EG UK Theory and Practice of Computer Graphics (2010)
John Collomosse, Ian Grimstead (Editors)

ID-Based Rendering of Silhouettes on GPU

Engin Deniz Diktas[1,2] and Ali Vahit Sahiner[2]

[1]Adeko Technologies, ULUTEK Technology Development Region, Uludag University, Bursa, Turkey
[2]Computer Engineering Department, Bogazici University, Istanbul, Turkey

Abstract

When rendering object-silhouettes preprocessing is generally done primarily on the CPU. To this end primitive normals must be made consistent and the silhouette-edges need to be extracted every time the view-point is changed. In this paper we propose a pure image-based GPU-method where IDs of triangles are rendered to a texture and silhouettes are extracted based on the information stored in that texture. With this method the geometry does not need to be preprocessed or reprocessed when the view-point or the geometry is changed. Another important advantage of the proposed method over any Z-buffer based method is that it does not require any threshold value to compare against the difference between depth-values of the neighboring pixels which is difficult to adjust in perspective projection.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Line and Curve Generation—

1. Introduction

Silhouette extraction and rendering is used in shadow volume construction, visibility tests as well as in non-photorealistic rendering as a tool to emphasize the outlines of objects in the absence of shading. Before silhouette extraction can be performed the geometry needs to be preprocessed in order to meet some specific mesh criteria. In the following discussion we restrict ourselves to triangle-meshes.

In order to extract the silhouettes from a mesh we need to determine the visibility of each triangle. After this we need to find out all edges whose one of its incident triangles is visible while the other one is not. In other words we need to trace the boundary between visible and invisible triangles. Since the silhouette boundary depends on the visibilities of all triangles, the visibility-tests as well as the silhouette-detection process must be repeated every time the view-point is changed. In order to correctly determine the silhouette, the normals of all triangles need to be consistent. That is all neighboring triangles must have the same winding. If even two triangles fail to meet this criterion, the visibility computation will not produce correct results. Another important property is that the mesh must have a connectivity information among the triangles over their edges: when visiting every edge we have to be able to check the triangles sharing that common edge: when all adjacent triangles are invisible or visible, then that edge is not a silhouette-edge, otherwise it is on the silhouette.

In the present paper we propose a method that does not rely on conventional cpu-based preprocessing and silhouette extraction methods. Our method is an online ID-based method which does not require any threshold values compared depth-based online silhouette rendering methods for which it is difficult to set the appropriate threshold Z-value, especially in perspective projection. In section 2 we review the corresponding literature. In section 3 we look at the conventional silhouette methods on CPU which require a preprocessing step every time the object topology changes (section 3.1) and silhouette extraction methods (section 3.2). In section 4 we propose our method by first discussing two possible candidates: ID-list-storage versus vertex-storage both using textures containing ID-outputs. We emphasize the difficulties with list-storage method and advantages of vertex-storage method which we choose of our implementation. Finally in section 5 we give test results.

2. Related work

Since our method is an image-based method, we will review the literature related to image-based silhouette extraction

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG10/017-024

methods. A broader review of silhouette edge rendering can be found in [IFH*03]. The common point of all image-based silhouette edge rendering algorithms is that they operate entirely on data stored in buffers obtained in a pre-rendering pass, and does not modify or know about the geometry in the scene. These methods have the advantage that they do not require any preprocessing step for rendering.

In [ST90] Saito and Takahashi first introduced the concept of a G-buffer, where information related to geometric properties of a scene (like depth, normals) are stored on a per-pixel basis. These buffers can also be used for deferred shading. Decaudin [Dec96] used G-buffers for toon-rendering. In these works the basic idea is to apply image processing techniques on various buffers of information in order to extract the silhouette boundaries. By looking for discontinuities in neighboring Z-buffer and/or normal values, most silhouette edge locations can be found.

Rendering the scene in ambient colors can also be used to detect edges which could be missed by the previous two techniques. Card and Mitchell [CM02] perform these image processing operations in real time by first using vertex-shaders to render the world space normals and z-depths of a scene to a texture. The normals are written as a normal map to the color channels and the most significant byte of Z-depths as the alpha channel. Once this image is created, the next step is to find the silhouette, boundary, and crease edges. The idea is to render a screen-filling quadrilateral with the normal map and the Z-depth map (in the alpha channel) and detect edge discontinuities. The idea is to sample the same texture six times in a single pass and implement a Sobel edge detection filter. The texture is sampled six times by sending six pairs of texture coordinates down with the quadrilateral. This filter is actually applied twice to the texture, once along each axis, and the two resulting images are composited. One other feature is that the thickness of the edges generated can be expanded or eroded by using further image processing techniques. This algorithm has a number of advantages: Since the method is image-based, meshes do not have to be connected or even consistent. CPU is not involved in creating and traversing edge lists. There are relatively few flaws with the technique: for nearly edge-on surfaces, the Z-depth comparison filter can falsely detect a silhouette edge pixel across the surface. Another problem with z-depth comparison is that if the differences are minimal, then the silhouette edge can be missed. For example, a sheet of paper on a desk will usually have its edges missed. Similarly, the normal map filter will miss the edges of this piece of paper, since the normals are identical [AMHH08]. One way to avoid problems related to methods using z-buffer and normal-buffer is to use ID-rendering. In [Dec96] such cases are detected by adding a filter on an ambient or object ID color rendering of the scene. This is still not foolproof: for example, a piece of paper folded onto itself will still create undetectable edges where the edges overlap [Her99].

3. Conventional silhouette extraction on CPU

In general, the triangular mesh does not come in a favorable format, where all normals are consistent and the necessary mesh structure is built. In our framework, the triangles are allowed to be loaded in a format called triangle soup, with no connectivity information and no guarantee of normal consistency. In addition to that vertices may be repeated for all triangles sharing them, so they need to be welded. Therefore we need to perform a preprocessing step where the mesh connectivity and normal-consistency properties of the triangle-soup are established. After these preprocessing steps a number of methods can be used to extract the silhouettes.

3.1. Preprocessing steps

The preprocessing step includes the following steps

1 Vertex Welding
2 Establishing triangle-connectivity via edges
3 Making all normals consistent in every connected component

Vertex welding is a necessary step, since otherwise we cannot find out which triangle-pairs share edges. So this is a necessary step for the second step, where all edges are assigned a unique pointer for each of their adjacent triangles.

For 2-manifold and closed (water-tight) meshes of arbitrary genus each edge will have exactly 2 adjacent triangles. If the object is 2-manifold but not closed, some edges will have not have two neighboring triangles: so they will always be on the silhouette. For general meshes, an edge may have an arbitrary number of adjacent triangles, so we chose to keep a list of pointers for fast access to neighboring triangles.

After the edge-structures are defined, we need to establish normal-consistency: for this we need to select a seed triangle and assume its winding is correct. Then starting with this seed-triangle we have to visit all of its neighbors via its edges and make their winding the same as the seed-triangle. During this visiting process we need to make sure that a previously processed triangle is not re-processed again or the iterations won't stop: this can be done by storing a flag for every triangle. When this breadth-first visiting is stopped, there may be other triangles not processed yet. This is the case if the mesh consists of more than one connected component. So we need an outer loop for checking if all triangles are processed or not. Every time the breadth-first normal-consistency recursion is finished, a new seed triangle needs to be selected from the remaining connected component(s) for the next recursion.

A note on deformable objects should be made at this point. As long as the deformation does not change the topology of the mesh, the preprocessed mesh structure can be reused in every extraction step. However, if the topology of the

object changes (e.g. due to rapture, fracture, splitting, genus-change etc), the preprocessing step needs to be updated for those parts of the objects where the topology changes occur. But since the mesh connectivity has been established prior to this topology modification, the updating procedure can be accelerated by keeping track of the topology-changing factors. These factors can be monitored directly from within the animation itself or from other methods that feed this topology modification information (e.g. an incremental surface reconstruction from time-varying 3D CT-data or other scalar field data). The important fact is that this preprocessing steps need to be continually checked and applied for the desired mesh properties in case of objects whose topology changes over time. In our framework we do not have any mechanism for incremental topology monitoring, we just rebuild whole data structures from scratch in every animation step.

3.2. Silhouette extraction

After pre-processing stage, the mesh is ready for silhouette-extraction: in this step the most straight-forward method is to find out visibility of every triangle and then loop on every edge and mark those with opposite visibility results. For non-time-critical applications this method is acceptable. However for complex scenes this process needs to be performed more rapidly. To this end there are some methods: In [SGG*00] a framework for efficiently clipping the rendering of coarse geometry to the exact silhouette of the original mesh model is described. A coarse mesh is obtained using progressive hulls that has the nesting property required for proper clipping. Given a perspective view, silhouettes are efficiently extracted from the original mesh using a precomputed search tree where hierarchical culling is achieved using pairs of anchored cones.

In [JC01] a hierarchical bounding volume hierarchy (sphere-tree) augmented with normal cone information is used to quickly determine the front-facing and back-facing triangle sets. Each node in the spatialized normal cone tree bounds both the enclosed geometry and the normals of the faces of that enclosed geometry, such that a hierarchy both in Euclidean space and normal space is constructed. Since many front and back-facing triangles are grouped together with the proper normal information, a single dot-product will be sufficient to eliminate all of the triangles and their shared edges contained therein, instead of performing as many dot-products as there are triangles in the bounding sphere of the corresponding normal cone.

Although these methods in the literature are quite fast when extracting silhouettes, their primary drawback is that they require expensive preprocessing steps, not just the ones we described before but also constructing certain hierarchial representations of the objects. For deformable objects and/or objects changing their topology, these construction phases must be repeated, which will inevitably degrade the overall rendering performance.

4. Proposed Method

As can be seen from the above discussions, CPU-based silhouette-extraction methods are quite involved, especially in cases where the topology of the triangle-mesh changes and performance is an issue. In order to avoid complex silhouette extraction methods and preprocessing steps, we decided to develop a method that will perform the silhouette drawing directly on the GPU and just with the triangle-soup information.

In order to render the silhouette of an object we need to be able to get some adjacency information for every pixel we render. This adjacency information can be obtained by looking at the neighboring pixels. Since we need to perform a neighboring pixel lookup, we have to generate this information in a pre-render pass. So we have to store this per-pixel information in a separate offscreen texture to be read in the final rendering pass.

In the second step, we have to think about what kind of information we need to store in the pixels of the offscreen texture: the cheapest way to encode the information about a triangle is to store its ID at every pixel location. Once we get the ID of a single triangle we can perform another look-up what kind of information is associated with that triangle.

Rendering the triangles' IDs is simple: we can pass this information as a vertex attribute at all vertices of the triangle. This attribute for every vertex needs to be passed as a FLAT varying variable to the fragment-shader. Finally the sole purpose of the fragment-shader is to output this ID-value (passed as flat varying variable to the fragment shader) to the texture for each pixel of the corresponding triangle. The FLAT requirement is necessary since we want to guarantee that the varying variable is never interpolated by the rasterization stage. This way we guarantee no approximation errors are introduced. The output of this first pass is a texture that holds the IDs of triangles at its pixels.

The second pass will be the rendering pass and we will render the silhouette based on information stored in this offscreen ID-texture. For this we render a full-screen quad and bind the silhouette-extraction shader. At every pixel we look at the current pixel coordinate and its 4 neighboring pixels. If all of the pixels have the same ID, then this means that all neighboring pixels belong to the same triangle and we do not need to check for any silhouette-pixel.

However if even one of the pixels have an ID different than the triangle-ID at the current location, then we need to check if the current pixel is on the silhouette. To this end we need to check if all triangles neighboring the current pixels are adjacent to the current triangle. For this we have two options: either this adjacency information can be encoded from a preprocessing step or we can deduce this information directly at fragment-shader. Both methods have advantages and disadvantages.

4.1. ID-List storage method

If the scene topology is static we can perform the previously-described preprocessing steps for once and construct a list of triangles adjacent to each triangle. This adjacency information can be encoded as ID-information. The advantage of this method is that the ID-information is very convenient and compact. For water-tight (closed 2-manifold) objects each triangle will have at most 3 neighbors. This ID-based adjacency information can be extracted with a single texture access, if the data is stored in a RGB-texture. The clear disadvantage is the preprocessing stage because for objects that may change topology we have to perform this preprocessing stage at every frame.

In addition to that vertices pose another problem, since they can be shared by many triangles, even by those which are not in the adjacency list of the triangle they belong to. This is shown in Figure 1: triangles adjacent to the triangle are numbered starting from 1 to 9. Triangles 1,5 and 7 shared the same edges as the center triangle, but all of the 9 triangles are adjacent to the center triangle at the vertices. Imagine taking samples at a position close to the top vertex: If the sampling strategy sees only the value 4 and 8, that pixel will be marked as a silhouette edge, since 4 and 8 are not in the adjacency list {1,5,7} of the center triangle. A low-sampling rate could easily miss the adjacent triangles and thus cause point-like artifacts. We have to increase the sampling rate close to the vertices to ensure that no such point-like artifacts occur. Since we do not know in advance how many triangles would be sharing this vertex we have to increase the sampling rate adaptively which would further complicate this strategy: a vertex with 8 adjacent triangles would require a sampling region consisting of 3x3=9 pixels while another vertex with 24 adjacent triangles would require a sampling region consisting of 5x5=25 pixels etc.

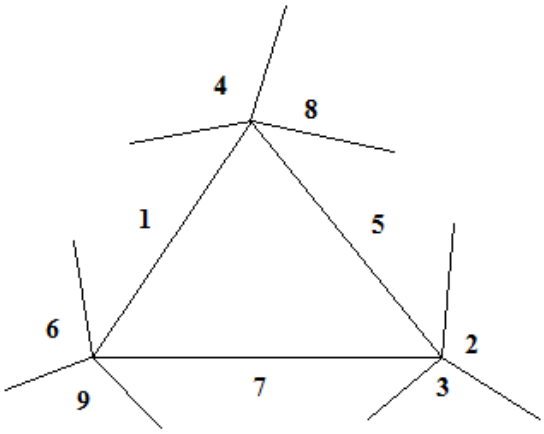

Figure 1: *Close to vertices there are more adjacent triangles*

Yet another problem with this method (based on ID-list storage of adjacent triangles) is related to general meshes, where a triangle may be adjacent to more than 3 triangles: such cases can occur when an edge is shared among more than 2 triangles, so triangles may have an arbitrary number of adjacent triangles and the adjacency information of each triangle needs to be laid out accordingly. Consider the case depicted in Figure 2. The center axis of the a cylinder is shared by many triangles. Each of the side triangles will have all the other side triangles as neighbors but they need to be tested along the edge only. For many pixels inside the triangle such tests will reduce the overall performance if done in a single pass. One way to increase the performance is to skip the edges in a first pass and then perform the edge-checks in a separate pass, but this will increase the complexity of implementation and still may not increase the performance.

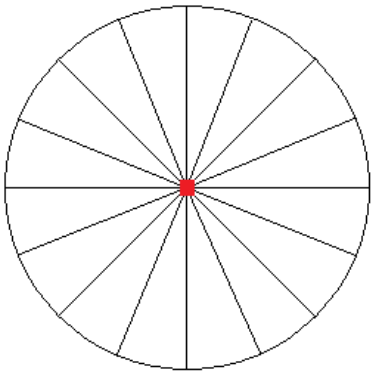

Figure 2: *Top-view of a cylinder with its center axis shared as an edge among many triangles: the red spot is the projection of the center-axis onto the top-view-plane. This axis is shared by all side triangles. As the number of side triangles increase, adjacency list of each side triangle will grow, causing severe degradation in performance in GPU silhouette detection algorithm based on ID-list storage. However, the performance of the GPU-method based on vertex-storage will remain the same.*

4.2. Vertex-storage method

Another possibility is to look for triangle-adjacency by checking the vertices of every triangle directly in the fragment-shader itself. This has the advantage of not having to perform a preprocessing step at the expense of repeated multiple vertex checks. To this end we need to get the 3-vertices for every triangle and check if one of the vertices of the current triangle is the same as one of the vertices of the adjacent triangles. As soon as we identify a triangle that does not share even a single vertex with the current triangle, we can safely state that the pixel is on the silhouette.

This method seems to be an inefficient method, since in the worst case we need to check 4 different neighboring triangles, which can all be neighbors to the current triangle: that is we will have $4 \times 3 = 12$ vertices and these need to

be checked against 3 vertices of the current triangle resulting $12 \times 3 = 36$ vertex-vertex comparisons which are vector-difference and dot product operations essentially. To extract all the necessary information we need to perform $5 \times 3 = 15$ texture access operations to extract 3 vertices for all 5 triangles. So this method seems to be inefficient. But remember: these worst cases tend to occur along the edges of the model. If the object consists of very big but small number of triangles, this method will probably work faster. But at this point we have to keep in mind that if one thread in the fragment shader takes a lengthy execution branch then the others will take that lengthy path as well, even if they do not satisfy the conditions for that branch. The clear advantages of this method are

- No preprocessing steps of any are required
- No special care needs to be taken for vertices
- All types of meshes are supported with no variance in performance
- Data size and thus its layout is fixed: animations on GPU can directly output their results on GPU (this will be explained in the next section)

Due to the advantages listed above we chose to implement this method. Below is the pseudo-code of the fragment shader to accomplish the silhouette rendering. *vTriangleID* is the ID of the fragment of the current triangle being rasterized, while *fCoord* is the coordinates of the fragment. The array *triangles* holds the IDs of the adjacent triangles. *GetTriInd* reads the triangle-IDs at the offset locations from the ID-texture while *GetVertices* reads the 3 vertices of the triangle whose ID is given from the vertex-data texture.

```
int numTriangles=0, triangles[4];
for( int i=0; i<4; i++ ) {
    index = GetTriInd(fCoord + offsets[i]);
    if( index != vTriangleID )
     triangles[numTriangles++] = index;
}

GetVertices(vTriangleID, rv);
for( int i=0; i<numTriangles; i++ ) {
 bool isNeighbor=false;
  GetVertices( triangles[i], cv);
  for( int j=0; !isNeighbor && j<3; j++ )
   for( int k=0; !isNeighbor && k<3; k++ )
    if( rv[j] == cv[k] ) isNeighbor = true;

  if( !isNeighbor ) {
   gl_FragColor = RED; // is silhouette
   return;
  }
} gl_FragColor = WHITE; // not a silhouette
```

4.3. Loading the triangle information to the GPU

Information related to triangles is one of the following

1 List of adjacent triangles to every vertex (ID-list storage)
2 Vertex-coordinates of every triangle (vertex-storage)

Choice for packing the second type information is easier compared to the first one since for each triangle we have 3 vertices and we do not have to worry about cases where a triangle is adjacent to more than 3 triangles, so that number of data per triangle to be stored is fixed. Each of the vertices can be packed as a single pixel having three single-precision floating components (RGB). In such a case each triangle needs 3 pixels to hold coordinates of its 3 vertices. These vertices can be laid out in an arbitrarily in column-major or row-major order. We decided to load them in column-major order. This choice should have a minor impact on the overall performance.

Choice for packing the first type information is more cumbersome: since *in case of general meshes* a triangle could be adjacent to an arbitrary number of triangles, packing this data is a little bit more involved. We could partition a whole texture and place the adjacency information in places corresponding slots. In such a case the size of each slot would depend on the maximum number of triangles shared by the corresponding vertex. But this would be very inefficient if one triangle has say 100 adjacent triangles (Fig 2) and all the remaining ones have say 3 triangles. In such a case the slot size for every vertex would consist of 100 locations, but all the other slots would waste a lot of space. So the data would become sparse, memory consumption would be high and texture caching would be ineffective since lots of the cached spaces would not contain any useful data at all. To avoid this we can encode the adjacency data in two separate textures: one containing all lists appended successively, and another texture holding pointer-table specifying the start-location and length of every list in the densely packed data-texture. In other words we use a pointer indirection for every triangle. Thus at the expense of giving up the array-like random access vertex-data read, we can pack the vertex adjacency data much more compactly using two separate textures: one for holding the data, and one for pointer indirection. In addition to the preprocessing stage, this has the obvious disadvantage that we have to pack the data densely and construct a pointer table at the same time: this is difficult to implement on the GPU using shader programming model, since it does not allow controlled scatter operations, nor does it allow a read/write access to a global variable for keeping track of the current position of the data written to the global array holding the densely packed data (the global variable is used for constructing the pointer table when packing the adjacency data).

4.4. Deformable Objects

For deformable objects with constant topology, the vertex coordinates have to be updated at every animation step and vertex-data in the GPU needs to be updated. This update can be done depending on where the animation is performed. If the animation is performed on the CPU, then the new vertex locations need to be reloaded to the GPU at every step. This

can reduce the performance considerably if animation rate is high and amount of data to be transferred to the GPU is high. An alternative would be to perform the animation on the GPU and render the new vertex-locations directly to the texture. This rendering processes is simple for the texture storing the 3 vertex-coordinates for each triangle, because we know all vertex location in advance. For the textures storing the adjacency information, we do not need to do anything since the topology is fixed.

For ID-list storage method: if topology of the deforming object changes then updating the adjacency information on the GPU would be very difficult because the adjacency-list of every vertex would change and rendering this changing information would require a controlled scattering operation. This scattering operation could be performed via geometry-shaders. But the limited number of primitives of the geometry shader step combined with the difficulties of coordinating the outputs of different geometry-shader threads running in parallel makes this a hard problem. However, NVIDIA's CUDA could be a solution to this problem, but how this can be done is outside the scope of our work.

In case only vertex-coordinates are stored (second type of texture), then topology change won't be any problem as long as the number of triangles remains fixed. However, if the number of triangles changes, the vertex-coordinate rendering problem becomes a little bit more involved but can still be solved since we do not perform any dense data packing which involves appending individual lists of varying length. The scattering operation can be still be encoded in geometry shader and is independent of the parallel threads because we do not have to combine lists of varying lengths into a single global list. All locations where the coordinates of the vertices are to be written can be predefined since they are *compact* and *fixed*.

The two methods clearly show the difference between performance and ease of implementation for both cases. Following the discussions above, it is easy to see that the second method, where only the 3 vertex locations of each triangle is stored seems to be most flexible and easy way to encode the information for rendering the silhouettes. That is why we chose this method over the other method to render the silhouettes.

5. Tests and Results

We implemented our algorithm in GLSL using OpenGL. As an example we performed a simple test, where an object is assumed to be deformable and all of the preprocessing steps are performed on the CPU and the extracted silhouette is drawn as lines. This method is compared to our method, where we loaded the triangle data to the GPU at every frame and compared the performance of both methods. The simulations are performed on a dual-core computer with 3 GHz processor (single-core implementation) and NVidia GTX-295 GPU.

For the teapot model with 6K, 25K and 61K triangles and Stanford Dragon models (up to and including 200K triangles) our method rendered at an average rate of about 21 FPS at 1400×900 resolution although there is a huge amount data transferred from CPU to the GPU. The same frame rate observed with different number of triangles can be explained with many texture-accesses and texture-caching, hence the method is bound by GPU memory access speed. For the dragon model with 870K triangles it dropped to 6 FPS. A snapshot of the silhouettes extracted from a teapot model (61K triangles) using our method is given in Fig. 3

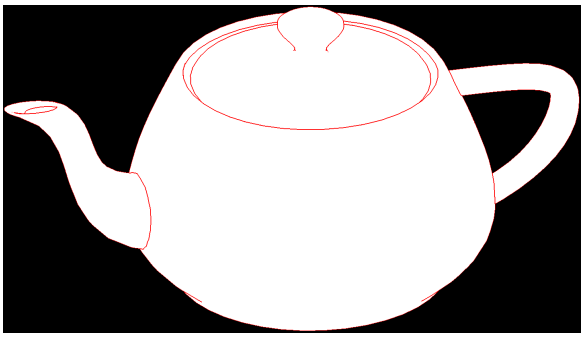

Figure 3: *Teapot model (61K).*

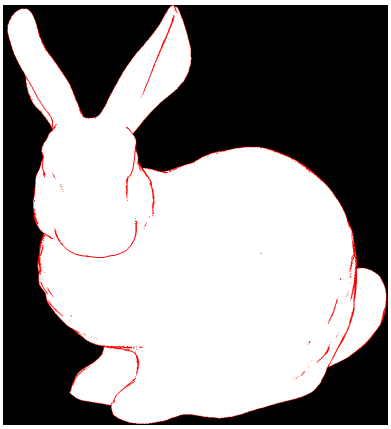

Figure 4: *Bunny model (70K).*

We also implemented the conventional CPU-based method where the objects are assumed to be deformable and topology-changing (worst-case), so at every rendering step the whole object is reprocessed by performing (1) vertex-welding, (2) making its normals consistent and (3) extracting the silhouette with the brute-force method. We did not implement a hierarchy-based method for silhouette-extraction since the hierarchy construction step itself would be more expensive than the brute-force method. The CPU-results are summarized in Table 1.

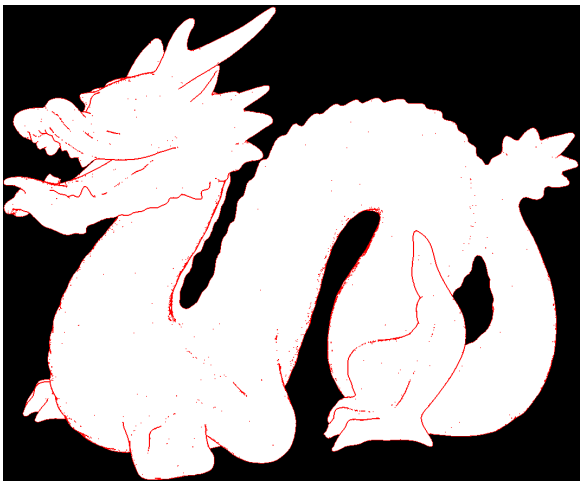

Figure 5: *Dragon model (200K).*

Model	GPU-FPS	CPU-FPS	CPU-Preprocess
Bunny-70K	22	1.9	474.7 ms
City-19K	29	4.8	182.8 ms
Dragon-200K	23	0.7	1454.0 ms
Kitchen-72K	22	2.0	434.8 ms
Teapot-6K	21	8.6	108.5 ms
Teapot-25K	21	4.8	189.2 ms
Teapot-61K	21	2.6	345.0 ms

Table 1: *Performance of extracting silhouettes with reprocessing at every rendering step on CPU.*

6. Conclusion and Future Work

The proposed method is a pure GPU-based silhouette-rendering method: it does not require any pre-processing steps and does not need any accelerated extraction step every time view-point or geometry is changed. For static scenes and deformable objects with non-varying topology a variation of this method can be used to accelerate the effectiveness of this GPU-based silhouette-rendering method. But care must be taken to encode triangle-adjacency information. For this we used two textures: one for pointer-indirection and the other for storing the adjacency data in a dense format. For deformable objects with varying topology, the method using the texture storing the vertex-coordinates for every triangle is more appropriate since no complex appending for dense data packing needs to be done. In case animation is performed on the GPU, it is more feasible to output the new vertex locations directly to the texture due to bandwidth limitations between GPU and CPU.

Comparing the performance of the CPU-based method we can see the effectiveness of the proposed method. The advantage of this method over any Z-buffer based method is that it does not require any threshold value to compare against the difference between depth-values of the neighboring pixels which is difficult to adjust in perspective projection. Therefore our method is robust compared to Z-buffer methods.

As part of our future work we plan to optimize this method for special cases (constant topology and/or geometry) as well as use CUDA to construct the complementary data structures.

Acknowledgements

This work is developed at and supported by Adeko Technologies and also by Turkish State Planning Org. (DPT) TAM Project no. 2007K120610.

References

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. 2

[CM02] CARD D., MITCHELL J. L.: Non-photorealistic rendering with pixel and vertex shaders. In *In Direct3D ShaderX, Wordware* (2002), Wordware Publishing, Inc, pp. 319–333. 2

[Dec96] DECAUDIN P.: *Cartoon-Looking Rendering of 3D-Scenes*. Tech. Rep. 2919, INRIA Rocquencourt, June 1996. 2

[Her99] HERTZMANN A.: Introduction to 3d non-photorealistic rendering: Silhouettes and outlines. In *Non-Photorealistic Rendering, SIGGRAPH Course Notes* (1999). 2

[IFH*03] ISENBERG T., FREUDENBERG B., HALPER N., SCHLECHTWEG S., STROTHOTTE T.: A developer's guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications 23* (2003), 28–37. 2

[JC01] JOHNSON D. E., COHEN E.: Spatialized normal cone hierarchies. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM, pp. 129–134. 3

[SGG*00] SANDER P. V., GU X., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *Siggraph 2000, Computer Graphics Proceedings* (2000), Akeley K., (Ed.), ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 327–334. 3

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 197–206. 2

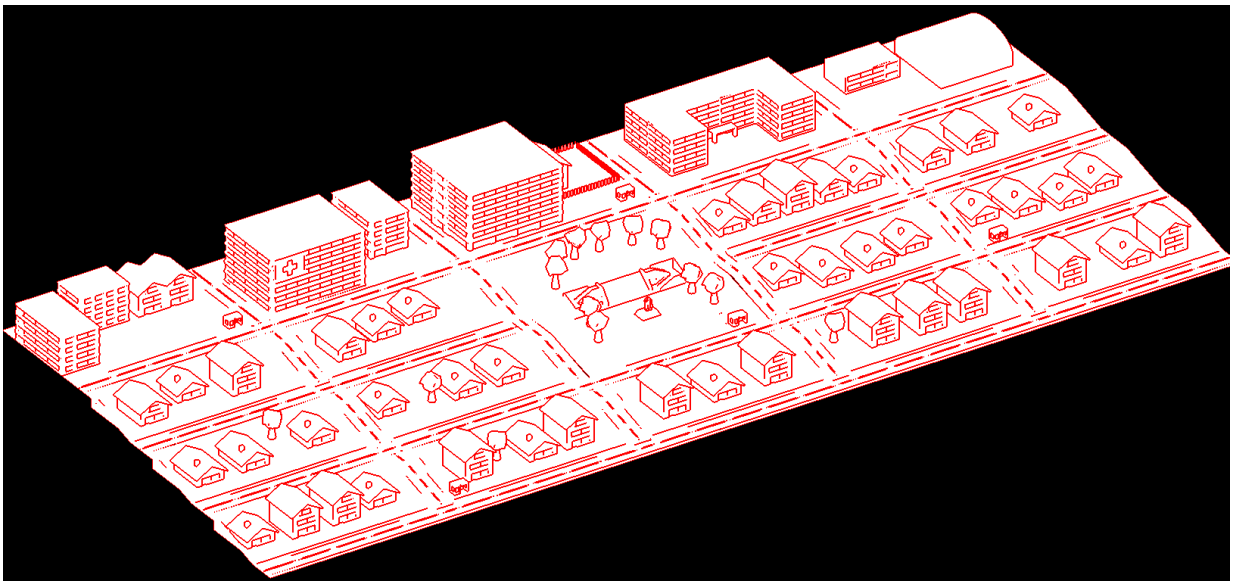

Figure 6: *City model (19K).*

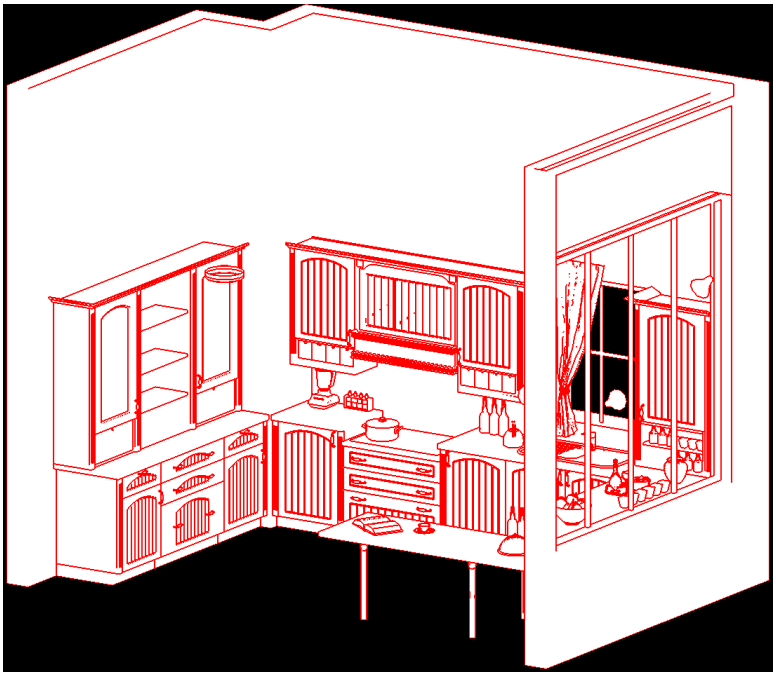

Figure 7: *Kitchen model (72K).*