Real-time Shadow Mapping

Joined Project in Computer Graphics for Rafael Armament Development Authority

Igor Naigovzin

Supervisors: Dr. Ayelet Tal - Technion
Doron Tal - Technion
Liav Adi - Rafael

March, 2006
Acknowledgements

The presented project is based on research conducted during my studies on Computer Science faculty at Technion – Israel Institute of Technology. I want to thank especially my supervisors Dr. Ayelet Tal, graduate student Doron Tal both from Technion and to Liav Adi from Rafael Armament Development Authority for their enormous help and advice.
Contents

Abstract

1 Introduction and Overview

2 Shadow Mapping
   2.1 Description
   2.2 First Pass
   2.3 Projective Texture Mapping
      2.3.1 Homogeneous Texture Coordinates
      2.3.2 Texture Projector
      2.3.3 Texture Coordinates Generation
      2.3.4 Reverse Projection
   2.4 Second Pass

3 Light Types

4 Self-Shadowing Artifacts

5 Filtering

6 Ambient Shadows

7 Projective Texture

8 Implementation
   8.1 Background
      8.1.1 OpenSceneGraph Library
      8.1.2 Shaders Mechanism
         8.1.2.1 Vertex Programs
         8.1.2.2 Texture Fetching
         8.1.2.3 Fragment Programs
   8.2 API
   8.3 Improvements and Optimizations

9 Hardware Requirements

10 Summary

11 Results

12 Future Work

13 References
Abstract

I present efficient real-time hardware accelerated method for rendering high-quality, antialiased shadows for complex scenes using the shadow mapping technique. The whole project is implemented by OpenSceneGraph library using shaders mechanism. The implementation includes both, CG and GLSL shader languages. This paper describes each step of the algorithm.
# 1 Introduction and Overview

Recent developments in graphics hardware have led to systems which provide very high polygon throughput combined with sophisticated lighting and shading models. However, improvements in image quality are still focusing on local properties, e.g. using per fragment Phong lighting. For realistic images, which should provide accurate impressions of the world in 3D, one must also consider the computation of global effects. One of the most important effects in this category are of course good-looking shadows.

Shadows make 3D computer graphics look better. Without them, scenes often feel unnatural and flat, and the relative depths of objects in the scene can be very unclear. If you’ve used OpenGL or Direct3D, you know that lights in these APIs do not automatically cast shadows. An example is shown in Figure 1. On the first image there is no shadows, making it very hard to determine whether the ball is on the ground or if it is floating above the ground. This is, on the other hand, easily determined on the following images, where shadows are shown.

![Figure 1](image-url)  
**Figure 1. Spatial Relationship.** Left: it is hard to determine whether the ball is on the ground or floating on the air. Middle and Right: this is easily determined due to the shadows cast by the ball.

There are lots of ways to compute shadows. The trouble with rendering high quality shadows is that they require a visibility test for each light source at each rasterized fragment. For ray tracers, adding an extra visibility test is trivial, but for rasterizers, it is not. Fortunately, there are a number of common cases where the light visibility test can be efficiently performed by a rasterizer.

In the field of real-time and interactive rendering, there are only a few widely used algorithms. Casting shadows onto large receiver polygons is mostly done using projected geometry, whereas more complex receivers can only be handled using the shadow volume approach. These two methods have in common that they still require CPU resources to compute geometry and may generate a large amount of additional polygons that have to be pushed down the graphics pipeline. The main benefit of these algorithms is that all computations are done in object space which usually results in highly accurate shadows. Even if these techniques are capable of producing high-quality shadows, they are not suited for real-time applications, due to their object based computations, such as video games. They are usually used to produce shadows for static scenes or to compute shadows for large receiver planes, caused by only a few occluder objects. Another class of algorithms is based on the shadow mapping or depth mapping algorithm. These methods operate in image space which means that they rely on a sampling scheme to process the scene geometry. The shadow mapping algorithm can be used to compute shadows for arbitrary receiver and occluder geometry. Since sampling is the fundamental principal of all rasterization-based graphics architectures, shadow mapping is the first choice for real-time rendering. However, sampling methods all suffer from undersampling artifacts when it comes to higher frequency parts and the same for the shadow mapping algorithm. Sampling problems occur during the generation phase as well as when performing the actual shadow test. The first can be solved by increasing the image resolution and by using stochastic sampling instead of sampling at regular grid points. In order to resolve sampling artifacts during the shadow test, Reeves proposes a filtering method called percentage closer filtering which will be described later in more details.
2 Shadow Mapping

*Shadow mapping* is an image-based shadowing technique developed by Lance Williams in 1978. It is particularly amenable to hardware implementation because it makes use of existing hardware functionality – texturing and depth buffering. The only extra burden it places on hardware is the need to perform a high-precision scalar comparison for each texel fetched from the *depth texture*. *Shadow mapping* is also attractive to application programmers because it is very easy to use, and unlike stenciled *shadow volumes*, it requires no additional geometry processing.

2.1 Description

The idea of the traditional *shadow mapping* consists of two steps. The first step serves the purpose of generating an image that contains depth information of the front-most pixel, seen by the light source. In computer graphics parlance, such an image is called a *depth texture* or *shadow map*. Such a *depth texture* is generating by setting up a camera with the position, direction, and view properties of the light source and rendering the scene. In the second pass, the scene is rendered once again from the actual camera point of view, but with the *depth texture* projected down from the light onto the scene using standard *projective texture mapping*. At each pixel, the depth sample (from the projected *depth texture*) is compared with fragment’s distance from the light. If the latter is great, the pixel is not the closest surface to the light source. This means that the fragment is shadowed, and that it should not receive light during shading.

Figure 2 illustrates the *shadow mapping* comparison. On the left panel of the figure, the point that is being shaded is in shadow, because the point’s depth is greater than the depth that is recorded in the *depth texture*. In contrast, the right panel of the figure shows a case where the point has the same depth as the recorded value in the *depth texture*. This means that there isn’t any object between the point and the light source, so the point is lit (not in shadow).

![Figure 2. The Shadow Mapping Depth Comparison. Left: the point that is being shaded is in shadow. Right: the point that is being shaded is not in the shadow.](image)
2.2 First Pass

First of all, the scene is rendered from the light’s point of view. This is done by setting the camera view to the light view by the transformation shown in Figure 3.

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} = \begin{bmatrix}
\text{Light view} \\
\text{Model view} \\
\text{look at} \\
\text{matrix}
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

Next, the scene is rendered. Optimization is performed in this stage: instead of full rendering (color buffer and depth buffer), only rendering of the depth buffer into the depth texture is done using ARB_depth_texture, which defines a texture internal format for GL_DEPTH_COMPONENT, complete with various bits-per-texel choices. It usually takes about half the time that a full rendering takes. Furthermore, it is important that the viewport is only the size of the depth texture. Figure 4 illustrates a scene rendered from eye’s point of view, light’s point of view and the depth buffer of a scene rendered from the light’s point of view.

![Figure 3. Transformation from world space into light space.](image)

![Figure 4. Depth texture rendering. Left: the scene is rendered from the eye’s point of view. Middle: the scene from the light’s point of view. Right: the depth buffer from the light’s point of view.](image)
It is important that the near and far planes are located very precisely because the depth buffer has a limited precision. It is done by minimizing the distance between the near and the far planes by calculating the optimal near and far plane values which cover all the scene objects. This is done by calculating the bounding sphere of the whole scene and setting the near plane to be in a distance from the light source to the center of the scene minus radius of the scene, and the far plane to be in a distance from the light source to the center of the scene plus radius of the scene. This minimizes errors while comparing the depth values. Figure 5 illustrates how the settings of the near and far plane can affect the depth texture. In addition, the field of view of the light frustum is calculated in such a way that only the part of the scene which is seen by the eye is included. The geometrical solution is to calculate the convex hull of the eye frustum, and then set the light field of view to include the convex hull of the eye frustum.

![Figure 5. Near and far plane settings. Left: the optimal near and far plane settings have been applied. It is seen that the contrast of the depth texture is large. Objects being close to the light source are nearly black. Right: the near and far plane not are optimized, the contrast of the depth texture degrades resulting in a less accuracy.](image)

The depth buffer needs to be copied into a depth texture. For a better performance, instead of rendering the scene into depth buffer and then copying it into the depth texture, rendering into depth texture is performed using fragment buffer object.
2.3 Projective Texture Mapping

*Projective texture mapping* is a method of texture mapping that allows the texture image to be projected onto the scene as if by a slide projector. Figure 6 illustrates some example screen shots available in the NVIDIA OpenGL SDK.

![Figure 6. Texture projected onto the scene.](image)

### 2.3.1 Homogeneous Texture Coordinates

*Projective texture mapping* is useful in a variety of lighting techniques, including *shadow mapping*. *Projective texture mapping* refers both to the way texture coordinates are assigned to vertices, and the way they are computed during rasterization of primitives. When performing *projective texture mapping*, we use homogeneous texture coordinates, or coordinates in projective space. When performing non-projective texture mapping, we use real texture coordinates, or coordinates in real space. For *projective 2D texture mapping*, the 3-component homogeneous coordinate vector \((s, t, q)\) is interpolated over the primitive and then at each fragment, the interpolated homogeneous coordinates are projected to a real 2D texture coordinate, \((s/q, t/q)\), to index into the texture image. For non-projective 2D texture mapping, the 2-component real coordinate vector \((s, t)\) is interpolated over the primitive and used directly to index into the texture image.

### 2.3.2 Texture Projector

Consider that the texture is being projected onto the scene by a slide projector. This projector has most of the same properties that cameras have – it has a *viewing transform* that transforms world space coordinate into projector space, and it has a *projection transform* that maps the projector space view volume to clip coordinates. Figure 7 illustrates the difference in transformations for a conventional camera vs. those for a projector. On the left image there is a sequence of transformations in the conventional pipeline, and on the right image there is a sequence of transformations for a projector.
2.3.3 Texture Coordinates Generation

Using automatic texture coordinate generation *texgen* it is possible to transform the homogeneous texture coordinates \((s, t, r, q)\) using the following matrix:

\[
\begin{bmatrix}
    s \\
    t \\
    r \\
    q
\end{bmatrix} = \begin{bmatrix}
    1/2 & 0 & 0 & 1/2 \\
    0 & 1/2 & 0 & 1/2 \\
    0 & 0 & 1/2 & 1/2 \\
    0 & 0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
    \text{Projector projection matrix} \\
    \text{Projector view matrix} \\
    \text{Model view matrix}
\end{bmatrix} \cdot \begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix}
\]

1. **Multiply by the model view matrix.** This applies any necessary model view transformations to the vertices. The model view matrix needs to be applied regardless of whether or not you are using projective texture mapping.

2. **Multiply by the projector view matrix.** This rotates and translates the vertices to the projector eye coordinates.

3. **Multiply by projector projection matrix.** This defines the projector’s frustum, including its field of view and aspect ratio. In the case of shadow mapping the depth texture is projected from the light, so the projection view and projection matrixes are the light’s one.

4. **Scale and bias the results.** Following steps 1 to 3, the transformed vertex values range from \((-1)\) to 1. However, textures are indexed from 0 to 1, so the results have to be mapped to this range. This is done by multiplying the \(x\), \(y\) and \(z\) components of the results by \(1/2\) and then adding \(1/2\). A single matrix multiplication accomplishes this mapping.
2.3.4 Reverse Projection

Unlike a real projector, the math of \textit{projective texture mapping} actually produces a dual projection. One along the projector’s view direction, and another in the opposite direction. Figure 8 illustrates the \textit{reverse projective texture mapping} effect. This problem is solved by comparing the value of $q$, cause the sign of $q$ becomes negative behind the projector, which inverts the texture image in the reverse projection.

Figure 8. Reverse projective texture mapping: Produces a reverse projection as well.
2.4 Second Pass

Next, the scene is rendered from the eye’s point of view, but with the depth texture projected down from the light onto the scene using standard projective texture mapping. At each pixel, the depth sample (from the projected depth texture in (s/q, t/q)) is compared with the fragment’s distance from the light that is r/q. If the latter is bigger (r/q > texture[s/q, t/q]), the pixel is not the closest surface to the light source. This means that the fragment is shadowed, and that it should not receive light during shading. If r/q ≈ texture[s/q, t/q], then the pixel is the closest surface to the light source. This means that the fragment is lit (not in shadow), and that it should receive light during shading. Figure 9 illustrates the whole process.

![Figure 9. Shadow mapping: The whole process.](image)
3 Light Types

The simple implementation of shadow mapping only supports spot and directional lights. This is due to the fact that a single shadow mapping cannot encompass the entire hemisphere around itself. Cube mapping could be used to allow point light sources, however this would require up to six depth textures and therefore six renderings of the scene, decreasing the speed of the algorithm considerably. Directional lights are implemented by setting up the light source far away from the scene accordingly to its direction.

4 Self-Shadowing Artifacts

The main problem when using shadow mapping is the limitation in resolution and precision of the depth texture. One artifact is self-shadowing which can be seen in figure 10. The cause is the resampling which take place when eye space pixel coordinates are transformed into light space to get the respective depth texture samples. Ideally, a surface point in eye space should map to the same surface point in the light coordinate system. Unfortunately, it is very likely that the result is slightly incorrect, due to machine rounding errors during the transformation of point from eye space into light space. The object might shadow itself, even if it is a lit part.

To resolve this problem, a small bias is added to the depth texture values used in the comparison. This can be done using glPolygonOffset(factor, units), which offsets the depth value by an amount equal to:

$$offset = factor \times DZ + units \times R$$

where \(DZ\) is the polygons’ slope and \(R\) is the smallest value guaranteed to produce a resolvable offset for the given implementation. The result of this can be seen in Figure 11. If the offset is too low, shadowing artifacts will appear. If the offset is too high, the shadow will move away from the object giving the impression that it is floating in case when the object is not. Finding the correct value is a challenging task which is highly scene dependant.
Another technique would be to render only back-facing polygons into the depth texture. Figure 12 illustrates the front-face culling process. This would result in the depth value when seen from the camera clearly being smaller than the corresponding depth texture value, thereby avoiding the biasing problem. This would, however, require the objects to be closed meshes, since false shadowing would otherwise occur, thereby loosing shadow mapping ability to calculate shadows independent of geometry.
5 Filtering

Although biasing errors are now minimized, there is still the problem of aliasing. Averaging depth values does not work. One way to reduce this problem is to use a filtering mechanism called \textit{percentage closer filtering (PCF)} to smooth the edges. Instead of just comparing the depth value to the closest \textit{depth texture} value, it is compared against a number of the closest values averaging the result. When shading each pixel in the eye view, \textit{PCF} returns a value that indicates the amount of shadowing at each shaded point replacing the traditional depth comparison of ordinary \textit{shadow mapping}. Figure 13 illustrates how \textit{percentage closer filtering (PCF)} works. Basically, \textit{PCF} works by reversing the order of filtering and comparing. Instead of first filtering the texture image over some specific region and using the resulting value for further processing, \textit{PCF} performs the comparison step. \textit{PCF} is based on the observation that as the size of \textit{PCF} kernel increases, the resulting shadows become softer. The challenge is to vary the filter size intelligently to achieve the correct degree of softness. When linear filtering is performed on the \textit{depth texture}, the graphics card hardware applies the 2x2 \textit{PCF}. Figure 14 illustrates the \textit{PCF} influence.

![Percentage Closer Filtering (PCF)](image)

**Figure 13.** Percentage Closer Filtering (PCF). Left: the pixel to be shaded. Top: the shaded pixel without PCF. Bottom: the shaded pixel with 2x2 PCF kernel.
6 Ambient Shadows

Rendering the depth texture with standard OpenGL does not look very good. The shadow is completely black. To alleviate this problem, ambient shadows coefficient is used that determines the fraction of light that is actually blocked. Objects tend to look too flat if all the light is blocked because there is no shading variations to reveal surface curvature. Lightening the shadows also keeps them from being too "harsh". Figure 15 illustrates the ambient shadows influence.
7 Projective Texture

As projective texture mapping is part of the shadow mapping process, we can use it for projecting texture on the scene. Where there is a shadow (behind occluders), projective texture is not applied, which is true for a real projector. Figure 16 illustrates the result of projected texture and shadows both applied on the scene using different projective texture brightness.

Figure 16. Projective texture mapping. Left: a scene with projected texture with brightness 0.2. Middle: a scene with projected texture with brightness 0.5. Right: a scene with projected texture with brightness 0.8.
8 Implementation

The whole project is implemented using OpenSceneGraph library (see section 8.1.1) and shaders mechanism in both CG and GLSL languages. The first and second passes is done in OpenSceneGraph, while the shadow calculations is done in fragment program without unnecessary use of vertex program (see section 8.1.2.1). The core of shadows is packed into dll, which can be used by OpenSceneGraph users.

8.1.1 OpenSceneGraph Library Background

The OpenSceneGraph is an open source, cross platform graphics toolkit for the development of high performance graphics applications such as flight simulators, games, virtual reality and scientific visualization. Based around the concept of a scene graph, it provides an object oriented framework on top of OpenGL freeing the developer from implementing and optimizing low level graphics calls, and provides many additional utilities for rapid development of graphics applications.

8.1.2 Shaders Mechanism Background

In August 1999, the first Graphics Processing Unit (GPU) has been introduced in PC industry. A GPU is a 3D processor that integrates the whole graphics pipeline in one chip. Graphics processors free the PCs processor from all 3D specific calculations so the normal CPUs can save performance for application specific computations.

Latest features of graphics cards are enabled by their drivers which establish the communications between the PC and application. These features work on all APIs which support them. If the underlying hardware does not support a feature, then the driver emulates it and it runs on the host CPU. An example of such a new technology is the programmable interfaces to graphics hardware. It allows the user to replace fixed wired functionality on the graphics chip with user defined programs. At the moment there are three stages in the graphics pipeline that can be accessed by user defined programs: the geometry of vertex processing unit, the texturing unit and the fragment processing unit.

8.1.2.1 Vertex Programs

Vertex programs are small assembly like programs written by the programmer. Once a vertex program is enabled, the fixed wired vertex processing unit is bypassed and substituted by the vertex program. Figure 17 illustrates this bypass. The advantage of vertex program is that it is much faster than software implementation.
*Vertex programs* are written in a SIMD instruction set which is strictly sequential. This means there is no instruction to return before the end and there are also no conditional flow controls available. Note that *vertex programs* have many advantages due to their flexibility and speed, but they also require the user to implement all basic functionality by himself.

### 8.1.2.2 Texture Fetching

Traditional *texture fetching* retrieves a value from a texture addresses by certain texture coordinates. For applying the projective texture sampling a different command is used in which the corresponding homogeneous textures coordinate s/q and t/q are used to access the texture and read the value at this position. In addition, when the above projective texture sampling is done on a *depth texture*, the hardware also makes the comparison.

### 8.1.2.3 Fragment Programs

*Fragment programs* operate on pixel information. They bypass the part of the graphics pipe which deals with constructing the final pixel color. *Fragment programs* have the same advantages as *vertex programs* have, and suffer from the same disadvantages as *vertex programs* do.

### 8.2 API

**Shadows:**
- **Enable** – enables/disables the shadow
- **UseShader** – defines which of the shader languages will be used (CG, GLSL or no shader)
- **SetAntialiasingMode** – sets the antialiasing mode (2x2, 3x3, 4x4 or no antialiasing)
- **SupportHardwareAntialiasing** – enables/disables hardware antialiasing support
- **SetLightDirection** – sets the light source to be directional light source
- **SetFrustum** – sets the user shadows frustum
- **CalculateFrustumAutomatically** – enables automatically frustum calculations
- **SetAmbientShadow** – sets the ambient shadow coefficient
- **SetDefaultBias** – sets the default bias factor to 1.1 and bias units to 4.0
- **SetBias** – sets the user bias factor and units
- **CullFaces** – enables/disables the front faces culling

**ProjectiveTexture:**
- **EnableProjectiveTexture** – enables/disables the projective texture
- **SetProjectiveTextureFileName** – sets the projective texture from file
- **SetProjectiveTextureBrightness** – sets the projective texture brightness
8.3 Improvements and Optimizations

In the first pass instead of full rendering, only rendering into depth buffer is done. In the first pass instead of rendering the scene into depth buffer and then copying it into the depth texture, rendering into the depth texture is performed using framebuffer object. Depth texture size can be defined by the user. Light frustum is efficiently calculated each frame. Reverse projection is eliminated. Spot and directional lights, biasing, antialiasing and hardware antialiasing, ambient shadows, projected texture, CG and GLSL shading languages are supported.

9 Hardware Requirements

Hardware accelerated shadow mapping is available today starting on NVIDIA GeForce 3 GPUs. It is exposed in OpenGL through the SGIX_shadow and SGIX_depth_texture extensions, and in Direct3D 8 through a special texture format. Antialiased shadow mapping is available starting on NVIDIA GeForce 4 GPUs using OpenGL through the ARB_shadow and ARB_depth_texture extensions. Framebuffer object is available starting on NVIDIA GeForce 6 GPUs using OpenGL through the EXT_framebuffer_object extension. The results described in section 11 tested on the following cards: NVIDIA GeForce 5200, 5900, 5950, 6600, 6800, 7800.

10 Summary

Shadow mapping is capable of computing shadow for any arbitrary geometry and best suited for real-time and interactive applications, because it can be completely ported to graphics hardware. The shadow mapping generation process growth linearly in time with the number of primitives. The total additional time is therefore roughly twice the time as without shadows. No lighting, shading and texture computation is required, while the depth texture is rendered, and only a few additional transformation calculations are needed. In contrast to these advantages, shadow mapping suffers from aliasing artifacts and insufficient depth texture resolution, leading to coarse shadows. Various techniques such as percentage closer filtering or adding a certain bias can be used to solve aliasing artifacts. Still, the limitation of the field of view and the waste of valuable depth texture space often remain an unsolved problem.
11 Results

Figure 18. Resulting images.
12 Future Work

This project implements a traditional shadow mapping algorithm, which is definitely can be improved. The common improvement for shadow mapping is **perspective shadow mapping**. This approach adapts the depth texture resolution according to the camera setup, and generates a single perspective depth texture by using non-uniform parameterization. This method exploits graphics hardware and reduces shadow mapping aliasing significantly, because the area of projected depth texture pixels in post-perspective space is almost equivalent to the area of eye view pixels. **Perspective shadow mapping** provide nearby objects high resolution, while more distant objects are rendered with decreasing resolution.

Aliasing artifacts and jagged edges occur on shadow boundaries. Thus, to obtain high-quality shadows the corresponding depth texture does not need an overall high resolution, but only requires high resolution on shadow boundaries. To determine these regions the **adaptive shadow mapping** algorithm evaluates how much depth texture pixels contribute to the total image quality, and refines the depth texture in such a way that it creates new higher resolution pieces if needed. On shadow boundaries, the depth texture resolution should be at least the same as the resolution in eye space to avoid coarse shadow.

Traditional shadow mapping is only useful in conjunction with a perspective projection using a reasonable filed of view (FOV). Therefore, shadow mapping for hemispherical or omnidirectional light sources fails. **Sphere mapping** was the first environment mapping technique supported by consumer level hardware. This method generates a 2D texture image in the following way. In a first step a perfectly mirroring sphere is centered around the objects of interest. The appearance of the surrounding environment is then derived from an orthographic camera facing this ball. The disadvantage of this mapping is that pixels along the perimeter of the flattened ball get extremely distorted. **Paraboloid mapping** is another mapping similar to sphere mapping. Using this mapping, the 2D texture image is obtained by the view of an orthographic camera facing a perfectly mirroring paraboloid instead of a sphere. In contrast to the spherical parameterizations, this method does not introduce that much artifacts due to a better sampling rate. For omnidirectional lights, two paraboloids are simply attached back to back.

Real life light sources are not point light sources, they cover a given area in space and therefore the transition of the shadow boundaries is usually more smooth. **Soft shadows** take care of this smoothing.

Till now we discussed about shadows generation for one single light source. A number of lights can be added to the scene, which gives it more realistic quality.
13 References

[1] Lance Williams, "Casting Curved Shadows on Curved Surfaces", SIGGRAPH 78.


