Hardware generated shadows

Márton Szabó
Dept. of Control Engineering and Information Technology
Technical University of Budapest
Hungary

Abstract

Recent advances of graphical hardware enable programmers to create more sophisticated looking real-time animations. These animations require some everyday, but not easily computable effects. As shadows are crucial in human perception, they are one of the most important effects. Since one of the most famous comprehensive survey on shadow algorithms (Woo [1], 1990) a plenty of practical methods have been written. My aim was to study some of these, which are well known, and change them to satisfy the needs of the programmable graphics hardware.
Keywords: Shadow, Soft Shadow, Hardware Shader

1 Introduction

Standing in shadow, or to see a piece of shadow moving on the wall is an everyday phenomenon. However, shadow rendering was not that obvious in real-time graphics applications until now, because shadow casting is a very resource demanding process.

This paper is divided into two main parts. The first part covers the description of some fast but physically not plausible shadow computing procedures, called hard shadows. I introduce the programmable graphics hardware implementation of the Simple Stencil Shadows, the Shadow Map, and Stencil Shadow Volumes algorithms. In the second part I discuss some algorithms, which can generate physically correct or approximate shadows, called soft shadows. I introduce some considerations on how to implement the Penumbra Wedges algorithm on GPU, and what optimizations can be done.

2 The programmable Graphics Hardware

The programmable graphics hardware is an architecture containing three processors. The first processor is the CPU, the computers general processor. The two other processors have unique instruction set, specialized on graphical computations, like multiplying vectors, matrices, etc. These are called the vertex shader, and the pixel (fragment) shader. The vertex shader is responsible for transforming the vertices into projection space, and for the execution of per vertex operations. The pixel shader colors the pixels one by one. The figure below shows a simplified view of this architecture.

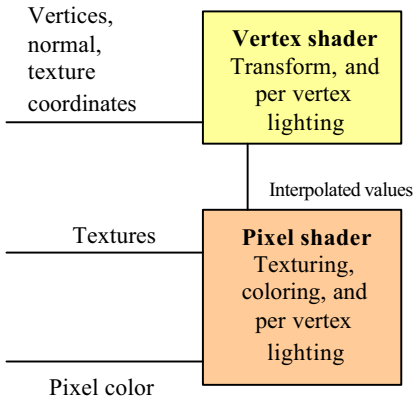

Figure 4.1 Architecture of the programmable graphics hardware

3 Shadows in Computer Graphics

3.1 What is the shadow?

Shadow is a piece of surface, what light cannot or only particularly can reach. We call these surfaces *receivers*. Bodies that prevent the light to reach the receivers we call *occluders*.

Researchers assign important role of shadows in understanding:

- The position and size of the occluder
- The geometry of the occluder
- The geometry of the receiver

4 What Are Shadows Good For?

According to the observations above shadows play important role in understanding geometry, so they should not be neglected. One does not care about existing shadows but everybody notices the lack of it.

Shadows bear much information about object geometry and help to determine the relative distances between the receiver and the occluder objects.

Figure 4.1 Relative object positions. On the left we cannot determine the position of the object

Figure 4.2 Shadow provide information about the geometry of the occluder.

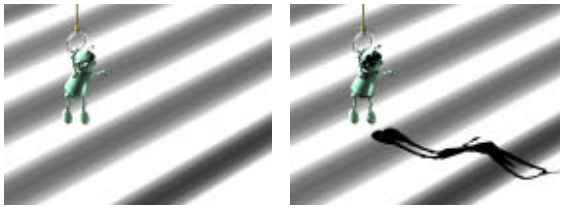

Figure 4.3 Shadows provide information about receiver geometry

5 Hard Shadows

Hard shadows are produced by point light sources. In this case, it is easy to decide whether a surface point is occluded or not. If the surface point cannot see the light source, it is in shadow. Hard shadows are not real but easy to model and can be computed in real-time applications.

5.1 Simple Stencil Shadows[2]

This is the simplest way to render shadows, however the results are rarely satisfying. The algorithm works only in case of planar receiver objects. To compute the shadow of the object relative to the light source, we have to transform all the points of the shadow casting object into the plane of the receiver.

After the "projection", we set every pixel of the object black. Given the position of the light, and the receiver plane, we can derive the transformation coefficient. We apply the transformation in homogenous coordinates, and finally complete the homogenous division.

- Position of the light: $\vec{l} = [l_x, l_y, l_z]$
- The plane:
  $S(\vec{r}) = \vec{n}(\vec{r} - \vec{r}_0) = 0 \; \vec{n} = [A, B, C] \; D = -\vec{n}\vec{r}_0$,
- Let ? be the light source–plane distance. Then the matrix that transforms the object into the receiver's plane is the following:

$$T_{shadow} = \begin{bmatrix} \gamma - l_x A & -l_y A & -l_z A & -A \\ -l_x B & \gamma - l_y B & -l_z B & -B \\ -l_x C & -l_y C & \gamma - l_z C & -C \\ -l_x D & -l_y D & -l_z D & \gamma - D \end{bmatrix}$$

5.2 Implementation

The program is a very simple one. I used only a single vertex shader. This shader gets the transformation matrix as input, and transforms all the vertices into the plane of the receiver. After transforming, it offsets the vertices a bit for the light source, to ensure that the z values of the shadow are in front of the receiver.

First we render the plane, with enabled stencil buffer. Second we render the shadow, if we find positive stencil values in the stencil buffer. This procedure ensures that the shadow is cast only onto the receiver.

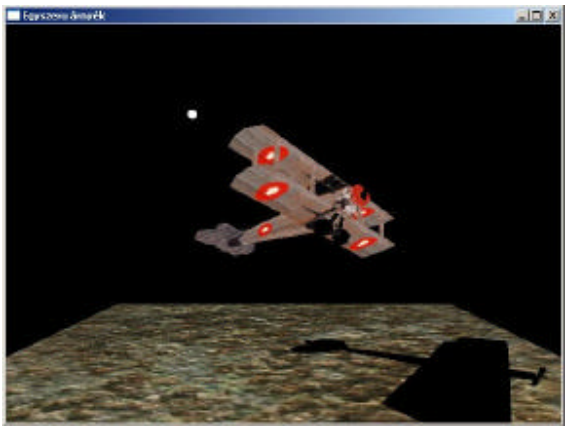

Figure 5.1 Simple Stencil Shadow screenshot

5.3 Simple Shadow Maps

The process to identify the shadowed areas is to identify the visible points from the light source.

- The first step is to render the scene from the light point of view, and store the depth values in a texture,

called shadow map. We will use this map when rendering the final image.
- Run the standard z-buffer algorithm to eliminate the invisible parts.
- After these we know the geometrical positions of the objects seen in the pixels. If the distance betw een an object and the light source is greater than the distance stored in the light map, then the object is in shadow. Else it is illuminated.
- If the part seen in the pixel is in shadow, we render it using only the ambient light/color component.

It is possible to use non standard extensions to implement shadow map with fixed function pipeline, like GL_SGIX_SHADOW, and it is not too difficult to implement it with shaders.

5.4 Shadow Map Implementation

To render the shadow map I used an offscreen buffer. First of all the scene has to be rendered from the light source. During rendering only the z values are stored. We change the render target, now it is the screen buffer. The vertex shader in my implementation is responsible for transforming the vertices into pro jection space, and to generate the input data for the pixel shader.

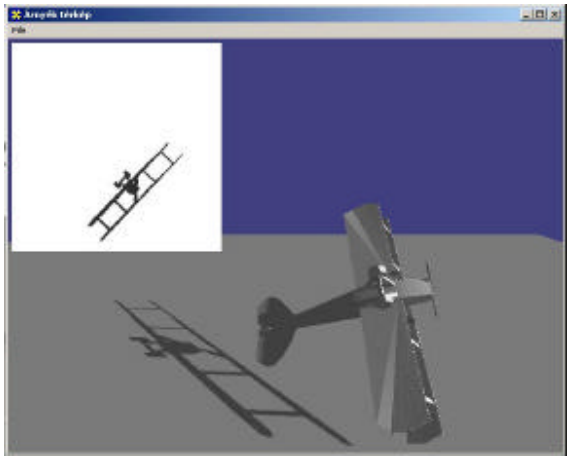

Figure 5.2 Shadow map implementation

An input needed is the world space z coordinates of the objects in the scene. The vertex shader writes these coordinates into a texture output register. The pixel shader is responsible for setting the colour values of the output pixels, and to decide whether a surface point is in shadow or not. The pixel shader samples the shadow map, and transforms the input coordinates from the vertex shader into the basis of the light source. After that, the comparison can be done.

5.5 Problems with Shadow Maps

The shadow map algorithm has some disadvantages. First of all, the z-buffer is not very precise, so artifacts may appear in self shadowing when surfaces get close to each other. This artifact can be avoided by offsetting the z values by a small bias.

The most disturbing problem is aliasing. If the light is far away from the viewer, individual pixels may become visible from the shadow map.
To avoid aliasing problems, a few methods were introduced.
- Storing object identifiers in the shadow map.
- Using deep shadow maps[3].
- Using adaptive shadow maps.
- Using perspective shadow maps[4].

5.6 Stencil Shadow Volumes

The shadow volume algorithm was first introduced by Crow, and Heidemann[5] created the first adaptation to programmable graphics hardware. The procedure is based on identifying the silhouette edges of the occluders, then extruding it along the light direction to form a shadow volume. Objects inside the shadow volume are in shadow, the others are illuminated.

The shadow volume is calculated in the following steps:
- The first step is to find the objects silhouette viewed from the light source. Edges shared by a triangle facing the light, and another one facing the opposite direction are worth to be kept.
- Then we form the shadow volume by extruding these edges along the light direction. The extruded faces are half planes. Together they define a closed volume (not closed in infinity). To decide whether a point in the scene is in shadow, we have to check if it is inside the volume.
- When rendering, we have to count how many times a ray from the viewpoint intersects the shadow volume. Front facing shadow volume faces increment the counter, while back facing faces decrement it. If the final number is greater than zero, then the point is in shadow.

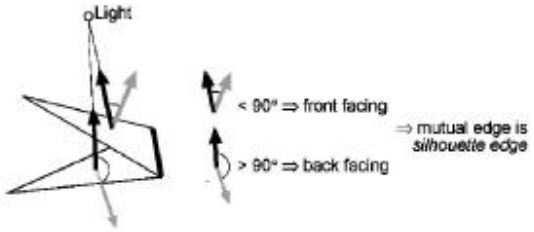

Figure 5.3 Detecting silhouette edges

We can easily implement the counting with the help of the stencil buffer. At the start of the rendering, the depth test has to be enabled. We render the faces facing the camera, increasing the stencil value by one at each face, in the second pass we render the back faces, decrementing the stencil value. Pixels that have positive stencil value are in shadow. This technique is called *z-pass*[7].

The whole algorithm is the following:

- Render the scene only with ambient lighting.
- Compute the shadow volume, and render it to the stencil buffer.
- Re-render the scene fully illuminated with stencil test, only pixels which have zero stencil value are updated.

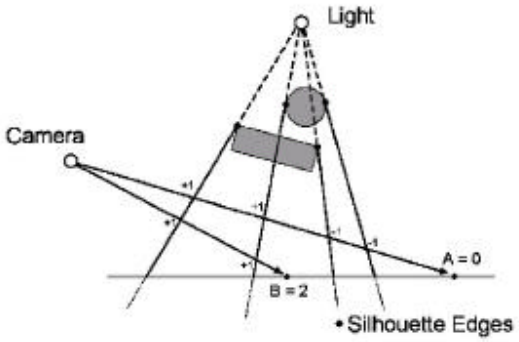

Figure 5.4 Z-pass algorithm.

5.7 Optimizing the Shadow Volume

The method does not support scenes, where the viewer stands in shadow. After rendering, the stencil buffer contains wrong values. Everitt[7] suggested a method that works in this case. This is called the *z-fail that* performs the steps of z-fail in reversed order.

- Render the scene with ambient light only.
- Render the back faces, incrementing the stencil value, if the depth test fails.
- Render the front faces, decrementing the stencil value, if the depth test fails.

5.8 Shadow Volume Implementation

Computer graphics programmers usually like Crow's shadow volume algorithm, because it computes the shadows in object space, so we have shadow information for every pixel. No shadow mapping technique can achieve high precision like that. Unfortunately, this precision is not for free, because silhouette determination puts high load on the processor, and the *fillrate* could become a bottleneck because of the large shadow volume surfaces. Another problem is that the data needed for shadow generation is stored both on the graphics hardware, and in the system memory, and the data have to be synchronized.

Because of the differences between the number representation of the system processor, and the graphical processor, the computation may be inaccurate. Programmable graphics hardware gives solutions for some problems.

- All computations run on the same hardware, so computed values will be accurate.
- The application gains more CPU time, because the graphics hardware does the shadow generation.
- We do not have to synchronize between the GPU and CPU.
- Silhouette detection executes parallel on many data units (depends on how many pixel pipelines our graphics hardware has), so we gain time.
- Using the algorithm is getting easier, because only local pre-processing is needed.

Figure 5.5 Shadow volume implementation

The first step in my implementation is to transform the object and light geometry into a common space. Both are transformed into the world space. This is view independent, and reusable for more light sources. Since every graphics hardware unit performs the view and projection transform concatenated, we load the identity matrix into the projection matrix. We label every vertex with a unique index number, to reference the vertex later.

Now we can load the data into the hardware shader. The output will be a texture, where every texel will hold a vertex' data. The place in the texture is defined by the index of the vertex. I used an RGBA floating point offscreen buffer the store the vertices. After that we load the edge information the same way. We can describe an edge with 4 vertex indices. Two of them mean the edge's end points, while the others mean the two remaining points of the adjacent triangles, which have the edge shared. We sample the generated textures in the pixel shader. For every edge we compute the faces normal, decide whether it is a silhouette edge, and then write it to the output to the edge's index if yes. We write vertex ordering of the face too. Finally, we have the silhouette edge indices in the pixel shader's output buffer.

6 Real Shadows (Soft Shadows)

6.1 Hard vs. soft shadows

In a simple case, a point is in shadow, or not. Shadows like that are always cast by point light sources. This

means, that a surface point can see the light source, or not. Unfortunately, light sources like that do not exist. Even the sun is an extended light source, so it generates soft shadow. Real light sources can be seen partially from a surface point. We call surface points that cannot see the whole light source the *penumbra*. Points that are totally occluded form the *umbra* region. All other points are totally illuminated. Computing the umbra and penumbra is a very complicated procedure, because we have to solve a visibility problem in the 3d space.

Figure 6.1 Soft shadow generated by a rectangular light source

Smoothness and quality of soft shadows depends on the distance between the light source, the occluder, and receiver objects. If the light source is a little one, and is far away, then we can approximate it with a point light source. Else soft shadow computing increases the visual quality of the rendered image.

6.2 Factors in Computing Soft Shadows

It is not too hard to compute shadows in simple scenes, but some factors have to be introduced for complicated scenes. For point light sources we can say, that

- In the case of more than one light source, the light is linear in nature.
- In the case of more than one occluder, the shadow is the union of the shadows of the occluders.

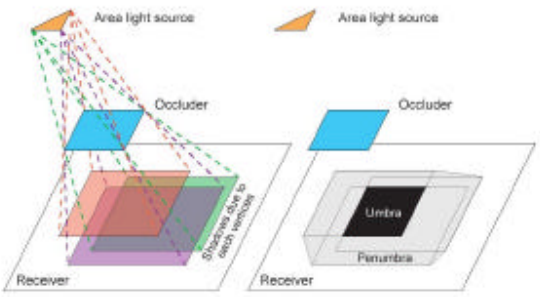

Figure 6.2 Area light source, and the generated umbra and penumbra region

These are not true for area light sources. It can happen, that a surface point is not totally shadowed by any of the occluders, yet it totally occluded by the shadows of the union of the occluders. This means, we can not combine the visibility functions. Shadow of union of occluders may be greater than the union of shadows of occluders. This is a serious problem, but not disturbing in fast moving scenes.

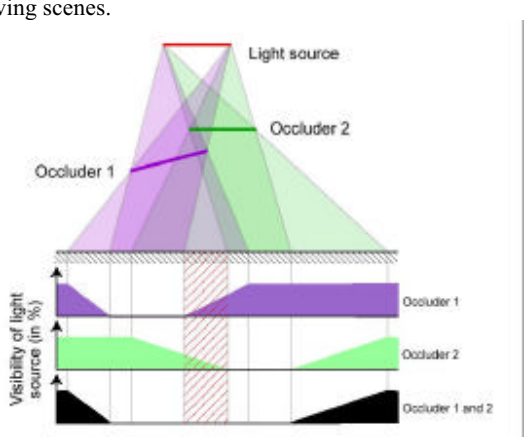

Figure 6.3 The lined area is not totally occluded by any of the objects, yet it is occluded by the union of them.

To accurately model the shadow cast by extended light sources, we have to identify all occluding points of an object, which can be seen from the light source. This is far more we can do in real-time, so many soft shadow algorithms compute the shadow only from one point of the light source, then simulate the penumbra from the computed hard shadow.

It is hard to notice the difference in fast moving scenes, but imagine a big-big light source close to the occluder, when the points of the light source see different sides of the occluder. If the occluder is extended along the surface normal of the light source, the penumbra may become very difficult to be computed. In this case:

- We can use the real model of the soft shadow. This is impossible in real-time applications.
- We can cut the light source into smaller light sources, and compute the shadow for each independently. This process discards some of the mistakes.
- Slice the occluder into smaller pieces, compute the shadows, and then combine them. Combination of the part shadows is difficult.

Computing the soft shadow in real-time means to compute the hard shadow, and extend it to soft shadow.

We have to remember the following when computing soft shadows:

- Softness of the penumbra region increases linearly as the light is getting further from the occluder. By the object it is zero.
- The umbra disappears when the light source is large enough.

- The method needs to be able to be implemented on programmable graphics hardware to reach real-time performance.
- Errors should be avoided. Low sample number on the light source could cause several hard shadows instead of one soft shadow.
- The algorithm should work on complex surfaces, and on fast moving scenes.

Unfortunately, all these conditions cannot be realized in any algorithm, we have to make simplifications.

7 Shadow Map Based Methods

There are some methods that compute the soft shadow using an image based approach.

- Computing the shadow as the combination of more shadow maps from different points of the light source[12].
- Using layered shadow maps, that stores depth information about all objects visible from at least one point of the light source.
- Using several shadow maps, and compute the percentage visibility of the light source from the surface points[9].
- Using image analysis techniques to compute soft shadow form shadow map [15].
- Convolving the shadow map with an image of the light source.

Figure 7.1 Shadow generated with 1, 4 and 1024 point samples

Some of these methods can reach real-time performance, but only with serious simplifications. I studied shadow volume based methods, which work faster, because they do not have to compute hundreds of samples per frame.

7.1 Shadow Volume Based Methods

We can compute the soft shadow by extending the shadow volume algorithm. This technique has many different implementations.

- We can compute more shadow volume, and combine them.
- By extending the shadow volume by heuristics, like Smoothies, Plateaus [16].
- By computing *penumbra wedges*, which are shadow volumes for all edges of the occluder[17].

7.2 The Penumbra Wedges Algorithm

The procedure replaces the half planes generated from the silhouette edges with penumbra volumes. To the first time the light source is supposed to be spherical. The light intensity (LI) means how much of the light source is visible in a single **p** point (s). If $s=0$, then the light source is in the umbra region, if $s>=1$, then the point is totally lit, if $0<s<1$, the point is the penumbra region. LI changes in the penumbra, the goal is to approximate LI in real-time.

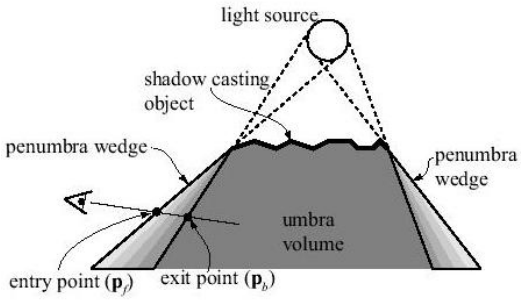

Figure 7.2 The ray penetrates the wedges, identifying the penumbra regions

The penumbra volumes implicitly define the umbra region. To reach sophisticated look for the scene, the light intensity interpolation has to be continuous. The idea is to introduce an easily rasterisable primitive, the *penumbra wedge*, that guarantees continuous light intensity interpolation.

Like shadow volumes, the penumbra wedges algorithm needs the use of the stencil buffer. To have more shades of grey in the penumbra region, we can use high precision stencil buffer (16 bits).

If we multiply LI with a k number, then we will have k different shades of grey. Let $k=255$, because a colour buffer consists of 8 bits. The penumbra wedge adds to, or subtracts from the value contained in the stencil buffer, for example a ray reaching the umbra region subtracts 256 from the buffer.

The algorithm consists of the following steps:

- Set all the values in the LI buffer to 256. This means we are outside of shadow.
- Render the scene with ambient lighting enabled.
- Compute penumbra wedges and light intensity.
- Modulate the scene with the light intensity.
- Add ambient lighting, and draw the scene.

7.3 Computing Penumbra Wedges

Penumbra wedges are approximated with half planes, front, back, and two sides. It would be more precise to extrude cones from the vertices, and contact them with patches, but it would degrade performance too much.

To compute the front and back plane, we create two points, $\boldsymbol{b} = \boldsymbol{c} + r\boldsymbol{n}$ and $\boldsymbol{f} = \boldsymbol{c} - r\boldsymbol{n}$, where $\boldsymbol{n}$ is the normal of

the shadow volume, ***c*** is the centre, ***r*** is the radius of the light source. Side planes are defined by the intersection of the adjacent wedges' front and back planes.

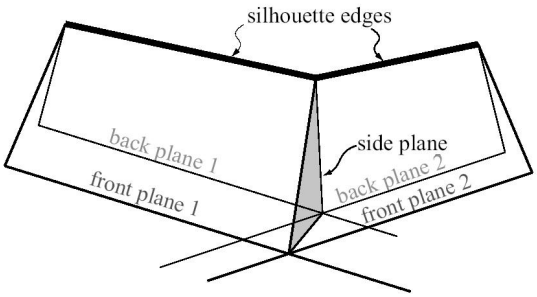

Figure 7.3 The front and back planes define the common side plane

Note that setting the light source radius to zero would create simple hard shadows.

7.4 Light Intensity Interpolation

The light intensity is interpolated with the following formula:

$$s = \frac{t_r}{t_r + t_l} s_l + \frac{t_l}{t_r + t_l},$$

where t_r, and t_l are the positive intersection distance with the side planes from point **p**. As the side plane directions are common between adjacent wedges, the method ensures C^0 continuity. The procedure does no sampling, so the LI interpolation is always continuous.

7.5 Optimization and Implementation

I implemented the penumbra wedges algorithm with a few modifications, and I used some source code from the internet site of the authors[12]. In the next chapters I will show what optimizations can be done.

7.6 Narrow wedges for rectangular light

If we have rectangular light source, we can extrude the corner points of the light source through the endpoints of the silhouette edges. This way we get rectangular cones. To close the wedge we can use the side faces of the rectangular cones, and we add a bottom plane to make z-fail technique usable.

This construction of the wedges created usually narrower wedges than the method in the previous chapter. Edges that intersect the light source have to be eliminated, because the wedge would close everything inside.

7.7 Optimized Shaders

To optimize rasterization, wedge creation, and final rendering, highly optimized shader programs have been created, one for rectangular light sources, one for spherical light sources.

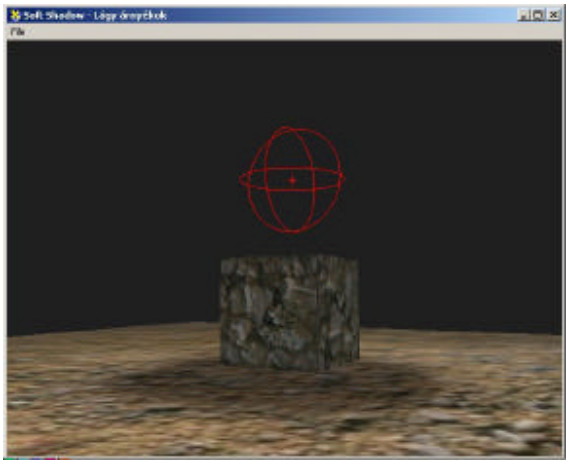

Figure 7.4 Soft shadow with large sphere light

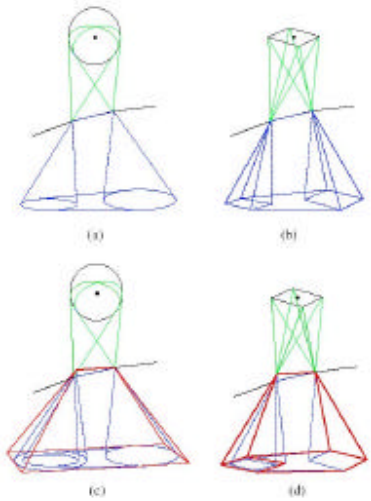

Figure 7.5 Penumbra wedge creation in the case of different light sources

The spherical shader uses a cone defined by the actual pixel, and a light source. It projects the silhouette edge onto the plane of the light source, and computes the occlusion percentage. In the case of rectangular light source, the silhouette edge has to be clipped before projection, because the opposed points may invert. The clipping and projection can be done in homogenous coordinates.

7.8 Backface Culling

Penumbra volumes affect those pixels that are inside the penumbra region. The z coordinate of these points is always inside a wedge. However penumbra wedges occlude more points than it affect. It is not necessary to start the pixel shader programs for these unaffected points. Backface culling has to filter points that are outside the actual wedge. This can be done by the

combination of the stencil and the depth test. The first pass we render the front faces, setting the stencil value to one for every drawn pixel. The depth test condition is "*greater than*", because we have to draw pixel further than the front face. In the second pass we render back faces with enabled stencil and depth test. Stencil test succeeds only when "*equal*", and depth test succeeds when "*less*". If any of the tests fail, the pixel is discarded.

Acknowledgements

It is always a hard task to decide weather to use hard or soft shadows. A computer has to deal with many different problems, not only graphical, but physical modeling problems, and maybe AI. In the case of a complex game, the computer may not have enough resources to compute soft shadows. This is the main case why we try to implement more and more in graphics hardware: to reduce the load of the CPU.
However, it is not a simple task. Sometimes we need more steps to compute shadows that may degrade the throughput of the shaders. Programmers have to decide what kind of shadows they use, and quality has to be adjustable during animation.
Unfortunately some features are still missing, so we cannot achieve the best performance. For example, we cannot read back tables in the vertex shader.
Future aims could be to find a new way to render soft shadows, or to optimize the recent methods to increase performance, using the features of the v3.0 shader architecture.

References

[1] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. IEEE Computer Graphics and Applications, 10(6):13–32, November 1990.

[2] Szirmay-Kalos László (editor): http://www.fsz.bme.hu/~szirmay/book.html. Theory of Three Dimensional Computer Graphics .Publishing House of the Hungarian Academy of Sciences, 1995, p 430.

[3] Tom Lokovic and Eric Veach. Deep shadow maps. In Computer Graphics (SIGGRAPH 2000), Annual Conference Series, pages 385–392. ACM SIGGRAPH, 2000.

[4] Marc Stamminger and George Drettakis. Perspective shadow maps. ACM Transactions on Graphics SIGGRAPH 2002), 21(3):557–562, 2002.

[5] Tim Heidmann. Real shadows, real time. In Iris Universe, volume 18, pages 23–31. Silicon Graphics Inc., 1991.

[6] Szabó Márton, A számítógépes grafika árnyékos oldala – "Dark side of computer graphics". University Student's conference paper, 2003, November.

[7] Cass Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. http://developer.nvidia.com/docs/IO/2585/ATT/RobustShadowVolumes.pdf, 2002.

[8] Stefan Brabec and Hans-Peter Seidel. Shadow volumes on programmable graphics hardware. Computer Graphics Forum (Eurographics 2003), 25(3), September 2003.

[9] Wolfgang Heidrich, Stefan Brabec, and Hans-Peter Seidel. Soft shadow maps for linear lights high-quality. In Rendering Techniques 2000 (11th Eurographics Workshop on Rendering), pages 269–280, 2000.

[10] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In Graphics Hardware, 2003.

[11] Cyril Soler and François X. Sillion. Fast calculation of soft shadow textures using convolution. In Computer Graphics (SIGGRAPH 1998), Annual Conference Series, pages 321–332. ACM SIGGRAPH, 1998.

[12] Michael Herf. Efficient generation of soft shadow textures. Technical Report CMU-CS-97-138, Carnegie Mellon University, 1997.

[13] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, Carnegie Mellon University, January 1997.

[14] Maneesh Agrawala, Ravi Ramamoorthi, Alan Heirich, and Laurent Moll. Efficient image-based methods for rendering soft shadows. In Computer Graphics (SIGGRAPH 2000), Annual Conference Series, pages 375–384. ACM SIGGRAPH, 2000.

[15] Stefan Brabec and Hans-Peter Seidel. Single sample soft shadows using depth maps. In Graphics Interface, 2002.

[16] Eric Haines. Soft planar shadows using plateaus. Journalof Graphics Tools, 6(1):19–27, 2001.

[17] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In Rendering Techniques 2002 (13th Eurographics Workshop on Rendering), pages 297–306. Springer-Verlag, 2002.

[18] J.M. Hasenfratz, M. Lapierre, N. Holzschuch, F.X. Sillion. A Survey of Real-time Soft Shadows Algorithms Eurographics 2003