MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA
FACULTY OF AUTOMATION AND COMPUTER SCIENCE

Light Rendering Algorithms with Dynamical Lights

LICENSE THESIS

Graduate: **Alexandru HARAGÂȘ**

Supervisor: **Prof. Dr. Ing. Dorian GORGAN**

2023

Contents

Chapter 1. Project Statement

The project's purpose is to implement and test a group of light rendering algorithms. The tests have the purpose of observing how the algorithms perform in different scenarios, most importantly, in a dynamic scene. The chosen algorithms are Indirect Specular Lighting, Ambient Occlusion and Indirect Diffuse Illumination.

Another purpose of the project is the creation of an app that someone could use to test those algorithms and see how they interact and how the performance is impacted. They should be able to select the object to be rendered, as well as the number of light and their properties.

The algorithms are going to be implement with C++ and OpenGL. Then, they are going to be tested in different environments: size of the scene, number and types of lights and if the scene is static or dynamic. The measurements for the tests are going to be the FPS of the scene and how long a frame takes to process.

Chapter 2. Introduction

The main motivation of doing this is because, since I was little, I loved playing video games. This resulted in me gaining a fascination of how they are made. So, for this thesis I wanted to do something related to those games. When it comes to video games, many things come together to make them look as close to reality as possible. And when it comes to realism, light is one of the hardest to get right.

Since the creation of computers, humanity wanted to create a simulation of the real world. In order for them to do so, they have to simulate how things interact. Light is one of them, and a hard one to simulate.

Light is defined as electromagnetic radiation that can bee seen by the human eye [1]. We can consider a ray of light a vector that can interact with any material in different ways depending on the material's properties. The light can be reflected or refracted by a material.

In the case of rendering lights, computing the way the light interacts with the environment requires a lot of resources. The computation has to be done on each pixel to see how it interacts with the light. The computations only increase in complexity if you were to take into account reflections and refraction. Another thing is to consider how the pixel interacts with the ones around it. For example, it could be indirectly illuminated from another point, or be obstructed by something.

When we render light, the minimum things that are usually rendered are: the ambient light, which is constant value used for the entire scene, the specular light, which is the light emitted from a flat surface, where all rays have the same direction, and diffuse light, the light that gets reflected by a rough surface, where the rays go in different direction. While those are enough to create a decent scene, it is nowhere close to reality. The algorithm presented in this paper are meant to make the scene more realistic by adding interaction between the objects and the environment.

The biggest difficulty when implementing the algorithm is the performance drop they can create. Both indirect specular lighting and indirect diffuse illumination could require a lot of resources and could take a lot of time to be processed.

Another difficulty would be getting the realism just right, some of them have some values that needed to be selected just right to look good. For example using ambient occlusion for multiple light sources used a two weights, and a bias to average them. Getting that bias can make the realism of the scene. Another example would be for indirect specular, where we have to decide how far do we want to go to get an intersection.

The final big difficulty that may arise is the technological one. As explained previously, there exists a need for resources. This results in the need of a machine with high specs, those being a lot of RAM and a powerful graphical processing unit, or GPU.

In this project I'm going to implement some algorithms created to obtain a more realistic interaction between objects and lights. In chapter 3 I'll show the bibliographic study, then, in chapters 4 and 5 the theoretical and technological considerations. In

chapter 6 I will show how I implemented them. In chapter 7 I will show the experiments and results.

Chapter 8 presents how to use the application, the user's manual. And finally, chapter 9 is the conclusion of the thesis.

Chapter 3. Bibliographic Research

I'm going to present the algorithm in more details in the next chapter, but here is brief history of the ones I'm going to use. The algorithms can be classified in 2 groups: standalone, and combined. I'll start with the standalone ones.

There were other works that tested those algorithm. One of them was Cristian Lambru Phd Thesis [2]. There the author presented, among others, those algorithms and tested them. The main difference is that those tests were done for a static scene with one light. This project intends to use multiple lights and a dynamic scene.

Deferred rendering is a method first created Michael Deering and Co. [3]. Later, Saito Takafumi and Tokiichiro Takahashi [4] introduced the the modern algorithm for it. The idea is to render the scene and save the information into a buffer, or a 2D image. This can then replace future renderings of the object.

Tiago Sousa and Co. [5] proposed to use this method to create four buffers, each holding information about the normal, depth, position and color for each pixel. Each stored in a separate buffer, or image, then use those to compute different things, like reflections. It can help with the speed of rendering, at the cost of some memory.

Reflective Shadow Maps, abbreviated as RSM, was proposed by Carsten Dachsbacher and Marc Stamminger [6], and it is deferred rendering from the light's point of view. The maps created are used to determine indirect lighting. This is done by projecting a pixel from screen space to RSM space, then sampling the area around it to determine how much indirect diffuse light it gets.

Ambient Occlusion [7] is a way to darken certain parts of the scene that are enclosed, or surrounded by geometry. Initially it was considered that the geometry needed to be checked to infinity distance around a point, until Zhukov and Co. [8] proposed a way to check only a finite distance amount around the point.

Later, after a few more improvements, like the proposal of adding a surface to cache the information [9], or trying to make the it feasible for real-time rendering [10], Mittring [11] developed a technique that could use only visible geometry. This was done by sampling the area around the point to compute the occlusion factor.

Screen Space Reflections is used to compute the indirect specular light. First proposed by Turner Whitted [5] and later adapted by Tiago Sousa and Co.[12] to work on buffers. The algorithm follows the reflected vector from camera's point of view to the pixel until it intersects the geometry or after a given distance. This algorithm would have been computational heavy, if not for Sousa's technique that lowers some of the complexity.

Now I am going to present the combination algorithm. They use at least two of the previous presented algorithm to make something different. Again, I will present them more in depth in the next chapter.

Indirect Specular Lighting was proposed by Sousa and Co.[12]. They combined Screen Space reflection and Reflective Shadow Maps. In short, if there is no intersection from camera's point of view, there may be one if we look from the light's point of view.

We use both the maps from deferred rendering, as well as the RSM as the buffer to follow the vector and see any intersection.

Ambient Occlusion for Multiple Lights used the method proposed by Mittring [11] to calculate the ambient occlusion from multiple points of view. An averaging method was proposed by Kostas Varids and Co. [13]. The weigh is determined by two factors: distance and if the point is visible from that point of view. There is a bigger importance given to the visibility, of course.

For **Indirect Diffuse Illumination** a method was first proposed by Carsten Dachsbacher and Marc Stamminger [6] in their RSM paper. Another method was proposed by Cristian Lambru an Co. [14]. This technique requires to consider all pixels around a point as being a light source. You take a set of those and use them to calculate the amount of indirect diffuse light they illuminate the point.

The difference between the two is the way the points are chosen, for Carsten Dachsbacher and Marc Stamminger, they took a fixed pattern around the point and used it to compute the value. Cristian Lambru an Co. instead used a smaller sample pattern, that they rotated around and combined it with the previous frame. This increases the speed of the algorithm, at the cost of more artifacts appearing.

Chapter 4. Theoretical considerations

4.1. Graphic Pipeline and Basic Graphics Notions

First I would like to explain the terms I am going to use for the rest of the thesis. "Rendering" is the process of an object being created and shown on the screen. A *vertex* represents information about a point in space, like position and colour. They are the input for the pipeline. A *primitive* is either a point, made out of one vertex, a line, made out of two vertices, or a triangle, made out of three vertices. Primitives are the output of the vertex shader. A *fragment* is the information about a pixel, they get created when the object is rendered and represent the output of the pipeline. A *buffer* is a matrix, or image that keeps the information of all fragments created. A *model* is the object rendered on the screen, it is made out of vertices and fragments. A *scene* are all objects rendered to be show on screen.

The graphics pipeline are the steps taken to render an object. The program starts by getting a vector of vertices as input. This is followed by rendering the vertex. Those inputs are then processed in the Vertex Shader, a program that can be modified by the programmer. The result are primitives. The next step is Rasterization, where the program determines the pixels, or fragment that are inside the primitives. It should be noted that the pipeline can discard primitives that do no face the viewer, called clipping, this is done after the Vertex Shader, but before Rasterization.

The next stage is Fragment Processing, where a fragment shader is used to process all fragments. Just like the vertex shader it can be programmed before starting the rendering process. This is the place we are interested in, because this is where we compute how much light each fragment gets. This is also where the colour and texture are given to the fragment. Due to this, the fragment shader usually uses the most resources to render the object. After the process is done, the fragment goes through a series of tests to check if it needs to be shown. For example, if there already exists a pixel that is in from on the computed one, then the latter one is discarded, as it is not needed to be shown.

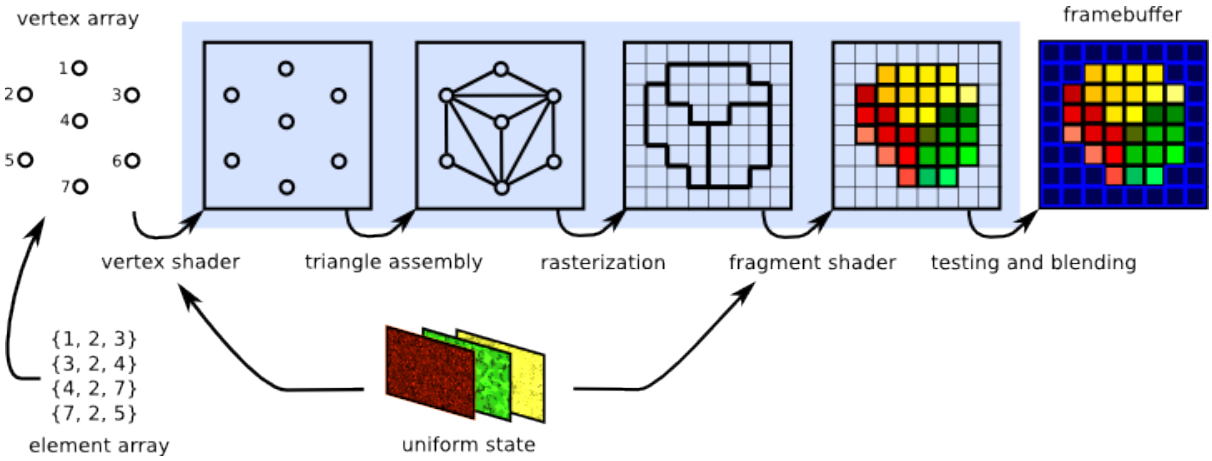


Figure 4.1: Graphics pipeline [15]

4.2. Standalone Algorithms

In this section I will present the basic algorithms used in this project. They are the basis and will be combined to create the important ones. The question you might have is: "why choose those specifically?". Well, all of them have something in common, they all either create or use buffers. Deferred Rendering and Reflective Shadow Maps create buffers, meanwhile Screen Space Reflection use these buffer to reduce the amount of resources they would have needed otherwise, and Ambient Occlusion uses the buffers to create a new one.

4.2.1. Deferred Rendering

For the project I used the method proposed by Saito Takafumi and Tokiichiro Takahashi [4] to create the buffer for Tiago Sousa and Co. [5] method. The idea is to render the objects once and keep the information in a G-Buffer. We can use that G-Buffer to store four separate buffers, each having some information about the pixel.

The four pieces of information are: position, normal, albedo or colour and specular component. The buffer is then used to compute more complex information, like ambient occlusion. This reduces the amount of computation needed to be done, since only the fragments that the camera sees need to be computed. You no longer have to render the object again, instead it is enough to use the pixels from the buffer to do the needed computation. Figure 4.2 shows the process done to create the scene.

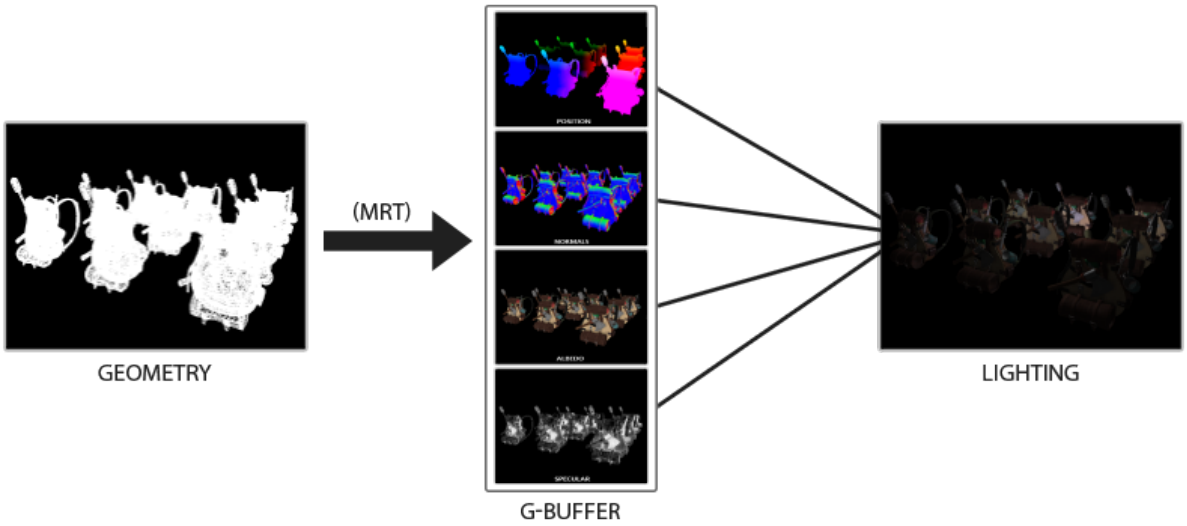


Figure 4.2: Steps needed to create the scene. First the geometry is rendered, then the buffers are created, and finally, the light is added to the scene[16]

This render can easily increase rendering time and light computation, at the cost of using more memory to keep the buffer. Due to this, other previously hard to do algorithms become easier to do. For example, the light from multiple sources can be used to determine how much each fragment has light, and instead of rendering the object again, we use the buffer to look around the fragment. Another great usage for deferred rendering is creating more realistic reflections, as we do not need to know the entire geometry, just what is in the buffer. Obviously, we could only create reflections of what is seen in the buffer, as we don't know anything other than what we see, but we could render the scene to see it from somewhere else, like the mirror's point of view and see what it could be reflected.

Overall deferred rendering created and way to do complex computation on less items, and it is an understatement to say that it has revolutionized the way objects are rendered.

4.2.2. Reflective Shadow Maps

Reflective Shadow Maps, or RSM, were proposed by Carsten Dachsbacher and Marc Stamminger [6]. It is a combination between deferred rendering and shadow maps. The fragments are projected from the light's point of view. Creating a buffer that has four maps, one each for: position, normal, depth and flux, as seen in figure 4.3. To project the object we either use an orthographic projection, if the light is a direct light, or perspective projection, if the light is specular, similar to a normal camera.

The same advantages and disadvantages apply to RSM as for deferred rendering. In addition, the memory usage and number of times the object has to be rendered increases with the number of lights used. This is because a reflective shadow map has to be created for each light.

The RSMs fixed what the G-Buffer lacked, the knowledge of other parts of the scene. The G-Buffer is locked to what the camera sees and cannot know what is hidden behind the geometry. It is like looking at the front of a house and not see it's back, there is no way for you to know if it exits. RSMs fix this by creating a new perspective that

can be used to see more. If a light looks at the back of the house in the example, we have a buffer to see it and can use that to compute some other things about the scene.

RSMs show how an idea can be taken and modified to fix some of it's shortcomings. Yes, it comes at the cost of memory, but more than compensates with the increase in performance.

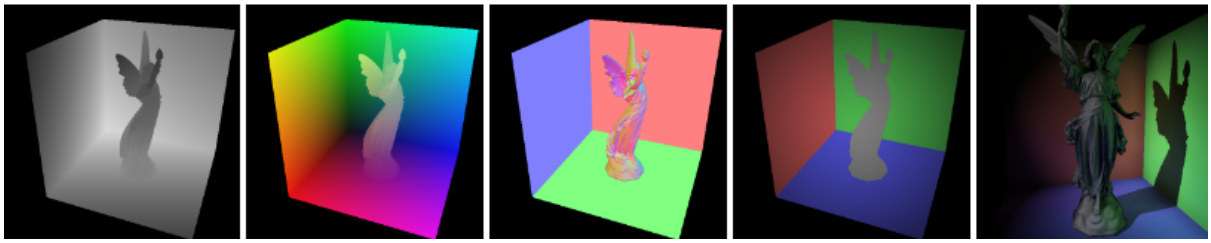


Figure 4.3: The Four buffers created in Carsten Dachsbacher and Marc Stamminger paper, as well as the scene. From left to right they are: depth, world coordinates, normal, flux and the scene.[6]

4.2.3. Ambient Occlusion

Ambient Occlusion [7] is a way to darken certain parts of the scene that are enclosed, or surrounded by geometry. Initially it was considered that the geometry needed to be checked to infinity around a point, until Zhukov and Co. [8] proposed a way to check only a finite distance around the point.

What we are interested is in the method developed by Mittring [11] in order to compute the occlusion factor. This is done by checking the depth values of the fragments around the point. This is done using a kernel. The number of samples is important, the more there are, the more precise it is, at the cost of time. To not take geometry that faces away from the fragment, or is behind it, a hemisphere oriented on the normal is used. Other factors also need to be taken into account, for example how far the geometry is from the point. An overview for this algorithm are shown in figure 4.4

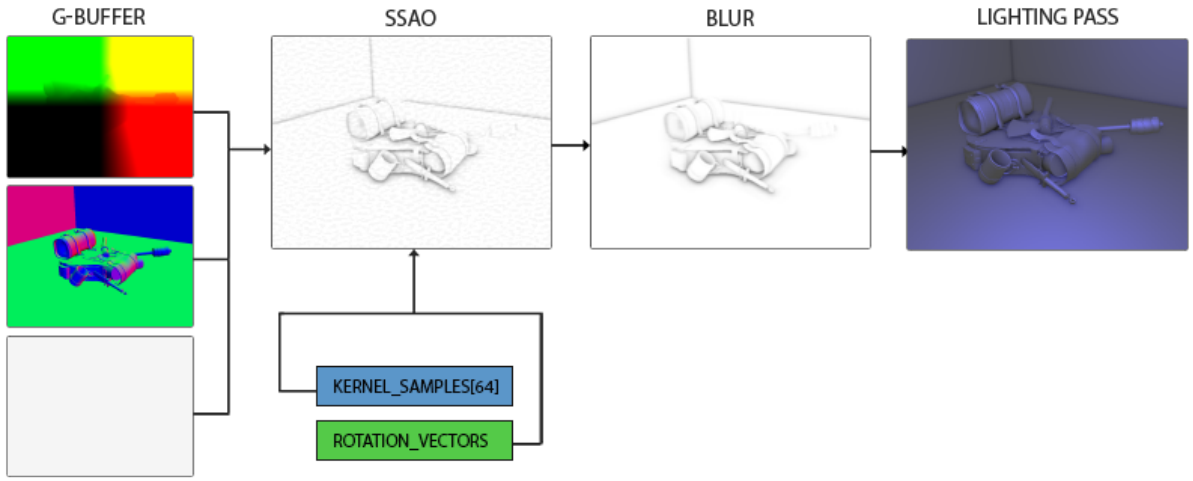


Figure 4.4: Overview of the steps used to create the SSAO buffer[16]

To make computations faster, and to not rely on geometry, deferred rendering is used before in order to create a G-Buffer. The result of the computation is also a buffer named SSAO map.

The formula for the algorithm is:

$$occlusion(p) = 1 - \frac{\sum_{n=1}^{k} depthComp(p.z, x_k.z) * rangeCheck(p.z, x_k.z)}{k}$$

Where k is the size of the kernel, p is the point we want to check the occlusion for and x_k is the kth point in the kernel. "depthComp" is a function that checks whether p is further from the viewer than x_k. It's formula is:

$$depthComp(p.z, x_k.z) = \begin{cases} 1, & \text{p.z} >= x_k\text{.z} + \text{bias} \\ 0 & \text{p.z} < x_k\text{.z} + \text{bias} \end{cases}$$

Where bias is a number chosen by the programmer. The results obtained by Mittring can be seen in figure 4.4 and 4.5.

Figure 4.5: The figure shows a SSAO map[11]

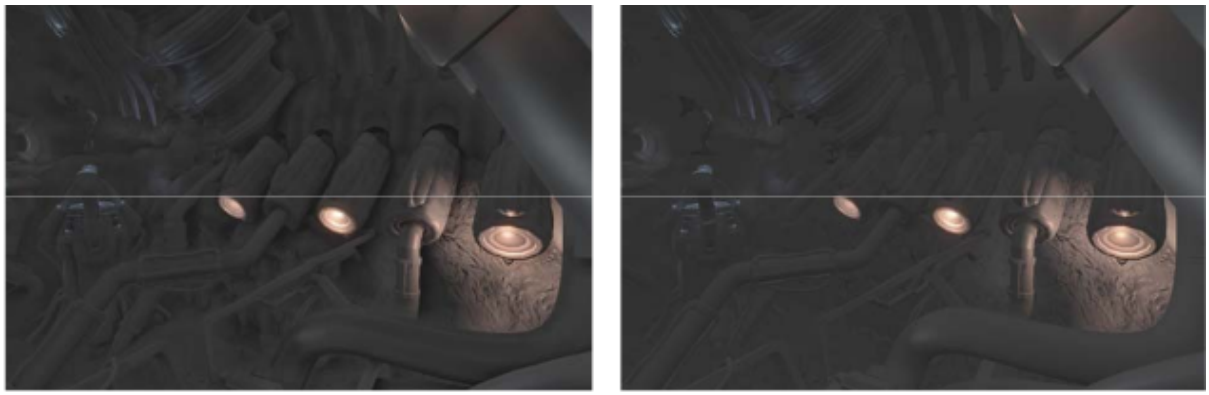


Figure 4.6: Comparison between the use of SSAO, left has SSAO, right does not[11]

This Ambient Occlusion is really just a way to add some realism to the map. One of the first games to use it was Crysis (2007), which at the time was appraised for it's

realistic look, as well as for the specifications a computer needed to have in order to run it. People really love when the game tries to be more realistic visual wise, even if it needs more power to work smoothly.

4.2.4. Screen Space Reflections

It is used to get the indirect specular component. First proposed by Turner Whitted [5]. He proposed the use of ray tracing on the reflection of the user's view to the point, and see if it intersects the geometry, the geometric representation is shown if figure 4.7. This is very heavy computation wise, as it requires to know the entire scene's geometry to work. And when you consider that this has to be done for every fragment, it is easy to understand why it was so hard to use in real time rendering. Figure 4.8 shows one of the scenes created by the author using this method.

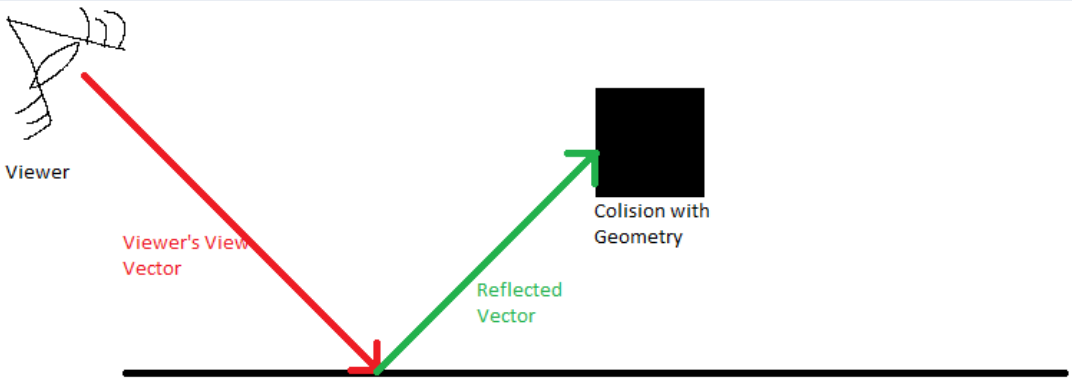


Figure 4.7: Reflected vector and Intersection with geometry

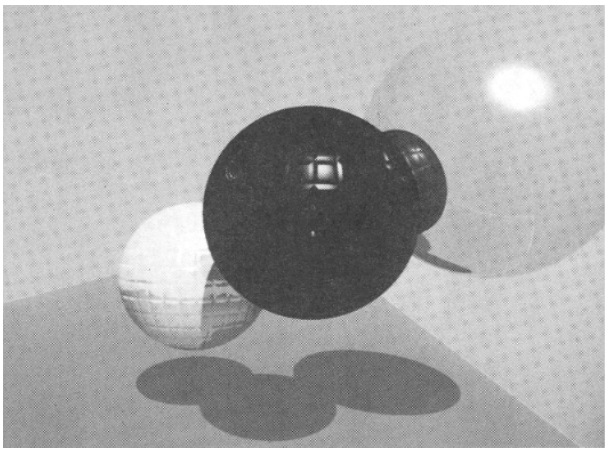


Figure 4.8: Example of a scene created by Turner Whitted[5], the rendering took 44 minutes

To make everything faster Tiago Sousa and Co. [12] adapted it to work on buffers. Similarly, a ray that is the reflection of the user's view to the point is traced, but instead of using geometry, the vector is projected on the buffer and the comparison happens

there. This is way faster and needs less resources to work. This is because the buffer has all the needed information to work, like the position and the normal of each pixel. The check is easy, if we are in front of the geometry in one step, and in the next we are behind, we can look for an intersection. Another place to check is when we are really close to geometry, as there is a high chance for an intersection.

There is still one problem with Screen Space Reflections if a buffer is used. If there exists geometry behind an object that the buffer can't see, it will be ignored, even though it should have intersected it. As explained with deferred rendering, the buffer cannot know what is behind of what it sees because the buffer is just the scene projected onto the screen.

To do the ray tracing, we go in a direction until we are close to an intersection, then we use a binary search approach. We start by going in the direction of the reflection with a given step S. At each step, we check if we are behind or in from of the current pixel, and move according to it.

The pseudocode is:

Algorithm 1 Ray Tracing Algorithm for Screen Space Reflections

```
step ← 0
binaryStep ← 0
dist ← D
while step < MaxSearchSteps do
    projectedCoordinate ← ProjectionMatrix * currentPossition
    projectedCoordinate.xy ← projectedCoordinate.xy/(projectedCoordinate.w*2)+
0.5
    depth ← bufferDepthMap[projectedCoordinate.x][projectedCoordinate.y].r
    if depth ≤ 1000 then
        dDepth ← currentPossition.z − depth.z
        if direction.z − dDept ≤ treshold and dDepth ≤ 0 then
            while binaryStep < MaxBinarySearchSteps do
                direction ← direction * 0.5
                if dDepth ≥ 0 then
                    dDepth ← dDpepth + dir
                else
                    dDepth ← dDpepth − dir
                end if
                binaryStep ← binaryStep + 1
            end while
            return currentPossition.xy
        end if
    end if
    step ← step + 1
end while
```

While I only follow the first bounce, the algorithm can be extended for multiple bounces. Usually one or two bounces are more than enough to represent a realistic scene. Since we care only for the indirect specular light, the first intersection is all I cared about.

4.3. Combining the Algorithms

In this section I'll present algorithms that combine two or more of the algorithms presented before. They are the ones used for the tests. Their objective is to create a more realistic scene.

4.3.1. Indirect Specular Lighting

Indirect Specular Lighting was proposed by Sousa and Co.[12]. They proposed that RSM could be used to get the indirect specular component, if there is no intersection by doing Screen Space Reflections on the G-Buffer. This method tries to fix the lack of knowledge of an object behind another one, something the buffer doesn't know.

In order to do this, we first try to do the intersection with the G-Buffer. In case this does not work, we repeat the process for each RSM until we either get an intersection, or we get to a certain distance. For the reflective shadow maps, binary search can be done, but is not as advantageous. This is because the buffer from the RSM can show the objects from further, thus making them show less details. In both cases the direction of the vector is the same. An overview is shown in figure 4.9.

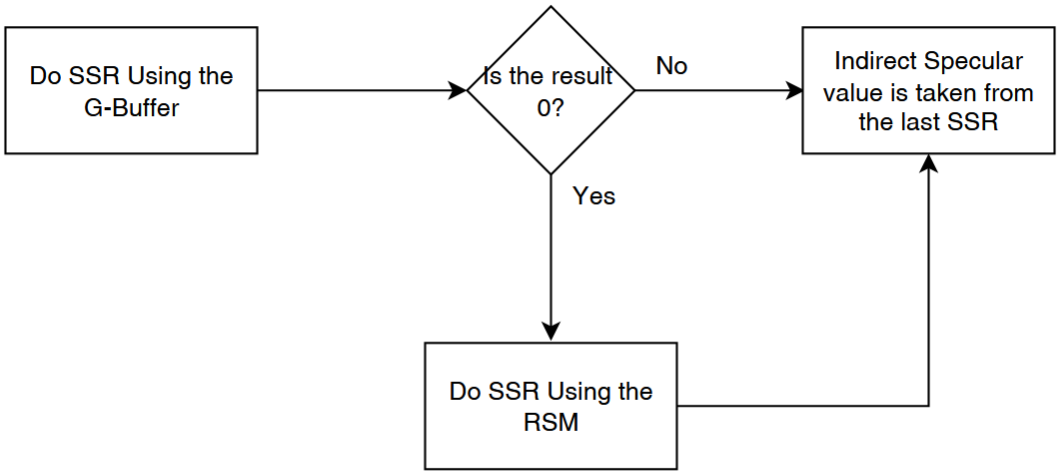


Figure 4.9: Overview of the algorithm. SSR stands for Scree Space Reflectionss

The formula for the indirect specular is:

$$IndiSpec(p) = IndirSpec_{G-Buffer}(p) + b * InidirSpec_{RSM}(p)$$

Where $IndirSpec_{G-Buffer}$ is the indirect specular taken using the G-Buffer, and $InidirSpec_{RSM}$ is the indirect specular taken using the RSM. b has values of 0 or 1 base on the values of $IndirSpec_{G-Buffer}(p)$, 1 if $IndirSpec_{G-Buffer}(p)$ is 0, and 0 otherwise.

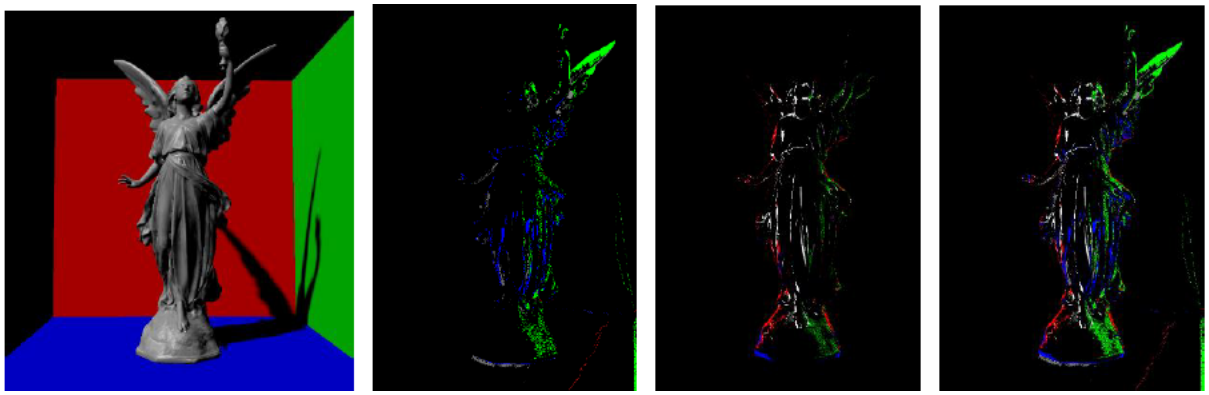


Figure 4.10: Example of the algorithms result [2]

Figure 4.10 shows the result of the algorithm. The first image is the scene under direct light. The second shows the indirect specular light created using the RMS. The third is the specular light created using the G-Buffer. And the fourth is the previous two combined into one.

4.3.2. Ambient Occlusion from Multiple Buffers

Proposed by Kostas Varids and Co. [13], the algorithm tries to create a more realistic ambient occlusion by using multiple views to calculate it. The final occlusion formula is:

$$occlusion(p) = \frac{\sum_{v=1}^{n} occlusion_v(p) * weight(v,p)}{\sum_{v=1}^{n} weight(v,p)}$$

Where $occlusion_v(p)$ is the occlusion in p from the view v, it has the same formula as the regular ambient occlusion. weight(v,p) is the weight function for the view v in point p. The weight function is the weighted average between two different weights.

The first one, which is more important and simple to determine, is the directional weight. It is really important for the fragment to appear in the buffer. To do this we take the dot product of the point's normal n and the direction from p to the center of the projection of the view v dir_v. Thus the function is:

$$dirWeight(v,p) = max(0, dir_v * n)$$

The other one is the distance weight. This one checks the distance from the fragment sampled to the point. This is done to remove any fragment that is really far from the point and doesn't add anything to the occlusion. The function is:

$$distWeight(v,p) = 1 - \frac{1}{K * r_{max}} \sum_{i=1} Kmin(||p - s_i||, r_{max})$$

Where r_{max} is the maximum distance a fragment can be from the point, K is the number of samples and $||p - s_i||$ is the distance from point p to the ith sample s.

Figure 4.11 show the example create by Kostas Vardis, and figure 4.12 shows an overview of the steps taken to create the new buffer.

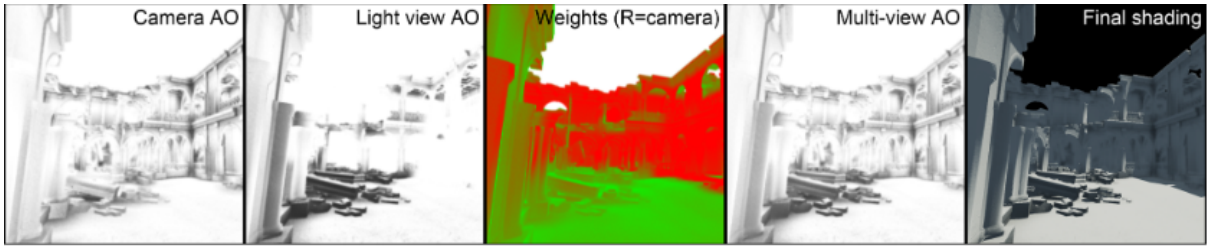


Figure 4.11: Example of the ambient occlusion from multiple sources[13]

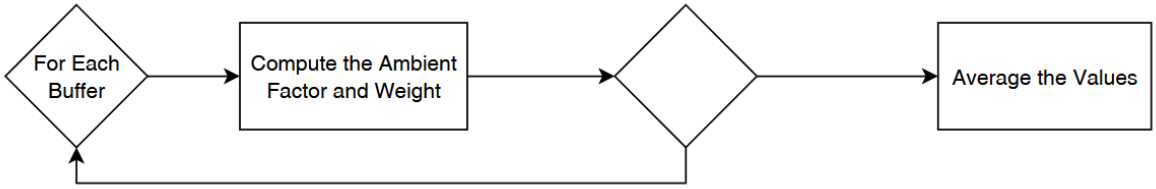


Figure 4.12: Overview of the algorithm

4.3.3. Indirect Diffuse Illumination

One Indirect Diffuse Illumination method was proposed by Cristian Lambru an Co. [14], they used something similar to what was presented in that RSM paper [6], but instead of doing 200 samples, they instead created a pattern of 10 that would be rotated every frame. Combining it with information from the previous frame can result in something similar, but faster. Although there exists a high chance of noise.

For my test, I kept the static one proposed by Dachsbacher and Stamminger in their RSM paper [6]. To do this we first need to take the fragments around a given point and consider them as sources of light. To randomly take those fragments, we first create a sampling patter and use it to take those values.

To take a random pixel we use:

$$s.x = p.x + radius * e1 * \sin(2\pi * e2)$$

$$s.y = p.y + radius * e1 * \cos(2\pi * e2)$$

Where p.x and p.y are the position of the pixel on the buffer, and s.x and s.y are the position of the sample pixel. e1 and e2 are random values and radius is the maximum distance from the point we can go. We also need to take into account that a closer pixel will have a higher impact on the light, as long as it is also close in space. This is done by dividing by the distance from the point to the sample.

The final value is

$$IndirectLight(p) = \sum_{i=1}^{N} e1_i^2 * flux_{s_i} \frac{max(\vec{d_{s_i}} * \vec{n_p}, 0) * max(-\vec{d_{s_i}} * \vec{n_p}, 0)}{||\vec{d_{s_i}}||^2}$$

Where $flux_{s_i}$ is the flux of the point s_i, that being the color multiplied by the specular component. d_{s_i} is the direction vector from the point spacial position of p to the spacial position of s_i. n_p is the normal vector of n. $||\ ||$ is the norm of the vector. $e1_i$ is also used as a weight, since a fragment adds more light the closer it is.

Figure 4.13: Example of the indirect Diffuse Illumination[6]

Figure 4.13 shows the result of Dachsbacher and Stamminger, the left image is direct illumination and shadow mapping, the center one is the indirect illumination, and the final one is the result from combining the two.

4.4. Multiple Lights

Adding more light doesn't change the algorithms that much. Light is additive, thus a simple addition of the values from each light is enough to compute how lit a pixel is. But more lights result in more computations needed for each of them. This can result in longer rendering time and less frames per second. As shown in figure 4.14.

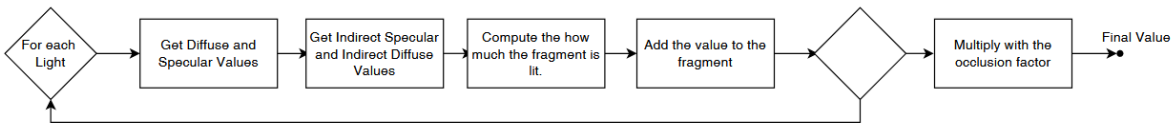


Figure 4.14: Overview on how to use multiple lights

The one that can cause the biggest drop in performance is the indirect diffuse illumination one, to be more precise, the amount of samples needed to be done. Meanwhile, the addition of another RSM to be rendered should not have a big impact on the performance, just increase the memory used.

Another aspect that should not impact the performance is the type of light, but that will be seen in chapter 7, as part of one of the tests. The only differences between

the two are the projection matrix used, and the fact that we have to take into account the angle the spotlight covers. The two types can coexist and the algorithms can work on either of them.

There can be different improvements to them, for example, unless the light or the object move, we do not to render the RSMs more than once. Another way is to compute the Indirect Specular Light from the G-Buffer once and then use it for the other lights if it exists.

Chapter 5. Technological considerations

5.1. Hardware

To implement those algorithms, a computer with high specs is needed. In my case, the computer I used has the following specs:

- GPU: NVIDIA GeForce RTX2060 SUPER
- CPU: AMD Ryzen 7 3700X 8-Core Processor
- RAM: 16GB

5.2. Software

For the implementation I have used C++ and OpenGL. OpenGL is a popular and highly used graphics API [17]. It was chosen due to its reliability, as well as the fact that it is open source and well documented. C++ was chosen because it pairs well with openGL, and because there are many libraries created to make the two work better.

One such library, that was used in this project is GLFW [18], a library created to make some processes, like creating a window, or receiving inputs a lot easier. This results in streamlined process of implementing anything in C++ and OpenGL.

Chapter 6. Implementation

6.1. Scene

The simpler parts of the code, like camera movement and the diffuse and specular components of lights were taken from an OpenGL tutorial [19]. Some additional algorithms, those being SSAO and Deffered Rendering were taken from learnopeng.com [16].

First I had to decide how the user could select how the scene should look, like the number of models or the number of lights. For that I decided to go with a configuration file where the user can write what they need. In the file the user can select the number of models, up to 5 and number of lights, up to 10.

For each model, the user has to specify the relative path to a ".gltf" file. They can also specify additional transformations that can be done to the object, those being translation, rotation and scaling, as well as how much of the object to be rendered.

To represent the light, I used 4 attributes: color, position and direction as vectors, and what type of light it is, direct or spotlight as an integer number. The decision of represent the light type as an integer number is to be able to add point lights for future extension. All of those traits have to be written by the user in the configuration file. In addition, the user can choose if the light is on or off at the start of the simulation.

```
\\Light Color
glm::vec4 lightColor;
\\Light Poisitoin
glm::vec3 lightPos;
\\Light Direction
glm::vec3 lightDir;
\\Light Type: 1 for direct light, 0 for spotlight
int lightType;
```

The file is read line by line, and it is where the values should be are fixed. The data read is, in order: the size of the window, which algorithms should be used, information about models and about lights. For each models they have to specify the: file location translations, rotations and the scale of the model and the percentage to be rendered. For light they have to specify what type of light it is, if it is on or off at start, the color, position and view direction of the light.

Another important thing that had to be implemented was the dynamic part of the light. For that I decided to let the user select which lights they want to move. Then they can either rotate the light, if it is a spotlight. They can also choose to turn the light on and off after selecting it. To show the selection, the light changes colour.

This part of the code checks for the number key corresponding to the light. To make things intuitive, I've decided that key '1' correspond to the first light in the list, which is light 0 in vector. The same is true for all light, except light 10 in the list, which

is connected to key '0'.

```
if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS) {
        lightMove[0] = 1;
        lightColor[0] = glm::vec4(1.0f, 0.5f, 0.31f, 1.0f);
}
```

When the key is released, the light return to the original color and can't be moved anymore.

```
if (glfwGetKey(window, GLFW_KEY_1) == GLFW_RELEASE) {
        lightMove[0] = 0;
        lightColor[0] = ogColor[0];
}
```

Every light has a projection matrix that will be used for the reflective shadow maps. The projection has to be changed when a light is rotated. The matrices are different based on the light.

For a direct light, we need to use an orthogonal projection matrix. Luckily, GLM has a function that let's us create one, as long as we give it the minimum and maximum x,y and z coordinates that should be taken when projecting the image. Then we compute the view matrix, which is used to transform coordinates from world space to light space. Again, GLM gives us a function to create it by giving it the position of the light, the direction and Up vector.

```
glm::mat4 orthgonalProjection = glm::ortho(-35.0f, 35.0f, -35.0f,
    35.0f, 0.1f, 75.0f);
lightView = glm::lookAt(lightPos, lightDir, glm::vec3(0.0f, 1.0f,
    0.0f));
lightProjection = orthgonalProjection * lightView;
```

For a spotlight thing are a bit different. Instead of an orthogonal matrix used for projection, we use a perspective matrix. To do this, we give it an angle, aspect and the near and far values, which represent how much the light can see from it's position. The other difference is that, for the view matrix, the view matrix requires the sum between the position of the light and it's direction to the center's position.

```
glm::mat4 perspectiveProjection = glm::perspective(glm::radians(90.0
    f), 1.0f, 0.1f, 100.0f);
lightView[i] = glm::lookAt(lightPos[i], lightPos[i] + lightDir[i],
    glm::vec3(0.0f, 1.0f, 0.0f));
lightProjection = perspectiveProjection * lightView;
```

The camera functions similarly to the lights. It has position and orientation, as well the ability to be rotated. In addition, the camera can be moved in six directions: up, down, left, right, forward and backward. Camera also has a projection matrix created similarly to the one for the spotlight. This has to be updated every time the camera moves or rotates.

The user can also turn the light on or off by pressing the Q key. The turning is done by using 1 minus the current state of the light

```
if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS) {
    \\Take only the first frame the key is pressed
        if (qPress == 0) {
                qPress = 1;
```

```
                for (int i = 0; i < noLights; i++) {
                        if (lightMove[i] == 1) {
                                renderLight[i] = 1 - renderLight[i];
                                recalculate[i] = 0;
                        }
                }
        }
}
\\Checks if the key was released
if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_RELEASE) {
        qPress = 0;
}
```

6.2. Standalone Algorithms

6.2.1. Deferred Rendering

To do deferred rendering, we first need to create a G-buffer and textures to hold the information. The framebuffers and textures are kept as indices, or pointers to the memory in the code. This is why they are declared as integers. We also need to bind the textures to a framebuffer.

The code bellow gives an example on how the buffer is bind.

```
//Declaring and generating G-Buffer
unsigned int gFBO;
glGenFramebuffers(1, &gFBO);
glBindFramebuffer(GL_FRAMEBUFFER, gFBO);

//Declaring and generating the texture used to hold the data
unsigned int gFluxMap, gNormalMap, gWorldMap, gDepthMap;

//Texture for world coordinates
glGenTextures(1, &gWorldMap);
glBindTexture(GL_TEXTURE_2D, gWorldMap);
//Declaring texture properties
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, rsmWidth, rsmHeight, 0,
   GL_RGB, GL_FLOAT, NULL);
//Binding the texture
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
   GL_TEXTURE_2D, gWorldMap, 0);
GLenum gDrawBuffers[] = { GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, gDrawBuffers);
```

The shaders are used to create the texture for the buffers. Neither of them do too much, other than projecting the vertices to know where they should be placed in the texture. The flux buffer combines the information from the diffuse texture, the color, and the specular texture, which has the specular value. Vertex Shader, which takes the data that will be placed in the buffers:

```
//World Space Coordinates
FragPos =  (model * addtranslation * translation  * addRot *
   rotation * addScale * scale * vec4(aPos, 1.0f)).xyz;
```

```
//Normal
mat3 normalMatrix = transpose(inverse(mat3(model * addtranslation *
    translation  * addRot * rotation * addScale * scale)));
Normal = normalMatrix * aNormal;
//Position for the depth buffer
gl_Position = camMatrix * model * vec4(aPos, 1.0f);
//Color of the vertex
Color = aColor;
//Binding the texture of the fragment
texCoord = mat2(0.0, -1.0, 1.0, 0.0) * aTex;
```

Frangment Shader, which places the data that in the buffers:

```
//World Coordinates Buffer
RsmPos = FragPos;
//Normal Buffer
RsmNormal = normalize(Normal);
//Flux Buffer
RsmFlux.rgb =  texture(diffuse0, texCoord).rgb * texture(specular0,
    texCoord).a;
RsmFlux.a = texture(specular0, texCoord).a;
```

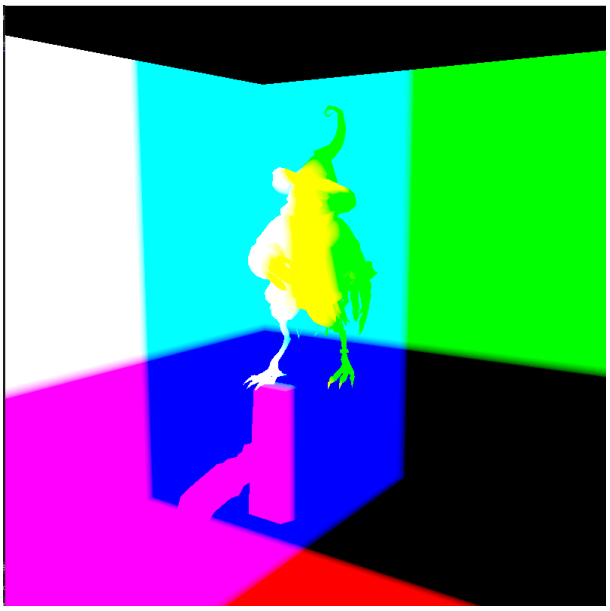


Figure 6.1: One of the textures created by the G-buffer: World Coordinates

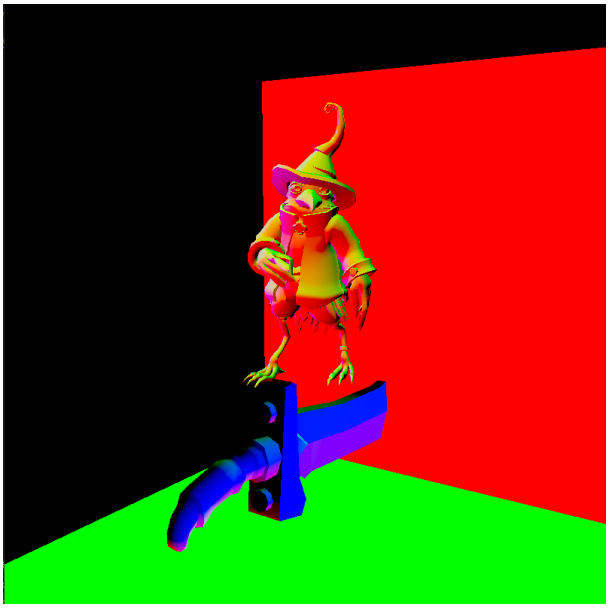


Figure 6.2: One of the textures created by the G-buffer: the Normals

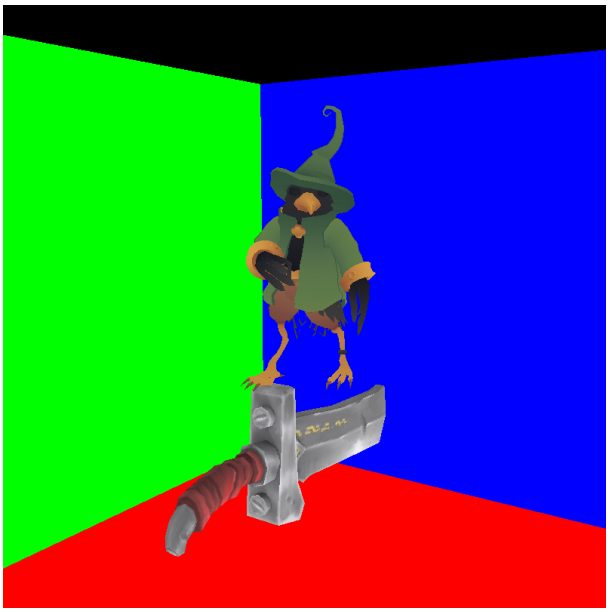


Figure 6.3: One of the textures created by the G-buffer: the Flux

The reason why the depth map is not shown in any figure is because it does not look good when you create a depth map from the camera's perspective. It works, but the values are really similar and we get an almost completely red screen.

6.2.2. Reflective Shadow Maps

Reflective Shadow Maps are almost identical to deferred rendering, and this is seen in the code. We first create a framebuffer and four textures for each light. We then need to render each object from that lights perspective. This means, that for each additional light, we need to do one more render of the objects.

The shaders are identical to the ones in defered rendering. Vertex Shader, creates the data to send to the fragment shader:

```
//World Space Coordinates
FragPos =  (model * addtranslation * translation  * addRot *
    rotation * addScale * scale * vec4(aPos, 1.0f)).xyz;
//Normal
mat3 normalMatrix = transpose(inverse(mat3(model * addtranslation *
    translation  * addRot * rotation * addScale * scale)));
Normal = normalMatrix * aNormal;
//Position for the depth buffer
gl_Position = lightProjection * model * vec4(aPos, 1.0f);
//Color of the vertex
Color = aColor;
//Binding the texture of the fragment
texCoord = mat2(0.0, -1.0, 1.0, 0.0) * aTex;
```

Frangment Shader, takes the data and puts it into framebuffers:

```
RsmPos = FragPos;
RsmNormal = normalize(Normal);
RsmFlux.rgb =  texture(diffuse0, texCoord).rgb * texture(specular0,
    texCoord).a;;
RsmFlux.a = texture(specular0, texCoord).a;
```

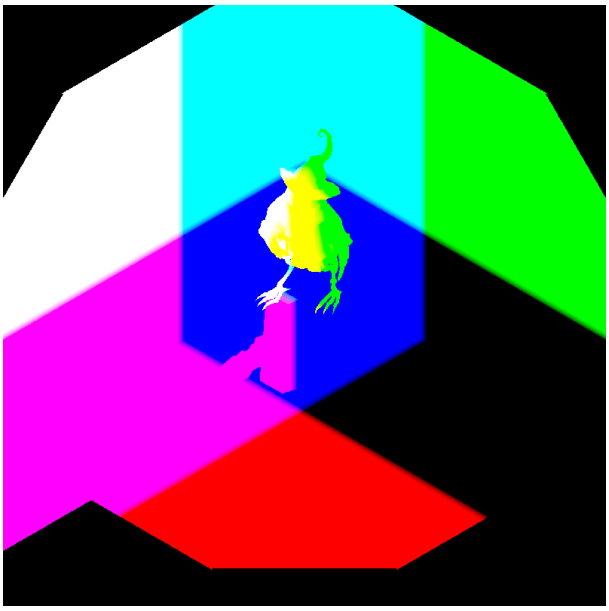


Figure 6.4: One of the textures created by the RSM: World Coordinates

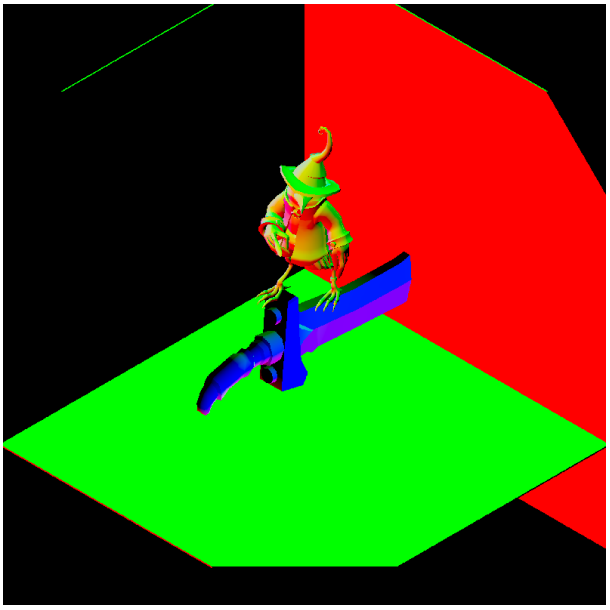


Figure 6.5: One of the textures created by the RSM: the Normals

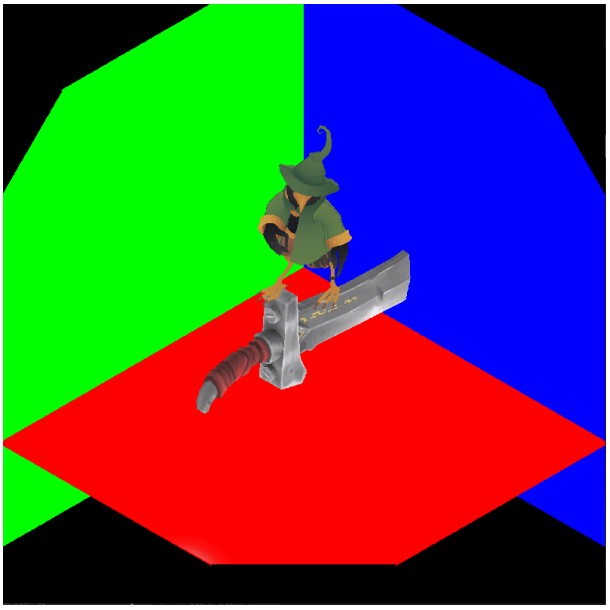


Figure 6.6: One of the textures created by the RSM: the Flux

Figure 6.7: One of the textures created by the RSM: the depth Buffer

There exists a way to slightly improve the rendering process. We can skip the rendering of the RSM under two conditions. If both the objects and the light are static, in which case it would be useless to remake it, as we would generate the same textures. The other is when the light is turned off, in which case we can skip the rendering because we have no use for it.

6.2.3. Ambient Occlusion

For this we need yet another buffer. To create it, we first must create a sampling kernel, the kernel is a vector formed of 3D vectors with the position of pixels. X and Y's values are between 1 and -1, Z is form 0 to 1. Z is from 0 to 1 because we use only half of the sphere around the point. This can be done by generating random numbers.

To create the random numbers we use the "rand()" function from C++, which creates a random number from 0 to 32767. To get numbers from -1 to 1, we divide the result with the maximum value of "rand()". This way we get values from 0 to 1. We them multiply the result with 2 and subtract 1. We don't do the final step for Z.

We also want to have the positions closer to the point more important. So we modify the vector to point somewhere closer in the same direction as before. This is done by first normalizing the vector, then multiplying it with a value.

```
glm::vec3 ssaoKernel[64];
for (int i = 0; i < 64; i++) {
    //Creating the kernel.
        ssaoKernel[i] = glm::vec3(((float)rand()) / ((float)RAND_MAX
            ) * 2.0 - 1.0, ((float)rand()) / ((float)RAND_MAX) * 2.0
            - 1.0, ((float)rand()) / ((float)RAND_MAX));
        ssaoKernel[i] = glm::normalize(ssaoKernel[i]);
        float scale = i / 64.0f;
        scale = 0.1f + scale * scale * 0.9f;
        ssaoKernel[i] *= scale;
}
```

We also create a kernel of random rotations. That way we can have fewer samples , instead we can rotate the previous kernel and get more samples that way.

```
//Creating the rotation texture
glm::vec3 ssaoNoise[16];
for (int i = 0; i < 16; i++) {
        ssaoNoise[i] = glm::vec3(((float)rand()) / ((float)RAND_MAX)
            * 2.0 - 1.0,
                ((float)rand()) / ((float)RAND_MAX) * 2.0 - 1.0,
                0.0f);
}
```

We use the kernel in the fragment shader to sample around the fragment and determine the occlusion factor. For each fragment we take the points around it. We compare if the fragment is behind the sample, and if it is we add to the occlusion factor. We also take into account how far the two points are and consider that a weight.

Finally we do a the average of the values. Since we use the value to decrease the amount of light a fragment receives, we also need to subtract it from 1. That way, the higher the occlusion factor, the smaller the amount of light will be when it gets multiplied with the occlusion factor.

```
vec3 tangent = normalize(randomVec - normal * dot(randomVec, normal)
    );
vec3 bitangent = cross(normal, tangent);
mat3 TBN = mat3(tangent, bitangent, normal);

float occlusion = 0.0f;

for(int i = 0; i < kernelSize; ++i)
{
    // get sample position
    vec3 samplePos = TBN * samples[i];
    samplePos = fragPos + samplePos * radius;

    // project the sample onto the G-Buffer
    vec4 offset = vec4(samplePos, 1.0);
    offset = projection * offset;
    offset.xyz /= offset.w;
    offset.xyz = offset.xyz * 0.5 + 0.5;

    // get sample depth
    vec3 texturePos = texture(textureUsed, offset.xy).xyz;

    float sampleDepth;
    sampleDepth = texturePos.z;

    // checks for distance from point and add the value to the
       occlusion factor
    float rangeCheck = smoothstep(0.0, 1.0, radius / abs(fragPos.z -
        sampleDepth));

    occlusion += (sampleDepth >= samplePos.z + bias ? 1.0 : 0.0) *
       rangeCheck;
```

```
}
return 1.0 - (occlusion / kernelSize);
```

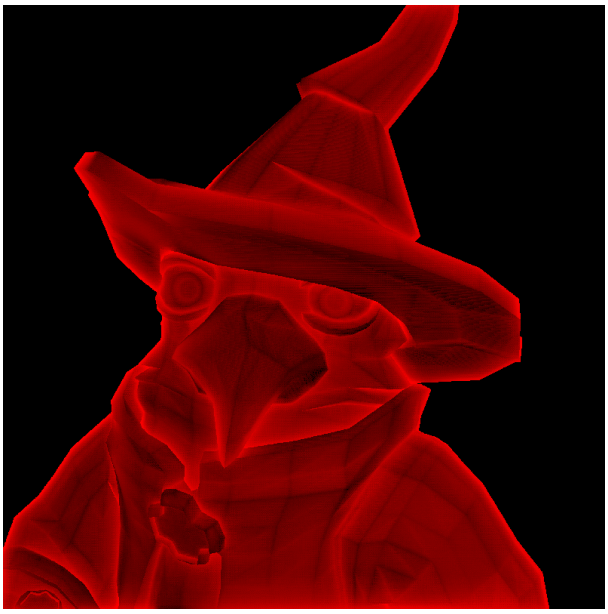


Figure 6.8: Textures created by the SSAO

There is one small problem that appears from this algorithm, and it can be seen only when looking closely at the object. As seen in figure 6.9, there exists some artifact created from the algorithm.

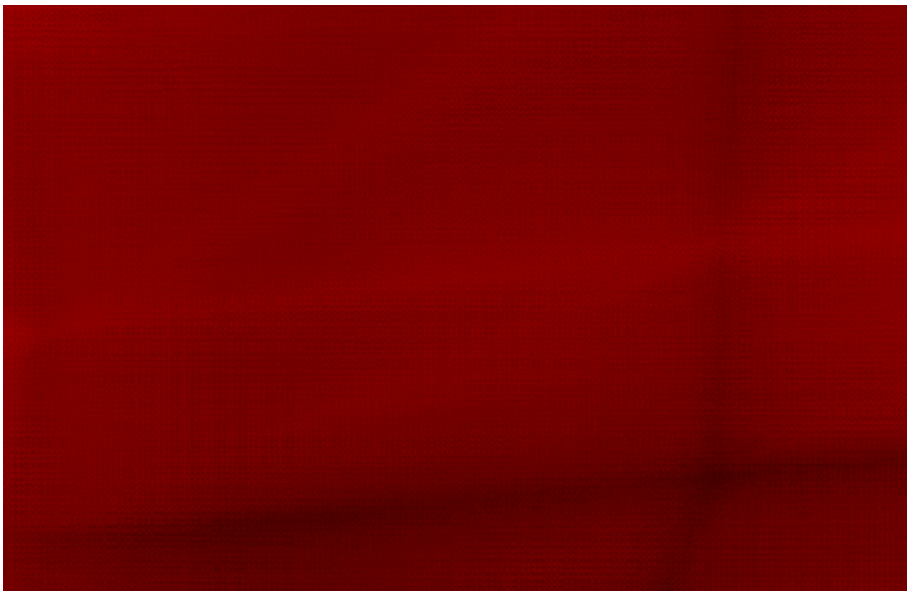


Figure 6.9: Section of the SSAO texture before blur. While a bit hard to see, there are small lines that seem to split the texture in many small squares.

A blur is required to be performed on the texture after it's creation. A close up of the new texture can be seen in figure 6.10. The full picture is shown in figure 6.11. The blur is done by averaging the pixels close together.

```
vec2 texelSize = 1.0 / vec2(textureSize(ssaoInput, 0));
float result = 0.0;
for (int x = -2; x < 2; ++x)
{
    for (int y = -2; y < 2; ++y)
    {
        vec2 offset = vec2(float(x), float(y)) * texelSize;
        result += texture(ssaoInput, TexCoords + offset).r;
    }
}
FragColor = result / (4.0 * 4.0);
```

Figure 6.10: Section of the SSAO texture after blur. Compared to 6.9, the texture looks smoother.

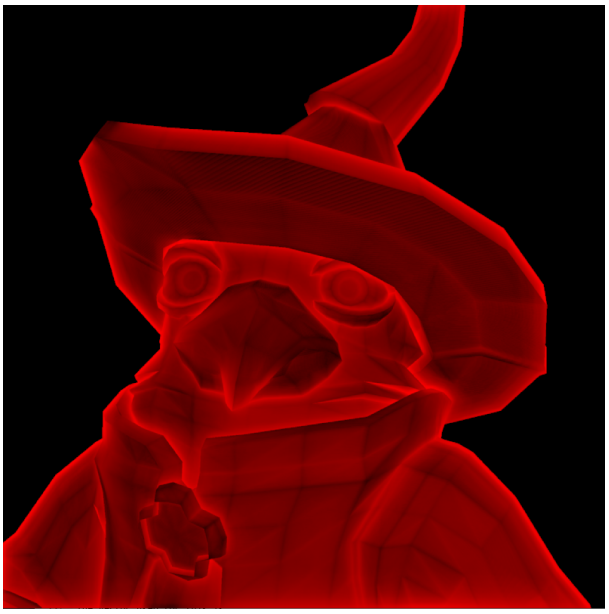


Figure 6.11: Textures created by the SSAO after Blur

6.2.4. Screen Space Reflections

For this one, ray tracing is used to determine an intersection with the geometry. In order to make everything less computational heavy, the check is done against the G-Buffer. We follow the ray until be get really close to the geometry, then use binary search to approximate the intersection. Everything is done in the fragment shader. The vector used for this is:

```
vec3 reflected = normalize(reflect(normalize(camPos), normalize(
    Normal)));
```

Where camPos is the position of the camera, or the user's point of view, and the Normal is the normal of the point.

We start by doing a broader search, where we go in the given direction with a bigger distance. We do this until we are close to a geometry. This can be done simply by comparing their depth values. If the two values are close, it means we found it. If our depth is close to that of the geometry, but are behind it, we start doing a binary search to find that intersection.

```
vec4 RayCast(in vec3 dir, inout vec3 hitCoord, out float dDepth){
        dir *= rayStep;

        float depth = 0.0f;
        int steps = 0;
        vec4 projCoord = vec4(0.0f);

        for(int i=0;i<maxSteps;i++){
        //Moves in the given direction and projects the coordinates
            on the GBuffer
                hitCoord += dir;
                projCoord= camMatrix * vec4(hitCoord, 1.0f);
                projCoord.xy /= projCoord.w;
                projCoord.xy = (projCoord.xy + 1.0f)/2.0f;
                depth = texture(gWorldCoordMap,projCoord.xy).z;

        //Checks if the geometry is too far, if it is, it moves on
            to the next coordinates
                if(depth > 1000.0f)
                        continue;

                dDepth = hitCoord.z - depth;

        //chceks if we are close to some geometry
                if((dir.z - dDepth) < 1.2f){
        //checks if we got passed it
                        if(dDepth<= 0.0){
                                return vec4(binSearch(dir,hitCoord,
                                    dDepth),1.0f);
                        }
                }
                steps++;
        }
```

```
        return vec4(projCoord.xy, dDepth,0.0f);
}
```

The reason why I use a vector of four elements instead of one of 3 is because I use the last value to determine if an intersection was found or not. This is used to determine if this has to be computed using reflective shadow maps.

For the binary search we keep the same directional vector, but change the distance we travel at each iteration. This part, while not really needed, gives more precision to the algorithm. The cost is that it needs more time to find the intersection.

```
\\Function used to do the binary srearch
vec3 binSearch(inout vec3 dir, inout vec3 hitCoord, inout float
   dDepth){
        float depth;
        vec4 projCoord;

        for(int i=0;i<numBinarySearchSteps;i++){
        //Projects the given coordinate on the G-Buffer
                projCoord = camMatrix * vec4(hitCoord,1.0f);
                projCoord.xy = (projCoord.xy/projCoord.w +1.0f)/2.0f
                    ;
                depth = texture(gWorldCoordMap,projCoord.xy).z;

        //Calculates the distance
                dDepth = hitCoord.z - depth;

        //Halves the distance used to travel
        dir *= 0.5;

        //Cheks if the current hitCoord is in front of or behind the
            geometry and moves towards the geometry
        if(dDepth > 0.0)
            hitCoord += dir;
        else
            hitCoord -= dir;
        }

        projCoord = camMatrix * vec4(hitCoord,1.0f);
        projCoord.xy = (projCoord.xy/projCoord.w +1.0f)/2.0f;
        return vec3(projCoord.xy,depth);
}
```

The results of the algorithm are the color value of the point the reflected vector intersect. This represents what the camera would see if the point was a mirror or reflective surface.

6.3. Combining the Algorithms

6.3.1. Indirect Specular Lighting

In order to calculate the Indirect Specular Lighting we extend the Screen Space Reflection code, from 6.2.4. What we do is check if the returned value is 0, if it is we

do the same process for the RSM. This process is done for each light, we first check the camera value, then change to light view. Then a binary search is used to be as close to the intersection as possible. This step is not really required to do, this is because RSM tend to be less detailed than the G-Buffer. For example, the light can be really far and see smaller objects that what they really are. The only change is the use of the light projection matrix and the reflective shadow map of that light, instead of the camera matrix and the G-Buffer. So let's go through the steps yet again: We first go in the given direction and get the depth value.

```
projCoord= lightProjection[light] * vec4(hitCoord, 1.0f);
projCoord.xyz /= projCoord.w;
projCoord.xy = (projCoord.xy +1.0f)/2.0f;
depth = texture(shadowMap[light],projCoord.xy).r;
```

We then compare the depth of our point with the one from the shadow map and see how close they are, and if it is wee start a binary search.

```
if(depth > 1000.0f)
        continue;
dDepth = hitCoord.z - depth;
if((dir.z - dDepth) < 1.2f){
    //checks for colision
        if(dDepth<= 0.0){
            return vec4(RsmBinSearch(light,dir,hitCoord,dDepth),1.0f
                );
    }
}
```

As for the binary search we do the same thing, we start with an initial distance that we can travel. The distance get smaller and smaller until we either hit the geometry or get past the threshold of steps.

```
vec3 RsmBinSearch(in int light,inout vec3 dir, inout vec3 hitCoord,
   inout float dDepth){
        float depth;
        vec4 projCoord;

        for(int i=0;i<numBinarySearchSteps;i++){
                projCoord= lightProjection[light] * vec4(hitCoord,
                    1.0f);
                projCoord.xyz /= projCoord.w;
                projCoord.xy = (projCoord.xy +1.0f)/2.0f;
                depth = texture(shadowMap[light],projCoord.xy).r;

                dDepth = projCoord.z - depth;

        dir *= 0.5;
        if(dDepth > 0.0)
            hitCoord += dir;
        else
            hitCoord -= dir;
        }

        projCoord = lightProjection[light] * vec4(hitCoord,1.0f);
```

```
        projCoord.xy = (projCoord.xy/projCoord.w +1.0f)/2.0f;
        return vec3(projCoord.xy, depth);
}
```

This way, we should get more point that are indirectly lit from somewhere else. For the final value we take into account the specular properties of the point that got intersected by the reflection. This is where we also have to take into account things like shadows.

```
indirectSpecularTotal = (1-b) * texture(gFluxMap, specCoord.xy) *
    (1.0f - shadow) + b * texture(rsmFluxMap[light], rsmSpecCoord.xy)
      * (1.0f - shadow);
```

This way we can get reflection of thing that the camera does not see, but the light does. This is rather useful for creating more realistic reflections witout needing to know the entire geometry.

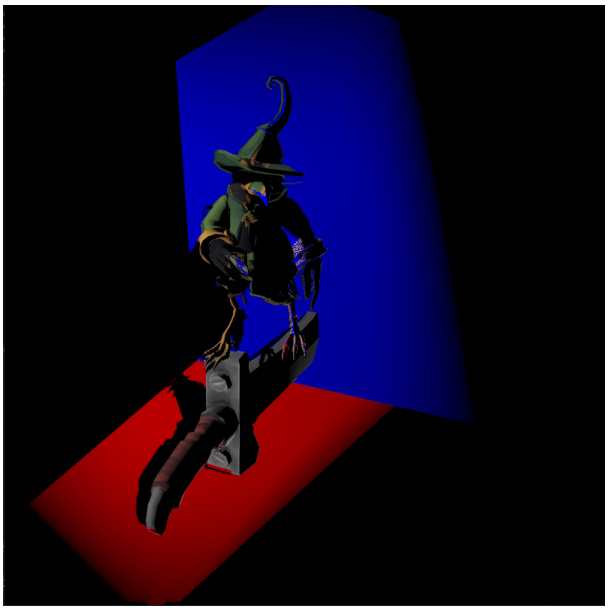


Figure 6.12: Result of Indirect Specular algorithm on screen space

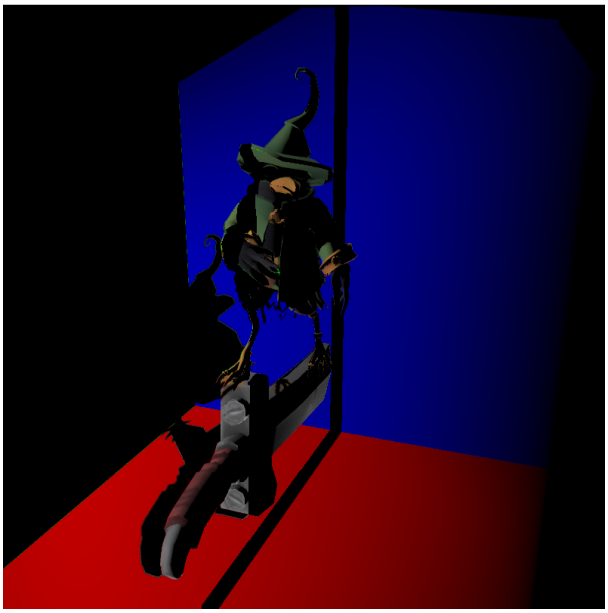


Figure 6.13: Result of Indirect Specular algorithm on reflective shadow maps

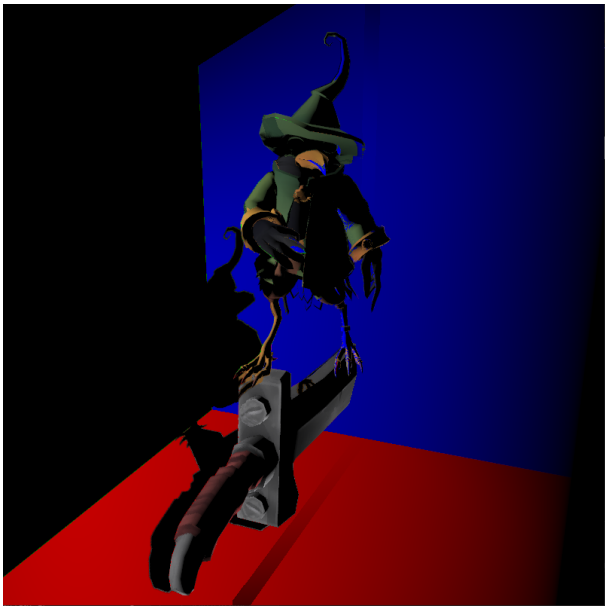


Figure 6.14: Result of the total Indirect Specular algoritm

The three figures (Figures 6.12 to 6.14) show the result of the algorithm. Some attenuation should be added based on the distance, but for the sake of making thing easier to see, I did not do that. Figure 6.15 shows an example of the reflection hitting an object that the camera does not see, but the light does. The green pixels on the foot are reflections from the wall behind the camera. A wall that the light can see in it's reflective shadow map.

Figure 6.15: Example of the reflection

6.3.2. Ambient Occlusion from Multiple Buffers

To do Ambient Occlusion with more buffers we need to take into account the two weights. The first one is the directional weight, which is determined by the point appearing in the buffer:

```
float directionalWeight = max(0.0f,dot(normal, center - fragPos))
    ;
```

The second on is the distance weight, which is determined by an average of the distances from out point to the sample points:

```
distWeight += min(length(fragPos - samplePos),radius);

distWeight = 1 - distWeight/(kernelSize * radius);
```

To do the final weight we use a bias values. This is a value chosen by the user. In my case, I choose to give more importance to the fact that the point is in the buffer, since this will give a better accuracy for its occlusion factor.

```
finWeight = (1 -  weightBias) * directionalWeight + weightBias *
    distWeight;
```

Other than that we do the same things as the regular ambient occlusion. We start by projecting our point on the buffer and get the values of the point around it.

```
vec4 offset = vec4(samplePos, 1.0);
offset = projection * offset; // from view to clip-space
offset.xyz /= offset.w; // perspective divide
offset.xyz = offset.xyz * 0.5 + 0.5; // transform to range 0.0
    - 1.0
vec3 texturePos = texture(textureUsed, offset.xy).xyz;
```

We then compare the depth of the point with the one of the sample points.

```
        sampleDepth = texturePos.z;
        float rangeCheck = smoothstep(0.0, 1.0, radius / abs(fragPos
            .z - sampleDepth));
        occlusion += (sampleDepth >= samplePos.z + bias ? 1.0 : 0.0)
             * rangeCheck;
```

At the end we return the average of the occlusion for each point, multiplied with the total weight.

```
return finWeight * (1.0 - (occlusion / kernelSize));
```

Then, in the main function we add the factors together and do the average of the occlusions from each buffer. This gives us the final values of the occlusion factor.

```
// get input for SSAO algorithm
vec3 fragPos = texture(gWorldCoordMap, TexCoords).xyz;
vec3 normal = normalize(texture(gNormMap, TexCoords).rgb);
vec3 randomVec = normalize(texture(texNoise, TexCoords *
    noiseScale).xyz);

// iterate over the sample kernel and calculate occlusion factor
float totOcclusion = getOcclusion( fragPos, normal, randomVec,
     gWorldCoordMap, camMatrix, camPos);
for(int i=0;i<noLights;i++){
    //Check if the light is on or off
    if(renderLight[i] == 1){
        totOcclusion += getOcclusion( fragPos, normal,
            randomVec, rsmWorldCoordMap[i], lightProjection[i],
            lightPos[i]);
    }
}
//average the occlusion facro
totOcclusion/= totalWeight;
FragColor = totOcclusion;
```

Figure 6.17: SSAO map created by having two lights

Figure 6.16: SSAO map created by having one light

Of course, just like with the simple one, there is still one more step do be done. And that is the blur of the screen space ambient occlusion buffer. This is to remove some artifacts (see figures 6.9 and 6.10). To do that I used a simple blur function.

```
vec2 texelSize = 1.0 / vec2(textureSize(ssaoInput, 0));
  float result = 0.0;
  for (int x = -2; x < 2; ++x)
  {
      for (int y = -2; y < 2; ++y)
      {
          vec2 offset = vec2(float(x), float(y)) * texelSize;
          result += texture(ssaoInput, TexCoords + offset).r;
      }
  }
  FragColor = result / (4.0 * 4.0);
```

6.3.3. Indirect Diffuse Illumination

To do this, we first need a bunch of randomly generated numbers between 0 and 1. Those number will represent the random directions we will take around the point. The first one, e1, represents how far we go, or the distance. It will be multiplied with a given value, named radius. The second one, e2, represents the angle between the point, it will be multiplied with 2Π to get the angle for the direction.

```
float e1[200];
float e2[200];
for (int i = 0; i < 200; i++)
{
    //First set of numbers
        e1[i] = ((float)rand()) / ((float)RAND_MAX);
    //Second set of numbers
        e2[i] = ((float)rand()) / ((float)RAND_MAX);
```

```
}
```

After that we use the RSMs to take the fragments around the point and use them as light sources. In short, we need to see how much they lit the point. This is based on distance. To do that we first project our point on the respective reflective shadow map.

```
vec4 rsmCoords= lightProjection[ligth] * vec4(crntPos, 1.0f);
    rsmCoords.xy /= rsmCoords.w;
    rsmCoords.xy = (rsmCoords.xy + 1.0f)/2.0f;
```

Then we go in the direction the two values created previously point to. To do so, we use the sine and cosine of $2\pi * e2$ to get the the X and Y distances. This is because the radius multiplied by e1 will give the hypotenuse, by using the values of the sine and cosine we can get the two other sides of a triangle.

```
for(int i=0;i<200;i++){
    vec2 newCoords = vec2(rsmCoords.x + r * e1[i] * sin(2*PI*e2[i])
        ,rsmCoords.y + r * e1[i] * cos(2*PI*e2[i]));
```

We use the new point and do the formula explained in chapter 4.3.3, where we take into account the distance between the two points. We also must consider the dot product between their normal and the direction vector from one to the other.

```
vec3 res= texture(rsmFluxMap[ligth],newCoords).rgb * texture(
    rsmFluxMap[ligth],newCoords).a * max(dot(Normal,-distance),0) *
    max(dot(texture(rsmNormMap[ligth],newCoords).rgb,distance),0) /
    pow(length(distance),4);
```

The final step before adding it to the sum is to multiply with the square of e1. This is do add a weight to each sample point.

```
res *= pow(e1[i],2);
sum+=res;
```

Finally, we divide the sum with the number of samples.

```
sum\=2000;
return vec4(sum,1.0f);
```

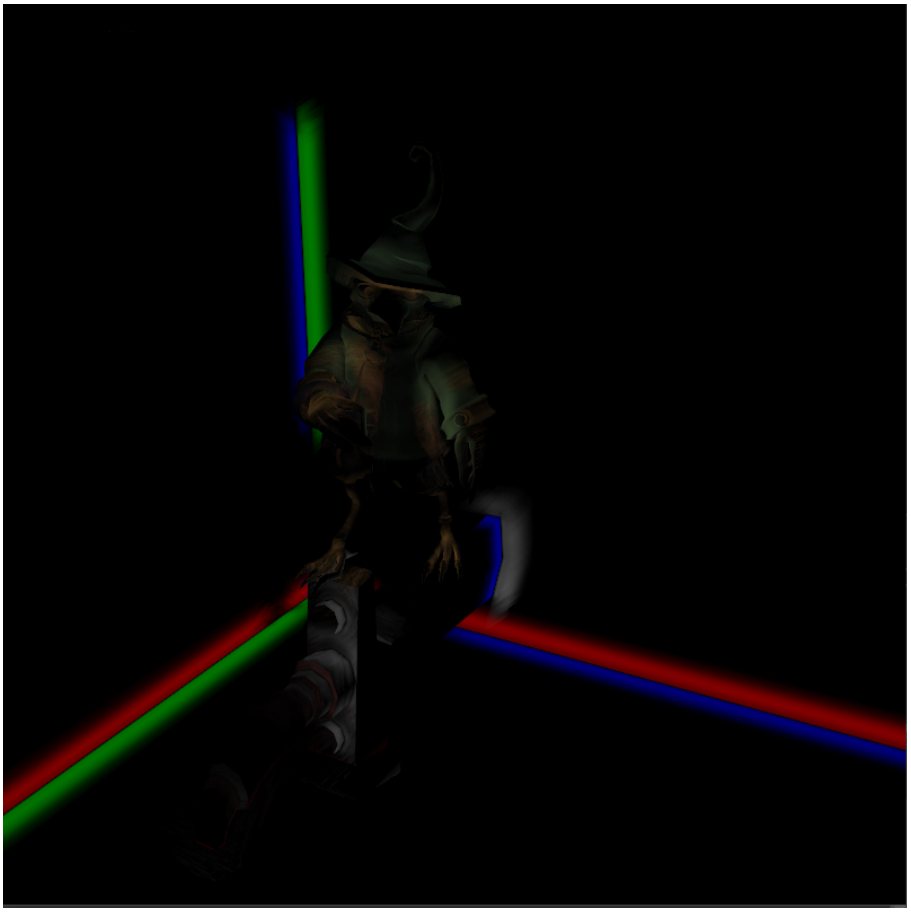


Figure 6.18: Result of indirect diffuse lightning algorithm

It has to be stated that figure 6.18 above has be contrast and brightness increased to make the colors pop a bit more. The result is that points closer to the intersection of two surfaces have the highest impact on the result. That can be seen at the intersection of the three surfaces in the back, where on surface gets the color of the other one. Similarly, while a bit hard to spot, the wall to the left gets a bit of light from the sword in the area they intersect.

Chapter 7. Testing and Validation

7.1. Experiments Presentation and Evaluation Metric

When it comes to rendering, the best way to experiment with an algorithm is to change the parameters used for the render, those being the number of vertices and fragments that need to be rendered and the resolution of the image. In addition, in our case we have to take into account three more thing, the number of lights, the types of lights, and if the lights move or not.

The metric for evaluation was the number of frames per second. or FPS that the programs runs at. This was done by averaging the number of frames for a period of time. Counting the number of frames is done by using a counter to keep track how many frames have passed, after a period of time, the counter is averaged with how much time has passed and is added to a sum. After one hundred of these sums, a final average is done to determine the average of frames per second.

For the scene I used a diorama of ancient Rome(figure 7.1) [20]. I used that model due to it's size, 2 million vertices, forming 4 million triangles. The camera and lights are placed in such a way that almost the entire object appears in their buffers. As for what I tested, I decider to test four thing: the amount of the object being rendered, the number of lights, the resolution of the screen and the number of lights moving at the same time.

Figure 7.1: The Image shows object used for testing [20]

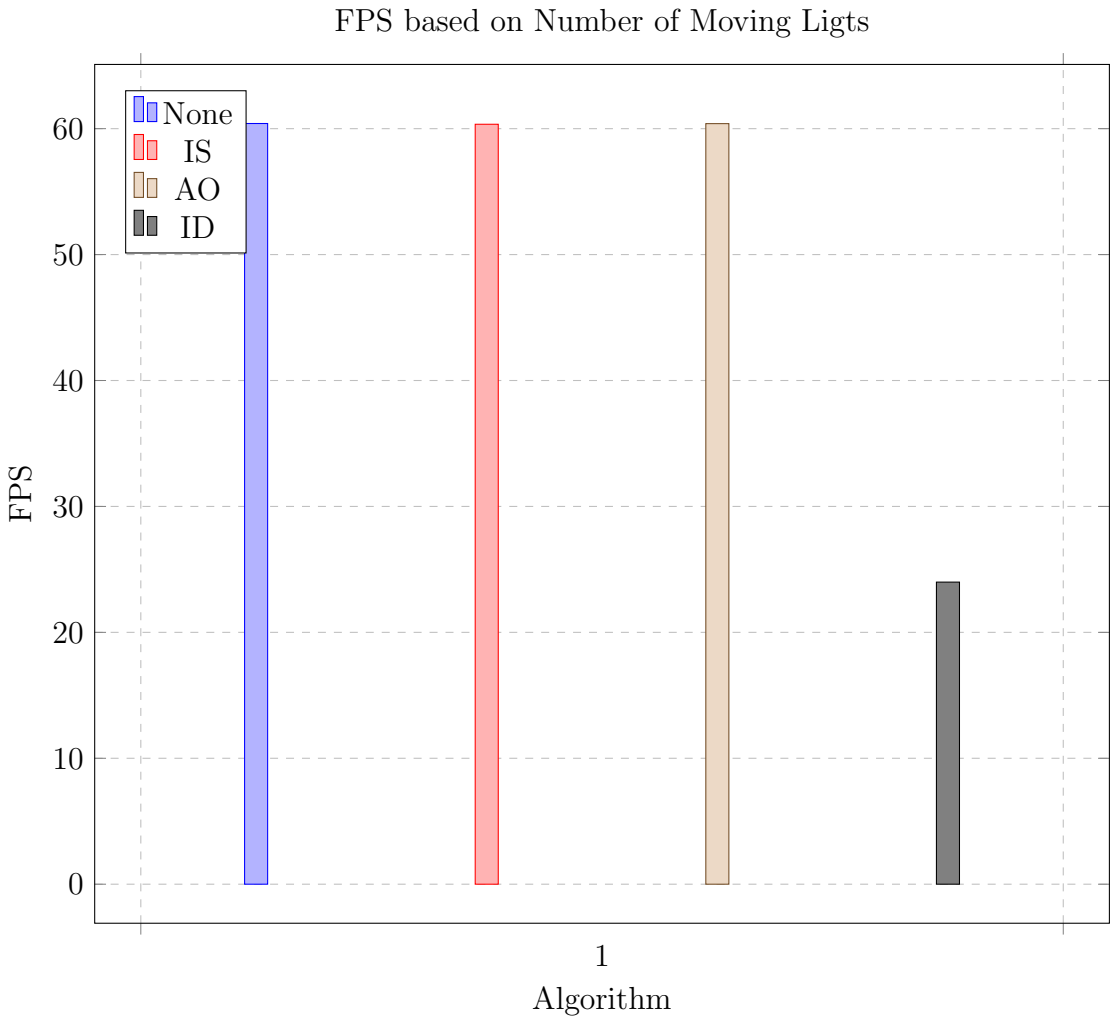
7.2. Algorithm Validations

For these test I have rendered the entire object, at 800x800 resolution and one light. For the sake of space, I am going to use the initials of each algorithms in the following table and graphic: indirect specular lighting will be noted as IS, ambient occlusion as AO and indirect diffuse illumination as ID.

The reason why only the combinations one are tested is because all of them use reflective shadow maps and deferred rendering plus one of the other algorithms. In this case RSMs and deferred rendering become constants.

Algorithms	Frames/Second(FPS)	Avg Time per Frame(ms)
None	60.41	16.60
IS	60.35	16.71
AO	60.40	16.61
ID	23.99	42.28
IS + AO	59.93	16.98
IS + ID	18.47	54.89
AO + ID	22.63	44.77
IS + AO + ID	18.08	56.00

Table 7.1: Frames and frame duration for different algorithms being used



We can see that indirect diffuse illumination has the highest impact on performance, beating both Indirect Specular Lighting and Ambient Occlusion by a big margin. This is because of the high number of samples used to determine the value for each fragment. On the other side, Ambient Occlusion barely makes a dent in performance. This is because it works with the buffers, not with the object itself.

This leaves Indirect Specular Lighting in the middle, this is because it works on fragment. Furthermore it has to do one or two ray tracing processes for each of them.

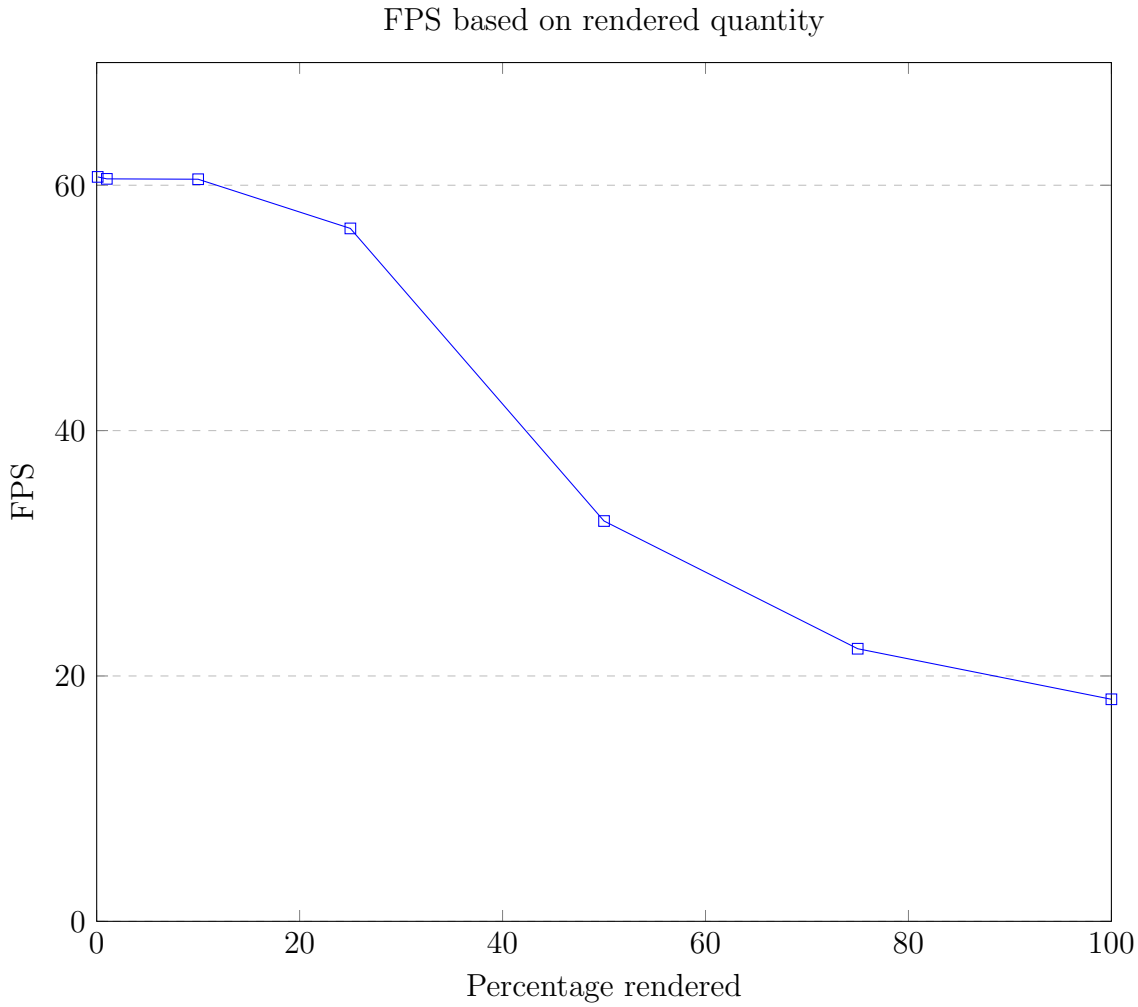
7.3. Size of the Object

For these tests I have rendered different percentages of the object. For this, in all tests, there was only 1 light, with no additional movement to the light, and a 800x800 window. The frame buffers used for the RSMs are 2048x2048. The result were rounded to two decimals.

Percentage Rendered(%)	Frames/Second(FPS)	Avg Time per Frame(ms)
0.1	60.68	16.53
1	60.52	16.54
10	60.49	16.60
25	56.48	18.19
50	32.63	30.98
75	22.22	45.44
100	18.10	55.96

Table 7.2: Frames and frame duration for different percentage of the object being rendered

To no one's surprise, how much you have to render directly impacts the performance. A graph can show better how much the amount rendered and the performance are connected.



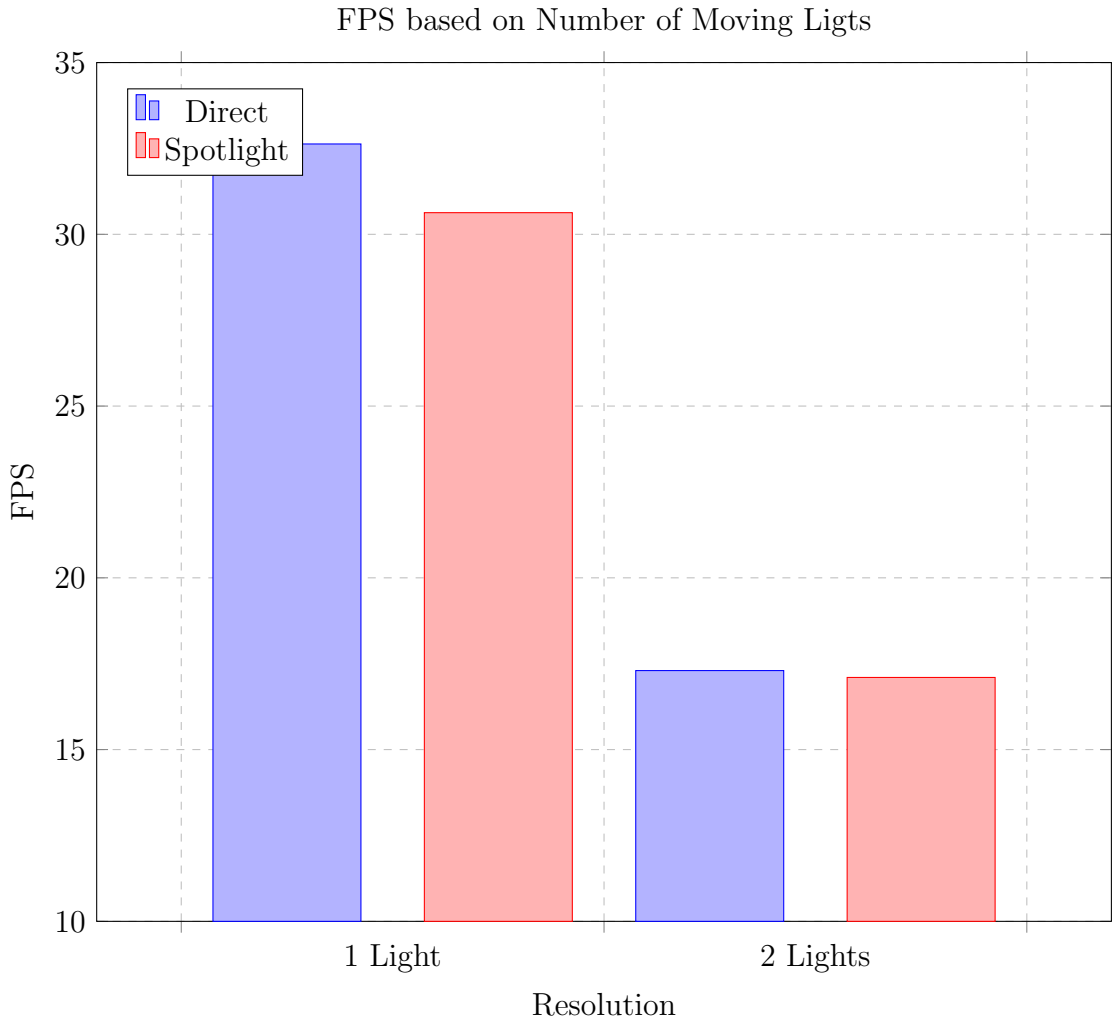
As the graph shows, there is a huge slope from 10% to 50%, then a smaller one from there. This can happen because, while we add more primitives to be rendered, many of them can end up either covering previous ones, or be already covered. Thus removing them from the latter stages of rendering.

7.4. Adding Lights

For these tests I have used a different number of lights to render the object. The object was rendered at 50% of it's total triangles. The render was done at 800x800. The frame buffers used for the RSMs are 2048x2048. Lights do not move. The test has two parts, the first is whether a direct light and a spotlight affect rendering. The second was increasing the number of light, no matter what type it is.

Light Type & Number	Frames/Second(FPS)	Avg Time per Frame(ms)
Direct 1	32.63	30.98
Spotlight 1	30.63	33.17
Direct 2	17.30	58.47
Spotlight 2	17.10	59.35

Table 7.3: Frames and frame duration for different number of lights of a certain type



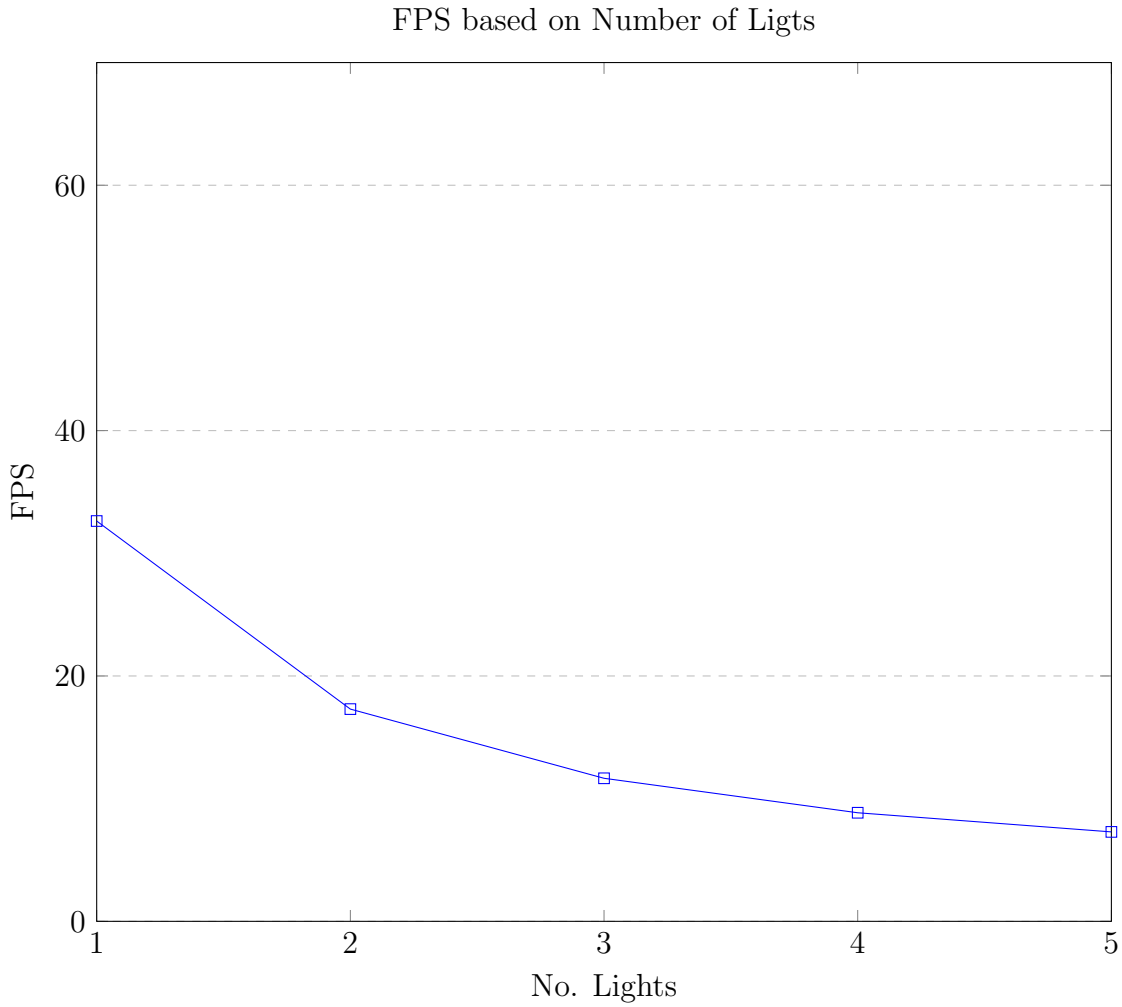
From the table 7.2 we can see that the type of light has a small impact. With the spotlights being a bit slower. This can be because the spotlight has a bit more steps added when it comes to rendering. To be more prcise it has to check if the fragment is in the cone of the light and find the angle between the two.

As for the number of lights, I have render the same scene with the number of light from one to five. Zero lights would have been pointless to test, since no fragment processing would have been done. The type of the light, in order they were added was: direct, spotlight, spotlight, direct, spotlight. The light were all around the object, above it.

No.Lights	Frames/Second(FPS)	Avg Time per Frame(ms)
1	32.63	30.98
2	17.30	59.03
3	11.66	86.85
4	8.85	115.42
5	7.29	144.49

Table 7.4: Frames and frame duration for different number of lights

Table 7.3 shows that the number of lights impact the performance. And it seems that the more lights are added, the less the drop in FPS. Let's put them in a graph.



As seen in the graph, each additional light has less impact on the render. This is rather surprising, as someone would expect the decrease to be linear.

7.5. Changing Resolution

Another important factor is the resolution of the rendering. For this test I have kept the object at 50% render, I've also left three light on. There were 5 tests, one for a 800x800 resolution, and the rest going from 360p to 1080p.

Resolution	Frames/Second(FPS)	Avg Time per Frame(ms)
480x360	24.39	26.10
640x480	24.15	41.88
800x800	11.66	86.85
1280x720	13.76	76.92
1920x1080	8.14	124.24

Table 7.5: Frames and frame duration for different Resolution

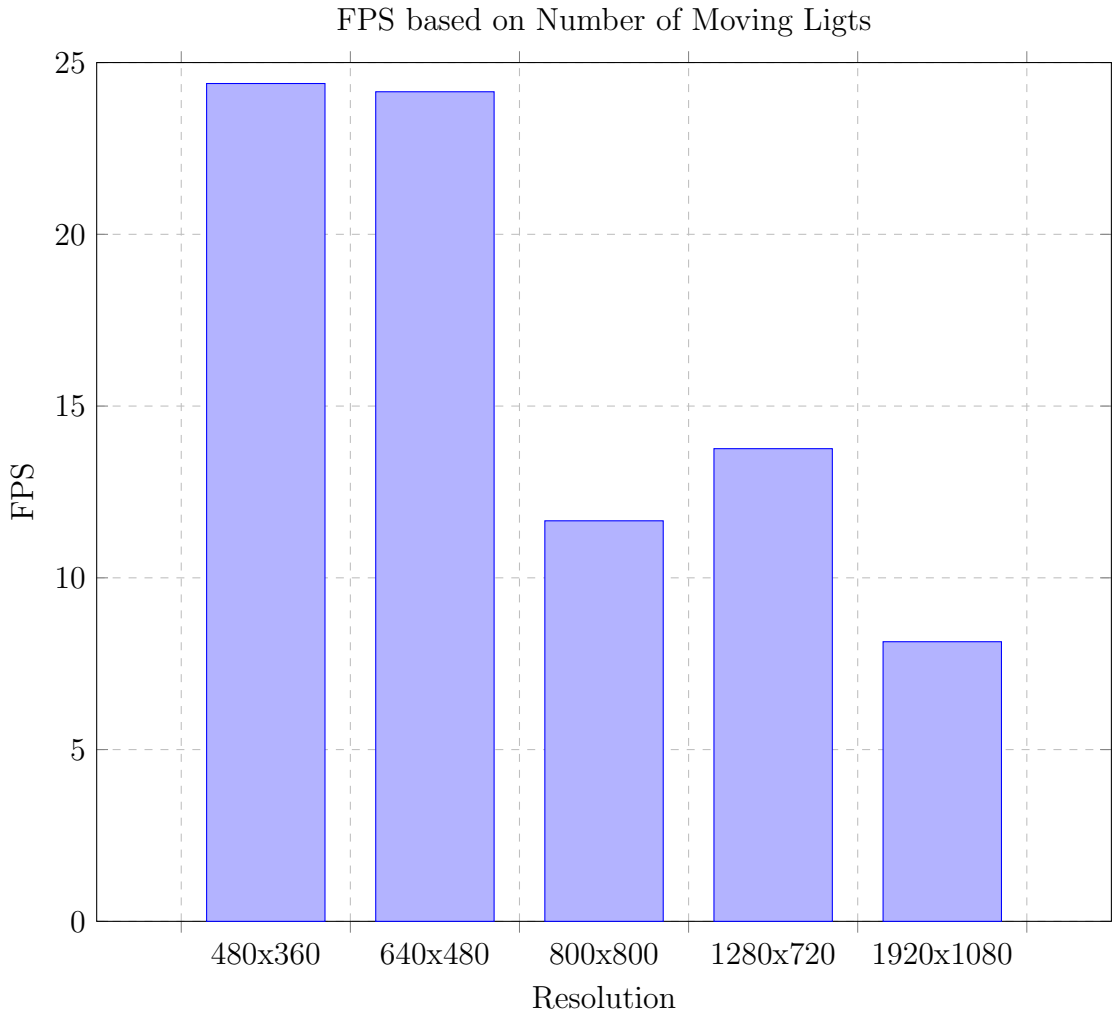


Table 7.4 shows us that that increasing the resolution decreases the performance. There is a weird and big dip at 800x800, which could be because of the difference in ratio between it and the other resolutions. It couldn't be because of the pixel count, as 800x800(640000) is smaller than 1280x720(921600). Further testing is required for different aspect ratios.

7.6. Adding Movement

For this test, after the light were render, I started controlling the light and rotate them around. This forces the program to render again both the RSM of the respective light, as well as the screen space ambient occlusion map for the scene fiewed from the camera's perspective.

The scene was similar to the previuos one: object rendered at 50%, 3 lights, all sportlights, resolution of 800x800 with an RSM resolution of 2048x2048.

No. Lights Moving	Frames/Second(FPS)	Avg Time per Frame(ms)
0	11.66	86.85
1	11.42	87.56
2	11.35	88.10
3	11.28	88.56

Table 7.6: Frames and frame duration for different number of moving lights

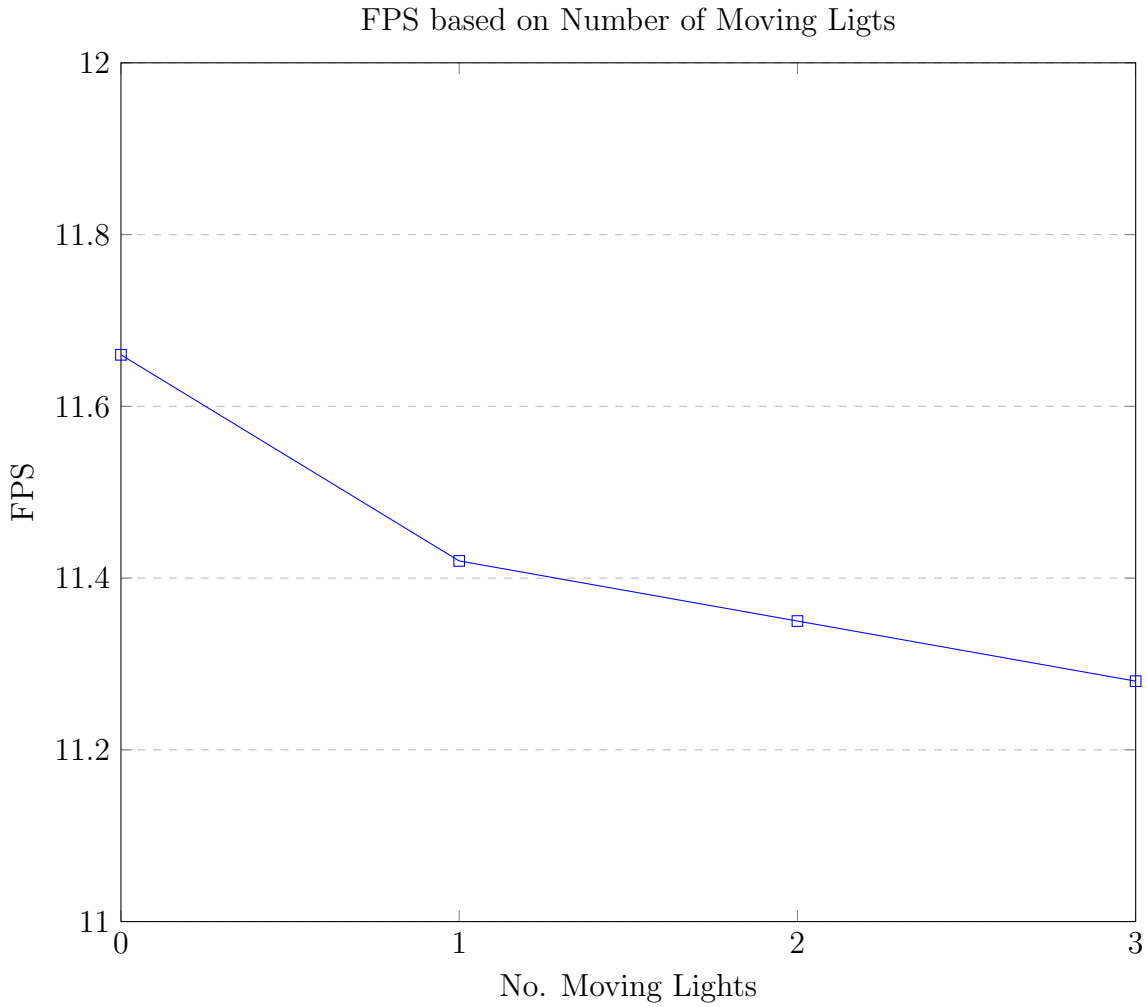


Table 7.5 shows us that, while there is a in FPS, it is not a big one. This makes sense, because the creation of an RSM does not take a lot of time. Furthermore, we can see that the first one has a higher drop, while the others are almost linear, as seen in the graph.This is because an RSM and the SSAO have to be recreated when we move one light, but when we add a second light to the movement, we practically and one more RSM that needs to be remade, thus a lower decrease in performance.

7.7. Result Analysis

From all the results from the tests we can know what affects the performance when using these algorithms. Those are the resolution of the screen, the number of lights and the complexity of the scene. With light movement and light type having a small impact. We also learned which of them have the biggest impact.

We also learned which one is the most performance costing of the three. Without any doubt, Indirect Diffuse Illumination takes the win with the greatest impact, almost cutting the FPS to a third of the maximum. This makes sense, as it needs the most recources to work. The other two do not need that much more to work.

But why do resolution of the screen, the number of lights and the complexity of the scene have the biggest impact? Well, all three of them have to do with the number of fragments needed to be computed. With an increase of resolution the objects becomes larger and formed of more fragments, thus having more fragments needed to be processed. Same thing for scene complexity, more objects, more fragments to process, more time to create a new frame. With more lights, each fragment has to do all algorithms for an additional light. For example, when going from one light to two, we almost double what is needed to be computed.

Chapter 8. User's Manual

In order to make the program work, the setting have to be noted in a configuration file. The fields are already written, only the values after '=' of each field have to be specified.Between "=" and the values, a space has to be left. There are comments above every field to remind the user what to write.

The fields are as follow:

- width: width of the screen, should be a natural number
- height: height of the screen, should be a natural number
- Indirect Specular Lighting, Ambient Occlusion and Inidrect Diffuse Illumination are used to select which algorithms are going to be rendered. The values have to either 1 for it to render, and 0 to not.
- noModels: the numbers of models to be rendered, can't be bigger than 5, the next 5 fields are repeated for each model:
  - modelPath: relative path to the model, has to be a .gltf file
  - Translation: the X,Y and Z values used for translation, should be left 0 if no translation needed, real numbers
  - Rotaion: the Yaw, Pith and Roll used for rotation, should be left 0 if no rotation needed, real numbers
  - Scale: the X,Y and Z values used for scaling, should be left 1 if no scaling needed, real numbers
  - percentage: the amount of the object to be rendered, has to be a real number from 0 to 1
- noLights: the number of lights the scene will use, can't be bigger than 10, the next 5 fields are repeated for each light:
  - type: what kind of light is used, 0 for a direct light, 1 for a spotlight
  - render: whether the light is on in the first frame, 1 for on, 0 for off.
  - colors: the red, green and blue values of the colors, should be a real number between 0 and 1.
  - position: the X, Y and Z coordinates of the light, it is recommended to use a smaller value for a direct light
  - direction: the X, Y and Z values of the direction of the light. Z should not be 0 for a spotlight.

After that, the project can be compiled and ran. During the simulation the user can input different thing using the keyboard and mouse. Those are:

Hold Left Click + Mouse Movement: Rotate the camera to look around
W: Move camera forward
S: Move camera backward
D: Move camera right
A: Move camera left
Spacebar: Move camera up

Left CTRL: Move camera down
Hold Left Shift: Move faster
Holding Number Keys: Select the respective light, 1 for the first, 2 for the second and so on until 9 for the ninth and 0 for the tenth
Arrow Keys: Rotates the selected lights, if those are spotlights
Q: Turns on and off the selected lights, turns on the ones that are off and the other way around

Furthermore, while the program is running the user can change what is shown. On default, the rendered scene is show, but the user can use O and P to rotate between the following:

- Regular Scene: the final scene render
- World Coordinates Map: the world coordinates part of the G-Buffer
- Normal Map: the normal part of the G-Buffer
- Flux Map: the flux part of the G-Buffer
- Depth Map: the depth map from the camera's perspective
- SSAO Map: the buffer created by the SSAO algorithm
- RSM World Coordinates Map: the world coordinates part of one of the RSMs
- RSM Normal Map: the normal part of the RSMs
- RSM Flux Map: the flux part of the RSMs
- RSM Depth Map: the depth map from one of the light's perspective

The user can further rotate between the RSMs of each light using the K and L keys. The program will jump to the first rendered RSM if there is none in between. For example, if only the first and third lights are rendered, the RSMs will jump between those two.

Chapter 9. Conclusion

The purpose of this project, as presented in chapter 2, was to test some algorithms used to simulate light in a scene rendered in real time. The chosen algorithms were Indirect Specular Lighting, Ambient Occlusion and Indirect Diffuse Illumination. The reason those three were chose is because all of them work with reflective shadow maps. In other words, they all work with a scene seen from multiple points of view. In chapter 3, which was bibliographic study I briefly presented those algorithm, their history and how they evolved in time. In chapter 4 I've talked in depth about the history, as well as how each algorithm does. I've also presented how the graphical pipeline works and how to use multiple lights. In chapter 5, I have shown the computer and software used.

As presented in chapter 6, in this project I was able to:

1. Learn and understand a few algorithms, as well as their history. They were all created with two things as the target. One is realism, as it is important to create a scene as close to reality for the purpose of simulations, as well as for better looking visuals in both pre-rendered and real time rendered scenes. The other one is the increase in performance and accuracy and decrease in rendering time and resources used. One such example, presented in chapter 3, is screen space reflection. The first proposal by Turner Whitted [5] used the geometry of the scene directly, which is costly and took a lot of time. Later Tiago Sousa and Co. [12] took that idea and used deferred rendering to create buffers that could be used instead of geometry. This is one of the things research should lead to, not only the creation of new ideas, but the improvement of older ones.
2. Create a testing environment where the user can customize the setting and change some parameters of the scene. The most important of them is the ability to choose which of the three combined algorithms are used. The user can also specify the models and lights used in the scene. The user can then change the rotation of the lights, turn the light on and off, move the camera and choose which buffer to see. They are also given the average FPS of the scene. All of those options makes the application a useful tool to test them and see their effect to the performance.
3. Implement and test the algorithms and see what affects their performance. I tested how they perform under different scenarios: the number of lights, the complexity of the object, if the lights move and the resolution of the scene. I also tested how each of the three algorithms affect the performance by themselves or combined with the others.

The results in chapter 7 show us how the algorithms perform. By themselves Ambient Occlusion and Indirect Specular Lightning do not affect the performance by a lot, so they could be used in real time rendering without a lot of problems. Indirect Diffuse Illumination is an entire different story, the way I implemented it results in a huge drop in FPS of almost 66% from the initial scene. Thus that version should not be used in real time rendering, except for small scenes.

As for how the scene affect the performance, I have presented what factors affect how well they perform. As presented in chapter 7, the resolution, object complexity and number of lights affect it the most. When it comes to resolution and object complexity, what creates the decrease in FPS is the bigger amount of fragments needed to be rendered, as each has to go through the same series of computations. The number of lights decrease the performance because it increases the number of computation each fragment has to go trough. This is because each fragment has to have the amount of light it gets from each light computed separately, then added together. The movement of light does not affect the performance that much, as it only has to re-render the reflective shadow map for said light and the ambient occlusion map, which is not that costly to do.

Further improvements can be done, for example using the G-Buffer and calculate the light for each pixel, instead of doing my way, of rendering the entire object again. Another improvement, that was presented in Cristian Lambru's PHD thesis[2], is to use a smaller sampler for indirect diffuse illumination. The author proposed the use of a smaller sample size, of just 10, and rotate the point around the center of the sample space and, combining it with the previous frame, should lead to similar results. The problem this creates is a higher number of artifact and pixels that don't look just right, but saves on resources.

The application in itself could be improved. One of them would be the ability to add more object and lights, as right now there can only be 5 and 10, respectively. Another improvement is the ability to change the setting in real time, not only before starting the application, as well as being able to make the object move. The final one would be the addition of other algorithms as we as the addition of point lights.

In conclusion, this project helped me understand the algorithms and their usage better. This project also helped me understand that I can do something interesting.

Bibliography

[1] G. Stark, "Light," Available at https://www.britannica.com/science/light (2023/02/07).

[2] C. Lambru, "Iluminare globală în timp real în jocuri video," Ph.D. dissertation, Bucharest, Romania, 2022.

[3] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: A vlsi system for high performance graphics," in *Proceeding of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'88*. New York, NY, USA: Associaton for Computing Machinery, 1988, pp. 21–30.

[4] T. Saito and T. Takahashi, "Comprehensible rendering of 3-d shapes," in *Proceeding of the 17th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'90*. New York, NY, USA: Associaton for Computing Machinery, 1990, p. 197–206.

[5] T. Whitted, "An improved illumination model for shared display," *Communication of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[6] C. Dachsbacher and M. Stamminger, "Reflective shadow maps," in *Proceeding of the 2005 Symposium on Interactive 3D Graphics and Games, I3D'05*. New York, NY, USA: Associaton for Computing Machinery, 2005, pp. 203–231.

[7] G. Miller, "Efficient algorithms for local and global accessibility shading," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques,SIGGRAPH '94*. New York, NY, USA: Associaton for Computing Machinery, 1994, p. 319–326.

[8] S. Zhukov, A. Iones, and G. Kronin, "An ambient light illumination model," *Eurographics Workshop on Rendering Techniques*, pp. 45–55, Springer,1998.

[9] M. Pharr and S. Green, "Ambient occlusion," *GPU Gems*, pp. 279–292, Addison-Wesley,2004.

[10] J. Kontkanen and S. Laine, "Ambient occlusion fields," in *Proceeding of the 2005 Symposium on Interactive 3D Graphics and Games, I3D'05*. New York, NY, USA: Associaton for Computing Machinery, 2005, pp. 41–48.

[11] M. Mittring, "Finding next gen: Cryengine 2." *ACM SIGGRAPH 2007 Courses, SIGGRAPH '07*, pp. 97–121, 2007.

[12] T. Sousa, N. Kasyan, and N. Schulz, "Secrets of cryengine 3 graphics technology," *ACM SIGGRAPH 2011 Courses, Advances in Real-Time Rendering in 3D Graphics and Games*, 2011.

[13] K. Vardis, G. Papaioannou, and A. Gaitzes, "Multi-view ambient occlusion with importance sampling," in *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D' 13*. New York, NY, USA: Associaton for Computing Machinery, 2013, pp. 111–118.

[14] C. Lambru, A. Morar, F. Moldoveanu, V. Asavei, and S. Ivascu, "Hybrid global illumination: A novel approach combining screen and light space information," *University Politehnica of Bucharest Scientific Bulletin Series C-Electrical Engineeringand Computer Science*, vol. 83, no. 2, pp. 3–20, 2021.

[15] G. Pipeline, Available at https://graphicscompendium.com/intro/01-graphics-pipeline (2023/05/07).

[16] J. D. Vries, "Learn opengl," Available at https://learnopengl.com/ (2023/02/07).

[17] OpenGL, Available at https://www.opengl.org (2023/02/07).

[18] GLFW, Available at https://www.glfw.org/ (2023/05/07).

[19] V. Gordan, "opengl-tutorials," Available at https://github.com/VictorGordan/opengl-tutorials (2023/02/07).

[20] B. Rogez, "Massive scale model of ancient rome," Available at https://sketchfab.com/3d-models/massive-scale-model-of-ancient-rome-2f9cb1fbc2eb470686eb8ba596b059cb (2023/02/07).

[21] L. Bavoil and M. Sainz, "Multi-layer dual-resolution screen-space ambient occlusion," in *SIGGRAPH 2009: Talks*. New York, NY, USA: Associaton for Computing Machinery, 2009.

[22] D. Bischoff, T. Schwandt, and W. Broll., "Efficient algorithms for local and global accessibility shading," in *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 1: GRAPP*. SciTePress„ 2017, p. 116–126.

[23] C. C. . the complete solution for next generation game development by crytek, Available at https://www.cryengine.com/ (2023/05/07).

[24] F. C. Crow, "Shadow algorithms for computer graphics," *SIGGRAPH Computer Graphics*, vol. 11, no. 2, pp. 242–248, 1977.

[25] C. Damez, K. Dmitriev, and K. Myszkowski, "State of the art in global illumination for interactive applications and high-quality animations," *Computer Graphics Forum*, vol. 22, no. 1, pp. 55–77, 2003.

[26] H. Kolivand and M. S. Sunar, "Survey of shadow volume algorithms in computer graphics," *IETE Technical Review*, vol. 30, no. 1, pp. 38–46, 2013.

[27] P. Lensing and W. Broll, "Efficient shading of indirect illumination applying reflective shadow maps," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '13*. New York, NY, USA: Associaton for Computing Machinery, 2013, p. 95–102.