Univerzita Komenského

Fakulta Matematiky, Fyziky a Informatiky

Shadow Mapping for Large Virtual Environments

2010 **Michal Ferko**

Univerzita Komenského

Fakulta Matematiky, Fyziky a Informatiky

Shadow Mapping for Large Virtual Environments

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 9.2.1 Informatika
Školiace pracovisko: Katedra Informatiky
Školiteľ: RNDr. Martin Samuelčík

Bratislava, 2010 **Michal Ferko**

Čestné prehlásenie

Čestne prehlasujem, že prácu som vypracoval sám s pomocou uvedenej literatúry.

Bratislava, 8. júna 2010 Podpis .....

Poďakovanie

Týmto sa chcem poďakovať vedúcemu práce RNDr. Martinovi Samuelčíkovi za podporu a poskytnutie odborných konzultácii.

Abstrakt

Táto práca popisuje metódu optimálneho vykresľovania tieňov v real-time 3D aplikáciach a 3D video hrách. Popísaný algoritmus zlepšuje presnosť algoritmu Shadow Mapping, často používanej metódy, ktorá využíva textúru na ukladanie hĺbkovej informácie zo scény. Táto hĺbková informácia je následne použitá na zistenie, ktoré časti scény sú zatienené a ktoré nie.

Algorimtus vylepšujúci túto metódu sa nazýva Parallel-Split Shadow Mapping, ktorý používa viacero hĺbkových textúr. Pohľadový objem kamery sa rozdelí na niekoľko častí a pre každú z týchto častí sa ukladá hĺbková informácia. Tento prístup redukuje aliasing takto vykreslených tieňov na úkor zníženého výkonu. Vďaka tomuto prístupu sa ale tiene vykresľujú iba pre tie časti scény, ktoré sú vidno, pričom pri bežnom Shadow Mapping-u sa musia vykresliť pre celú scénu.

Kľúčové slová: OpenGL, Real-time Rendering, Shadow Mapping, Parallel-Split Shadow Mapping

Abstract

This thesis describes a method for optimized shadow rendering used in real-time 3D applications and 3D video games. The algorithm is a modified version of Shadow Mapping, a widely used shadowing technique that employs an additional pixel buffer to store depth information from the scene. This information is then used to determine which parts of the scene are lit and which are in shadow.

The improved algorithm for Shadow Mapping is named Parallel-Split Shadow Mapping, which uses multiple shadow maps. The camera's viewing frustum is split into several parts and one shadow map is rendered for each of these parts. This reduces the aliasing of the shadows at the cost of additional computational time. In addition, the shadows are synthesized every frame only for the parts of the scene that are inside the camera viewing volume, whereas in Standard Shadow Mapping, the shadows for the whole scene have to be synthesized.

Keywords: OpenGL, Real-time Rendering, Shadow Mapping, Parallel-Split Shadow Mapping

Contents

List of Abbreviations

OpenGL - Open Graphics Library
GLSL - OpenGL Shading Language
GPU - Graphics processing unit
CPU - Central processing unit
API - Application programming interface
GLU - OpenGL Utility Library
SM - Shadow Mapping
PSSM - Parallel-Split Shadow Mapping
PCF - Percentage-closer filtering
AABB - Axis-aligned bounding box
FOV - Field of view
FPS - Frames per second

List of Tables

List of Figures

List of Listings

Introduction

Real-time rendering is a crucial part of all modern video games, which are meant to be as realistic as possible. With increasing number of triangles rendered per frame and complex lighting calculations, optimization becomes very important. Shadows are indispensable in the complex lighting models, since they increase realism of the rendered images rapidly and provide the viewer with depth information.

Rendering shadows fast and efficient has been troubling developers for a long time and only two viable options emerged: Shadow Volumes and Shadow Mapping. Shadow Volumes (introduced in [Crow77]) are precise and are usually synthesized by rendering to an additional stencil buffer. The problem that emerges is that it requires to detect the silhouette of the shadow casters, which can be very costly for scenes with hundreds of thousands of triangles. Shadow Volumes will not be discussed in this thesis.

Shadow Mapping, on the other hand, can be done without silhouette detection and requires very little computation on the CPU. Crucial parts of the algorithm can be executed on the GPU, which is usually faster due to parallelization of the algorithms. The bottleneck is that the scene has to be rendered multiple times, with the minimum of 2 for simple Shadow Mapping. More complex Shadow Mapping techniques usually require more rendering passes but the generated results are preciser. Since the shadow maps are just 2D images with a certain resolution, they suffer from aliasing and can therefore produce unrealistic results.

Parallel-Split Shadow Mapping is an extension to Shadow Mapping. The

main idea is to split the camera viewing volume using split planes parallel to the near and the far plane and rendering one shadow map for each of these splits. This reduces shadow aliasing significantly, especially for a distant light source. We can then control the aliasing by moving the split planes accordingly.

Chapter 1 (Basic Terms) explains the basic terms needed to understand the concept of Shadow Mapping. These terms include camera representation in a 3D scene, the meaning of a depth buffer, rendering to texture and using shaders within OpenGL through the OpenGL Shading Language (GLSL).

Chapter 2 (Shadow Mapping) explains the basic Shadow Mapping algorithm, its limitations and problems.

Chapter 3 (Parallel-Split Shadow Mapping) explains a modification to Shadow Mapping which improves the precision and decreases the aliasing significantly by rendering multiple shadow maps for the viewing volume. The algorithm is derived from Shadow Mapping explained in Chapter 2.

Chapter 4 (Implementation) describes the implementation of rendering to texture, Shadow Mapping and Parallel-Split Shadow Mapping. The implementation is written in C++, using OpenGL for rendering and The OpenGL Shading Language for shaders.

In Chapter 5, we show some results and compare the results from the two techniques mentioned. We also compare the performance of both algorithms, and finally conclude discuss future work.

Chapter 1

Basic Terms

Before understanding Shadow Mapping, you will need to get familiar with certain basic terms.

1.1 OpenGL

In the implementation part of this thesis, we will be using the Open Graphics Library (OpenGL), which is the most widely used graphics library supported by almost all platforms. Since 1992, when OpenGL was introduced by Silliconn Graphics International (SGI), OpenGL has become a standard graphics application programming interface (API). It is a constantly evolving, easy to use, reliable and portable solution. Specifications and documentation for OpenGL can be found at [GLregistry].

One of the main purposes of OpenGL is to provide a hardware independent interface for using the graphics card to perform 2D and 3D rendering. Another is to provide as much functionality as possible on the target machine using OpenGL extensions, which can be platform-specific or graphics card vendor specific.

Usually, an OpenGL application creates a system-specific window and creates an OpenGL context that is interconnected with the window. Afterwards, the application calls OpenGL functions to render to the window and

display primitives like triangles, lines, etc. on the screen.

1.2 Camera and Frustum

Always when rendering an image, there is a camera that represents the viewer's position, facing and also some other parameters that affect the output image. Therefore, the camera is a set of parameters that specify where the camera is, which way it is facing, and most importantly which objects or parts of the scene will be seen by the camera.

A camera in a 3D world can have many representations. Every camera representation has the camera position, but other camera attributes may vary from implementation to implementation. The type we will explain is used the most.

1.2.1 Camera's Frustum

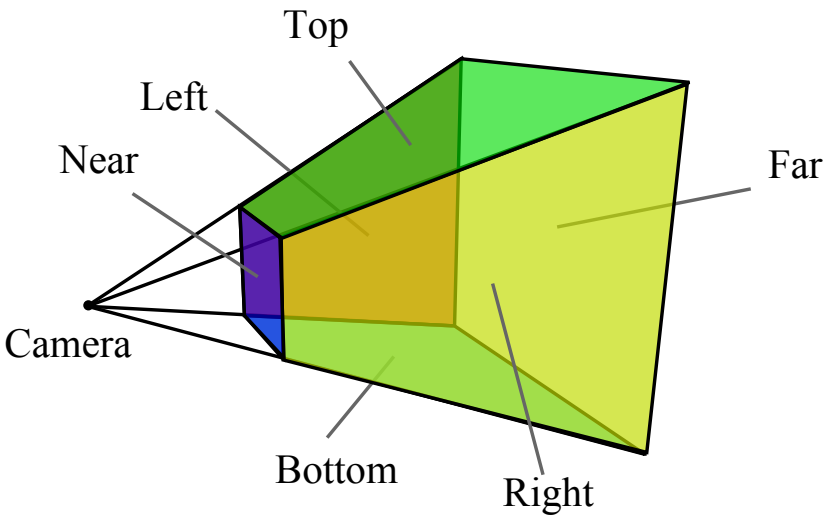


Figure 1.1: Typical camera perspective

Each camera can alternatively be defined by it's frustum. The camera's frustum is defined by six clipping planes (near, far, top, bottom, left, right),

as shown if Figure 1.1. If a point is outside the camera's frustum, that point will not be rendered when rendering using the camera, because the camera actually cannot see the point. Each of the clipping planes are in the form $ax + by + cz + d = 0$. The normal vector (a, b, c) of these planes is defined to point to the inside of the frustum. This representation (thanks to the normals pointing inside) offers a simple algorithm to determine if a point is inside the frustum, as shown in Listing 1.1.

```
1    point(x, y, z)
2    foreach( plane(a, b, c, d) in frustum )
3        if(a*x + b*y + c*z + d < 0)
4            return outside;
5    return inside;
```

Listing 1.1: Point in frustum test

The frustum can be recovered from a camera setup such as the one described in 1.2.2. It is done by creating a clip matrix from the matrices returned by `gluLookAt` and `gluPerspective`. From this clip matrix, the plane recovery is simply a matter of constructing a 4D vector for each plane from the columns of the clip matrix.

1.2.2 Typical Camera Setup

For an application programmer in OpenGL, there are a few ways to define a camera. The camera we will describe is set up by the functions `gluLookAt` and `gluPerspective`. Despite the fact that both of these functions are from the GLU package, they simply create a 4×4 transformation matrix from a few parameters. Therefore, anyone can write a replacement function without having to worry about the dependency on the package. These two functions can apply the camera, which means setting up the transformation matrices so that the rendered image will be from the camera's point of view.

The function `gluLookAt` takes 9 parameters that can be grouped into three 3D vectors. The first one being the camera's position, the second one the camera's target, meaning that it is the point that will be in the middle of

the rendered image. Finally, the third vector represents the up vector of the camera that tells us which direction is "up" for the camera. The up vector is usually set to $(0, 1, 0)$, resulting in objects that have higher y coordinate to be higher in the final rendered image.

The function `gluPerspective` takes 4 parameters: the field of view angle, the aspect ratio of the final image and the near and far values. It is closely described in Figure 1.2. The near and far values determine the extents of the camera's viewing volume. The near value specifies the near clipping plane, which causes all the objects that are closer to the camera than the near plane will not be rendered. And similarly, objects farther away from the camera than the far clipping plane will not be rendered as well.

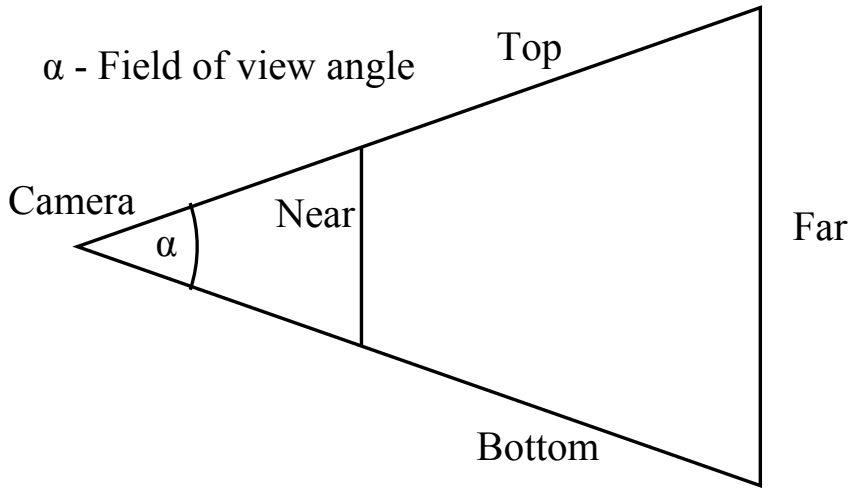


Figure 1.2: Camera perspective - side view

The aspect ratio should be $\frac{w}{h}$ where w is the width of the final rendered image and h is the height of the image. If set otherwise, the image will be fit to the $w \times h$ rectangle, resulting in incorrect scaling.

Another type of projection is an orthographic projection, in which all the planes of the camera's frustum are either orthogonal or parallel (parallel to each other are the pairs near-far, top-bottom and right-left. Such a projection is set in OpenGL using the `glOrtho` command, which has six parameters that

define the positions of the clipping planes.

Such a projection is used when rendering top views, side views, etc., when we want all the parallel lines to be parallel even when rendered. In Shadow Mapping, it is used for directional lights such as the sun, which is so far away that the clipping planes of the perspective projection would look much like an orthographic projection.

1.3 Rendering to Texture

Rendering to texture is a way to render to an off-screen buffer instead of the OpenGL window, which is usually the target for all rendering calls.

Shadow maps are dynamically created depth textures, meaning that they are rendered before applied. Therefore, we will need a possibility to render to a texture.

In OpenGL, there are these three options to render to a texture:

- **Copying pixels** - works on all hardware since it actually renders to the framebuffer and then copies the pixels (using `glCopyTexImage2D` or `glCopyTexSubImage2D`) from the framebuffer to a texture. It's the slowest possible solution, but also the most portable. The problem that occurs is that every time a texture is rendered, the framebuffer has to be cleared, which can be costly. Another problem is that the resolution of the target texture cannot exceed the resolution of the window.

- **P-buffers** - Are Windows-specific and require the `WGL_ARB_pbuffer` extension. No copying of pixels is done and it actually renders to a P-buffer, which can then be bound as a texture. Bounding a depth texture is available only through the `WGL_NV_render_depth_texture`, which is an NVIDIA specific extension. To use P-buffers on ATI and other video cards for rendering depth textures, a pixel copy has to occur after the rendering into the P-buffer is done. A P-buffer has it's own rendering context and it can be costly to switch between contexts, especially when done quite often.

- **Framebuffer Objects** - Are the newest and probably the best performance solution for rendering to a texture. Another framebuffer is created in the video memory and similar to a P-buffer, the framebuffer can be used as a texture. The problem with this implementation is that it requires newer graphics cards. It requires the `GL_ARB_framebuffer_object` extension and it's platform-independent in comparison to P-buffers.

We will provide an implementation of the first option in Chapter 4. The demo program is using pixel copy or P-buffers, depending on the extensions supported by the graphics card.

1.4 The OpenGL Shading Language

The OpenGL Shading Language(GLSL) is a language that allows OpenGL to execute user-defined per-vertex and per-pixel operations on the GPU when rendering primitives. GLSL is a language derived from C, enriched by additional keywords, specific types and operations. In [Rost06], authors provide every information you might need about GLSL as well as a nice palette of examples. The whole language specification for different language versions can be found at [GLregistry].

1.4.1 OpenGL Rendering Pipeline

To better understand how GLSL works, a better understanding of the OpenGL rendering pipeline is needed. An OpenGL program sends vertex information such as position, normal or texture coordinate to the GPU. The raw vertex data is processed and the result is a list of triangles with attributes assigned to each vertex.

A vertex shader is a replacement of the OpenGL's built-in per-vertex operations. After these operations (such as transforming the vertex into eye clip space or storing the color of the vertex) complete, rasterization occurs. It is a process of converting stored primitives in clip space into pixels and storing them in the framebuffer (which will then be shown on the screen).

Then, for each pixel, per-fragment (also known as per-pixel) operations occur. These operations include applying the right color of the pixel using material color, vertex color, lighting, textures, ...

A simplified rendering pipeline is shown in Figure 1.3. The Vertex Transformation field is replaced with a vertex shader and the Fragment Texturing and Coloring field is replaced with a fragment shader.

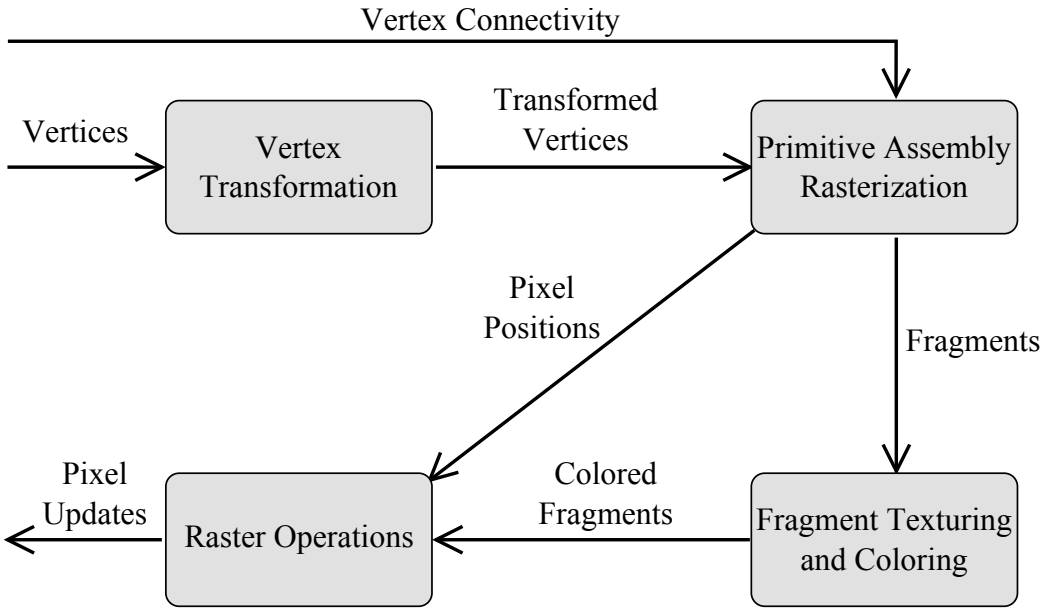


Figure 1.3: Simplified OpenGL Rendering Pipeline

1.4.2 Special Variables

Vertex shaders and fragment shaders need to communicate with each other. The output of a vertex shader can be the vertex color for each corner of a triangle. Since the application only specifies these corners, some kind of interpolation has to occur between the corners of the triangle to actually make a smooth gradient between the corners. `varying` variables allow that. A `varying` variable is set in a vertex shader (let it be the color of the vertex) and is read in a fragment shader. But the value read in the fragment shader

is always bilinearly interpolated between all three corners of the triangle. Meaning that if every corner of a triangle has a different color, then the resulting coloring of the triangle will be a smooth color gradient between the colors specified in the corners. The further the fragment is from a corner, the less color is applied from that specific corner.

Shaders also need to be controlled by the application. Telling the shader what color the lights are or how the vertices should be transformed cannot be done by sending additional vertex information. Therefore, `uniform` variables are present in both vertex and fragment shader to allow global input variables to be sent to the shaders. The application programmer only has to recognize the right `uniform` variable (access it by name) and can then set it's value using one of the `glUniform` functions. Setting an `uniform` variable will make all future render calls have the value specified by the programmer stored in the shader's variable. That way, the programmer can control the shader (such as disable texturing for objects that do not have texture coordinates or use normal mapping for objects that have a normal map).

The shader programmer can also define custom vertex attributes (using the `attribute` keyword) and send custom vertex data using the `glVertexAttrib` functions. These variables are accessible only by a vertex shader, since the fragment shader has nothing to do with triangle vertices, only with pixels.

1.4.3 Shader Example

Vertex and fragment shaders are usually coupled together. This is because of the shared `varying` and `uniform` variables. A simple vertex shader looks like the one in Listing 1.2.

```
1 void main()
2 {
3     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
4 }
```

Listing 1.2: Simple vertex shader

Each shader, be it vertex or fragment, has to have a `main` method, similar to a C executable program. The vertex shader in Listing 1.2 takes a built-in `uniform` variable that is a transformation matrix from world space to camera clip space. This matrix is simply the OpenGL modelview matrix multiplied with the OpenGL projection matrix. It transforms the built-in vertex attribute `gl_Vertex`, which actually contains the value specified by a program call to `glVertex`, and stores it into `gl_Position`, which is the output of the vertex shader (besides the `varying` variables that are set in the shader).

```
1 void main()
2 {
3     gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
4 }
```

Listing 1.3: Simple fragment shader

A hello-world-like fragment shader is shown in Listing 1.3. The only thing that happens in the shader is that the color of the fragment is set to red. This results in any geometry that is drawn to be completely red (if the clear color is set to a different color, you will be able to distinguish the background and the objects).

Shaders contain a number of built-in functions and types for simplifying certain operations. The types are vectors, matrices, samplers, etc. and the functions are usually basic operations with those types, such as matrix multiplication, linear interpolation, vector dot product, ... The samplers are used for sampling a texture in a shader.

The key function for us is `shadow2DProj(sampler2DShadow, coord)`. It samples a depth texture bound to a `sampler2DShadow` variable at the specified texture coordinate. It automatically performs depth comparison and therefore simplifies the shaders used for Shadow Mapping rapidly. The function returns a 4D vector, but only the first coordinate is interesting. The first coordinate will be 1 if the fragment's depth is greater than the sampled value from the depth map, 0 otherwise.

Shaders can do a lot of computation, not just simple math. An example of a per-pixel Phong shading is shown in Listing 1.4.

```
uniform sampler2D texture0; // Texture
varying vec4 color;         // Input color
varying vec2 texCoord;      // Input texture coordinate
varying vec3 position;      // Input position of point in 3D
void main() {
    vec4 outColor = texture2D(texture0, texCoord) * color;
    float d = distance(position, light[0].position);
    float att = 1.0 / (1.0 * d + 0.1 * d * d);
    vec3 N = normalize(normal);
    vec3 V = normalize(-position);
    vec3 L = normalize(light[0].position - position);
    vec3 R = normalize(reflect(L, N));
    float NdotL = max(dot(N, L), 0.0);
    if(NdotL > 0.0) {
        vec4 vAmb = att*light[0].ambient*outColor;
        vec4 vDif = att*light[0].diffuse*NdotL*outColor;
        float RdotV = max(dot(R, V), 0.0);
        vec4 vSpe = light[0].specular * pow(RdotV, material.
           shininess);
        outColor.rgb = vec3(vAmb + vDif + vSpe);
    }
    gl_FragColor = outColor;
}
```

Listing 1.4: Per-pixel Phong shading - Fragment Shader

Chapter 2

Shadow Mapping

Shadow Mapping was first introduced by Lance Williams in [Williams78]. The algorithm requires only an additional render pass for the whole scene geometry. Therefore it is relatively easy to implement when compared to shadow volumes and it fully utilizes the power of the graphics processor.

The algorithm works in multiple rendering passes, usually one for each light in the scene and some additional passes to render the final scene.

To simplify things, let's have just one light casting shadows. How do we determine which points in our scene are lit and which are shadowed? If we could extend the rays emitting from the light, only the closest point to the light (hit by a ray) will be lit by it. All other points will be shadowed by the object containing the closest point, as shown in Figure 2.1: The point P on the object has distance d from the light, and therefore the point Q, which is farther away on the line defined by the light's position and the position of P will be in shadow. We just have to do this for every single point that could be lit. A straightforward approach would be very costly, but thankfully the idea of Shadow Mapping will reduce the computation time significantly.

By rendering the scene from the light's point of view and storing the rendered depth values into a depth texture (called the shadow map) from that render pass, we will acquire the distances of all the points that are lit by the light. The final rendering pass would only need to compare the right

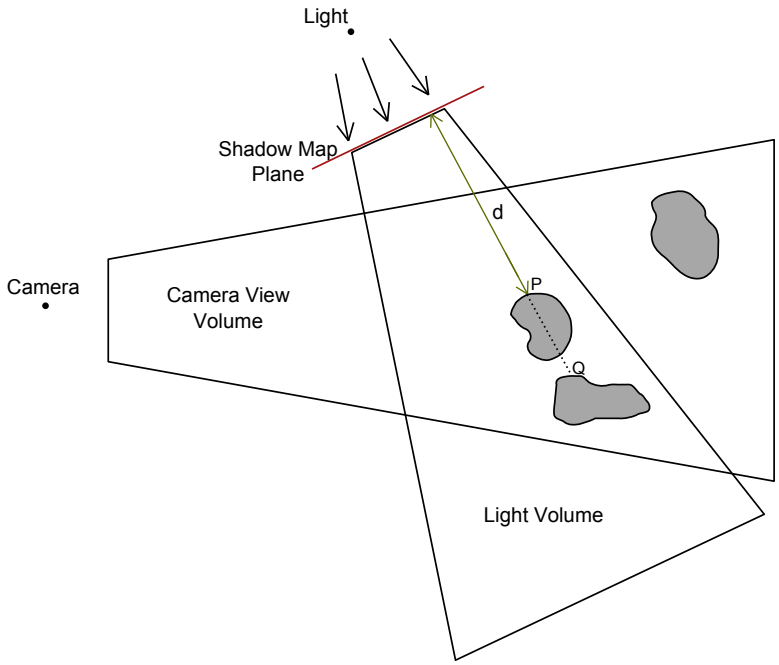


Figure 2.1: Shadow Mapping geometry

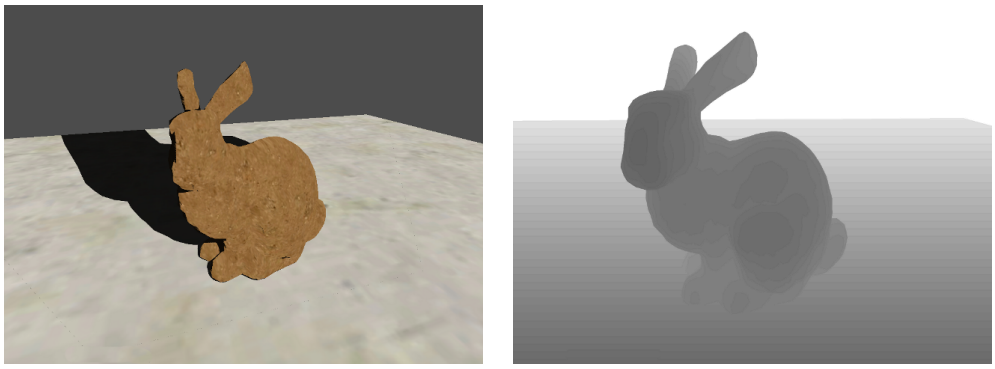
depth value from our stored shadow map with the point that is currently being rendered. If the distance of the point is almost exactly the value stored in the shadow map, then this point is lit by the light. If the distance is greater, it is in shadow.

2.1 First pass

In the first pass, the scene is rendered from the light's point of view. We only need the depth values from this render pass, therefore no special computations for lighting or other effects are needed (only the vertex positions should be taken into account, ignoring vertex color and textures). We will save the depth values into a texture (or render into it directly).

A depth texture can be interpreted as a grayscale image, where darker color means higher depth (or higher distance from the camera position).

In Figure 2.2, you can see a scene with shadows synthesized using Shadow Mapping, and the shadow map for that scene represented as a grayscale image.



(a) Scene with shadows

(b) Scene's shadow map

Figure 2.2: Shadow Mapping output

2.2 Second pass

We have a texture with depth values (our shadow map) that tell us how far those pixels are from the light. But when pixels are rendered during the second pass, the algorithm has to have a position in the shadow map to read the values from. Therefore, we need to prepare a transformation matrix to transform the pixels from object space to the light's clip space. We had to use this matrix in the first pass when rendering from the light's point of view. Therefore, we only need to save the matrix during the first pass and then use it during the second pass.

Let us consider these 4×4 transformation matrices: V_L will be the light's view matrix, P_L will be the light's projection matrix. Then the matrix transforming the coordinates from object space to the light's clip space will be

$$T = P_L \cdot V_L.$$

Consider a point in object space $p = (x, y, z)$. Applying the matrix T to it would represent

$$T \cdot p = (P_L \cdot V_L) \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = P_L \cdot q = r = \begin{pmatrix} x_r \\ y_r \\ z_r \\ 1 \end{pmatrix}.$$

Where q is p transformed into the light's eye space and r is q projected into the light's clip space. That means that r(or (x_r, y_r)) represents the position of the pixel to be queried in the shadow map.

Calculation of this matrix is done on the CPU since it only has to be done once per light. Afterwards, it will be sent to the shader that renders the second pass.

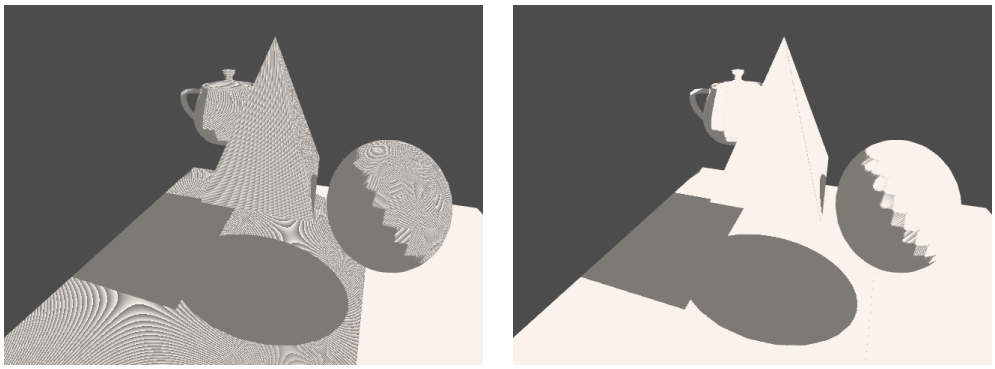
We have the shadow map and the transformation matrix prepared. The second pass is not that different from a normal scene rendering. It only has to compare the depth values for each pixel it renders and either perform light computation (if it is lit by the light) or apply the shadow color. The shadow color doesn't neccessarily have to be black and in the shader implementation proposed, it can be either a fixed color or a simple "absence of light", meaning the light won't contribute to the pixel's output color.

If there were more lights in the scene, it becomes more complex. Without shaders, it would require one final pass per light, resulting in a total of $2n$ passes, where n is the number of lights. We will not discuss this option here, since shaders can handle second passes for each light in one single pass, resulting in a total of $n + 1$ passes.

2.3 Problems and Limitations

There are certain problems that occur when using this technique. One of them is the floating point number precision problem. When the depth is stored in the shadow map and the final scene is rendered, all the rendered pixels suffer from floating point precision problems. This results in so-called

Moire patterns, which are unwanted artifacts shown in Figure 2.3. The Moire patterns are caused by the fact that two lit pixels near to each other might (due to precision problems) be evaluated so, that one of them is lit and the other is not. To avoid this, we will need to add a small bias to the depth values in the shadow map. That way, values queried from the shadow map will be slightly higher than the actual distance from the light. OpenGL has a function for this called `glPolygonOffset`.



(a) Without polygon offset, unacceptable (b) With polygon offset, no moire patterns

Figure 2.3: Polygon offset

Another limitation is the resolution of the shadow map. The higher the resolution, the preciser the shadow map. But increasing the shadow map size decreases performance rapidly and the increased precision is not usually worth it. It depends on the hardware used and the complexity of the rendered scene as well as a lot of other aspects, it is usually good to experiment a bit and find the right resolution that still doesn't decrease the performance too much but is large enough. To improve it further, filtering of the shadow map is present in most algorithms.

If we were to cover an entire scene with shadow-mapped shadows, we would have to use 6 different shadow maps (like a cube map), since the light viewing volume cannot cover an entire scene. Imagine a room with a point light in the exact middle, meaning that shadows can be cast in all ways. Certain Shadow Mapping techniques get rid of this problem, and using

directional lights avoids this problem.

2.3.1 Filtering

There are many different techniques how to filter a shadow map. A nicely filtered shadow map could be used to produce soft shadows. A straightforward approach is to read multiple shadow map values when doing the shadow lookup in the fragment shader. This technique is called percentage-closer filtering (PCF). It's main drawback is that multiple texture lookups decrease the performance rapidly, as seen in Section 5.2.

An additional render step could occur between the first and the second pass to filter the depth texture that was the output of the first pass. Using a gaussian blur or a different convolution could produce quite good soft shadows, but there are lots of things that need to be taken into account. In [Donnelly06], authors introduced a way of rendering soft shadows without the need to actually filter the shadow map. Also, the authors of [Lloyd08] used reparametrization and analysis of shadow map aliasing to reduce aliasing.

Chapter 3

Parallel-Split Shadow Mapping

The simple Shadow Mapping algorithm is not precise enough. Therefore, many algorithms that improve the shadow map precision are being used. The shadow map size is limited by hardware (most graphic cards cannot handle texture with resolution higher than 4096×4096). Therefore, the idea of using multiple shadow maps at once was introduced. Also, when using the simple Shadow Mapping for a large scene such as a city, the shadow map has to cover the whole city and thus the coverage of the entire scene by this one shadow map will result in strongly decreased precision.

Parallel-Split Shadow Mapping is one of the techniques that uses multiple shadow maps for one light. The main idea is to split the view frustum with parallel splits (parallel to the near and far plane) into smaller frusta and then synthesizing one shadow map per frustum. It was first introduced in [Zhang06] and further examined in GPU Gems 3 [Nguyen07], in Chapter 10, which is dedicated to PSSM.

How the algorithm works is shown in Figure 3.1. There are 4 splits, meaning that for each of these parts, a single shadow map is created. Point P on the first object is lit, this information is retrievedby looking up the value in shadow map for the third split. Point Q is in shadow, because it has the same lookup position in shadow map as point P, but is farther away. If point Q was in the last subfrustum, we would be querying if it

is shadowed or not from the last shadow map. All the shadow maps have the same resolution, therefore the near and far planes of the split parts are the controlling elements and by altering them, we adjust the aliasing. The point is to use preciser shadow maps for objects that are closer to the viewer (since these objects will be larger in the final image) and thus improving the precision.

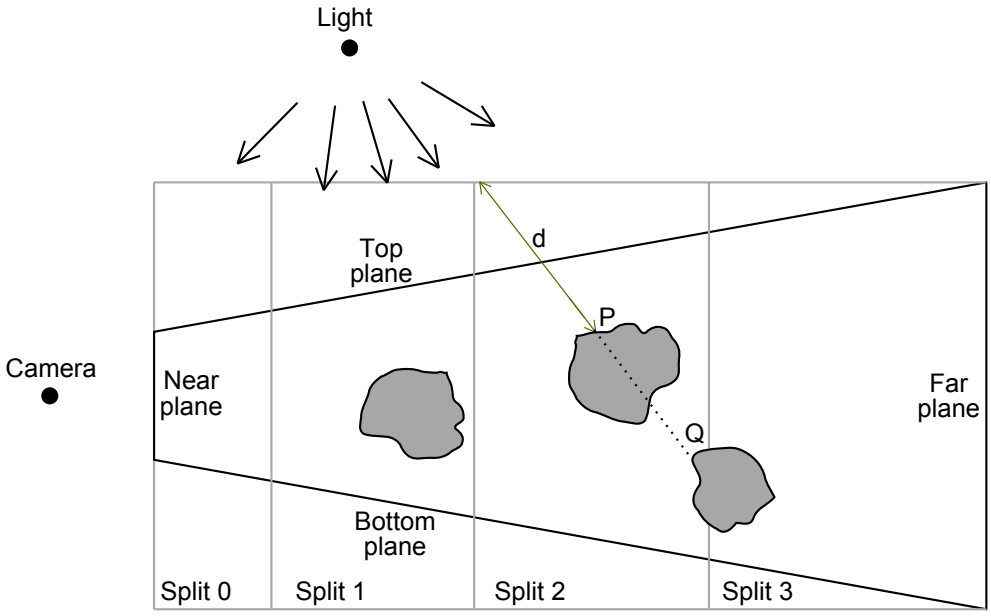


Figure 3.1: Parallel-Split Shadow Maps

The algorithm can be divided into these steps:

1. Split the view frustum into n parts depending on the split scheme
2. Calculate the light's view matrix for each of these parts
3. Generate shadow maps (with resolution $w \times h$ for each of the splits) by rendering the scene from the light's point of view
4. Render the final scene with shadows applied, choose which shadow map to use based on the depth of the pixel being rendered

3.1 Step 1 - Splitting the View Frustum

The problem that emerges is how to place the splitting planes. Analyzing how Shadow Map Aliasing depends on the distance of rendered objects (in the final scene render, when applying the shadow map) will help us find the optimal split positions.

When rendering the scene from a certain point of view, objects that are farther away from that point will be smaller in the output image. Meaning that when we render the scene from the light's point of view (as described in Chapter 2), objects that are far away from the light will be very small, covering only a few pixels in the result image. But when the camera is close to this object when rendered with shadows applied, the actual objects are large but the pixels in the shadow map representing their distance from light cover only a small area. This results in strong aliasing artifacts when performing simple Shadow Mapping.

The first thing that comes to mind would be to split the frustum uniformly. The position of the split plane i would then be

$$C_i^{uni} = n + (f - n) * \frac{i}{m}.$$

where n is the near plane, f the far plane and m the number of splits. This is called the uniform split scheme.

But it does not describe how the pixel count decreases with increasing distance from light. The decrease is much more similar to an exponential function rather than a simple linear function. This means that the uniform scheme will cause over-sampling in parts that are farther and under-sampling for objects that are close to the camera. All this happens because the first sub-frustum of the camera frustum is too large and the last sub-frustum is too small. A logarithmic split scheme seems much more appropriate:

$$C_i^{log} = n + \left(\frac{f}{n}\right)^{\frac{i}{m}}.$$

This scheme is much more fitting but it causes opposite problems to the problems caused by the uniform split scheme. Over-sampling occurs for close

objects and under-sampling for distant objects. Therefore, a combination of the logarithmic and the uniform split scheme should be sufficent. The final scheme looks like

$$C_i = \lambda \cdot C_i^{log} + (1 - \lambda) \cdot C_i^{uni}.$$

Where $0 \leq \lambda \leq 1$. The parameter λ allows us to control the sampling efficiently. In our experiments, $\lambda = 0.5$ seemed like the best compromise for close as well as distant objects. The parameter could also be changing dynamically according to scene conditions, but it would require some scene analysis.

3.2 Step 2 - Generating Shadow Maps

Before the final scene render can occur, one shadow map for each of the splits needs to be rendered. It is simply a matter of setting the right light view volume for each part and render a shadow map the same way the simple Shadow Mapping algorithm does it. That means we need one view-projection matrix per split for the light, looking from the light's position on the split.

There are two ways this can be done. The Scene-Independent projection constructs an axis-aligned bounding box (AABB) for each subfrustum and makes a view-projection matrix that is looking exactly at the AABB. Afterwards, the near and far plane of the light viewing frustum can be altered to encapsulate the subfrustum as tight as possible. But the near plane cannot be moved too far away from the light, because there can be some objects casting shadows between the light and the subfrustum. As shown in Figure 3.2, if we move the near plane of the light viewing frustum too close, shadow values for object P would not be stored in the shadow map and therefore, the object would not be casting shadows.

The Scene-Dependent projection takes all the objects in the subfrustum and calculates an AABB that contains all the objects. This may result into a smaller AABB than the Scene-Independent projection and thus using the same resolution shadow map for smaller space, resulting in increased shadow

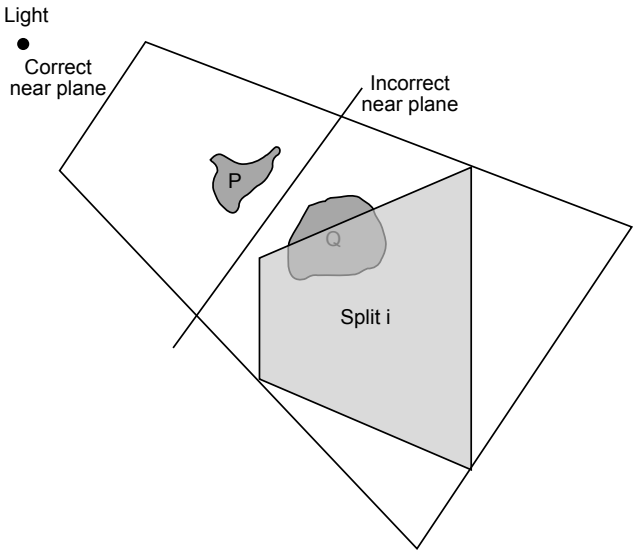


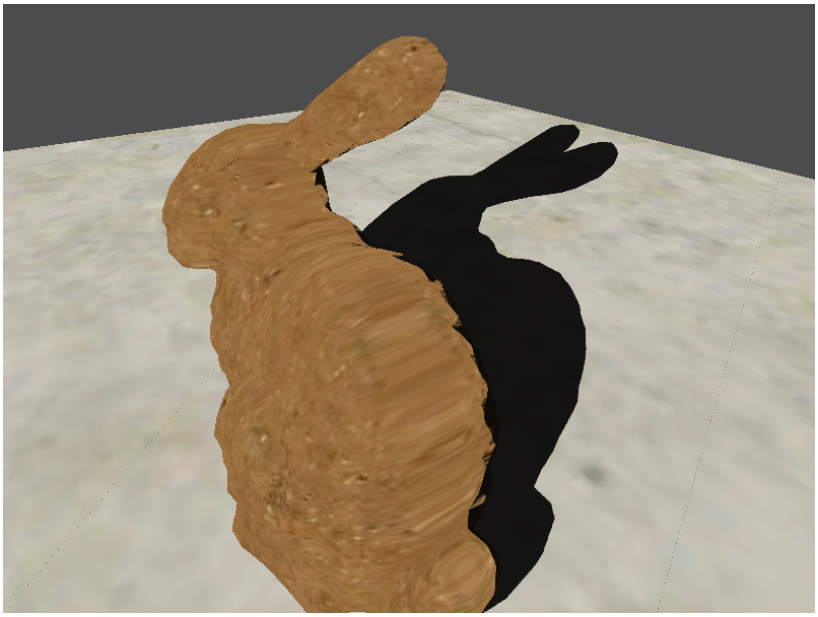
Figure 3.2: PSSM - Light's viewing frustum

map precision. When the light's viewing frustum is altered (to encapsulate the AABB of the split), the near plane of the frustum must not be moved, since there could be objects casting shadows between the light and the sub-frustum of the split.

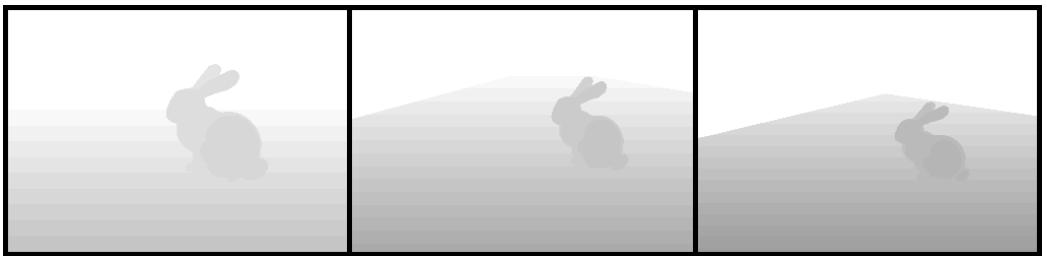
In Figure 3.3, you can see a scene rendered with PSSM shadows and the shadow maps for that scene. Notice that the Bunny is in each of those shadow maps, that is because of the near plane positioning, since the bunny's shadow actually lies in all of the splits. There are two sets of maps shown, one using the Scene-Independent projection and one using the Scene-Dependent projection where the Bunny is casting shadows and the ground and the Bunny are recieving shadows.

3.3 Step 3 - Rendering the scene

Finally, all the shadow maps are applied to the scene when doing the final render. It is very similar to the simple Shadow Mapping, but each of the splits has it's own light view-projection matrix and it's own shadow map.



(a) Scene with PSSM shadows



(b) The shadow maps - Scene Independent projection

(c) The shadow maps - Scene dependent projection

Figure 3.3: Parallel-Split Shadow Mapping output

The scene can be synthesized in multiple passes (one for each split), resulting in decreased performance. Doing one final pass for each split would result in

$2n$ rendering passes (one to generate the shadow map, one to apply it).

Using a complex shader can replace the final renders with one single render. The fragment shader loops through all the shadow maps (starting with number 0, the closest to the camera) and tests if the fragment's depth is in the range for the current shadow map. If it is not, it just continues to the next map. If it is, it does the shadow lookup, applies the shadow color depending on the lookup result and breaks the loop. This way, the number of rendering passes is decreased to $n + 1$.

3.4 Problems and Limitations

Despite the fact that the algorithm proposed in this chapter improves shadow map precision significantly, it has a few limitations. The shadow map aliasing is reduced to an acceptable level and only small parts of the shadow maps are not being used.

The first drawback is the fact that everytime the camera moves, the shadows need to be recalculated. In the simple Shadow Mapping algorithm, a static scene with a static light would require the shadow map to be rendered only once and then applied when rendering. The gained performance is usually negligible, since most scenes are not static.

There are certain unwanted artifacts appearing when using the parallel-split approach. When moving the camera quickly, the edges of the shadows might "shake". This happens due to the constant changes of the actual shadow maps, which occur when cropping the light's viewing volume for each split part. The shaking is usually not noticable, but could be a major distraction when using smaller resolution shadow maps. Applying a blur filter to the shadow map might solve the problem and even provide soft shadows.

The second drawback is that each of the shadow maps are rendered to a different render target. Switching between render targets might be costly, especially when using pixel copying due to the need to clear the window's framebuffer everytime the rendering to the texture is complete (see Section

1.3). One solution would be to use different render target technique such as the already mentioned P-buffers or framebuffer objects.

Another solution would be to pack the rendered shadow maps into one texture, placing the separate shadow maps next to each other. It would require more complex shaders, shadow map coordinate calculation etc., but the gained performance might be worth it. On the other hand, we would once again be limited to the maximum texture size.

3.5 Further Optimizations

It is possible to further reduce the number of rendering passes to 2 using rendering to a texture array in the first pass and thus generating all the shadow maps in one pass. Another approach with only 2 passes is using geometry shader cloning. For more information about these approaches in DirectX 10 and higher, see Chapter 10 in [Nguyen07].

In OpenGL, it is possible to render to multiple textures in one rendering pass when using the Framebuffer Objects for rendering to textures. A Framebuffer Object can have different textures attached to each of the Framebuffer's `GL_COLOR_ATTACHMENT*_EXT`. In this kind of setup, the output of a fragment shader is not the variable `gl_FragColor` but an array of output vectors named `gl_FragData`. Setting the value of `gl_FragData[i]` sets the pixel of the texture attached as `GL_COLOR_ATTACHMENTi_EXT`.

We will not be able to avoid the geometry shader, since the fragment shader is only able to store values to each texture on one fixed coordinate. Despite the `gl_FragData` variable, the output values written are always on the same (x, y) coordinate in each of the attached textures.

Geometry shaders are available through the `GL_EXT_geometry_shader4` extension in OpenGL. Every triangle in the scene should be cloned $n-1$ times, resulting in one triangle copy per split, and then rendered to the right depth texture. Access to shadow maps will be through the color attachment to the Framebuffer Objects. This way, the total number of rendering passes

becomes 2 for one light.

It would be possible to synthesize shadows for multiple lights (and multiple splits) in one rendering pass. It would require a similar approach, but the number of shadow maps/splits/lights will be limited by the maximum number of attachable textures to a Framebuffer Object(less than 16 even for newer video cards). Since Parallel-Split Shadow Mapping works best for directional lights such as sunlight, more lights will not usually be needed.

Chapter 4

Implementation

With the theory explained, we will now provide an implementation of the algorithms mentioned. This includes rendering to a texture, which is a crucial part of the algorithms, since we could not continue without it. We also propose an interface to hide different implementations of the rendering to texture techniques. As next, we will show an implementation of the standard Shadow Mapping, packed into a reusable, easy-to-use class. In the final section, we will extend the Shadow Mapping class to incorporate Parallel-Split Shadow Mapping.

4.1 Rendering to Texture

We are looking for a unified interface that would encapsulate all the methods that allow rendering to a texture. We called the interface `IRenderTarget`. Specific implementations of the rendering to texture will be inherited from this interface. The `IRenderTarget` interface will have the methods shown in Listing 4.1.

The function `FetchTexture` binds the already rendered texture to a texture layer. Usually, this just calls `glBindTexture` with a previously created texture. When a texture is bound, it should not be rendered into, therefore we also incorporate a `ReleaseTexture` function, which releases a previously

fetched texture. The releasing of the fetched texture is not neccessary for certain implementations (e.g. when only copying pixels to a texture from the framebuffer, releasing a texture becomes unneccessary).

```
1   class IRenderTarget
2   {
3   public:
4       uint32 GetWidth();
5       uint32 GetHeight();
6       virtual void FetchTexture(uint32 layer) = 0;
7       virtual void ReleaseTexture() = 0;
8       virtual void MakeCurrent() = 0;
9       virtual void OnFinishRendering() {}
10      RenderTargetFormat GetFormat() { return format; }
11  protected:
12      RenderTargetFormat format;
13  };
```

Listing 4.1: `IRenderTarget` interface

The function `MakeCurrent` makes the texture the current render target, resulting in all drawing operations called after calling `MakeCurrent` to be executed in our render target (and therefore rendering to the texture).

Finally, the `OnFinishRendering` function is called whenever the active render target becomes inactive (being replaced by another render target that makes a call to `MakeCurrent`, where the render target can be the window's framebuffer).

We will now provide an implementation of the copying pixels rendering to texture. The class `CCopyPixels` will implement the `IRenderTarget` interface. The constructor prepares a new texture with certain width and height as well as a desired pixel format. A texture in OpenGL can either have color channels or depth channel, not both at one time. Therefore, if the pixel format has both color bits and depth bits, 2 different textures need to be created (one color and one depth texture). Then, the `FetchTexture` function has to have two versions, one for each of the textures used.

The destructor only deletes these two textures (if any of them was cre-

ated). The `MakeCurrent` function saves the values of the current window viewport (because it should be restored once the rendering to texture is finished) and sets the viewport to the size of the texture. The problem that occurs here is that the size of the rendered texture cannot exceed window resolution, it would result in errorneous results in the parts of the texture that are not inside window bounds, because OpenGL would not be able to render in those parts since the viewport would have to be bigger than the window itself.

`MakeCurrent` also has to clear the framebuffer using `glClear`, there would otherwise be artifacts from previous rendering. The `OnFinishRendering` method should be executed as well for the current render target to allow it to finish it's current rendering and restore values it has changed. This step is crucial, because switching between different types of render targets (for example having one P-buffer and one copying pixels render target) can cause a lot of errors.

The function `OnFinishRendering` has to actually copy the pixels rendered from the window's framebuffer to the texture. It is only a matter of calling `glCopyTexImage2D` accordingly (and for both the color and the depth textures, of course). It should also clear the framebuffer, otherwise rendering to the window afterwards would have problems with kept depth values from the previous render. The depth test would simply read the values that remained in the framebuffer and therefore would most likely produce errorneous images. Restoring of the viewport is also crucial, as well as storing it when `MakeCurrent` is called.

`FetchTexture` simply binds the texture(s) to the right texture units. `ReleaseTexture` can remain empty, since no releasing of textures needs to occur.

4.2 Shadow Mapping

There are lots of different implementations of simple Shadow Mapping. One of them even uses 3 rendering passes ([PaulsProjects]) for one light Shadow Mapping. We have chosen an implementation in OpenGL and OpenGL Shad-

ing Language for shaders. It requires only the mentioned $n + 1$ passes for n lights. We create a class implementing a simple interface, so that anyone can use it without even knowing how it works:

```
1   class CShadowMapping {
2   public:
3       CShadowMapping(uint32 depthBits, uint32 w, uint32 h);
4       ~CShadowMapping();
5       void StartRendering(const CLight& l);
6       void EndRendering();
7       void FetchTexture(uint32 layer, bool fetchForRender);
8       void ReleaseTexture();
9   private:
10      Shader* plainShader;
11      Shader* shadowMapShader;
12      IRenderTarget* renderTarget;
13      mat4 lightMatrix;
14  };
```

Listing 4.2: CShadowMapping class

What most programmers want is an easy-to-use interface for working with a class. Such usage is shown in Listing 4.3.

```
1   CShadowMapping sm(24, 2048, 2048);
2   void RenderScene() {
3       CLight light(...);
4       // first pass, render shadow map
5       sm.StartRendering(light);
6           scene.Render();
7       sm.EndRendering();
8
9       // second pass, render scene with shadows
10      sm.FetchTexture(1, false);
11          scene.Render();
12      sm.ReleaseTexture();
13  }
```

Listing 4.3: `CShadowMapping` usage

The class `CLight` contains light information that we will need (specifically only position and direction is required, the color values are not important for creating the shadow map). The light's position will be used when storing the `lightMatrix` variable and also during the first pass.

In Listing 4.3, an instance of our class is created and a 2048×2048 shadow map with 24-bit depth buffer is prepared. Then, in the function `RenderScene`, which is called once every frame, we have a light with certain position and direciton. We use this light's values to set up the light view and projection matrix and then we render the whole scene. We are using a plain shader that will ignore colors or lighting computation, since we need only the depth values for each pixel.

Then comes the second pass, where the shadow map is bound to the second (the first has number 0) texture unit. It also selects the shadow map shader for rendering, which contains the depth comparisons as well as color and light computation, since it's the final pass. Afterwards, depending on the implementation, the shadow map might need to be released.

4.2.1 Core functions

First of all, we need an implementation of the `IRenderTarget` interface. We won't be paying attention to that since it can be implemented in many ways, one of them described in 4.1. In the class constructor, the `renderTarget` is prepared with parameters as specified and the shaders are loaded. The `renderTarget` should have 0 color bits as well as stencil bits to save video memory.

The class destructor just deletes what the constructor has allocated. The most important functions are `StartRendering` and `EndRendering`. A simplified implementation is shown in Listing 4.4 and Listing 4.5.

`StartRendering` first switches to our render target, so we can render the shadow map into it as well as selecting the `plainShader` for rendering. Then, we save the modelview and projection matrices(only to ensure that calling `StartRendering` and `EndRendering` won't alter them). Afterwards, we set the

polygon offset to avoid floating point precision problems. And then we just set the projection and modelview matrix to look from the light in the light's direction. We also store the light matrix for further computation (will be needed to calculate T (see Section 2.2)).

```
1  void CShadowMapping::StartRendering(const Light& l) {
2      MakeCurrent(renderTarget);  plainShader.Use();
3      // Save modelview and projection matrix here
4      // Enable polygon offset
5      glPolygonOffset(2.0f, 1.0f); glEnable(
           GL_POLYGON_OFFSET_FILL);
6      SetProjectionMatrix(Perspective(45.0, 1.0, 1.0, 50.0));
7      SetViewMatrix(LookAt(l.pos, l.dir, vec3(0, 1, 0)));
8
9      // Save the light matrix for the final render
10     lightMatrix = GetProjectionMatrix() * GetViewMatrix();
11 }
```

Listing 4.4: `StartRendering` function

```
1  void CShadowMapping::EndRendering() {
2      glDisable(GL_POLYGON_OFFSET_FILL);
3      // Load saved matrices from StartRendering
4      plainShader.BackToFixedFunctionality();
5      MakeCurrent(window);
6  }
```

Listing 4.5: `EndRendering` function

The perspective projection has Field of view (FOV) set to 45° and aspect ratio to 1 since it's the best setting for saving precision. The near and far plane can be adjusted depending on the scene or what the user requires from the algorithm. By setting the far plane too close to the near plane, certain parts of the shadow map might no be rendered correctly (since what is about to be rendered will be behind the far plane), and setting the near plane to a number close to 0 will reduce the precision of the depth values. These parameters should always be chosen wisely.

`EndRendering` simply returns to the state before `StartRendering` was called and makes the current render target the window.

`FetchTexture` calculates our matrix T and sends it to the shader, which will apply the transformations for rendered primitives to enable the application of the shadow map. An implementation is shown in Listing 4.6.

```
1  renderTarget->FetchTexture(layer);
2  if(!fetchForRender) {
3      shadowMapShader.Use();
4      shadowMapShader.SetUniform("shadow", (int32)layer);
5  }
6  if(fetchForRender)
7      return;
8  float32 bias[16] = {
9      0.5f, 0.0f, 0.0f, 0.0f,
10     0.0f, 0.5f, 0.0f, 0.0f,
11     0.0f, 0.0f, 0.5f, 0.0f,
12     0.5f, 0.5f, 0.5f, 1.0f };
13 mat4 VcInv = GetViewMatrix().Invert();
14 mat4 T = mat4(bias) * lightMatrix * VcInv;
15 shadowMapShader.SetUniform("shadowMatrix", T);
```

Listing 4.6: `FetchTexture` function

Lines 1, ..., 5 set up the texture specific parameters and prepare the shader for Shadow Mapping. The remaining part of the function calculates the matrix T and sends it to the shader. The bias matrix just maps the corners of the shadow map from (0..1, 0..1) to (-1..1, -1..1) and thereby stretching the shadow map to cover the clip space. If we omitted the bias matrix, only the top right quarter of the output image would be affected by the shadow map, and thus would be errorneous.

If the `fetchForRender` flag is `true`, the texture (shadow map) will be fetched for rendering, meaning it can be rendered as a normal grayscale image where the lightness of the pixel represents the depth value.

The `ReleaseTexture` function is usually not needed and strongly depends on the implementation of the `IRenderTarget`.

4.2.2 Shaders

Now we will clarify what happens on the GPU side of the algorithm. The `plainShader` is a very simple "hello world - like" shader that only renders all primitives white. It is only used for rendering the shadow map, being otherwise quite useless.

```
1  // vertex shader
2  void main()
3  {
4      gl_Position = ftransform();
5  }
6
7  // fragment shader
8  void main()
9  {
10     gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
11 }
```

Listing 4.7: Plain shader

The `shadowMapShader`, on the other hand, is much more complex. The actual depth comparison occurs here and all lighting computation as well (which we can skip).

The vertex shader in Listing 4.8 does the following: Sets the `shadowCoord` variable, which will be the lookup position in the shadow map for the vertex. By taking a closer look at line 6, this happens:

$$r = \begin{pmatrix} x_r \\ y_r \\ z_r \\ w_r \end{pmatrix} = T \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

where r is the `shadowCoord` variable. We transform the vertex position in object space into the light's clip space and therefore resulting in the shadow map coordinates for this vertex.

Lines 7 and 8 of Listing 4.8 simply prepare the texture coordinates and place the vertex.

```
uniform mat4 shadowMatrix;
varying vec2 texCoord;
varying vec4 shadowCoord;
void main()
{
    shadowCoord = shadowMatrix * gl_Vertex;
    texCoord = (gl_TextureMatrix[0] * glMultiTexCoord0).st;
    gl_Position = ftransform();
}
```

Listing 4.8: Shadow Mapping Vertex Shader

The fragment shader is a bit more complex, it has to compare the value in the shadow map and depending on the return value to either let the fragment as it is or apply the shadow color to it.

```
uniform vec4 shadowColor;
uniform sampler2D texture;
uniform sampler2DShadow shadowMap;
varying vec2 texCoord;
varying vec4 shadowCoord;
void main()
{
    vec4 output = gl_Color * texture2D(texture, texCoord);
    float val = shadow2DProj(shadow, shadowCoord).r;
    vec3 cp = shadowCoord.xyz / shadowCoord.w;
    if(cp.x >= 0.0 && cp.x <= 1.0)
    if(cp.y >= 0.0 && cp.y <= 1.0)
    if(shadowCoord.w > 0.0)
        output.rgb = mix(output.rgb, shadowColor.rgb,
                shadowColor.a * (1.0 - val));
    gl_FragColor = output;
}
```

Listing 4.9: Shadow Mapping Fragment Shader

First, as Listing 4.9 shows, we calculate the lit value of the pixel by combining the texture in the first texture unit and the pixel's base color. The resulting color is then stored to `output`. Afterwards we use the built-in shadow comparison function `shadow2DProj` that performs the lookup of the shadow map.

`shadow2DProj` takes 2 parameters: `shadow` which is a binding of the shadow map and `shadowCoord` which specifies where to look. The result is a 4D vector, but only the first value (the r component) is important, since our depth texture has only one color channel. The result is either 0 or 1, telling us if the pixel is shadowed or lit, respectively. As next, the normalized coordinates in the shadow map need to be compared whether they are in range [0, 1], because the 2D coordinate (0, 0) is one of the corners of the shadow map and (1, 1) is the opposite corner. Therefore, values that are out of range cannot possibly contain useful data.

Finally, depending on how the 4D vector `shadowColor` is specified, the color is mixed. If the alpha component of `shadowColor` is 0, the shadow's color will not have any effect, and if the alpha is 1, the shadow's color will completely replace the color of pixels that are shadowed. That all is done by the `mix` function.

4.2.3 Multiple lights

The implementation we showed only produces shadows for one light. Producing shadows for multiple lights is possible with only a few minor changes to the shaders and the program. The vertex shader has to recieve the light view matrix for each of the lights and transform the position into the each light's clip space. Meaning that we will need a `varying` shadow coordinate variable for each light.

There has to be one texture sampler per light as well, since we will need to access multiple shadow maps in the fragment shader. Then, depending on how the shadows are actually displayed (either "absence of light" or applied color), we apply all the shadow maps at once. There are no limitations and no problems occur when combining shadow maps this way, therefore it is relatively easy to implement and the results are promising.

4.3 Parallel-Split Shadow Mapping

The implementation will be very similar to the simple Shadow Mapping in Section 4.2, using OpenGL and GLSL. The goal is, once again, to create a class that someone could use without knowing what happens inside the implementation. The interface class will be very similar, but it will contain certain changes. There will be the need to determine how many splits there actually are, and also how many should be used (so we can change the number of splits while running the application). It should also provide access to the alias-controlling variable λ, if the programmer would want to attach some aliasing adjusting. The proposed class interface is shown in Listing 4.10.

```
1  class CPSSM
2  {
3  public:
4      CPSSM(uint32 maxSplit, uint32 dBits, uint32 w, uint32 h);
5      void StartRendering(CLight l, uint32 s, CCamera c);
6      void EndRendering();
7      void FetchTexture(uint32 i, uint32 l, bool fetchForRender
           );
8      void ReleaseTexture(uint32 i);
9      void FetchTextures(uint32 startLayer);
10     void ReleaseTextures();
11     uint32 activeSplits;
12     float32 lambda;
13 private:
14     array<IRenderTarget*> renderTargets;
15     Shader* plainShader;
16     Shader* pssmShader;
17 };
```

Listing 4.10: `CPSSM` class interface

The constructor only creates n render targets, where n is the maximum number of shadow maps used. It also loads the shaders used. The `plainShader` stays the same as in the simple Shadow Mapping algorithm. The destructor cleans up all these. `StartRendering` and it's pair function `EndRendering` will

be used to synthesize each shadow map. Finally, `FetchTextures` binds those shadow maps to different texture units so that they can be applied in the final render. `ReleaseTextures` simply releases previously fetched textures (if neccessary).

```
1  ...
2  pssm.activeSplits = 4;
3  for(int i = 0; i < pssm.activeSplits; i++) {
4      pssm.StartRendering(light, i, camera);
5      scene.Render();
6      pssm.EndRendering();
7  }
8  FetchTextures(2);
9  scene.Render();
10 ReleaseTextures();
11 ...
```

Listing 4.11: `CPSSM` usage

As you can see in Listing 4.11, the interface remains clean and easy to use, very similar to the one used in simple Shadow Mapping. At first, the number of splits used is set. Afterwards, each of the shadow maps is synthesized by rendering the scene multiple times.

If we were to modify the class to incorporate the Scene-Dependent projection, which tries to create a tighter bounding box, the `StartRendering` function would require another parameter, an array of scene nodes contained in the current split. The modification would not be very complex.

4.3.1 Core functions

As shown in listing 4.11, there are 4 important public functions that are used in the class.

`StartRendering` is the most complex. It has to calculate the light's view matrix for the current split, as well as the split's near and far plane. It also stores the values calculated since they will be needed in the final render. First, the view frustum for the current split is calculated and the 8 corners

of this frustum are calculated.

Then, a light view-projection matrix is created to look from the light's point of view in the specified direction. Afterwards, the frustum planes are adjusted to encapsulate all the 8 corners of the split's frustum. This results in a view-projection matrix that (when applied) will surely render everything that is contained in the current split.

Finally, the render target is changed and the light's view-projection matrix is applied. The `plainShader` is also activated to decrease rendering time. Polygon offset needs to be used once again to get rid of inaccuracies. A simplified `StartRendering` function is shown in Listing 4.12.

`EndRendering` only disables polygon offset and makes the window the current render target.

```
1    // Retrieve the split positions i and (i + 1)
2    float32 split1 = GetSplitPosition(i, near, far);
3    float32 split2 = GetSplitPosition(i + 1, near, far);
4    CCamera camera2 = camera1;
5    camera2.SetNearFar(split1, split2); // sub-frustum
6    vec3 corners[8];                    // Frustum
        corners
7    camera2.GetFrustumCorners(corners);
8    BoundingBox b;          // Bounding box in light's clip
        space
9    for(uint32 i = 0; i < 8; i++) {
10       b.AddPoint(lightViewProj * corners[i]);
11   }
12   // Clamp BoundingBox to (-1, 1) interval
13   float32 newFar = b.max.z;
14   // Construct crop matrix from BoundingBox x and y extents
15   lightProj = crop * lightProj * lookAtMatrix;
16   plainShader->Use();     // select shader
17   MakeCurrent(rt[i]);     // select render target
18   // Enable Polygon Offset
19   // Set lightProj matrix
```

Listing 4.12: `StartRendering` function

`FetchTextures` performs almost the same as `FetchTexture` from the simple Shadow Mapping algorithm, only it does it for each of the shadow maps. It also sends each matrix $T_i (i = 0, \ldots, n-1)$ to the shader, where T_i is the same matrix as the one in 2.2 only that the camera view frustum is now not the whole frustum, but the subfrustum for the split i.

The function also has to send the information about how many splits are currently active and what the split positions actually are. This is needed since fragment shader has to decide which shadow map should it look into, and that is done through a depth lookup. The depth of the fragment will be exactly the distance from the camera's near plane and therefore, the values have to be delivered to the shader in a way.

4.3.2 Shaders for final render

With the CPU side of the algorithm explained, the only missing part is the `pssmShader` that renders the final scene. To keep it simple, only the crucial parts of the shaders are listed here.

```
1  const int MAX_PSSM_SPLITS = 9;
2  uniform mat4 pssmMatrix[MAX_PSSM_SPLITS];
3  uniform int pssmCount;
4  varying vec4 pssmCoord[MAX_PSSM_SPLITS];
5  void main()
6  {
7      for(int i = 0; i < pssmCount; i++)
8          pssmCoord[i] = pssmMatrix[i] * gl_ModelViewMatrix *
               gl_Vertex;
9      gl_Position = ftransform();
10 }
```

Listing 4.13: PSSM Vertex Shader

As you can see, the vertex shader is simple. It differs from the simple Shadow Mapping only by looping through all the splits and calculating the texture coordinates for each split separately.

```
1  const int MAX_PSSM_SPLITS = 9;
2
3  uniform float pssmSplit[MAX_PSSM_SPLITS + 1];
4  uniform int pssmCount;
5  uniform vec4 pssmShadowColor;
6  uniform sampler2DShadow pssmMap[MAX_PSSM_SPLITS];
7
8  varying vec4 pssmCoord[MAX_PSSM_SPLITS];
9  void main()
10 {
11     vec4 output = texture2D(texture0, texCoord);
12     float distance = -pssmCoord[0].z;
13     for(int i = 0; i < MAX_PSSM_SPLITS; i++) {
14         if(i >= pssmCount)
15             break;
16         if(distance < pssmSplit[i + 1]) {
17             float val = shadowLookup(i);
18             if(val > 0.0)
19                 output = applyShadow(output, pssmShadowColor,
                       val);
20             break;
21         }
22     }
23     gl_FragColor = output;
24 }
```

Listing 4.14: PSSM Fragment Shader

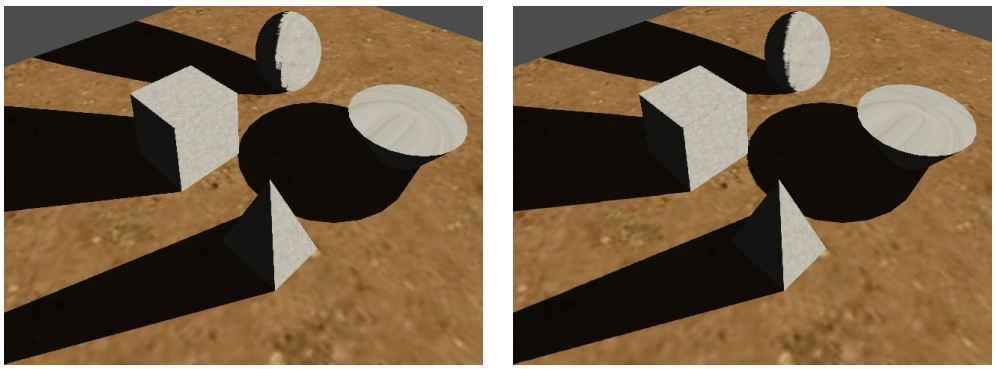
The fragment shader is also looping through the splits. Since the first split (having the number 0) is the closest and each other is further away, a depth check has to occur. It is used to determine in which subfrustum the current pixel is (and thus which shadow map should be used). If we already find a corresponding shadow map, we break the loop to increase performance.

Chapter 5

Results

We are going to compare the output images of the two algorithms described in Chapters 2 and 3. We will be using a few test scenes. The first one will be the Stanford Bunny, the second one will be the Dragon and the final one will be some basic geometric objects. All the scenes are shown in Figure 5.5.

5.1 Quality Comparison

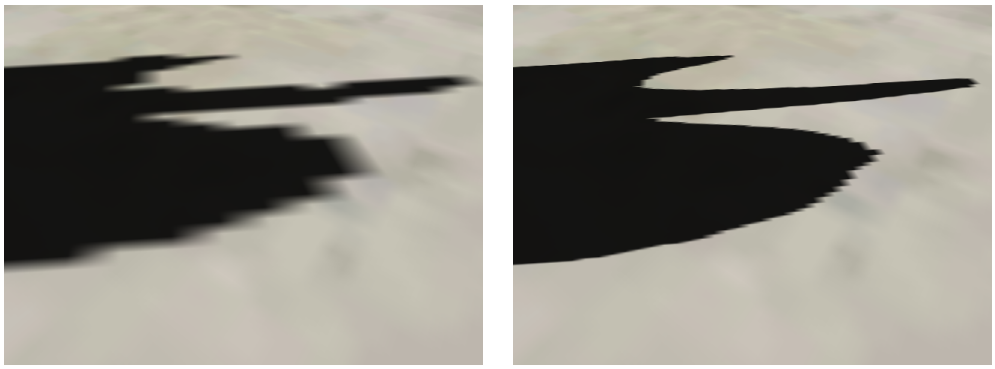


(a) Shadow Mapping (b) Parallel-Split Shadow Mapping

Figure 5.1: SM and PSSM - Shadows far-away from camera

We will now compare the quality of the rendered shadows. As seen in Figure 5.1, the difference is not visible when the camera is far away from

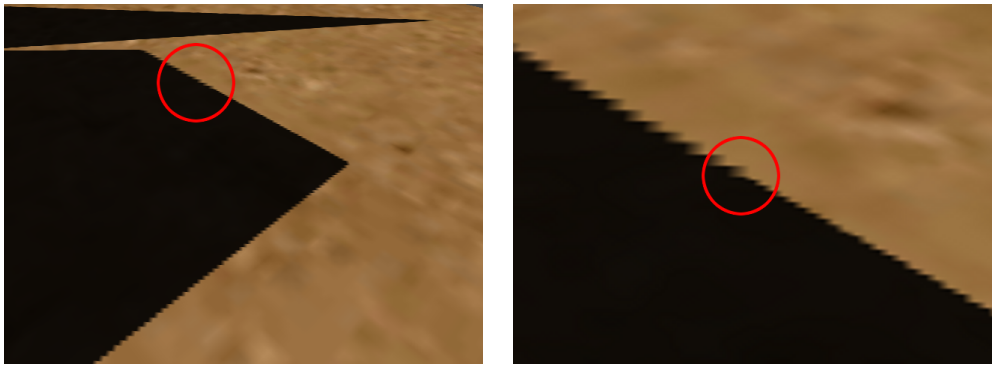
the objects. Shadow Mapping in Figure 5.1(a) uses one 2048×2048 shadow map, while the Parallel-Split Shadow Mapping in Figure 5.1(b) uses 3 splits, each of them 1024x1024 and $\lambda = 0.5$.



(a) Shadow Mapping (b) Parallel-Split Shadow Mapping

Figure 5.2: SM and PSSM - Shadows close to camera

The difference becomes noticable when the camera is close to the objects and the light is far enough. A huge leap in quality is seen between pictures in Figure 5.2. This is what Parallel-Split Shadow Mapping is best at. The first split is small enough and therefore the precision is much better than the simple Shadow Mapping algorithm.



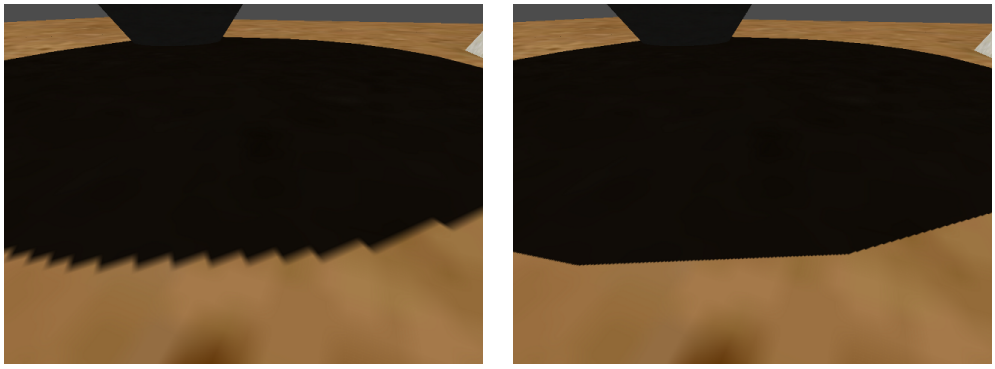
(a) Precision change (b) Zoom-in

Figure 5.3: PSSM - Jump in precision

PSSM retains a constant quality of shadows thanks to the sampling being unaffected by the camera's position. But there is one quality problem, which is the jump in precision on the split planes.

As shown in Figure 5.3, you can see the change in precision and therefore recognize the split. The question is if a player would notice this in a video game. It can be altered by changing λ, but when λ is close to zero or to one, the average sampling makes even bigger jumps on the split planes.

A good example of how λ alters the output quality, close-up screenshots with $\lambda = 1.0$ (corresponding to the logarithmic split scheme) and $\lambda = 0.0$ (corresponding to the uniform split scheme) are shown in Figure 5.4.



(a) $\lambda = 0.0$ (b) $\lambda = 1.0$

Figure 5.4: PSSM - λ quality change

5.2 Performance

The performance of both algorithms is crucial, since video games need to maintain a high frame rate, otherwise they are unplayable. The optimal way is to find a solution that retains good precision, but is still quick enough.

The machine for testing was a Lenovo G560 Notebook with Intel Core i3 330M 2.13 Ghz and an NVIDIA GeForce 310M. All tests were performed in fullscreen in native 1366 $\times$ 768 resolution. No hidden-surface removal techniques or frustum culling were used, rendering to texture was done using

P-buffers and scene rendering was done through vertex arrays (no Vertex Buffer Objects were used).

The quality was measured only visually and after compared to each other, the best and the worst were chosen.

		1024×1024		2048×2048	
Test scene	**Triangles**	**FPS**	**Quality**	**FPS**	**Quality**
Primitives	1300	328.3	Medium	180.0	Medium - High
Stanford Bunny	2947	320.0	Medium	161.3	Medium - High
Dragon	47826	75.9	Low	71.0	Medium

Table 5.1: SM - Different resolutions comparison

In Table 5.1, we compare Shadow Mapping with a 1024×1024 shadow map and Shadow Mapping with a 2048×2048 shadow map. The results show that there is not a big difference in quality between the two resolutions, but on simpler objects like the Primitives or the low-poly Bunny, decreasing the precision doubles the performance.

The shadow quality is lower for the Dragon, which does not have smooth edges like the other two scenes tested. Low quality means that it would strongly distract the player of the game.

In Table 5.2, we compare Parallel-Split Shadow Mapping using 3 maps and $\lambda = 0.5$. Two resolutions were chosen, 512×512 and 1024×1024. The lower resolution had worse results than the 1024×1024 Shadow Mapping due to the strong "shaking" of the shadows. The higher resolution, on the other hand, had much nicer results while retaining a quite good performance.

One paradox occured with the Dragon, and that was that the lower resolution achieved the same average FPS as the higher resolution. This is probably due to the fact that sending large amounts of triangles to the graphics card was the bottleneck of the rendering, and therefore the actual rendering became unimportant. If we used some optimization techniques such as Vertex Buffer Objects to reduce the CPU - GPU transfer, the results would

		$3 \times 512 \times 512$		$3 \times 1024 \times 1024$	
Test scene	**Triangles**	**FPS**	**Quality**	**FPS**	**Quality**
Primitives	1300	231.0	Low - Medium	162.7	High
Stanford Bunny	2947	214.3	Medium	147.2	High
Dragon	47826	36.9	Low - Medium	36.9	High

Table 5.2: PSSM - Different resolution comparison

certainly differ.

The second part of the tests uses percentage-closer filtering to filter the shadow map with a simple 7×7 convolution kernel. Such a convolution decreases performance rapidly, since for every fragment, the shader has to read 49 values from the shadow map instead of 1 value. But the shaking in PSSM is decreased strongly and the overall quality increases. The results are shown in tables 5.3 and 5.4

		1024×1024		2048×2048	
Test scene	**Triangles**	**FPS**	**Quality**	**FPS**	**Quality**
Primitives	1300	68.0	Medium	55.0	Medium
Stanford Bunny	2947	70.3	Medium	61.0	High
Dragon	47826	42.0	Low	38.0	Medium

Table 5.3: SM - 7×7 Percentage closer filtering

		$3 \times 512 \times 512$		$3 \times 1024 \times 1024$	
Test scene	**Triangles**	**FPS**	**Quality**	**FPS**	**Quality**
Primitives	1300	55.0	Low	53.0	High
Stanford Bunny	2947	65.0	Medium	51.2	High
Dragon	47826	27.5	Medium	24.3	High

Table 5.4: PSSM - 7×7 Percentage closer filtering

Notice in the Figure 5.5, that the dragon's shadow is not as smooth as the shadows of the bunny or the geometric primitives. This causes the shadows for the dragon to require better shadowing techniques, otherwise the details will be lost and the results will be more distracting than in the other two test scenes.

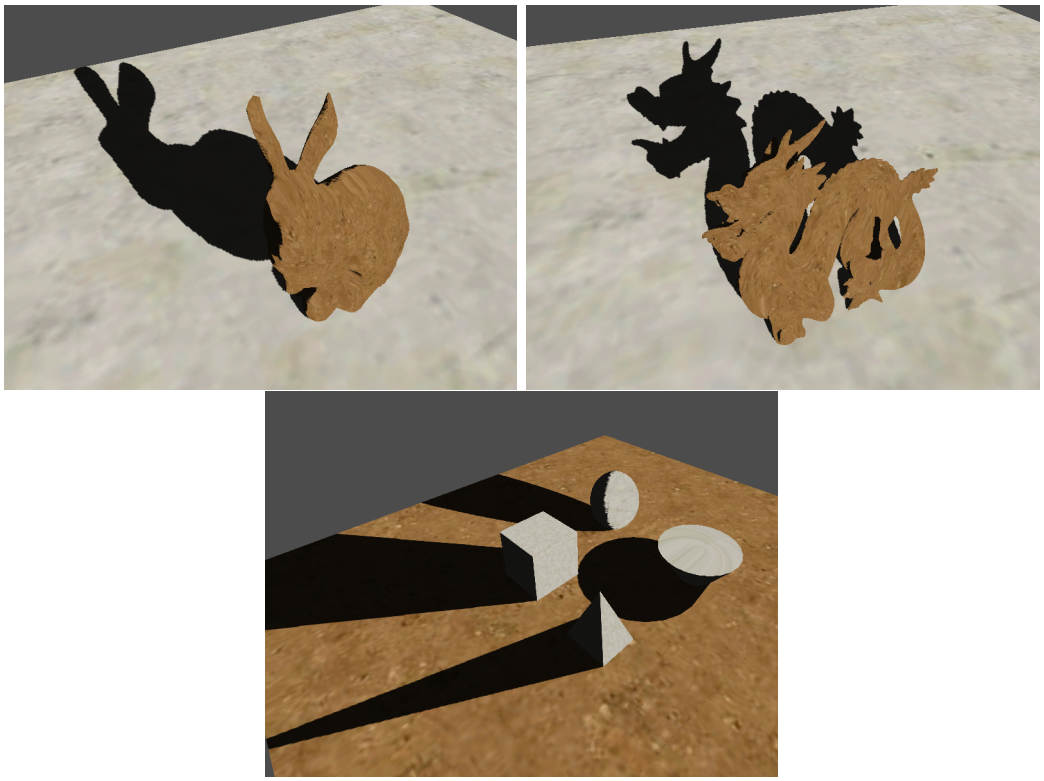


Figure 5.5: Test scenes used

Conclusion and Future Work

In this thesis, we have presented two algorithms that render real-time shadows fast and efficient, while trying to reduce the error and improve the visual appearance to be as realistic as possible.

The first approach is Simple Shadow Mapping, which incorporates the rendering to an off-screen depth texture from the light's point of view. That way, we have a lookup texture that contains distance information for all the objects that are visible from the light's position (and thus are the only ones lit). Our implementation uses OpenGL and GLSL and only uses one light source casting shadows. The algorithm is easily extendable to support multiple light sources while retaining a low number of rendering passes.

The second approach is Parallel-Split Shadow Mapping, a technique that was presented in [Zhang06], which uses multiple shadow maps for one light source to increase precision. The camera's viewing frustum is split into a few parts using split planes parallel to the near and the far plane and one shadow map is used for each subfrustum. The algorithm is more complex than simple Shadow Mapping and decreases peformance, but it is a good quality enhancement.

In the future, a combination of some other Shadow Mapping improvements and the Parallel-Split Shadow Mapping would be possible. These include filtering of the shadow maps to achieve better results, because there are lots of techniques that work better and faster than percentage-closer filtering. If we were to produce soft shadows, we could incorporate another rendering pass after the shadows are synthesized and before the final scene

is rendered. Making a simple convolution on each of the shadow maps would provide us with soft shadows, but the performance drop would be similar to percentage-closer filtering.

Packing the shadow maps into one could also provide some performance increase. Maybe even allow us to reduce the number of rendering passes. But a more straightforward approach is to incorporate the geometry shader as mentioned before to reduce the number of rendering passes.

Bibliography

[Williams78] Williams, Lance. 1978. "Casting Curved Shadows on Curved Surfaces." In Computer Graphics (Proceedings of SIGGRAPH 1978) 12(3), pp. 270-274.

[Crow77] Crow, F. C. 1977. Shadow algorithms for computer graphics. SIGGRAPH Comput. Graph. 11, 2 (Aug. 1977), 242-248.

[Zhang06] Zhang, F., Sun, H., Xu, L., and Lun, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In Proceedings of the 2006 ACM international Conference on Virtual Reality Continuum and Its Applications (Hong Kong, China). VRCIA '06. ACM, New York, NY, 311-318.

[Nguyen07] Nguyen, H. 2007 GPU Gems 3. Addison-Wesley Professional.

[Rost06] Rost, R. J. 2005 OpenGL(R) Shading Language (2nd Edition). Addison-Wesley Professional.

[Donnelly06] Donnelly, W. and Lauritzen, A. 2006. Variance shadow maps. In Proceedings of the 2006 Symposium on interactive 3D Graphics and Games (Redwood City, California, March 14 - 17, 2006). I3D '06. ACM, New York, NY, 161-165.

[Lloyd08] Lloyd, D. B., Govindaraju, N. K., Quammen, C., Molnar, S. E., and Manocha, D. 2008. Logarithmic perspective shadow maps. ACM Trans. Graph. 27, 4 (Oct. 2008), 1-32.

[PaulsProjects] http://www.paulsprojects.net/tutorials/smt/smt.html
OpenGL Shadow Mapping Tutorial - Paul's Projects

[GLregistry] http://www.opengl.org/registry/
OpenGL Registry