Hardware-Accelerated Visual Hull Reconstruction and Rendering

Ming Li Marcus Magnor Hans-Peter Seidel

Computer Graphics Group
Max-Planck-Institut für Informatik

Abstract

We present a novel algorithm for simultaneous visual hull reconstruction and rendering by exploiting off-the-shelf graphics hardware. The reconstruction is accomplished by projective texture mapping in conjunction with the alpha test. Parallel to the reconstruction, rendering is also carried out in the graphics pipeline. We texture the visual hull view-dependently with the aid of fragment shaders, such as nVIDIA's register combiners. Both reconstruction and rendering are done in a single rendering pass. We achieve frame rates of more than 80 fps on a standard PC equipped with a commodity graphics card. The performance is significantly faster than that of previously reported similar systems.

Key words: Image-Based Modeling and Rendering, Visual Hull, Hardware-accelerated Rendering, Projective Texture Mapping.

1 Introduction

For the past several years, real-time 3D object reconstruction and rendering from real scenes have become research hotspots in both computer graphics and computer vision. Promising applications, such as 3D interactive TV and immersive tele-communication, stimulate the emergence of a number of real time image-based modeling and rendering systems. Some of the most prominent examples [10, 11, 12] are all based on the concept of *Visual Hulls* [9].

The visual hull is an approximate geometry representation resulting from the *shape-from-silhouette* 3D reconstruction method [18]. It is the maximal object silhouette-equivalent to the actual 3D geometry (refer to [9] for an in-depth analysis of the visual hull). Although visual hulls have been successfully employed in the real-time systems mentioned above, the speed and quality of reconstruction and rendering still need further improvements before this technique might be applicable in consumer applications.

In this paper we present a novel algorithm for reconstruction and rendering of the visual hull by exploiting the capabilities of graphics hardware. Our method has several advantages: (1) It reconstructs a polyhedral visual hull robustly and has no voxelization artifacts. (2) Reconstruction and rendering are merged into one process. Hence, there is no latency problem between the reconstruction and rendering. (3) The algorithm is fully hardware-accelerated and shows significant performance improvements over previous methods. Frame rates reach more than 80 fps for four input silhouette images.

The remainder of this paper is organized as follows. Section 2 reviews some previous work. Section 3 and 4 present our hardware-accelerated visual hull reconstruction and rendering algorithm. Section 5 gives implementation details and compares our system's performance with similar systems. Conclusion and future research directions are discussed in the last section.

2 Previous Work

A lot of effort has been put into voxel-based reconstruction and rendering of visual hulls. An early off-line system was developed by Moezzi et al. [14]. The voxel-based approach can be accelerated by processing a set of planes instead of individual voxels. Matsuyama et al. [20] distribute the visual hull computation among a cluster of PCs. Reconstruction is achieved at interactive frame rates. However, rendering is still off-line. Kautz et al. [7] present a hardware-accelerated displacement mapping technique, which can be used as well to reconstruct and render visual hulls. They draw a stack of slice planes with texture mapping and applying the alpha test. Lok's on-line 3D reconstruction system [10] bases on a similar idea. A major drawback of voxel-based systems is that they suffer from quantization and aliasing problems.

Image-based visual hulls [11] overcome the aliasing problem by directly generating the desired view from the silhouette images. Matusik et al. [12] compute an exact polyhedral representation of the visual hull by intersecting all the cones generated from the silhouettes. The intersection computation is made more efficient by reducing it from 3D to 2D. However, reconstruction runs at a lower frame rate than rendering.

Unlike the visual hull approach, which only employs the silhouette information, *Voxel Coloring* [17] and *Space Carving* [8] check color consistency across multiple views. However, the color consistency check is

very time-consuming. Even with today's computational power, they are still too slow to be ready for real-time applications.

The visual hull can also be computed using CSG (*constructive solid geometry*) intersection of silhouette cones. Rappoport et al. [15] use the stencil buffer to render general CSG models interactively. But they do not handle textured 3D objects.

Debevec et al. [5] and Buehler et al. [3] map images from multiple viewpoints onto a 3D object using projective texturing [16]. The geometry information of the 3D object must be known *a priori*. This implicates that the real-time performance of these algorithms is only related to the rendering. Compared with them, our algorithm achieves real-time performance for the whole reconstruction and rendering process.

3 Visual Hull Reconstruction

All visual hull reconstruction algorithms take the silhouettes of an object as input. In this section, we first describe briefly how we obtain the silhouettes of the foreground object in the scene. Then the hardware-accelerated reconstruction algorithm is explained.

3.1 Silhouette extraction

The background of our scene is assumed to be static. Multiple fixed video cameras are mounted around the scene. The dots in Figure 1 indicate the camera positions. In the initialization phase of our system, we compute a background image for each view. While the system is running, the moving foreground object in the scene is segmented from the background using *image differencing* [1]. Morphological operators are then applied to fill small holes in the foreground object silhouette mask. Finally, the contour is retrieved from the silhouette as a 2D polygon. The edges of the polygon are called silhouette edges. After the silhouette edges of all viewpoints are available, we are ready to reconstruct the visual hull representation of the foreground object.

3.2 Reconstruction algorithm

With the camera calibration data, we back-project the silhouette edges to form a silhouette cone, composed of silhouette faces. Then the visual hull is computed by intersecting the silhouette cones from multiple viewpoints, illustrated in Figure 1.

We classify the intersection into two types: face-cone intersection and polygon-polygon intersection. The first one is the intersection of a silhouette face with a silhouette cone from another viewpoint. It produces one polygon on the silhouette face. When repeating this intersection for multiple silhouette cones, we obtain a set of polygons on the silhouette face. Then these polygons are used

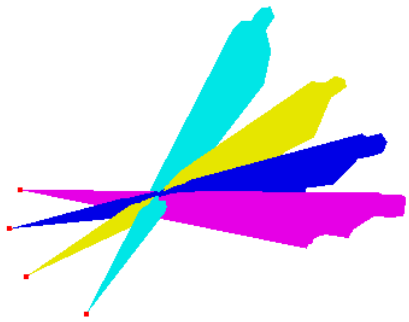

Figure 1: Intersection of silhouette cones. These cones are generated from silhouette images taken from different viewpoints.

in the second kind of intersection — polygon-polygon intersection. The result is one visual hull face. An example is shown in Figure 2a. Consider the silhouette face ABC_2. The face-cone intersection $ABC_2 \bigcap S_1$ and $ABC_2 \bigcap S_3$ produce the polygon $KLMN$ and $PQRS$, respectively (for clarity, the face-cone intersections are not drawn in the figure). By applying polygon-polygon intersection between $KLMN$ and $PQRS$, the visual hull face $PLMS$ is obtained.

In [12], the face-cone intersection is computed by projecting the silhouette face into each image plane and intersecting the projected face with the silhouette edges. These intersection results are lifted from the image planes back to the silhouette face again. Then the polygon-polygon intersection on the silhouette face is computed. The image-based visual hull technique [11] shares the same spirit in reducing the intersection computation from 3D to 2D. The difference is that the two kinds of intersections are discretized into line-polygon and segment-segment intersections. Nevertheless, both approaches carry out the intersections geometrically.

In our reconstruction algorithm, the intersection computation is performed in image space. It does not suffer from any numerical instability problems which exist for geometric intersection methods. Therefore, our algorithm works robustly for arbitrarily complex input silhouette shapes.

Given a novel viewpoint, we compute the face-cone intersection by rendering a silhouette face using projective texturing. The texture is set to the silhouette image from another view. We also load the projective texture matrix according to the calibration data associated with that view. The alpha value of the texture is set to 1 for the foreground object and 0 for the background. As a result, when the silhouette face is rendered, an alpha value of 1 is only assigned to the intersection part. The other part can be removed by enabling the alpha test. Figure 2b

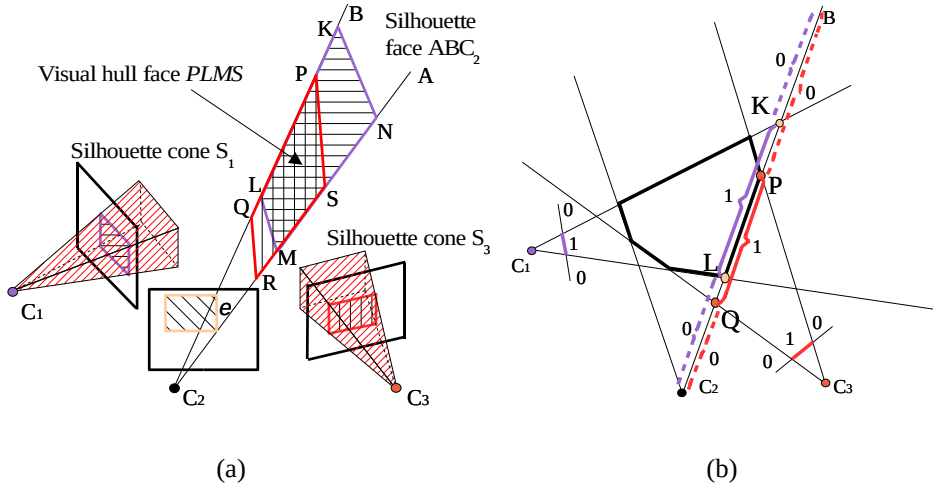

Figure 2: The principle of our visual hull reconstruction. C_1,C_2,C_3 are source cameras locations. We assume the visual hull is reconstructed from these 3 views. (a) Face-cone intersection and polygon-polygon intersection. The hatched area on each image plane is the silhouette of the object. For the silhouette face ABC_2, the polygons $KLMN$ and $PQRS$ are the face-cone intersection results with respect to silhouette cone S_1 and S_3. The polygon-polygon intersection between $KLMN$ and $PQRS$ produces the visual hull face $PLMS$. (b) Flatland version of Figure 2a. For view 1, the alpha value of KL is textured to one. For view 3, the alpha value of PQ is one. By multiplying the alpha value per-pixel, only the common part PL gets texture value 1 in the alpha channel.

illustrates the idea in 2D.

The polygon-polygon intersection is accomplished by multiplying the alpha channels of the projective textures from all views except for the view $\widehat{k}$ that produces the silhouette face currently being rendered. This alpha modulation can be expressed as:

$$A = \prod_{\substack{k=1 \\ k \neq \widehat{k}}}^{N} A_k \,, \qquad (1)$$

where N is the total number of input images, and A_k denotes the alpha channel of the ***k***-th projective texture. This formula tells us that only the intersecting region of all polygons coming from the face-cone intersection has the alpha value 1. Again we use the alpha test to retrieve this intersection region.

So far we have described the intersections for only one silhouette face of a silhouette cone. When we iterate this computation over all silhouette faces of all silhouette cones, we obtain the intersection result of multiple silhouette cones, the visual hull. For example, in Figure 2b, the area enclosed by thick lines is a cross-section of the resulting visual hull. Since the reconstruction is coupled with the rendering process, we give the reconstruction algorithm in pseudo-code together with the rendering algorithms (Figure 3 and Figure 5) in the next section.

4 Visual Hull Rendering

4.1 Flat-shaded visual hull

The flat-shaded visual hull (see Figure 4) is useful to visualize which surfaces on the visual hull comes from which silhouette cone. This leads to a better understanding of the visual hull reconstruction, and helps to perform acquisition planning for a visual-hull based visualization system.

In order to render a flat-shaded visual hull, we load the silhouette images as intensity textures into the texture units on the graphics card. In RGBA format, the texture color for the foreground object is (1, 1, 1, 1), and for the background it is (0, 0, 0, 0). As explained in Section 3, for each silhouette face, if we set the alpha value of each vertex to 1 and the texture environment to *GL_MODULATE*, the visual hull reconstruction will be done in the alpha channel of the rendering pipeline. For the color channel in the same rendering process, when we specify a different color for each silhouette cone, the *GL_MODULATE* texture environment automatically generates the desired result, assigning the different color to the visual hull faces. Reconstruction and rendering are performed simultaneously. Figure 3 gives the pseudo-code for the rendering algorithm. Figure 4 shows the rendering result of the flat-shaded visual hull. In this figure, we can observe some aliasing artifacts along the intersection boundaries due to the discrete nature of the image space computa-

Set and enable alpha test
foreach view *i*
  Load the silhouette mask image *i* as an intensity texture T_i and enable it
  Set up the projective texture matrix for T_i
  Set the texture environment to GL_MODULATE
End foreach
foreach silhouette cone S_i
  Disable texture T_i
  Set the color associated with the cone S_i
  foreach silhouette face(triangle) $\triangle_j$ of S_i
    Draw $\triangle_j$
  End foreach
  Enable texture T_i
End foreach

Figure 3: Flat-shaded visual hull reconstruction and rendering algorithm.

tion. By using higher-resolution input images these artifacts can be easily alleviated.

Figure 4: Flat-shaded visual hull. Different colors represent different silhouette cones. The visual hull is rendered from four input images.

4.2 Textured visual hull

To achieve realistic rendering results, the 3D object must be textured with the color images. In the context of image-based modeling and rendering, original images taken from different viewpoints are often used to map onto the recovered 3D geometry. This way, the user can freely choose a novel viewpoint to examine the object, which resembles its counterpart in the real world. One technical problem to be tackled here is that one single image normally cannot cover all surfaces of the object. Therefore, we must stitch multiple textures. For the parts covered by more than one texture, we blend them to achieve a smooth appearance.

Multiple texture blending

If we have ***N*** input images, the rasterized fragment color C_f for a visual hull face ***f*** can be computed using the following formula:

$$C_f = \left[\sum_{k=1}^{N} V_{k,f} * W_k * T_k\right] / \sum_{k=1}^{N} V_{k,f} * W_k \,, \qquad (2)$$

where T_k is the texture color from the ***k***-th input image, W_k is a weighting factor (see below) and $V_{k,f}$ is the visibility function for face ***f*** with regard to view *k*:

$$V_{k,f} \quad = \quad \begin{cases} 1 & , \text{ f is visible from view k} \\ 0 & , \text{ otherwise} \end{cases} \qquad (3)$$

There are two points that need to be clarified for this visibility function. First, among the reference views, there exists one view $\hat{k}$, from which face ***f*** is generated. In this case, we state that ***f*** is invisible from view $\hat{k}$ (namely, $V_{\hat{k},f} = 0$) since the view $\hat{k}$ does not contribute to the reconstruction and rendering of face ***f***.

Secondly, the surface normal can be used to compute the visibility if the silhouette cone is convex. For concave geometry, partially visible or self-occluded faces may occur. This visibility issue can be addressed by either clipping the silhouettes faces in advance or using shadow mapping [16].

We achieve view-dependent texturing [5] by including the weighting factor W_k in Equation 2. The criterion to choose the weight is the angle deviation between the viewing direction of the reference view and that of the target view. A smaller angle gets higher weight. This function can be defined as:

$$W_k = 1/acos(\bar{d_k} \bullet \bar{d_t}) \,, \qquad (4)$$

where $\bar{d_k}$ and $\bar{d_t}$ represent the reference and target viewing direction, respectively.

In Equation 2, there is an expensive per-pixel division operation. In order to avoid this, we first normalize $V_{k,f} * W_k$:

$$\widehat{W_{k,f}} = V_{k,f} * W_k / \sum_{k=1}^{N} V_{k,f} * W_k \qquad (5)$$

Then by substituting Equation 5 into Equation 2, we obtain the multiple texture blending function which is going to be evaluated in graphics hardware:

$$C_f = \sum_{k=1}^{N} \widehat{W_{k,f}} * T_k \qquad (6)$$

Rendering algorithm

In order to implement multiple texture blending, we need a more complex fragment coloring mechanism than the simple OpenGL texture environment. Fortunately, the OpenGL extension *Register Combiners* [4] provides a flexible way to compute per-fragment color. This is used to implement our rendering algorithm.

The register combiners take interpolated colors, filtered texel values, and some other registers as input. After some computations are performed in a number of general combiner stages, a final combiner stage output an RGBA value for each fragment. On the Geforce3 graphics card, four texture units are available. This means we are able to handle four silhouette images in one rendering pass. To evaluate the fragment color expressed in Equation 6, we make use of four general combiner stages and the final combiner stage.

For each vertex of a silhouette face $\boldsymbol{f}$, we encode the normalized weights in the color/alpha channel of the primary color and the red/green channel of the secondary color[1]. At the first general combiner stage, we use the dot product to separate the weights in the secondary color into two registers. For the general combiner stage $k\,(k = 2, 3, 4)$, we modulate the texel value T_k with $\widehat{W_{k,f}}$ and perform the accumulation at the same time. The final stage adds the contribution from $\widehat{W_{1,f}} * T_1$. This way, multiple textures are blended together with appropriate weights to produce the color values of the silhouette face.

The above register combiner configuration is used only for the color evaluation of the silhouette face. In addition, we must compute the alpha value and enable the alpha test to remove the extra part of the silhouette face. The alpha portion of the register combiners is configured to simulate the *GL_MODULATE* functionality, which is used in reconstructing flat-shaded visual hull. Again, reconstruction is carried out on-the-fly parallel to rendering.

The rendering algorithm is presented in Figure 5. We show snapshots of the rendering results from two novel viewpoints in Figure 6. Since the separation of foreground objects from background is not perfect, some black pixels can be observed along the silhouettes on the person. However, this problem can be fixed by using a better segmentation algorithm.

5 Implementation and Results

The proposed algorithm has been implemented in a real-time visualization system. The system architecture follows a client-server paradigm. Image acquisition and silhouette extraction are performed on the client machines. This provides good scalability and allows us to use more cameras for acquisition without substantially decreasing overall system performance.

[1]The function glSecondaryColor only accepts 3-component color. Therefore, we cannot encode the weights in the same way as the primary color.

Set and enable alpha test
foreach view *i*
  Load the silhouette color image *i* as an RGBA texture T_i and enable it
  Set up the projective texture matrix for T_i
End foreach
Configure Register Combiners and enable this extension
foreach silhouette cone S_i
  foreach silhouette face(triangle) $\triangle_j$ of S_i
    Compute visibility and weight vector for all views
    Compute normalized weight vector
    Encode normalized weight vector in the primary and secondary color of each vertex of $\triangle_j$
    Draw $\triangle_j$
  End foreach
End foreach

Figure 5: Textured visual hull reconstruction and rendering algorithm.

We use four Sony DFW500 FireWire cameras connected to four client computers which communicate with the server via a standard TCP/IP network. All cameras are calibrated in advance, and video acquisition is synchronized at run-time. The server is a P4 1.7GHz dual-processor machine with a GeForce3 graphics card. The clients are Athlon 1.1GHz computers. The video images are acquired at 320x240-pixel resolution.

We have carried out experiments using four video streams. Each 2D silhouette polygon consists of 100 to 120 edges. The resolution of the rendered novel view is set to 640x480 pixels. Without background rendering, we achieve 124 fps for the flat-shaded visual hull and 84 fps for the textured rendering. Video clips demonstrating the fast reconstruction and rendering results are available on our website `http://www.mpi-sb.mpg.de/~ming/DynaVisualHull.html`. For the time being, the only bottleneck of our system is the speed of synchronized video acquisition at 15 fps.

A performance comparison between our system and similar systems is given in Table 1. Our algorithm shows considerable improvements regarding reconstruction and rendering performance.

Notice that our rendering algorithm is not limited only to nVIDIA graphics cards or the OpenGL API. Similar multi-texture blending computations can be performed on ATI's graphics cards as well by using the OpenGL exten-

	volume (voxels)	number of cameras	input image resolution	processing power	frame rate (fps)
IBVH	N/A	4	256x256	quad 500MHz PC(4x600MHz PC)	8
GVE	2m x 2m x 2m(Res.:2cm)	9	640x480	unknown(9x600MHz PC)	offline(8.77)
OMR	8ft x 6ft x 6ft (Res.:1cm)	5	720x486	SGI Reality Monster	12-15
PVH	N/A	4	320x240	dual 933MHz PC(4x600MHz PC)	30(15)
HAVH	2m x 2m x 3m	4	320x240	dual 1.7GHz PC(4x1.1GHz PC)	84

Table 1: Performance comparison. In the column "processing power", if the system is in client-server mode, we distinguish the rendering server from the clients. The machines listed in parentheses are client PCs. For the frame rate, the number in parenthesis is the reconstruction frame rate when rendering is decoupled from reconstruction. ***IBVH****: Image-Based Visual Hulls [11].* ***GVE****: Generation, Visualization and Editing of 3D Video [20].* ***OMR****: Online Model Reconstruction for Interactive Virtual Environments [10].* ***PVH****: Polyhedral Visual Hulls for Real-Time Rendering [12].* ***HAVH****: Our system. Hardware-Accelerated Visual Hull Reconstruction and Rendering.*

(a)

(b)

Figure 6: Textured visual hull. (a) novel front view. (b) novel back view. The visual hull is rendered by blending the textures from multiple viewpoints. The background scene is modeled as a box. Four reference views are used.

sion ATI_fragment_shader [6]. The DirectX 8 API also provides an alternative for implementing our algorithm.

In our current implementation, the number of input silhouette images is restricted to the maximum of texture units available on the graphics hardware. To allow for more input images, one possible way is to divide the rendering process into multiple passes. Thus, the blending functionality of the frame buffer can serve the purpose of accumulating the individual rendering results. However, for the multi-pass rendering, the frame rate drops down as the pass number increases. Fortunately, the next generation graphics hardware, e.g. the Radeon 9700 from ATI or the GeForce FX from nVIDIA, supports eight pixel pipelines. This means eight input images can be processed in one single pass. A plot in the work [13] shows that by using eight silhouettes, the rendering quality of visual hulls improves considerably. Therefore, the new graphics hardware is capable of providing convincing visual hull rendering results in real-time.

6 Conclusions and Future Work

In this paper we have presented a new hardware-accelerated algorithm to reconstruct and render a polyhedral visual hull from multiple-view video streams in real time. While the acquisition system is implemented using several PCs, the reconstruction and rendering algorithm itself runs on only one PC equipped with a common graphics card. We support two different styles of rendering: flat-shaded and textured visual hull. The former one provides an intuitive visualization of the reconstruction process of visual hulls, whereas the latter yields realistic rendering results ready for various applications. Thanks to the fast progress in graphics hardware development, the reconstruction and rendering speed is greatly increased compared with the performance of previously reported similar systems.

An obvious improvement to our system will be to incorporate more elaborate blending schemes such as feathering [19] to smooth the transition from one texture to another. The more complex computation will be alleviated by the introduction of generally programmable fragment shaders [2] on the new graphics hardware.

We will also enhance the algorithm with fully-correct visibility handling by using shadow mapping, as suggested in Subsection 4.2.

The rendering of the background scene is another important component of the system because it gives the user the sense of immersion. So far we apply multiple background rendering passes. With more texture units available on future graphics hardware, it is not difficult to reduce the number of passes to one.

References

[1] M. Bichsel. Segmenting simply connected moving-objects in a static scene. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(11):1138–1142, November 1994.

[2] OpenGL Architectural Review Board. ARB_fragment_program OpenGL extension. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.

[3] Chris Buehler, Michael Bosse, Leonard McMillan, Steven J. Gortler, and Michael F. Cohen. Unstructured lumigraph rendering. In *SIGGRAPH'01 Proceedings*, pages 425–432, August 2001.

[4] NVIDIA Corporation. NV_register_combiners OpenGL extension. http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners.txt.

[5] Paul E. Debevec, George Borshukov, and Yizhou Yu. Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Rendering Workshop*, pages 105–116, June 1998.

[6] ATI Technologies Inc. ATI_fragment_shader OpenGL extension. http://oss.sgi.com/projects/ogl-sample/registry/ATI/fragment_shader.txt.

[7] J. Kautz and H.-P. Seidel. Hardware accelerated displacement mapping for image based rendering. In *Graphics Interface 2001*, pages 61–70, June 2001.

[8] K. N. Kutulakos and Steven Seitz. A theory of shape by space carving. In *Technical Report TR692, Computer Science Dept., U. Rochester*. May 1998.

[9] Aldo Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Machine Intell.*, 16(2):150–162, February 1994.

[10] Benjamin Lok. Online model reconstruction for interactive virtual environments. In *Proceedings 2001 Symposium on Interactive 3D Graphics*, pages 69–72, March 2001.

[11] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. In *SIGGRAPH'00 Proceedings*, pages 369–374, July 2000.

[12] Wojciech Matusik, Chris Bueler, and Leonard McMillan. Polyhedral visual hulls for real-time rendering. In *Proceedings of Twelfth Eurographics Workshop on Rendering*, pages 115–125, June 2001.

[13] Wojciech Matusik, Hanspeter Pfister, Addy Ngan, Paul Beardsley, Remo Ziegler, and Leonard McMillan. Image-based 3D photography using opacity hulls. In *SIGGRAPH'02 Proceedings*, pages 427–437, July 2002.

[14] Saied Moezzi, Arun Katkere, Don Y. Kuramura, and Ramesh Jain. Reality modeling and visualization from multiple video sequences. *IEEE Computer Graphics and Applications*, 16(6):58–63, November 1996.

[15] Ari Rappoport and Steven Spitz. Interactive Boolean operations for conceptual design of 3-D solids. In *SIGGRAPH'97 Proceedings*, pages 269–278, August 1997.

[16] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH'92 Proceedings*, pages 249–252, July 1992.

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

[18] Richard Szeliski. Rapid octree construction from image sequences. *CVGIP: Image Understanding*, 58(1):23–32, July 1993.

[19] Richard Szeliski and Heung-Yeung Shum. Creating full view panoramic image mosaics and environment maps. In *SIGGRAPH'97 Proceedings*, pages 251–258, August 1997.

[20] Matsuyama Takashi and Takai Takeshi. Generation, visualization, and editing of 3d video. In *1st International Symposium on 3D Data Processing Visualization and Transmission (3DPVT'02)*, pages 234–245, June 2002.