

Performance comparison of CPU and GPU silhouette extraction in a shadow volume algorithm

Shi Yazheng

January 27, 2006

Master's Thesis in Computing Science, 10 credits

Supervisor at CS-UmU: Anders Backman

Examiner: Per Lindström

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

Abstract

With the recent development of graphic hardware, more algorithms tend to migrate the general computation from the Center Processing Unit (CPU) into the Graphic Processing Unit (GPU). A GPU silhouette-extracting algorithm was published two years ago but is still not widely used. This paper implements the shadow volume algorithm with both the CPU and the GPU silhouette extraction methods to explore the performance of these two methods in practice. An improved GPU silhouette extracting method is also introduced.

Contents

1	Introduction	1
2	Problem Description	3
2.1	Problem Statement	3
2.2	Methods	3
2.3	Previous work	4
3	Algorithms of silhouette extracting	5
3.1	Definition of a Silhouette	5
3.2	Traditional CPU silhouette algorithm	6
3.3	GPU silhouette algorithm	7
3.3.1	Graphic hardware pipeline	7
3.3.2	S. Brabec and H. Seidel's algorithm	8
3.3.3	Improved GPU silhouette algorithm	9
4	Accomplishment	13
4.1	Shadow volume overview	13
4.2	Shadow volume with CPU silhouette extraction	14
4.2.1	Implementation overview	14
4.2.2	Class description	15
4.2.3	Work flow	16
4.3	Shadow volume with GPU silhouette extraction	16
4.3.1	Implementation overview	16
4.3.2	Class description	17
4.3.3	Work flow	17
5	Results	19
5.1	Evaluation overview	19
5.2	Evaluation results	19

6	Conclusions	23
6.1	Restrictions	23
6.2	Limitations and future work	23
7	Acknowledgements	25
	References	27

List of Figures

3.1	Definition of a silhouette edge.	5
3.2	GPU working pipeline	7
3.3	GPU silhouette - first step.	9
3.4	GPU silhouette - second step.	10
3.5	Improved GPU silhouette	11
4.1	Shadow volume algorithm	13
4.2	The tree structure of scene	14
4.3	Class diagram of CPU shadow	15
4.4	Structure of the ShadowVolumeGPU	17
4.5	Class diagram of GPU shadow	18
5.1	Scalability when triangles increase in CPU silhouette	20
5.2	Scalability when triangles increase in GPU silhouette	20
5.3	Scalability when lights increase in CPU silhouette	21
5.4	Scalability when lights increase in GPU silhouette	21

Chapter 1

Introduction

Silhouette extracting techniques play a pivotal role in several areas of Computer Graphics, including non-photorealistic rendering(NPR), shadow volume computation [7] and silhouette clipping [13]. With the improvement of graphic hardware(GPU), some new silhouette extracting algorithms that depend on the programmable Graphic hardwares have been proposed. Compared with the traditional methods that are based on the CPU, the GPU method surely have some attractive features, but it is still not widely used in the newest games like DOOM3. This thesis project will explore the pros and cons of the two categories of method in practice. An improved GPU silhouette approach will also be introduced.

The following content of this paper is divided into six chapters. The second chapter describes the problem statement and some previous work. The third chapter describes algorithms related to this paper. The fourth chapter describes how the implementation was accomplished. The fifth chapter describes the evaluation results of the two approaches to silhouette extraction. The sixth is a summary and conclusions.

Chapter 2

Problem Description

2.1 Problem Statement

Silhouette extracting methods can roughly be divided into two main categories - methods based on the CPU and methods based on the GPU. Several methods that are based on the CPU have been presented. A good survey of these methods can be found in the paper "Silhouette Algorithms" [5]. All silhouette extracting method that are based on the CPU have a common issue that generating the necessary silhouette information can put a heavy load on the CPU as reported in the paper [1]. With the improvement of programable graphics hardware, a silhouette extracting method that is based on the GPU was proposed by the paper "Shadow volumes on programmable graphics hardware" [1]. At first glance, migrating the expensive computation into powerful GPU can save CPU resource, but it is still not widely used. The purpose of this thesis is to explore the performance and scalability of these two approaches of silhouette extracting in shadow volume computation.

2.2 Methods

As described in the paper "Real-time shadows" [6], shadows are important elements to provide the user with visual cues about object placement in creating a realistic image. Stencil shadow volume algorithm is presented by Heidmann in 1991 [7], which is based on Crow's shadow volumes [3]. Shadow volume algorithm can cast shadows onto arbitrary objects by clever use of the stencil buffer. It is one of the most popular real-time shadow creating method. Silhouette extraction is the key portion of the shadow volume computation. The performance of the shadow volume computation is decided by the performance of the silhouette extraction. The algorithm of shadow volume will be briefly introduced in chapter 4.1.

In the implementation of the CPU silhouette extraction we choose the brute force method even if there exists optimized silhouette extracting methods [5]. The reasons for this are:

1. Most optimized silhouette extracting algorithms that are based on the CPU require specific data structure [8] or hierarchy structure of the model. Our implementation is based on OpenSceneGraph(OSG)[12] and this special structure is not available in the normal OSG model.

2. Brute force silhouette extraction is easy to implement and runs nearly as fast as more complex methods for a scene under 10,000 polygons[5].
3. The GPU silhouette extracting method is a brute force method.

2.3 Previous work

Previous silhouette extracting methods are mainly operating on the CPU. The brute force method iterates over each edge to test whether or not it is a silhouette, which is the basic silhouette algorithm and will be introduced in the next chapter in detail. A probabilistic method was published by Markosian etc.[11], which uses inter-frame coherence of silhouette edges. This method can achieve high performance but may skip some silhouette edges that should be detected. Buchanan and Sousa presented an edge buffer method, which iterates over all triangles instead of edges [2]. In practice the edge buffer algorithm actually runs slightly slower than the method that iterates over the edges[5]. Some methods require a progressive mesh [9] or a tree structure of triangles, a detail description of them can be found in the paper "Silhouette algorithms" [5].

The paper "Shadow volumes on programmable graphics hardware" [1] presented a method that implement the silhouette extracting on programmable graphics hardware. This method use float texture store all vertices. The edge and its adjacent information were transfered into the GPU as a vertex. The silhouette extracting computation was performed in the fragment shader. Next chapter will detailedly introduce this approach and an alternative approach which is used in the implementation.

Chapter 3

Algorithms of silhouette extracting

3.1 Definition of a Silhouette

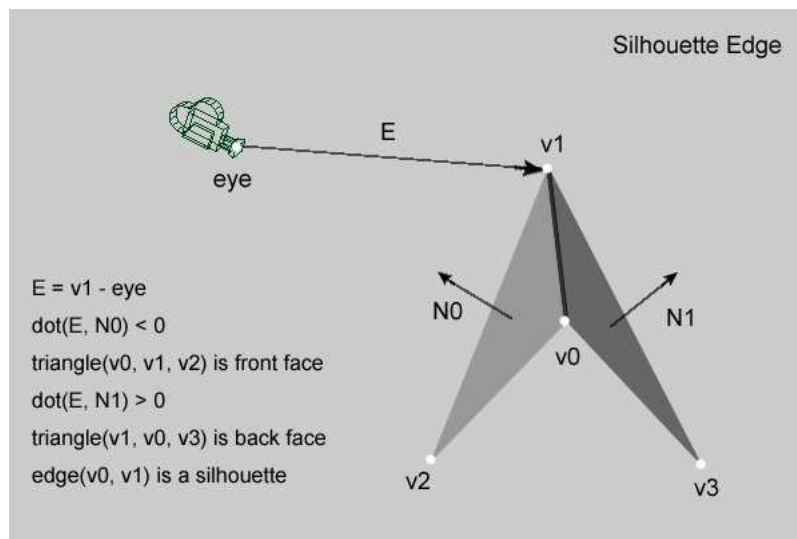


Figure 3.1: Definition of a silhouette edge.

The silhouettes in this paper specifically indicate the object space silhouettes for well-defined polygonal models. A polygonal model is well-defined if the model is only composed by triangles and each edge has two and only two adjacent triangles.

Loosely speaking, a silhouette edge is an edge where a front facing triangle meets a back facing triangle. As shown in the figure 3.1: triangle($v0, v1, v2$) meets triangle($v0, v3, v1$) at edge($v0, v1$). We can see that triangle($v0, v1, v2$) is a front facing triangle and triangle($v0, v3, v1$) is a back facing triangle relative to the eye point, so edge($v0, v1$) is a silhouette edge.

The algorithm to compute if the triangle is a front face or not is:

Given an eye point and a triangle(v_0, v_1, v_2).

1. Compute the normal of the triangle(v_0, v_1, v_2).

$$N = \text{cross}(v_1 - v_0, v_2 - v_0)$$
2. Compute the eye vector.

$$E = v_0 - \text{eye}$$
3. If $\text{dot}(E, N) > 0$
 triangle(v_0, v_1, v_2) is a front face
 Else
 triangle(v_0, v_1, v_2) is a back face

3.2 Traditional CPU silhouette algorithm

There are some basal requirements for the model format. The model should at least have a vertex list, edge list and triangle normal list to provide the adjacent information. If the mesh of the model may changed, for example in skeleton animation, the normal of the triangle will be changed between frames. The triangle list is also required so that the normal of the triangles can be computed in each frame.

The traditional silhouette-extracting algorithm is straight forward. In summary, the algorithm for a single frame runs through the following two steps.

1. For each triangle, compute if the triangle is a front face or not. The algorithm is introduced in the chapter 3.1.
 2. For each edge, if one adjacent triangle is front face and another is back face, mark this edge as a silhouette edge.
- The pseudocode in c++ style:

```

struct geometry
{
    struct triangle
    {
        int vertex[3];
    };

    struct edge
    {
        int triangle[2];
    };

    std::vector<triangle> triangles;
    std::vector<edge> edges;
    std::vector<Vec3> vertices;
};

geometry geo;
Vec3 eyePosition;
// Suppose geo and eyePosition have been initlized.
// for each triangle, judge if it is a front face.
bool *isFrontFace = new bool[geo.triangles.size()];
for (int i=0; i < geo.triangles.size(); i++)
{
    Vec3 v0 = geo.triangles[i].vertex[0];
    Vec3 v1 = geo.triangles[i].vertex[1];
    Vec3 v2 = geo.triangles[i].vertex[2];

    Vec3 E = v0 - eye;
    Vec3 N = cross(v2-v1, v1-v0);
}

```

```

if (E * N > 0) // dot(E, N)
{
    isFrontFace[i] = true;
}
else
{
    isFrontFace[i] = false;
}
}

// for each edge, judge if it is a silhouette
bool *isSilhouette = new bool[geo.edges.size()];
for (int i=0; i < geo.edges.size(); i++)
{
    int tri0 = geo.edges[i].triangle[0];
    int tri1 = geo.edges[i].triangle[1];
    if ((isFrontFace[tri0] && !isFrontFace[tri1]) || (!isFrontFace[tri0] && isFrontFace[tri1]))
    {
        isSilhouette[i] = true;
    }
    else
    {
        isSilhouette[i] = false;
    }
}
}

```

3.3 GPU silhouette algorithm

3.3.1 Graphic hardware pipeline

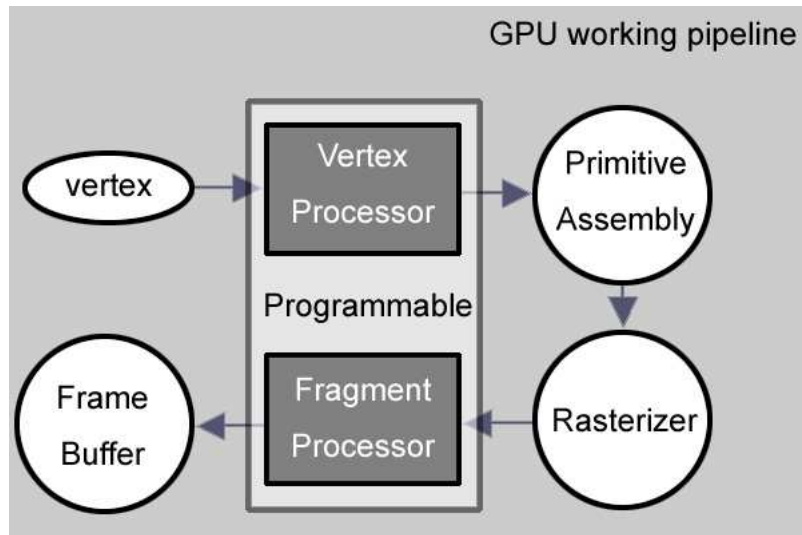


Figure 3.2: GPU working pipeline

Some center parts of this thesis are related to the programmable graphic hardware, a so-called Graphic Processing Unit (GPU). The simplest GPU working pipeline model is shown in figure 3.2. The input of the vertex processor is a set of attribute of the vertex such as position, normal, color and texture coordinates. In the fixed function, some operations performed in vertex processor are:

- vertex position transformation
- per-vertex lighting computation
- texture coordinate generation

The transformed vertices are sent to the primitive assembly stage as well as the connection information. The assembled primitives are rasterized into fragments in the rasterizer stage. The fragment location, texture coordinate and color are computed by interpolation and sent to the fragment processor. The main operations performed in assembly and rasterizer stages are:

- the location of the fragment in the frame buffer.
- the interpolated value of the attributes for each fragment include color, texture coordinates and so on.

In the fragment processor, the texture is applied to the fragment by the texture coordinates. The result is computed and is applied to the frame buffer. The result of this stage is a per fragment color value.

Recent graphic cards(GPU) give the programmer the ability to define the functionality of vertex/fragment processor. Vertex/fragment shader can be written to replace the fixed functionality in the vertex/fragment processor. GPU often have multiple pipelines so that multiple vertices/fragments can be performed at same time. The parallel process results that the programmability of the vertex/fragment processor is at the vertex/fragment level and cannot access the information of adjacent vertices/fragments. The silhouette-extracting algorithm requires the global adjacent information. In the traditional using of the GPU, the silhouette extracting is not suitable for the graphic hardware pipeline. S. Brabec and H. Seidel presented a GPU silhouette method that utilizes the texture-accessing feature of the vertex/fragment processor. By using the texture to store the global information, they broke a new path on the utilization of the GPU.

3.3.2 S. Brabec and H. Seidel's algorithm

S. Brabec and H. Seidel presented a GPU silhouette approach in 2003 [1] which implement the whole computation of shadow volume on programmable graphics hardware. Their contribution is that they migrate the CPU-expensive silhouette computation into the GPU.

In S. Brabec and H. Seidel's approach, the first step is to submit the position of each vertex to the GPU. As shown in the frame B of the figure 3.3, the positions of the vertices should be in the same coordinate system and in one global index system. The position and the index number of each vertex are submitted to GPU. A vertex is rendered as a point and occupies one texel. The index number of the vertex decide the texel position on the texture. The color of the texel is decided by the position of the vertex. The result is a three-channel(RGB) texture in float precision(frame C of figure 3.3). All vertex positions are stored in the texture in sequence. The position of any vertex can be accessed by the index number of the vertex.

The next step is to submit all edges into the GPU. Each edge has two end vertices and two adjacent vertices that belong to the two adjacent triangles. As shown in figure 3.4 vertex e , g is the two end vertices of the edge E , vertex d , f is the two adjacent vertices. As shown in the frame A of figure 3.4, based on the indices of the four vertices, edge E

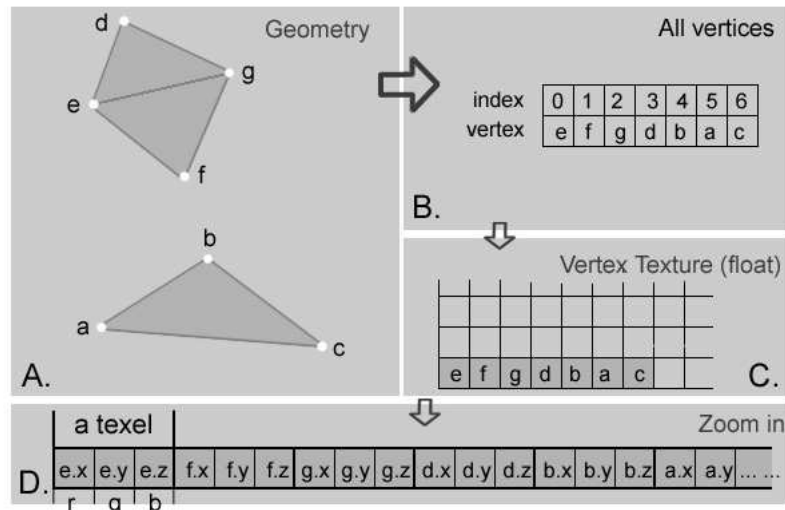


Figure 3.3: GPU silhouette - first step.

is encoded as $E(0,2,3,1)$. In the shader (frame C), the four positions of the vertices can be accessed from the vertex texture (frame B) by the four vertex indices. The normal of the two adjacent triangles can be computed. The position of the light also can be accessed in the shader. Then whether the edge is a silhouette or not can be computed. The result of this step is also a texture (frame D). Each texel contains a silhouette result of an edge with all lights (frame E). The index of the edge decides the position of the texel in the texture.

3.3.3 Improved GPU silhouette algorithm

The S. Brabec and H. Seidel's approach have several limitations.

1. The requirement of a texture with float precision to store all vertices.
2. Recomputation of the normals. Suppose a well-defined polygonal model contains t triangles and $t*3/2$ edges. In S. Brabec and H. Seidel's approach, for each edge two normals have to be computed to judge if the edge is a silhouette. So for all edges, there are $t*3$ normals need to be computed. This results in a recomputation of $t*2$ normals.

We introduce an improved approach that avoids the use of float texture and recomputations of the normals. The first step is to compute the facing property (front face or back face) of all triangles. This step is mainly operated on the GPU. The index of the triangle and the position of the triangle's vertices need to be transferred to the GPU. In our implementation, we use `glColor`, `glNormal` and `glVertex` to submit the three vertices' position to the vertex shader. The index of the triangle is submitted to the shader as an attribute which is used to position the texel in the texture.

```
glBegin(POINTS);
for (int i=0; i

```

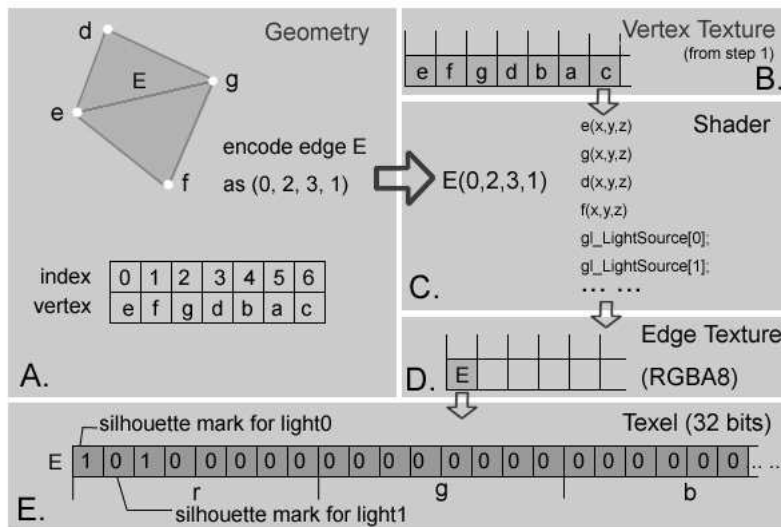


Figure 3.4: GPU silhouette - second step.

```

    glColor3fv(v0.ptr());
    glNormal3fv(v1.ptr());
    glVertex(v2.ptr());
}
glEnd();

```

In the vertex shader, the position of the resulting texel is calculated using the triangle index and the texture width. The positions of the three vertices are transformed to the fragment shader by three varying variables. The normal of the triangle is computed in the fragment shader as the fragment shader is more powerful than vertex shader. The vertex shader in OpenGL shading language:

```

uniform float bufferWidth; // pixel per row
uniform float texelWidth; // 2/screen width
uniform float texelHeight; // 2/screen height
attribute float triangleIndex;

varying vec3 v0;
varying vec3 v1;
varying vec3 v2;

const float left = -1.0;
const float bottom = -1.0;

void main(void)
{
    int bw = (int)bufferWidth;
    int ti = (int)triangleIndex;
    int y = ti / bw;
    int x = ti % bw;
    float xx = left + (x+0.5) * texelWidth;
    float yy = bottom + (y+0.5) * texelHeight;
    gl_Position = vec4(xx, yy, 0.0f, 1.0f);

    v0 = gl_Color.xyz;
    v1 = gl_Normal;
    v2 = gl_Vertex.xyz;
}

```

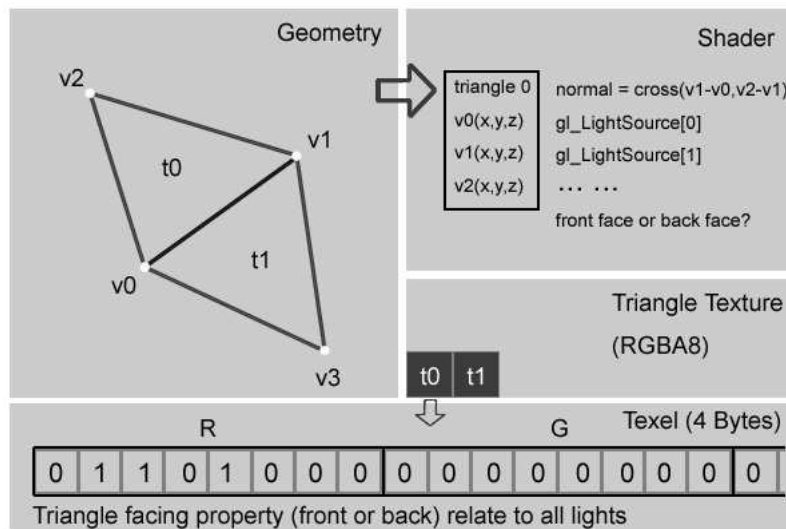


Figure 3.5: Improved GPU silhouette

The fragment shader in OpenGL shading language:

```

uniform mat4 lightNumbers;
varying vec3 v0;
varying vec3 v1;
varying vec3 v2;

void main()
{
    vec3 normal = cross((v1-v0), (v2-v1));
    normalize(normal);
    int r = 0;
    for( int i=0; i < 8; i++) // only support 8 lights
    {
        int x = i    int y = i / 4;
        if (lightNumbers[y][x] == 1.0f)
        {
            vec4 lightPos = gl_LightSource[i].position;
            vec3 lightVector;
            if(lightPos.w > 0.0f) // Point light
            {
                lightVector = v0 - lightPos.xyz;
            }
            else // lightPos.w == 0, Directional light
            {
                lightVector = lightPos.xyz;
            }
            normalize(lightVector);
            if (dot(lightVector, normal) < 0.0f) // Front face
            {
                r += exp2(float(i));
            }
        }
    }
    gl_FragColor = vec4(float(r)/256.0f, 0.0f, 0.0f, 1.0f);
}

```

The next step is to iterate over all edges to get the adjacent facing property (front of back) from the edge texture. If the edge has one front facing triangle and one back

facing triangle, the edge is judged to be a silhouette edge. In our implementation this step is performed on the CPU.

Chapter 4

Accomplishment

4.1 Shadow volume overview

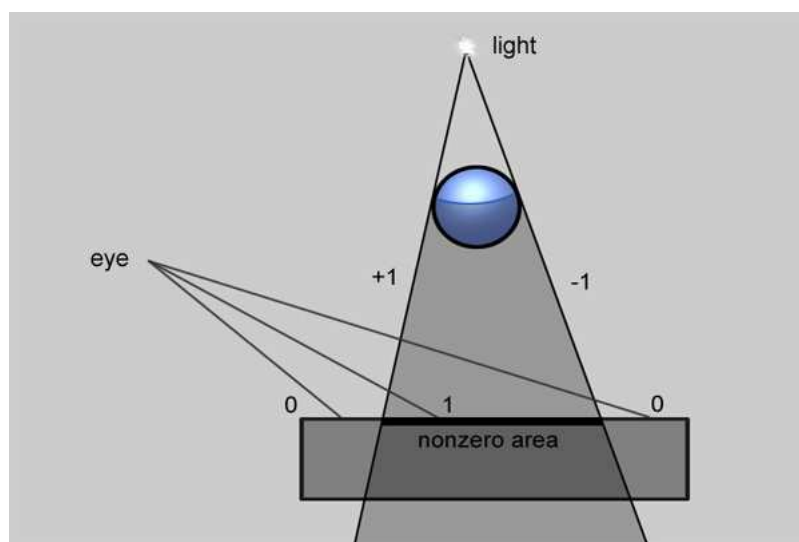


Figure 4.1: Shadow volume algorithm

"Shadows are important elements in creating a realistic image and in providing the user with visual cues about object placement"[6]. Shadow volume algorithm was presented by Heidmann in 1991 [7], which is based on Crow's shadow volumes [3]. Heidmann's algorithm can cast shadows onto arbitrary objects by clever use of the stencil buffer[6]. Shadow volume is one of the most popular real-time shadow generating methods. The detail description of the shadow volume algorithm can be found in [6].

As described in [4], the basic concept of the stencil shadow algorithm is to use the stencil buffer as a masking mechanism to prevent pixels in shadow from being drawn during the rendering pass for a particular light source. This is accomplished by rendering an invisible shadow volume for each shadow-casting object in a scene using stencil operations that leave nonzero values in the stencil buffer wherever light is blocked. Once

the stencil buffer has been filled with the appropriate mask, a lighting pass only illuminates pixels where the value in the stencil buffer is zero. Shadow volume algorithm have two alternative techniques, z-pass and z-fail. z-pass technique suffers an inverse shadow problem when the eye point is in the shadow volume. z-fail technique has not the inverse shadow issue but runs slower than the z-pass technique. An introduction of these two techniques can be found in the paper "The Theory of Stencil Shadow Volumes" [10].

A general list of steps to implement stencil shadow volumes would be [10]:

1. Render all the objects using only ambient lighting and any other surface-shading attribute. Rendering should not depend on any particular light source. Make sure depth buffer is written.
2. Starting with a light source, clear the stencil buffer and calculate the silhouette of all the occluders with respect to the light source.
3. Extrude the silhouette away from the light source to a finite or infinite distance to form the shadow volumes.
4. Render the shadow volumes in stencil buffer.
5. Using the updated stencil buffer, do a lighting pass to shade (make it a tone darker) the fragments that corresponds to non-zero stencil values.
6. Repeat step 2 to 5 for all the lights in the scene.

4.2 Shadow volume with CPU silhouette extraction

4.2.1 Implementation overview

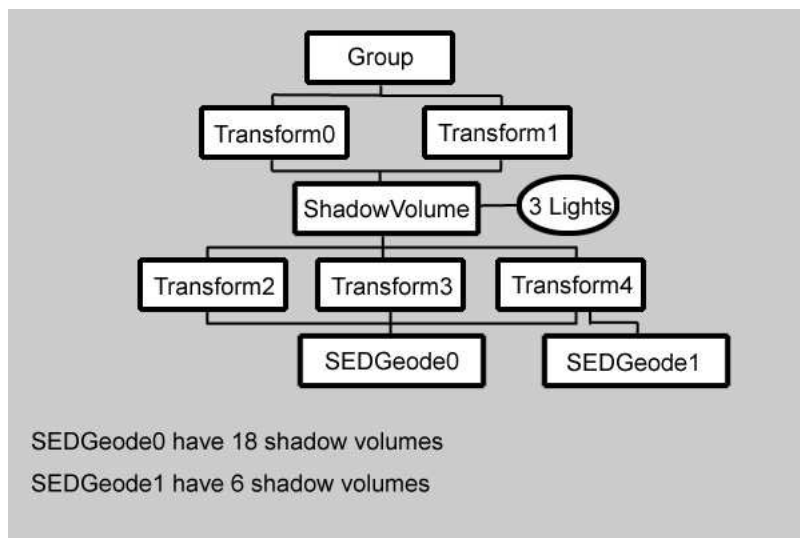


Figure 4.2: The tree structure of scene

Our implementation of the shadow volume algorithm is implemented in the Open Source Scene Graph(OSG). OSG maintain the scene as a tree structure. The leaf nodes of the tree are often the nodes that have drawable geometries. A culling visitor is used to collect drawables by traveling the tree and store these drawables in a render list.

There are several challenges in the implementation shadow volume algorithm in OSG. The first one is that the most common leaf node of the OSG - `osg::Geode` does not contain the edge/triangle adjacent information directly and the adjacent information is necessary for the silhouette extraction. To solve this problem we introduce a new Geode class deriving from `osg::Geode` which is named as `SEDGeode`(the Geode for Silhouette Edge Detection). The `SEDGeode` is a child class of `osg::Geode` so it can work just like `osg::Geode` with additional edge/triangle adjacent information.

The second challenge is how to store the generated shadow volume. There are often multiple shadow volumes need to be created for one `SEDGeode`. For example, in figure 4.2 `SEDGeode0` will be traversed by the culling visitor 6 times. For each traverse, each light, a shadow volume will be generated. That means `SEDGeode0` generates 18 shadow volumes each frame. In our solution, these shadow volumes are stored in `SEDGeode` as a map. The travel number is counted in the `ShadowVolume` and in the `SEDGeode`. The travel number in `ShadowVolume`, the travel number in `SEDGeode` and the light number decide the index used to access the map. For example, for the travel `Transform1` \rightarrow `ShadowVolume` \rightarrow `Transform4` \rightarrow `SEDGeode0` in figure 4.2 the travel number in the `ShadowVolume` is 1. The travel number in the `SEDGeode0` is 2. For the second light, the index to access the shadow volume is 121. For the third light the index is 122.

The third challenge is that the shadow volume algorithm needs several render passes. Some render passes need to render the models, some need to render the shadow volumes. That means the culling visitor needs to collect the original geometry from a `SEDGeode` in some render passes and collect the geometry of the shadow volume from the same `SEDGeode` in some other render passes. In our solution, an informing visitor is used to solve this problem. Before each render pass, the informing visitor travel the scene to tell each `SEDGeode` what will be rendered in this render pass.

4.2.2 Class description

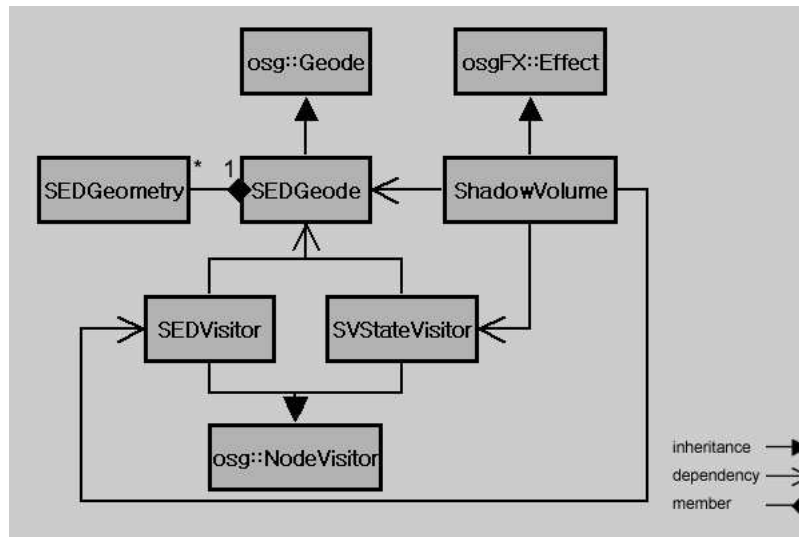


Figure 4.3: Class diagram of CPU shadow

SEDGeode and SEDGeometry

As mentioned in the last section SEDGeode is inherited from `osg::Geode` with additional edge/triangle adjacent information. The adjacent information is offered by the SEDGeometry. `osg::Geode` have a list of `osg::Geometry` which store the vertex and primitive information. In SEDGeode the `osg::Geometry` works as usual and for each `osg::Geometry` a SEDGeometry is generated which shares the vertex list with the `osg::Geometry` and have a list of edges and a list of triangles.

The visitors

Visitors are used to traverse the tree structure scene in OSG. In our implementation, a SEDVisitor is used to notice the SEDGeode to build the shadow volume with several parameters like which lights are used and what method is used(z-pass or z-fail). These parameters are kept in the ShadowVolume and are needed when SEDGeode build the shadow volume. SVStateVisitor is used to notice SEDGeode what to render(shadow volume or geometry) and which shadow volume is picked to be rendered.

ShadowVolume

ShadowVolume is inherited from `osgFX::Effect` which is used to handle multiple render passes with different states of the shadow volume algorithm. All classes that are mentioned above are used in ShadowVolume.

4.2.3 Work flow

The work flow is:

1. When the ShadowVolume node is visited by CullVisitor, SEDVisitor is used to travel the subscene to build the shadow volume for each SEDGeode.
2. After the shadow volume is built, let the CullVisitor traverses the subscene to render the scene with ambient light and depth buffer enabled.
3. SVStateVisitor traverses the subscene to notice all SEDGeode that the render mode is shadow volume.
4. CullVisitor traverses the subscene to render the shadow volume in the stencil buffer.
5. SVStateVisitor traverses the subscene to notice all SEDGeode that the render mode is original geometry.
6. CullVisitor traverses the subscene to render the shadow volume with diffuse light.
7. Clear the stencil buffer.
8. Repeat step 3 to 7 for all the lights.

4.3 Shadow volume with GPU silhouette extraction

4.3.1 Implementation overview

The shadow volume with the GPU silhouette has an entirely different configuration than the CPU one, which is shown in the figure 4.5. The models that cast and receive shadows is kept in the Sub-SceneGraph as a group. All vertices are transformed into eye coordinates. Edges and triangles are collected in the TriangleDrawable. A CameraNode

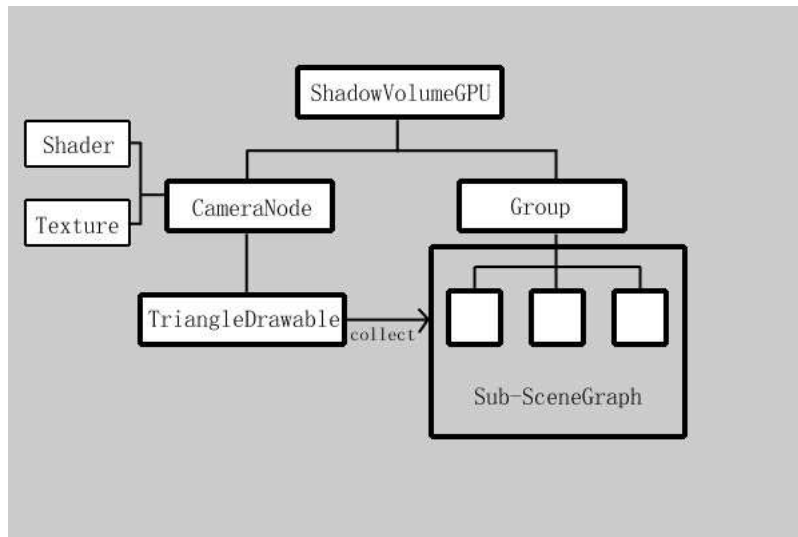


Figure 4.4: Structure of the ShadowVolumeGPU

is attached with shaders and a texture, which is used to draw the facing property (front or back) of all triangles on the texture. The render passes of the shadow volume algorithm is handled by ShadowVolumeGPU class.

4.3.2 Class description

VETCollector and TriangleDrawable

VETCollector is inherited from `osg::NodeVisitor` and is used to travel the SubSceneGraph to collect all vertices edges and triangles. TriangleDrawable is used to submit each triangle as a list of vertex attributes to the GPU.

ShadowVolumeDrawable

ShadowVolumeDrawable is used to generate and render the shadow volumes. ShadowVolumeDrawable can get triangle facing property from the triangle texture. All silhouette edges are extracted in the `drawImplementation` of ShadowVolumeDrawable. Then the silhouette edges are extruded into the geometry of shadow volume and rendered.

ShadowVolumeGPU

ShadowVolumeGPU is inherited from `osgFX::Effect` and is used to handle the different render passes of shadow volume algorithm.

4.3.3 Work flow

The work flow is:

1. When the ShadowVolumeGPU node is visited by CullVisitor, VETCollector is used

Chapter 5

Results

5.1 Evaluation overview

The test results are base on the Intel P4 2800MHz CPU, NVIDIA GeForce 6200 GPU and 1024MB memory with Linux operating system. A scene with 1000 triangles with one light in shadow volume algorithm means about 2200 triangles will be rendered in each frame with z-pass method and about 3200 triangles will be rendered with z-fail method. If the scene has three lights, there will be about 4600 triangles in z-pass method and about 7600 triangles in z-fail method need to be rendered each frame. Suppose t is the number of triangles in the scene , k is the number of triangles that the shadow volume geometry have, $numLights$ is the number of lights. The number of rendered triangles are:

Z-pass method: $TotalTriangles = t + (t + k) * numLights$

Z-fail method: $TotalTriangles = t + (2t + k) * numLights$

5.2 Evaluation results

The figure 5.1 and figure 5.2 show the scalability of these two silhouette methods when the number of triangles are increased. The figure 5.3 and figure 5.4 show the scalability when number of lights are increased. All these figures show that in the shadow volume algorithm, silhouette extracting takes most part of computation time especially in the z-pass shadow volume method. Silhouette extracting is sensitive of the number of triangles, which makes that the shadow volume algorithm have a bad scalability. Compared with the CPU silhouette method, the GPU silhouette method has the same scalability but has much lower performance, which only could get about 50% FPS compared with the CPU one. The reason is that the facing property results are stored on a texture. A per-textel accessing is needed to extract the silhouette. Accessing lots of data from the texture can be a bottleneck, which will slow down the system. As a creative technique the GPU silhouette did show a new approach to let the GPU do some general computation, but it is still not mature. Unless the bottleneck can be eliminated by some new techniques, the GPU silhouette method is not worth to be widely used.

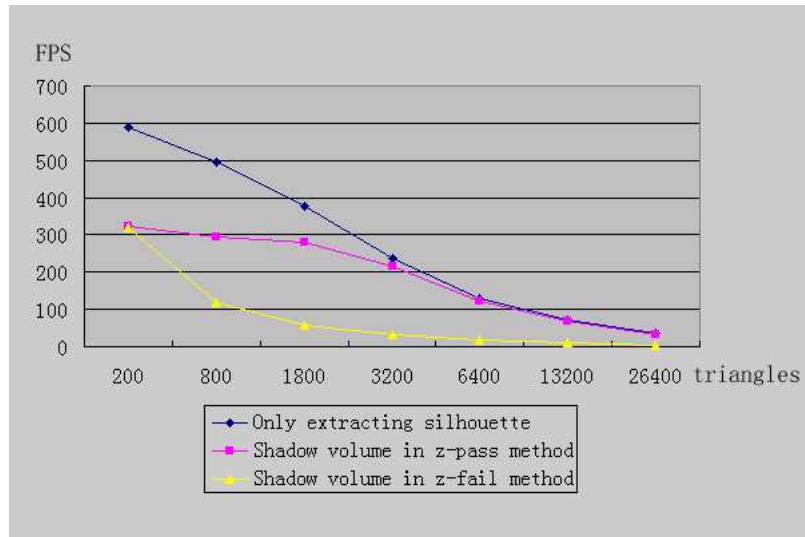


Figure 5.1: Scalability when triangles increase in CPU silhouette

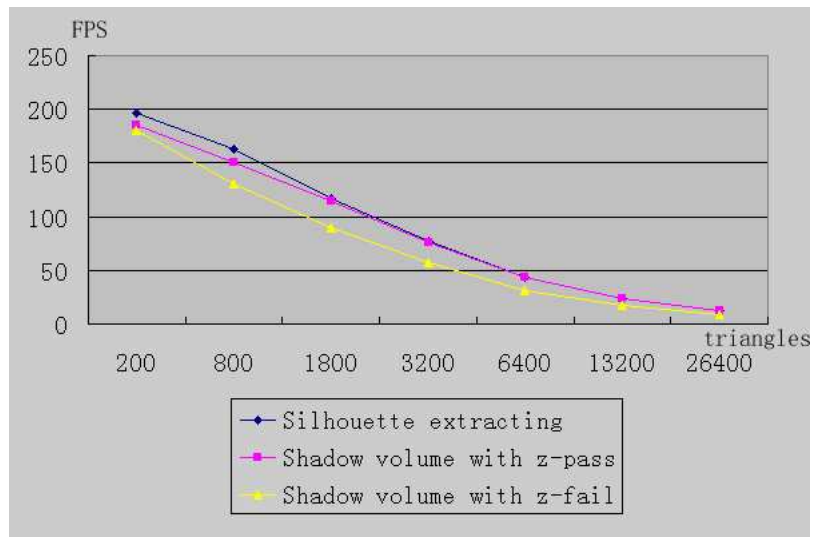


Figure 5.2: Scalability when triangles increase in GPU silhouette

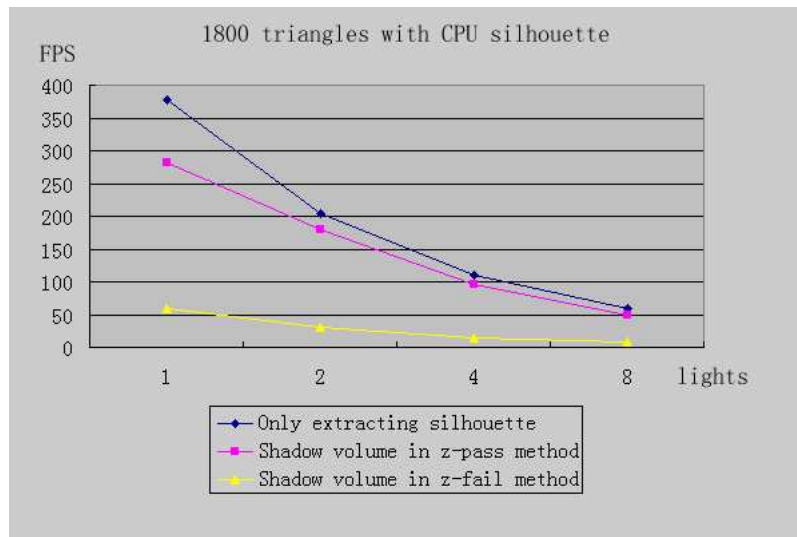


Figure 5.3: Scalability when lights increase in CPU silhouette

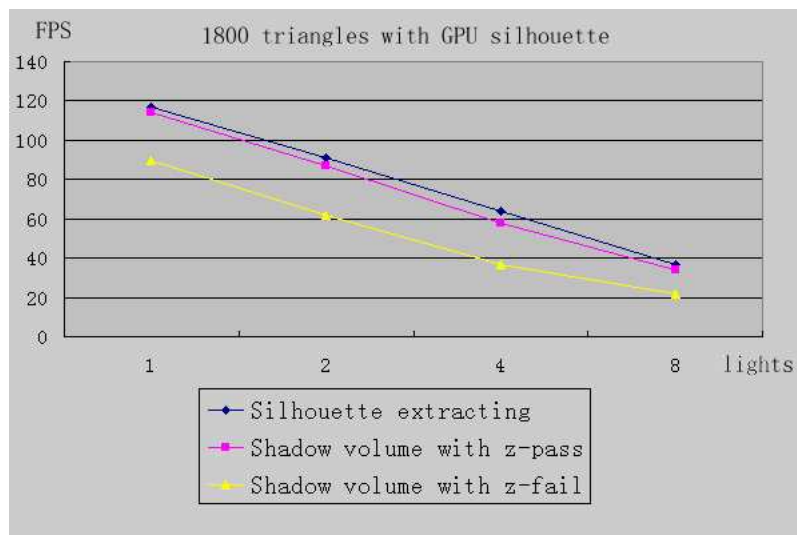


Figure 5.4: Scalability when lights increase in GPU silhouette

Chapter 6

Conclusions

In this paper we have shown how to perform the shadow volume with two kind of silhouette extracting approaches in OSG. An alternative GPU silhouette extracting approach is introduced. We also compared the performance of the two silhouette approaches.

6.1 Restrictions

In chapter 2.3, some other CPU silhouette extracting approaches were introduced. The performance of these approaches is not explored in this paper. This paper focus on the performance of silhouette extracting in shadow volume computation. The performance of GPU silhouette in other use like NPR or silhouette clipping are also not explored. So we can not say that the performance of CPU silhouette extracting and the GPU silhouette extracting are fully and deeply compared in this paper. We only compared them in a general aspect.

6.2 Limitations and future work

In this paper we did not focus on the optimization of the shadow volume algorithm. One suggested future work is to make the OSG shadow volume optimized by adding some optimizing features like infinite view frustums, occluder cullings and choosing z-pass or z-fail automatically. Another suggested future work is to explore the performance of GPU silhouette in other use like NPR etc..

Chapter 7

Acknowledgements

This paper received tremendous amount of help from Mr.Anders Backman in both academic area and programming area. The project leader of OSG Mr.Robert Osfield also gives some advice on the OSG programming.

References

- [1] Stefan Brabec and Hans-Peter Seidel. Shadow volumes on programmable graphics hardware. In Pere Brunet and Dieter W. Fellner, editors, *EUROGRAPHICS 2003 (EUROGRAPHICS-03) : the European Association for Computer Graphics, 24th Annual Conference*, volume 3-22 of *Computer Graphics Forum*, pages 433–440, Granada, Spain, September 2003. Eurographics, Blackwell.
- [2] John W. Buchanan and Mario C. Sousa. The edge buffer: a data structure for easy silhouette rendering. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 39–42, New York, NY, USA, 2000. ACM Press.
- [3] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2):242–248, July 1977.
- [4] Lengyel E. The mechanics of robust stencil shadows, October 2002. http://www.gamasutra.com/features/20021011/lengyel_01.htm.
- [5] Bruce Gooch, Mark Hartner, and Nathan Beddes. Silhouette algorithms. Technical report, SIGGRAPH, Blacksburg, Virginia, 2003. http://pages.cpsc.ucalgary.ca/mario/npr/projects/sigg03/lec2/hand_1.pdf.
- [6] Eric Haines and Tomas Möller. Real-time shadows. In *Game Developers Conference 2001 Proceedings*, March 2001.
- [7] Tim Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991. Silicon Graphics, Inc.
- [8] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.
- [9] D. Kirsanov, P. V. Sander, and S. J. Gortler. Simple silhouettes for complex surfaces. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, 2003.
- [10] Hun Yen Kwoon. The theory of stencil shadow volumes. www.gamedev.net, <http://www.gamedev.net/reference/articles/article1873.asp>.
- [11] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. *Computer Graphics*, 31(Annual Conference Series):415–420, 1997.

- [12] Robert Osfield. openscenegraph. <http://www.openscenegraph.org>.
- [13] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 327–334, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.