Online Accelerated Rendering of Visual Hulls in Real Scenes

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

This paper presents an online system which is capable of reconstructing and rendering dynamic objects in real scenes. We reconstruct visual hulls of the objects by using a shape-from-silhouette approach. During rendering, a novel blending scheme is employed to compose multiple background images. Visibility artifacts on the dynamic object are removed by using opaque projective texture mapping. We also propose a dynamic texture packing technique to improve rendering performance by exploiting region-of-interest information. Our system takes multiple live or pre-recorded video streams as input. It produces realistic real-time rendering results of dynamic objects in their surrounding natural environment in which the user can freely navigate.

Keywords
Image-based Modeling and Rendering, Hardware-accelerated Rendering, Projective Texture Mapping, Shadow Mapping, Visual Hull Rendering.

1 INTRODUCTION

In the past few years, Image-Based Modeling and Rendering systems configured with multiple video cameras [Moe96][Ved98][Mat01] have been developed to visualize dynamic objects in real scenes. These systems enable a wide range of applications such as 3D interactive TV, computer games, immersive tele-collaboration, sports analysis, etc. In many cases, real-time performance is a critical aspect of these applications.

Recently, the shape from silhouette (*SfS*) approach [Sze93] has been successfully used in real time systems [Mat00][Lok01][Mat01]. The reconstruction result of this approach is the *Visual Hull* [Lau94], an approximate shell that envelopes the true geometry of the object. The visual hull allows real-time reconstruction and rendering, yet some improvements are still possible to obtain faster and better rendering results. In this paper, we make three contributions to the background and visual hull rendering algorithm.

We propose a novel blending scheme to hide boundary seams when composing background images. We then extend the projective texture mapping technique to remove the artifact which we call shadow leak. Finally, we make use of region-of-interest information to pack multi-view textures in order to improve rendering performance.

The remainder of this paper is organized as follows. Section 2 discusses some previous work related to our topic. Section 3 gives an overview of our system. We explain how to reconstruct the dynamic 3D model in Section 4 and describe the rendering algorithm in Section 5. After showing some results, we conclude this paper and suggest several future research directions.

2 RELATED WORK

The problem of reconstructing 3D objects from multiple images has been investigated for decades. Some model-based methods try to minimize an objective function [Deb96], which characterizes the difference between the 2D projection of parameterized features and their counterparts in real images. Such approaches usually involve laborious user interaction. Other methods like Depth from Stereo (*DfS*) [Nar98] and Shape from Silhouette (*SfS*) can be fully automated

Journal of WSCG, Vol.11, No.1., ISSN 1213-6972
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.

and are promising candidates for online 3D reconstruction systems. The *SfS* method has become quite popular recently because the result is robust compared to the noisy output of *DfS*. More importantly, a consistent 3D model can be directly obtained using *SfS*, while in the stereo case, one must post-process the depth results from different image pairs in order to obtain a consistent 3D model [Cur96].

Hardware-supported projective texture mapping is widely adopted for mapping multi-view images onto 3D objects. Debevec et al. [Deb98] describe a view-dependent texture mapping technique which combines three views for each polygon. Buehler et al. [Bue01] create a blending field and use it as weights to blend different projective textures. This approach ensures spatial continuity in the final rendering result.

Shadow mapping is an image-space technique developed by Williams[Wil78]. Segal et al. [Seg92] propose to use projective texture mapping hardware to accelerate shadow generation. Heidrich [Hei99] presents a dual-texture approach to make shadow mapping technique available on most common graphics platforms. The fast speed of shadow mapping is exploited by [Saw02] for removing the visibility artifact of projective texture mapping. This method is extended in this paper to remove another artifact — shadow leaking during projective texturing.

3 System Overview

Our system consists of six Sony DFW500 FireWire cameras arranged along a half-circle. These cameras are connected to 3 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. Fig. 1 shows a schematic diagram of our system setup.

The initialization phase of our system includes recording a background image for each camera and sending this background image together with camera calibration information to the server. After initialization, the client machines extract silhouettes using background differencing [Bic94] and deliver these silhouettes to the server for every frame. The server computes the visual hull from multiple silhouettes and renders the dynamic object together with the surrounding background.

The system architecture has been designed to distribute the computational load between the server and the clients. Image acquisition and silhouette extraction are performed on the client machines. This provides good scalability and allows us to use more cameras without significantly decreasing overall system performance. In addition to using live video streams as input, we can also record multiple synchronized video streams, which can be played back later. No matter whether the input is live-acquired or pre-recorded, the visual hull reconstruction and rendering is performed in real-time.

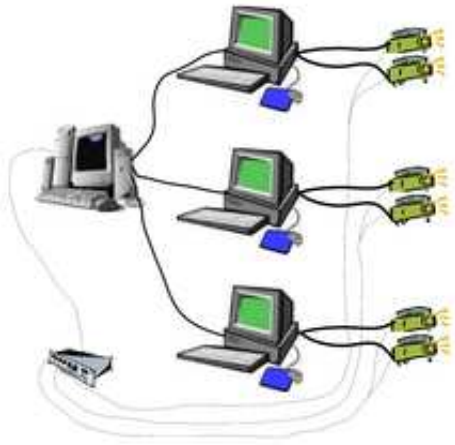


Figure 1: Schematic diagram of our system setup. The box in the lower left corner is a synchronization device which sends the synchronization signal to all cameras.

4 3D Reconstruction

There are two different approaches for visual hull reconstruction: voxel-based [Moe96] and polyhedron-based [Mat01]. The latter method has been chosen for our system because it is amenable to fast rendering.

The 3D reconstruction is rather straightforward. For each frame, the silhouette of the moving foreground object is segmented using the previously acquired background. Then we approximate its contour as a 2D polygon and transfer it to the server. On the server, silhouettes from all cameras are available. The server computes a polyhedral visual hull by intersecting back-projected cones of the silhouettes, as illustrated in Fig. 2. An open source library is used to compute the 3D intersection [Bek].

We have a tunable parameter for controlling the precision of the 2D polygon approximation. The server can notify all clients and demand a different resolution of 2D contours. This leads to a change in coarseness of the visual hull model. The on-line precision adjustment is a convenient tool for the user to choose the trade-off between reconstruction quality and speed.

(a) (b)

Figure 2: Reconstruction of the polyhedral visual hull. (a) Generalized cones formed by back-projecting the silhouettes. (b) The intersection result of these cones — the polyhedral visual hull. Small arrows around the visual hull indicate camera positions.

(a) (b)

Figure 3: Clipping with polygon offset. Interpenetration patterns can be seen in the left image without polygon offset. This artifact disappears in the right image when we use a different polygon offset for rendering each background image.

5 ACCELERATED RENDERING ALGORITHM

5.1 Background rendering

For simplicity, we model the surrounding background as a box with the dimensions of our image acquisition room. To render the background, we map multiple background images from different viewpoints onto this box using multi-pass projective texture mapping. Improved rendering results are achieved by clipping and blending. Rendering speed is accelerated using a display list.

5.1.1 *Clipping*

Without special treatment, the geometry behind a camera also gets textured during projective texturing. This "negative projection" can be simply removed by setting up a clipping plane for each background image as suggested in [Saw02]. The clipping plane is defined by the image plane equation, which can be derived from the camera parameters associated with that image. However, when clipping is applied to a polygon, the rasterization behavior of the polygon is changed. Thus, problems occur when we texture the geometry several times with multiple background images using different clipping planes. The same surface in different rendering passes could penetrate itself due to slightly different Z values generated from inconsistent rasterization. Severe artifacts can be observed in Fig. 3(a). By setting a different polygon offset [Seg] for each rendering pass to enlarge the difference of Z values, the penetration artifact is completely removed, as shown in Fig. 3(b).

5.1.2 *Blending*

Due to the limited field of view of each camera, one background image can only cover parts of the scene. If we simply overlay them one by one, visible artifacts between boundaries can be observed. To alleviate this artifact, we need to use blending to reduce the color contrast in the vicinity of boundaries. A simple blending scheme can be expressed as follows:

$$C = \frac{1}{N} * \sum_{k=1}^{N} C_k$$

where N is the total number of textures, C_k is the source color from each texture image, C is the final color. Since different regions of the background model are textured by different number of textures, we must compute the blending color in a per-pixel manner. However, with the limited color precision and functionality of current graphics hardware, it is impossible to accumulate the color for each pixel and then perform a final division. To overcome this problem, we propose an approximate implementation of this equal-weight blending scheme on current graphics hardware.

The basic idea is to generate a series of alpha values $A^i = \frac{1}{i+1}(i \geq 0)$ for each pixel. The color can then be computed by

$$\begin{aligned} C_d^{i+1} &= C_s^{i+1} * A^i + C_d^i * (1 - A^i) \ \ (i \geq 0) \\ &= C_s^{i+1} * \frac{1}{i+1} + C_d^i * (1 - \frac{1}{i+1}) \end{aligned} \quad (1)$$

C stands for pixel ***c**olor*. The subscript d means ***d**estination* color, and s denotes ***s**ource* color. The superscript indicates the index number of background image. This notation is also used in the rest of this subsection.

In order to give a feeling for how it works, let us give an example. Initially, the destination color is $C_d^0 = 0$. If this pixel will be textured by 3 images, according to Eq. 1, the final pixel value C_d^3 can be computed as follows:

$$\begin{aligned} C_d^1 &= C_s^1 \\ C_d^2 &= C_s^2 * \frac{1}{2} + C_d^1 * (1 - \frac{1}{2}) \\ &= C_s^2 * \frac{1}{2} + C_s^1 * \frac{1}{2} \\ C_d^3 &= C_s^3 * \frac{1}{3} + C_d^2 * (1 - \frac{1}{3}) \\ &= C_s^3 * \frac{1}{3} + C_s^2 * \frac{1}{3} + C_s^1 * \frac{1}{3} \end{aligned}$$

As expected, each source color gets a weight of $\frac{1}{3}$. Here the key problem is how to generate the $\frac{1}{i+1}$ series. Although currently there is no graphics hardware available to compute the exact solution, we can use the following equation to approximate this series:

$$A^{i+1} = 0.06 + \frac{A^i}{2} \quad (i \geq 0) \tag{2}$$

This is an inhomogeneous linear recurrence equation with constant coefficient [Bal91]. The boundary condition is $A^0 = 1$. Eq. 2 produces a sequence of real numbers: 0.56, 0.34, 0.23, 0.175, 0.1475$\cdots$. Including the boundary condition, we get an approximation of $\frac{1}{i+1}(i \geq 0)$. The constant value 0.06 in Eq. 2 is adjustable. An optimal value might be obtained in the sense of lease squares. We have tried several values ranging from 0.05~0.07. The difference between the rendering results are indistinguishable.

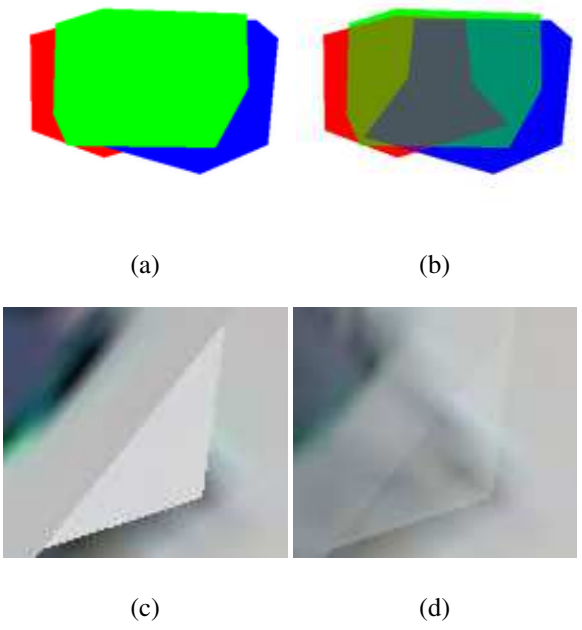
Now that the mathematics is clear, the implementation is straightforward. Like [Pee00], we treat each OpenGL rendering pass as a SIMD instruction. We use the OpenGL blending function to compute the recurrence sequence. Initially, the frame buffer is cleared using color $(0, 0, 0, 1)$ and the alpha channel of all background images is set to 128 (i.e. 0.5 for the source alpha value). The border alpha value of each image is set to zero so that we can use the alpha test to kill fragments outside the field of view. For each background image, we render the scene two times. First we enable the RGB channel and disable the alpha channel. The RGB colors are composed using the blending function as follows:

$$C = C_s * DST\_ALPHA + C_d * (1 - DST\_ALPHA)$$

During the second pass, we disable the RGB channel and enable the alpha channel. The blending equation is set to:

$$A = A_s * 0.12 + A_d * SRC\_ALPHA \tag{3}$$

The notation used in the above two blending equations is adopted from [Seg]. SRC_ALPHA is the alpha value of the incoming fragment, whereas DST_ALPHA is the alpha value of the fragment in the current frame buffer. Eq. 3 evaluates the exact function in Eq. 2 because of $A_s = SRC\_ALPHA = 0.5$. The 2-pass rendering for each background image can be reduced to one pass when the recently-approved OpenGL 1.4 standard [Seg] is supported by the graphics card. There is a new function *glBlendFuncSeparate*, which allows to specify the blending factor separately for color and alpha channel. In Fig. 4, (a) and (b) give an extreme example to illustrate the effectiveness of our blending algorithm. (c) and (d) show the blending result for real background images.



(a) (b)

(c) (d)

Figure 4: Background blending. (a)(b) Using red, green, blue background images. (c)(d) A carpet corner in the real scene. (a) Without blending. (b) With blending. In the region where there is only one texture image available, the final color is the one of that texture. In the region covered by two images, they are blended using the weights 1/2. The blending weights become 1/3 in the region of three images. (c) Without blending. (d) With blending.

5.1.3 *Display List*

Since the surrounding background is static, we can compile the background rendering commands into a display list. This display list only needs to be updated when the number of background images changes. According to our experiment, when we use 4 background

images (resolution:320x240 pixels), the frame rate usually increases by about 20% compared to rendering without using display list.

5.2 Opaque projective texture mapping

Debevec et al. [Deb98] suggest that shadow maps can be used to solve the visibility problem in projective texture mapping. But because of lacking hardware support at that time, they resort to a pre-processing step to determine the visibility on a per-polygon basis. Recently, [Saw02] implemented the shadow-map-based visibility algorithm by employing state-of-the-art graphics hardware. We refer to this algorithm as *opaque projective texture mapping*. In practice, visible artifacts remain when we apply this technique to the reconstructed dynamic visual hull. As seen in Fig. 5(b), the front surfaces do not occlude the back part completely, and the contour leaks onto occluded surfaces. We call this artifact *shadow leak*. In the following, we first give a brief description of the *opaque projective texture mapping* algorithm and then extend it to solve the shadow leak problem. Moreover, we use the geometry information of the visual hull to achieve optimal precision for the shadow map.

5.2.1 *Basic rendering algorithm*

The basic opaque projective texture mapping algorithm can be implemented using multiple texture units and the texture environment mode *GL_COMBINE*. They are both available on mainstream graphics cards. The rendering process is as follows:

For each reference view, the RGBA color image is loaded into the first texture unit. The shadow map is generated by OpenGL shadow extensions in the second texture unit. In order not to destroy the depth values already in the frame buffer, the P-buffer can be used for shadow map generation. In the texture application stage, each shadow map texel is modulated with the alpha value of the texel in the first texture unit. By enabling the alpha test, the occluded part on the object does not get textured. It should be mentioned that the tangent surfaces with respect to each camera position are culled in advance to avoid being colored by ill-sampled textures.

5.2.2 *Shadow leak*

The shadow leak problem is caused by insufficient sampling of the shadow map. But increasing the sampling rate requires a larger resolution of the depth image which decreases rendering performance considerably. Even for a depth image with 640x480 pixels, this artifact still cannot be completely eliminated. Our aim is to remove this artifact with a moderate shadow map size.

One can perform a morphological erosion operator on the color image to shrink the contour. Then the boundary pixels will not project onto the object so that the shadow leak is removed. But the erosion operation usually also removes thin features in the image, which is undesirable. The solution to the shadow leak problem is to fatten the object a little bit when generating the shadow map. Simply rendering an upscaled version of the object does not work because of the mismatches between the shadow map and projective texture coordinates. Ideally, we need to keep the inner part of the object unchanged and only thicken the contour of the object. This can be accomplished by a two-pass shadow map rendering technique.

In the first pass, we choose a thick line width and render the visual hull in line mode. In the second pass, we switch to the normal fill mode to render the visual hull. Thus, the contour of the depth map will be fattened a little bit and the interior regions will correctly keep their original depth. A subtle problem here is to use the polygon offset to remove self-shadowing. We need to use a larger polygon offset in the line mode so that the thickened line will not occlude the region adjacent to the edge.

(a) (b) (c)

Figure 5: Removing shadow leak artifacts. To better illustrate the problem, the visual hull is only rendered using a single image. (a) Without considering visibility. Notice the showing-through of the hands onto the back surfaces. (b) Using opaque texture mapping, the occluded parts do not get textured. However, along the contours, there are conspicuous shadow leak artifacts. (c) Shadow leak artifacts removed by 2-pass shadow map rendering. The resolution of the shadow map is only 320x240 pixels, the same as the original color image.

5.2.3 *Precision of the shadow map*

Because of perspective projection, the precision of the shadow map is not uniform. A tight range between near and far plane can efficiently improve the precision. Since the 3D intersection algorithm that we use also computes bounding box information for the visual hull, we can transform world coordinates of this bounding box to the camera coordinate system. The minimum and maximum z values of this bounding box can be computed and used as the near and the far plane. This computation is performed for each camera. Therefore, the precision of the shadow map is kept optimal for all cameras. The cost of computation is trivial since only the 8 vertices of the box need to be considered.

5.3 Dynamic texture packing

Normally, in order to project multiple images onto the 3D object, we need to download the images to the graphics card one by one. But texture switching is an expensive operation, especially when we have a large number of source images. Texture packing combines multiple images from different viewpoints into one big image and sends it to the graphics card in one. This reduces the number of texture switching and improves rendering performance.

Texture packing is also beneficial for reducing the amount of texture data. In the case of our visual hull system, only the moving foreground object is needed for projective texturing mapping. Therefore, we can compute the bounding box of the foreground object on the client machine and transfer it to the server. Then this *region-of-interest* (ROI) information can be exploited to pack multi-view images and reduce the packed texture size significantly. One might argue that hardware-accelerated compressed textures can reduce the amount of texture data as well. But compression and decompression take extra time. Usually this penalty is even bigger than using uncompressed data.

Our basic packing principle is very simple: We just take the ROIs from the source images and concatenate them into the packed image along the horizontal direction. The offsets of the ROIs between the original and the packed images are recorded and used later for computing correct texture coordinates.

There are two problems when using this simple scheme. First, the copying operation from the source images to the packed image takes some time. This time can be spared by directly updating the subregion of the texture, which proves to be a more efficient way to manipulate texture images. In addition, since the object is moving, the silhouette generated from each view is different from frame to frame. Therefore, the size of the packed texture also varies. If we change the texture size for every frame, the relocation of texture memory slows down rendering performance. To prevent frequent change in texture size, we can take advantage of the fact that the packed texture size does not change much between several consecutive frames, owing to temporal coherence. We reserve some additional space both for the width and height of the packed texture. This way, some size change can be tolerated. We dynamically expand or shrink the packed texture size only when the change is beyond a predefined threshold (20 pixels in our system). Fig. 6 shows an example of the packed texture.

Figure 6: Dynamic texture packing. 4 textures are packed together. The top and right sides have some reserved empty space. The size of the original texture images are 320x240, while the packed texture size is only 315x211: It is even smaller than a single original texture.

Our dynamic texture packing technique can be applied to the background images and shadow maps as well, which gives us further rendering speed improvement.

6 Results

In our current system configuration, the server is a P4 1.7GHz dual-processor machine equipped with a GeForce 3 graphics card. The clients are Athlon 1.1GHz computers. The video images are acquired at 320x240 pixel resolution. Using 6 silhouettes, we achieve reconstruction rates of 15 fps, which is also the highest possible speed of our synchronized video acquisition system.

We use multiple threads to receive data via the network and to decouple reconstruction from rendering. The rendering thread can run faster than that for the reconstruction. This provides the user with a good sense of interactivity even when the reconstruction is slow.

The rendering frame rate is about 23 fps for a video sequence acquired from six views. The number of triangles of the dynamic object ranges from 400 to 500.

For dynamic texture packing, we carried out a test by using different numbers of source images. The performance improvement can be seen in Table 1.

Number of Source Images	Performance		
	Unpacked Texture(fps)	Packed Texture(fps)	Increase(%)
2	111	134	20.7
3	85	105	23.5
4	72	90	25.0
5	54	70	29.6
6	46	63	36.9

Table 1: Performance improvement of texture packing. The frame rates are measured without background and shadow map rendering.

Thanks to the polyhedral representation of the 3D model, our system can easily visualize the dynamic object in different ways, as seen in Fig. 7(a)-(c). Fig. 7(d) shows that multiple dynamic objects can also be reconstructed and rendered by our system. Video demos are available at the web site `http://www.mpi-sb.mpg.de/~ming/DynaVisualHull.html`.

7 CONCLUSIONS and FUTURE WORK

A distributed real-time 3D reconstruction and rendering system is presented in this paper. The visual hulls of the dynamic objects can be reconstructed on line from live videos. We have introduced a novel blending algorithm to hide boundary seams for rendering the surrounding environment. Opaque projective texture mapping is extended to handle shadow leak artifacts. We have proposed a dynamic texture packing technique to reduce texture download overhead. The visual quality of both the dynamic objects and the background is improved. Meanwhile, our rendering algorithm is fully hardware-accelerated and achieves real time rendering performance.

There are still plenty of areas that need to be explored in real time reconstruction and rendering from real scenes. One line of future research is geometry refinement. Visual hulls usually have a lot of rough-edged surfaces which severely lower the visual quality. This problem might be dealt with surface fairing techniques [Tau95]. For background rendering, even our blending method shows visible seams when the images differ two much. A better blending scheme can be implemented using the feathering technique [Sze97]. Hardware-accelerated implementations of this scheme need more texture units and the advanced OpenGL extension FRAGMENT_PROGRAM, which will be supported soon by top graphics vendors. Of course, the same blending scheme can be applied not only to the background but also to the dynamic foreground objects.

So far, we render the dynamic object using images from all reference viewpoints. Actually, depending on the destination viewpoint, we are able to select a subset of source images and only transfer these to the server. This way, an approximately constant rendering frame rate can be maintained even when a large number of cameras are deployed. Recovering the lighting and surface properties of real scenes can be helpful for relighting. Real time recovery still remains a challenging problem. It is also an important aspect toward building a fully-automated, dynamic reconstruction and rendering system.

8 REFERENCES

[Bal91] Balakrishnan, V. K. *Introductory discrete mathematics*, chapter 3. Recurrence Relations. Prentice Hall, 1991.

[Bek] Bekaert, P. Boundary representation library. http://breplibrary.sourceforge.net/.

[Bic94] Bichsel, M. Segmenting simply connected moving-objects in a static scene. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(11):pp. 1138–1142, 1994.

[Bue01] Buehler, C., Bosse, M., McMillan, L., Gortler, S. J., and Cohen, M. F. Unstructured lumigraph rendering. In *SIGGRAPH'01 Proceedings*, pp. 425–432. 2001.

[Cur96] Curless, B. and Levoy, M. A volumetric method for building complex models from range images. In *SIGGRAPH'96 Proceedings*, pp. 303–312. 1996.

[Deb96] Debevec, P. E., Taylor, C. J., and Malik, J. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *SIGGRAPH'92 Proceedings*, pp. 11–20. 1996.

[Deb98] Debevec, P. E., Borshukov, G., and Yu, Y. Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Rendering Workshop*, pp. 105–116. 1998.

[Hei99] Heidrich, W. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. Ph.D. thesis, University of Erlangen, Computer Graphics Group, 1999.

[Lau94] Laurentini, A. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Machine Intell.*, 16(2):pp. 150–162, 1994.

(a) (b) (c) (d)

Figure 7: More rendering results. The object in (a)-(c) is reconstructed from 6 views. (a) Wireframe visual hull. (b) Flat shaded visual hull. Different color corresponds different source camera. (c) Textured visual hull with visibility handling. (d) Two textured visual hulls reconstructed from 4 views.

[Lok01] Lok, B. Online model reconstruction for interactive virtual environments. In *Proceedings 2001 Symposium on Interactive 3D Graphics*, pp. 69 – 72. 2001.

[Mat00] Matusik, W., Buehler, C., Raskar, R., Gortler, S. J., and McMillan, L. Image-based visual hulls. In *SIGGRAPH'00 Proceedings*, pp. 369–374. 2000.

[Mat01] Matusik, W., Bueler, C., and McMillan, L. Polyhedral visual hulls for real-time rendering. In *Proceedings of 12th Eurographics Workshop on Rendering*, pp. 115–125. 2001.

[Moe96] Moezzi, S., Katkere, A., Kuramura, D. Y., and Jain, R. Reality modeling and visualization from multiple video sequences. *IEEE Computer Graphics and Applications*, 16(6):pp. 58–63, 1996.

[Nar98] Narayanan, P., Rander, P., and Kanade, T. Constructing virtual worlds using dense stereo. In *Proc. of the Sixth ICCV*, pp. 3–10. 1998.

[Pee00] Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J. Interactive multi-pass programmable shading. In *SIGGRAPH'00 Proceedings*, Computer Graphics Proceedings, Annual Conference Series, pp. 425–432. 2000.

[Saw02] Sawhney, H., Arpa, A., Kumar, R., Samarasekera, S., Aggarwal, M., Hsu, S., Nister, D., and K.Hanna. Video flashlights – real time rendering of multiple videos for immersive model visualization. In Debevec, P. and Gibson, S., (eds.), *Proceedings of 13th Eurographics Workshop on Rendering*, pp. 163–174. Springer Wien, New York, NY, 2002.

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

[Seg92] Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., and Haeberli, P. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH'92 Proceedings*, pp. 249–252. 1992.

[Sze93] Szeliski, R. Rapid octree construction from image sequences. *CVGIP: Image Understanding*, 58(1):pp. 23–32, 1993.

[Sze97] Szeliski, R. and Shum, H.-Y. Creating full view panoramic image mosaics and environment maps. In *SIGGRAPH'97 Proceedings*, pp. 251–258. 1997.

[Tau95] Taubin, G. A signal processing approach to fair surface design. In *SIGGRAPH'95 Proceedings*, pp. 351–358. 1995.

[Ved98] Vedula, S., Rander, P., Saito, H., and Kanade, T. Modeling, combining, and rendering dynamic real-world events from image sequences. In *Proc. 4th Conference on Virtual Systems and Multimedia (VSMM98)*, pp. 326–332. 1998.

[Wil78] Williams, L. Casting curved shadows on curved surfaces. In *SIGGRAPH'78 Proceedings*, pp. 270–274. 1978.