Improved Hardware-Accelerated Visual Hull Rendering

Ming Li,Marcus Magnor and Hans-Peter Seidel

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, D-66123, Saarbrücken, Germany
ming,magnor,hpseidel@mpi-sb.mpg.de

Abstract

The visual hull is an efficient shape approximation for the purpose of reconstructing and visualizing dynamic objects. Recently, rapid progress in graphics hardware development has made it possible to render visual hulls from a set of silhouette images in real-time.

In this paper we present several new algorithms to improve the generality and quality of hardware-accelerated visual hull rendering. First, a multi-pass approach employs texture objects and the stencil buffer to enable the visual hull rendering algorithm to deal with arbitrary numbers of input images. Secondly, flexible programmability of state-of-the-art graphics hardware is exploited to achieve smooth transitions between textures from different reference views projected onto visual hulls. In addition, visibility problems with projective texture mapping are solved by using the shadow mapping technique. We test our rendering algorithms on various off-the-shelf graphics cards and achieve real-time frame rates.

1 Introduction

In the past few years, reconstructing and visualizing dynamic objects in real scenes has been attracting a lot of attention, and it finds its way into a wide range of applications like 3D TV, interactive games, telepresence etc. The concept of the visual hull [7] was introduced by Laurentini to characterize the best geometry that can be reconstructed from the *shape-from-silhouette* method [14]. A number of systems prove that the visual hull is an efficient shape approximation for the purpose of reconstruction and visualization [13, 11, 12, 9, 21].

Recently, the rapid development of graphics hardware has made it possible to render textured visual hulls directly from a set of silhouette images in real-time. The hardware-accelerated algorithm [8] is such an example. However, in the original rendering algorithm, the number of input silhouette images is limited to the number of texture units available on graphics hardware. The quality of rendering is also restricted by hardware resources and programmability.

In this paper we present several new algorithms to improve the generality and quality of hardware-accelerated visual hull rendering. First, a multi-pass approach employs texture objects and the stencil buffer to allow using arbitrary number of silhouette images as input for rendering. Secondly, flexible programmability of state-of-the-art graphics hardware is exploited to achieve smooth transitions between textures from different reference views projected onto visual hulls. In addition, the "projecting-through" artifacts in projective texture mapping are removed by combining the shadow mapping technique. All our rendering algorithms run at real-time frame rates on a variety of commodity graphics hardware.

The rest of the paper is organized as follows. After a review of some related research areas, Section 3 briefly describes the basic idea of using graphics hardware to render visual hulls. Section 4 presents our new rendering algorithms in detail. Performance measurements of our algorithms tested on various graphics cards are given in Section 5. Finally, we conclude this paper and provide some ideas for future research in Section 6.

2 Related work

The basic principle of visual hull computation is fairly intuitive. Given the viewing information associated with each silhouette image known from the camera calibration, we are able to back-project each silhouette into 3D space. This generates a silhouette cone containing the actual 3D object in question.

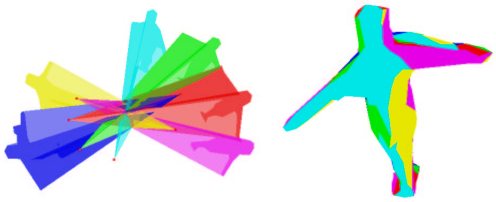


Figure 1: Visual hull reconstruction. **Left**: Six cones are generated from silhouette images taken from different viewpoints. **Right**: Reconstructed 3D surface.

Theoretically, visual hulls must be reconstructed using silhouette images from all possible views. However, in practice, we can only employ a limited number of images. From these images, an approximate visual hull can be reconstructed. Figure 1 illustrates this process.

There exist two classes of visual hull reconstruction algorithms: voxel-based and polyhedron-based. The former approach tessellates a confined 3D space into discrete volume elements, known as voxels. Then each voxel is projected onto all reference image planes. Voxels whose projection fall outside of any silhouette will be carved away. Szeliski et al. [18] use an octree data structure to accelerate the carving process. Matsuyama et al. [10] distribute the visual hull computation among a cluster of PCs to achieve interactive reconstruction rates. The main drawbacks of voxel-based approaches are the large memory requirement and quantization artifacts.

Instead of reconstructing a volumetric representation, Exact polyhedral visual hulls can be computed by performing 3D intersection of all cones generated from the silhouettes. Matusik et al. [12] make the computation more efficient by reducing the intersection from 3D to 2D. However, the polyhedron-based approach involves a lot of geometric computation. The reconstruction speed drops drastically when more reference views or complex silhouette shapes are used. Moreover, the computation often runs into numerical stability problems.

If the goal is rendering visual hulls from novel viewpoints, the reconstruction does not need to be explicit. The image-based visual hull technique [11] embeds the reconstruction in the rendering process to directly generate the desired view from silhouette images. However, the whole computation is a pure software solution, and no hardware acceleration is used.

Kautz et al. [6] present a hardware-accelerated technique to render an object from several color-plus-depth images without reconstructing the actual object. This technique can be used to render visual hulls as well. The implicit 3D reconstruction is done by drawing slice planes from different orthogonal views with texture mapping. The alpha test and the stencil test serve to discard fragments not belonging to the object. Lok's on-line 3D reconstruction system [9] is based on a similar idea except that perspective views are used. Rather than drawing a large number of planes through a volume, in a recent paper [8] we propose a hardware-accelerated algorithm in which visual hulls are rendered from only a few silhouette cones with projective texture mapping. As a result, much higher frame rates are achieved.

3 Basic hardware-accelerated visual hull rendering

The basic idea of using graphics hardware to render visual hulls is to trim silhouette cone surfaces with alpha maps. First, the silhouette images are loaded as RGBA textures. The alpha values of the textures are set to 1 for the foreground objects and to 0 for the background. Given a novel viewpoint, each silhouette cone is projectively textured by the silhouette images from all other views. In the texture units, different computations are carried out for the alpha and the color channel.

In the alpha channel, alpha values from multiple textures are modulated as shown in Figure 2. The result is that on a silhouette cone, only the region projected by all the other views gets the alpha value 1. This region is actually the reconstructed visual hull surface. The rest of the region on the cone has the alpha value 0 and will be discarded by enabling the alpha test. Parallel to the alpha computation, color values are calculated as a weighted sum in order to blend color contributions from different views. The weighting factor considers view-dependent effects and the visibility of each silhouette face.

When the silhouette cones from all reference views are rendered, a textured visual hull is finally generated for the novel viewpoint.

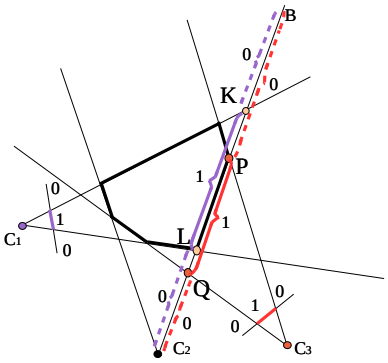


Figure 2: The alpha channel computation of hardware-accelerated visual hull rendering (illustrated in 2D). The silhouette cone extruded from view C_2 is being rendered. When applying projective texturing from view C_1, each alpha value belonging to KL is textured to 1. The projective texture from view C_3 assigns the alpha value 1 to PQ. By multiplying the alpha values per-pixel, only the common part PL gets the alpha value 1.

4 Improved hardware-accelerated visual hull rendering algorithms

The basic visual hull rendering algorithm has several limitations. First, the upper bound on the number of input reference images is equal to the number of texture units. As a result, with a graphics card that only has two or four texture units, the reconstruction is coarse and the rendering quality is limited. Secondly, the programmability of the graphics hardware used before is not flexible enough to do complex per-fragment computations, which is critical for producing high-quality rendering results. In this section, we are going to present several advanced rendering algorithms to address these issues. For simplicity, the OpenGL [16] terminology is used in the following explanations, but the algorithms presented here also apply to other graphics libraries such as Direct3D.

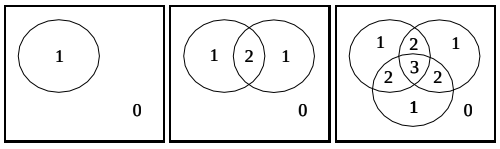
4.1 Multi-pass hardware-accelerated visual hull rendering

The basic visual hull rendering algorithm assigns each silhouette image and projective texture matrix to a fixed texture unit. In multi-pass rendering, all silhouette images are loaded as texture objects. During each rendering pass, we can switch to the appropriate set of textures and their corresponding projective matrices.

Within each rendering pass, the alpha values from different texture units can be multiplied by utilizing the simple programmability of graphics hardware, exposed in the form of OpenGL extensions such as NV_register_combiners [3]. Between multiple rendering passes, we use stencil testing to modulate the alpha multiplication results from different passes. In the OpenGL graphics pipeline, the alpha test is performed before the depth test. Therefore, the alpha test can control the update of the stencil value indirectly through its influence on the depth test. We exploit this fact to use the stencil buffer to record the alpha multiplication results of each rendering pass.

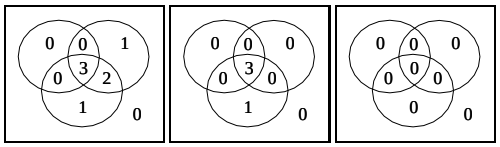
Initially, the stencil values in the stencil buffer are all cleared to 0. The alpha test is set to "GL_GREATER than 0". The depth test is set to GL_ALWAYS. While rendering, the stencil comparison is configured to test whether the stencil value equals the index number of the current rendering pass. The stencil update operation is specified in such a way that the stencil value is increased by 1 only if both the depth test and the stencil test are passed. An example shown in Figure 3 clarifies the settings. In the first pass, the index number of rendering is 0. All fragments pass the stencil test because the initial stencil value is 0. On the other hand, only the fragment with the alpha value 1 survives the alpha test and hence passes the later depth test. The overall effect is that the stencil value is updated to 1 where the alpha value is 1 (see Figure 3a). In the second rendering pass, the current pass number becomes 1. According to the stencil test setup, the stencil value increases to 2 only for the fragment which has the stencil value 1 and the alpha value 1 (see Figure 3b). After the third rendering pass the stencil buffer will look like Figure 3c. The region where the alpha values equal 3 is the modulation result of alpha values from all three passes.

Compared to the alpha channel, the computation in the color channel is less complex. We need to accumulate the weighted sum results from all color rendering passes. This is done by setting the frame buffer blending function to GL_ADD and blending



(a) (b) (c)

Figure 3: Stencil buffer update for modulating the alpha results from multiple rendering passes. The three figures show the stencil buffer after the first, second and third alpha rendering pass, respectively.



(a) (b) (c)

Figure 4: Stencil buffer clearing during the color rendering passes. The three figures show the stencil buffer after the first, second and third color rendering pass, respectively. Each color rendering pass corresponds to the alpha rendering pass used for stencil buffer update in Figure 3.

factors to 1. The content in the stencil buffer generated by all alpha rendering passes serves as a mask for the color rendering.

The rendering passes needed by either the alpha or the color computation can be determined from the number of reference views and the number of available texture units using the following formula:

$$nPassNum = \left\lceil \frac{nRefViews - 1}{nTextureUnit} \right\rceil$$

The total number of rendering passes required to render visual hulls is simply twice that value, namely $2 \bullet nPassNum$.

Since we have to use additional passes for the color computation anyway, we can take this opportunity to reset the stencil values, modified during the alpha channel computation. This avoids clearing the whole stencil buffer for each visual hull surface, which is expensive. Notice that, for the first $(nPassNum - 1)$ passes, the regions tagged as the visual hull surface by the alpha rendering passes should be kept intact in order to continue to serve as a mask for the subsequent color rendering pass. Figure 4, corresponding to Figure 3, illustrates how to clear the stencil buffer.

Although the alpha and color rendering passes are separated, the rendering cost is not doubled. This is because the alpha rendering passes are used only for the stencil generation. So color and depth buffer writing can be disabled for these passes.

4.2 Hardware-accelerated visual hull rendering using fragment programs

Graphics hardware is in the midst of a revolutionary transition. Traditional fixed-function graphics pipeline evolves into a fully programmable rendering engine. A fragment program [1] is a sequence of assembly-like instructions that can compute color and depth values for each fragment. It takes the fragment attributes (position, colors, texture coordinates) and a set of constant registers as input. Within the program, mathematical computations and texture lookups can be performed. In the following, we will exploit the flexible programmability of state-of-the-art graphics hardware to enhance the visual hull rendering.

4.2.1 Smooth texture transition using distance transformation

In hardware-accelerated visual hull rendering, artifacts are most noticeable along the silhouette boundaries. One reason is that the segmentation algorithm is not powerful enough to deliver a perfect foreground object mask. The other important reason is that the applied multi-texture blending scheme is too simple due to limited programmability. In this scheme, on each visual hull face, the view-dependent weighting factor remains constant for each silhouette texture. New graphics hardware allows the weighted average to be computed in a per-fragment manner. Therefore, we can exploit this capability to devise a more sophisticated blending scheme, which considers not only the constant view-dependent weighting factors but also a varying weighting factor for each texel within a texture.

The idea of using varying weighting factors to make smooth transition is not new. Szeliski et al. [19] apply this idea to mosaic a panoramic view from a set of images. In order to hide the boundary seams between overlapped images, they define a bilinear weighting function so that for each image the

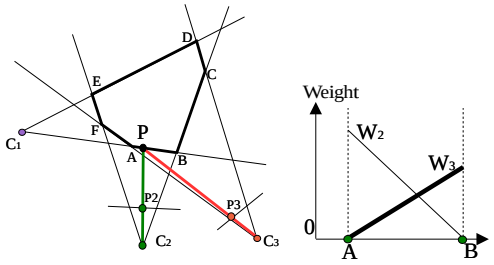


Figure 5: Smooth texture transition on a visual hull using variable weighting factors. **Left**: ABCDEF is a 2D visual hull reconstructed from three views. The line segment AB is textured by the image associated with C_2 and C_3. To smooth the transition, if a point P is closer to A, its projection p_2 should get a bigger weight, while its projection p_3 should get a smaller weight. **Right**: Varying weight factors defined for the line segment AB. The thicker line represents the weighting function defined for the texture associated with view C_3. The thinner line represents the weighting function defined for the texture associated with view C_2.

weight is 1 in the center and fades to 0 at the image borders. When a weighted average is computed for each pixel in the overlapped region, the boundary seams are completely removed and the smooth transition between different images is achieved. For visual hull rendering in the 3D case, this idea is also applicable. For the sake of clarity, we illustrate how this works using a 2D visual hull slice, as shown in Figure 5.

Analogous to the bilinear weighting function used by Szeliski, we need to define a weighting function for each pixel in the silhouette mask. The criterion is that the weight is bigger for the pixels far from the silhouette boundary and drops to 0 at the boundary. The distance transformation is such a function suitable for our purpose. For each pixel in a binary image, the distance transformation [2] calculates the distance from the pixel to the nearest zero pixel. Mathematically, it can be expressed in the following formula:

$$W(p) = min\{dist\,(p\,,q)\,,\ q \in R\}$$

The notation p and q are pixel positions. R stands for the region where the pixel values are 0. $dist$ is

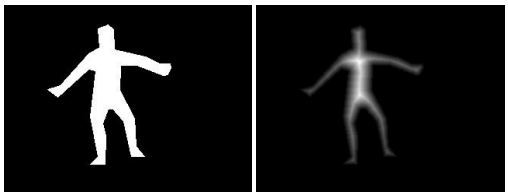


Figure 6: Distance transformation. **Left**: Original foreground object mask. **Right**: Foreground object mask after applying distance transformation. It looks thinner because the pixels close to the contour are too dark to be seen.

the distance between the pixel p and q. If we apply this transformation on the binary silhouette mask, the effect shown in Figure 6 is obtained. Notice that different silhouette masks have different maximum distance values. In order to ensure that distance values in different silhouette masks have the same range (in our case $0 - 255$), an additional scaling operation is required. The distance transformation and subsequent scaling are computed in real time using OpenCV library [5].

Instead of encoding the binary silhouette mask in the alpha channel of an RGBA texture, we encode the distance-transformed silhouette mask. In the fragment program, we multiply the view-dependent weighting factor with this distance value and then perform a weighted average of the colors from different textures. Figure 7 compares the rendering results with and without using the distance-transformed alpha map. Obviously, with the help of the distance transformation, the multi-texture blending is smoother.

4.2.2 Solving visibility problems using shadow mapping

It is well known that projective texture mapping can "project through" surfaces within the viewing volume. In order to avoid this, the basic hardware-accelerated visual hull algorithm uses the normal vectors to compute the visibility of silhouette surfaces. Then for the invisible back-facing surfaces, the visibility information is encoded in the weighting factors which we used for view-dependent texturing. This approach works fine if the reconstructed visual hull is convex, but it fails for concave surface regions.

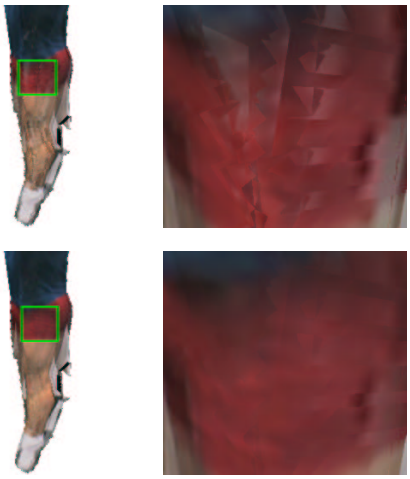


Figure 7: Comparison of rendering results with and without using distance-transformed alpha maps. **Upper left**: The rendering result without using the distance-transformed alpha map. **Upper right**: Details of the region in the box of the upper left image. **Bottom left**: The rendering result when the distance-transformed alpha map is used. **Bottom right**: Details of the region in the box of the bottom left image.

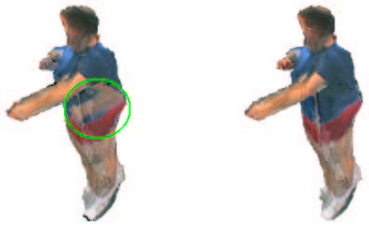


Figure 8: Comparison of rendering results with and without the use of shadow mapping. Both images are rendered from six silhouette images**.** Due to limited fragment program instruction length, distance-transformed alpha maps are not used here. **Left**: A fake arm on the body when no shadow mapping is used. **Right**: The artifact is removed when the shadow mapping technique is applied.

Shadow mapping [20] is designed originally for generating shadows. It has been implemented on common graphics hardware [17, 4] and widely used in real-time rendering systems. Sawhney et al. [15] integrate this technique into their "Video Flashlight" real-time system to solve the visibility problem of projective texture mapping. This technique can be used to enhance our hardware-accelerated visual hull rendering algorithm as well.

Because the visibility problem in projective texture mapping must be solved for all reference views, we need to generate a shadow map for each of them. This is accomplished by rendering visual hulls from each reference viewpoint. This means we need n rendering passes for rendering shadow maps for n reference views. Fortunately, the cost of these rendering passes is low because we only need the depth buffer information.

When the shadow maps are generated and loaded as textures, we can render visual hulls with correct visibility when applying projective texture mapping. One thing to point out is that every silhouette image and its associated shadow map share the same projective texture matrix, which implies that each fragment obtains the same texture coordinate for both the silhouette image and the shadow map. This fact suggests that the number of texture coordinate sets that we need is half of the number of the texture image sets. Coincidentally, latest graphics chips such as nVIDIA's GeForce FX, ATI's Radeon 9700 and Radeon 9800 support 8 texture coordinate sets and 16 texture image units. This design suits our rendering algorithm perfectly. Without the limit of the fragment program length, we are able to render from 9 reference views using such kinds of graphics hardware. However, because of the limit of the fragment program instruction length on the graphics hardware that we use, only six silhouette images can be employed to render visual hulls with shadow mapping. Figure 8 compares the rendering results with and without applying shadow mapping.

Distance transformation and shadow mapping enhancement can be applied in tandem. It is not difficult to combine them if the graphics hardware supports longer fragment programs. In addition, nothing prevents us from extending the enhanced rendering algorithms to multiple passes, which allows us to render high-quality visual hulls from more input silhouette images.

5 Results

We have built a real-time distributed visualization system to implement the proposed algorithms. Eight Sony DFW500 FireWire cameras are connected to four client computers which communicate with a rendering server via a standard TCP/IP network. All cameras are calibrated in advance. Image acquisition, silhouette extraction and distance transformation are performed on the client machines, providing good scalability. Video acquisition is synchronized at run-time, delivering video streams at about 15 fps. 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.

Our system is capable of running in on-line mode, in which the client machines acquire images, extract silhouettes, perform distance transformation and send the image data along with calibration information to the server for rendering. But in order to measure performance for different types of graphics hardware, we carry out the experiments using two synchronized video streams. A male dancer is recorded from seven views, and a female dancer is recorded from eight views. (One accompanying video clip shows the segmented female dancer video streams, the foreground object masks, and the distance-transformed foreground object masks.)

Three different hardware setups are compared in our test. Two of PCs have P4 1.8GHz CPUs, and are equipped with a Quadro2 Pro (GeForce2-class) and a Quadro4 700 XGL (GeForce4-class) graphics card, respectively. The third one has a P4 2.4GHz CPU and a Fire GL X1 (Radeon-9700-Pro-class) graphics card. The resolution of the rendered novel view is always set to 640x480 pixels.

We have tested the multi-pass rendering algorithm on the Quadro4 machine. Five reference views are used. We have executed our multi-pass rendering algorithm using the male dancer sequence on the Quadro2 machine and the female dancer sequence on the Quadro4 machine using the two video streams. The rendering speed is about 14 and 18 frames per second, respectively. Two techniques, distance transformation and shadow mapping, are used to enhance the hardware-accelerated visual hull rendering with the help of fragment programs. We have implemented both of them on the machine with the Fire GL X1 graphics card. Table 1 provides the statistics of the rendering performance.

graphics card	Fire GL X1	
number of reference views	8	6
Enhanced Technique	distance transformation	shadow mapping
frame rate (fps)	26	20

Table 1: Performance measurement of hardware-accelerated visual hull rendering using fragment programs.

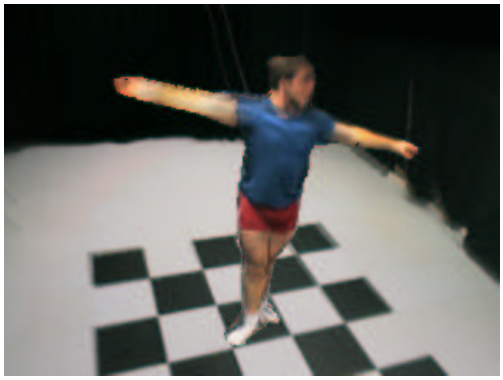


Figure 9: Visual hulls rendered in its surronding environments.

Figure 9 shows the rendered male dancer embedded in real-world surrounding environments. Two submitted video clips present 3D animations which show the female dancer in our video room and in the *Uffizi* environment, respectively. In the animations, only the distance transformation is applied because not all eight views (maximum six for Fire GL X1) can be employed when using shadow mapping.

6 Conclusions and future work

We have improved the basic hardware-accelerated visual hull rendering algorithm [8] in several aspects. The original algorithm is made more general by allowing more input silhouette images. Arbitrary numbers of silhouette images can be used as input. In addition, the rendering quality is improved thanks to the flexible programmability of

modern GPUs. Two techniques, distance transformation and shadow mapping, are employed to produce smooth rendering results and ensure that textures do not project through surfaces. We are able to generate realistic renderings of visual hulls from multi-view video streams at real-time frame rates.

Unclean segmentation impairs the rendering results considerably. A better segmentation algorithm will definitely improve the rendering quality. We also plan to test our algorithms on synthetic datasets in order to quantify rendering errors caused by the approximate geometry representation. Playing with the programmable graphics hardware, more special effects could be added to reconstructed visual hulls.

Acknowledgment

We would like to thank Harald Krytinar and Anna Hagermark for performing the dancing, Christian Theobalt for providing the video sequences.

References

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

[2] G. Borgefors. Distance transformations in digital images. *CVGIP*, 34(3):344–371, 1986.

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

[4] Wolfgang Heidrich. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. PhD thesis, University of Erlangen, Computer Graphics Group, 1999.

[5] Intel. Open source computer vision library (OpenCV). http://www.intel.com/research/mrl/research/opencv/.

[6] J. Kautz and H.-P. Seidel. Hardware accelerated displacement mapping for image based rendering. In *Proceedings of GI'01*, pages 61–70, June 2001.

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

[8] M. Li, M. Magnor, and H.-P. Seidel. Hardware-accelerated visual hull reconstruction and rendering. In *Proceedings of GI'03*, Halifax, Canada, 2003. To appear.

[9] B. Lok. Online model reconstruction for interactive virtual environments. In *Proceedings of I3D'2001*, pages 69–72, March 2001.

[10] Takashi Matsuyama and Takeshi Takai. Generation, visualization, and editing of 3D video. In *3DPVT'02*, pages 234–245, June 2002.

[11] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan. Image-based visual hulls. In *SIGGRAPH'00 Proceedings*, pages 369–374, July 2000.

[12] W. Matusik, C. Bueler, and L. McMillan. Polyhedral visual hulls for real-time rendering. In *Proceedings of EGRW'01*, pages 115–125, June 2001.

[13] S. Moezzi, A. Katkere, D. Y. Kuramura, and R. Jain. Reality modeling and visualization from multiple video sequences. *IEEE CG&A*, 16(6):58–63, November 1996.

[14] M. Potmesil. Generating octree models of 3D objects from their silhouettes in a sequence of images. *CVGIP*, 40:1–20, 1987.

[15] H.S. Sawhney, A. Arpa, R. Kumar, S. Samarasekera, M. Aggarwal, S. Hsu, D. Nister, and K.Hanna. Video flashlights – real time rendering of multiple videos for immersive model visualization. In *Proceedings of EGRW'02*, pages 163–174, June 2002.

[16] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.4)*. Silicon Graphics, Inc., http://www.opengl.org/∼developers/documentation/version1_4/glspec14.pdf.

[17] 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.

[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] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH'78 Proceedings*, pages 270–274, August 1978.

[21] S. Würmlin, E. Lamboray, O. Staadt, and M. Gross. 3D video recorder. In *Proc. of IEEE PG'02*, pages 96–103, October 2002.