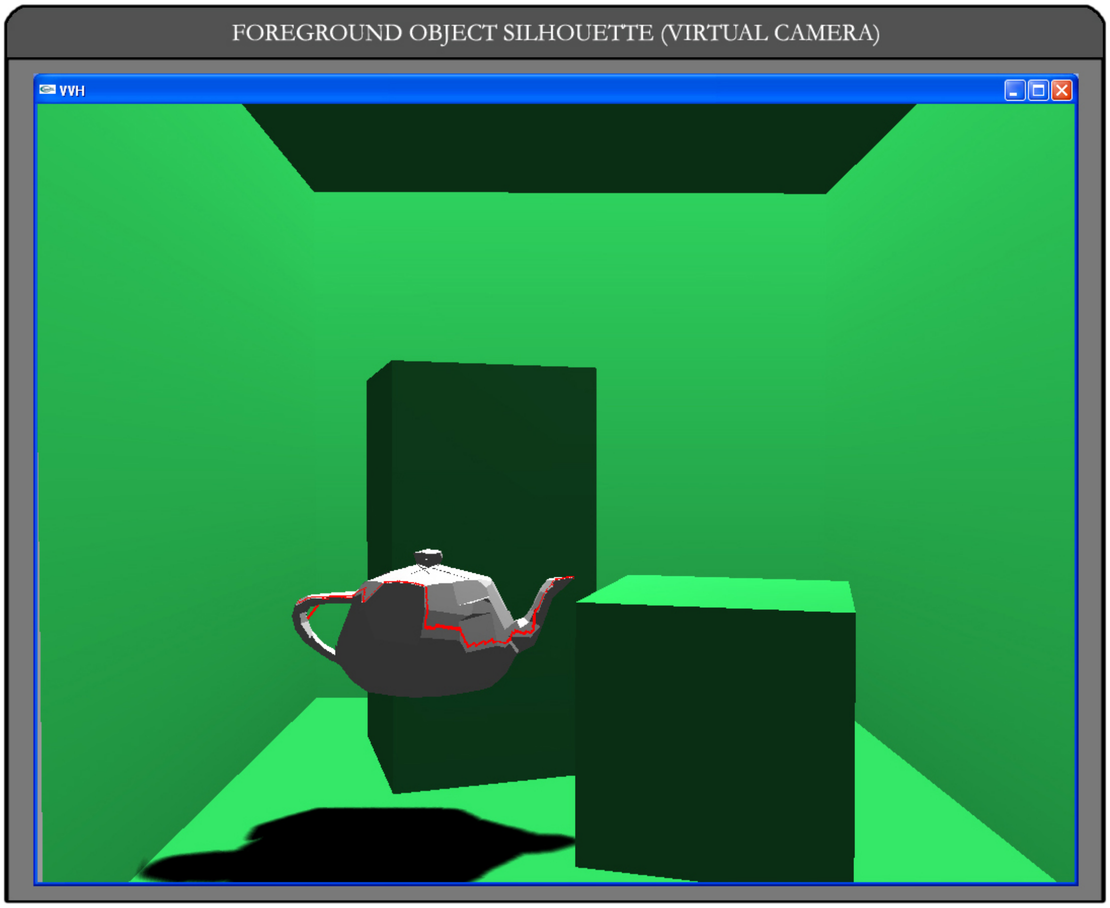


Figure 5.12: *Above is displayed the object's silhouette contour (red) as seen by the virtual camera.*

object, the stencil buffer has a value greater than zero everywhere the object is present. There are two primary advantages to working with the stencil buffer over the depth buffer. First, it is faster to read back the stencil buffer than the depth buffer, as the stencil buffer stores one byte per pixel, and the depth buffer stores four. In addition, it is easier to find the contours in a binary image than in a floating point depth image. By thresholding the stencil buffer, we are left with a binary image that can quickly be parsed by OpenCV's contour finding routine.

When determining the contours of the object as seen by the light, there is a trade off to be made with the contour's granularity. With a fine resolution, the number of edge segments will be higher, and the cone and sheet geometry will become increasingly complex. On the other hand, a coarse resolution will result in fewer edge segments but those segments will begin to deviate from the object's true edges. If used in conjunction with a shadow volume algorithm, this disparity would not be a large issue. The only consequence would be that the shadow is either slightly larger or slightly smaller than the actual object, depending on how the contour was approximated. This would most likely only be perceivable near contact points or where the object is in close proximity to the surface being shadowed. However, when paired with the penumbra map approach, visually displeasing artifacts can arise. In the penumbra map algorithm, a pixel is first projected into the shadow map to determine if it is the foremost surface as seen by the light at that location. If this is not the case, and the pixel projects to a depth value larger than that stored in the shadow map, then the pixel is set to black in the image, thus indicating it is completely shadowed. If the pixel is not in complete shadow, then a lookup operation is performed on the penumbra map to determine the pixel's relative intensity. The artifacts surface where there is a gap between the actual edge of the object and the edge retrieved during the contour finding operation. As seen from the point of view of the light, any pixel that lies between the object's computed contour and its actual contour will result in a bright spot in the shadow. This is because the pixel is outside the bounds of the object, and thus is not marked as in umbra during the shadow map lookup. Then, when the penumbra map lookup occurs, the pixel is inside the extent of the contour, and thus will not be covered by the cone and sheet geometry built around the silhouette edge. The result is the occasional pixel that is completely lit surrounded by pixels that are in shadow. See Figure 5.13 for an example.

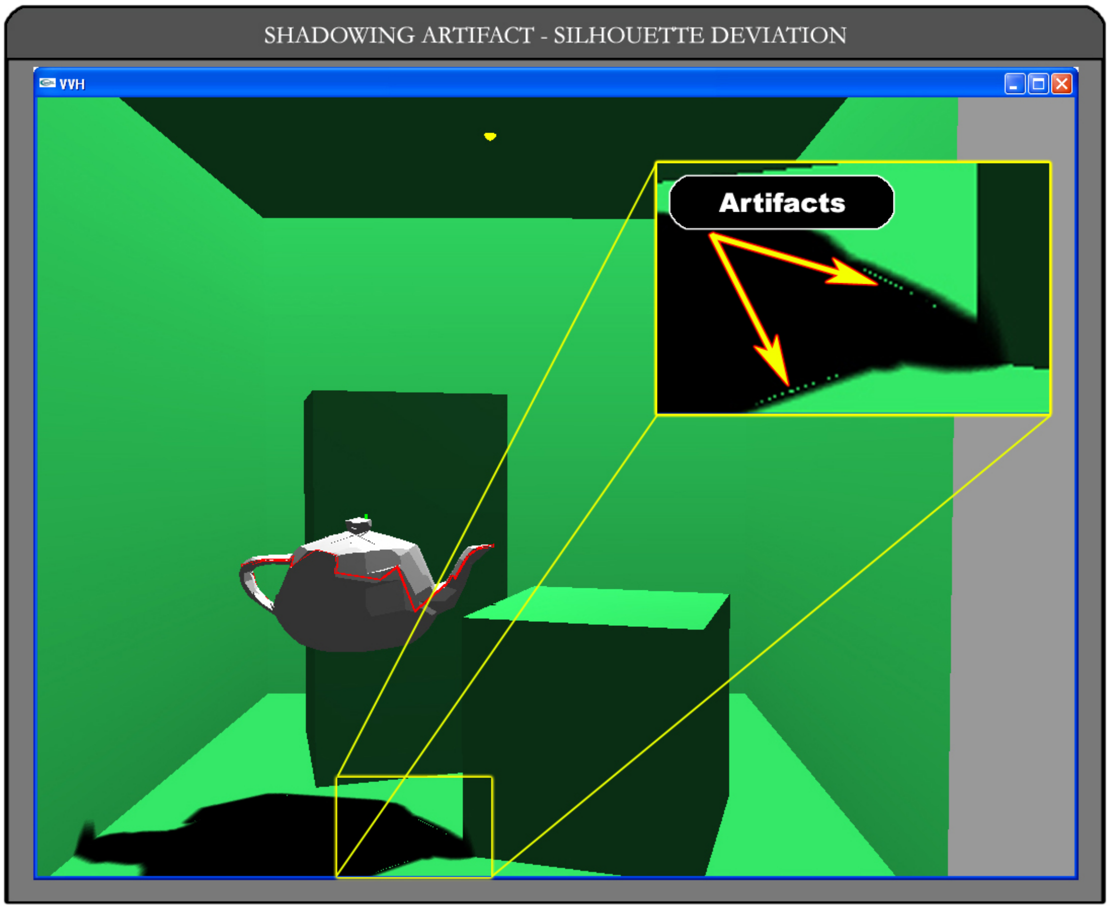


Figure 5.13: *Artifacts arise in the shadow when the silhouette extends outside the object. The red line on the teapot indicates the silhouette boundary that was used for the generation of the penumbra map geometry.*

To eliminate the disparities between the actual silhouette and the calculated silhouette, we devised a method for shrinking the borders of the object just slightly. This has the effect that when the object's silhouette is found, it is just inside the object. Because the silhouette is contained within the object, the case where pixels fall outside the object but inside the silhouette is eliminated. This removes the artifact whereby fully lit pixels exist in the shadowed region. The shrinking of the object has to happen such that its borders push away from the rest of the environment. For example, while the exterior boundary needs to contract slightly, holes in the object actually have to expand by a small amount so that the boundary is a bit inside the actual object silhouette. This effect cannot easily be achieved in object space, so we perform an image space operation instead.

First the object is rendered from the light's point of view, with the stencil test enabled. The stencil test is configured such that the stencil buffer is incremented each time a pixel is drawn to that location. In this manner, after the object has been rendered, the stencil buffer has the value zero where the object is not present, and some value greater than zero where the object is present. This portrays where the object is seen in space by the light. The stencil buffer is read back and stored in an array. The next step would be to run an edge detection algorithm on the contents of the stencil buffer; however, first we perform our object-space silhouette reduction routine. This function iterates over every pixel in the image and determines if it is a silhouette border pixel. The criteria to be a border pixel is that it must be non-zero, and consequently not part of the background, and it must have a zero/background pixel as a neighbor. All of the border pixels are set to zero, or marked as background. This effectively shrinks the silhouette of the object by a width of one pixel all the way around. As a result, when we run the contour finding algorithm, and use the returned silhouette to generate penumbra map shadows, the shadows look natural and do not exhibit artifacts.

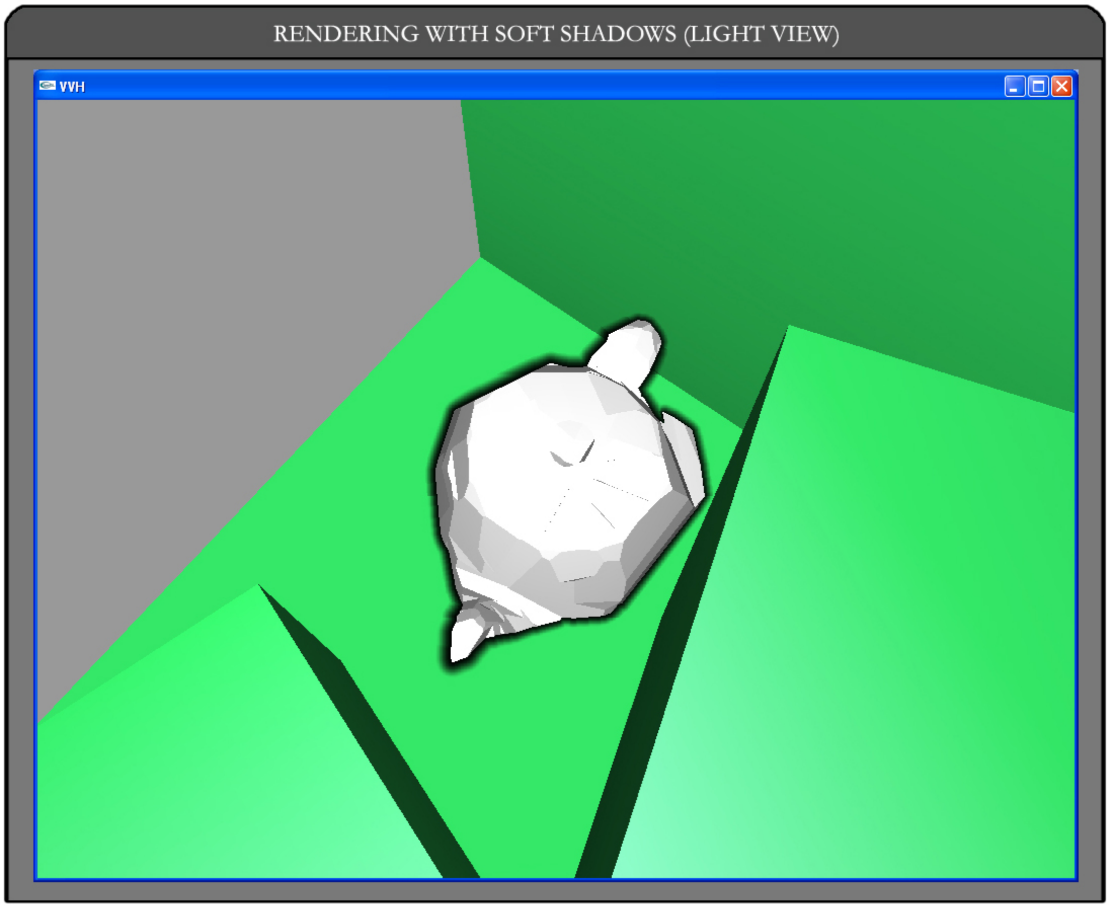


Figure 5.14: *The image above displays a teapot casting a soft shadow onto the background environment. The camera was aligned with the light's viewpoint for this rendering.*

5.4 Casting Shadows on the Foreground Object

There are several different methods that could be used to cast shadows on to the foreground geometry from the background scene. One approach that we experimented with involved using the RTGI system to perform direct lighting calculations on the foreground pixels. This method was developed before we had access to the scene geometry in our software, and we were relying upon the RTGI system to provide us with a depth image of the background environment. The algorithm

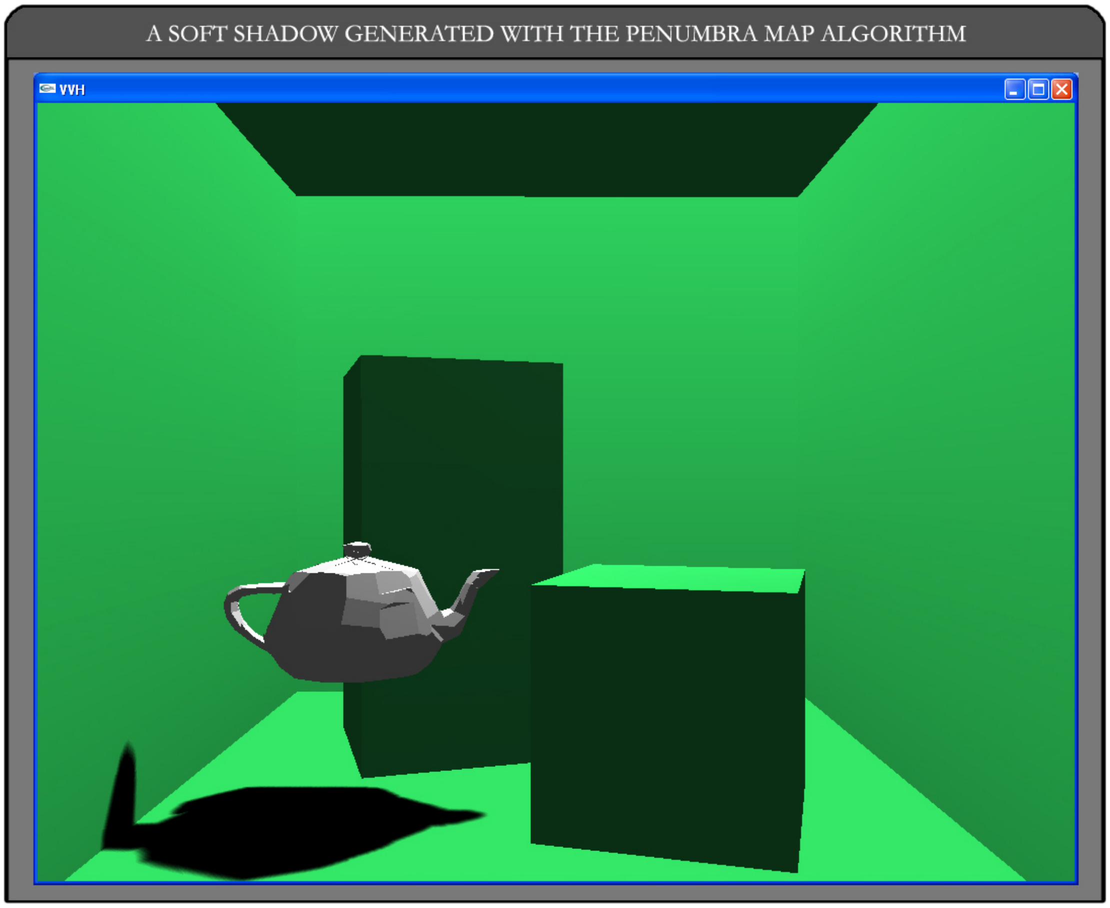


Figure 5.15: *The image above displays a teapot casting a soft shadow cast on to the background environment as viewed by the virtual camera.*

consisted of first loading the RTGI-generated background depth image into the depth buffer, and then enabling the stencil test and rendering in the foreground geometry. Next, by reading back the stencil buffer, we were able to determine which pixel locations corresponded to the inserted foreground model. At each of these foreground pixel locations we would then query the graphics board for the pixel's depth buffer value. By taking the window's (x,y) coordinate pair and the depth buffer value, we unprojected each pixel and found the corresponding (x,y,z) world space location. All of these world space locations, each of which corresponded to a point on the foreground object, were then packed together into an array and sent across the network to the RTGI system. RTGI then performed direct lighting calculations by casting a ray from each world space position to the primary light source in the scene. Either a one or a zero was then returned to our software for each location, indicating that it was either occluded, and thus in shadow, or visible to the light source in the scene. Based on the returned values, our software performed the necessary shading calculations in hardware.

This method proved to work well, but was slow for several obvious reasons. It involved reading back data from both the stencil buffer and the depth buffer on the graphics card, and read-back operations are notoriously cumbersome. Also, it relied upon the network transfer of the world space locations, thus adding latency into the system. In addition, the RTGI system had to perform ray-casting operations on each pixel location, which proved expensive if the number of foreground pixels was large.

Once we transitioned our system such that the background model was available to our software as well as RTGI, it became much simpler to cast shadows on to the foreground geometry. By reusing the same shadow map that we used to compute the penumbra map, we could quickly cast hard shadows on to the foreground object as produced by the surrounding background environment. If we were to cast soft shadows instead, we would need to find the silhouette edges for all geometries that stand between the light and the foreground object. Using our modified penumbra map algorithm, this would require many passes, reading back the stencil buffer, depth buffer, and computing the silhouette for each background object in the scene.

5.5 Scaling for Multiple Lights

Our current system configuration is not well suited to handling multiple light sources. For each additional light source that we wanted to incorporate, a new shadow map and penumbra map would have to be created. When dealing with high resolution shadow/penumbra maps, this can quickly fill up the available graphics memory. From a computational perspective, the calculations required to find the silhouette contour and render the cone and sheet geometry for each light scale linearly with the number of illumination sources. During the final rendering of the scene, an additional lookup must be made into each shadow map and each penumbra map, so once again the scaling is linear. In addition, the penumbra map algorithm does not perform well when it comes to combining soft shadows. Each shadow simply modulates the previous one by the new intensity, and this can lead to visual artifacts where several soft shadows overlap. To avoid the aforementioned complexities, we have configured our system to only handle a single light source. As discussed in Chapter 7, future enhancements to our system could easily be made to remove this restriction and increase the versatility of our software.

Chapter 6

Results

This chapter describes our system performance, both in terms of running time, and in terms of the accuracy and visual quality of the final images generated. The chapter is subdivided into three sections. In Section 6.1, we cover the performance of our system, identifying the major modules, discussing the running time associated with each module, and describing the existing bottlenecks. We also present some ideas as to how the bottlenecks might be alleviated. In Section 6.2, we present a study of the accuracy of the geometry that is reconstructed with our system. Finally, in Section 6.3, we conclude with a sequence of composite images, allowing the reader to determine for themselves the visual quality of our system's output.

6.1 Performance

There are a variety of components that come into play when measuring the overall performance of our system. This is complicated by the fact that many of the operations are computed in parallel, and the complexity of each stage of the system scales differently based on the number of cameras and the complexity of the object being captured.

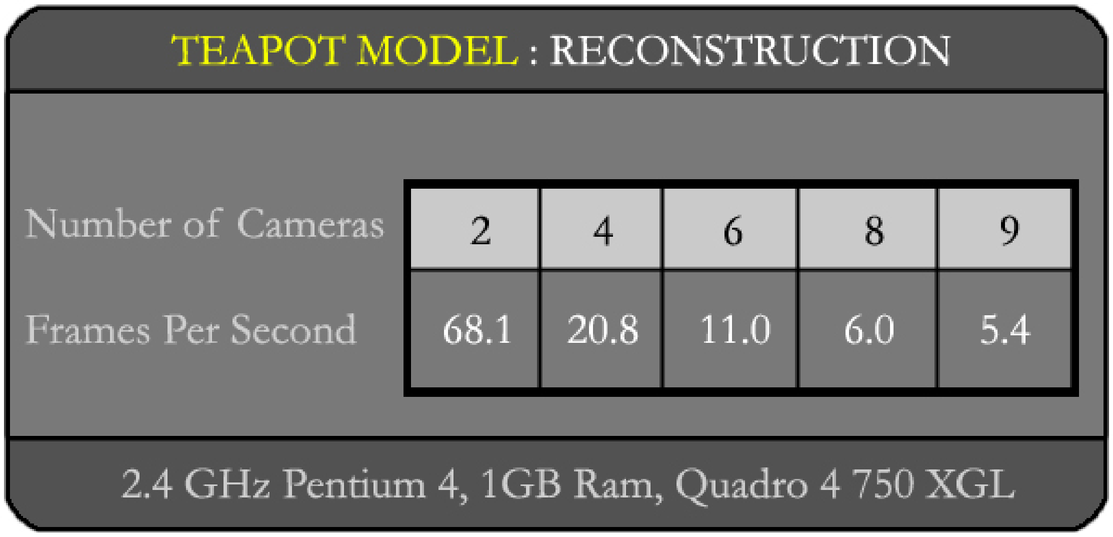
Before any processing can be performed on the image data, the captured frame must first be transfered from the video camera to its respective client computer. This transfer operation is done over FireWire 400 cabling, which supports a bandwidth of 400Mbps. As the camera has progressive scan output and is capturing at a resolution of 1024x768, the size of each frame is 2.25MB. The transfer time is therefore 0.047 seconds per frame, assuming optimal transfer speeds (see Figure 3.9).

The first algorithmic stage in our system consists of the image processing operations that are executed on the client computers. Because we are using a single-computer-per-camera model, the image segmentation and contour finding is done in parallel on each of the client processors. As a result, this initial image processing can be computed in constant time irrespective of the number of cameras. There are, however, several factors upon which the performance of this section is dependent. These include the image resolution, the projected area of the foreground object on the image plane, and the specified Gaussian kernel size used to filter the segmented image before the contours are found. The projected foreground area is important because after the object of interest has been segmented, we perform all further image processing operations, including the Gaussian filtering, on only that subset of the frame. Thus there is a direct relationship between the projected area and the amount of processing performed. Even at the cameras' highest capture resolution, 1024x768, we are able to process each frame in under 1/20th of a second, assuming a sufficiently small Gaussian kernel size, defined by the radius of the Gaussian filter, and an object whose bounding box occupies roughly half the image plane. If we disable Gaussian blurring, we are able to process each frame in roughly 1/40th of a second.

After the raw frame has been processed, the portion of the image containing the foreground object and the contour data are transferred to the server. Each of the client computers has its own direct Gigabit Ethernet connection to the server, making transfer rates of up to 119.2MB per second possible. If we assume that the image-space bounding box that contains the foreground object occupies 1/2 of the total number of image pixels, then the data being transferred to the server is roughly 1.125MB, provided that the size of the contour data is negligible. This results in optimal transfer times of 0.009 seconds per client computer. As we are performing these transfers in series, the total transfer time would be 0.036 seconds for all four client machines.

At the core of our system is the reconstruction algorithm which generates a polyhedral mesh based on silhouette input. The execution time of the reconstruction algorithm is $O(N^2)$, where N is the number of cameras, as each contour edge is projected on to each other image plane and intersected with that camera's silhouette. In addition to the number of cameras, the performance is also affected by both the complexity of the object and the resolution used when estimating the foreground contours. In our distributed four camera configuration, the reconstruction process takes roughly 0.04 seconds for a humanoid object. Table 6.1 displays the timing results for the reconstruction of a teapot model. To ensure that each run of the system was deterministic, we used synthetic input data for these time trials. In addition, all the processing was conducted on a single computer, as opposed to our distributed system.

Another component that influences the execution time is the creation of the penumbra map in hardware. While rendering the shadow map tends to be a simple



TEAPOT MODEL : RECONSTRUCTION

Number of Cameras	2	4	6	8	9
Frames Per Second	68.1	20.8	11.0	6.0	5.4

2.4 GHz Pentium 4, 1GB Ram, Quadro 4 750 XGL

Figure 6.1: *The figure above displays a timing table for the reconstruction of a teapot model. The resolution of the input images was 640x480, and a moderately coarse contour approximation was selected.*

operation that is only dependent on scene complexity, generating the penumbra map is a relatively time consuming process. This is due to the fact that in order to generate penumbral intensities, we are computing a fairly sophisticated pixel shader on each fragment that passes through the graphics pipeline. The size of the penumbra map geometry shares a direct relationship with the radius of the light and an inverse relationship with the distance separating the light from the foreground object (Figure 5.4). Therefore, as the light radius grows, or the object moves closer to the light, the resulting penumbra cone and sheet geometry expands as well. The result is that many more fragments are produced and consequently the GPU time required to process the fragments increases. It is very difficult to predict an execution time for this portion of the system; however, if the object is sufficiently far from the light and the light radius is not too large, the penumbra map can be generated in about 0.04 seconds. In our system, we only deal with a single light source in generating shadow effects. However, if the number of lights were permitted to expand, then the time requirement associated with this portion of the code would scale linearly with each additional light.

The generation and acquisition of the background image from RTGI also plays an important role in the timing of our system. The running time of the image generation phase is highly dependent on the nature of the background scene. Key factors include the scene's complexity, the global illumination effects that we are attempting to realize, and the number of nodes in the parallel compute cluster that we want to use to drive the pixel shaders. For the majority of our testing, we used 16 Intel dual processor 1.7GHz Xeon computers. We enabled the Render Cache system developed by Bruce Walter [WDP99] and we used the light clustering shader developed by Sebastian Fernandez [Fer04] in order to produce soft shadows from area light sources interactively. The background images, which are rendered at a resolution of 512x512, can be produced in 1/30th of a second. The RTGI system is networked to our server over Gigabit Ethernet, and we have been able to achieve transfer speeds of about 0.006 seconds per frame.

The final compositing sequence involves updating the OpenGL textures with the new video frames, loading the RTGI image into the color buffer, and rendering both the background and foreground models. With the exception of updating the OpenGL textures, which scales linearly with the number of cameras, all the other operations performed in this final stage can be run in constant time. The principal time consideration here is the complexity of both the background model and the reconstructed mesh. This is the fastest operation in our system and for most scenes the final rendering pass can be accomplished in under 0.005 seconds. Figure 6.2 displays the timing for our entire system.

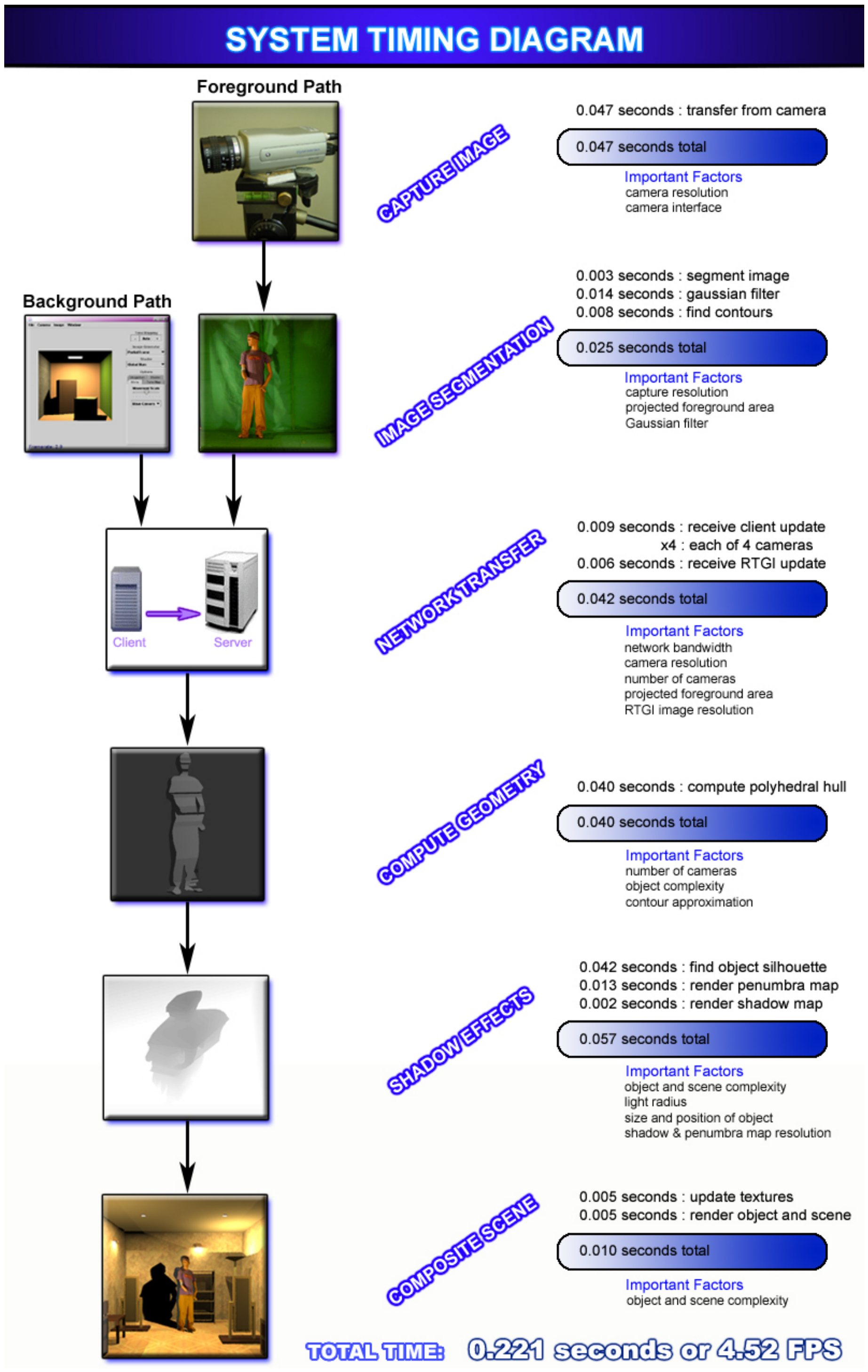

SYSTEM TIMING DIAGRAM
Foreground Path
Background Path
CAPTURE IMAGE
0.047 seconds : transfer from camera
0.047 seconds total
Important Factors
camera resolution
camera interface
IMAGE SEGMENTATION
0.003 seconds : segment image
0.014 seconds : gaussian filter
0.008 seconds : find contours
0.025 seconds total
Important Factors
capture resolution
projected foreground area
Gaussian filter
Client
Server
NETWORK TRANSFER
0.009 seconds : receive client update
x4 : each of 4 cameras
0.006 seconds : receive RTGI update
0.042 seconds total
Important Factors
network bandwidth
camera resolution
number of cameras
projected foreground area
RTGI image resolution
COMPUTE GEOMETRY
0.040 seconds : compute polyhedral hull
0.040 seconds total
Important Factors
number of cameras
object complexity
contour approximation
SHADOW EFFECTS
0.042 seconds : find object silhouette
0.013 seconds : render penumbra map
0.002 seconds : render shadow map
0.057 seconds total
Important Factors
object and scene complexity
light radius
size and position of object
shadow & penumbra map resolution
COMPOSITE SCENE
0.005 seconds : update textures
0.005 seconds : render object and scene
0.010 seconds total
Important Factors
object and scene complexity
TOTAL TIME: 0.221 seconds or 4.52 FPS

Figure 6.2: *The diagram above shows the performance of our system when capturing a human figure with a camera resolution of 1024x768, shadow and penumbra maps of size 1024x1024, and a background scene containing 195 quadrilaterals.*

6.1.1 Bottlenecks

As can be seen in Figure 6.2, there currently do not appear to be any major bottlenecks in our system. There are, however, several routines which have considerably longer running times than other portions of the code. These areas include:

- transferring the images from the camera to the client machines
- blurring the segmented frames
- transferring the image data to the server
- computing the polyhedral hull
- generating the penumbra map

The data transfer rate from the video cameras to the client computers is not an area we can readily improve. At the moment we are using the fastest available interface that our cameras support, FireWire 400.

Blurring the segmented images is not a strict requirement, however, it helps to eliminate camera noise and produce smoother contours. This portion of the code has been heavily profiled and can not be easily improved further. The simplest way to remove the performance hit of the filtering is just to disable it completely. Although the contour quality does tend to degrade slightly, if the capture environment has sufficiently good lighting conditions then the results are still very usable.

There are several methods which one could employ to decrease the transfer time between the client machines and the server. One option would be to try and compress the data to reduce the size of the data sent. A second possibility would be to run the data transfer routine in a separate thread. This would allow the data transfer to happen asynchronously, in parallel with the hull reconstruction of the previous frame.

Like the Gaussian filtering code, the polyhedral hull algorithm has undergone substantial profiling. In order to improve the performance of our reconstruction code, we implemented an internal memory management system. By initially allocating a large array of matrices and then distributing them on demand, we were able to speed up this portion of the system by over 30%. The reason we were able to see such a dramatic improvement is that the polygon intersection routine, used to generate the polyhedral hull, is continually allocating dynamic arrays. When these heap allocations and deallocations are done at runtime, the ongoing negotiations with the operating system tend to become a bottleneck. By allocating the memory all at once on startup, these problems can be easily avoided. To measure the effects of our perfomance tuning, we used Intel's VTune Performance Analyzer 7.0. Further improvements could be made through the development of more complicated data structures designed to maintain coherence.

The generation of the penumbra map has two costly operations, the calculation of the foreground object's silhouette and the rendering of the penumbra geometry. Computing the foreground silhouette is limited by the read-back speed from the graphics board. Thus the problem will be alleviated with future advances in graphics hardware. The rendering of the penumbra map might be improved simply by further optimizing the fragment shader to take better advantage of the hardware. Another approach might be to provide some functionality for automatically adjusting the light camera's clipping planes when rendering the penumbra map such that they form the tightest bounding box possible around the foreground object and receiver geometry. This would reduce the size of the cone and sheet geometry, and thus minimize the number of fragments on which the shader has to be run. Currently it is a difficult process to profile code written for the GPU. Since this was our first endeavor in pixel shader development, it is possible that our code is naive in implementation.

6.2 Reconstructed Geometry

In this section we will discuss the quality of the reconstructed polyhedral hull. It is our intent to show that the meshes which we generate are sufficiently accurate for casting visually plausible shadows and for new view synthesis between the reconstruction camera viewpoints.

6.2.1 Geometric Accuracy

In order to make an assessment as to the geometric accuracy of a reconstructed hull, we need to have a metric of evaluation. The method that we have chosen for gauging the quality of our results is an image space comparison of the reconstructed object's projected pixels. We render the reconstructed foreground object from a virtual viewpoint within our system and then we count the number of the pixels that it occupies on the image plane. Next we render an exact model of the object, from a camera-matched viewpoint, in Discreet's 3D Studio Max software. This image is referred to as the gold standard. By counting and comparing the number of image plane pixels occupied by the gold standard to that of the reconstructed rendering, we can compute the projected error for the viewpoint in question. In order to provide a control for the experiment, we render the generated model from the four reconstruction camera positions used during the reconstruction process.

Because the model is being viewed from the camera locations corresponding to the original input silhouettes, the projected error is kept to a minimum. Some error is still introduced, however, as the contour finding algorithm which we run on the segmented image retrieves the silhouette as a series of line segments. This approximation does not exactly represent the object's original shape, and thus errors arise along the border pixels.

For our experiment, we used a teapot model as the foreground object to be captured. Figure 6.3 shows the four, six, and eight camera configurations. The figure also shows the six virtual viewpoints which we selected for error analysis. The first three viewpoints are positioned midway between the upper and lower reconstruction camera planes, while the later three viewpoints share the same plane as the lower reconstruction cameras.

The results of the control experiment are displayed as difference images in Figure 6.4, where black pixels indicate that the rendered polyhedral hull deviated from the gold standard image. The error values associated with the exact camera-matched viewpoints are all less than 3.0%, and as expected, the difference pixels fall along the boundary of the object.

In Figures 6.5 and 6.6, rendered images of the reconstructed teapot model are compared to their gold standard counterparts at each of the six virtual viewpoints. This comparison is made for the teapot model as reconstructed by four, six, and eight cameras. It can be seen that as more cameras are added as input to the reconstruction process, the final mesh approximates the original object more accurately. This is exhibited by the fact that with each camera added, the projected error is decreasing, converging to the case where the viewpoints are exactly aligned. With an infinite number of cameras, the reconstructed hull would have the correct

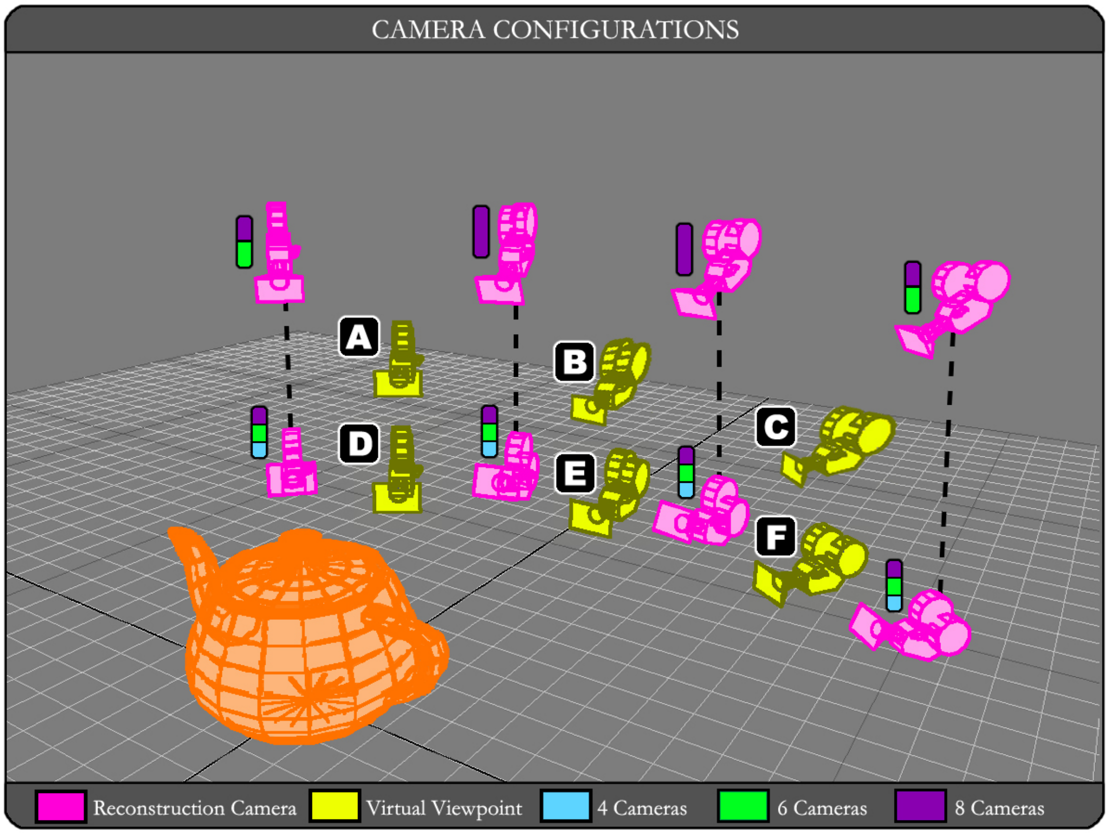


Figure 6.3: *The image above shows the four, six, and eight camera configurations used for geometric reconstruction. It also identifies the six virtual viewpoints which we selected for evaluating the accuracy of our computed models. These virtual viewpoints are labeled A, B, C, D, E and F.*

projection from any viewpoint, minus the error introduced by the contour approximations. However, this is not to imply that the reconstructed geometry would be true to the original form, as the reconstruction algorithm is unable to capture interior concavities that are not visible in any silhouette. Figure 6.7 displays the same set of virtual viewpoints as shown in Figure 6.6, however, in this sequence of images the teapot has been shaded. This illustrates the improvement in rendering quality that can be attained by using additional reconstruction cameras.

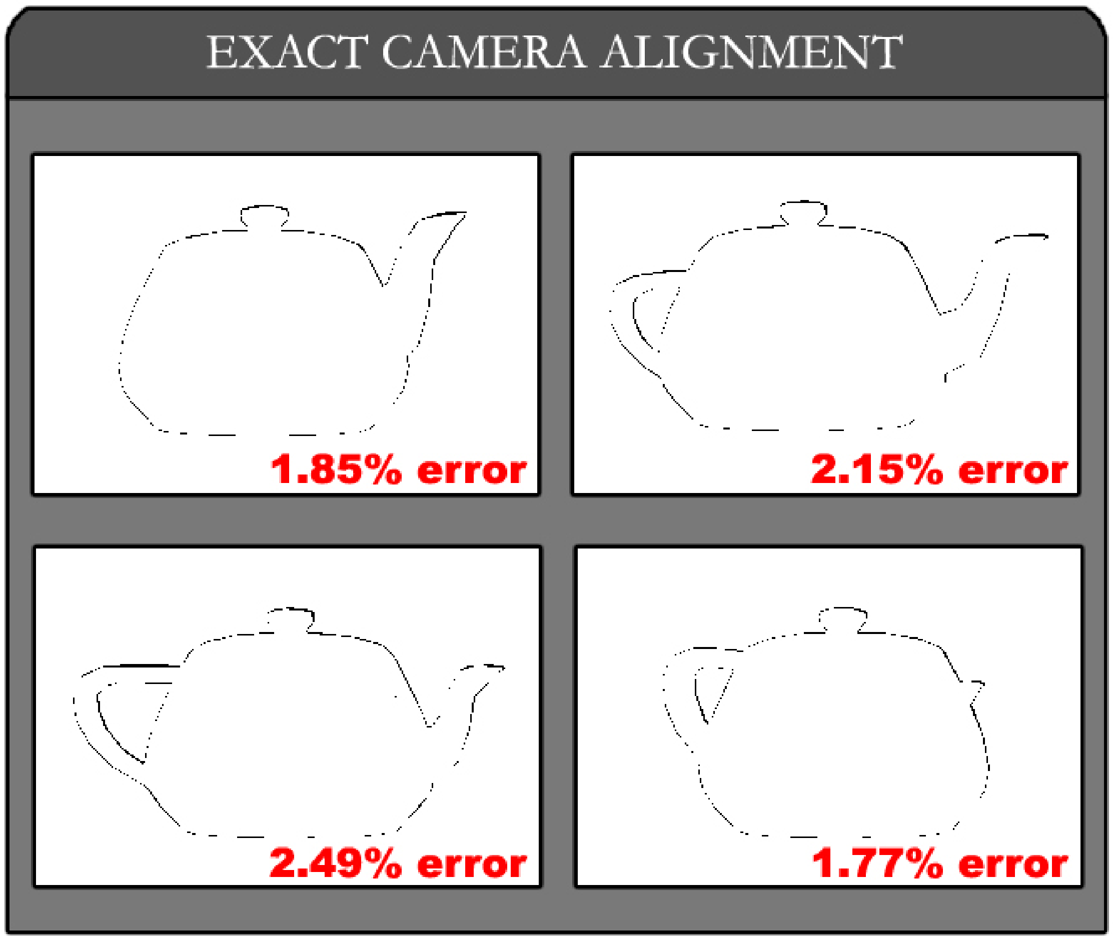


Figure 6.4: *The figure above shows the difference images that were generated by rendering the model from each of the four reconstruction camera locations. Black pixels indicate that the image of the reconstructed teapot varied from the image of the original teapot. White pixels indicate that the two images were the same. Note that the error pixels all lie along the border of the teapot. This is because the contour extraction algorithm has approximated the exact silhouette with line segments. These images were rendered at a resolution of 640x480.*

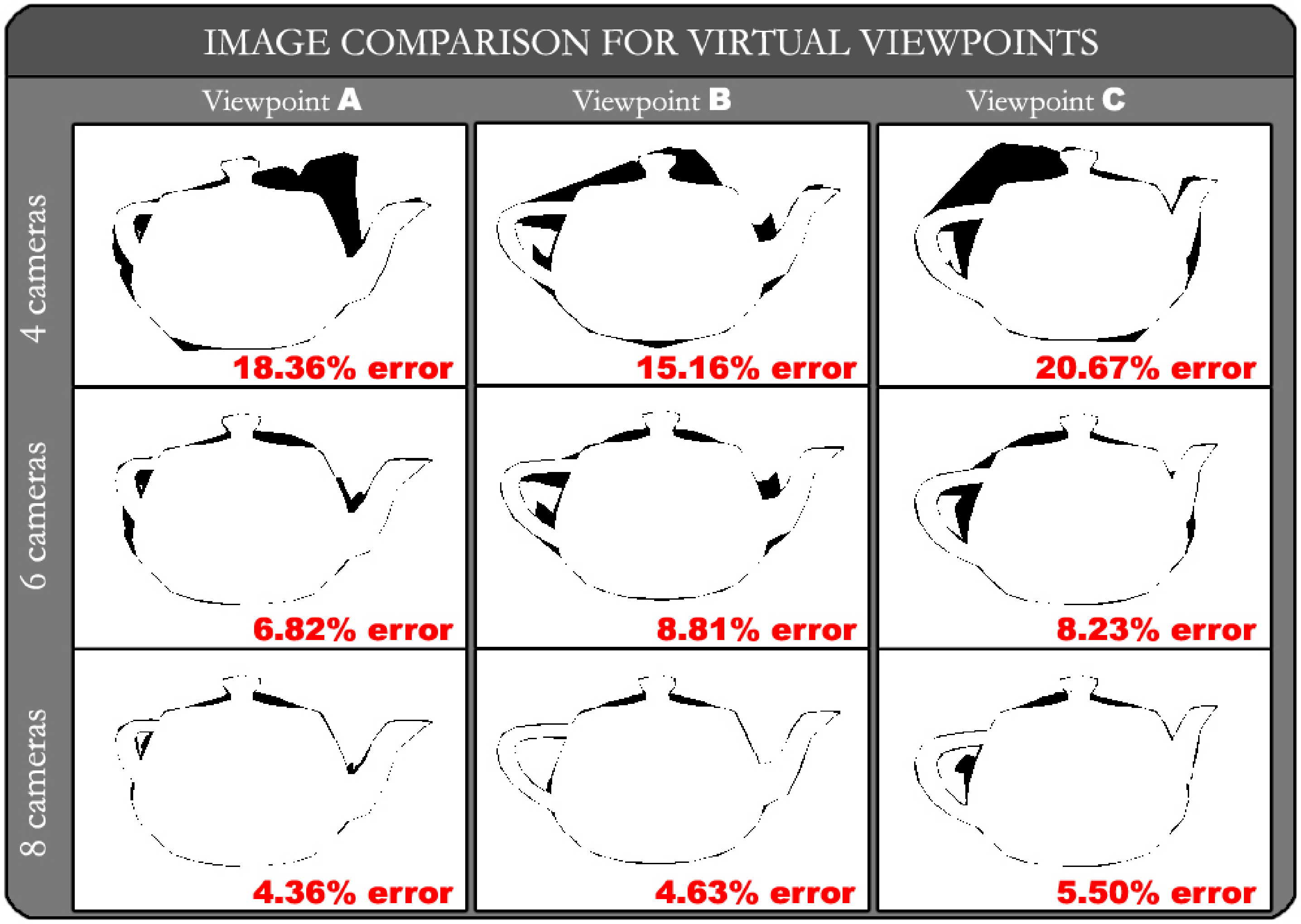


Figure 6.5: *The figure above displays the difference images as seen from virtual viewpoints A-C with varying numbers of reconstruction cameras.*

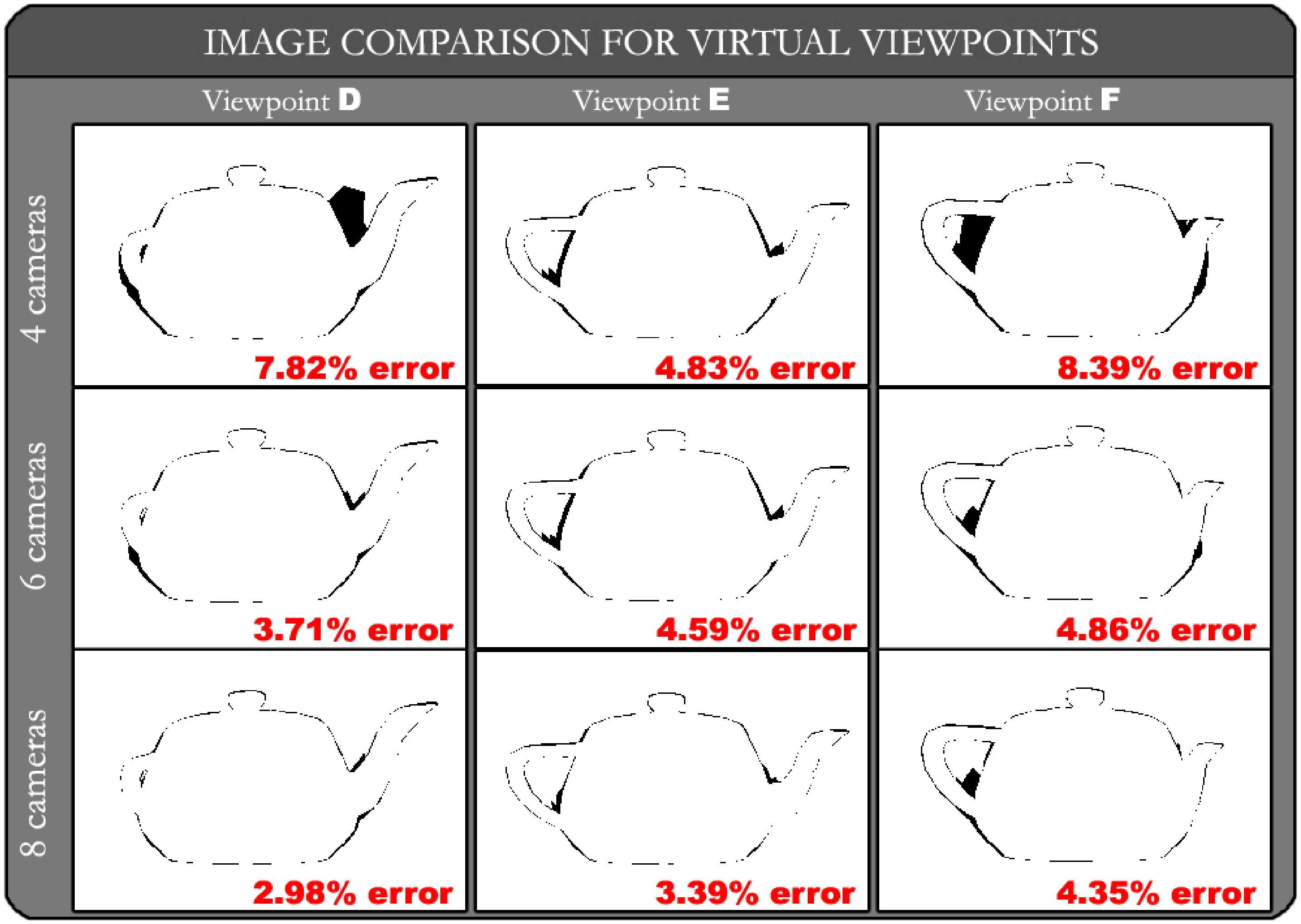


Figure 6.6: *The figure above displays the difference images as seen from virtual viewpoints D-F with varying numbers of reconstruction cameras.*

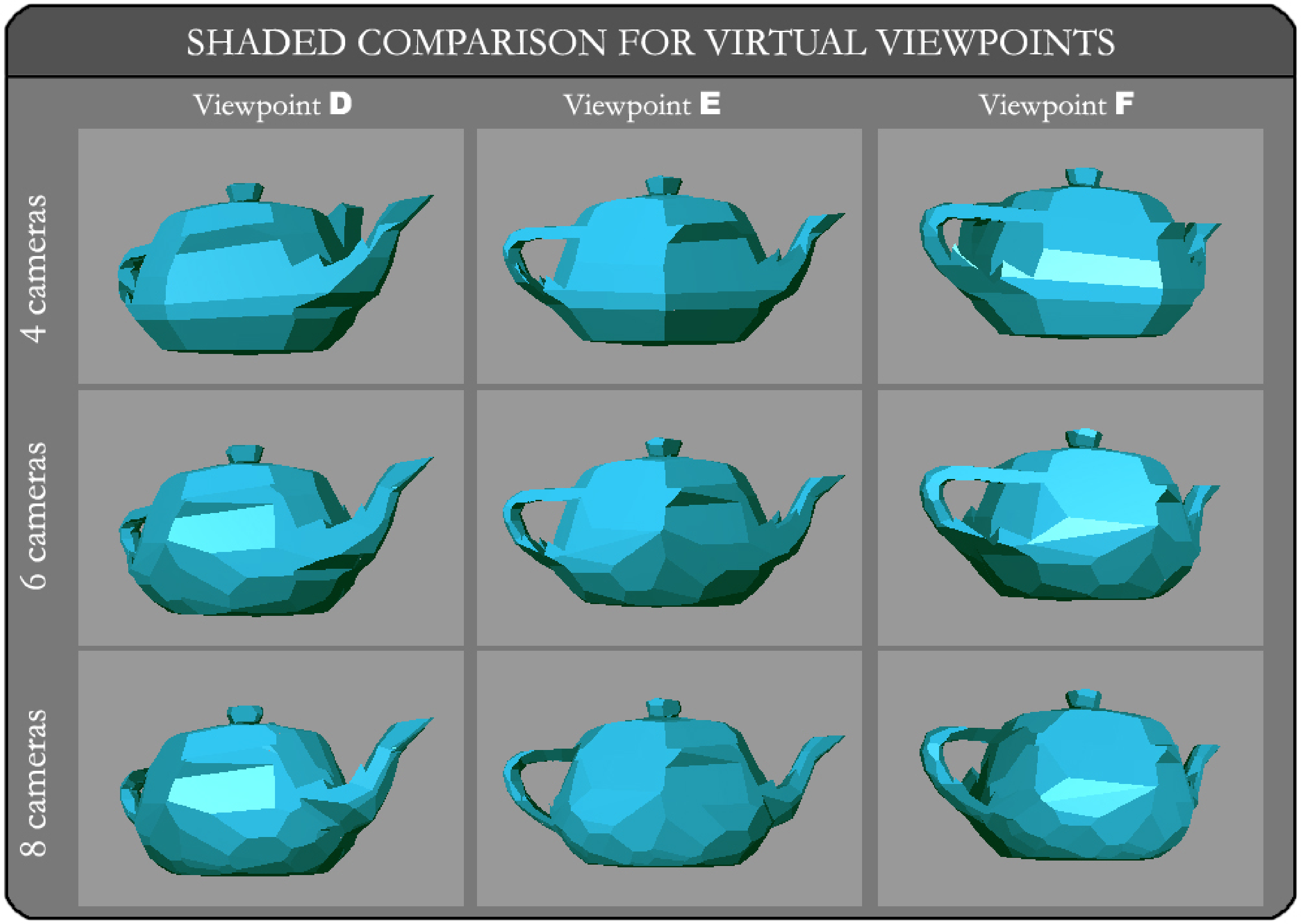


Figure 6.7: *The figure above displays the teapot's hull as seen from viewpoints D-F for 4, 6, and 8 cameras. Although the reconstructed hull provides a reasonable approximation, the shaded images reveal inaccuracies in the reconstructed geometry.*

6.3 Final Composites

In this section we will present a sequence of images that reflect the quality of results our system is able to produce. We will start with the output of individual portions of the algorithm and conclude with a series of final composite images that show everything working together. Figure 6.8 displays two screen captures of our software, the first showing the video frames as seen by each of our four cameras, and the second showing the extracted silhouette contours overlaid on the video frames. This sequence of video was then used to reconstruct the model seen in Figure 6.9. We present two views of the reconstructed model, both of which are novel and not aligned with any of the reconstruction viewpoints. In Figure 6.10 we show the results of another video sequence. In this example, we have captured a person swinging a golf club, and inserted their model into a synthetic environment. The geometry is displayed in wire frame mode to accentuate the dynamic nature of the mesh, and show that it is recomputed each frame. Figure 6.11 displays the golf composite sequence rendered in shaded polygon mode. In Figure 6.12, we present a series of images that illustrate all the components of our system working together. The foreground object is a person walking around and bouncing a ball in a synthetic environment. The reconstructed hull has been inserted into this simulated background, and shadows are being cast both onto the foreground geometry from the background model and onto the background scene from the foreground object. Finally, in Figure 6.13, we illustrate the difference between hard contact-shadows and soft shadows produced by our system.

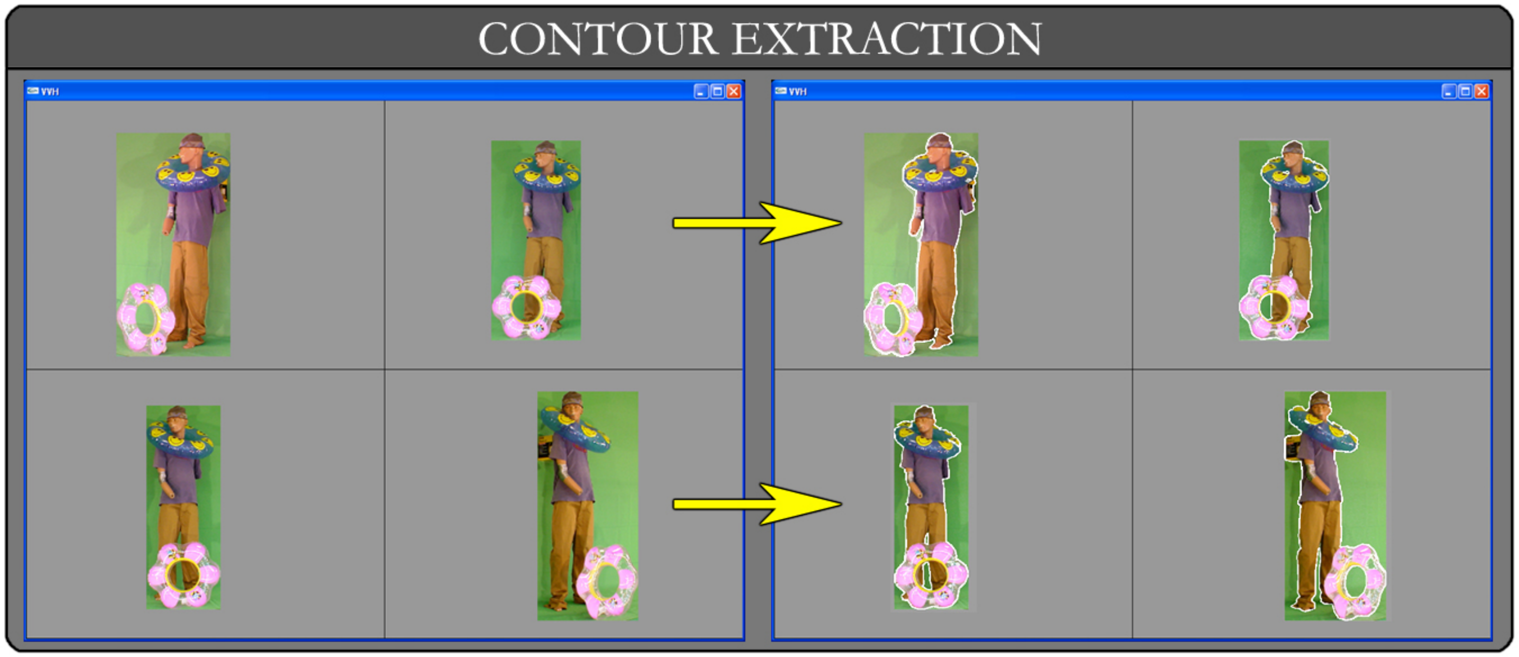


Figure 6.8: *The above figure displays a foreground object as seen by each of the cameras in our system. The left half of the figure reveals only the raw video frames, while the right half has the silhouette contours overlaid in white.*

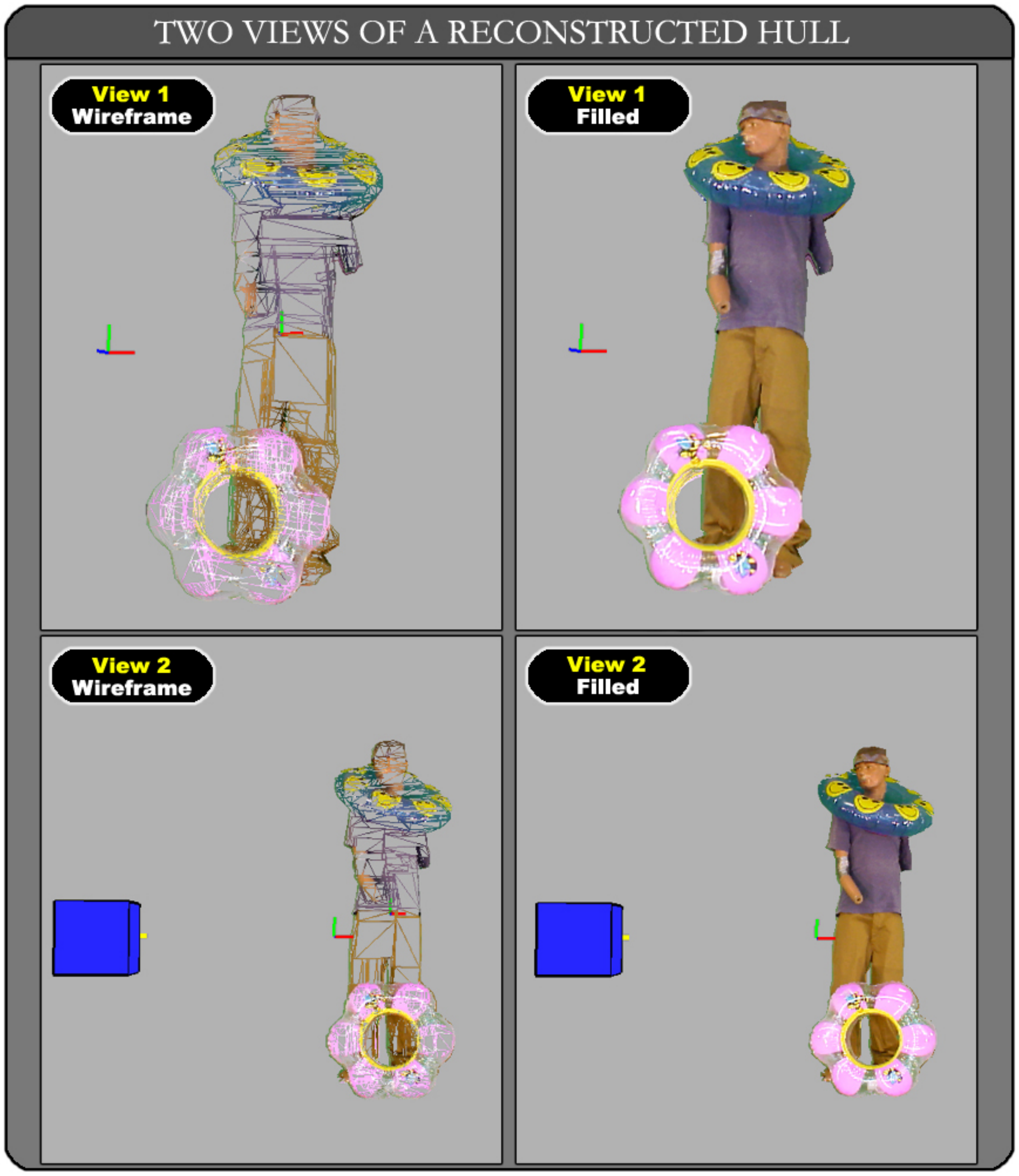


Figure 6.9: *The figure above shows two reconstructed views of our mannequin, Howie, with some beach toys. He is displayed in both filled mode and wire frame mode to show off the quality of the reconstructed mesh.*

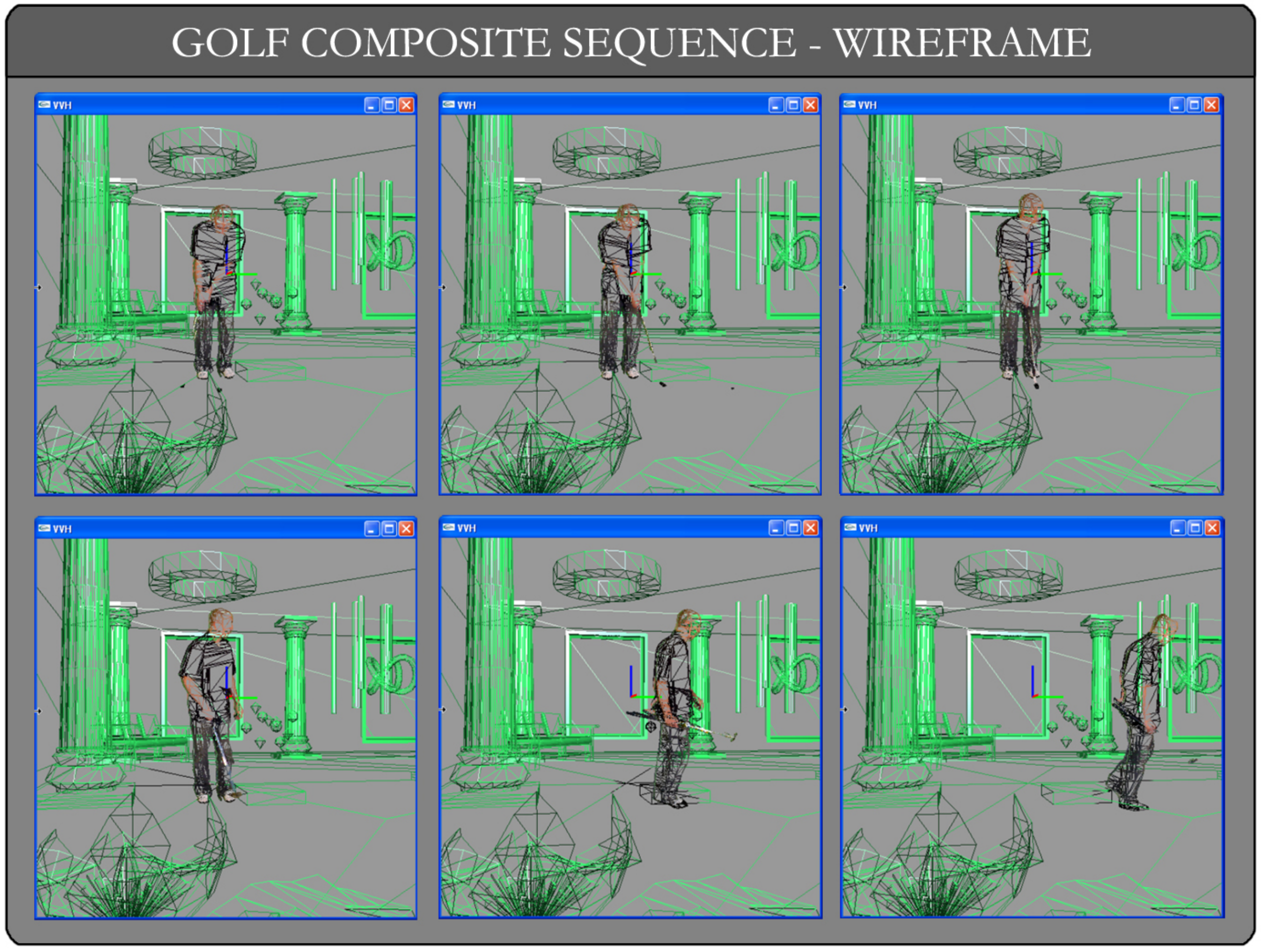


Figure 6.10: *In the wireframe image sequence above, a reconstructed model of a person golfing is inserted into a virtual environment. Note that although the entire image is displayed in wireframe, only the foreground model of the golfer was reconstructed by our system.*

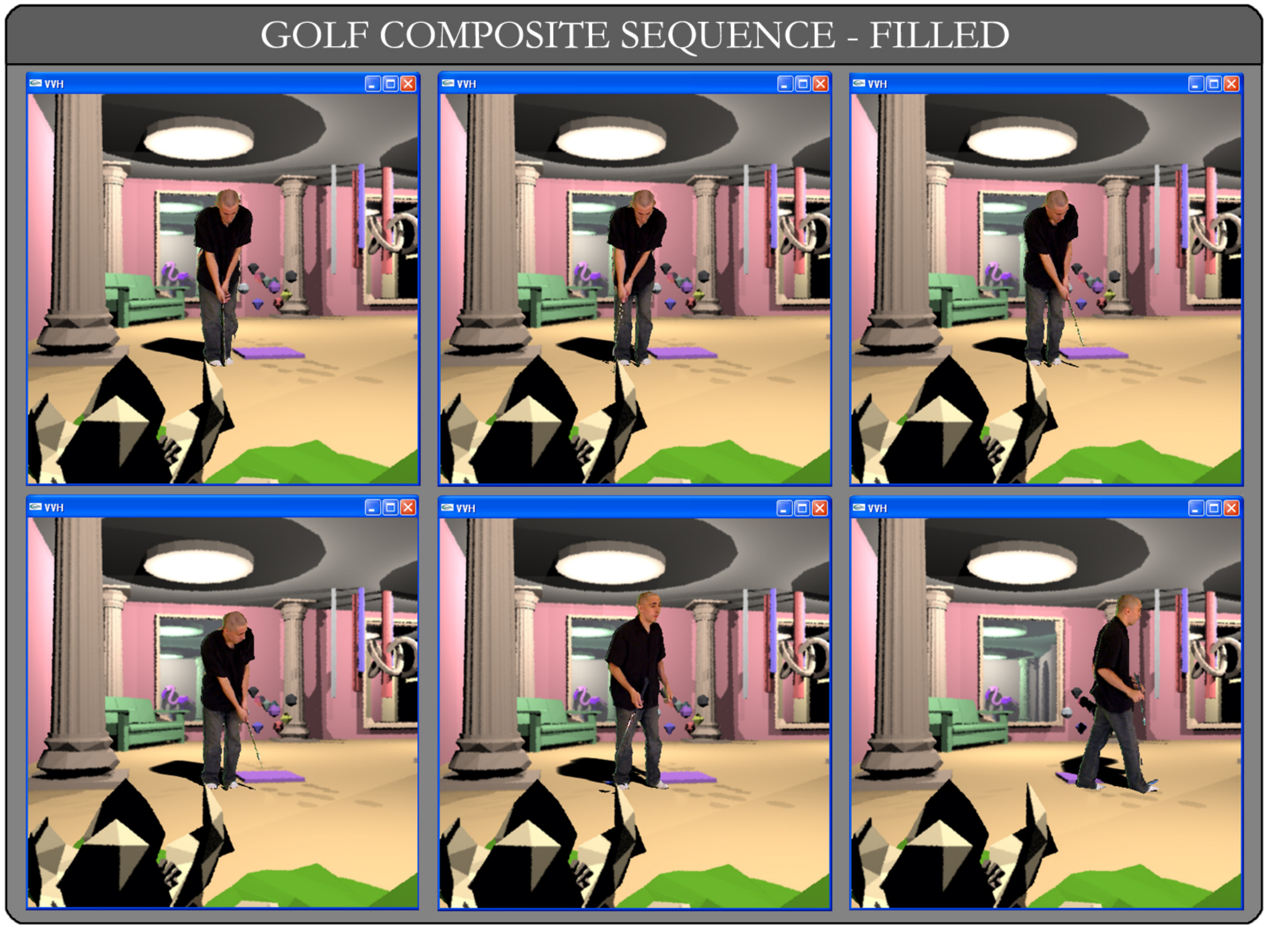


Figure 6.11: *The image above displays a filled version of the golfing composite sequence seen in Figure 6.10. Note the golf club's poor reconstruction. This is due to its thin geometry, as well as the fact that it is highly specular. Specular surfaces suffer increased colorbleeding from the greenscreen, and thus are more likely to be incorrectly segmented as background.*

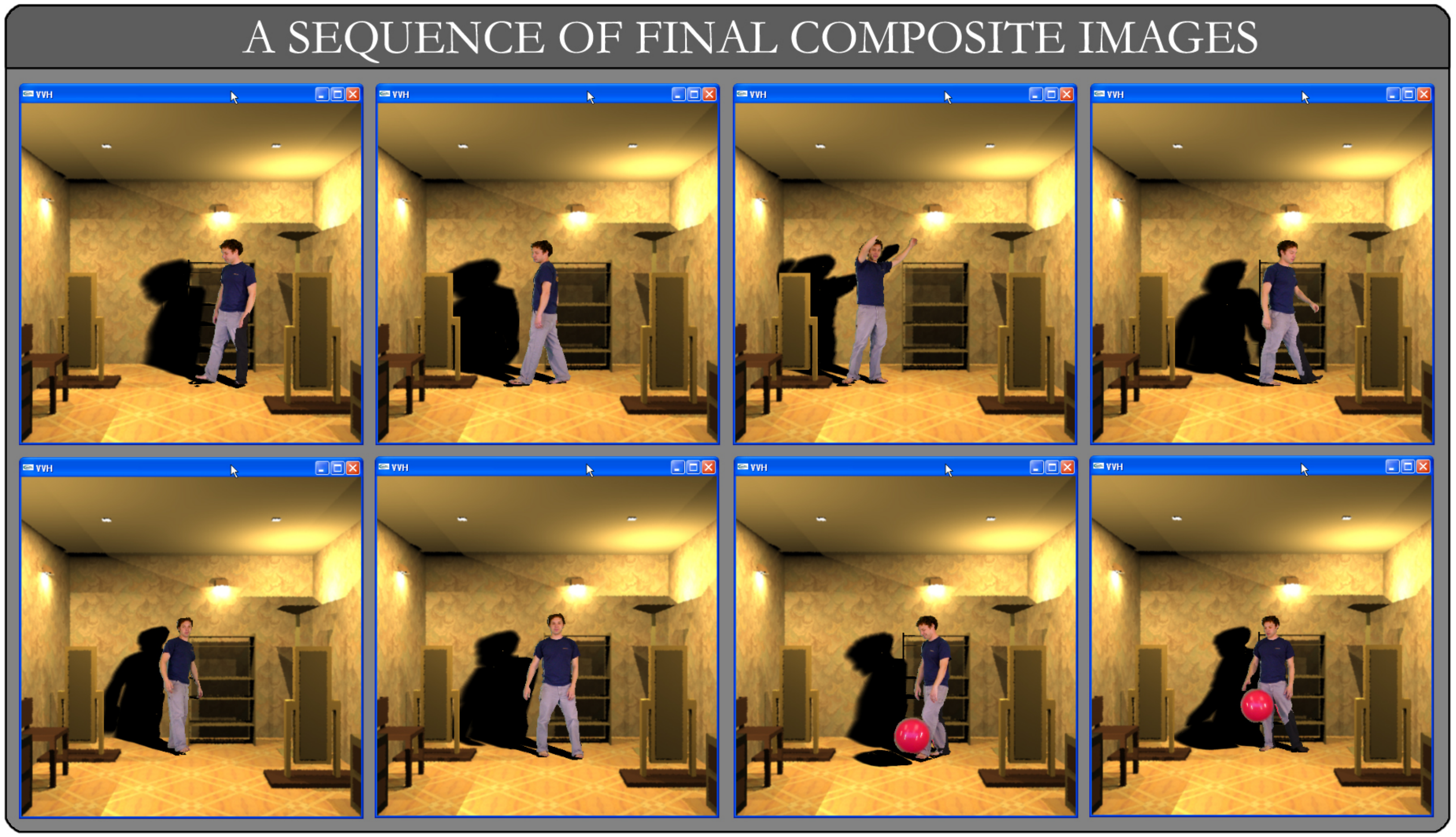


Figure 6.12: *In this sequence of images, the reconstructed foreground object has been composited into a background environment with shadowing effects enabled.*

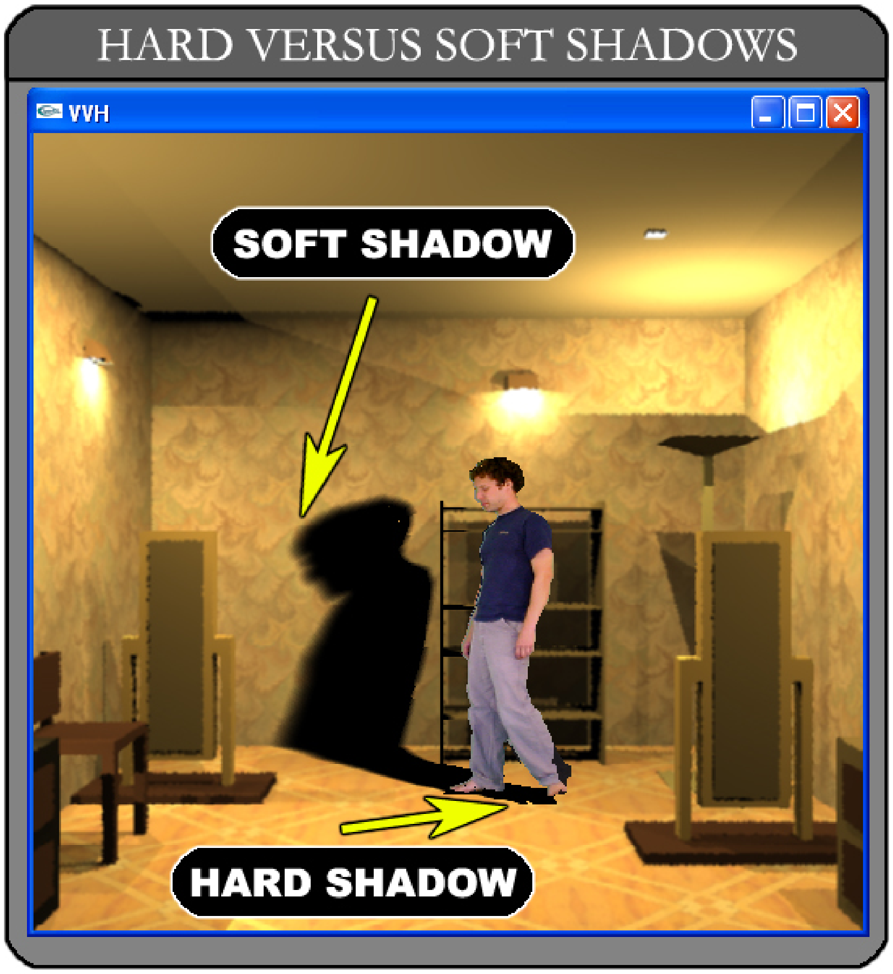


Figure 6.13: *The figure above illustrates the different types of shadows that our system is capable of generating. The person's feet are casting a hard contact-shadow onto the surface of the floor. The person's head, however, is casting a soft shadow onto the far wall of the room.*

Chapter 7

Conclusion

In this thesis we present a novel compositing system that leverages multiple views of the foreground object to reconstruct a 3D approximation of its shape. Having a three-dimensional representation of the subject allows us to generate more visually plausible composites than would be possible with only a 2D matte. We perform the entire compositing process in three dimensions, permitting us to properly handle occlusions and simulate physically-based global illumination effects. Furthermore, in our system the user is free to move the camera to an arbitrary location, thus eliminating the traditional view-dependent restriction of a 2D compositing system. We demonstrate an approach for combining the reconstructed geometry with synthetic imagery created by the Program of Computer Graphics' Real-Time Global Illumination (RTGI) system, and we introduce a technique for casting shadows between the foreground and background environments such that neither set of geometries has self-shadowing computed twice.

We have shown that the foreground reconstruction and image compositing can be accomplished in a real-time system that works interactively with the video capture process. We use a distributed computing paradigm to separate the image segmentation and matte operations from the reconstruction and compositing. Our system also utilizes modern graphics hardware for computing soft shadows in pixel shaders executed on the GPU. We have developed and implemented our three-dimensional compositing system using consumer grade hardware, making it appealing as a potential replacement to current compositing systems.

7.1 Future Work

A number of short-term improvements could augment our system's functionality and robustness.

One limitation of our existing system is that it can only handle a single light source. To relax this constraint, a separate shadow map and penumbra map has to be generated for each additional light in the virtual scene. A method for computing how overlapping shadows combine would also need to be determined. Adding this functionality to our system would increase its versatility in terms of the types of usable background environments and provide even more realistic imagery.

A better improvement would be to pass the generated polyhedral mesh to the RTGI system before rendering. This would simplify the compositing process, as all the rendering would be performed with one cohesive integrated environment, and RTGI is capable of generating arbitrary levels of global illumination depending on the desired application. The existing barrier that must be overcome is RTGI's inability to handle dynamic geometry creation. Adding new geometry would require that the scene's data structure be either updated or completely rebuilt, and there is currently no infrastructure in place for performing this operation on a frame-to-frame basis.

The use of background subtraction routines as an alternative to greenscreening would afford our system more flexibility in regard to camera placement, as we would no longer require a green backdrop for extracting the foreground matte.

Finally, the geometric reconstruction algorithm could be improved to more gracefully degrade as objects move in and out of view of the cameras. In our current system, if a portion of the foreground object falls outside the view of any of the reference cameras, then that part is clipped away. This is because the hull is reconstructed using the intersection of all of the cameras' viewing cones. Instead, a visibility check could be performed on each potential hull surface, such that the final surface is the intersection of the viewing cones from each camera whose frustum encompasses that location. This would result in more coarse geometry where only a few cameras could see the object, but would make the system more adaptive to varying situations.

One potential avenue of research would be to either improve upon, or remove completely, our system's matched lighting restriction. If the foreground environment's lighting is known, then a user-interface could be designed to alter the locations, colors, and intensities of the lights in the background scene such that they arbitrarily approximate the foreground illumination. To aid in this process, light probes could be used to capture and store the foreground lighting configuration. Alternatively, an inverse-lighting operation could be run on the video textures, thus removing the original lighting. Then the textures could be re-lit according to the illumination in the virtual set. Unfortunately, both of these steps require knowledge of the foreground object's material properties. Estimating these properties in an interactive application is a daunting task. One alternative would be to have the user manually estimate material properties for each foreground object before the compositing process started.

Another area of research concerns how to maintain frame coherence, reusing previous model data to further refine the current mesh. If rigid-body motion is assumed, then tracking a few points would be sufficient to determine how the model's location and orientation is changing from frame to frame. Subsequent frame data could then serve as additional constraints on the object's volume, "chipping away" at the polyhedral hull so that it more closely approximates the actual volume. If the object is deformable, as is the case with human subjects, then simple point tracking is not sufficient to determine the subject's location and pose. For this scenario, a more sophisticated technique would be required. For example, the captured silhouettes could be matched to different poses of a parametric humanoid model. Performing this matching in real-time is a difficult problem to solve. Assuming the matching problem was feasible, one idea would be to store a database of potential foreground objects. Then, after the object had been identified and its pose and parameters determined, the more accurate database model could be textured with the video frames and used for the composite rendering.

Finally, research could be done to determine the optimal camera placement for reconstructing the shape of an object. In this thesis we only conducted some preliminary investigations regarding the measurement of the projected error for varying numbers of reference cameras. However, further studies could be done to measure how the hull of an object improves with each additional camera added, to determine when the point of diminishing returns has been reached, and to evaluate if the placement of cameras is scene specific, or if there is an optimal global configuration.

Appendix A

Camera Calibration

The camera calibration toolbox for MATLAB was used exclusively for determining the intrinsic and extrinsic parameters of the cameras in our system. The input to the toolbox is a sequence of digital images, each containing the surface of a checkerboard. The user selects the four corners which define the outer extents of the checkerboard, and the software can then perform automatic extraction of the internal grid corners. As the toolbox computes the grid corners, it prompts the user to enter a value for the adjustment of radial distortion (if desired). The main calibration routine can then be executed, which utilizes a gradient descent technique to minimize the reprojection error of the grid pixels. This returns the internal and external parameters of the camera.

The format of the returned data is such that each camera is considered its own world reference frame, and the rotation and translation vector to the grid origin are given. This is not ideal for our purposes, as we require a single world reference frame, in terms of which each camera basis and position are given. To make this conversion, we invert each camera's rotation matrix, by taking its transpose, and then we negate each camera's translation vector and multiply it by the camera's new rotation matrix. In MATLAB syntax, this resolves to the following for a single camera:

```
Camera1_RotationMatrix = Rc_1'
Camera1_Translation = Camera1_RotationMatrix * (-Tc_1)
```

Where “Rc_1” is the rotation matrix returned by the toolbox for Camera 1 and “Tc_1” is the translation vector returned by the toolbox for Camera 1. After we have the location and pose of each camera in terms of a global reference frame, we convert the reference frame such that it is the same right-handed basis that our system uses internally. In our system, the x-axis is positive to the left, the y-axis is positive in the up direction, and the z-axis is positive going into the screen. The MATLAB syntax to make this conversion for a single camera is:

```
Cam1_RotTmp = [Cam1_RotMat(2,:); -Cam1_RotMat(1,:); Cam1_RotMat(3,:)]
Cam1_RotationFixed = [-Cam1_RotTmp(:,1) -Cam1_RotTmp(:,2) Cam1_RotTmp(:,3)]
Cam1_TranslationFixed = [Cam1_Trans(2,:); -Cam1_Trans(1,:); Cam1_Trans(3,:)]
```

The “fixed” rotation matrices and translation vectors, along with the principal point and focal length, are stored in a settings file that is loaded into our program each time it is executed. This provides the necessary information for computing the cameras’ projection matrices as well as the fundamental matrices which relate the cameras.

Appendix B

Trigger Circuit Design

Each of the DFW-X700 cameras has an external trigger that can be used to drive the capture process, by sending a low pulse of at least 1ms duration on the "TRIG IN" connector, pin 3. The cameras have an internal pull-up resistor, so in order to trigger the low signal, we simply pull pin 3 to ground when we want an image to be captured. For our trigger design, we take advantage of the serial port, COM1, on the central server. According to the RS232 standard, the serial port transmits a '1', or logic high, as -3 to -25 volts and a '0', or logic low, as +3 to +25 volts. We use the data transmit pin on the DB9 serial port connection to control our trigger circuit. When no data is being actively transmitted, our serial port outputs -11 volts on the data transmit pin. When we want the signal to go high, we write out a packet of zeros, and the data transmit pin goes to +7.5 volts.

The trigger circuit is quite simple and consists of an NPN transistor and a 2.2k resistor. The signal ground (SG) from the serial port, pin 5, is tied to the emitter pin on the 2N3904 transistor, as is the ground from each of the four trigger cables which run to the cameras. The data transmit pin (DT), or pin 3, from the serial port is tied to the base pin on the transistor through the 2.2k resistor, and the collector pin is attached to the positive leads of the trigger cables. When our software, running on the server, writes a string of zeros to the serial port, the DT pin goes high, permitting the flow of electrons from the emitter to the collector, and pulling the “TRIG IN” pins on each of the cameras to ground. This causes the trigger to fire, and an image to be captured.

Bibliography

[AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, July 2003.

[ADMAM03] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In *Graphics Hardware 2003*, pages 33–40, July 2003.

[AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 297–306, June 2002.

[ARHM00] Maneesh Agrawala, Ravi Ramamoorthi, Alan Heirich, and Laurent Moll. Efficient image-based methods for rendering soft shadows. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 375–384, July 2000.

[AWG78] P. Atherton, Kevin Weiler, and Donald P. Greenberg. Polygon shadow generation. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 275–281, August 1978.

[Bli88] James F. Blinn. Jim blinn's corner: Me and my (fake) shadow. *IEEE Computer Graphics & Applications*, 8(1):82–86, January 1988.

[Box04] http://www.boxofficemojo.com, 2004.

[CD03] Eric Chan and Frédo Durand. Rendering fake soft shadows with smoothies. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pages 208–218, June 2003.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, volume 11, pages 242–248, July 1977.

[Deb96] Paul E. Debevec. *Modeling and Rendering Architecture from Photographs*. PhD thesis, University of California at Berkeley, December 1996.

[DTM96] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 11–20, August 1996.

[FB03] Jean-Sébastien Franco and Edmond Boyer. Exact polyhedral visual hulls. In *Fourteenth British Machine Vision Conference (BMVC)*, pages 329–338, September 2003. Norwich, UK.

[Fer04] Sebastian Fernandez. *Interactive Direct Illumination in Complex Environments*. PhD thesis, Cornell University, August 2004.

[FFBG01] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 387–390, August 2001.

[Hai01] Eric Haines. Soft planar shadows using plateaus. *Journal of Graphics Tools*, 6(1):19–27, 2001.

[Hec92] Paul Heckbert. Discontinuity meshing for radiosity. In *Third Eurographics Workshop on Rendering*, pages 203–226, May 1992.

[HH96] Michael Herf and Paul S. Heckbert. Fast soft shadows. pages 145–145, 1996.

[HH97] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, January 1997.

[Kra04] Adam Kravetz. Polyhedral hull online compositing system: Texturing and reflections, August 2004.

[Lau94] Aldo Laurentini. The visual hull concept for silhouette-based image understanding. 16(2):150–162, 1994.

[LMS03] Ming Li, Marcus Magnor, and Hans-Peter Seidel. Hardware-accelerated visual hull reconstruction and rendering. In *Proceedings of Graphics Interface 2003*, Halifax, Canada, 2003.

[LSS02] Ming Li, Hartmut Schirmacher, and Hans-Peter Seidel. Combining stereo and visual hull for on-line reconstruction of dynamic scenes. In *Proceedings of IEEE 2002 Workshop on Multimedia and Signal Processing*, pages 9–12, December 2002.

[LTG92] Daniel Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics & Applications*, 12(6):25–39, November 1992.

[MBM01] Wojciech Matusik, Chris Buehler, and Leonard McMillan. Polyhedral visual hulls for real-time rendering. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 115–126, June 2001.

[MBM02] Wojciech Matusik, Chris Buehler, and Leonard McMillan. An efficient visual hull computation algorithm. Technical report, MIT, February 2002.

[MBR$^+$00] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 369–374, July 2000.

[McC00] Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics*, 19(1):1–26, January 2000.

[MPN$^+$02] Wojciech Matusik, Hanspeter Pfister, Addy Ngan, Paul Beardsley, Remo Ziegler, and Leonard McMillan. Image-based 3d photography using opacity hulls. *ACM Transactions on Graphics*, 21(3):427–437, July 2002.

[MPZ$^+$02] Wojciech Matusik, Hanspeter Pfister, Remo Ziegler, Addy Ngan, and Leonard McMillan. Acquisition and rendering of transparent and refractive objects. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 267–278, June 2002.

[OK93] M. Okutomi and T Kanade. A multiple-baseline stereo. 15(4):353–363, 1993.

[PSS98] Steven Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. October 1998.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 283–291, July 1987.

[SBK$^+$99] Hideo Saito, Shigeyuki Baba, Makoto Kimura, Sundar Vedula, and Takeo Kanade. Appearance-based virtual view generation of temporally-varying events from multi-camera images in the 3d room. April 1999.

[SD97] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. *Proc. Computer Vision and Pattern Recognition Conf.*, pages 1067–1073, 1997.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. *ACM Transactions on Graphics*, 21(3):557–562, July 2002.

[Sel03] Jeremy Selan. Merging live video with synthetic imagery. Master's thesis, Cornell, 2003.

[SFG] http://www.sfgate.com.

[SKvW$^+$92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 249–252, July 1992.

[SS02] Gregory S. Slabaugh and Ronald W. Schafer. Image-based photo hulls for fast and photo-realistic new view synthesis. February 2002.

[SVZ00] Dan Snow, Paul Viola, and Ramin Zabih. Exact voxel occupancy with graph cuts. In *Proc. Computer Vision and Pattern Recognition Conf., volume 1*, pages 345–352, 2000.

[WDP99] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In *Eurographics Rendering Workshop 1999*, June 1999.

[WH03] Chris Wyman and Charles Hansen. Penumbra maps: Approximate soft shadows in real-time. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pages 202–207, June 2003.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 270–274, August 1978.

[Zha00] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, 2000.