October 2007

GPU Optimization of an Existing Free-Viewpoint Video System

Neal C. Orman
Worcester Polytechnic Institute

Project Number: RL1 MQP JP01

GPU Optimization of an Existing Free-Viewpoint Video System

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

____

Neal Orman

October 22, 2007

Approved:

____

Professor Robert Lindeman, Major Advisor

Abstract

With the advances in quality and affordability of video cameras, people's lives are being recorded more than ever. However, video cameras have the distinct disadvantage of only capturing a two-dimensional image. This means that a lot of information about a particular scene is lost due to occlusion and a lack of depth information. However, when video is recorded from multiple viewpoints, it is possible to regain this lost information and use it to construct a more complete model of the scene being recorded. By using this model to interpolate between the known viewpoints, it is possible to reconstruct the scene from a variety of angles. This was the goal of the project developed at Advanced Telecommunications Research Institute International (ATR) in Kyoto, Japan. The concept of the project was to capture the actions of hospital staff in order to better understand what happens in a medical environment so that such knowledge could help to prevent medical errors and improve patient care.

While the Cinematized Reality system succeeded to generate a good model of the captured scene and render the scene from any arbitrary angle, the processing time required to render each frame totaled nearly a full minute. In order to improve the performance, some of the processing steps for each frame were moved from the CPU to the graphics processing unit (GPU). While the GPU performs many of the same kinds of operations as the CPU, it is designed to operate in a manner better suited to a traditional graphics pipeline. While this offers a number of challenges in designing programs to work on a GPU, it offers the potential for great performance benefits for certain types of programs.

This paper presents the state of the system prior to optimization, the techniques used to implement the GPU-optimized version of the Cinematized Reality system, and the performance benefit gained. Despite requiring several significant changes to the system to adapt it for GPU processing, Cinematized Reality proved to be well suited to GPU computation, resulting in certain parts of the system running over 100 times faster.

Acknowledgment

I would like to take this opportunity to thank the dedicated researchers at ATR, for their overwhelming support and assistance with the completion of this project. My supervisors Kim-san and Sakamoto-san specifically and the Knowledge Science Laboratories director Kogure-san have been critical in their guidance and support of the project from day one through the final days. A special thanks to the SHIEN department and the planning section of KSL for helping with everything that was necessary to make the project possible and my stay in Japan as comfortable and enjoyable as possible. To my co-workers at ATR, whose support, interest, and encouragement was unending.

Thank you to the Interdisciplinary and Global Studies Department of WPI for their organization and support in helping to make this opportunity for me a reality seemingly against all odds. A special thanks to the Computer Science Department at WPI for their inspiration, support, and guidance without which this project could never have been completed.

Most importantly, I would like to thank Professor Lindeman, without whom none of this would ever have been possible. Without his support and guidance, this opportunity would never have existed. From day one his dedication and vision have driven this project and helped make it a reality. His commitment to the project has been made clear time and time again by his personal investment in every aspect of the creation and continuation of the project.

Table of Contents

List of Figures

List of Tables

1. Introduction

A recurring topic in the field of computer graphics is that of computer vision. For many years, scientists have pondered ways in which the computer can extract meaningful information from real images. The range of topics in computer vision covers many concepts and techniques, from face recognition to performing accurate measurements. Computer vision concepts have found applications in a variety of fields, from robotics to medicine, military to entertainment. Often research in computer vision finds its practicality across a variety of applications, and research across application areas often builds on each other. With the growing number of cameras in place in everyday places (as security cameras or otherwise), and increasingly powerful computer hardware becoming affordable, it would seem that the possible applications of computer vision are only increasing.

While cameras allow these events to be captured from a single or handful of perspectives, they have their limitations. Representing a three-dimensional scene as a series of two-dimensional images inherently fails to properly record information due to occlusions, lighting problems, and the like. The rise of interactive three-dimensional applications has certainly shown the value of being able to actively control the perspective from which a scene is rendered. Similarly, movie directors rarely record a scene using a single camera angle – typically a variety of perspectives are used. From this collection of cameras, the final film is produced by selecting the perspective for each part of the scene that best conveys the information and emotions that the director is trying to convey. Certainly being able to apply this kind of cinematography to interesting events that occur in everyday life could prove to be a valuable tool for understanding and interpreting interesting situations.

It is this concept from which Cinematized Reality was formed. The goal of the project is to devise a system capable of capturing unexpected moments and to create movies from them using cinematic concepts. The system should be implemented in an unobtrusive way and be affordable for office and hospital environments. The system has potential applications in entertainment, telepresence, and medicine. The knowledge gathered by this system could be used to make visual content based on the real world more appealing, or it could be used to enhance our knowledge of complex situations, due to its ability to show a captured scene from any angle. For example, if such a system were integrated into a hospital environment, it could be used to accurately record what hospital workers do in complex situations, which could lead to better health care and accident prevention.

While these “free-viewpoint” systems have been constructed previously for a variety of purposes, none have been constructed that would be well suited to this type of application [3][2]. Previous systems have often required a large number of specialized cameras or other custom equipment that is costly and would interfere with the usability of the space being recorded. Cinematized Reality seeks to generate film-like footage of a scene from a small collection of standard video cameras.

In fact, the Cinematized Reality system has had this capability prior to my participation in the project. However, the processing involved in generating the free-viewpoint video was prohibitively slow, taking nearly a minute of total processing time per frame. Because of this, critical parts of the system needed significant optimization. This paper presents the process by which the system was optimized using commonly available graphics hardware. However, first some key concepts in computer vision and the programming of graphics hardware are introduced. Additionally, the key parts of the original implementation of Cinematized Reality system are presented before introducing the GPU-based optimizations. Finally, conclusions are drawn from the results in the form of performance benchmarks and their analysis.

2. Computer Vision Background

For the past 30 years, computer vision has been an active topic for computer science research. Computer vision, by most definitions, deals with the extraction of 3D information from multiple 2D images. In the case of Cinematized Reality, computer vision refers to the process of detecting foreground objects, reconstructing models of their shape, and then using these models to render the objects from novel viewpoints. While many techniques have been proposed for each of these steps, only a few will be presented here.

Before any information can be taken from captured images, however, certain information must be known about the cameras recording the scene. Thus, a model of how the camera works must be constructed. For the purposes of this paper, we can assume a 'pinhole' camera model shown in Figure 1, which is similar to the way in which 3D scenes are rendered onto 2D image planes. In this model, light enters the camera and is focused by a series of lenses to a single focal point. Behind this focal point lies the Charged Coupled Device (CCD) which converts the light energy that hits it into a measurable electrical signal [18].

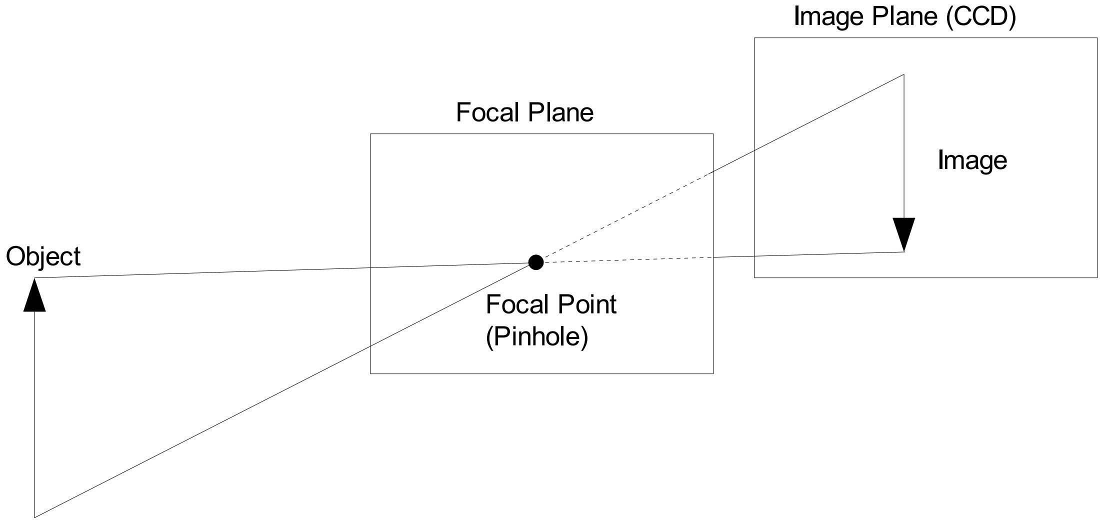


Figure 1. Simplified pinhole camera model

The distance between the focal plane and the image plane is the effective focal distance. It is important to note that this is not necessarily the same as the focal distance of the lens used – the effective focal distance will also depend on other optical components of the camera. Additionally, the projected image will not occupy the entire CCD, and the mapping between CCD cells and output pixels will not be 1:1 [21]. Additionally, lenses have an inherent tangential and radial distortion that have to be compensated for [6]. As such, parameters must be calculated for each camera to complete the model so that a mapping can be calculated between the 3D coordinate space and the camera's 2D image coordinate space. The parameters to compensate for the variables discussed above are referred to as intrinsic parameters, as they deal with the internal properties of the camera itself, and need only be calculated once for each camera. Extrinsic parameters refer to the camera's position and orientation relative to a fixed coordinate space, and must be recalculated whenever the camera is moved [6].

2.1 Camera Calibration

Techniques for calculating these parameters for off-the-shelf cameras usually rely on calibration cards or surfaces with easily identifiable points, such as the corners of black squares in a grid [19]. Recent techniques have allowed calibration to take place with a minimal amount of setup or precise knowledge of calibration pattern position or orientation [23]. However, extra care has to be taken when a collection of cameras is being calibrated to a global coordinate system, as each calibration point must be used by every camera that can see it. While calculating each camera's parameters based on the fixed calibration points is a well-known technique, it is also possible to calculate these parameters by estimating a projection matrix for the camera and then deriving the parameters from this estimate [18]. Because this is computationally simpler, this is the method used by Cinematized Reality.

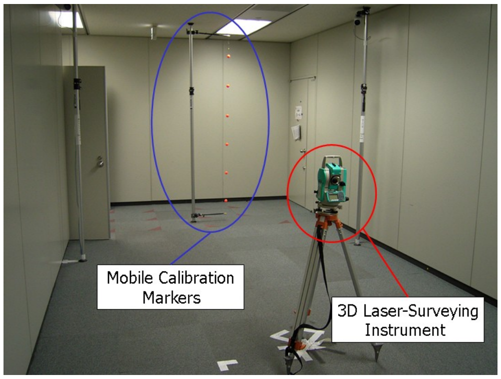


Figure 2: Calibration markers [7]

2.2 Object Reconstruction

Several techniques for obtaining information about the shape of an object from captured images exist, each with benefits and drawbacks. Many systems propose a system to calculate depth information based on the differences of images captured by a pair of side by side cameras, referred to as stereo depth cues[10][16]. This is useful for robotics applications, where a limited number of cameras are mounted on a movable platform, but due to the lack of a comprehensive set of captured viewpoints, the resulting model is only accurate when rendered from a narrow range of viewpoints. Most of the remaining techniques project image data from a series of cameras into a 3D space to generate the required shape information [5]. When the silhouettes of foreground objects are projected in this manner to carve out the space, this is referred to as "shape-from-silhouette" (Figure II). Alternatively, it is also possible to calculate the shape of an object from depth or shading information calculated from calculated images [5], but this often relies on specialized hardware and lighting.

A few papers have also presented methods for rendering a scene from novel viewpoints without creating a model of the scene as a whole. These image-based-rendering systems use the captured images directly to render the scene. While this rendering method does not depend on scene complexity, the rendered viewpoint cannot lie too far from the cameras, resulting in systems that require a large number of cameras [21].

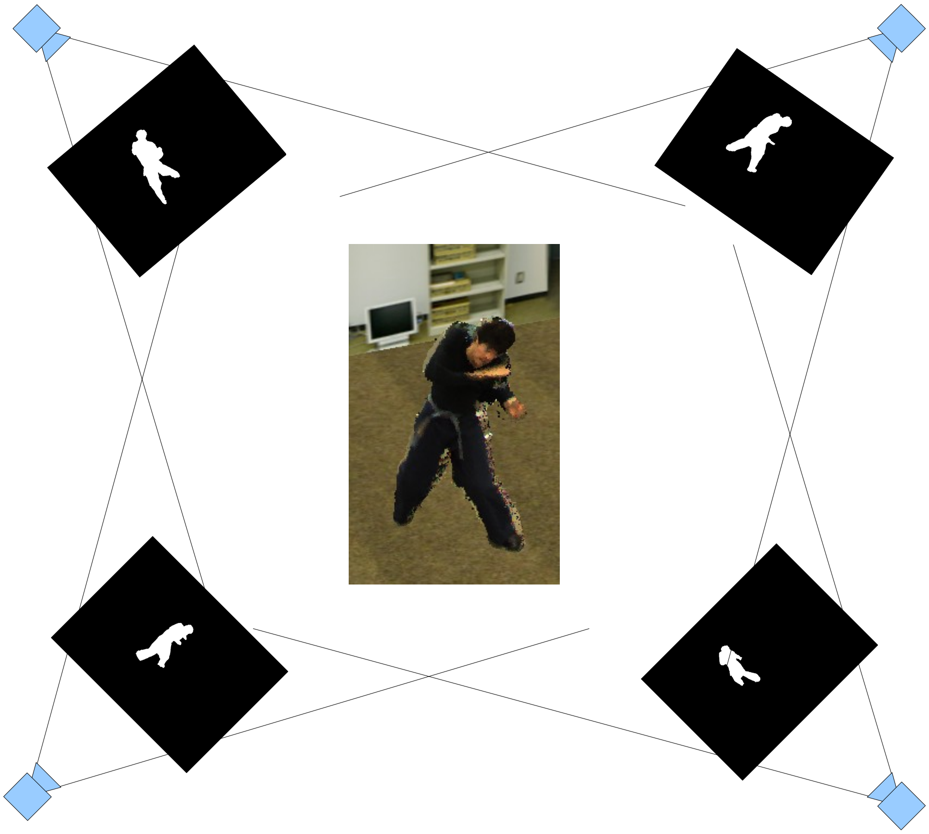


Figure 3. 3D Reconstruction using shape-from-silhouette

The shape-from-silhouette (SFS) technique also has its limitations. As Laurentini points out, SFS is unable to model concave surfaces accurately [11], Kim et al. showed that SFS is sensitive to segmentation errors, and present a method for correcting for this [8]. Despite these limitations, ATR decided that utilizing a SFS technique along with advanced segmentation would provide the best results. Details of the final system are found in Section 3.

3. Background on the GPU

3.1 Introduction to GPUs & architecture

With the rise of computer games, interactive media, and the availability of 3D rendering and computer graphics on consumer-level hardware, the graphics card has become an increasingly important and powerful component. In recent years, it has evolved to support a large number of computations per second, as well as gaining the ability to be programmed using specialized programs referred to as “shaders”. While the CPU is designed to support all kinds of computation, the GPU has been designed specifically for processing streams of information. This creates some limitations on the types of computations the GPU can perform, and creates some additional challenges for writing programs for it. However, it allows the manufacturers to optimize the hardware for a specific subset of computational tasks. As such, the GPU contains hardware acceleration for a number of functions that would be computationally expensive to perform on a CPU. Additionally, the GPU has an inherently parallel architecture, meaning it is able to process many vertices, polygons, or fragments simultaneously. Due to this structure, the latest GPUs are very complex – NVIDIA estimates that its 8800 series of GPUs has

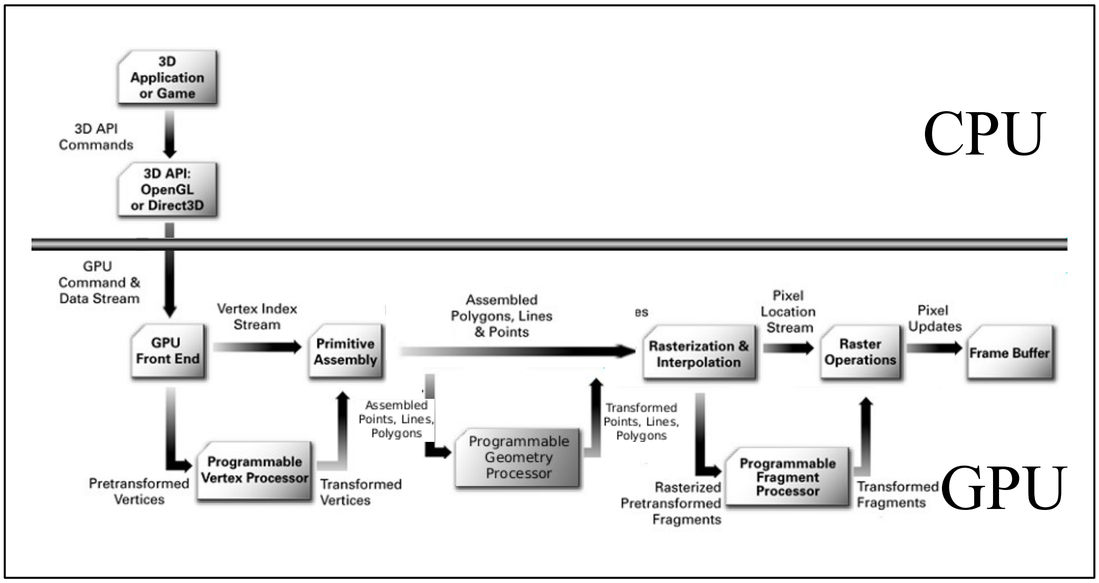

3D Application or Game
3D API Commands
3D API: OpenGL or Direct3D
CPU
GPU Command & Data Stream
GPU Front End
Vertex Index Stream
Primitive Assembly
Assembled Polygons, Lines & Points
es
Rasterization & Interpolation
Pixel Location Stream
Raster Operations
Pixel Updates
Frame Buffer
Pretransformed Vertices
Programmable Vertex Processor
Transformed Vertices
Assembled Points, Lines, Polygons
Programmable Geometry Processor
Transformed Points, Lines, Polygons
Rasterized Pretransformed Fragments
Programmable Fragment Processor
Transformed Fragments
GPU

Figure 4: GPU pipeline with shaders [14]

approximately 681 million transistors [20].

With the advent of programmable graphics hardware, the architecture of the GPU has changed significantly. The latest video cards now contain many “stream” processors that can be dynamically assigned to different parts of the rendering pipeline, instead of being dedicated to a specific task. Each of these steam processors is built on the graphics-centric instruction set of the earlier programmable GPUs.

3.2 Integration of GPU acceleration into a computer graphics pipeline

GPUs provide hardware acceleration for many of the common tasks in the graphics rendering pipeline, such as transformation, lighting, texturing, and rasterization. Modern GPUs, however, have allowed for custom programs to replace part of this fixed-function pipeline in three key areas, each of which has a specific shader type associated with it. These are referred to as the vertex, geometry and fragment (or pixel) shaders. Figure 4 shows the GPU pipeline and the integration of the three types of shaders.

The vertex shader replaces the vertex transformation section of the pipeline. As such, the vertex shader must at a minimum take global vertex position data and transform it to camera or eye space. In the fixed function pipeline, this is done as a matrix multiply with the modelview matrix. The vertex shader can choose to replicate this functionality as well as add other per-vertex operations, or it can leave this transformation for the geometry shader.

The geometry shader is the latest addition to the graphics pipeline and has no fixed function equivalent. The geometry shader is called after the vertex shader, and receives primitives (lines, polygons, or points) as inputs. The geometry shader is utilized to create or modify geometry at the primitive level on the hardware, which can be faster than creating the modifications on the CPU. It functions in camera or “eye” space (provided the transformation was performed earlier), and has loose requirements for what its inputs and outputs are. While the type of primitive to be used as inputs and outputs must be defined, the geometry shader is free to output many, one, or no output primitive(s) for each input primitive. Because it is a new feature, geometry shaders are currently only supported on the latest GPU's and graphics APIs.

Finally, the fragment (or pixel) shader performs its computations on each pixel being drawn on screen. This allows for more complex and dynamic lighting and texturing effects, such as bump mapping. Due to its position in the rendering pipeline, the fragment shader cannot modify its position – typically only color and occasionally depth are used as outputs. However, because the fragment shader is typically called once for each pixel on the screen, the GPU is optimized to support faster memory accesses for fragment shaders.

3.3 Introduction to GPGPU programming concepts

The addition of programability to graphics hardware has made it possible to use GPUs for computations unrelated to the traditional 3D pipeline. While the architecture is designed with rendering in mind, the hardware acceleration of matrix and vector operations, along with the parallel execution capabilities, allow for performance benefits in a variety of applications, such as audio processing, physics simulation, or artificial intelligence computation. While ideas about data structures, program flow, debugging, and other factors have to be re-evaluated when dealing with GPU programming, the benefits have sparked a discussion of General Purpose computation on the GPU, or GPGPU [gpgpu.org].

The primary data structure of the GPU is the texture. While this normally represents visual information, it can be thought of as a general-storage, multi-dimensional array. Textures can be one, two, or three dimensional, and can hold up to four values at each texture array location, or texel.

Textures can be used as both inputs to and outputs from GPU programs formed by a set of shaders. One common GPGPU technique is to set up the render target so that each fragment that gets processed corresponds to a single texel on the output texture. This ensures that the result of each computation will be stored to exactly one location in the texture. Because this process of “rendering” always results in a two-dimensional image, three-dimensional data is processed one layer of texels at a time.

Similarly, the ways in which the computation is processed is slightly different from that of the CPU. Because GPUs have multiple shader processing units (referred to as stream processors), each shader operates on processing one element at a time (a vertex, polygon, or fragment depending on the type of shader), but this is happening in parallel with other stream processors. Thus, instead of each data element being processed in order, data flows into and out of the GPU in parallel. One of the challenges with this programming model is that each shader must be able to operate on a piece of data independently of any processing that is occurring on the other pieces. For example, during the execution of a fragment shader, the intermediate results of neighboring fragments cannot be read. If these results are needed, the process is usually broken up into two passes, and the intermediate results are stored as an output texture of the first pass, and passed as an input to the second pass.

3.4 Advantages and disadvantages of using GPUs

While the GPU is a powerful piece of hardware, it also has its drawbacks. First, the parallel nature of the GPU allows data to be processed faster, but only when the computations that must be done can be accomplished on each piece of data in isolation. In this sense, the execution of a GPU program is like a pipeline – data flows in and out and is changed in some way in the process. Second, the different types of shaders allow for data to be processed in new and interesting ways, but processing does not always fit into the three categories that the shader types provide. Many applications can perform their computations in a single fragment shader, providing one output value for each input value, but this does not match every possible application.

Additionally, the specialized hardware develops bottlenecks in processing in ways that most programmers are not used to. For example, matrix and vector operations are accelerated by hardware and execute very quickly, but accessing memory in the form of texture memory is significantly slower than a CPU accessing main memory. Thus, contrary to the CPU, it might be faster to recompute certain values rather than perform a lookup. Finally, programming for the GPU requires the additional steps of sending data from main memory to graphics memory, and retrieving the results back to main memory over the video card bus. The PCI-Express bus allows for a high data bandwidth, but this still causes a significant overhead in processing time. Specifically, retrieving results from graphics memory has proven to be more expensive than sending data. This makes sense from a graphics perspective, as textures are sent to the GPU but typically are not read back, but this is not the case in the realm of GPGPU.

4. Cinematized Reality prior to GPU optimization

4.1 Purpose of Cinematized Reality

The purpose of the Cinematized Reality system is to capture and recreate unexpected moments in everyday life while allowing the viewer to change their viewpoint freely. While a single camera is able to capture the essence of an event from one angle, this is often insufficient for a number of reasons. A single camera will reduce the scene to a moving two-dimensional image, and does not store any depth information. Along with the problems of occlusion and perspective foreshortening, a single camera will only capture a small piece of what it is viewing. By using multiple cameras, this information is able to be recovered, and a full 3D model of events can be reconstructed. Much work has been done to allow computers to recreate 3D models in a variety of ways, but every system has its limitations. A number of inexpensive systems relying on stereoscopic cameras have been developed, but are unable to capture an entire scene due to the inability of determining the full shape or texture of an object. By contrast, many specialized systems have been developed to capture events from all angles using many specialized cameras. However, the most effective of these systems rely on a large number of expensive cameras and very specific lighting conditions. As a result, such systems are impractical in everyday environments. The purpose of Cinematized Reality is to be able to capture and reconstruct these events using a limited number of off-the-shelf cameras in a normal office environment.

4.2 Hardware used in Cinematized Reality

In order to accurately capture a scene, multiple off-the-shelf cameras are set up around the area of interest, directed towards a center point. In the case of the current system, up to nine cameras are used, each capable of capturing images at a resolution of 1024x768 at a rate of 30 frames per second (fps). Once the cameras are set in place, their intrinsic and extrinsic parameters were calculated and recorded. Because the cameras do not move, rotate, or zoom, these parameters need only be calculated once. Each camera is attached via a IEEE 1394 (firewire) connection to a computer which records the frames to disk. Due to the complexity of the computations required, these frames are transferred to another computer for offline processing. All offline processing was completed by a standard office PC with a 2.4 GHz Core 2 CPU, with 2 GB of RAM, and a NVIDIA GeForce 8800 GTS graphics card.

4.3 Software

The image processing computer is running Windows XP with Service Pack 2. The offline processing software was compiled under Microsoft Visual Studio 2003 using Intel's C++ compiler. Libraries utilized include OpenGL, a freely available rendering library; OpenCV, an open-source computer vision library; GLEW, an extension loader for OpenGL; and cg, NVIDIA's GPU language and compiler.

4.4 Program Structure

Processing is divided into three major steps: segmentation, modeling, and rendering as shown in Figure 5. Originally, all three steps were computed on the CPU. After optimization, the later two steps were performed on the GPU, with segmentation still performed on the CPU. Segmentation operates on each image captured from a single camera individually, and produces a segmentation mask, indicating the presence or absence of a visible foreground object at each pixel. After this is computed for each camera for a single frame, these images are passed onto the voxel modeler. Using these masks along with the camera calibration parameters, the voxel modeler carves the voxel grid by checking each voxel against all segmentation masks, to determine whether it is “filled” or empty space.

Once the voxel map has been completed for a frame, the renderer uses the voxel data to construct a representation of the model from an arbitrary viewpoint, using the original captured images as textures. This final rendering process has been implemented in three different ways. First, a cube is drawn for each filled voxel, creating a model that is “true” to the structure of the data. Secondly, a microfacet rendering function was created, which draws a single four-sided polygon (quad) for each filled voxel. Thirdly, a true modeling function was created based on the “Marching Cubes” algorithm which defines polygons in between data points in the voxel grid based on whether they have been determined to be “inside” or “outside” the surface of the model [12]. Details on these rendering processes, their implementations, benefits, and drawbacks are provided in the optimization overview in Section 5.

Regardless of which rendering process is used, the model is textured by mapping the original images captured by the environmental cameras to the vertices of the model. In order to do this effectively, the renderer must choose which camera to use for each voxel or point in the model. Initial selection is done by checking the angles between the cameras and the voxel in question to determine

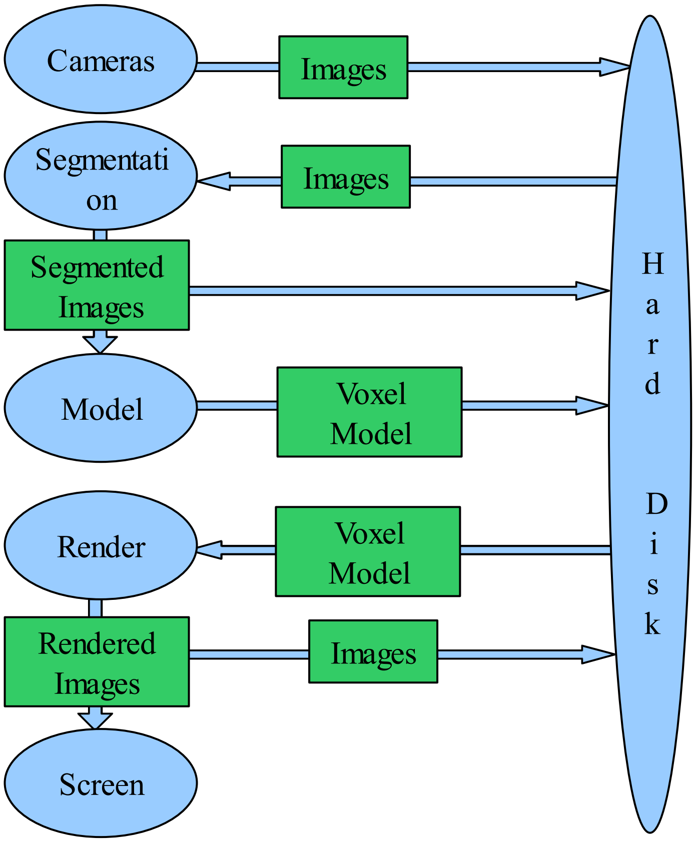


Figure 5. Cinematized Reality Data Flow Model

which camera has the viewpoint most closely related to the current viewpoint. This choice can be overwritten, however, if it is determined that the selected camera does not “see” the current voxel because it is occluded by another part of the model. Once the appropriate camera is selected for a polygon, the texture coordinates for each vertex are calculated before rendering.

4.5 Data Flow

While the processing involved in the Cinematized Reality system is not simple, the flow of data for each frame to be computed is very linear. Frames recorded by the environmental cameras are transmitted over the network to central storage on a hard disk for offline processing. These images are read in by the segmentation routine, which passes image masks to the modeler and back to the hard disk. The modeler uses these image masks to construct a voxel mask which is then written to disk. This is done for each frame of data before moving on to the rendering pipeline. The renderer reads in the voxel model and original images for each frame, and generates new images based on a virtual camera. These new images are shown on the screen for instant feedback, as well as written to disk as the final output. While the the implementation of each step has changed in the GPU-optimized version of Cinematized Reality, the basic functionality of each step has remained the same.

4.6 Results

The original Cinematized Reality system was able to produce images and videos of good quality from nearly any angle as presented in Figure 6. As such, the goal of this MQP was not to improve the *quality* of the output. Instead, the goal was to improve the *performance* of the system to more-acceptable speeds. The original system would take ~10 seconds for segmentation, ~25 seconds for modeling, and ~1 seconds to render a single frame of data. These numbers were further inflated by the need for constantly saving and loading intermediate results to and from the hard disk. While it is not necessary for the system as a whole to run in real-time, these results meant that 10 seconds of video recorded at 30 frames per second would take over three hours to process. In order to improve the performance of the system, some of the processing load was moved to the GPU.

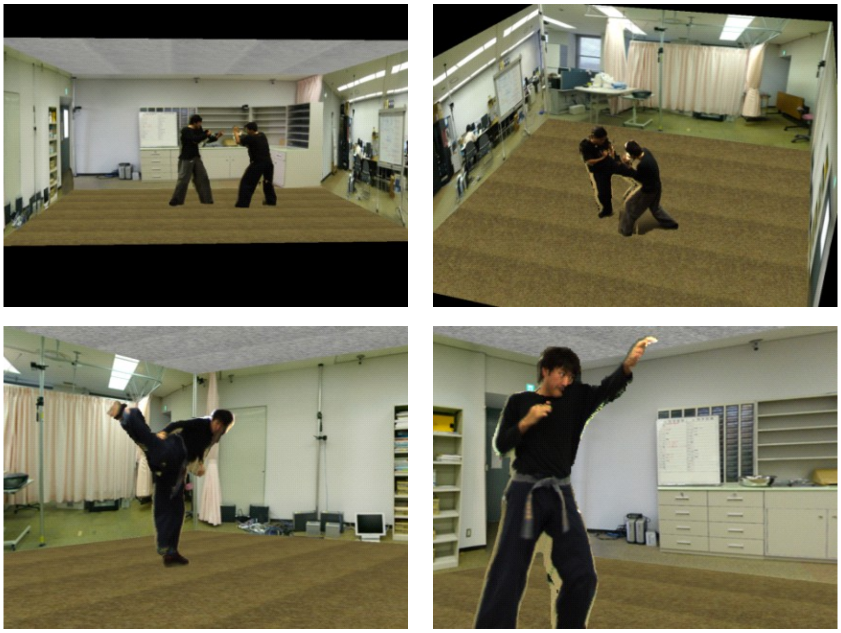


Figure 6: Original Cinematized Reality Result [8]

5. Cinematized Reality Optimization

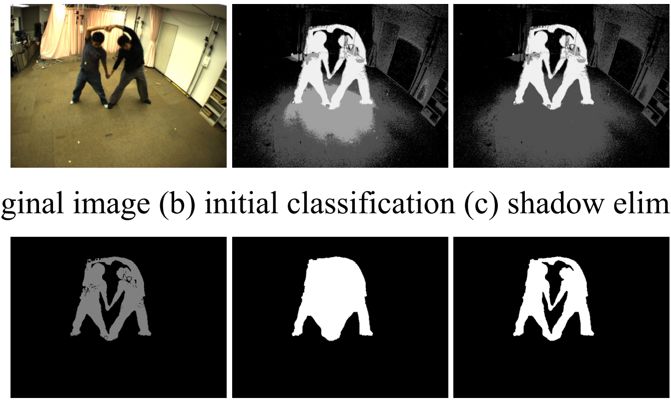
5.1. Initial Plan

In approaching the task of optimizing the offline processing of the Cinematized Reality system, the performance of the system as a whole was analyzed first. The majority of the time was being spent in computing the segmentation, model reconstruction, and rendering itself, as these tasks were computationally expensive. It was decided that priority would be given to the reconstruction and rendering processes. Additionally, the program was initially configured to write many intermediate steps to disk, breaking the process up into several different steps. Optimization would take place primarily by moving algorithms onto the GPU. At first, the steps would be optimized independently of each other However, later these steps would be combined by using the intermediate results directly from graphics memory. This would result in further performance gains for the system as a whole.

5.2. Optimization Techniques Utilized

5.2.1. Segmentation

While the segmentation process was not optimized as it was still under refinement, the quality of the results of segmentation have a large impact on the modeling process and the overall output quality. Additionally, [8] presents a method which could be used to detect and eliminate some of the segmentation errors in future versions of the system. As such, it is important to understand the segmentation process in the context of the system as a whole. Figure 7 presents the individual steps of the segmentation process.



(a) original image (b) initial classification (c) shadow elimination

(d) labeling (e) Silhouette extraction (f)final mask

Figure 7: Stages of Segmentation[6]

In order to extract background and foreground regions from a video sequence, first a model of the background must be constructed from a number of frames of video with no foreground objects. Both luminance and color components are calculated, as luminance is sensitive to shadow, where color components are sensitive to noise. Hue is used as the color component of the model according to Eq. 1, where luminance is calculated directly. Both components are modeled from a number of empty frames from which the model is calculated utilizing a Laplace distribution as detailed in [6].

$$I = \max(R, G, B)$$

$$S = \begin{cases} (I - \min(R, G, B)) / I & \text{if } I \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H = \begin{cases} (G - B) \times 60 / S & \text{if } I = R \\ 180 + (B - R) \times 60 / S & \text{if } I = G \\ 240 + (R - G) \times 60 / S & \text{if } I = B \end{cases}$$

$$\text{if } H < 0 \text{ then } H = H + 360$$

Eq. 1 [6]

Initial classification of pixels is done by finding the difference between the luminance component of the current frame and that of the background model. The resulting value is used to place the pixel in one of four categories: Reliable Background, Suspicious Background, Suspicious Foreground, and Reliable Foreground, as shown in Eq. 2. $L_i(p)$ is the luminance of the current frame, L_g

is the luminance of the background model, and σ is the standard deviation of the background model. K_1, K_2, and K_3 are discovered through training the algorithm against ground-truth (hand-segmented) images.

$$\begin{cases} BD < K_1\sigma(p) & \Rightarrow \text{(a) Reliable Background} \\ K_1\sigma(p) \leq BD \leq K_2\sigma(p) & \Rightarrow \text{(b) Suspicious Background} \\ K_2\sigma(p) \leq BD \leq K_3\sigma(p) & \Rightarrow \text{(c) Suspicious Foreground} \\ K_3\sigma(p) \leq BD & \Rightarrow \text{(d) Reliable Foreground} \end{cases}$$

Eq. 2 [6]

At this point, many of the background pixels are labeled as suspicious foreground, due to the difference in luminance caused by shadows. These regions are merged with the suspicious background regions using the color component of the background model. While shadows will change the brightness of an area, they will generally have little effect on the hue, making this a reliable way of detecting false foreground pixels due to shadows.

From here, each region of the segmentation mask is labeled with a unique identifier. Due to areas of noise, there are a number of small regions labeled as foreground regions that do not contribute significantly to the shape of the object. As such, any regions below a predefined size (in pixels) are eliminated from the mask. Next, a profile extraction technique is applied based on the work of Kumar [9]. A one pixel thick line is draped elastically over the model to smooth the edges and fill in gaps. This process is performed from all four edges of the screen for each labeled region, smoothing the edge of the silhouette, and filling in any real holes inside the object. Thus, in the end any regions of reliable background pixels larger than a certain tolerance are added to the background. The results of each step of the segmentation are shown in Figure 7.

The resulting segmentation masks are not perfect, but show the shape of any foreground objects. Most segmentation errors involving pixels incorrectly labeled as foreground will not have an effect on the reconstructed model, due to the algorithm used to construct it. However, overzealous segmentation errors create noticeably incomplete models and can cause occlusion errors in texturing [8].

5.2.2. Visual Hull Reconstruction

Visual hull reconstruction is the process by which a scene is modeled in 3D based on captured images of the scene from various angles [11]. Cinematized Reality utilizes a SFS algorithm on a voxel grid in the target space. Empty voxels in this grid are culled by projecting the silhouette of each camera into the captured space. Voxels that lie within an object will be in the foreground areas of each silhouette and will be labeled as "solid." However, voxels that are empty are contained within the background section of at least one camera's silhouette and will be culled from the voxel model [7].

For example, a series of cameras C_m (m=1,...,N), where N is the number of cameras, is oriented on an area and records a scene. Silhouettes S_m are then calculated for each frame of each camera's video using each camera's calibration parameters in a projection matrix Pm. Any arbitrary point p that lies within a voxel V_n can be projected onto C_m's image plane using P_m. If the volume that V_n occupies is part of an object in the scene, then p should be located in the foreground region of S_m when projected onto the image plane of C_m. Thus, if any point p in V_n is projected into the background region of any segmentation mask S_m, then V_n must be empty and is culled. Using this model, we can construct a visual hull of any object recorded in this system by checking each voxel against each segmentation mask using the appropriate calibration parameters to determine whether it is solid or empty [6].

In order to implement this on the GPU, we represent the voxel grid as a 3D texture. The result is computed using a fragment shader that iterates over the voxel texture one "slice" at a time. This is achieved by binding one layer of the texture at a time to a framebuffer object (FBO) the same size as the layer, and rendering a quad to occupy the full size of the framebuffer. As a result, the fragment shader is called once per voxel (represented as a texel), due to the fact that the resolution of the framebuffer matches the resolution of the slice of the voxel texture.

Figure 8 shows pseudo-code for the fragment shader used for reconstruction. Experimental results have revealed that the GPU algorithm produces the same output as the CPU-based version. Performance benchmarks are provided and analyzed in Section 6.

```
Modeler(indices) {
        position  = CalculateGlobalPosition(indices);
        effective_camera = 0, actual_camera = 0;
        empty = false;
        for each camera C {
                imageCoordinates = projectPoint(position, C);
                if (imageCoordinates.isValid()) {
                        color = SegmentationMasks[cam].getPixel(imageCoordinates);
                        if (color == 0.0) {
                                 empty = true;
                                break;
                        }
                }
        }
        if (empty)        return black;
        else              return white;
}
```

Figure 8: Visual Hull Reconstruction Fragment Shader Pseudocode

The fragment shader takes the camera calibration parameters and the segmentation masks as inputs, as well as the layer number upon which it is rendering. The system also passes in the texture coordinates (in 2D) of the texel it is rendering to. From these parameters, the shader computes the 3D location of the voxel it is computing, and projects its position of the voxel is projected onto each camera's image plane. From this projected point, the shader computes the appropriate texture coordinates. If the texture coordinates lie outside of the range of the captured image, the camera is skipped because it cannot “see” the voxel. Otherwise, the shader checks the segmentation mask by using a texture lookup. If the result is a background pixel, the shader returns “empty”, represented by the color black. If the result is a foreground pixel, the rest of the cameras are checked in a similar manner. If the voxel is in the foreground of every camera that can see it, the shader indicates that the voxel is filled by returning a white texel.

5.2.3. Rendering

Two different approaches are implemented for rendering the voxel model. First, a polygonal mesh is constructed and rendered using a variant of the marching cubes algorithm [12]. Secondly, a microfacet billboarding technique was tried [4]. The texturing portion of the rendering process depends on camera selection, which is covered in Section 5.3.

5.2.3.1. Marching Cubes/Tetrahedra

The marching cubes algorithm generates a polygonal mesh representing a level set of a grid of datapoints. It operates on each cube in the grid, using the data points as vertices. It computes the intersection of the level set with the cube based on the vertices and generates polygons based on these intersections. Due to the fact that each vertex must lie either within or outside of the level mesh, there are only 256 possible configurations. If the algorithm is further optimized by allowing for rotations and reflections of the same configuration, there are only 14 unique configurations that require polygons to be generated (if all vertices lie within or outside of the level set, no polygons need to be drawn for that cube). Up to four triangles can be generated for each cube, depending on the configuration of the vertices. Unfortunately there is a minor ambiguity in one of the unique cases where a single configuration can be validly represented by two different configurations of polygons [13]. If the wrong one is chosen, this can result in “holes” visible in the generated mesh.

Implementing this algorithm is best done using geometry shaders, as the vertices can be passed in as inputs, and polygons can then be passed out and down the render pipeline. Unfortunately, the geometry shader can take a maximum of six vertices as inputs, so it is not possible use a full cube. The alternative is a variation called marching tetrahedra. Marching tetrahedra splits each cube up into five or six tetrahedra, and a similar algorithm is performed [17]. However, with only four vertices, there are only 14 possible polygon-producing configurations, which can be further reduced by eliminating redundant cases. An additional benefit of using the marching tetrahedra algorithm is that the ambiguous case seen in marching cubes is eliminated.

When applying this algorithm to visual hull reconstruction for Cinematized Reality, it is important to know that each voxel is specified only as inside or outside of the object. Hence, all intersections of the surface with the edge between the voxels produced by this algorithm will always lie at the midpoint of the intersecting edge.

In order to pass four vertices to the geometry shader, the shader is configured to take lines with adjacency information as inputs. The output of the geometry shader is at most to two triangles, as that is the maximum number of triangles the algorithm can produce for each tetrahedra. An edge lookup table is generated by hand ahead of time, and entry in the table lists the edges that should be used to generate the triangles for each configuration. The shader begins by calculating the index into this table based on the status of the four vertices. Then, the edge indices are retrieved from the table and the midpoints of all six edges are calculated. Finally, the geometry shader emits the vertices specified by the retrieved edge indices. Figure 9 shows the pseudocode for the geometry shader.

```
void marchingTetrahedraGS( solid[4], positions[4] )
{
        edgeIndex = calculateEdgeIndex(solid);

        edges[4] = edgeTable[edgeIndex];

        numEdges = lookupNumEdges[edgeIndex];
        if (numEdges == 0) return;

        points[6] = calculate_midpoints(positions);

        for each edge E
                useEdge = edges[2];
                if (useEdge == 0) continue;
                emitVertex(points[useEdge-1] : POSITION);
        }
}
```

Figure 9: Marching Tetrahedra Geometry Shader Pseudocode

The model produced by the geometry shader is then rendered to the screen with the fragment shader. The output matches the data in the voxel model well, but the algorithm executes too slowly, mainly due to the high level of tesselation across the entire model, irrespective of actual model complexity. While mesh-reduction techniques could improve the performance of this method, it was decided to utilize a computationally simpler algorithm.

5.2.3.2. Microfacet Billboarding

While marching tetrahedra will produce up to 12 triangles per voxel, microfacet billboarding represents each voxel with a single polygon (two triangles) [1]. The microfacet is defined as, “a slice which intersects the center of the voxel and is vertical to the viewing direction” [22]. This means that the normal for the microfacet will be rotated as the virtual viewpoint changes to ensure that the polygon always directly faces the viewpoint. Given a regular voxel grid with s units of distance between neighboring voxels, this means that each must measure at least $\sqrt{3}\cdot s$ units in each direction to ensure that no gaps are formed in between neighboring microfacets. While a polygonal mesh of the data is never formed, microfacet billboarding appears to produce the same shape as the more computationally expensive marching tetrahedra algorithm regardless of the angle at which the model is viewed. This is likely related to the grid-aligned nature of the data generated by the visual hull reconstruction algorithm, which makes the added computation of generating a full polygonal mesh unnecessary.

The rendering process for microfacet billboarding is also well suited to the geometry shader for two main reasons. First, it allows only a single point (the center of the voxel) to be passed to the video card per voxel. Second, geometry shaders work in camera space, which allows for trivial computation of the vertices required to draw a quad whose normal is parallel to the view vector. In fact, in the absence of texturing, the algorithm for microfacet billboarding itself is trivial. The geometry shader takes in the position of the center of the voxel in camera coordinates and adds fixed offset values to the x and y values to produce the vertices of the quad. Consequently, this algorithm runs many times faster than the marching tetrahedra algorithm. The pseudocode is in Figure 10.

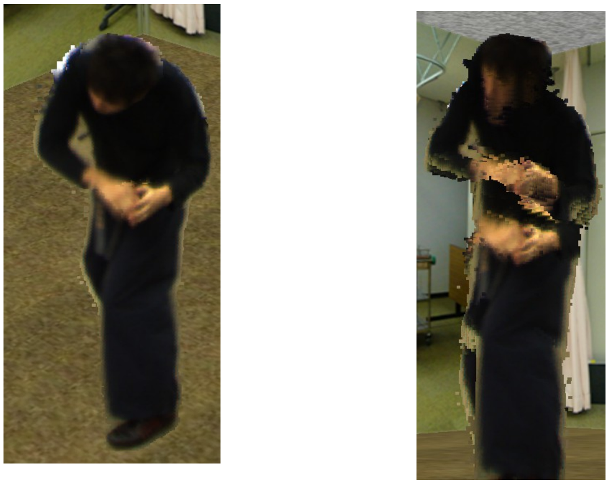
```
microfacet(position) {
        emitVertex(position + (-offset.x, offset.y, 0));
        emitVertex(position + (-offset.x, -offset.y, 0));
        emitVertex(position + (offset.x, offset.y, 0));
        emitVertex(position + (offset.x, -offset.y, 0));
}
```

Figure 10: Microfacet Billboarding Geometry Shader Pseudocode

5.3. Additional Issues & Optimizations

5.3.1 Camera Selection & Occlusion detection

In order to texture each microfacet from the captured camera images, there must be a way to select which camera to use in a given situation. The initial selection criteria is the viewing angle of each camera – the texture is more accurate the closer the camera's viewing angle is to the viewpoint's viewing angle of any given voxel. However, this is not sufficient criteria, as there are other factors that would prevent a camera from being able to see a voxel. For example, the camera may be oriented in such a way that the image coordinates of the voxel would place it outside of the camera's captured image. Additionally, when non-trivial objects are recorded by the system, the model generated often occludes itself in one or several of the captured camera images. For example, when the person being recorded holds an arm up between one of the cameras and the body, the image of the arm would be used to texture not only the arm itself, but also part of the body. For example, Figure 11 demonstrates this program. Figure 11(a) displays the scene rendered from the camera's angle, while Figure 11(b) shows the same scene rendered from a slightly higher angle. Notice that the texture of the hand is



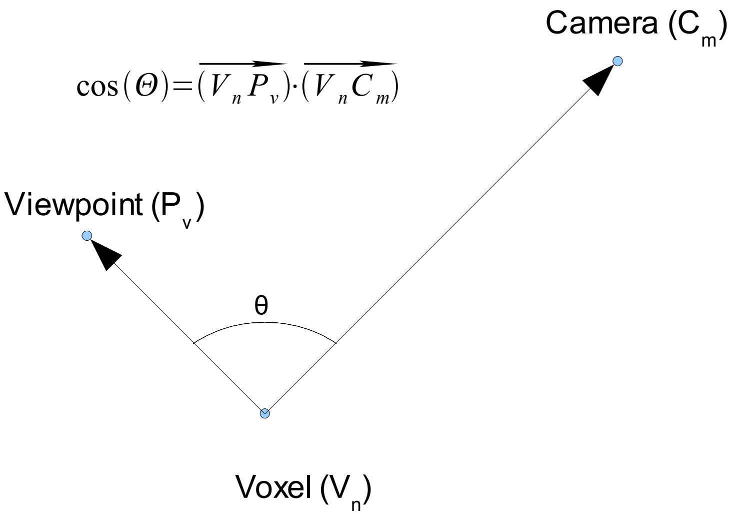
(a) hands in front of body (b) hands projected onto body

Figure 11: Occlusion Problem

rendered on the chest as well. Proper segmentation and occlusion detection can help to alleviate this problem.

As a result, the system must take all of these factors into account before selecting which camera to use (Figure 12). Given a voxel V_n, viewpoint P_v, and camera C_m, the system computes the difference in the viewing angles of the viewpoint and the camera by computing the angle between the vectors V_nP_v and V_nC_m according to the diagram and formula in Figure 12.

This angle serves as the basis for a “visibility score” given to each camera. From this point, the visibility score is modified by projecting the coordinates of the center of the microfacet onto the image plane of the camera. If the center lies outside of the image, then the camera's visibility score is incremented to an arbitrarily large value to prevent its use. Finally, occlusion is accounted for. However, due to the differences in the programming models used in CPU and GPU programming, this must be done in different ways.



$$\cos(\Theta) = \overrightarrow{(V_n P_v)} \cdot \overrightarrow{(V_n C_m)}$$

Figure 12: Difference in viewing angles between a real and virtual camera

5.3.1.1. Bresenham

Given a viewpoint P_v, a voxel V_n is considered occluded if there exists a voxel W, such that W lies between V_n and P_v. Thus, one method of checking for occlusion is to use a three-dimensional version of Bresenham's line drawing algorithm to obtain the coordinates of every voxel in between V_n and P_v [1]. Bresenham's line drawing algorithm uses pure integer math to draw the ideal line between two points by tracking an error term as the algorithm steps along the line. If applied to the voxel grid, the pixel locations become the voxel locations between V_n and P_v. If any of the voxels checked is solid, then V_n is occluded, and the rest of the cameras are checked until the best non-occluded camera is picked. Due to the fact that the line drawing algorithm works in pure integer math, this algorithm runs quickly, but requires fast access to the voxel model [15]. This is not a problem when the computation is being performed on the CPU, as it has easy access to main memory. However, when this algorithm is run on the GPU, each access to the voxel model is a texture access, which is the slowest operation a GPU can perform, especially when it must occur in a vertex or geometry shader. While a few texture lookups can be performed with only a little overhead, the number of lookups required for the Bresenham algorithm makes it prohibitively slow.

5.3.1.2. GPU depth texture

The GPU version of the rendering algorithm uses a novel method to avoid many texture lookups by precomputing depth maps for each camera. This depth map contains the depth from the camera to the closest voxel at each pixel location across the image. This is accomplished by rendering the model from the perspective of each camera using the microfacet billboarding technique. When this rendering pass is performed, the depth between the camera and the center of the voxel is used as the color and the depth of the microfacet. Depth sorting ensures that each pixel of the resulting depth texture will contain the color of only the closest microfacet rendered at that location.

When the scene is rendered from the perspective of the virtual camera, the vertex shader sorts the cameras based on their visibility score. At this point the visibility score is based off of the camera's viewing angle and its ability to see the voxel, but does not account for occlusion. The vertex shader passes the indices of the best three cameras on to the geometry shader, which then calculates the texture coordinates of the four vertices of the microfacet for the best camera. It uses these coordinates to compare the distance from the camera to the closest voxel to the distance from the camera to the voxel being rendered. If all of the vertices are within an acceptable tolerance of the closest voxel, then that camera is used. If any of the vertices is significantly farther than the distance retrieved from the depth texture, however, the voxel is considered occluded for the current camera, and the other cameras are checked in order until one is found that is not occluded. If all three of the best cameras are occluded, there is not sufficient visual information to texture the microfacet, but the best camera (based on the visibility score) is defaulted to for the sake of visual consistency.

5.3.2. Bus Transfer speed

Despite the recent advances in graphics card buses with the advent of PCI-Express, the time to transfer data to and from graphics memory is still a significant bottleneck to system performance. Specifically, retrieving data from graphics memory is very time consuming. Sending the captured camera images and segmentation masks takes a non-trivial amount of time, but retrieving the voxel map, depth textures, and final render take much longer. In order to alleviate this problem, the GPU version of the system is arranged to minimize the number of data transfers to and from video memory. In fact, by leaving intermediate results in video memory, the only result that has to be retrieved is the final render.

Figure 13 demonstrates the flow of data through the various rendering passes involved in voxel reconstruction and rendering. Camera images and segmentation masks are passed into the visual hull reconstructor which outputs the voxel model. The depth texture generator then uses this voxel model to generate the depth textures for each camera. Finally, the rendering pass uses all of the data in video memory to render the scene from the viewpoint of the virtual camera. While the voxel map is no longer able to be saved to disk using this model, the performance of the GPU-optimized system is able to render the scene from segmentation masks at interactive speeds, eliminating the need to store the voxel map on disk.

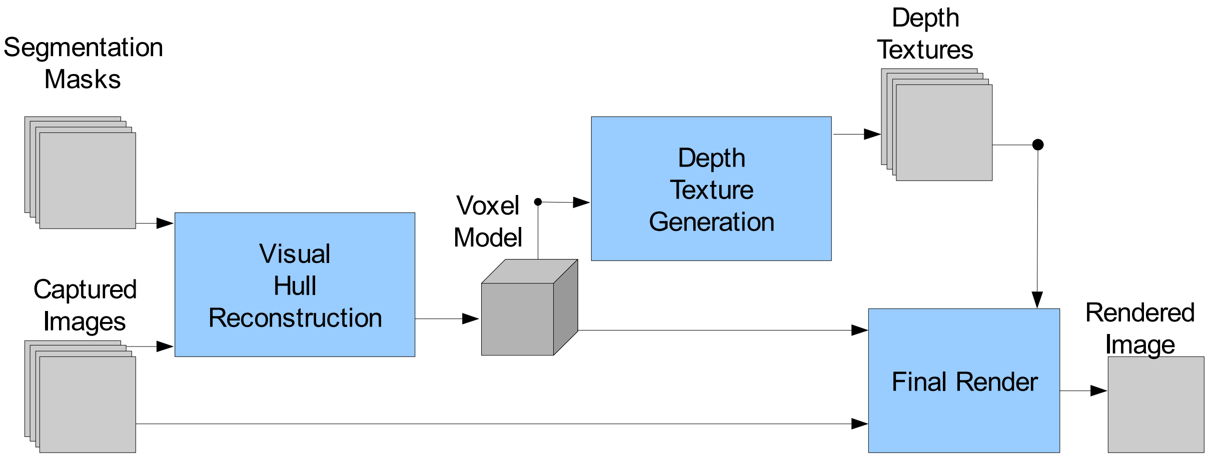


Figure 13: Data Flow Diagram for GPU-Optimized Pipeline

In addition, each render of the voxel model requires the coordinates of every voxel to be passed to the GPU. This occurs once for each depth texture generated and once for the final render. Because each position is stored as three floating point values, each position is 12 bytes long. In a system with seven cameras and a voxel grid of 300x200x300, this means that 1.6 gigabytes are transferred to the video card each frame in voxel positions alone. In order to reduce the amount of data transferred, the system was optimized to use a display list. This display list stores the location of every voxel in the grid and is stored in graphics memory, so that it only has to be transferred once before computation begins. Then, the OpenGL API calls tell the graphics card to draw the display list, and a minimal amount of data is passed to the graphics card.

6. Results of GPU Optimization

6.1. Experimental Setup

Video was captured from seven synchronized IEEE 1394 color cameras arranged around the edges of a space measuring approximately 5.5m×5.5m×2.5m and connected to three standard office PCs. The cameras were set to record video at a resolution of 1024×768 at a rate of 30 frames per second. Prior to the capturing process, these cameras were affixed in place and calibrated to obtain their intrinsic and extrinsic parameters. In addition, the background image of each camera was captured to aid in the segmentation processing. Frames of raw animation data were recorded to disk for the processing to take place offline. Segmentation masks were precomputed using the segmentation method presented in Sectoion 4.2.1 [8].

Images and segmentation masks for the dataset where then transferred to a separate PC for visual-hull reconstruction and rendering. This PC has a Pentium IV Core 2 processor, with 2 Gigabytes of main system memory, and an NVIDIA 8800 GTS video card with 640 Megabytes of video memory. The computation was performed under Windows XP with service pack 2, compiled with Intel's C++ compiler under Visual Studio 2003. Reconstruction took place on a 300x200x300 voxel grid with a voxel spacing of 1 cm. Output images were generated from camera position information stored as an XML file, and were rendered at a resolution of 1024×768 pixels.

6.2. Factors affecting results

While the results in VI.3 represent two different versions of the same system producing the same results, there are some inherent differences in the way in which the internal processing takes place. The most obvious of these factors is the difference in the way occlusions are calculated. The CPU version of the system utilizes Bresenham's line drawing algorithm to check each voxel for occlusions as it does not require any precomputation and the CPU has fast access to main memory. The GPU version utilizes the depth texture method presented earlier in this paper as it is able to precompute them quickly and greatly reduce the number of texture accesses required during the rendering step.

In addition, however, the storage and processing of the voxel list varies between the CPU and GPU versions of the Cinematized Reality system. The CPU version takes the list generated from the voxel reconstruction algorithm and compacts it down to a list of only the solid voxels. Experimentation has found that this extra precomputation step allows the system to function faster on a per-frame basis than stepping through every voxel and checking whether it is solid or emtpy during the rendering step. The GPU, however, stores the state of every voxel as a texture and is able to "throw out" empty voxels early in the rendering step. Thus, the CPU version has the additional step of "compressing" the voxel list, while the GPU version has the additional step of generating depth textures. Each of these additional steps are considered part of the rendering process as opposed to the reconstruction process, as is indicated in the reported data in the next section. The computation times for these additional steps are shown for comparison purposes. The two versions of the rendering process can be compared directly using the total rendering time.

Additionally, the numbers presented in the next section show only the computation time required for each algorithm to run and the time required for the GPU version to send information between main and graphics memory. However, it does not include the time required to load the image and segmentation mask data, nor the time it takes to save the resulting images to disk, as these times are equivalent in the CPU and GPU versions. The CPU version of Cinematized Reality takes long enough that the time spent loading and saving images is not significant. However, the time spent transferring data between the hard disk and main memory currently is greater than the total processing time for the GPU version. While this presents an interesting opportunity for optimization, the differences in the CPU and GPU versions of the system do not have an impact on the loading times. As such, some ideas are presented in Section 8 to alleviate this problem, but the implementation of these suggestions is beyond the scope of this project.

6.3. Performance data and output

Performance benchmarks show that the optimization of the Cinematized Reality system shows great improvement in total computation time. Three hundred frames of the "karate" dataset shown in Figure 15 were used as a test of both the CPU and GPU versions of the system. Camera control was performed by an XML file that ensured that both versions would render the scene from the exact same viewpoint. Table 1 displays the average time it takes for each system to render a single frame in milliseconds based on the "karate" dataset, as well as the resulting performance improvement.

Step		CPU version (ms / frame)	GPU version (ms / frame)	Improvement
Modeling		23147.7	179.77	128.8x
Microfacet Billboarding	Voxel Compression	0.0793	-	-
	Depth Texture	-	0.89	-
	Render	666.4	0.77	861.3x
	Subtot.	666.5	1.66	401.0x
Total		23814.2	181.4	131.3x

Table 1: Average processing times for a single frame of data

In addition, the system was tested using the "nurses" dataset shown in Figure 16. This scene is particularly interesting because it involves multiple people, which results in a high number of occlusions. Also, there is a table in the middle of the target space that is regarded as background due to the fact that it is static. This causes segmentation errors that result in modeling errors, as the legs of the participants are often occluded by the table from the perspective of one or more cameras. The scene also utilizes two additional cameras: one attached to the ceiling of the target space, and one in front closer to the people. This causes the computation time to increase slightly for the dataset, however the performance of the GPU version is still similar to the performance under the “karate” dataset. Unfortunately, a camera control file has not yet been generated for this dataset, and as such a proper benchmark is unable to be generated at this point. However, the quality of the renders generated by the “nurses” dataset shows that the depth texture method of detecting occlusions works well. The camera selection algorithm of the GPU version shows that it is reliable even when the two nurses are standing close to each other.

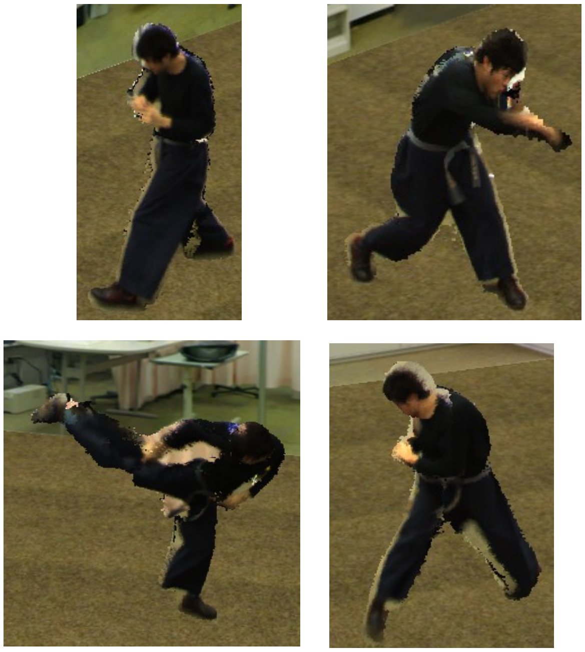


Figure 14: Sample output of “karate” dataset

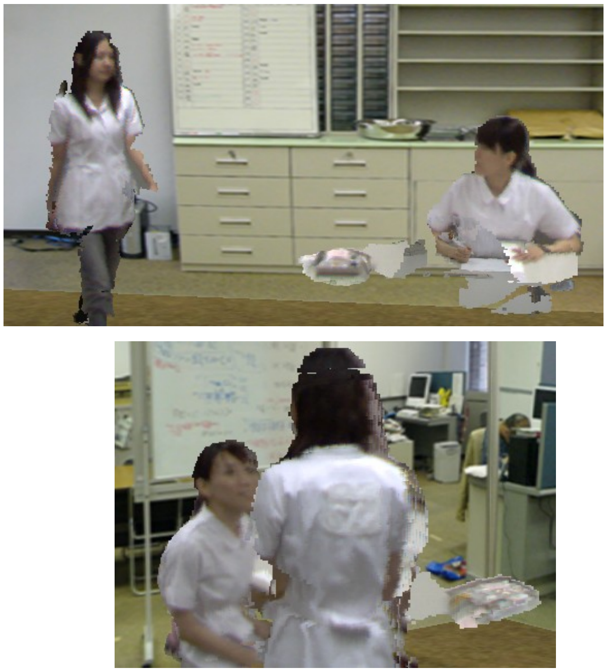


Figure 15: Sample renders of “nurses” dataset

6.4. Analysis of Results

The GPU version of the system vastly outperforms the CPU version in both visual hull reconstruction and rendering, despite the fact that the CPU is running at a higher clockspeed than the GPU and they both have the same amount of data to process. There are several factors that contribute to this. First, the stream processing model of the GPU allows the computation of multiple voxels of data simultaneously in every step, reducing the number of clock cycles necessary by several times. Second, the specialized nature of the GPU allows it to use hardware acceleration on a number of operations that would take the CPU several cycles to accomplish. For example, all matrix and vector operations are hardware accelerated, as are distance and trigonometric functions. As the system relies primarily on this type of mathematical computation, this offers a significant boost in speed. Finally, the different types of shaders allowed data to be processed at several different levels quickly and efficiently.

The resulting renders from the GPU-optimized version of the Cinematized Reality system do not show any significant differences from that of the CPU version, despite utilizing a different camera selection algorithm. This is intentional as the goal of this project was not to change the quality or type of rendered images produced, but to simply improve the performance of the existing system.

7. Future Work

While the optimizations presented here have greatly accelerated the Cinematized Reality system, there are still many avenues for future work, both within the realm of GPU optimization and outside. The system is performing significantly faster as a whole but is far from operating as a real-time system. In addition, the success of the optimization presented here indicates that GPGPU techniques could potentially offer great acceleration possibilities to other computationally expensive problems.

While the reconstruction and rendering processes were moved to operate on the GPU, the segmentation process remained a CPU-bound operation. Where the reconstruction and rendering passes can be completed in ~2.5 seconds (factoring in time for loading and saving all necessary files), the segmentation process alone takes ~10 seconds for a single frame of data in a 7-camera system. While not all of the steps presented in the segmentation algorithm operate on the pixel level, the process of image segmentation is a highly parallel process and could benefit from the GPU programming model and hardware acceleration. If the segmentation process were to see optimization success on the same order of magnitude as the reconstruction and rendering processes, the total processing time for a frame could theoretically be brought under a second, making a near real-time system closer to reality, especially with the pace at which computer hardware is progressing. While this would not eliminate the speed limitations caused by I/O, it would help to reduce the problem, as only initial images and final renders would have to be loaded and saved.

Secondly, the system as a whole could be optimized in the future by streamlining the I/O process. While this cannot be sped up by GPU processing, it is an important step in the optimization of the system as a whole. In fact, currently moving data from one form of memory to another takes significantly more time than actual processing in the reconstruction and rendering process. Initial steps to streamline this process would be to reduce the amount of data that needs to be transferred. This was already accomplished for the locations of the voxels; the voxel indicies for each voxel in the grid is transferred to video memory before processing begins so that it does not need to be transferred every time microfacet billboarding takes place (n + 1 times per frame, where n is the number of cameras). However, this can be accomplished in the image data by compressing the data, both on disk and in graphics memory. In addition, currently the loaded images have to undergo a conversion process in order to be passed into OpenGL as textures. Custom image loading routines could be written or a different library could be used to load image data directly from file into texture memory.

Data flow could further be optimized by tweaking texture formats to transmit the minimum information necessary. Currently, not all channels of all textures are used. Reducing this, or the bitdepth of textures that require less detail, would reduce the amount of data that needs to be transferred. However, as the graphics card will generally store textures as 4-element textures in any case, the process of extracting the relevant data may cause sufficient slowdown as to prevent this from being a practical optimization. In addition, currently the hard drive is spinning up and spinning down as the files are loaded for each frame, despite the short amount of time in between file I/O operations. While this contributes to the 2.5 second frame computation time, it is also caused by it. If multiple frames of data were loaded at a time without waiting for the processing of each frame, the speed would increase. This can be accomplished by offloading the file i/o process to a separate thread of execution and buffering the images to be loaded and saved, so that the file i/o process can be continuous.

The “nurses” dataset presents a new problem that the system is currently unable to handle: background or static objects that occlude dynamic objects in the captured space. Segmentation itra- and inter- reliabilities can be used to minimize the impact of bad segmentation due to occlusion [8], but is not applicable to this scenario as the legs of the participants are occluded in several of the cameras simultaneously. In addition, objects placed on the table appear to be floating in space. This can be solved by adding a computer-generated model of the objects that are classified as background, however this does not solve the occlusion problem. It can potentially be solved by precomputing segmentation masks for the objects in the capture space and combining them with the segmentation masks that are computed for each frame, but this assumes that these objects will never move. Further research is necessary to determine the best approach to solve this problem.

In addition, it would be an interesting experiment to try a similar set of optimizations on other three-dimensional systems. For example, can the same performance boost be expected in a stereo system that relies on image disparity? How about a shape from shading or using a specialized lighting setup? Similar methods of optimization could be applied to systems that must calculate a surface model and thus cannot utilize microfacet billboarding.

8. Conclusions

The computation and rendering of visual hulls is a complex process that requires a significant amount of raw processing power. Previous iterations of the Cinematized Reality system were crippled by the long processing times that this process entailed. Analysis of the algorithms driving the process revealed that the computation was being performed linearly, while the process could inherently be performed in parallel. The independence of each voxel of data from its neighbors allowed this processing to be done in an unsynchronized fashion. As such, GPU processing was explored as an avenue for optimization.

Current generation GPUs offer a large amount of computational power and are becoming made consistently more flexible as the technology matures. However, the process of performing computations on a graphics card comes with a unique set of challenges and complications. Forcing the hardware to perform computations it was not designed for has been made easier with recent developments in the GPGPU community, but programmers pursuing this method must still be aware of these issues, where they come from, and techniques for managing them. In particular, the reliance on textures as data storage imposes a number of limitations in the types of computations that can be performed. Furthermore, the structure of the shader types that are available and their limitations in the data that they can access and manipulate restricts the types of computations that can be performed.

With these limitations in mind, it is still possible to utilize the computational power of the GPU in order to accelerate certain types of operations. The reconstruction and rendering processes of Cinematized Reality proved to be a good candidate for optimization as demonstrated by the results presented in Section 6. This process was not without its difficulties, as data had to be carefully managed in order to fit the memory model of the GPU and optimize the performance. In addition, while most algorithms were able to be translated to the GPU with good performance, this was not the case for all steps. In particular, the occlusion detection process had to be replaced by a new system utilizing precomputed depth textures. This is not due to an inability of the GPU to check for occlusion using the Bresenham-inspired approach present in the original version of the system, but rather due to the performance that resulted from the high number of memory accesses that the algorithm required.

When the optimizations presented in this paper were applied to real-world data, the performance benchmarks showed dramatic improvement in the computational time required to reconstruct and render the scene. While this does not make Cinematized Reality a real-time system yet, the results show a very promising future for the system should optimization continue as presented in Section VII. The benchmarks show that the choice to focus the optimization efforts on the utilization of the GPU's resources was a worthwhile endeavor.

Glossary

- Binocular depth cue – see stereo depth cue
- Camera Calibration – The process of calculating a camera's intrinsic and extrinsic parameters
- CCD (Charged Coupled Device) – the photosensitive element that most cameras use to capture optical information as electrical signals.
- Depth cue – an observable quality of an image or series of images that implies information about the depth of an object relative to a viewpoint.
- Extrinsic Parameters – parameters that specify a camera's position and orientation relative to a fixed coordinate system in 3D space.
- Intrinsic Parameters – parameters that model the camera's optical properties, such as focal length and lens distortion
- Pinhole camera – a model for computing the optical properties of a camera. It assumes that the optics of a camera focus all incoming light down to a single focal point, behind which is the CCD onto which the scene is projected.
- shape from silhouette – a technique for modeling an object by projecting its silhouettes from a series of cameras into the captured space. The volume occupied by the intersection of the projection cones represent an upper bound of the object.
- Stereo depth cue – differences in a pair of images taken from two closely related viewpoints that indicate the depth of an object.
- Voxel – a 'volume pixel'. A single data point in 3D, often modeled as a cube.

References

1. Bresenham, J. “Algorithm for Computer Control of a Digital Plotter”. *IBM Systems Journal* vol. 4, no. 1, pp 25-30. 1965.
2. Hoppe, H.; DeRose, T.; Duchamp, T.; McDonald, J.; Stuetzle, W. “Surface reconstruction from unorganized points”, *Proc. 19th annual conference on Computer graphics and interactive techniques*, pp. 71-78, July 1992 .
3. Kanade, T.; Rander, P. and Narayanan, P.J. "Virtualized reality: constructing virtual worlds from real scenes ," *Multimedia, IEEE* , vol.4, no.1, pp. 34-47, 1997.
4. Kim, H; Sakamoto, R.; Kogure, K.; and Kitahara, I., "Cinematized Reality: Cinematographic 3D Video System for Daily Life Using Multiple Outer/Inner Cameras," *2006 Conference on Computer Vision and Pattern Recognition Workshop* (CVPRW'06), pp. 168. 2006 .
5. Kim, H., “A 3D Modeling System Using Multiple Stereo Cameras,” Dissertation. Yonsei University, 2005.
6. Kim, H., Sakamoto, R., Kitahara, I., Toriyama, T., and Kogure, K. “Robust Foreground Segmentation from Color Video Sequences Using Background Subtraction with Multiple Thresholds”. *Proc. KJPR*, pp. 188-193. 2006.
7. Kim, H.; Kitahara, I.; Sakamoto, R.; Kogure, K. "An Immersive Free-Viewpoint Video System Using Multiple Outer/Inner Cameras," *Proc. 3D Data Processing, Visualization, and Transmission, Third International Symposium on* , pp.782-789, June 2006.
8. Kim, H.; Sakamoto, R.; Kitahara, I.; Toriyama, T.; and Kogure, K., “Reliability-Based 3D Reconstruction in Real Environment (Periodical style—Accepted for publication),” *ACM Multimedia*, to be published.
9. Kumar, P; Sengupta, K; and Ranganath, S, “Real time detection and recognition of human profiles using inexpensive desktop cameras,” *Proc. ICPR*, pp.1096-1099, 2000.
10. Labatut, P.; Keriven, R.; Pons, J.P. "Fast Level Set Multi-View Stereo on Graphics Hardware," *Proc. Third International Symposium on 3D Data Processing, Visualization, and Transmission* (3DPVT'06), pp. 774-781, 2006.
11. Laurentini, A., "The visual hull concept for silhouette-based image understanding," *Pattern Analysis and Machine Intelligence, IEEE Transactions on* , vol.16, no.2, pp.150-162, Feb 1994.
12. Lorensen, W.E. and Cline, H.E., “Marching Cubes: A High Resolution 3D Surface Constructing Algorithm,” *Computer Graphics* (Proc. SIGGRAPH), pp. 163-169. July 1987.
13. Matveyev, S.V., "Approximation of isosurface in the Marching Cube: ambiguity problem," *Visualization, 1994., Visualization '94, Proceedings., IEEE Conference on* , vol. 17, no. 21, pp.288-292, Oct 1994.
14. NVIDIA, “CG Toolkit User's Manual”, nvidia.com.
15. Orman, N.; Kim, H.; Sakamoto, R.; Toriyama, T.; Kogure, K.; Lindeman, R.; “GPU-based Optimization of a Free-Viewpoint Video System,” Submitted for publication.
16. P. Fua, “A Parallel Stereo Algorithm that Produces Dense Depth Maps and Preserves Image Features,” *Machine Vision and Applications*, 1993.
17. Treece, G.M.; Prager R.W.; and Gee, A.H. “Regularised marching tetrahedra: improved iso-surface extraction.” Technical Report, Cambridge University Engineering Dept, September 1998. http://citeseer.ist.psu.edu/treece98regularised.html.
18. Trucco, E. and Verri, A. “Introductory Techniques for 3-D Computer Vision.” New Jersey: Prentice Hall, 1998.
19. Tsai, R., "A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses," *Robotics and Automation, IEEE Journal*

of, vol.3, no.4, pp. 323-344, Aug 1987.

20. Wasson, S., “Nvidia's GeForce 8800 graphics processor,” The Tech Report. Nov. 8, 2006. http://techreport.com/articles.x/11211. Accessed on Oct. 10, 2007.
21. Willems, G.; Verbiest, F.; Vergauwen, M.; Gool, L.V. "Real-Time Image Based Rendering from Uncalibrated Images," *Proc. Fifth International Conference on 3-D Digital Imaging and Modeling* (3DIM'05), pp. 221-228, 2005.
22. Yamazaki, S.; Sagawa, R.; Kawasaki, H.; Ikeuchi, K.; and Sakauchi, M., “Microfacet billboarding,” *Proc. 13th Eurographics Workshop on Rendering, 2002*, pp. 175–186, 2002.
23. Zhang, Z., "A flexible new technique for camera calibration," *Pattern Analysis and Machine Intelligence, IEEE Transactions on* , vol.22, no.11, pp. 1330-1334, Nov 2000.