GPU-Based Optimization of a Free-Viewpoint Video System

Neal Orman[*], Hansung Kim[+], Ryuuki Sakamoto[+], Tomoji Toriyama[+], Kiyoshi Kogure[+], and Robert Lindeman[*]

**Computer Science Department, Worcester Polytechnic Institute, Worcester MA USA +01-508-831-5000 {Neal.Orman@gmail.com, gogo@wpi.edu}*

+Knowledge Science Lab, Advanced Telecommunications Research Institute International, Kyoto Japan, +81-774-95-1401 {hskim, skmt, toriyama, kogure}@atr.jp

Received 15 May 2008; accepted 12 June 2008.

Abstract

We present a method for optimizing the reconstruction and rendering of 3D objects from multiple images by utilizing the latest features of consumer-level graphics hardware based on shader model 4.0. We accelerate visual hull reconstruction by rewriting a shape-from-silhouette algorithm to execute on the GPU's parallel architecture. Rendering is optimized through the application of geometry shaders to generate billboarding microfacets textured with captured images. We also present a method for handling occlusion in the camera selection process that is optimized for execution on the GPU. Execution time is further improved by rendering intermediate results directly to texture to minimize the number of data transfers between graphics and main memory. We show our GPU based system to be significantly more efficient than a purely CPU-based approach, due to the parallel nature of the GPU, while maintaining graphical quality.

Key Words: 3D Reconstruction, GPGPU, Image-Based Modeling and Rendering, Microfacet Billboarding.

1 Introduction

3D reconstruction and free view rendering has been an important topic in computer graphics and computer vision for many years. A variety of techniques have been proposed to solve this problem, with a wide range of quality and speed, and hardware requirements [1-4]. While early techniques were slow (partially due to limited hardware resources) and produced crude-looking results, recent advances in hardware complexity and computer vision algorithms have produced more realistic images from novel viewpoints in near-real time [2][5]. Most implementations fall under two categories: geometry-based or image-based. Geometry based solutions utilize traditional rendering techniques, and thus require a high level of detail to generate accurate surfaces. Image-based rendering attempts to solve this drawback by performing the rendering step using captured images. As such, it is not necessary to have an exact geometric model of the object being rendered [6]. A common technique for rendering such scenes is to create a rough model from the captured images and then render the model using image-based techniques. The most common way of creating this model from silhouette data is the shape-from-silhouette (SFS) technique [7-10]. Many vision systems have taken this approach [11-13] after Kanade et al. proposed "Virtualized Reality" as a new visual medium for manipulating and rendering pre-recorded scenes [4].

However, the SFS approach has the downside of high computational complexity due to the need to back-project each point onto all silhouettes. This can be handled by using clusters to process the real-time modeling and rendering as Matsuyama et al. and Ueda et al. chose to do, but this increases system costs[7][14].

Correspondence to: <Neal.Orman@gmail.com>

ELCVIA ISSN: 1577-5097
Published by Computer Vision Center / Universitat Autonoma de Barcelona, Barcelona, Spain

Meanwhile, advances in consumer level programmable graphics hardware have given researchers access to powerful hardware acceleration. Despite the fact that these GPUs (Graphics Processing Units) are programmable, their specialized nature and Concurrent Read Exclusive Write Single Instruction Multiple Data (CREW SIMD) programming model restricts the kinds of operations that can be performed efficiently. Despite these limitations, many researchers have begun to take advantage of the specialized nature of these processors to accelerate General-purpose Programming on the GPU (GPGPU) [15].

A number of computer vision techniques already take advantage of GPGPU techniques to perform their computations [16-18]. However, the hardware industry is constantly adding new features to their hardware which means that such techniques soon are unable to utilize the full potential of the latest graphics hardware. Recently, the emergence of geometry shaders and the shader model 4.0 has added a new level of power and flexibility that most GPGPU techniques have not yet taken advantage of.

In this paper we present a method for accelerating the process of reconstructing and rendering a visual hull from multiple segmented camera images using the latest features of widely available consumer-level graphics hardware. While we build upon a number of established methods such as voxel carving using shape-from-silhouette, we also integrate a novel method for solving camera selection on the GPU using depth textures. Additionally, the system we present here provides full hardware-acceleration for all modeling and rendering tasks, and avoids swapping intermediate results between main memory and graphics memory by rendering directly to texture memory. The result of this work is a GPU- based approach to the reconstruction and rendering stages of the imaging system, which we show to be more efficient than a CPU-based approach.

In section II we present an overview of a CPU based free-viewpoint system that takes advantage of powerful techniques such as shape-from-silhouette for 3D reconstruction, and microfacet billboarding for rendering. In section III we optimize these techniques using vertex, geometry and fragment shaders under shader model 4.0. Finally, we quantify the impact that these optimizations have had on system performance and accuracy in section IV. Section V wraps up with conclusions and an overview of future work.

2 3D Reconstruction and Free-Viewpoint Rendering

System Overview

Our free-viewpoint video system takes images from standard video cameras and constructs a 3D model of the objects contained within, from which it renders video of the scene from virtual cameras from novel viewpoints. This is accomplished by segmenting the captured images to extract foreground silhouettes and using these silhouettes to construct a voxel-based model of the object. The original images are then used to texture this model during the rendering process.

The free-viewpoint system uses a series of “environmental” cameras oriented towards the center of the capture space (Fig. 1). The cameras are off-the-shelf cameras permanently affixed to the project space. Prior to recording, each camera's intrinsic parameters (distortion coefficients, etc) and extrinsic parameters (position and orientation) are calculated using the calibration method presented in [1]. A robust segmentation algorithm presented in [19] calculates the foreground of each image and generates an image mask (hereafter referred to as a segmentation mask).

After the segmentation masks have been computed (as seen in the top-right corner of Fig. 1), the system reconstructs the visual hull based on a Shape-From-Silhouette (SFS) algorithm which carves the space using the segmentation masks and precomputed camera calibration parameters [19]. The space is modeled as a regular voxel grid, with the value of each voxel indicating the presence of a foreground object. The voxel model is carved by projecting the silhouette into the voxel space and culling any voxels that lie in the background region in the 3D modeling step presented in Fig. 1. Once the model is computed, we employ a version of the microfacet billboarding technique originally presented in [20] to render the scene from any viewpoint. The resulting rendered images are textured by mapping the individual microfacets to the images captured by the cameras. We have found through experimentation that correct camera selection for each

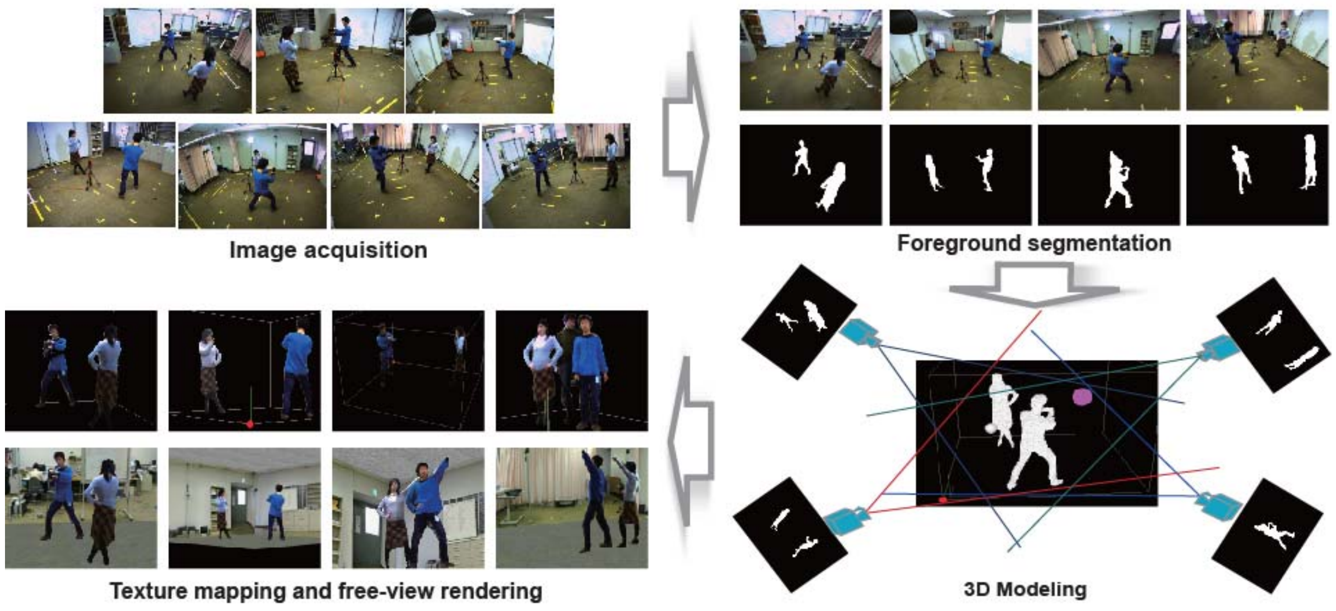


Fig. 1. Flow diagram of free-view imaging system.

microfacet is critical to the perceived quality of the final render. Figure 1 presents a graphical overview of the system.

Visual Hull Reconstruction

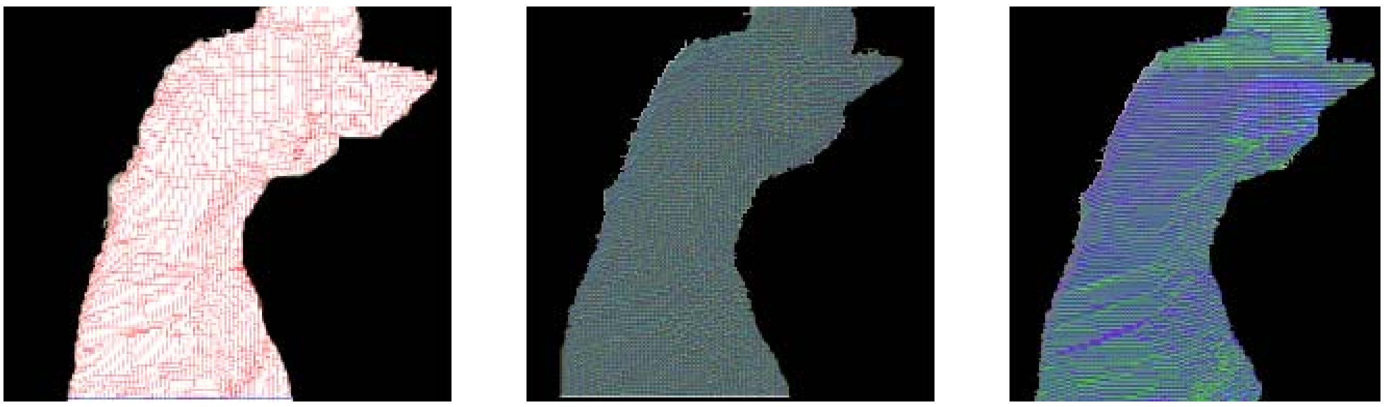
Visual hull reconstruction is the process of computing a 3D model of a scene from image data [21]. Our system does this with a 'shape-from-silhouette' method on a voxel grid that occupies the target space. The resulting hull is formed by projecting the silhouettes obtained from the segmentation masks into the voxel grid. Any voxel contained within the intersection of these projections is considered “solid” and is included in the visual hull. However, if a voxel lies outside of any of the silhouettes, it is culled, and no further processing takes place.

For example, define a system with N cameras labeled C_m (m=1,...,N), silhouettes S_m, and projection matrices P_m. Now, assume that a point p lies within a voxel V_n. If V_n lies within the volume occupied by the hull, then p must lie within S_m when projected onto the image plane of C_m, for every m=1,...,N. If p projects onto the background area of $S_{m\ for}$ *any* m=1,...,N, then it must lie outside of the object and is culled. Thus, we can calculate the visual hull of an object in this manner based solely on silhouette information and camera calibration parameters.

Rendering

Many techniques exist for rendering voxel-based data, each with its strengths and weaknesses. One approach is to render each voxel as a cube [22]. This approach is appealing because of its low computational complexity. However, this approach produces an image that appears unnatural for organic forms. Figure 2(a) shows the results of this rendering method.

Another popular method of rendering voxel-based data is the marching cubes algorithm originally published in [23]. Since each vertex must lie within or outside of the surface being modeled, there are a finite number of possible configurations for a given cube's geometry. Lookup tables and vertex interpolation allow the appropriate polygons to be generated quickly for each cube. The resulting mesh is more detailed, but has a very high polygon count as seen in Fig. 2(b).



(a) cube-based render (b) marching cubes render (c) microfacet billboarding render

Fig. 2. Three methods for rendering a voxel-based mesh

Microfacet billboarding, by contrast, generates a single polygon for each voxel that defines the model [24]. The microfacet is defined as, "a slice which intersects the center of the voxel and is vertical to the viewing direction" (see Fig. 2(c)) [20]. Thus, as the virtual camera is rotated around the model, the normals of each rendered microfacet are adjusted to be parallel to the virtual camera's view vector. Each polygon generated is sized to match the size of the voxel it represents. In a regular voxel grid, each side of the facet must be at least three times the size of the voxel length.

This ensures that the voxel will cover enough area in the final image to prevent gaps between facets regardless of viewing angle.

Through experimentation, we have determined that the results of the microfacet billboarding technique form the exact same shape as the results of the marching cubes algorithm with significantly less computation required. We believe this is due to the nature of the voxel representation of the visual hull. The disadvantage to microfacet billboarding lies in the fact that it ignores the surface properties of the model being rendered, specifically the normals. Due to the fact that we are using captured images as the basis for texturing our objects, we do not need to do any lighting calculations. As a result, this limitation does not have a significant impact on the quality of our results.

Camera Selection and Texturing

Our system is based on image-based rendering, as it uses real images to produce results that do not require a high level of geometric complexity [2]. Because of occlusion and changes in a surface's appearance due to lighting conditions and viewing angle, great care must be taken to select the appropriate camera to use as a texture for each microfacet. Improper camera selection can lead to unrealistic images due to many factors, including distortion, clipping and occlusion.

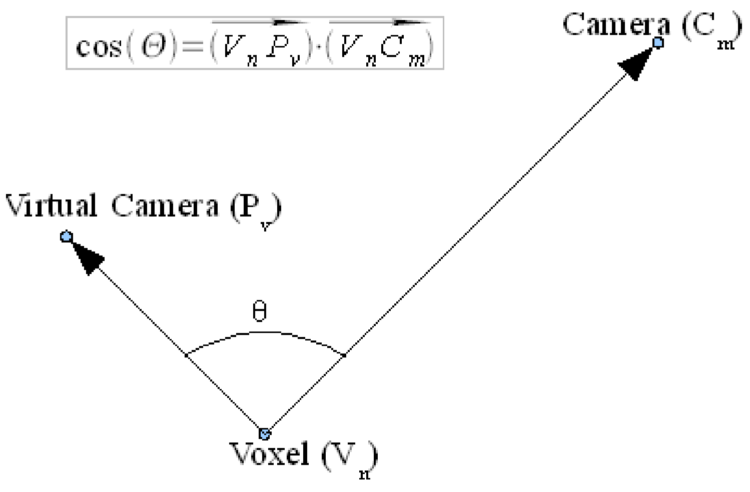

$\cos(\Theta)=\overline{(\overrightarrow{V_n P_v})}\cdot\overline{(\overrightarrow{V_n C_m})}$
Camera (C_m)
Virtual Camera (P_v)
θ
Voxel (V_n)

Fig. 3. Calculation of viewing angles.

Our system implements a system by which every non-empty voxel ranks the environmental cameras based on several criteria. This is accomplished by computing a “visibility score” for each camera and then sorting the cameras based on this value.

The first criterion considered is the angle at which the camera is viewing the voxel. Given a physical camera C_m, virtual viewpoint P_v, and a voxel V_n, our system computes $\Theta_{n,m}$. This angle represents the difference in the viewing angles between the physical and virtual camera. It is computed by taking the inverse cosine of the dot product of the vectors V_nP_v and V_nC_m (shown in Fig. 3). $\Theta_{n,m}$ is used as the initial visibility score for each camera Cm in the scene so that the camera with the smallest $\Theta_{n,m}$ can be selected to texture the voxel. Additionally, the texture coordinates of the voxel are computed for each camera. If the texture coordinates lie outside of the captured image, the visibility score is set to an arbitrarily high value to prevent the use of that camera.

The primary problem with this method, however, is that it ignores the possibility of occlusion. When non-trivial objects are recorded by the proposed system, the model generated often occludes itself in one or several of the captured camera images. For example, when the person being recorded held their arm up between one of the cameras and their body, the image of their arm would be used to texture not only the arm itself, but also part of their body. For example, Fig. 4 demonstrates this problem. Figure 4(a) displays the scene rendered from the camera's angle, while Fig. 4(b) shows the same scene rendered from a slightly higher angle. Notice that the texture of the hand is rendered on the chest as well. Proper segmentation and occlusion detection can help eliminate this problem.

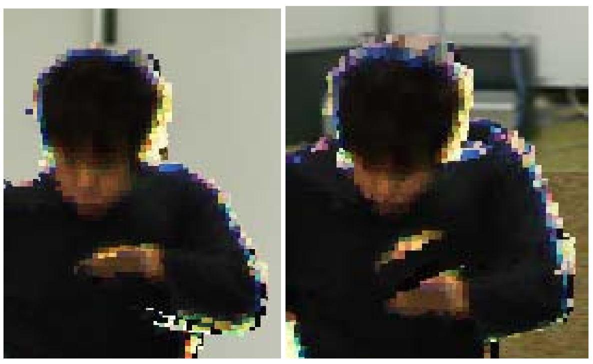

(a)Perspective from camera (b) Higher angle

Fig. 4. Occlusion problem example.

Given a viewpoint P_v, a voxel V_n is considered occluded if there exists a voxel W, such that W lies directly 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 [25]. If any of the voxels checked is solid, then V_n is occluded, and another camera must be chosen. Due to the fact that the line drawing algorithm works in pure integer math, this algorithm runs quickly, but still requires many accesses into the voxel model.

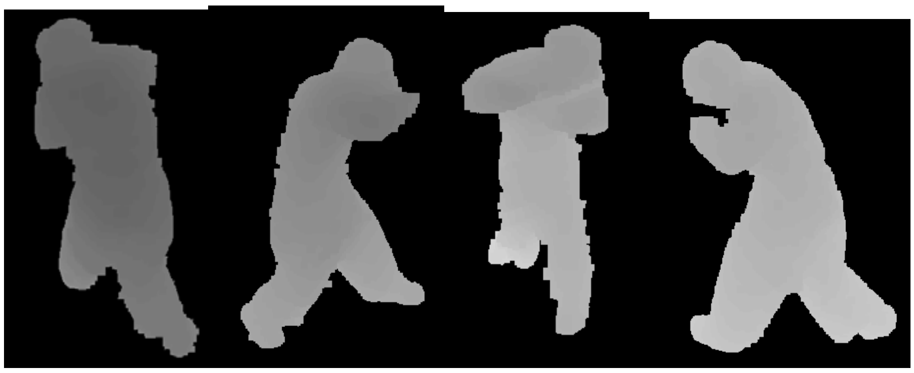


Fig. 5 Sample depth textures

Alternatively, it is possible to avoid this many lookups into the voxel model by doing some precomputation. Before the model is rendered from the perspective of the virtual viewpoint, it is rendered from the perspective of each camera.

For a given camera C_m, if every voxel V_n is rendered as a microfacet to a depth buffer using the perspective of camera C_m, a depth texture T_d is generated. Given a texel t in T_d, the value of t represents the distance from C_m to the closest microfacet along the ray projected from C_m through t. Example results of the depth textures generated by this method are shown in Fig. 5. Note that the intensity of each pixel represents its depth; the lighter parts of the images are further away than the darker parts.

Thus, when rendering, it is possible to use these depth textures to check for occlusion. First, a distance d_{vc} is calculated between the voxel, V_n, and the camera C_m. Then, the vertices of the microfacet are projected onto the image plane of C_m, resulting in a set of texture coordinates. The distance from C_m to the closest microfacet at each of the corners of these texture coordinates, defined as d_1-d_4 is calculated by performing a texture lookup into the depth texture for C_m at the calculated texture coordinates. If $d_n > d_{vc}$ for any $n=\{1,4\}$, then V_n is at least partially occluded by another voxel. Otherwise, the texture coordinates are used to texture V_n with the image captured by C_m. If V_n is occluded, then the algorithm continues to check the remaining cameras based on their priority until one is found that is not occluded.

Each approach has its strengths and weaknesses, and thus is applicable in different situations. Using Bresenham's algorithm is preferable for a CPU-based version because it requires no precomputation and has a low computational complexity. While it requires many accesses to the voxel grid, array lookups on the CPU are faster than on the GPU. Thus, the algorithm is useful when the voxel grid and computation is done in main memory on the CPU. In addition, the CPU is not able to generate the depth-textures as easily as the GPU can, making a line-drawing based algorithm preferable for the CPU. When the computation and the voxel grid are performed on the GPU, however, the cost of accessing the voxel texture is much higher. Precomputing depth textures requires an additional step before rendering, but it reduces the computational complexity of the occlusion check for each voxel to a single texture check and comparison. Additionally, copying the depth textures from graphics memory to main memory takes a significant amount of time. As such, we use Bresenham's algorithm to check occlusion in the CPU version of our pipeline, but precomputed depth textures when the processing is performed on the GPU.

3 GPU Acceleration of the Rendering Process

In our implementation of the acceleration techniques presented here, we use the Cg language under shader model 4.0 in conjunction with OpenGL. Algorithms 1-4 show pseudo code for each of the main shaders described in the following sections.

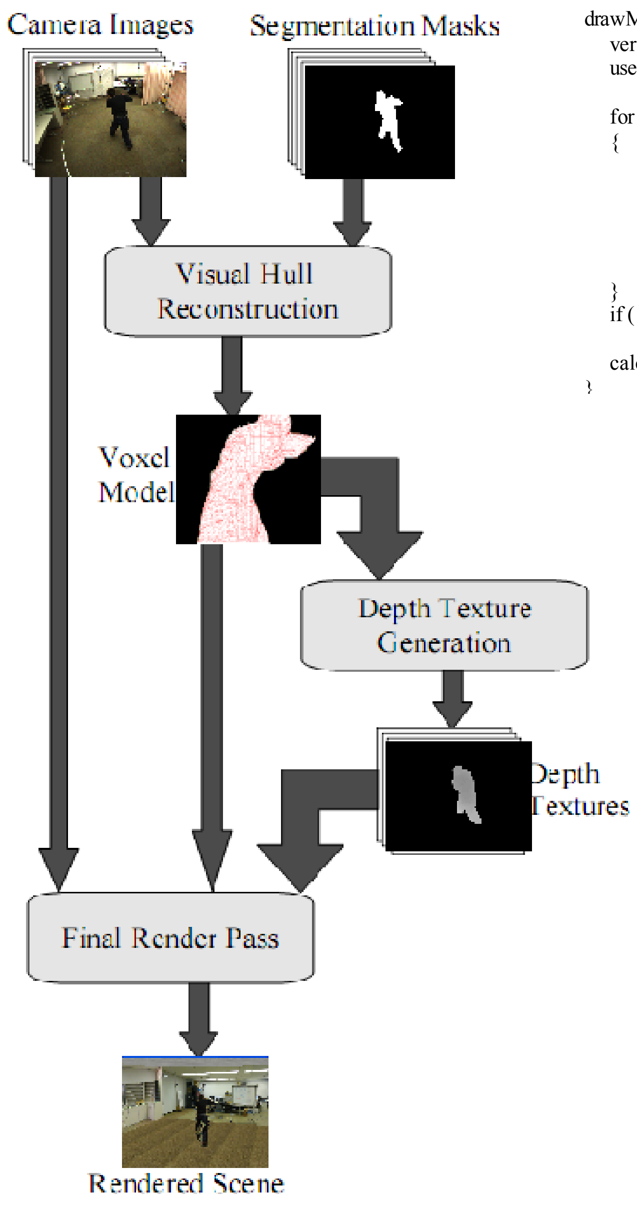


Fig. 6. Data-flow analysis of GPU-accelerated 3D reconstruction and rendering.

```
drawMicrofacet(position : POSITION, cameraIndices: COLOR) {
    vertices[4] = calculateMicrofacetVertices(position, viewpoint);
    useCam = NULL;

    for (int camNum = 0; camNum < 3 && useCam == NULL; camNum++)
    {
            C = getNextCamera();
            dist = distance(position, C);
            imageCoords[4] = projectPoints(vertices, C);
            occlusions[4] = false;
            if (all(depthTexture(imageCoords) < dist)) useCam = C;
    }
    if ( useCam == NULL) useCam = getFirstCamera(cameraIndices);

    calculateMicrofacet(vertices, useCam);
}
```

Algorithm 4: Final render geometry shader

Data Flow Overview

Figure 6 presents an overview of the flow of data through the hardware accelerated portion of the free-viewpoint video system. Processing begins with images and segmentation masks being passed into the visual hull reconstruction module, which produces a voxel representation of the hull stored as a texture. This is passed along with camera parameters to the depth texture module which produces depth textures for each camera. Finally, the render module takes these textures as parameters and produces the rendered result. For performance reasons, all intermediate results are maintained in graphics memory.

Shader-Based Visual Hull Reconstruction

Visual hull reconstruction is implemented using a single fragment shader. Results are rendered directly to a three-dimensional texture using a framebuffer object (FBO) one layer at a time. Each layer is rendered such that the fragment shader is run once per texel.

The fragment shader takes a number of input parameters including intrinsic and extrinsic parameters of each camera, the texture location of the voxel being computed, and the captured camera images and segmentation masks. Processing begins by transforming the voxel position into the global coordinate system. Then, for each camera, this coordinate is projected onto the camera's image plane using its calibration parameters. This process utilizes traditional three-dimensional projection onto a two-dimensional plane but also accounts for the distortion of the camera lens. The resulting coordinates are checked against the bounds

```
Modeler(int3 indices) {
    float3 position  = CalculateGlobalPosition(indices);
    int effective_camera = 0, actual_camera = 0;
    bool empty = false;
    for (int cam = 0; cam < numCameras && !empty; cam++) {
            camera C = getCamNum(cam);
            float2 imageCoordinates = projectPoint(position, C);
            if (imageCoordinates.isValid()) {
                    color =Masks[cam].getPixel(imageCoordinates);
                    if (color == 0.0) empty = true;
            }
    }
    if (empty)  return black;
    else      return white;
}
```

Algorithm 1: 3D reconstruction fragment shader

```
createMicrofacet(float3 indices, camera C) {
    if (!isVoxelFilled(indices)) return;

    float3 position = calculateGlobalPosition(indices);
    float dist = distance(position, C);
    float2 imageCoords = projectPoint(position, C);

    for (int vertex = 0; vertex < 4; vertex++) {
            float2 corner = imageCoords + offset[vertex];
            emitVertex(corner : POSITION, dist : COLOR);
    }
}
```

Algorithm 2: Depth texture geometry shader

of the image, and if they are within the bounds, then the shader performs a texture lookup on the segmentation mask. If the voxel lies in the background region of the mask for any camera, the fragment returns a black pixel signifying an empty voxel. If it lies in the foreground for every camera, a white pixel is the output signifying a solid voxel. The resulting texture represents the foreground objects in the scene represented on a three dimensional voxel grid. Algorithm 1 provides pseudocode for this fragment shader.

Depth Textures Using Microfacet Billboarding

Depth textures are generated by rendering all space-occupying voxels as microfacets from the perspective of a given camera. Depth sorting ensures that only the microfacets that are closest to the camera will be displayed in the final texture. We use this property to our advantage by setting each microfacet's color equivalent to its distance from the camera. Thus, the final texture will contain the distance to the closest microfacet at each texel.

To implement microfacet billboarding, our system relies primarily on the geometry shader because it operates in camera (view) space. This means that the vertices of the microfacet are trivial to compute, as it involves the addition of fixed offsets in the *x* and *y* directions. This ensures that the resulting polygon will always be facing the camera directly.

The geometry shader is responsible for taking in each voxel as a point and outputting the appropriate polygon located at the voxel's center. After checking that the voxel must be rendered, the position of the voxel is calculated, and the distance between the voxel and the camera is calculated. The center of the microfacet is calculated by projecting the position of the voxel onto the camera's image plane. From this position, four vertices are created adding fixed offsets in the *x* and *y* directions. This ensures that the resulting polygon is of a uniform size and oriented to face the camera directly. We do not have to change the size of the microfacet based on the distance from the camera because perspective projection is performed automatically by hardware after the geometry shader to transform the coordinates from camera space to pixel or image space. The color of the polygon is set to the distance calculated earlier. The process is then repeated for every camera used to record the scene. Algorithm 2 contains the pseudocode that describes this process.

Shader-Accelerated Final Render

In order for the final image to be rendered, the correct camera must be selected using the method in section II.*D*. First, the vertex shader selects the three best cameras for texturing each voxel. The geometry shader then checks for occlusions and renders the voxel as a microfacet.

The vertex shader begins by checking the voxel texture to ensure that the voxel needs to be rendered. If it does, the voxel's position in the global coordinate system is calculated. Then, the vertex shader calculates $\Theta_{n,m}$ for each camera in the scene based on the formula in Fig. 3. This is used as the basis of a “visibility score” for each camera. The voxel's position is then projected onto the image plane of the camera to check its coordinates. If the voxel lies outside of the viewable image space, its visibility score is increased to a large value to prevent its use.

Once the visibility score is computed for each camera, the vertex shader sorts the list of cameras and picks the “best” three. This is accomplished by a modified insertion sort. The vertex shader only allocates enough memory for the top three camera indices and visibility scores. Then, it loops through each camera, inserting the camera into the output list only if its visibility score is better (lower) than the three stored values in the output arrays. For example, if the current camera's visibility score is lower than all three, it becomes the “best” camera, and the former top two choices are moved to the second and third choices. Once the sorting algorithm has looped through each camera once, the output memory will store the best three cameras.

```
SelectCamera(voxelPosition) {
    if (!isVoxelFilled()) return;
    visibilityScore[num_cameras];
    viewAngle = calculateViewAngle(voxelPosition, viewpoint);
    for (int cam = 0; cam < num_cameras; cam++) {
            camera C = getCamera(cam);
            camAngle = calculateViewAngle(voxelPosition, C);
            visibilityScore[cam] = abs(camAngle – viewAngle);

            imageCoords = projectPoint(voxelPosition, C);
            if (!imageCoords.isValid() )
                    visibilityScore[cam] = inf;
    }
    sort(visibilityScore);
    float3 color = float3(visibilityScore[0], visibilityScore[1],
visibilityScore[2]);
    return vertex(voxelPosition : POSITION, color : COLOR);
}
```

Algorithm 3: Final render vertex shader

```
drawMicrofacet(position : POSITION, cameraIndices: COLOR) {
    vertices[4] = calculateMicrofacetVertices(position, viewpoint);
    useCam = NULL;

    for (int camNum = 0; camNum < 3 && useCam == NULL; camNum++)
    {
            C = getNextCamera();
            dist = distance(position, C);
            imageCoords[4] = projectPoints(vertices, C);
            occlusions[4] = false;
            if (all(depthTexture(imageCoords) < dist)) useCam = C;
    }
    if ( useCam == NULL) useCam = getFirstCamera(cameraIndices);

    calculateMicrofacet(vertices, useCam);
}
```

Algorithm 4: Final render geometry shader

While this is not the most efficient sorting algorithm in the general case, it is simple to implement on the GPU and only requires enough memory to store the desired output. In addition, there are typically less than ten cameras to sort and section IV reveals that GPU computation time is only a small portion of the overall system execution time. As such, optimization of this sorting process is unlikely to produce significant performance benefits. Algorithm 3 shows the pseudocode for this vertex shader.

The geometry shader then receives this list of cameras along with the voxel information. It applies offsets to this position along the *x* and *y* axes to generate the corner points of the microfacet. The geometry shader calculates the distance d_1 between the first selected vertex U_1 and the first camera C_1. In order to see if U_1 is occluded, it is projected onto the image plane of C_1. A simple texture lookup retrieves the depth texture value, R, at this texel. If $R < d_1$, then another solid voxel is closer to C_1 at that texel than U_1, meaning U_1 is occluded for C_1. The shader continues execution with the other selected cameras until the first one is found that is not occluded. After computing the best camera for all four vertices of the microfacet, the shader chooses the best camera that can see all four vertices. If all three cameras show occlusion for one or more of the vertices, then the first camera is selected by default. Finally, the selected camera's image is used to texture the microfacet. This geometry shader is outlined by the pseudocode in Algorithm 4.

4 Experimental Results

Configuration

We have implemented a video system composed of 3 PCs capturing video from seven calibrated and synchronized IEEE 1394 Point Grey Dragonfly 2 cameras. The captured area measures approximately 5.5m×5.5m×2.5m, with the cameras affixed on the wall around the space. All cameras were oriented to the center of the space and captured video at a rate of 30 fps at a resolution of 1024×768. Captured images were

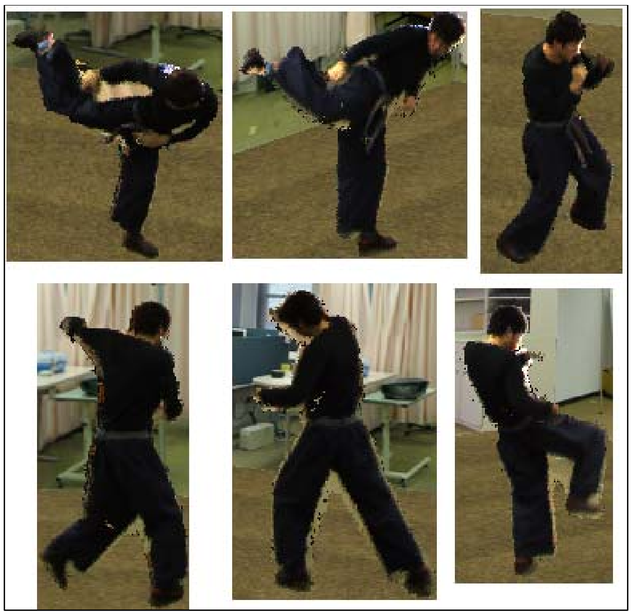


Fig. 7. Frames from “karate” dataset using arbitrary viewpoints

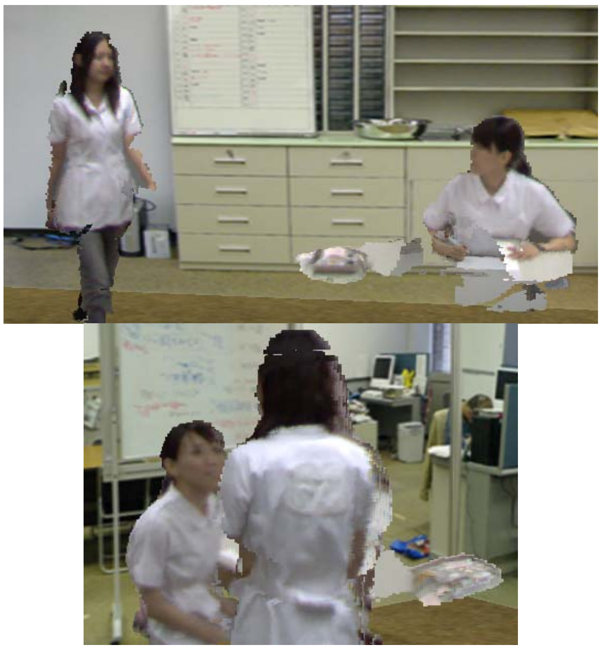


Fig. 8. Frames rendered from “nurses” dataset using arbitrary viewpoints

streamed over a network to storage for offline processing. Segmentation of captured images was performed in advance of modeling and rendering through the technique presented in [19].

All modeling and rendering functions were performed on a PC with a Pentium IV Core 2 processor, 2 GB of RAM and an NVIDIA 8800 GTS video card running Windows XP with Service Pack 2 installed. Voxel reconstruction was performed on a 300×300×200 grid with a voxel spacing of 1 cm^3. Output images were rendered at a resolution of 1024×768.

Runtime Performance

Through harnessing the computational power of the GPU using the concepts presented in this paper, we have managed to greatly improve the performance of our system. Table 1 presents the results of the modeling and rendering steps performed on 300 frames of the “karate” dataset shown in Fig. 7. All times shown represent the computed average time for a single frame.

The system was also able to render the “nurses” dataset shown in Fig 8. with proper texture selection despite the large number of occlusions that come from having two people in a scene. This shows that the depth texture method of occlusion detection performs well even in complex scenes. System performance of the GPU version was similar across both datasets, but performance statistics are not included for the “nurses” dataset due to the fact that a reliable benchmark of the dataset has not been generated using the CPU version at this time.

There are two main reasons for the differences in data flow. First, the CPU version of microfacet billboarding algorithm relies on the existence of a compact list of solid voxels. Because the modeling function outputs an array of voxels that are both filled and empty, we must compress the voxel array down to a list of the positions of only the filled voxels. We have determined through experimentation that computing this “voxel compression” is more computationally efficient than requiring the CPU-based microfacet billboarding algorithm to cull the empty voxels. By contrast, the GPU version of the system is able to skip over empty voxels by terminating the shader's code early. Thus, voxel compression is not necessary for the GPU version of the system.

Table 1. Performance Analysis of GPU based visual hull reconstruction and rendering

Step		CPU Version (ms / frame)	GPU Version (ms / frame)	Improvement
Modeling		23147.6819	179.7707	128.8x
Microfacet Billboarding	Voxel Compression	0.0793	-	-
	Depth Texture	-	0.8881	-
	Render	666.419	0.7737	861.34x
	SUbtot	666.4983	1.6618	401.07x
Total		23814.1802	181.4325	131.26x

Secondly, the occlusion problem in camera selection differs between the CPU and GPU version. The CPU version uses a three-dimensional version of Bresenham's line drawing algorithm which requires no precomputation, while the GPU version uses precomputed depth textures to minimize texture accesses.

Finally, the GPU version has the additional overhead of transferring data to graphics memory before executing and retrieving the results after the computation is complete. In Table 1, the time it takes to transfer all camera images and segmentation masks from main memory to graphics memory is included in the modeling process. Similarly, the microfacet billboarding rendering step contains the time necessary to retrieve the final rendered result from the framebuffer.

It is also important to note that the times listed in Table 1 do not include the time to load camera images and segmentation masks into main memory from the hard disk, as Table 1 focuses on the time it takes to compute the results. While the necessary time to load these files from disk was not a concern with the computation time of the CPU version, it has been consistently larger than the computation time of the GPU version. However, optimization of this process is outside of the scope of this paper.

Despite the fact that the GPU is performing the exact same computations as the CPU, the GPU version runs much faster for several reasons. First of all, the stream processing model that the GPU uses allows the computation of multiple voxels simultaneously without the overhead or synchronization issues of multi-threading. Additionally, the GPU is a specialized processor that is optimized to do a number of operations that are useful to our algorithm. For example, all of the matrix operations involved in projecting a point onto the image coordinates of a camera's image plane were able to be hardware-accelerated. Finally, the GPU instruction set and drivers are finely tuned for the types of operations that our algorithm performs, such as texture lookups vector arithmetic, and distance calculation.

5 Conclusion

In this paper we have presented a practical method of optimizing free-viewpoint imaging systems on consumer-level graphics hardware. The central concept that the optimizations function on is the division of computation into discrete bits that the GPU processes in parallel. Additionally, by performing multiple computations in series on the GPU, we have avoided reading back intermediate results into main memory. We proposed a system that adapts known algorithms to specialized hardware to take advantage of the latest advances in GPU hardware. Our evaluation shows that the optimization is very effective in reproducing visually accurate models at speeds much faster than a CPU-based approach. With the continuing advances in hardware and further work it is feasible that future imaging systems will be able to utilize these image-based techniques to recreate recorded scenes from novel viewpoints in near-real time.

Currently, image segmentation is performed on the CPU and takes a significant amount of computation time. Several methods for performing segmentation on graphics hardware have been proposed, but to our knowledge few have been integrated into a single hardware accelerated pipeline with modeling and rendering algorithms [3][16][18]. Additionally, the camera selection algorithm can be improved upon to minimize the impact of the areas of the model where texturing switches from one camera to another.

Blending of multiple camera images to produce the texture for each microfacet could create seamless transitions, but would have a negative impact on performance and could cause a blurring effect. Overcoming these limitations would be a useful topic for further research.

ACKNOWLEDGMENT

This research was supported by the National Institute of Information and Communications Technology of Japan.

References

[1]H. Kim, I. Kitahara, R. Sakamoto, K. Kogure, "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.
[2]G. Willems, F. Verbiest, M. Vergauwen, L.V. Gool, "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.
[3]P. Labatut, R. Keriven, J.P. Pons, "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.
[4]T. Kanade, P. Rander and P.J. Narayanan, "Virtualized reality: constructing virtual worlds from real scenes ," *Multimedia, IEEE* , vol.4, no.1, pp. .34-47, Jan-Mar 1997.
[5]H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, "Surface reconstruction from unorganized points", *Proc. 19th annual conference on Computer graphics and interactive techniques*, pp. 71-78, July 1992 .
[6]P.Debevec, Y. Yu, and G. Borshukov. "Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping", *Proc. Eurographics Rendering Workshop 1998*, pp. 105-116, 1998.
[7]T. Matsuyama, X. Wu, T. Takai, and T. Wada, "Real-Time Dynamic 3-D Object Shape Reconstruction and High-Fidelity Texture Mapping for 3-D Video," IEEE Trans. CSVT, vol. 14, no. 3, pp.357-369, 2004.
[8]S.N. Sinha and M. Pollefeys, "Visual-Hull Reconstruction from Uncalibrated and Unsynchronized Video Streams," Proc. 3DPVT, pp.349-356, 2004
[9]W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan, "Image-based visual hulls," SIGGRAPH, pp.369-374, 2000.
[10] G. Cheung, T. Kanade, J.Y. Bouguet, and M. Holler, "A real time system for robust 3d voxel reconstruction of human motions," Proc. CVPR, pp.714-720, 2000.
[11]M.A. Magnor, Video-Based Rendering, A K Peters, 2005.
[12]T. Kanade and P.J. Narayanan, "Historical Perspectives on 4D Virtualized Reality," Proc. 3DCINE workshop in CVPR, pp.165, 2006.
[13]M. Gross et al., "Blue-c: A spatially immersive display and 3D video portal for telepresence," SIGGRAPH03, pp.819-827, 2003.
[14]M. Ueda, D. Arita, and R. Taniguchi, "Real-Time Free-Viewpoint Video Generation Using Multiple Cameras and a PC-Cluster," proc. PCM 2004, LNCS 3331, pp. 418-425, 2004.
[15]J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. "A survey of general-purpose computation on graphics hardware." *Eurographics 2005, State of the Art Reports*, pp. 21--51, Aug. 2005.
[16]M. Li, M. Magnor, H.-P. Seidel, "Hardware-accelerated visual hull reconstruction and rendering", *Proc. of Graphics Interface* (GI'03). 2003.
[17] J. Fungand, S. Mann, "OpenVIDIA: parallel GPU computer vision", *ACM multimedia 2005*, pp. 849–852. 2005.
[18]A.Griesser, S. D. Roeck, A. Neubeckand, and L. J. V. Gool. "GPU-Based Foreground-Background Segmentation using an Extended Colinearity Criterion". *Vision, Modeling, and Visualization (VMV)*, 2005.
[19]H. Kim, R. Sakamoto, I. Kitahara, T. Toriyama and K. Kogure, "Reliability-Based 3D Reconstruction in Real Environment (Periodical style—Accepted for publication)," *ACM Multimedia*, to be published.

[20]S. Yamazaki, R. Sagawa, H. Kawasaki, K. Ikeuchi, and M. Sakauchi, “Microfacet billboarding,” *Proc. 13th Eurographics Workshop on Rendering, 2002*, pp. 175–186, 2002.
[21]A. Laurentini, "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.
[22]H. Kim, I. Kitahara, K. Kogure and K. Sohn, "A Real-time 3D Modeling System Using Multiple Stereo Cameras for Free-viewpoint Video Generation," *Proc. ICIAR,* pp. 237-249, Sep. 2006.
[23]W.E. Lorensen and H.E. Cline, “Marching Cubes: A High Resolution 3D Surface Constructing Algorithm,” *Computer Graphics* (Proc. SIGGRAPH), pp. 163-169. July 1987.
[24]H. Kim, R. Sakamoto, K. Kogure and I. Kitahara, "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 .
[25]J. Bresenham, “Algorithm for Computer Control of a Digital Plotter”. *IBM Systems Journal* 4, 1 (1965), 25-30.