HELSINKI UNIVERSITY OF TECHNOLOGY
Telecommunications Software and Multimedia Laboratory
15.4.2002
T-111.500 Seminar on Advanced Rendering Techniques
Spring 2002: jkainula@cc.hut.fi

Stencil Shadow Volumes

Jukka Kainulainen

47942F

Stencil Shadow Volumes

Jan Jukka Kainulainen
HUT
jkainula@cc.hut.fi

Abstract

This paper discusses shadow generation techniques in a 3D-space and especially stencil shadow volumes. Stencil shadow volumes are a popular, but hard to implement, technique to produce credible real-time shadows. Stencil shadows are needed especially when the scene is too dynamic for precalculated shadows. Modern rasterization libraries support this technique through special "stencil buffer".

1 INTRODUCTION

The modern computer graphics has evolved greatly in the last ten years. The visual appearance of a modern computer game is something that was beyond our dreams just a decade ago. The ever growing need for realism and visual perfection sets high demands for both the programmers and the hardware that creates the image.

The means for creating more realistic images are many. One of the hardest fields in realistic image generation is shadow generation. Realistic shadow generation is computationally expensive and is only now beginning to be possible in real-time. This is due to the tremendous increase in the processing power of both the CPUs and the graphics accelerator cards. A modern graphics accelerator is as complex as the CPU and includes an equal amount of transistors, or even more.

As the graphics cards get more and more sophisticated they relieve the CPU from the processing of the lighting and allow it to do additional calculation elsewhere. The complex shadow calculations can also be done on the specific hardware designed for transform and lighting (T&L). The programmable T&L chips can be used to create shadows in real time.

In chapter 2 most of the currently used shadow generation techniques are introduced and briefly discussed. Chapter 3 introduces the shadow volume technique and discusses the problems of the basic implementation and also available solutions. Chapter 4 includes the stencil buffer to the algorithm and discusses the possibility of GPU driven algorithm. Chapter 5 deals with OpenGL specific details while chapter 6 concentrates on DirectX.

2 COMMON SHADOW CREATION TECHNIQUES

There are many shadow generation techniques being used in today's applications and different applications have different needs. For 3D-modelers the image quality and physical correctness is important but for computer games and simulators the need for real time image generation is dominant. Whatever the application, shadows make the produced image more realistic and provide important positional information. What's introduced in this chapter are a couple of the most widely used shadow generation techniques. These techniques are parallel to the idea of shadow volumes but can also be used to complement it.

2.1 Projected Planar Shadows

Projected planar shadows (Blinn, 1988) are probably the simplest shadow generation algorithm still in wide use. It suffices for single objects scenes throwing shadows on a plane. The main idea is to draw a projection of the object on the plane (Figure 1).

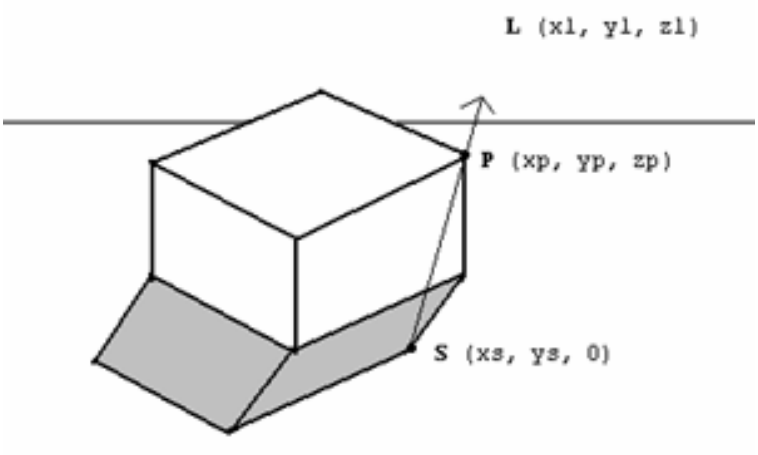


Figure 1: Ground plane shadow

It should be easy to see that the calculation of **P**'s projection **S** is not difficult. **L** denotes the position of the light source throughout this document. The coordinates (x, y and z) denote the position in the Cartesian coordinate system. The Blinn's model does not really suffice anymore due to the fact that modern scenes are often so complex that the calculation of shadows to all possible surfaces this way would be too time consuming.

2.2 Shadow Z-buffer a.k.a. Shadow Mapping

Another quite simple method developed by Williams (1978) is called the shadow Z-buffer. The technique requires a separate Z-buffer for each light source. The algorithm works in two phases:

1. Render the scene using the light source as a view point. This calculates a 'depth-image' of the scene from the light source.

2. Render the scene using a normal Z-buffer algorithm. If a point is visible from the view point, then transform the coordinates also to the light source coordination. Now, if the z-component is bigger than the one already in the Z-buffer there is something between this point and the light and it's in shadow.

This algorithm and its problems, such as aliasing, are further discussed by Watt (1993). It should be noted here that the shadowing is view independent; the position of the viewer doesn't have to be considered when calculating the shadow maps. This encourages the use of precalculation. The visual appearance of different stages is illustrated in Figure 2. The leftmost image shows the resulting image, the middle one the view from the light source and the rightmost the Z-buffer.

Figure 2: Shadow mapping in stages. Figure from Cebenoyan (2001).

2.3 Global Illumination Models

The global illumination models can naturally be used to generate shadows. The first global illumination model was ray tracing, which was first used in computer graphics by Appel (1968). Appel used ray tracing only to solve the famous hidden surface problem and the algorithm has evolved greatly since those days. The basic idea is to trace rays backwards from the eye through every point in the image plane. As the ray intersects an object it is reflected and refracted and thus the algorithm becomes recursive.

The calculation of intersections, reflections and refractions is not trivial for an arbitrary surface. This makes even the basic algorithm quite hard to implement. On the other hand the algorithm can be quite easily used in a distributed environment with multiple processors. This is due to the fact that the rays used for different pixels are independent of each other. Still the algorithm is not suitable for real time image synthesis.

From the theory of heat transfer a method called radiosity was developed by Goral et. al. (1984). It models the diffuse interaction between objects in a closed environment where light sources are considered as objects with self-emittance. Radiosity itself is defined as the energy per unit area leaving a surface per unit time:

$$\text{Radiosity} \times \text{area} = \text{emitted} + \text{reflected energy} \qquad (1)$$

The basic radiosity algorithm is slow and is, again, often used for precalculated scenes due to the fact that the camera position has no effect on radiosity. Radiosity and ray tracing are further discussed by Watt (1993). Also, the visual quality of radiosity lighting can be examined, for example, in the computer game Max Payne.

3 SHADOW VOLUMES – THE THEORY

Currently many of the real time 3D-engines first render a lit scene and afterwards subtract or modulate out luminosity in the areas they recognize as being in the shadow. As an example the above projected planar shadows method works this way. A 3D-engine is basically the mathematical (vector) calculus used to estimate the position and the color of different vertices. Using the Phong illumination model, this leads roughly to the following algorithm:

$$I = I_a k_a + I_i (k_d (\mathbf{L} \bullet \mathbf{N}) + k_s (\mathbf{N} \bullet \mathbf{H})^n) \qquad (2)$$

where I refers to intensity, k to surface attributes, **L** vector to the light source, **N** is the surface normal and **H** is a the vector halfway between **L** and the viewing vector. n is a constant simulating roughness. Figure 3 illustrates this all. Now as the 3D-engine decides that the vertex in hand is in the shadow of some light source it just multiplies I with a constant < 1. It should be obvious that for multiple light sources this is plain wrong.

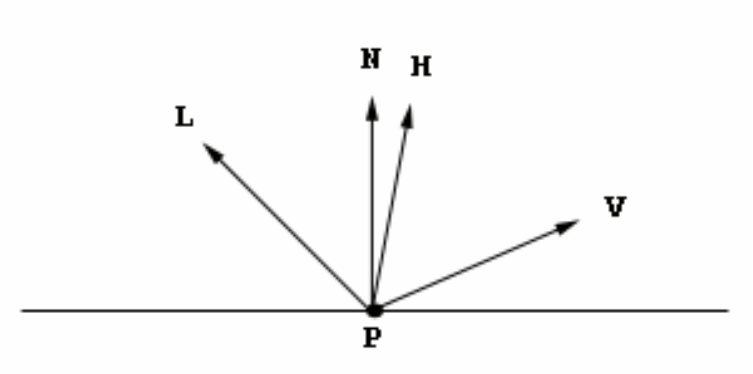


Figure 3: Common illumination model

The above reason is the one that will drive the modern 3D-engines to an approach where shadows are calculated simultaneously with the lighting. That is, start with a black scene and add light contributions for each light source. It is also possible to first render the parts not in shadow and then do a second pass to render the shadowed regions.

3.1 Analytical Way

Crow (1977) discussed the matter of different algorithms for shadow generation and came to a conclusion that all things considered a method called shadow volumes gave the best overall result at that time. What Crow suggested was the creation of shadow volumes enclosed by so called shadow polygons. These shadow polygons are then

included in the surface data when calculating hidden surface removal (HSR). First let's make an important remark about a polygonal object: A so called *contour edge* of an object is an edge which either:

1. Is an extreme of an open polyhedron

2. Separates a front facing and back facing polygon where the surface curves behind itself, thus forming a silhouette for the object

Now given by the planes defined by the light source position and the contour edges of an object we can create the shadow polygons. One polygon is formed by one contour edge, two lines defined by the light source position and the endpoints of the contour edge and the endpoints of the influence of the light (in the direction of the two lines) or by the boundaries of the view volume. Figure 4 illustrates a simple case. The shadow volume consists of view volume boundaries and polygons created by the silhouette edges and the light source position.

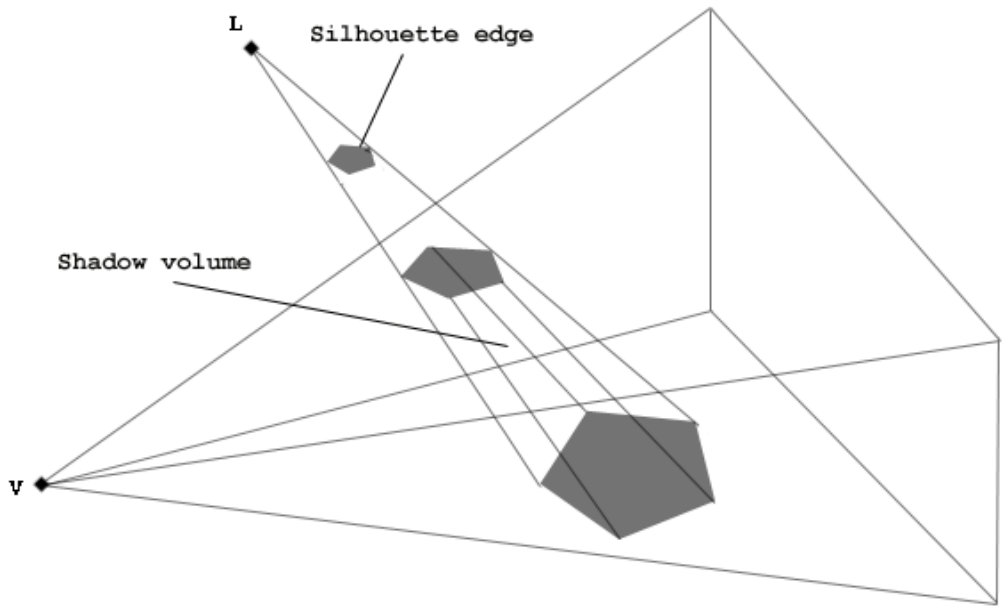


Figure 4: Shadow volume of an open polyhedron

In the figure **L** indicates the place of the light source in hand while **V** is the position of the camera. Camera's view volume is illustrated as a pyramid. The shadow planes caused by the polygon intersect the view volume and thus a volume which is in the shadow of the polygon is formed.

So now we have a closed shadow volume and the polygons that enclose it. What comes next is clever and simple: We just add the enclosing polygon data to our normal polygon data when doing HSR. The shadow polygons are naturally invisible and they are never

drawn to the screen, but they make a crucial contribution to determining whether an arbitrary vertex is in the shadow or not.

When looking from the camera's point of view a front facing shadow polygon puts everything behind it into shadow and a backfacing shadow polygon cancels the effect. If the shadow polygon nearest to the camera is back facing then it must be so that the camera is inside the shadow volume and everything in front of the polygon is in shadow.

If the above algorithm is run on a traditional scan line HSR rasterizer not many modifications need to be made once the calculation of shadow polygons is made. Also, remembering that shadow polygons are invisible we can ignore scan lines including only these polygons. The downside is that the shadow polygons are typically quite large (comparing to those of a single object) and increase the average complexity of a scene.

The following things are needed, extra to traditional polygons and a Z-buffer, to use the shadow volume technique proposed above:

1. Extra information in the polygonal object about the adjacent polygons.

2. Calculation of shadow polygons per light source and per object. Recalculation is needed if something else than the camera moves in the scene.

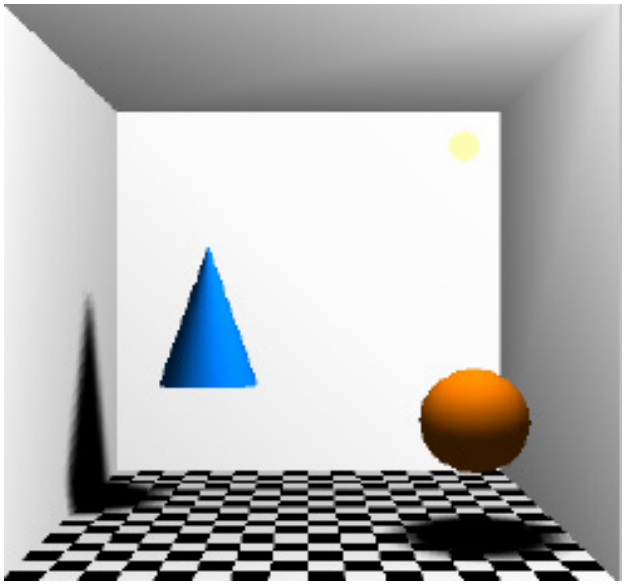
3. Modifications to the inner loop of the rasterizer so that it handles the combination of shadow polygons correctly. This matter is discussed in detail in chapter 4.

3.2 Pseudo Global Illumination

What the shadow volume approach offers is something like pseudo global illumination. The model is not local, since objects can shadow each other and themselves. Also objects with holes can be easily dealt with. But the model is not global since it offers no way to produce effects such as reflection; it is not its purpose.

On the other hand the algorithm offers a way to produce shadows with penumbra. These shadows are often referred to as soft shadows. The way to do this is to shift the place of the light source a bit and create multiple shadow volumes. While rasterizing the number of the volumes the vertex is inside of determines the actual effect of the shadow. The model is a rough approximation of a correct area light source, but results in adequate image quality especially while in motion.

The downside with soft shadows is that it requires a great deal of fill rate from the graphics hardware. A rather old example is shown is figure 5, where the soft shadows produced by altering the position of the light are clearly visible. The light source is in the upper right area of the room.



Firgure 5: Soft shadows produced by moving the light source. Figure from Kilgard (99).

3.3 Problems and Solutions

The basic algorithm works well for simple cases and well defined models. Complicated scenes and some special cases cannot be dealt with without extra effort. For example near plane clipping can cause anomalies in the produced shadow. Philippe Bergeron (1986) discussed the problems further and proposed his own “General” version of the algorithm.

Especially Bergeron’s method solves the problem with nonplanar or concave polygons. These problems are not very interesting since today in almost all cases polygonal models are made of planar and convex polygons. This is due to the fact that all current accelerator cards only render such polygons. Bergeron also solves the problem with penetrating polygons.

However, there is one solution in Bergeron’s method that is of some importance. That is the case with open models. If the model is not closed it might not produce an equal amount of front- and back facing shadow polygons relative to a ray from the viewpoint. Let’s introduce a variable called *depth count*. When we start from the viewpoint every time we cross a front facing shadow polygon we add +1 to it and a back-facing polygons adds -1 and thus cancels the effect. So for a single object and a single light source the depth count should be the same after leaving their influence as it was before we entered the shadow. For open models this might not be the case.

The problem can be solved by introducing a new variable *Nb_C*. If the polygon crossed is generated from a case 2 edge (see chapter 3.1 above) the variable is +2 (-2 if back facing) otherwise it will be +1 (-1). When this value is added to the depth count every time a shadow polygon is crossed the overall result of a single object will be zero. This is just what we needed. Is should be noted here that every open model can be closed quite easily so even this case is not extremely important anymore.

4 STENCIL SHADOW VOLUMES – THE PRACTICE

Now that we're done with theory it's time to get into the actual means of making it work and to the features currently supported by the hardware. The support of the hardware is crucial to speed of the shadow volumes. Even with hardware a general implementation of the shadow volumes is **not** trivial. What's introduced next is an enhancement to the above original shadow volumes algorithm. The enhancement is called stencil shadow volumes.

4.1 Stencil Buffer

Both major APIs, OpenGL and DirectX, nowadays support an additional buffer called the stencil buffer. It so happens that we can use the stencil buffer to model the depth count discussed above. Stencil buffer is a per-pixel test buffer similar to a depth buffer and thus gives us just the additional control to the pipeline we need to implement shadow volumes.

The stencil buffer is used to counting the times we enter or leave the shadow volume. This requires two passes: First pass renders the front faces and increments when depth test passes and the second pass renders back facing polygons and decrements when the test passes. Maybe in the future this can be made with a single pass of a two-sided stencil test proposed by Everitt (2002). Figure 6 illustrates the result of partition into different parts of space according to the stencil buffer. The figure is in 2D to keep things simple.

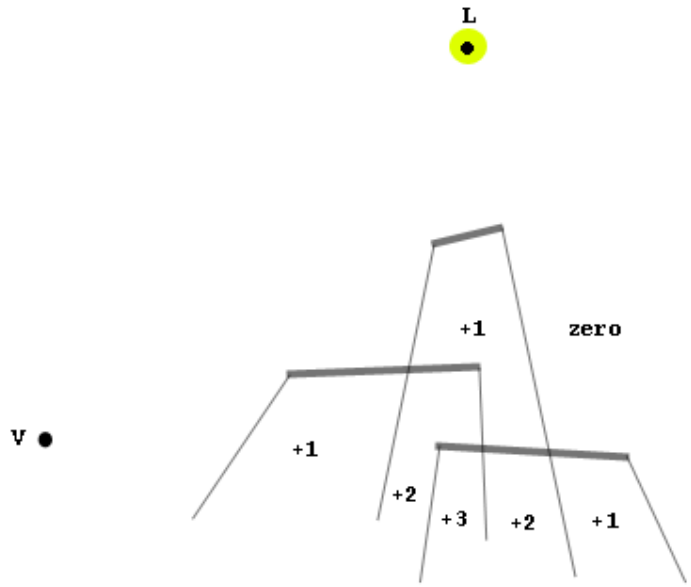


Figure 6: The stencil buffer value for different parts of space.

The algorithm illustrated by the figure 6 is known as stencil shadow volumes algorithm. It differs from Crow's original shadow volumes algorithm in the usage of stencil buffer to partition the space into shadowed and lit parts. Also other enhancement like those of

Bergeron's are often used to speed up and improve the basic algorithm. We'll call the above method of stencil usage the *Zpass* approach to be consistent with Everitt.

4.2 Carmack's Reverse

In the late 90's Carmack (2000) reversed the usage of stencil buffer. He proposed an order in which the stencil value is incremented when the depth test *fails* for a back facing polygon and decremented when the test fails for a front facing one. The resulting values, for example, of the figure 6 would be the same; they'd just have to be calculated starting from the back (right to left in the figure). This is known as the *Zfail* method.

The advantage of the reverse is that the near plane clipping problem is solved. Imagine that the viewpoint in figure 6 would start to move from left to right. At certain point the near plane would start to cut through the first shadow polygon and the traditional algorithm would be in trouble in the lower part of the screen. Carmack's reverse fixes this, but causes a similar problem at the far plane. Fortunately this is not a great issue since shadow volumes usually have a limited extent and the far plane is usually quite far away anyway. It is also possible to avoid the far plane clipping completely, see Everitt (2002).

Carmack's reverse and other solutions are also discussed by Kilgard (2001). We'll probably have to wait and see the visual quality of the method in the Doom III computer game.

4.3 Vertex Shaders

One problem with the stencil shadow volume algorithm is the generation of shadow polygons, which can be quite CPU intensive. Also, it cannot be done in the T&L hardware without a clever hack: We can stretch the already existing vertices of the model away from the light source. This can be done with the modern vertex shaders. The downside is that it only works for well defined models. Figure 7 presents what the stretching is about. What happens is we grab all the back facing polygons and force them apart from the geometry. The space left between is filled with quads and what's inside the thus formed geometry is in the shadow of the light source **L**.

Vertex shaders are an assembly like language controlled part of the modern graphics pipeline. They give programmers back the long awaited run-time control to the pipeline. Vertex shaders are applied to the vertex data before clipping and perspective transformation, while pixel shaders are applied after them. The used instruction set, for example, in DirectX 8 includes the common assembly instructions.

The used registers are 128 bits wide so they each hold four floats (x, y, z, w). There are in general five kinds of registers: input (vxx), output, constant (cxx), temporary (rxx) and address (a0). A good tutorial to vertex shaders can be found from Microsoft's DirectX SDK 8.1. Figure 8 shows the visual quality of the vertex shader approach from ATI's demo "RadeonShader" featuring multiple objects and two light sources. A close inspection of the right side wall shows the correct behavior of the shadows: The shadows of the goblet and the plate are cumulated correctly from the two light sources.

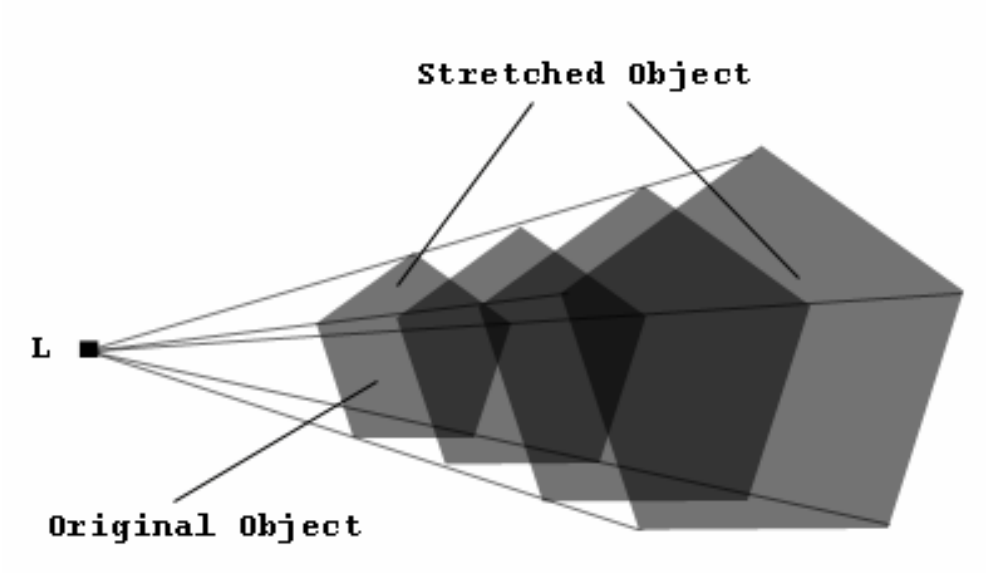


Figure 7: Stretching an object from the light source.

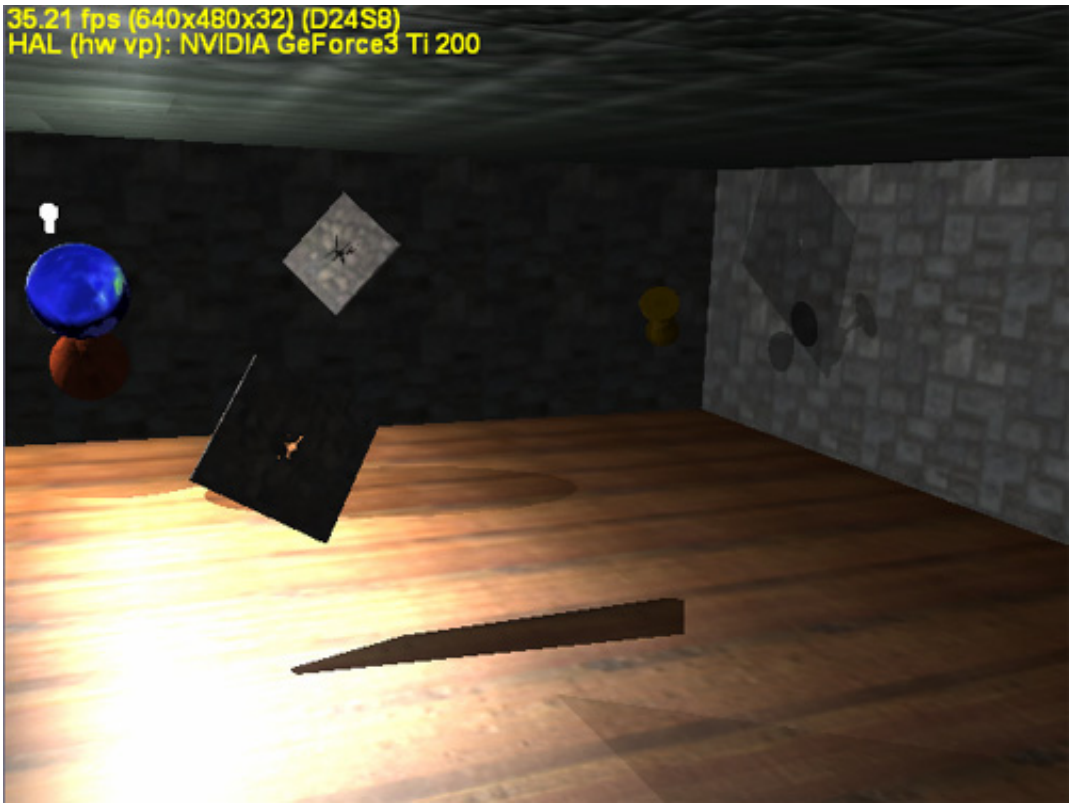


Figure 8: The high-quality result of a GPU driven stencil shadow volume algorithm.

4.4 Remaining Difficulties

There still remain some difficulties with the stencil shadow volumes. First of all the deal with shadow polygons is fill rate extensive. It requires a lot of GPU time to ‘draw’ with

these invisible polygons. And the fact that it has to be done for each light and each object separately adds to the complexity and sets up hopes for simple and well defined objects. On the other hand we can often omit the shadow generation for objects far enough from the camera because the shadows would not make a great difference there.

There is one more problem which is of more important value: Light sources very close to model (polygon) can cause very large shadow volumes to be formed. Clipping of these large volumes can be difficult and even when properly clipped the visual appearance might not be what we hoped for; the shadow will probably be huge.

5 THE OPENGL WAY

OpenGL is one of the two major APIs used today. Below, there is a brief introduction to the use of the stencil buffer with OpenGL in Microsoft Windows environment. This is done so that the paper would not rely only on theory and to show that what's discussed can actually be implemented. The "GL" prefix is omitted in parameters.

5.1 Requesting Mode

First of all, when setting up the pixel format we need to request an 8-bit stencil buffer by filling up the variable *cStencilBits* of the PIXELFORMATDESCRIPTOR structure with the number 8. The structure is then given to the ChoosePixelFormat –function and to the SetPixelFormat –function which sets the pixel format for the given device. After that we just normally call the wglMakeCurrent –function which creates our rendering context (all calls to OpenGL will affect this device).

5.2 Initialization and Use

We can just initialize the stencil with glStencilFunc(ALWAYS, 0, ~0) and turn it on with glEnable(STENCIL_TEST). The operation made to stencil can be controlled with glStencilOp(stencil_fail, z_fail, zpass). Parameters are for example KEEP, KEEP, INCR for front facing polygons and KEEP, KEEP, DECR for the back facing ones. Front and back facing polygons can be filtered with glCullFace(mode).

The above method could be used for shadow generation after rendering with ambient light which fills the Z-buffer with correct values. As the procedure fills the stencil its contents can then be used to determine shadowed vertexes. Then we can render the scene again with the light source on just ignoring the vertexes with stencil value $\neq 0$.

6 THE IMPLEMENTATION: DIRECTX 8

Microsoft introduced vertex and pixel shaders in DirectX 8.0. They can both be used for many kinds of advanced effects on the GPU. Currently, the newest version of DirectX API is 8.1 and it is used hereafter. This example is made specifically for this paper by the author. The *C++* compiler used is Microsoft Visual *C++* 6.0 SP5. A comprehensive guide to the *C++* language can be found from Stroustrup (1997).

6.1 Initialization

The basic initialization of the DirectX is very easy thanks to the new AppWizard-functionality. The wizard offers a base class for the usage of DirectX which is then inherited by the implemented application class, for example, *CMyD3DApplication*. This class can then override the *virtual* functions defined in the base class.

What's important here is to remember to set the value of *m_dwMinStencilBits* –variable to 8. This must be done in the constructor of the application. The base class then tries to acquire a device capable of handling 8 bits. What also must be done is a check in the callback function *ConfirmDevice* that we have a proper version of vertex shaders, unless we really want to run on software.

6.2 Used Structures

The finesse of the used data structures can have a great effect on the speed of the algorithm. The organization of the data used here is the same as proposed in ATi's RadeonShader demo.

The run-time creation of the objects shadow volume is enabled by precalculating an invisible extra quad between every adjacent face in the object. This makes the extruding of the back faces easy: The invisible quads in the silhouette stretch away from the light source and form the shadow volume for the object and light source pair. This is just like in the figure 7: All back facing polygons are extruded away from the light source by the influence of the light. No run-time geometry creation is needed.

6.3 Used Algorithm

The algorithm suggested here is basically the same "general" version as in chapter 5 with the exception that this algorithm uses a multipass version of Carmack's reverse. As code the basic loop is something like the following:

1. `m_pd3dDevice::Clear(0L,NULL,D3DCLEAR_ZBUFFER | D3DCLEAR_STENCIL | D3DCLEAR_TARGET,0xff,1.0f,0);`
2. For every object
    - `RenderAmbientPass();` // to fill the zbuffer
3. For every light
    - `m_pd3dDevice::Clear(0,NULL,D3DCLEAR_STENCIL,0xff,1.0f,0);`
      `m_pd3dDevice::SetLight(0,&light);`
    - For every object
        1. `RenderShadowVolume();` // to fill the stencil buffer

- For every object

    1. `RenderLitPass();` // with stencil test on

Once the vertex data of the shadowing object reaches the vertex shader (the function `RenderShadowVolume()`) something like the following is applied (the instructions are pseudo):

1. Calculate a vector from the light to the vertex in hand and normalize it.
    - **res1** = **P** - **L**
    - res2 = sqrt(**res1** • **res1**)
    - **res1** = **res1** / res2
2. Calculate the extrusion length (L_{range} is the maximum range of light).
    - res3 = L_{range} – res2
3. Do the extrusion.
    - **res4** = **P** + (res3 * **res1**)
4. Test that the polygon is back facing to the light. If not then use the original **P** instead of **res4**.
    - res5 = (**res1** • $\mathbf{P}_{norm}$) > 0

All this must be done in two passes because of the lack of two sided stencil test: First for the back facing polygons and then for the front faces. The actual code is not as simple as the above because, for example, calculating the length of a vector really requires two instructions. Also the fact that there is **no** *jump* command makes the actual implementation of 4 quite tricky, but still the above gives an idea of the needed work in the assembly level.

As can be seen the implementation requires multiple passes so fill rate from the hardware is really needed. And as can be seen this is not a basic technique, but the result is of adequate quality and speed.

7 CONCLUSION

What's discussed is the stencil shadow volume algorithm of generating shadows in real time. The original shadow volume algorithm was proposed by Crow in 1977 and the later published improvements, especially the stencil buffer, made the algorithm more mature. The algorithm is suitable, e.g., for computer games where the polygonal objects are not very complex. It is probably most effective when used with other techniques such as precalculated shadow maps.

The basic case of the algorithm is not very difficult but the various special cases with the relative position of the camera and the shadow volumes make a reliable implementation difficult and CPU consuming. Fortunately the modern GPUs can give us a hand in most of the needed calculation. The “final” form of the algorithm still remains to be seen.

The last chapters proposed techniques that could be use to implement the algorithm. Chapter 5 presented the basic case which was then refined in chapter 6. The proposed algorithms are not complete but give a quick tour on the usage of functionality of the most common APIs.

REFERENCES

Appel, A. 1968. Some Techniques for Machine Rendering of Solids. AFIPS 1968 Spring Joint Computer Conf.

Bergeron, P. 1986. A General Version of Crow’s Shadow Volumes. IEEE Computer Graphics and Applications 1986, 6(9). Universite de Montreal, Canada.

Carmack, J. 2000. CarmackOnShadowVolumes (Personal Communication between Carmack and Kilgard). Referenced: 10.4.2002. Available: *http://developer.nvidia.com/view.asp?IO=robust_shadow_volumes*.

Cebenoyan, C; Everitt, C; Rege, A. 2001. Hardware Shadow Mapping. Referenced: 1.4.2002. Available: *http://developer.nvidia.com/view.asp?IO=hwshadowmap_paper*.

Crow, F. C. 1977. Shadow Algorithms for Computer Graphics. SIGGRAPH 1977. New York, NY.

Everitt, C; Kilgard, M. J. 2002. Practical and Robust Stencil Shadow Volumes for Hardware Accelerated Rendering. Austin, Texas. NVIDIA. Referenced: 5.4.2002. Available: *http://developer.nvidia.com/view.asp?IO=robust_shadow_volumes*.

Kilgard, M. J. 2001. Robust Stencil Shadow Volumes. CEDEC. Tokyo, Japan. NVIDIA. Referenced: 15.3.2002. Available: *http://developer.nvidia.com/view.asp?IO=cedec_stencil*.

Kilgard, M. J. 1999. Creating Reflections and Shadows Using Stencil Buffers. GDC 99. Referenced: 15.3.2002. Available: *http://developer.nvidia.com/view.asp?IO=reflections_shadows_stencil_buffers*.

Stroustrup, B. 1997. The C++ Programming Language. Third Edition. Addison Wesley. 911 p.

Watt, A. 1993. 3D Computer Graphics. Second Edition. Addison Wesley. 500 p.