13-12

MVA2007 IAPR Conference on Machine Vision Applications, May 16-18, 2007, Tokyo, JAPAN

A Fast Surface-Based Visual Hull Reconstruction

Sofiane Yous and Masatsugu Kidode
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara 630-0192, Japan
{yous-s, kidode}@is.naist.jp

Abstract

In this paper, we describe a new method for shape reconstruction from a set of silhouette images. We propose a method for traversing the depth layers of the projected viewing cones from all viewpoints based on the principle of the Constructive Solid Geometry (CSG). Taking advantage of the modern Graphics Processing Units (GPU), the proposed CSG-like method achieves a fast full reconstruction of VH, rather than rendering a looking of the VH from a virtual viewpoint, in an interactive frame rate. For each viewpoint, the viewing edges are computed separately in a parallel manner. Subsequently, the edges are merged together to generate the final surface-based Visual Hull (VH). The method was tested on several datasets including real data and the results will be presented in this paper.

1 Introduction

Acquiring the 3D shape of a real object is a key issue in a variety of modeling and 3D multimedia applications. The straightforward estimation of this shape can be obtained from the silhouette images of the object taken from different viewpoints. This concept was first introduced by Baumgart [4] and given the name of Visual Hull (VH) later by larentini [3]. Based on this concept, the VH is the maximal approximation of the object that reproduces the silhouettes of all viewpoints.

Estimating the 3D shape from silhouette images was motivated by the ease of obtaining silhouette images, especially in indoor environments with known camera parameters, and by the ease of implementation. Several methods have been proposed for VH reconstruction and/or rendering. Depending on the application, the VH hull is processed differently. For visualization applications, such as new viewpoint synthesis or interactive rendering, the exact reconstruction of the VH is not required. Image-based VH (IBVH) technique [5] suffices to generate the VH looking from a desired view. Hardware-based acceleration of IBVH was proposed through texture mapping-based visual cone trimming [9] or direct Constructive Solid Geometry (CSG) rendering [6,7,8] to speed up the processing.

Applications such as object modeling and 3D digital archiving require a full reconstruction of the object's shape. Voxel-based VH reconstruction tends to recover a volumetric representation of the object. Volume carving methods split up the 3D space into a 3D grid of voxels. These voxel are tested for their belonging to all silhouette cones and labeled as volume voxels if so. This class of methods, however, suffers from the extensive computation load and the memory overhead. This approach was first proposed by Martin and Aggarwal [12]. Later on, octree representation was proposed by Chien and Aggarwal [13] to substitute the voxel representation for less memory and computation demand. For the same purpose, Marching Intersections (MI) was proposed by Rocchini [2] as a re-sampling method for surface management and adapted later by Tarini et al. [1] for volumetric shape reconstruction.

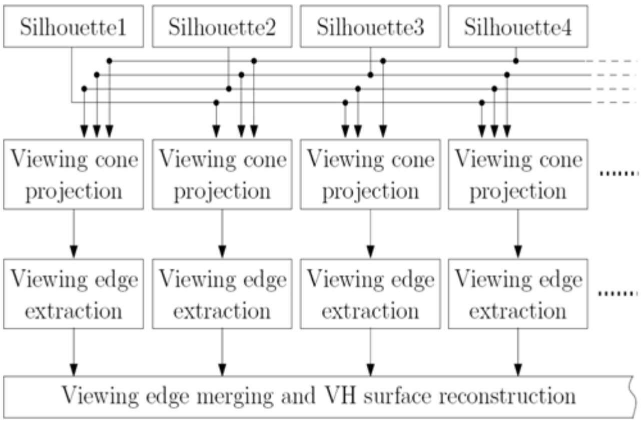
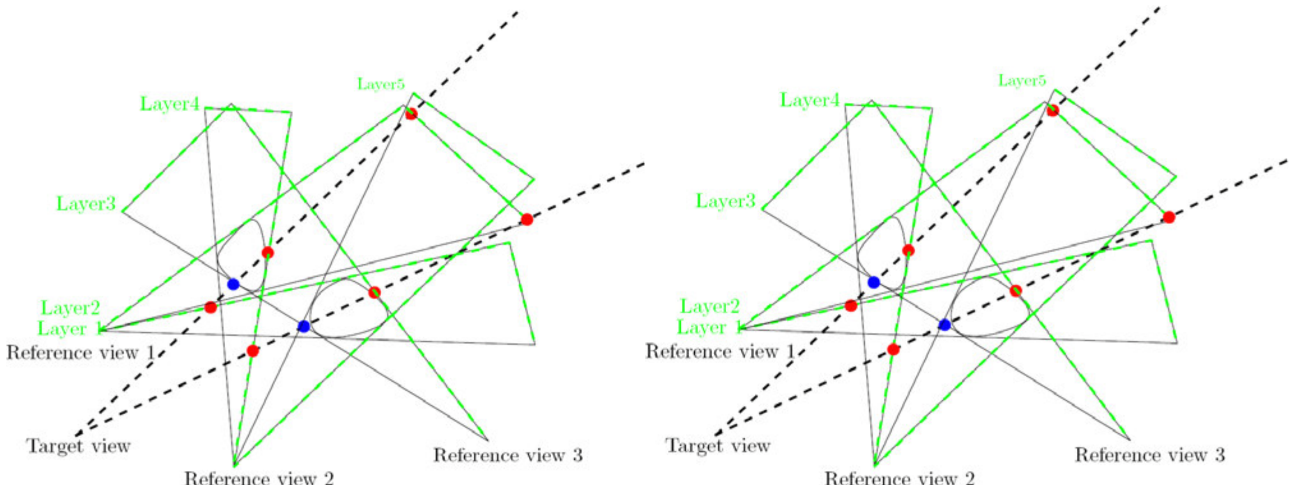


Figure 1. The VH reconstruction scheme

Surface-based methods also target an exact reconstruction of the VH, but as a 3D polyhedral surface. The surface vertices and faces are estimated by intersecting the generalized cones generated from the occluding contours of the silhouette images, rather than all the silhouette points. Baumgart [4] was the first to propose such a strategy, and later Koenderink [15], Cippola [16], and Boyer [17]. This class of methods produces visual artifact-free VH and requires much less computations and memory, comparing to the previous one. However, intersection in the 3D space is very sensitive to numerical instabilities, especially between complex objects. In this paper, we propose a new surface-based VH reconstruction from a set of silhouette images. The reconstruction is based on a modification of the direct CSG method [6,10] often used in image-based rendering. We propose a fast depth layer traversal method based on which, the viewing edges issued from the occluding contours are computed for each view in a parallel manner. For each viewpoint, the generalized cones from the other viewpoints are drawn in the 3D space. A multi-pass rendering using an off-screen is performed to traverse the depth layers of the drawn scene and extract only the line segments which lie to all cones. A modification is made to the direct CSG in order to allow a faster traversal of depth layers and identification and extraction of the viewing edges, rather than just rendering the depth of the shape from a desired view. Moreover, the viewing edges in our method are computed only where the local occluding points occur. This scheme is designed to run completely in the Graphics Processing Unit (GPU) and only one readback from the GPU memory is required at the end of the process to recover the viewing edges. We



(a) Direct CSG

(b) Our traversal method

Figure 2: Direct CSG Vs. our traversal method in 2D: The traversed depth layers are drawn in green dashed lines. The points shown in red are the tested points. The Blue points are the saved points and those bounded by a red circle are tested and saved. The number of tested points is 5 in the direct-CSG method and 3 in ours.

propose a storage method to allow the accumulation of the extracted edges at each pass in a shared buffer allocated on the GPU memory. Next, the viewing edges from all viewpoints are merged together to construct the final 3D shape after rectification of the 3D positions of their vertices. (See Figure 1)

2 Computation of the Viewing Edges

Given a set of N silhouette images associated to a set of N calibrated cameras C_n ($n = 1..N$) having c_n as center, the viewing edges for each view are the line segments parts of the rays associated with the occluding contour points of the silhouette and passing through the VH, and hence, lie to the silhouette images of all other views. Usually, the viewing edges are extracted by projecting each ray from each view to the silhouettes of all other views and find the line segments that intersect all silhouettes. This method is expensive in terms of processing time, especially when the number of occluding contour points is large. To speedup the computations, often one starts by approximating the occluding contours by polygons to reduce the number of points. However, this approximation yields a loss of details in the reconstructed VH.

In the method we propose, no approximation is applied. Instead of projecting the rays to all cameras, we employ a CSG-like method. For each viewpoint, the viewing cones from the other cameras are projected to the 3D space. Next, we traverse the depth layers of the drown scene and keep only those lying to all viewing cones. The intersections of these layers with the ray issued from the occluding contour points of the local camera define vertices in the VH surface. A viewing edge is a line segment which is part of a ray and defined by two vertices intersecting two adjacent opposite layers. The first point with respect to the camera center belongs to a front facing, while the other point belongs to a back facing layer. A front fragment is a region of a cone projecting to a pixel in the local image plane, where the dot product between its outward normal vector and vector joining the camera center and the fragment is negative. Otherwise, the fragment in question is a back fragment.

The CSG-based rendering was proposed by Goldfeather [18] and used later by Guha [6] and Li et al.[10] for GPU-based view-dependent VH rendering. CSG is based on the representation of a complex 3D object as a normalized tree of operations ($\cap, \cup, \setminus$) on primitive shapes. Let us consider the complex object expressed by the following expression:

$$(O_1 \cup O_2 \cap O_3) \setminus O_4 \tag{1}$$

This object can be represented by the following tree:

$$\underbrace{O_1 \cap \overline{O}_4}_{p_1} \cup \underbrace{O_2 \cap O_3 \cap O_4}_{p_2} \tag{2}$$

p_1 and p_2 are two products of the tree that can be processed in a parallel way and merged later on. If we refer by $f(d,p)$ and $b(d,p)$ to the number of, respectively, front and back faces with smaller depth than a point p and with respect to a desired viewpoint d, then p belongs to the product if:

$$f(d,p) - b(d,p) =| P | \tag{3}$$

where $| P |$ is the number of products in the tree.

2.1 Direct CSG-Based Rendering

A VH reconstructed from a set of viewpoints can be expressed by the intersections of all unions of cones, each of which is generated by the outer contours of one silhouette, and the complements of unions of cones, each of which is issued from the inner contours (holes) of one silhouette. The direct CSG-based rendering [6,10] can be

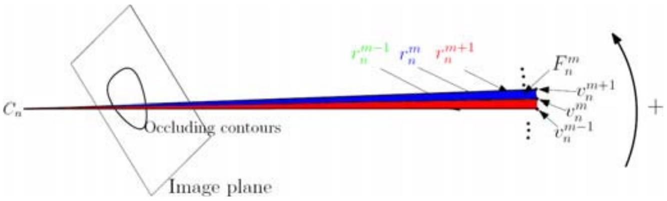


Figure 3: The silhouette generalized cone

summarized in the following steps:

- Repeat for all front depth layers:
  1. Project the next (first for the first iteration) depth layer of front faces.
  2. Count the front faces separating the traversed depth layer and the desired view position.
  3. Count the back faces separating the traversed depth layer and the desired view position.
  4. Save the depths of the points that verify (2).

2.2 Our CSG-Based Depth Traversal Method

We can see from figure 2(a) that a front face who is candidate to be a valid intersection is always immediately preceding a back face with respect to the camera center of the target view. This means that only the last of a succession of front faces can be a candidate, all remaining faces can be skipped. This can be done by rendering a back layer and then the farthest front layer with shorter depth than the rendered back layer. Also, only the first of a list of back faces is traversed, all the others can be skipped by rendering the first front face farther than the rendered back layer and then rendering the first back layer with longer depth than the rendered front layer. The new algorithm is as follows:

1. If this is the fist iteration, render the first depth layer of back faces. Otherwise, render the next depth layer of back faces having depth longer that the rendered front layer (skip all front faces).
2. Count the back faces separating the traversed depth layer and the desired view position.
3. Count the front faces separating the traversed depth layer and the desired view position and keep the depth of the last depth with respect to the camera.
4. Save the depths of the points that verifies (2).
5. Render the first front layer having a depth greater than the current back layer (skip all back layers separating the two layers).
6. Finish the process if no layer is returned, otherwise go to 1.

The advantage of this new algorithm is the reduction of the number of rendering passes and hence, probability of missing some layers.

2.3 Application to Viewing edges computing

In the method we propose, no approximation is applied. An occluding O_n with M points ($m = 1..M$) is drawn as a generalized cone of M faces. Each face F_n^m is bound by the rays r_n^m and $r_n^{(m+1)\%M}$, see Figure 4. We consider the pinhole camera model and we refer by A_n to the camera matrix of the camera C_n, by c_n to its center, and by f_n to its focal length. If o_n^m is a point of O_n with the coordinates (x_i, x_i) in the image plane, then its local 3D coordinates are (x_i, x_i, f_n).

Each point v_n^m of the ray r_n^m associated to the contour point o_n^m has the following coordinates in the world coordinate system:

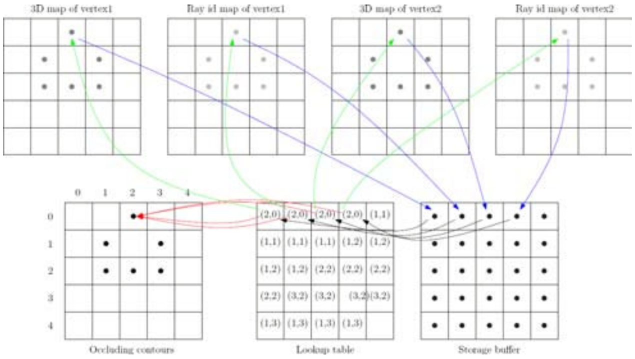


Figure 5. Viewing edge storage scheme.

$$v_n^m = c_n + \alpha_n A^{-1} o_n^m \qquad (4)$$

where α_n is a real constant.

We set α_n in (4) to an appropriate value that determines the depth of each cone face to be drawn. We take into account the distance D_n between the camera and the farthest point of the 3D covered area as follows:

$$\alpha_n = \frac{D_n}{f_n} \qquad (5)$$

This setting ensures that each ray intersects all viewing cones whatever the position of the object in the covered scene. After setting α_n, it becomes possible to find for each ray r_n^m, the farthest point v_n^m from the camera center c_n. A cone face F_n^m associated to a ray r_n^m is defined by the ordered vertices (c_n, v_n^m, v_n^{m+1}). In order to be able to identify the viewing edges sharing the same vertices, we label each ray with a unique id (cone face), as shown in Figure 4. This id is passed to the cone face during the drawing step as color information. We employ the previously described algorithm with the possibility of saving all valid edges instead of the only first intersection. The straightforward solution for the storage is to read back the data after each iteration. However, the readback is the main bottleneck of the GPU. The depth peeling [19] was just proposed to overcome this limitation by offering the possibility to update at each iteration the depth for only those intersections which haven't been set yet. However for us, not only the first intersections are targeted, but rather all intersections.

2.4 Viewing edges storage

As explained, we are interested in the occluding contour points only. These points are few as compared to the image points. The idea we propose is to save the edges passing the test to a storage buffer allocated as a RGBA texture in the GPU memory. This buffer is read-back once all edges extracted. We need for that to add one more rendering pass. This pass consists in drawing a full screen quad in a projective geometry. Five textures are attached as inputs: two textures for each vertex containing the 3D position and the id of the corresponding intersecting ray, and one texture loaded once at the beginning and serving as a lookup table for each point to get the coordinates of the texture point to store. Let us refer by *3DMap1* and *IdMap1* the 3D and color maps of the first vertex, and by *3DMap2* and *IdMap2* to those of the second vertex of the

```
Algorithm 1 Lookup table initialization
for $i = 0$ to $it$ do
  for $j$ to $M$ do
    for $k = 0$ to $4$ do
      $lut[(i * j + k) \ mod \ width, \ (i * j + k) \ div \ width]$
          $\leftarrow coordinates(C[j])$;
    end for
  end for
end for
```

```
Algorithm 2 Storage kernel
if $M * it * 4 <= y * width + x < M * (it + 1) * 4$ then
  $(a, b) \leftarrow lup[x, y]$;
  if $(y * width + x) \ mod \ 4 = 0$ then
    storage[x,y]=3DMap1[a,b];
  end if
  if $(y * width + x) \ mod \ 4 = 1$ then
    storage[x,y]=IdMap1[a,b];
  end if
  if $(y * width + x) \ mod \ 4 = 2$ then
    storage[x,y]=3DMap2[a,b];
  end if
  if $(y * width + x) \ mod \ 4 = 3$ then
    storage[x,y]=IdMap2[a,b];
  end if
end if
```

edge. The color map contains the id of the intersecting edges. Also we refer by *lut* to the lookup table texture, by *width* and *height* to the texture and image size, by M to the number of occluding contour points, and by *it* to the number of iterations. *lut* is initialized once and load loaded to the *GPU* memory. It contains a list of subsequent occurrences of the list of the occluding points, each of which is duplicated four times, as shown in Figure 5. The lookup table initialization is illustrated by Algorithm 1.

The kernel (fragment shader) invoked at point level, to store the edge vertices, reads the coordinates from *lut* and uses them to locate the information to store from one of the four vertex textures. This is done only if the invoking point is located within the region concerned by the current iteration. If the coordinates of this point in the storage buffer are (x, y), then the storage is as in Algorithm 2: The maximum number of iteration that can be processed within the storage capacity of one buffer is given by:

$$MaxIt = \frac{width \times height}{4 \times M} \tag{6}$$

2.5 Implementation

We implemented the described edge extraction scheme as a multi-pass rendering on GPU. We made use of OPENGL as an API and C-like shading language (CG) of NVIDIA to write the shaders. We made use of a Frame Buffer Object as an off-screen rendering target instead of the screen. To this FBO, we bind a depth buffer, a stencil buffer, and a shadow buffer. The depth and shadow buffer serve to the two-sided buffer test [6], while the stencil buffer is for counting the layers. We bind also a storage buffer and a lookup texture to the FBO. At each rendering step, appropriate textures are attached as input(s) and output(s). In addition, one fragment and/or one vertex shaders are loaded to the programmable vertex and fragment processors in order to achieve one step of the extraction algorithm. Figure 6 shows the extracted edges using four silhouette images of a bunny taken from 4 viewpoints.

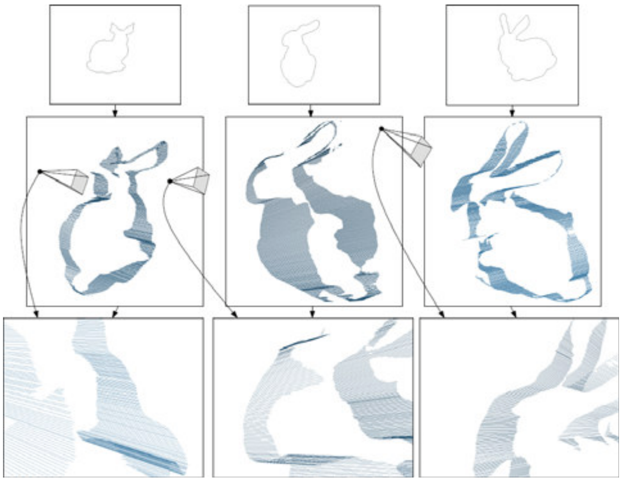


Figure 6. The extracted viewing edges.

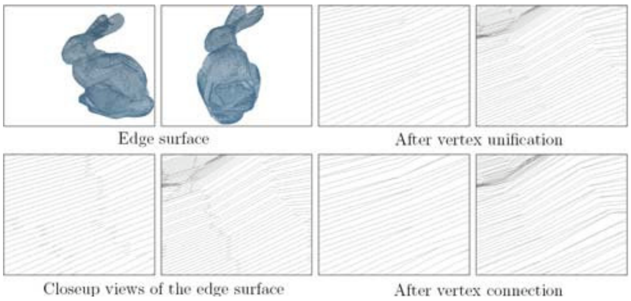


Figure 7. Edge surface.

3 VH Surface Construction

After been extracted from all views, the viewing edges are merged together to construct the VH surface as shown in Figure 7. A vertex, being the intersection of two or more edges issued for different cameras, can be computed with slightly different 3D position in each camera. This fact makes the extracted edges disconnected from each other. Thus, we need to recover a unique 3D position for each vertex. We compute a unique 3D position as the mean of its coordinates estimated by all views. Even after connecting the edges, still some edges remains disconnected. This fact is due to the resolution difference between the cameras. We join these edges to the closest neighboring vertices (issued from a neighboring point of the same contour). The VH face generation can be processed for each camera separately in a step prior to the rectification of the 3D positions of the vertices. The faces are generated by connecting the appropriate edges generated by neighboring contour points. We need to consider the predefined order of the contours in generating the faces. The reconstruction results will be presented in the

Table 1: Processing time evaluation (in *ms*).

	Bunny		Shark		Maiko	
	Points	time	Points	time	Points	time
Camera 1	887	110	720	109	1109	156
Camera 2	1062	140	680	109	1306	172
Camera 3	960	125	1256	140	1209	172
Camera 4	971	125	887	125	1075	156
Camera 5	1052	140	703	109	1316	170
Camera 6	1066	140	1069	125	1565	156
Camera 7	1024	141	1159	140	1185	156
Camera 8	1060	140	966	125	1413	172

Table 1: Comparison with the reconstruction using down-scaled images.

	640×480		320×240	
	Points	Time(ms)	Points	Time(ms)
Camera 1	1334	172	666	46
Camera 2	1028	140	508	31
Camera 3	973	141	482	31
Camera 4	1242	156	611	31
Camera 5	1302	172	648	47
Camera 6	1121	156	559	32
Camera 7	1093	141	543	31
Camera 8	1061	141	531	32

next section, in addition of the evaluation of the overall VH reconstruction.

4 Experimental results

The presented scheme was implemented on a P4 PC with 1[GB] memory and equipped with a NVIDIA GeForce 9700 graphics card. We tested the reconstruction scheme on 2 synthetic datasets. We used 2 shapes provided by Princeton Shape Benchmark [20] to generate the silhouettes from 8 viewpoints. We also tested the reconstruction scheme on real data provided by Matsuyama Laboratory of Kyoto University in the form of 8 silhouette images of a Kimono Lady (Maiko) and the related camera parameters. Figure 8 shows four virtual views of the reconstructed VH of the three examples.

Table 1, summarizes the processing time for each camera and for each dataset. This can allow us to get an idea about the processing time when the scheme is distributively implemented on multiple PCs, each of which is connected to one camera. In this case, the processing time is the largest time among all cameras, added to the time needed for vertex unification, which is 31[ms]. The processing time varies from one camera to another due to the complexity of the scene that varies with respect to each viewpoint, yielding different number of depth layers. Also it is due to the area occupied by each silhouette. In fact, we used scissoring technique to speedup the rendering time. Thus, the rendering is allowed only in the region defined by the bounding rectangle of the silhouette.

As to evaluate the processing time of the proposed algorithm, we considered the algorithm proposed by Matusik [21] which is supposed to be the first to compute the VH polyhedral representation in an interactive frame rate. Implemented on a 1[GHz] Pentium III machine with 1[GB] of RAM, this method reconstruct the VH in 2[sec] for 8 viewpoints with 641 contour points in each view. From Table 1 and if we consider an implementation on one PC, the processing time varies between 1012 [ms] for the *shark* dataset and 1343[ms] for *Maiko*, with much more contour points for each view. In order to get similar contour points, we scaled down the silhouette images. Table 2 summarizes the processing time for the reconstructed VH using 8 640×480 images and using the same images but scaled down to 320×240. To total processing time passes from 1250[ms] in the original scale to [312 ms] in the lower scale. This means that we could speedup the process 4 times by down scaling the image to the half size (in each direction).

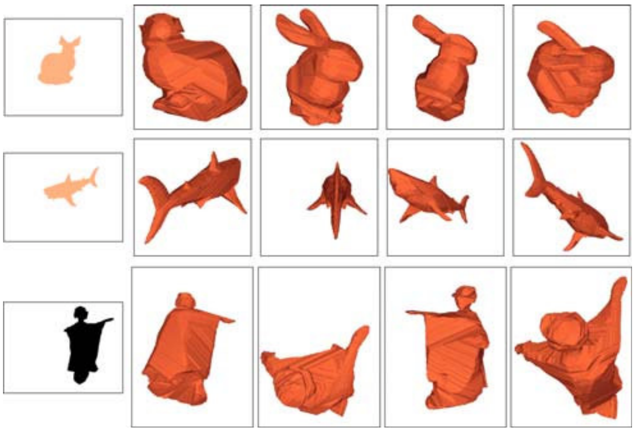


Figure 8: VH reconstruction result.

The graph of Figure 9 shows the number of traversed depth layers of the drawn cones from different number of viewpoints using both traversing methods (Direct CSG and ours). We can notice that our method requires less iterations than the native CSG method do to visit all candidate depth layers. Also, the difference increases for more cameras. In addition, we plot the number of viewing edges extracted after each iteration on the graph of Figure 10. 12 iterations is the number of iteration required by all cameras to recover all viewing edges. In our tests, we set the number of iterations to 15 for all models.

5 Conclusion

In this chapter, I presented a new method for shape from occluding contours. I proposed a CSG-like method for a fast depth layer traversing and viewing edge computing, rather than just rendering the depth of the shape from a desired view. The viewing edges are extracted for each camera separately without camera-camera projection. This fact allows the system to be implemented in a distributed system where each camera is connected to one PC and operates independently of the rest. This design will provide a faster processing. The proposed reconstruction scheme doesn't need any approximation of the silhouette and, hence, preserves the details of the shape.

References

[1] I.M. Researcher: “Read My Excellent Paper,” Some Great Journal, vol.xx, no.xx, pp.xx-xx, 200X.

[2] MVA Conference:
http://www.cvl.iis.u-tokyo.ac.jp/mva/

[1] M. Tarini, M. Callieri, C. Montani, C. Rocchini, “Marching Intersections: An Efficient Approach to Shape from

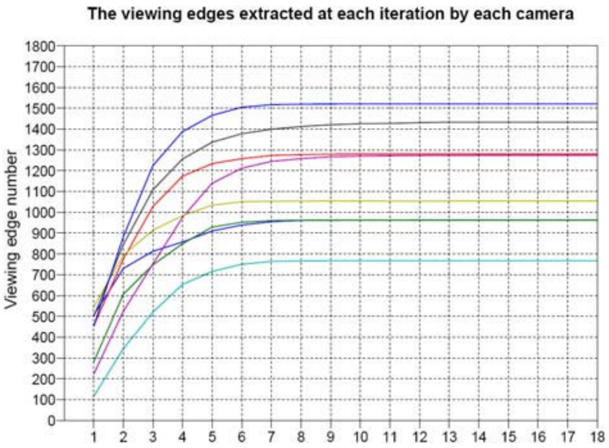


Figure 10: Number of extracted edges within iterations.

Silhouette", In Proceedings of the Vision, Modeling, and Visualization Conference, pp. 255-262, 2002.

[2] C. Rocchini, P. Cignoni, F. Ganovelli, C. Montani, P. Pingi, R. Scopigno, "Marching Intersections: an Efficient Resampling Algorithm for Surface Management", In Proceedings of the International Conference on Shape Modeling and Applications, pp. 296-305, 2001.

[3] A. Laurentini, "The visual hull concept for silhouettebased image understanding", IEEE Transactions on Pattern Analysis and Machine intelligence, Vol. 16(2), pp.150-162, 1994.

[4] B.G. Baumgart: "Geometric Modeling for Computer Vision", PhD thesis, Stanford University, 1974.

[5] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan: "Image-based visual hulls," In Proceedings of the ACM Computer Graphics (SIGGRAPH), pp. 369-374, 2000.

[6] S. Guha, S. Krishnan, K. Munagala, and S. Venkat: "Application of the two-sided depth test to CSG rendering," In Proceedings of Symposium on Interactive 3D Rendering, pp. 177-180, 2003.

[7] N. Stewart, G. Leach, and S. John: "An improved Zbuffer CSG rendering algorithm," In Proceedings of the SIGGRAPH/Eurographics workshop on graphics hardware, pp. 25-30, 1998.

[8] T. F. Wiegand: "Interactive rendering of CSG models," Computer Graphics Forum, Vol. 15(4), pp. 249-261, 1996.

[9] M. Li, M. Magnor, and H.P. Seidel: "Hardware-accelerated visual hull reconstruction and rendering," In Proceedings of Graphics Interface, pp. 65-71, 2003.

[10] M. Li, M. Magnor, and H.P. Seidel: "A Hybrid Hardware-Accelerated Algorithm for High Quality Rendering of Visual Hulls," In Proceedings of Graphics Interface, pp. 41-48, 2004.

[11] K.M. Cheung, T. Kanade, J.Y. Bouguet, and M. Holler: "A real time system for robust 3d voxel reconstruction of human motions," In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Vol. 2, pp. 714-720, 2000.

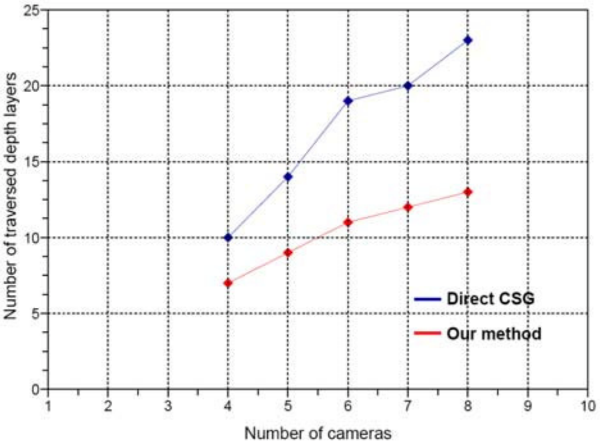


Figure 9: Direct CSG vs. Our depth layers traversal method: Number of depth traversed.

[12] W.N. Martin and J.K. Aggarwal: "Volumetric description of objects from multiple views," IEEE Transactions on Pattern Analysis and Machine intelligence, Vol. 5(2), pp. 150-158, 1983.

[13] C.H. Chien and J.K. Aggarwal: "Volume/surface octrees for the representation of three-dimensional objects," Computer Vision, Graphics and Image Processing, Vol. 36(1), pp. 100-113, 1986.

[14] R. Szeliski: "Rapid Octree Construction from Image Sequences," Computer Vision, Graphics and Image Processing, Vol. 58 (1), pp 23-32, 1993.

[15] J.J. Koenderink: "What Does the Occluding Contour Tell us About Solid Shape?," Perception, Vol.13, pp. 321-330, 1984.

[16] R. Cipolla and A. Blake: "Surface Shape from the Deformation of Apparent Contours," International Journalof Computer Vision, Vol. 9, pp. 83-112, 1992.

[17] E. Boyer and M.-O. Berger: "3D surface reconstruction using occluding contours," International Journal of Computer Vision, Vol. 22(3), pp. 219-233, 1997.

[18] J. Goldfeather, J. P. M. Hultquist, and H. Fuchs: "Fast constructive-solid geometry display in the pixel powers graphics system," In Proceedings of the ACM Computer Graphics (SIGGRAPH), pp. 107-116, 1986.

[19] C. Everit: "Interactive order-independent transparency," Technical report, 2002, Nvidia Corporation, http://developer.nvidia.com.

[20] Princeton Shape Retrieval and Analysis Group, 'Princeton Shape Benchmark', http://shape.cs.princeton.edu/benchmark/

[21] W. Matusik, C. Buehler, L. McMillan, and S. Gortler: "An Efficient Visual Hull Computation Algorithm," Technical Memo 623, LCS, MIT, 200