

GPU-Based Shape from Silhouettes

Sofiane Yous^{*}
NAIST, Japan

Hamid Laga[†]
TITECH, Japan

Masatsugu Kidode[‡]
NAIST, Japan

Kunihiro Chihara[§]
NAIST, Japan

Abstract

In this paper, we present a new method for surface-based shape reconstruction from a set of silhouette images. We propose to project the viewing cones from all viewpoints to the 3D space and compute the intersections that represent the vertices of the Visual Hull (VH). We propose a method for fast traversal of the layers of the projected cones and retrieve the viewing edges that lie to the surface of the VH. Taking advantage of the power of Graphics Processing Units (GPU), the proposed method achieves a real-time full reconstruction of VH rather than rendering a novel view of the VH. The experiments on several data sets, including real data, demonstrate the efficiency of the method for real-time visual hull reconstruction.

CR Categories: I.2.10 [Artificial Intelligence]: Vision and Scene Understanding—Shape; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.4.8 [Image Processing and Computer Vision]: Scene Analysis—Shape;

Keywords: shape from silhouettes, constructive solid geometry, graphics processing unit

1 Introduction

The 3D shape of a real object is a requirement in a variety of modeling and 3D multimedia applications. This shape can be obtained from the silhouette images of the object taken from different viewpoints. This concept was first introduced by Baumgart [Baumgart 1974] and later given the name of Visual Hull (VH) by Laurentini [Laurentini 1994]. The VH is defined as the maximal approximation of the object that reproduces the silhouettes of all viewpoints.

The use of silhouette images to estimate the 3D shape of an object was motivated by the ease of obtaining silhouette images and by the ease of implementation. Several methods have been proposed for VH reconstruction and/or rendering. For visualization applications, the exact reconstruction of the VH is not required. Image-based VH (IBVH) technique [Matusik et al. 2000] suffices to generate a novel view of the VH from a desired viewpoint. Hardware-based acceleration of IBVH was proposed through Direct Constructive Solid Geometry (CSG) rendering [Guha et al. 2003; Stewart et al. 1998; Wiegand 1994].

Applications such as object modeling and 3D digital archiving require a full reconstruction of the object’s shape. To this end, voxel-based VH reconstruction recovers a volumetric representation of the object. Volume carving methods split up the 3D space into a 3D grid of voxels that are labeled as volume voxels if they belong to all silhouette cones. This class of methods, however, suffers from the extensive computation load and the memory overhead. To overcome these limitations, octree representation was proposed by Chien and Aggarwal [Chien and Aggarwal 1986] to substitute the voxel representation. For the same purpose, Marching Intersections (MI) was proposed by Rocchini [Rocchini et al. 2001] as a re-sampling method for surface management and adapted later by Tarini et al. [Tarini et al. 2002] for volumetric shape reconstruction.

Surface-based methods also target an exact reconstruction of the VH, but as a 3D polyhedral surface [Matusik et al. 2000]. The surface vertices and faces are estimated by intersecting the generalized cones generated from the occluding contours of the silhouette images [Baumgart 1974; Koenderink 1984; Boyer and Berger 1997]. This class of methods produces artifact-free VH and requires much less computations and memory as compared 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. Unlike the previous methods, we propose to draw the viewing cones from all viewpoints to the 3D space and compute the intersections that represent the vertices of the VH. We use the power of GPUs to explore the drawn scene and retrieve the viewing edges in real time.

2 Overview and contribution

Figure 1 illustrates the method we propose to reconstruct the shape of an object given a set of silhouettes taken from different viewpoints. For each viewpoint, we project the viewing edges from the other viewpoints. The vertices of the viewing edges related to this viewpoint are computed by traversing the layers of the drawn scene and recovering all points that 1) project to the occluding points (points of the occluding contour) on the image plane and 2) belong to the intersection of all drawn cones. Subsequently, the viewing edges computed from all viewpoints are merged together to reconstruct the VH surface.

Implementing this method on CPU would require a long processing time. We propose a GPU-based implementation of this method where the vertices of the viewing edges are computed using a CSG-like method. We propose a selective depth-layers traversal method instead of the direct CSG used in image-based rendering. Our method requires less passes to traverse the depth layers.

We also propose a new storage method that runs on GPU. The goal is to avoid reading back from the GPU after each pass to recover the computed vertices knowing that the readback is the main GPU bottleneck.

The rest of this paper is organized as follows: in the next section, we will introduce the traversal of the drawn scene and the storage of the viewing edges. In section 4, we explain the VH surface reconstruction using the stored viewing edges. The overall scheme is evaluated in section 5 before concluding this paper in section 6.

^{*}e-mail: yous-s@is.naist.jp

[†]e-mail: hamid@img.cs.titech.ac.jp

[‡]e-mail: kidode@is.naist.jp

[§]e-mail: chihara@is.naist.jp

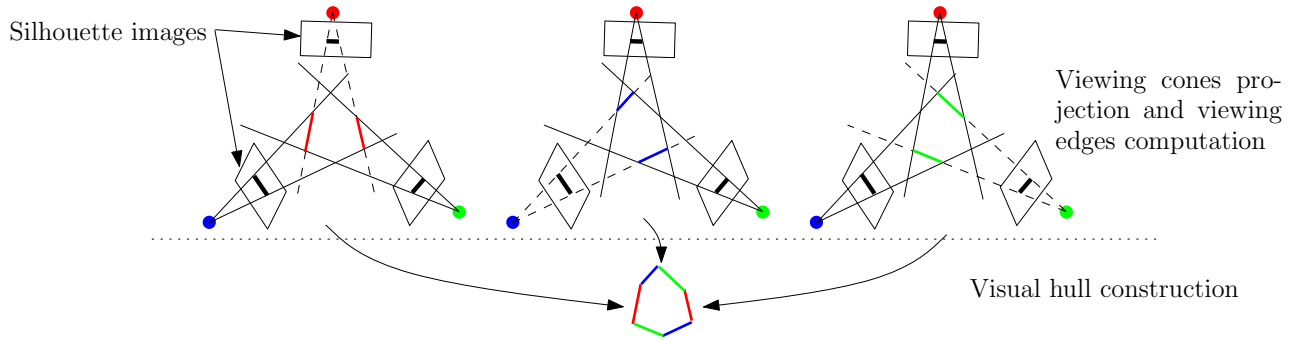


Figure 1: The Visual Hull reconstruction scheme (2D)

3 Viewing Edges Computation

When we go through the drawn cones with respect to a given viewpoint, we can meet two types of layers: front and back layers. For each occluding point (a point on the contours of the silhouette) corresponds, at least, one viewing edge in the drawn cones. This viewing edge is a line segment which is part of the ray generated from the viewpoint and passing through the occluding point and defined by two points. This line segment is limited by two points. The closer point to the viewpoint is the intersection of the ray with a front layer, while the other point is the intersection of the same ray with the back layer immediately facing the first front layer. Both layers must lie to the viewing cones of all viewpoints. However, checking this condition for one layer is enough.

Direct-CSG [Guha et al. 2003; Li et al. 2004] is an image-based rendering that uses the depth layer traversal principle to render a view of the VH from a novel viewpoint. The goal is to recover the intersection of the rays issued by all image points of the target viewpoint with the first front layer that belong to all projected viewing cones. This method can be adapted to the reconstruction of the VH instead of rendering a novel view of it. To achieve this, we need to consider that:

1. The traversal is processed with respect to all reference viewpoints, instead of one virtual viewpoint.
2. For a given reference view, the traversal is processed with respect to the silhouette occluding only.
3. For a given occluding point of a given viewpoint, we must compute intersections of the corresponding ray with all layers (front and back) that belong to the viewing cones of all viewpoints.

The Direct-CSG method checks all front layers if they belong to the viewing cones of all viewpoints. However, a front layer that belong to the surface of the VH is always facing a back layer. We made use of this property to propose a selective depth layer traversal method.

3.1 Selective Depth layers Traversal

The CSG-based rendering was proposed by Goldfeather [Goldfeather et al. 1986] and used later by Guha [Guha et al. 2003] and Li et al. [Li et al. 2004] 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. Suppose the following tree:

$$(O_1 \cup (O_2 \cap O_3)) \setminus O_4 \quad (1)$$

This tree is normalized as follows:

$$\underbrace{O_1 \cap \overline{O_4}}_{P_1} \cup \underbrace{O_2 \cap O_1 \cap \overline{O_4}}_{P_2} \quad (2)$$

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 with respect to a desired viewpoint d , then p belongs to the product if:

$$f(d, p) - b(d, p) = |P| \quad (3)$$

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

A VH reconstructed from a set of viewpoints can be regarded as a normalized tree 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 [Guha et al. 2003; Li et al. 2004] traverse all front depth layers and keep the intersection of the rays from the image points that intersect the layers that satisfy Equation 3.

We can see in Figure 2 that a front face that belongs to the VH is always immediately facing a back face. This means that, given a succession of front faces, only the last face can verify this condition, all remaining faces can be discarded. We can access directly to this face by rendering a back layer followed by rendering the farthest front layer with shorter depth than the rendered back layer. Similarly, only the first of a list of back faces is traversed, all the others can be discarded 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. Based on this, we propose the following algorithm for layers traversal:

1. Render the first depth layer of back faces.
2. repeat
 - (a) Compute the difference between the front and back faces separating the traversed depth layer and the desired view position and keep the depth of the last depth with respect to the camera.
 - (b) Save the depths of the points that satisfies Equation 3.
 - (c) Render the first front layer having a depth greater than the current back layer (skip all back layers separating the two layers).
 - (d) Render the next depth layer of back faces having depth longer than the rendered front layer (skip all front faces).

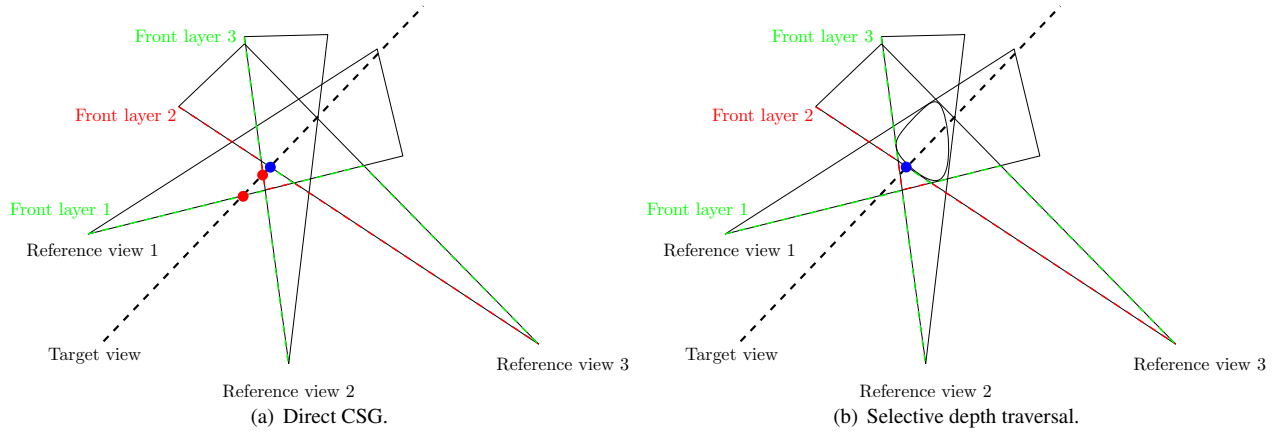


Figure 2: Direct CSG vs. Selective depth layers traversal in 2D: The traversed depth layers are drawn in dashed lines. The points that are positively tested to their belonging to all viewing cones are shown in blue, while those which failed in red.

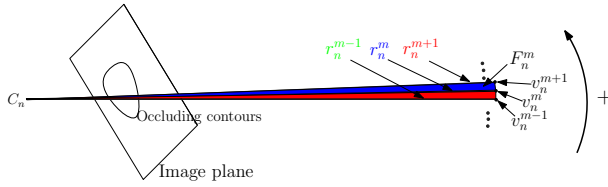


Figure 3: The silhouette generalized cone.

(e) Until no layer is returned.

The advantage of this new algorithm is to reduce the number of needed renderings and reduce the probability of missing some layers if the number of rendering passes is set to a fixed value. This can be noticed in Fig. 2. In the direct-CSG rendering method, 3 layers were traversed against 1 in our method.

The other advantage is the fact that if we add more cameras, it is not necessary to increase the number of passes iterations, unless the complexity of the scene increases.

3.2 Application to Viewing Edges Computing

In the proposed method, unlike most of the surface-based VH reconstruction proposed so far, no approximation of the occluding contours is required. We perform the 3D reconstruction from N silhouette images taken by N calibrated cameras C_n , $n = 1 \dots N$. We denote by c_n the center of the camera C_n . An occluding contour O_n with M points is drawn as a generalized cone of M faces. Each face f_n^m ($m = 1 \dots M$) is bound by the rays r_n^m and $r_n^{(m+1)\%M}$, see Figure 3. We consider the pinhole camera model and we refer by A_n to the matrix on 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, y_i) in the image plane, then its 3D coordinates in the camera coordinates system are (x_i, y_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:

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

where α_n is a real constant.

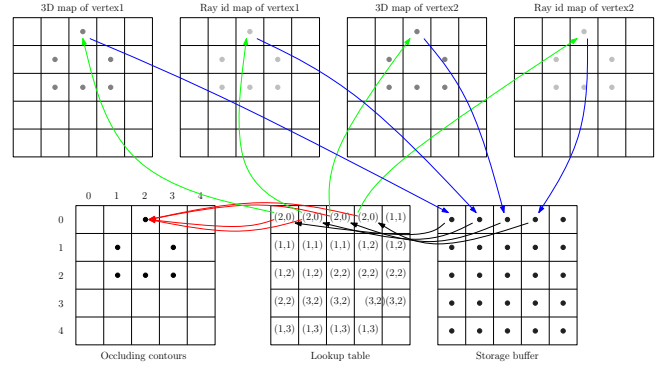


Figure 4: Viewing edge storage scheme.

We set α_n 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:

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

After setting α_n in Equation 5, finding the farthest point from the camera center c_n for each ray r_n^m becomes straightforward. A cone face F_n^m associated to a ray r_n^m is defined by the ordered vertices $(c_n, v_n^m, v_n^{(m+1)\%M})$. 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 3. This id is passed to the cone face during the drawing step as color information. We employ the algorithm described so far in this section with the possibility of saving all valid edges instead of the only first intersection.

3.3 Viewing Edge Storage

As explained in section 2, 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 an 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 of drawing a full screen quad in a projective geometry. Five textures are attached as input: two textures for each vertex containing the 3D position and

Algorithm 1 Lookup table initialization

```
for  $i = 0$  to  $it - 1$  do
  for  $j = 0$  to  $M - 1$  do
    for  $k = 0$  to 3 do
       $lut[(i * j + k) \bmod width, (i * j + k) \div width]$ 
         $\leftarrow coordinates(C[j]);$ 
    end for
  end for
end for
```

Algorithm 2 Storage Algorithm

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

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 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 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 described in Algorithm 1 and Figure 4.

The fragment shader (kernel) 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:

$$MaxIt = \frac{width \times high}{4 \times M} \quad (6)$$

3.4 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 (FBO) as an off-screen rendering target instead of the screen. To this FBO, we bind a depth buffer and a stencil buffer. We replace the shadow buffer in the two-sided buffer test [Guha et al. 2003], using a second test at fragment level. The stencil buffer is for counting the layers to check the belonging of a layer to the VH following Equation 3. We bind also a storage buffer and a lookup texture to the FBO. At each rendering

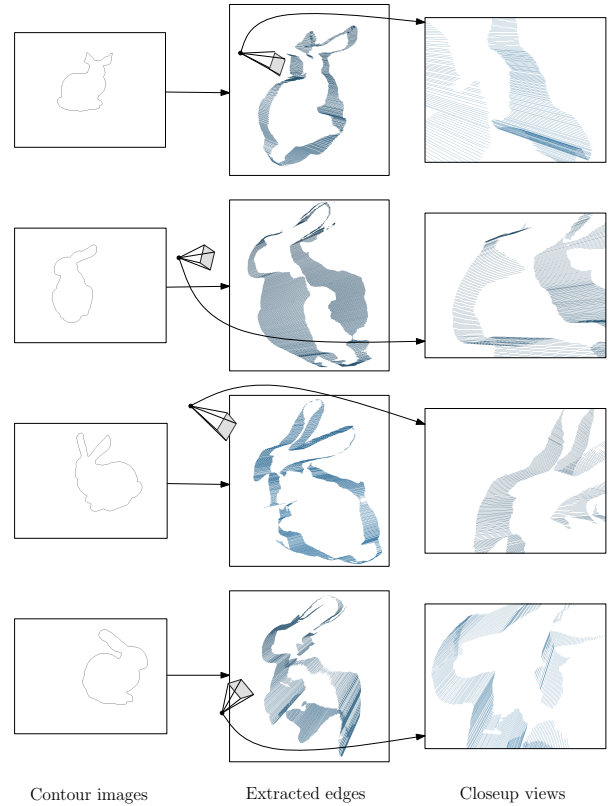


Figure 5: Viewing edge extraction steps.

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 5 shows the extracted edges using four silhouette images of a bunny taken from 4 view-points.

4 VH Surface Construction

After being extracted from all views, the viewing edges are merged together to construct the VH surface as shown in Figure 6(a). 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 (Figure 6(b)). 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 (Figure 6(c)). 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), as in Figure 6(d). 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 adjacent contour points. We need to consider the predefined order of the contours in generating the faces. We may have several cases in generating these faces, as shown in Figure 7. The reconstruction results will be presented in the next section, in addition of the evaluation of the overall VH reconstruction.

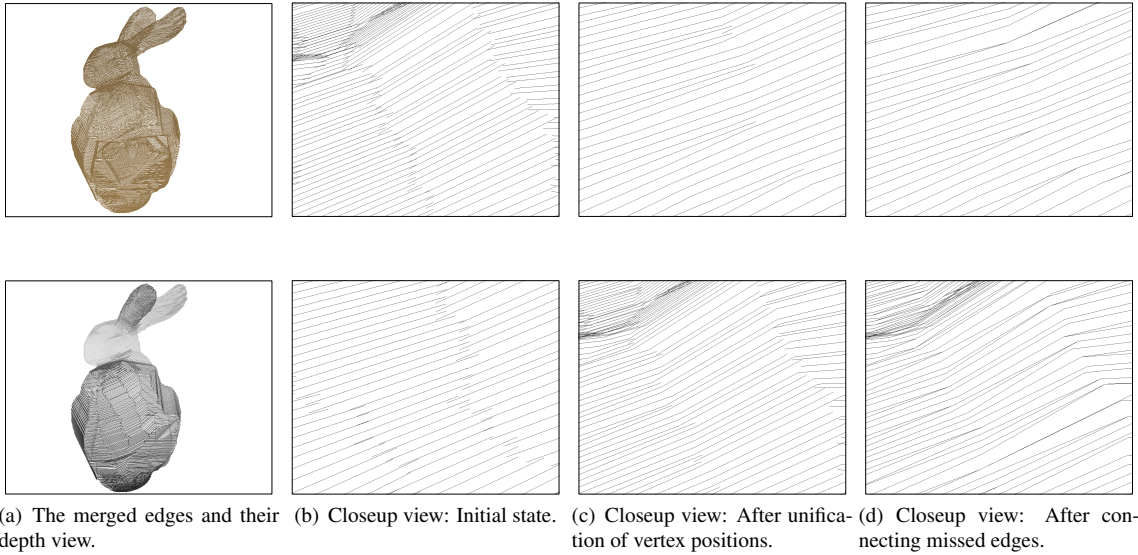


Figure 6: Surface construction: (a) A view of the viewing edge merged together from 8 viewpoints. (b) The viewing edges are disconnected from each other after extraction. (c) The edges are connected to each other using the associated id. (d) The disconnected edges due to resolution difference between views are connected.

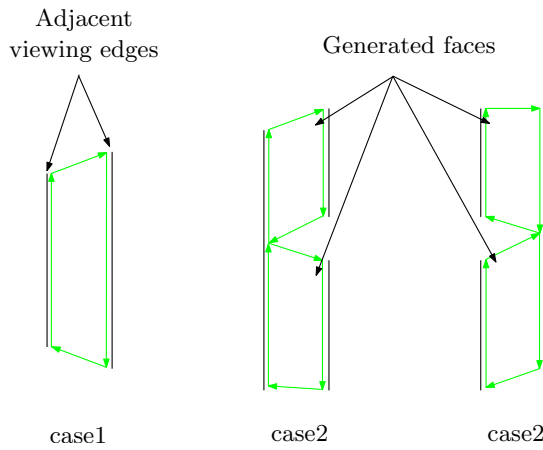


Figure 7: The different cases of face generation.

5 Experimental Results

The proposed scheme was implemented on a *Pentium4 3GHz* PC with 1GB memory and equipped with a *NVIDIA GeForce 7900 GTX* graphics card. We tested the reconstruction scheme on two synthetic datasets. We used two shapes provided by Princeton Shape Benchmark [Pri] to generate the silhouettes from eight viewpoints. We also tested the reconstruction scheme on real data provided by Matsuyama Laboratory of Kyoto University in the form of eight silhouette images of a Kimono Lady (Maiko) and the related camera parameters. The results of reconstruction are shown in Figures 8(a), 8(b), and 8(c) where the upper row shows the silhouettes and the last row, four virtual views of the reconstructed VH.

Table 1, summarizes the processing time for each camera and for each dataset. When the scheme is distributively implemented on multiple PCs, the processing time is the largest time among all cameras, added to the time needed for vertex unification, which is 31ms. The processing time varies from one camera to another due to the

Camera	Bunny		Shark		Maiko	
	Point count	Time (ms)	Point count	Time (ms)	Point count	Time (ms)
Camera1	887	110	720	109	1109	156
Camera2	1062	140	680	109	1306	172
Camera3	960	125	1256	140	1209	172
Camera4	971	125	887	125	1075	156
Camera5	1052	140	703	109	1316	172
Camera6	1066	140	1069	125	1565	156
Camera7	1024	141	1159	140	1185	156
Camera8	1060	140	966	125	1413	172

Table 1: Processing time evaluation: The processing time is calculated for each camera and for each dataset. The shown time concerns the viewing edge extraction and face generation. 'Point count' columns refer to the number of occluding contour points.

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.

From Table 1 and if we consider an implementation on one PC, the processing time varies between 1012ms for the 'shark' and 1343ms for 'Maiko'. Note that no approximation was applied to the silhouettes. If the application does not require all the details, the approximation of the silhouettes is recommended since it results in a faster processing. This approximation can be achieved by scaling down the silhouette images before extracting the occluding contours. Figure 9 shows the reconstructed VH using eight (8) 640×480 images and using the same images but scaled down to 320×240 . Table 2 summarizes the processing time where we can see that this processing time is 1250ms in the initial scale, while it is 312ms in the lower scale. That is to say that we could speedup the process four (4) times by down scaling the image to the half size in each direction. In [Matusik et al. 2003], the VH reconstruction using 8 viewpoints with 641 contour points in each view was achieved in 2sec on a 1GHz Pentium3 PC with 1GB of RAM.

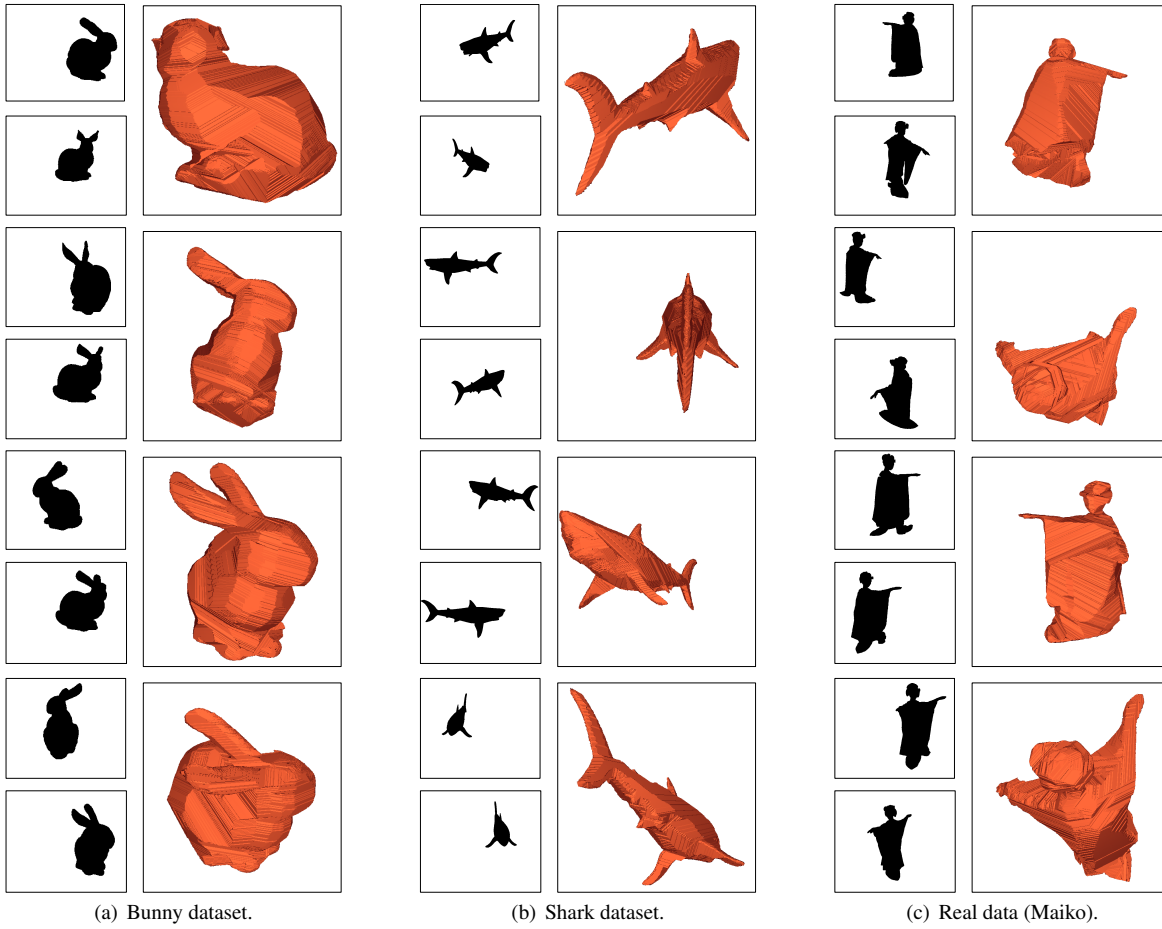


Figure 8: VH reconstruction result.

Camera	640 × 480		320 × 240	
	Points count	Time(ms)	Points count	Time(ms)
Camera1	1334	172	666	46
Camera2	1028	140	508	31
Camera3	973	141	482	31
Camera4	1242	156	611	31
Camera5	1302	172	648	47
Camera6	1121	156	559	32
Camera7	1093	141	543	31
Camera8	1061	141	531	32

Table 2: Comparison with the reconstruction using down-scaled images.

In order to prove the effectiveness of the depth layer traversal scheme, we computed the number of the traversed depth layers of the drawn cones with different number of cameras using the direct-CSG method and ours. We show the result in the graph presented in Figure 10, where we can notice that our method requires less iterations than the native CSG method do to visit all candidate depth layers. Furthermore, the more are the cameras the bigger is the difference. Also, we plot the number of viewing edges extracted after each iteration on the graph of figure 11. 12 iterations are required to permit all cameras to recover all viewing edges. In our tests, we set the number of iterations to 15 for all models.

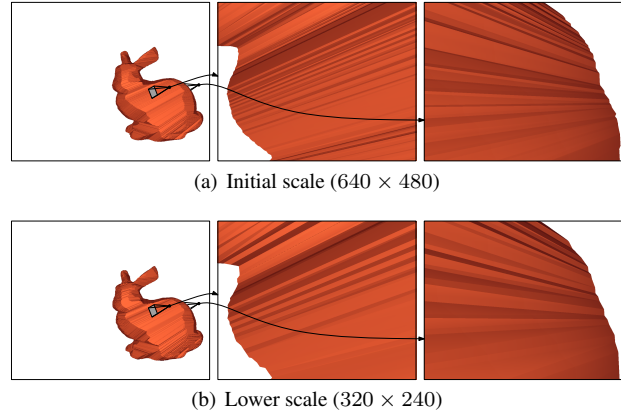


Figure 9: Reconstruction at lower scales.

6 Conclusion

In this paper, we proposed a new method for shape from occluding contours. We 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 pro-

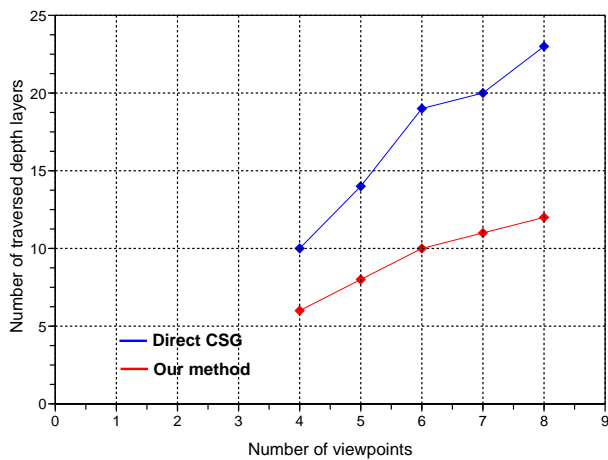


Figure 10: Evaluation the proposed method for depth layer traversing: Comparison with the native direct CSG.

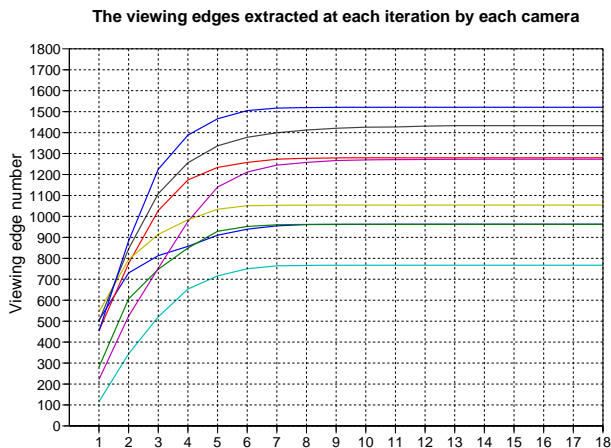


Figure 11: Variation of the number of extracted faces within iterations.

jection. 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

- BAUMGART, B. 1974. *Geometric Modeling for Computer Vision*. PhD thesis, Stanford University.
- BOYER, E., AND BERGER, M. O. 1997. 3d surface reconstruction using occluding contours. *Perception* 22, 3 (Mar.), 219–233.
- CHIEN, C., AND AGGARWAL, J. 1986. Volume/surface octrees for the representation of three-dimensional objects. *Computer Vision, Graphics and Image Processing* 36, 1 (Oct.), 100–113.
- GOLDFEATHER, J., HULTQUIST, J. P. M., AND FUCHS, H. 1986. Fast constructive solid geometry display in the pixel-processor graphics system. In *Proceedings of the ACM Computer Graphics - SIGGRAPH1986*, vol. 20, 107–116.

- GUHA, S., KRISHNANAND, S., MUNAGALA, K., AND VENKAT, S. 2003. Application of the two-sided depth test to csg rendering. In *Proceedings of Symposium on Interactive 3D Rendering*, 177–180.
- KOENDERINK, J. 1984. What does the occluding contour tell us about solid shape? *Perception* 13, 3, 321–233.
- LAURENTINI, A. 1994. The visual hull concept for silhouette-based image understanding. *IEEE Transactions on Pattern Analysis and Machine intelligence* 16, 2 (Feb.), 150–162.
- LI, M., MAGNOR, M., AND SEIDEL, H. 2004. Hybrid hardware-accelerated algorithm for high quality rendering of visual hulls. In *Proceedings of Graphics Interface*, 41–48.
- MARTIN, W., AND AGGARWAL, J. 1983. Volumetric description of objects from multiple views. *IEEE Transactions on Pattern Analysis and Machine intelligence* 5, 2 (Feb.), 150–158.
- MATUSIK, W., BUEHLER, C., RASKAR, R., GORTLER, S. J., AND MCMILLAN, L. 2000. Image-based visual hulls. In *Proceedings of the ACM Computer Graphics - SIGGRAPH2000*, 369–374.
- MATUSIK, W., BUEHLER, C., , AND MCMILLAN, L. 2001. Polyhedral visual hulls for real-time rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering*, 115–125.
- MATUSIK, W., BUEHLER, C., MCMILLAN, L., , AND GORTLER, S. 2003. An efficient visual hull computation algorithm. Technical Memo 623, LCS, MIT.
- Princeton shape retrieval and analysis group - princeton shape benchmark. In <http://shape.cs.princeton.edu/benchmark/>.
- ROCCHINI, C., CIGNONI, P., GANOVELLI, F., MONTANI, C., PINGI, P., AND SCOPIGNO, R. 2001. Marching intersections: an efficient resampling algorithm for surface management. In *Proceedings of the International Conference on Shape Modeling and Applications - SMI2001*, 296–305.
- STEWART, N., LEACH, G., AND JOHN, S. 1998. An improved z-buffer csg rendering algorithm. In *Proceedings of SIGGRAPH/Eurographics workshop on graphics hardware*, 25–30.
- TARINI, M., CALLIERI, M., MONTANI, C., AND ROCCHINI, C. 2002. Marching intersections: An efficient approach to shape-from-silhouette. In *Proceedings of the Vision, Modeling, and Visualization Conference*, 255–262.
- WIEGAND, T. F. 1994. Interactive rendering of csg models. *Computer Graphics Forum* 15, 4 (Oct.), 249–261.