A Hybrid Hardware-Accelerated Algorithm for High Quality Rendering of Visual Hulls

Ming Li Marcus Magnor Hans-Peter Seidel

Computer Graphics Group
MPI Informatik

Abstract

In this paper, a novel hybrid algorithm is presented for the fast construction and high-quality rendering of visual hulls. We combine the strengths of two complementary hardware-accelerated approaches: direct *constructive solid geometry* (CSG) rendering and texture mapping-based visual cone trimming. The former approach completely eliminates the aliasing artifacts inherent in the latter, whereas the rapid speed of the latter approach compensates for the performance deficiency of the former. Additionally, a new view-dependent texture mapping method is proposed. This method makes efficient use of graphics hardware to perform *per-fragment* blending weight computation, which yields better rendering quality. Our rendering algorithm is integrated in a distributed system that is capable of acquiring synchronized video streams and rendering visual hulls in real time or at interactive frame rates from up to eight reference views.

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

1 Introduction

In the past few years, real-time reconstruction and rendering from multiple views has been attracting much attention because it can open the way to various attractive applications such as 3D TV, interactive computer games and sport analysis. A number of systems [18, 21, 17] employ *Visual Hulls* (VH) [16] for shape representation and demonstrate very promising results.

Theoretically, the exact visual hull is defined as the maximal shape that reproduces the silhouettes of a 3D object from any viewpoint [16]. According to this definition, visual hull reconstruction is a straightforward procedure. We first back-project silhouette images from all possible directions outside the convex hull of an object and generate a *visual cone* for each view. The intersection of these cones produces the visual hull of the object. In practice, we can only consider a limited number of silhouette images from which an approximate visual hull is reconstructed.

Traditional methods reconstruct visual hulls explicitly on CPUs and render them separately [22, 21]. Recently, progress in graphics hardware has made it feasible to accomplish both reconstruction and rendering on the GPU at interactive frame rates or even in real time. In such cases, visual hull reconstruction is performed implicitly, which is sufficient as long as our primary goal is to render the object from novel viewpoints.

Among the hardware-accelerated algorithms, the one proposed by Li et al. [17] exploits texture mapping hardware to perform visual hull reconstruction and rendering in a single pass and achieves high rendering speed. However, the use of projective texture mapping for reconstruction causes aliasing artifacts between visual hull faces generated from different visual cones. These artifacts become especially noticeable when the visual hull is examined in a close view. This drawback is inherent and cannot be completely overcome even if high-resolution input images are used.

Another class of visual hull reconstruction algorithms utilizing graphics hardware are called direct CSG rendering methods, which consider front and back faces in rasterized image space and determines the intersection of a set of 3D objects. The basic principle was presented by Goldfeather et al. [12] and later implemented on commodity graphics hardware by Wiegand [29]. This method is able to render visual hulls without the aforementioned aliasing artifacts. However, even accelerated by hardware, the performance of this algorithm is still relatively slow due to high fill-rate consumption. As a result, it has not yet been applied in the context of real-time novel view synthesis.

Our hybrid approach takes advantage of both methods characterized above. It attains high quality from direct CSG rendering while keeping the fast speed with the help of the texture mapping-based approach. Additionally, we propose a view-dependent texturing scheme which further improves the rendering quality by employing more accurate *per-fragment* blending weight computation. Our rendering algorithm is integrated in a distributed system with a client-server architecture. Client computers are responsible for acquisition of synchronized live video streams, while the server performs on-line rendering of visual hulls of real dynamic objects in high fidelity at interactive to real-time frame rates.

The remainder of this paper is organized as follows. Section 2 reviews some previous work. Section 3 briefly describes the principles of the two basic visual hull rendering algorithms. Section 4 explains our hybrid rendering algorithm in details. Section 5 presents our system implementation and rendering performance. The paper concludes with some discussions of future research directions.

2 Previous Work

2.1 Visual hull reconstruction

Visual hull reconstruction methods can be classified into two categories: voxel-based and polyhedron-based approaches. The first approach discretizes a confined 3D space into voxels and carves away those voxels whose projections fall outside the silhouette of any reference view [22]. This carving process is often accelerated by octree data structures [27]. Hardware-accelerated algorithms presented in [14, 18] render visual hulls by projectively texturing a stack of planes. Related to visual hull reconstruction, *Space Carving* [15] checks color consistency across multiple views and achieves more accurate reconstruction results. However, it is too time-consuming for real-time applications. A common drawback of all voxel approaches is the quantization problem coming from space tessellation.

Polyhedron-based approaches represent each visual cone as a polyhedral object. Although visual hulls can be reconstructed by general 3D intersection of polyhedral visual cones [3], the intersection computation between complex 3D objects is very sensitive to numerical instabilities. Matusik et al. [21] make the intersection more robust and efficient by reducing it from 3D to 2D. Sullivan et al. [26] fit polyhedral visual hulls with triangular spline surfaces and produce smoother models. The *image-based visual hull* technique [20] directly computes a view-dependent version of a visual hull for a desired view. Intersections between lines and polyhedral visual cones are involved implicitly. Texture mapping-based visual cone trimming [17] and direct CSG rendering [29, 25, 13], the two hardware-accelerated algorithms that form the basis of ours, are also polyhedron-based approaches. Both compute intersections of polyhedral visual cones in rasterized image spaces and thus avoid the numerical instability problem completely.

2.2 Multi-view texture mapping

In the field of image-based modeling and rendering, original images taken from multiple viewpoints are often

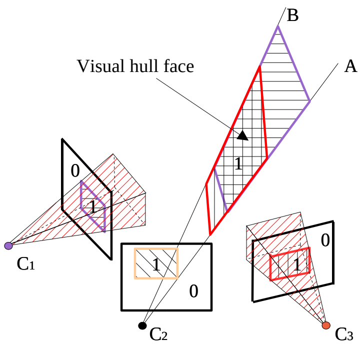

Figure 1: Principle of the texture mapping-based visual cone trimming. ABC_2, one of the visual cone faces extruded from view C_2, is being rendered. By multiplying the alpha values projectively textured by view C_1 and C_3, only the common part (cross-hatched) has the alpha value 1.

mapped onto recovered 3D geometry in order to achieve realistic rendering results [10]. For those parts visible in several views, it is necessary to blend between these views to obtain smooth transitions. Meanwhile, view-dependent effects are achieved by view-dependent texture mapping [9, 5]. This technique is also exploited to render textured visual hulls. Unlike with previous methods, we compute blending weights for each rasterized fragment instead of at each vertex. Our method is not only fast due to the use of graphics hardware, but also more accurate and hence yields improved rendering quality.

3 Principles of Basic VH Rendering Algorithms

3.1 Texture mapping-based visual cone trimming

The key idea of Li et al.'s method [17] is to trim visual cones with alpha maps (shown in Figure 1). First, alpha channels of silhouette images are set to 1 for foreground objects and to 0 for background scenes. Given a novel viewpoint, each visual cone is rendered by projecting the images from all views except the one associated with the cone currently being drawn. In the texture units, alpha values from multiple textures are modulated. As a result, only those fragments projected with the alpha value 1 from all the other views produce the output alpha value 1. The other fragments are discarded by enabling the alpha test.

Ideally, we would have silhouette edges that can be represented analytically, so that clear-cut intersections occur on visual cone faces. However, silhouette edges composed by discrete texels are mapped onto 3D faces. This discretization causes jagged aliasing along intersection boundaries. A strong point of this method is its speed. Since each visual cone is only rendered once, the time complexity is $O(n)$, where n is the number of visual cones.

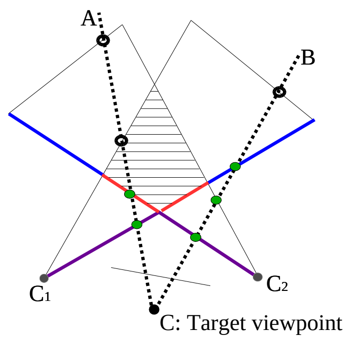

Figure 2: Principle of the direct CSG rendering shown in 2D. Two visual cones extruded from C_1 and C_2 are intersected. The intersection region is filled with horizontal lines. The first depth layer is drawn in purple and the second layer is colored in red and blue. The red parts are the intersection result rendered from the target viewpoint. The hollow circles are fragments behind the second layer. The green circles are fragments in front of or on the second layer.

3.2 Direct CSG Rendering

Direct CSG rendering methods [29, 25, 13] exploit the stencil buffer to perform 3D Boolean operations and do not have the aliasing problem mentioned above. The basic idea is counting front and back faces with the help of the stencil buffer. Given a target view, we traverse through all depth layers of front faces using *depth peeling*. For each layer, we set the depth test to "less or equal" and rasterize front faces of all objects. A stencil value is increased by 1 if the associated fragment passes the depth test. When back faces are rendered, we decrease the stencil value if a fragment survives in the depth test. After all depth layers are traversed, only those fragments in the intersection parts are marked with the stencil values which are equal to the total number of the objects. In the end, a complete depth map of the visual hull for the current viewpoint is generated. As an example, the intersection of two visual cones is illustrated in 2D using Figure 2. For the second depth layer, two front-facing fragments pass the depth test along the viewing ray CA, and hence yield a stencil value of 2 (equal to the number of the visual cones). For the viewing ray CB, two front-facing fragments and one back-facing fragment pass the depth test and produce a stencil value of 1.

Most direct CSG rendering methods [29, 25] need depth buffer readback, which is their main performance bottleneck. Very recently, Guha et al. [13] utilize the *depth peeling* technique [11] to remove this requirement to gain higher performance. However, the number of rendering passes needed by this algorithm is the same as the number of total depth layers. In visual hull reconstruction, all visual cones are intersected with each other. Consequently, the minimum number of depth layers is equal to the number of visual cones (denoted as *n*). This means that the time complexity of the CSG rendering method is $O(n^2)$ in contrast to $O(n)$ for the texture mapping-based method.

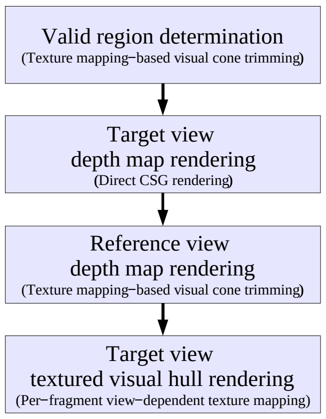
Valid region determination
(Texture mapping−based visual cone trimming)
Target view depth map rendering
(Direct CSG rendering)
Reference view depth map rendering
(Texture mapping−based visual cone trimming)
Target view textured visual hull rendering
(Per−fragment view−dependent texture mapping)

Figure 3: Work flow of our hybrid visual hull rendering algorithm. The texts in parentheses indicate the technique utilized in each rendering step. They are described in detail from Section 4.1 to Section 4.4.

4 Hybrid Visual Hull Rendering

In order to increase the quality of the texture mapping-based method and to accelerate the performance of the direct CSG rendering method, we propose a hybrid approach combining the strengths of both. The whole rendering process consists of four steps (Figure 3) which are described in the rest of this section.

4.1 Valid region determination

When applying the direct CSG rendering to reconstruct visual hulls, polygons of visual cones have to be rasterized multiple times. These polygons are usually very large and consume a large amount of fill-rate capability of graphics hardware. In practice, intersected results, i.e. visual hulls, only occupy part of the output window. We call this partial region a *valid region*. If we know this region *a priori*, the fill rate can be greatly reduced by applying the scissor test. During the CSG rendering, another operation also benefits from predicting the valid region. The *depth peeling* technique needs frame-buffer-to-texture copying or rendering-to-texture operations when sweeping through all depth layers. With the knowledge of the valid region, the copying or rendering operations can be carried out on this region only instead of the whole window area.

For the determination of the valid region, we employ the texture mapping-based method to render the visual hull in a very small off-screen window (e.g. 80 x 60 pixels). Writing to the color buffer and to the depth buffer are both disabled. The stencil buffer is only written where the visual hull covers. The stencil buffer is read back to the main memory, and a rectangular region enclosing the visual hull is calculated. This region is expanded by one pixel in order to tolerate some rounding errors introduced by rendering a smaller version of the visual hull. In case the visual hull contains some thin features that might be missed during the down-sampling, we should use a larger off-screen window or expand the region by more pixels. As a last step, the rectangular region is scaled by the size ratio between the final target viewport and the small off-screen window.

Notice that the aliasing artifacts of the texture mapping-based visual hull rendering method do not have any effect on the valid region determination since only a region is needed here. As to the performance issue, although buffer readback operations are generally expensive, reading the stencil buffer of a small window does not cost much.

4.2 Target view depth map rendering

Once the valid region has been identified, visual cones can be rasterized in the constrained area by applying the CSG rendering method proposed by Guha et al. [13]. The result is a depth map of visual hulls in the target view. This depth map guarantees that the final visual hull rendering is free of jagged artifacts. Although this rendering step only produces depth maps, we include a final rendering quality comparison (shown in Figure 4) to demonstrate that the CSG method is able to generate significantly better rendering results compared to the texture mapping-based method.

The original CSG method used by Guha et al. performs well for visual hull reconstruction as long as target viewpoints are outside of any visual cone. Once a target viewpoint falls inside a visual cone, parts of the visual cone's faces will be clipped by the near plane of the view volume. This leads to wrong face counting and incorrect rendering results. Such limitation is undesirable in the context of novel view synthesis of visual hulls.

To solve this problem, we come up with an idea inspired by Carmack's *zfail* shadow volume algorithm [6] and count front faces and back faces between *infinity* and the first visible rasterized fragment (The hollow circles in Figure 2 give examples). Accordingly, the updating strategy for the stencil buffer is changed. A stencil value is decreased for front faces or increased for back faces when the depth test *fails*. In this way, we produce correct counting results without worrying about the near clipping plane. Apart from this modification, we provide some other enhancements to the original CSG rendering method, such as terminating depth traversal using hardware-accelerated occlusion tests [7] and reducing the number of rendering passes with two-sided stencil operations [8].

4.3 Reference view depth map rendering

Visibilities with respect to reference views are critical to multi-view texture mapping. For those parts that are invisible in a reference view, the corresponding color information should be ignored when blending multiple textures. Debevec et al. address this issue in object space [9]. Triangles of objects are split so that they are either fully visible or fully invisible to any source view. This process takes a long time even for a moderately complex object and is not suitable for real-time applications.

We handle the visibility problem in image space by employing the hardware-accelerated shadow mapping algorithm [24], which needs a depth map for each reference view. Depth maps of the reference views can be generated from the visual hull by either method discussed in Section 3. We choose the texture mapping-based method because of its considerably higher speed. One concern is that depth maps produced using this method have aliasing artifacts and the quality is not as good as that produced by the CSG method. In practice, however, due to smooth blending of multiple textures, the artifacts introduced by the quality difference of depth maps are hard to notice in the final rendering result.

4.4 Textured visual hull rendering

Since the depth map of a visual hull has already been created for the target view as described in Section 4.2, we render the textured visual hulls by enabling the "equal" depth test and rasterizing all visual cones using the multi-view texture mapping. Note that when a visual cone is textured, the reference view corresponding to this cone cannot be used because it would project texels along silhouette contours throughout the whole surfaces of the cone. In the following, we will discuss multi-view texture mapping and explain how to achieve better rendering quality using per-fragment blending weight computations.

View-dependent multi-view texture mapping

In order to obtain smooth transitions across different views, we represent a fragment's color C as a convex combination of the texture values in the reference views:

$$C = \frac{\sum_{k=1}^{N} W_k * T_k}{\sum_{k=1}^{N} W_k}\,, \tag{1}$$

where N is the number of reference views, T_k and W_k are the texture color and its associated weighting factor for the k-th input image, respectively. The weight W_k consists of four components and is defined as follows:

$$W_k = V_k * W_{kf} * W_{ks} * W_{kd}\,. \tag{2}$$

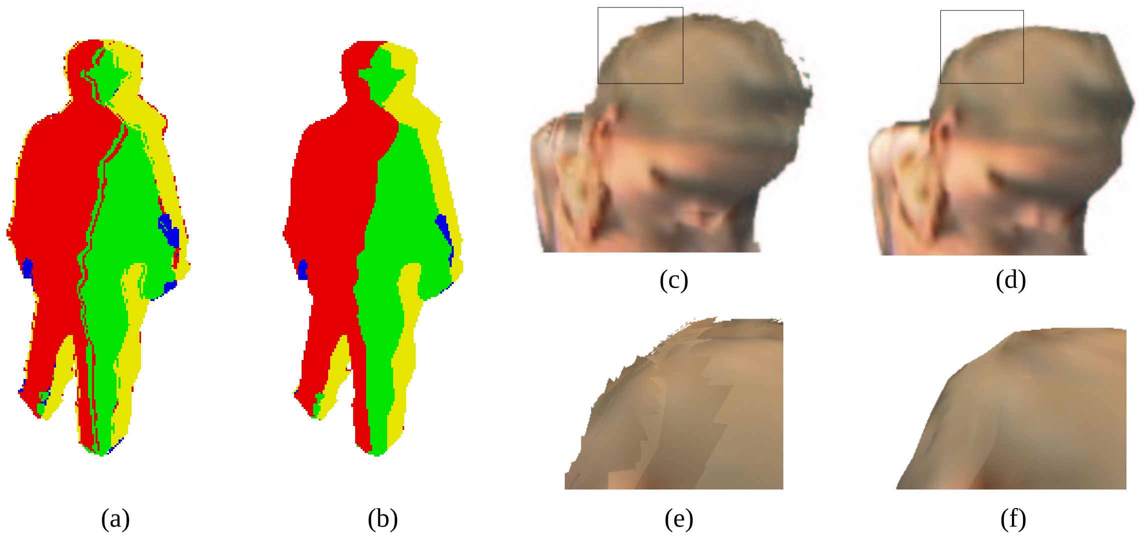

Figure 4: Quality comparison between visual hulls rendered by textured-mapping-based visual cone trimming and direct CSG rendering. (a)(c)(e) employ the former method, whereas (b)(d)(f) apply the latter. (a) and (b) are rendered in flat-shaded style. One can clearly observe the inter-penetrations along intersection boundaries. (c)(d)(e)(f) show textured visual hulls. (e) and (f) are close views of the boxed regions in (c) and (d), respectively.

The first component V_k is the visibility function with respect to the reference view k. Since the depth map corresponding to the view k has been created as explained in Section 4.3, V_k can be easily determined by the standard shadow mapping algorithm [24].

W_{kf} is a feathering weight. It serves for eliminating seams that arise from sampling different reference views at silhouette boundaries [10, 28, 23]. The weight values ramp down from 1 in the center area to 0 at the silhouette outlines. We apply a distance transformation [4] to the binary silhouette masks to obtain the feathering weight maps. These maps are computed in real time on client computers. Then, they are embedded in the alpha channels of the silhouette images and transferred to the rendering server.

W_{ks} is referred to as the surface obliqueness weight. It penalizes surfaces that are oblique when observed from a reference viewpoint. We use the following definition:

$$W_{ks} = \max(\bar{d}_k \bullet n\,,\, 0)^{\alpha}\,. \tag{3}$$

For a point p, which is associated with the fragment under consideration, the vector $\bar{d}_k$ is the viewing direction from p toward the k-th reference camera. The vector n denotes the normal of the point p. The constant α is a tunable parameter that makes relative weight differences larger.

The last weighting component W_{kd} is dependent on the target viewpoint. It gives a higher weight to the fragment whose reference viewing direction $\bar{d}_k$ is closer to its target viewing direction $\bar{d}_t$. The weight function is expressed as:

$$W_{kd} = (\bar{d}_k \bullet \bar{d}_t + 1)^{\beta}\,. \tag{4}$$

The constant β plays a similar role as the constant α in Equation 3. The addition of the number 1 guarantees that W_{kd} is non-negative, which is necessary to obtain a convex combination for the final pixel color. Some researchers [9, 5] represent this view-dependent weight as a function of the angle between $\bar{d}_k$ and $\bar{d}_t$. However, computing the angle requires an inverse cosine operation. The accurate evaluation is very costly on current graphics hardware, whereas approximation by a table lookup consumes an additional texture image unit.

Per-fragment blending weight computation

In order to compute the weighting factors W_{ks} and W_{kd} accurately, the 3D coordinate associated with each fragment must be used to determine the viewing vectors $\bar{d}_k$ and $\bar{d}_t$. Most previous view-dependent texture mapping methods [23, 9, 5] only employ vertex coordinates for weight computation and then perform a linear interpolation across surfaces to obtain the weights for each fragment. However, in the case of hardware-accelerated visual hull rendering, we do not have vertex information of visual hulls. The only available geometry entities are visual cones. In this case, the weight computation based on a per-vertex basis has large errors and causes noticeable rendering artifacts.

Fortunately, the vertex and fragment programmability [2, 1] of recent graphics hardware can assist us in calculating these weights on a per-fragment basis to achieve high rendering quality. Our per-fragment view-dependent texture mapping procedure proceeds as follows. First, 3D coordinates of visual cone vertices and normal vectors of visual cone faces are passed to a vertex program [2],

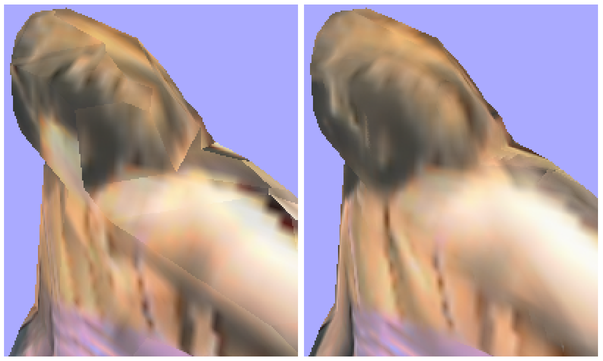

Figure 5: Comparison of rendering quality between per-vertex and per-fragment blending weight computations. (a) and (b) are rendered from the same viewpoint. ***(a)*** *Per-vertex weight computation.* ***(b)*** *Per-fragment weight computation. Rendering artifacts diminish considerably because of more accurate weight computation.*

through which a 3D coordinate together with a normal vector is generated for each fragment by the graphics rasterizer. Subsequently, our fragment program [1] takes these inputs to compute the per-fragment weights W_{ks} and W_{kd}. Both constants α and β are empirically set to a value of 5 in our implementation. The viewing vector normalization in Equations 3 and 4 is replaced by a normalization cube map [30] lookup for acceleration purposes. In the same fragment program, V_k is computed from the shadow maps, while W_{kf} and T_k are obtained from the alpha and color channels of the projective silhouette textures. Finally, the fragment color is evaluated according to Equation 1. We compare the rendering results using per-vertex and per-fragment weight computations in Figure 5.

Note that this view-dependent texture mapping algorithm is not restricted to visual hull rendering. Given an explicit 3D model and multiple reference views, it also achieves better rendering quality due to the more accurate weighting function computation.

5 System Implementation and Performance

The whole rendering algorithm has been integrated into a distributed system. Eight pre-calibrated Sony DFW500 FireWire cameras are deployed in a convergent fashion. They acquire video color images at 320x240-pixel resolution. Four camera-controlling client computers connect to a rendering server via a standard TCP/IP network. Each has an Athlon 1.1GHz CPU and acquires synchronized video at 15 fps. In addition, they are responsible for real-time image processing tasks, such as foreground/background segmentation, silhouette contour extraction and distance transformation. The server is a P4 1.7GHz processor machine and carries out the hybrid rendering algorithm on a GeForce FX 5800 Ultra graphics card.

# Reference views	Rendering performance (fps)
3	27.0
4	13.2
5	8.8
6	6.4
7	4.7
8	4.0

Table 1: Performance of our hybrid visual hull rendering algorithm.

The distributed system runs on Linux. The high-level shading language Cg [19] is employed to implement the per-fragment view-dependent blending algorithm presented in Section 4.4. Our Cg code can be compiled into low-level shaders suitable for different graphics hardware and different graphics APIs (e.g. OpenGL, Direct3D).

	Timings (ms)	
	w/ valid region determination	w/o valid region determination
Valid region determination	5	N/A
Target view depth map rendering	40	115
Reference view depth map rendering	51	51
View-dependent texture mapping	96	300
Texture loading and miscellaneous	58	58
Total	250	524
Overall performance	4.0 fps	1.9 fps

Table 2: Comparison of the rendering performance with and without valid region determination. Eight reference views are used. Timings of individual rendering steps are presented.

We set the resolution of the rendered novel view to 640x480 pixels and execute the hybrid rendering algorithm using various numbers of reference views from one set of multi-view images. The average number of polygons of each visual cone is about 50. The valid region occupies about one third of the entire viewport. Under this setting, we have measured the rendering performance for different numbers of reference views shown in Table 1. For eight reference views, we also render the visual hull with pure visual cone trimming and CSG methods. The frame rates are 7.5 fps and 2.9 fps, respectively.

Furthermore, we compare the rendering performance with and without using the valid region determination. Table 2 presents the comparison result as well as the timings of individual rendering steps. Notice that the rendering speed is approximately doubled when employing the valid region determination because both CSG rendering and final view-dependent texture mapping profit from the scissor test. For depth map rendering in the reference views, using the visual cone trimming method, it takes 51 ms to generate eight depth maps at 320x240-pixel resolution. If the CSG method is employed, it takes about 130 ms according to our measurement. Obviously, the former method performs better. Another characteristic of our rendering algorithm is that more graphics power is consumed at the fragment stage. This suggests the geometric complexity of visual cones does not degrade overall performance very much. In our tests, if the number of polygons of visual cones is increased by a factor of two, rendering speed slows down by about 10%. More snapshots of the visual hull rendering results from both real video streams and synthetic animation sequences are presented in Figure 6.

One limitation of our algorithm is that the number of input silhouette images is restricted to the maximum of texture coordinate sets available on the graphics hardware (eight in our case). However, we expect it will not take long to see the debut of powerful graphics hardware with more texture resources. As a tentative measure, this limitation can be lifted by adding more rendering passes and compositing intermediate results in the frame buffer.

6 Conclusions and Future Work

In this paper we have presented a hybrid hardware-accelerated algorithm to render high-quality visual hulls from multi-view video sequences. The performance ranges from interactive frame rates to real time depending on the number of reference views. Compared to analytical intersection methods, our algorithm works robustly also for highly complex objects since all intersections are performed in image space. The two algorithms that form the basis of our approach, texture mapping-based visual cone trimming and direct CSG rendering, benefit from each other. The inherent aliasing problem limiting the former method is elegantly solved, and the latter method's slow performance is accelerated. Additionally, rendering quality of textured visual hulls has been improved thanks to per-fragment view-dependent texture mapping.

There are several directions into which our work can be extended. Reconstructed visual hulls often exhibit some regions which are invisible from any reference view. These regions are currently drawn in black. It is desirable to interpolate color information from the local neighborhood to fill in these holes.

Since visual hulls are only approximations to actual 3D shape, the geometric inaccuracies often introduce texture distortions. We are exploring new algorithms that are able to incorporate color consistency checks to overcome this limitation.

Our current system setup supports up to eight source video streams. Up to now, network transmission has not been a bottleneck. If more reference views are available, however, suitable compression and streaming techniques must be considered.

References

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

[2] OpenGL ARB. ARB_vertex_program OpenGL extension. http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt.

[3] B. G. Baumgart. *Geometric modeling for computer vision*. PhD thesis, Stanford University, October 1974.

[4] G Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34(3):344–371, 1986.

[5] C. Buehler, M. Bosse, L. McMillan, S. J. Gortler, and M. F. Cohen. Unstructured lumigraph rendering. In *SIGGRAPH 2001*, pages 425–432, August 2001.

[6] J. Carmack. *zfail* stenciled shadow volume rendering. Unpublished correspondence. http://developer.nvidia.com/attach/5628. Early 2000.

[7] HP Corporation. HP_occclusion_test OpenGL extension. http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt.

[8] NVidia Corporation. EXT_stencil_two_side OpenGL extension. http://oss.sgi.com/projects/ogl-sample/registry/EXT/stencil_two_side.txt.

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

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

[11] C. Everitt. Interacive order-independent transparency. http://developer.nvidia.com/object/Interactive_Order_Transparency.html, 2002.

[12] J. Goldfeather, J. P. M. Hultquist, and H. Fuchs. Fast constructive-solid geometry display in the pixel-powers graphics system. In *SIGGRAPH 1986*, pages 107–116, 1986.

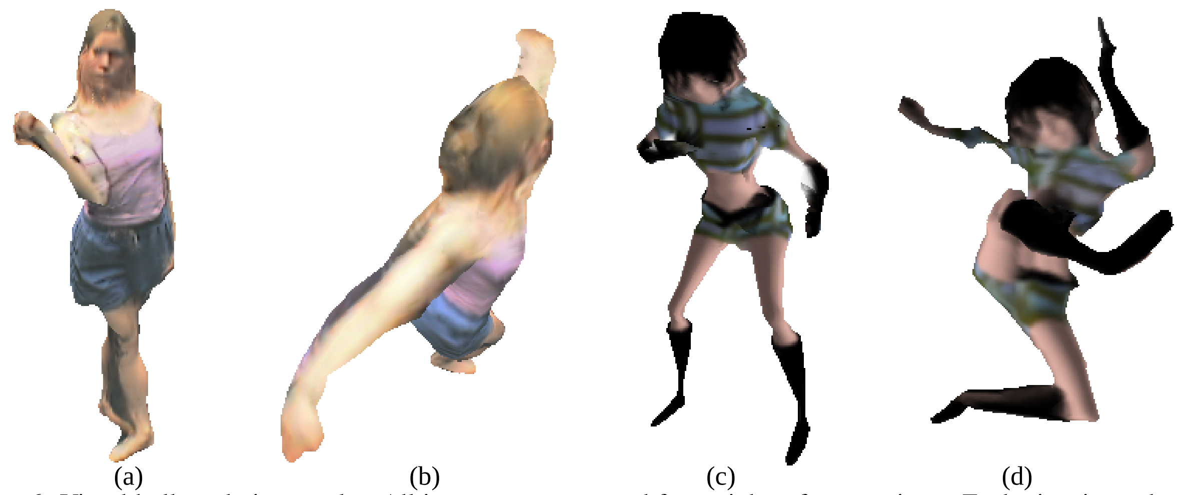

Figure 6: Visual hull rendering results. All images are generated from eight reference views. Each view is rendered at least 20° away from the closest reference view. Real video streams are used to render (a) and (b). Synthetic animation sequences are employed to create (c) and (d). Two demonstration videos are available on our website `http://www.mpi-sb.mpg.de/~ming`.

[13] S. Guha, S. Krishnan, K. Munagala, and S. Venkat. Application of the two-sided depth test to CSG rendering. In *2003 Symposium on Interactive 3D Rendering*, pages 177–180, 2003.

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

[15] K. Kutulakos and S. Seitz. A theory of shape by space carving. Technical Report 692, Computer Science Dept., U. Rochester, May 1998.

[16] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. PAMI*, 16(2):150–162, February 1994.

[17] M. Li, M. Magnor, and H.-P. Seidel. Hardware-accelerated visual hull reconstruction and rendering. In *Graphics Interface 2003*, pages 65–71, 2003.

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

[19] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH 2003*, pages 896–907, 2003.

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

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

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

[23] K. Pulli, M. Cohen, T. Duchamp, H. Hoppe, L. Shapiro, and W. Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. In *8th Eurographics Workshop on Rendering*, pages 23–34, 1997.

[24] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH 1992*, pages 249–252, July 1992.

[25] N. Stewart, G. Leach, and S. John. An improved Z-buffer CSG rendering algorithm. In *Graphics Hardware 1998*, pages 25–30, Aug 1998.

[26] S. Sullivan and J. Ponce. Automatic model construction and pose estimation from photographs using triangular splines. *IEEE Trans. PAMI*, 20(10):1091–1097, October 1998.

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

[28] R. Szeliski and H.-Y. Shum. Creating full view panoramic image mosaics and environment maps. In *SIGGRAPH 1997*, pages 251–258, August 1997.

[29] T. F. Wiegand. Interactive rendering of CSG models. *Computer Graphics Forum*, 15(4):249–261, 1996.

[30] C. Wynn and S. Dietrich. Cube maps. http://developer.nvidia.com/object/cube_maps.html.