universität innsbruck

igs iis

Visual Computing Project Report — Shadow Mapping

Matthias Ebner & Frederik Hirsch & Paul Prünster

February 6, 2024

Distribution of Tasks

The initial setup of the project was done mostly by Paul with some help from Frederik. The main phase of implementing the shadow map was done by Matthias and Paul. Some later refinements were made by Matthias and Frederik. The presentation slides were made by Frederik, and the report by all of us.

Introduction

Realistic shadows play a crucial role in enhancing the visual quality and perceived depth of 3D graphics. Accurate shadow rendering adds a layer of detail and realism that significantly improves the viewer's immersion in the scene. This is particularly important in applications such as video games, where realistic shadows enhance the gameplay experience by making the virtual world feel more immersive and interactive or film and animation, where effective use of shadows helps convey mood, depth, and realism, contributing to the overall storytelling experience.

Shadow mapping is a fundamental technique in computer graphics that aims to realistically simulate the way light and shadows interact with objects in a 3D scene. We begin by unraveling the fundamental concept: rendering the scene from the light's perspective. Instead of focusing on colors and textures, this "shadow pass" captures only the depth information of objects visible to the light. This depth data, stored in a special texture called a shadow map, becomes the key to rendering realistic shadows.

In the subsequent "lighting pass," we revisit the scene from the usual camera viewpoint. But this time, for each pixel, we consult the shadow map. By comparing the pixel's depth to the depth stored in the map, we determine whether the light ray from that pixel can reach the object or if it gets blocked by another object in the scene. This simple "shadow test" allows us to efficiently identify shadowed areas and adjust their shading accordingly.

However, the path to perfect shadows is not without its problems. We ran into common challenges faced by shadow mapping, such as aliasing artifacts, bias issues, and limitations in shadow resolution. We explored techniques designed to mitigate these challenges, such as (dynamic) bias and filtering techniques, showcasing how they enhance shadow quality and maintain rendering efficiency.

Fortunately, there are lots of good resources when it comes to this topic. We especially found the *LearnOpenGL* [1] and *opengl-tutorial* [2] articles helpful.

Background

While the basic concept of shadow mapping is intuitive, its theoretical foundation involves concepts from linear algebra and projective geometry. Shadow maps are essentially depth textures rendered from the light's perspective. This necessitates transforming scene geometry into light space (view space from light's perspective), achieved by firstly multiplying each vertex position with the $WorldPos$ matrix to transform it from model to world space, then with the $LightView$ matrix to transform it to light space and finally the $LightProj$ matrix, which is an orthographic projection because we have a directional light source:

$$LightSpacePos = LightProj \cdot LightView \cdot WorldPos$$

This transformation ensures that the shadow map accurately reflects the depths as seen by the light.

During the second pass, for each fragment on the screen, we need to determine if it's in shadow. This involves comparing its depth in light space with the depth value stored in the shadow map at the corresponding texel. The corresponding texel coordinates are calculated using another perspective transform:

$$\begin{aligned} ShadowMapTexel = & ScreenToNDC \cdot Viewport \cdot \\ & LightProj \cdot LightView \cdot WorldPos \end{aligned}$$

Here, ScreenToNDC maps screen coordinates to normalized device coordinates (NDC), and Viewport accounts for the shadow map's position and size on the screen. The actual shadow test involves comparing the fragment's depth (f_depth) with the sampled depth from the shadow map (s_depth):

```
if (f_depth > s_depth + bias) {
    # fragment is in shadow
} else {
    # fragment is lit
}
```

Listing 1: Testing for shadowss

The bias term is crucial to address the problem called shadow acne, which results from the fact that the shadow map is discrete in its nature. As we can see in figure 1, if the light hits the surface at a steep angle, some pixels can appear further from the light source than the entry in the light map, even if they should be directly in the light. This results in weird artifacts because all these fragments get marked as shadows.

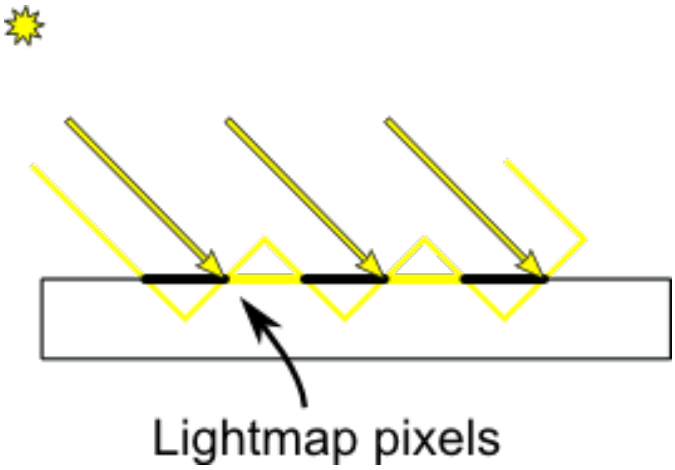

Figure 1: Explanation for shadow acne. [2]

Method

We started with the TinyObjLoader project template and began by loading the model. The model was not triangulated correctly so we had to fix this in blender, the model also had to be scaled down and centered.

The next step was to add a new Frame buffer for the shadow map and render the scene from the light's perspective onto it. This was done with a different vertex and fragment shader that outputs its fragments onto the shadow map depth buffer.

In the second render pass, we sample this map with different shaders that the shadow map is passed to, to check whether a fragment is in a shadow or not. And so we ran into our first problem, quite a common one: shadow acne. This was initially fixed by adding a bias to the depth sampled on the shadow map.

Later, we implemented an improved version of the bias: **dynamical bias**. Depending on the angle the light rays hit a surface, the needed bias varies. This means that the bias can be smaller when the light is at a small angle, as the acne would not be so strong. On the other hand, if the angle is very steep, we need a bigger bias to account for stronger acne. We can adapt the bias dynamically with the angle between the surface and the light.

To do this, we calculate the angle θ with the dot product of the normal vector n and the inverse light direction vector l: $\theta = arccos(n \cdot l)$. The exact bias b, needed to eliminate shadow acne, would then be (in relation to the texel width w): $b = \frac{1}{2}w \cdot sin(\theta)$. For a visual representation helping you understand the formulas see figure 2. In the demo, the dynamic bias is toggled with the **2 key**, initially it is turned off.

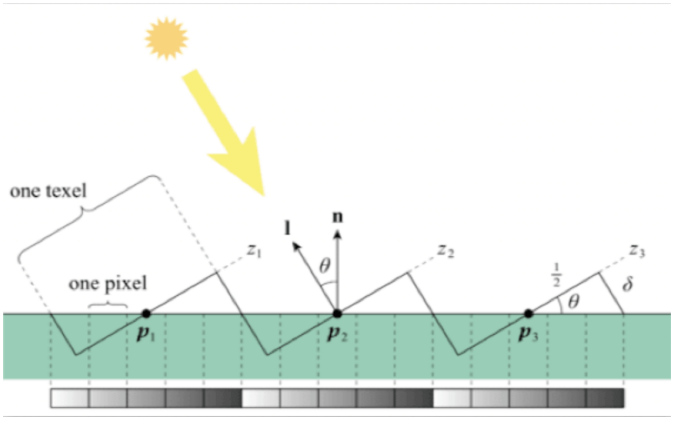

Figure 2: Dynamic bias [3]

We then played around with multiple anti-aliasing strategies, some faster than other and some prettier than others.

- **Percentage Closer Filtering**: changing the type of the texture to sampler2DShadow, so that the hardware samples multiple values of the texture on its own and interpolates between them. This makes the shadows a bit smoother and is also very fast.
- **Poisson sampling**: In this method, the edges of the shadows are samples around a so called "poissonDisc", where the Shadow map is sampled around a point with certain offsets.
- **Stratified Poisson sampling**: in this method is similar to the above method with the difference that the index of the next sample on the disc is chosen randomly. This gives an interesting style of shadows.

```
for (int i=0;i<4;i++){
```

```
    vec2 pd = poissonDisc[i]/3000.0;
    float z=(ShCoord.z-bias)/ShCoord.w;
    texture(
        shadowMap,
        vec3(ShadowCoord.xy + pd, z)
    )
}
```

Listing 2: Aplying normal Poisson sampling

For example in Listing 2, we can see a part of the implementation of the normal Poisson sampling, per fragment we look at 4 different positions in the shadow map and take the average, the more points are in the shadow, the darker the fragment. After a lot of playing around with the values, we were quite happy with the normal Poisson disc sampling and also the stratified Poisson sampling, so we left them both in the demo. The different light stages can be toggled with the **1 key**.

Lastly, we played around with different resolutions of the shadow map, from 1024 to 10 times that. The results were way better with a better resolution as expected, this also meant we could decrease our bias to a smaller number to reduce **Peter Panning**. Peter Panning is the effect where shadows appear detached from their objects, because the bias prevents the drawing of shadows near the edge of an object. In the end, we settled for a resolution of 4×1024 as it was the best tradeoff between performance, memory space and visual results.

Results

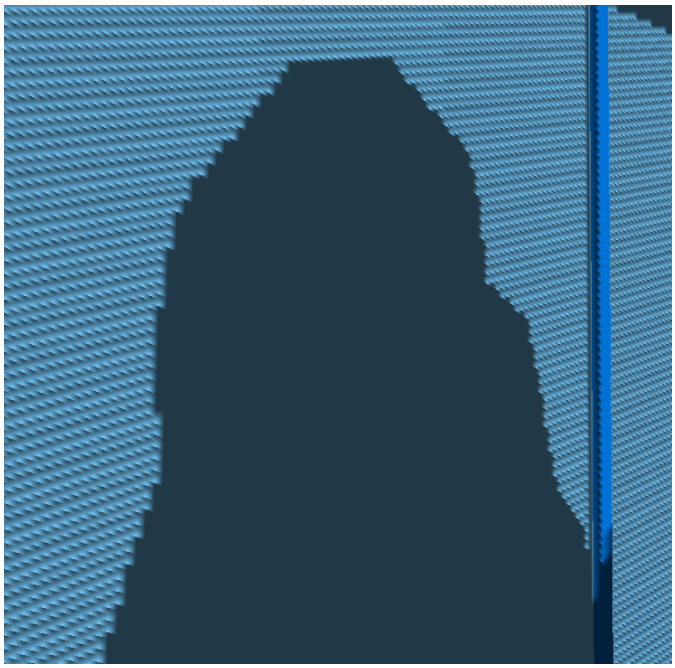

Figure 3: Shadowing without any bias

In figure 3, we can see how the plain shadowing without any bias creates shadow acne. The acne lines are very thin as we increased the resolution of aour shadow map to 4096 by 4096.

Figure 4: If we add a *bias* to our shadow map, we can get rid of the shadow artifacts.

In figure 4 we see the result of adding a fixed bias to the shadow sampling, the result is that there are no shadow artifacts on the ground due to the discreteness of the map, as it is offset into the ground. Later on we implemented a dynamic bias, but even after playing around with the constant factor in our bias term, the results didn't really improve upon the static bias of 0.005. We still left it in the demo, where you can toggle it on and off to see the difference for yourself. In this picture, we can see that the edges of the shadows are still pretty rough, but we can solve this by sampling multiple spots around the edges and interpolating the result.

Figure 5: To smooth out the shadow lines, we use *poisson sampling*.

In figure 5 we can see the result of poisson sampling with nice soft shadows. But still we can see some sharp edges and individual texels as this is basically just 4 less strong shadows that are offset by a bit from and overlayed over each other.

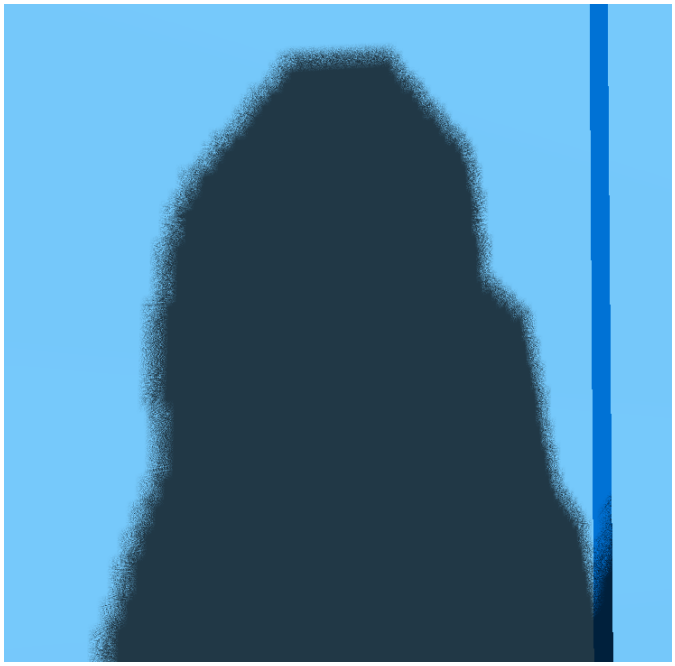

Figure 6: A different method to smooth out the shadow lines, we can use *stratified Poisson sampling*.

Another strategy for sampling the shadow data to provide even smoother edges that blur the individual texels even more is stratified Poisson sampling 8. This simply means we are not sampling the same spots over and over again, but there is an element of randomness. This gives this kind of dithering look to the shadows, which does not look very realistic but still a nice effect.

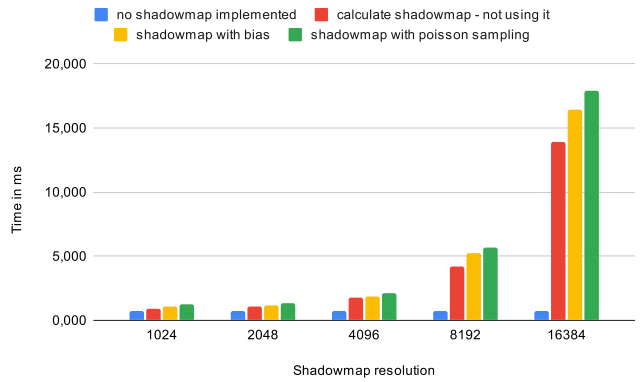

Figure 7: Comparison of calculation time per frame of different shadow settings.

Using a higher resolution of our shadow map, decreases the needed bias and the peter pan effect. But it also increases the calculation time per frame drastically. We can see in figure 7 that the calculation time of a frame without shadow map is 0.7 ms. With the same scene and a shadow map of size 16384 x 16384, together with a stratified Poisson sampling, the calculation time increases to 17.883 ms. In other words, the frame rate decreases from 1428 to 56. So we need to find a tradeoff between a good shadow and calculation time. But sharp edges on the shadows do not always result in better shadows. So using a smaller shadow map and an antialiasing technic will result in more realistic shadows and faster calculation time per frame. We used the same scene, the same light position and the same window size, for each time measured. With this setup, we calculated the average calculation time per frame over a period of time. We used a 16 $\times$ AMD Ryzen 7 5700U with Radeon Graphics and 16GiB of RAM for these benchmarks.

Figure 8: The final result of the city with shadows using poisson sampling.

Conclusion

In conclusion, we can say we had a lot of fun on this project, implementing such a useful feature and playing with different techniques and parameters. The implementation was pretty straight forward, due to some well documented implementations, examples, and this not being the newest shadow technique. We had the most problems getting started with loading the model and applying the texture. The TinyObjLoader was extremely helpful to get started. We did non encounter any major problems and learned a lot from this project. We found C++ and OpenGL very straight forward to work with, after some getting used to in the other assignments. Shaders are famously hard to debug, and some playing around with a shader tool could help with the learning curve in this course.

References

[1] JoeyDeVries. LearnOpenGL - Shadow Mapping, 2022. https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

[2] opengl tutorials. Tutorial 16 : Shadow mapping, 2018. https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/.

[3] Javier Salcedo. Rendering shadows in real-time applications 2: Shadow maps, 2023. https://shorturl.at/bgoD9.