MASTER'S THESIS | LUND UNIVERSITY 2016

Multi-layered G-Buffers for Real-Time Reflections

Mattias Simonsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-33

Multi-layered G-Buffers for Real-Time Reflections

Mattias Simonsson
dat11msi@student.lu.se

August 24, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Michael Doggett, Michael.Doggett@cs.lth.se
Examiner: Flavius Gruian, Flavius.Gruian@cs.lth.se

Abstract

This thesis evaluates the use of screen space reflections combined with a deep G-buffer in computer graphics applications. Both visual inspection of rendered images and computational methods are used to evaluate the quality and performance of the reflections.

Rendering reflections using screen space methods is generally faster than rendering them with other methods. The main drawback is that only parts of the scene visible to the viewer can be reflected. A deep G-buffer negates a part of this drawback by allowing obscured parts of the scene to be used for computing the reflections.

We implement screen space reflections using ray tracing on the GPU and construct a deep G-buffer using depth peeling. Additionally, we approximate reflections on rough surfaces using Monte Carlo methods.

We show that a deep G-buffer does improve the quality of screen space reflections, but greatly increases the time required to render the scene.

Keywords: Ray tracing, reflection, importance sampling, deep G-buffer, screen space

Acknowledgements

I would like to thank my supervisor Ass. Prof. Michael Doggett for his excellent advice and guidance.

Contents

Chapter 1
Introduction

In real-time computer graphics, developers are always trying to render images that are as realistic as possible in a limited amount of time. In an interactive real-time graphics application, each image must be rendered in a few hundredths of a second to feel responsive. With hardware steadily growing more powerful, more and more rendering can be done within this time. One way to add realism to images is to render reflections, which is what this thesis focuses on.

1.1 Rendering

Real-time rendering is typically done using a method called rasterization. A simplified way of describing this is that scene objects are composed of triangles and each triangle is described by three vertices. Each vertex contains a position coordinate and other optional information (color, normal vector, texture coordinate). When the image is rendered, the graphics card determines which pixels of the image each triangle covers, and the color of a pixel is determined by the closest triangle that covers that pixel.

1.1.1 Deferred rendering

Deferred rendering [10] is a method of doing rasterization rendering that is focused on rendering lighting at high performance, especially lighting with multiple light sources. In deferred rendering, the rendering is split up into multiple passes. Each pass typically receives some data as input and produces a result. The result is stored in one or more textures, depending on how much data the rendering pass outputs.

The number of rendering passes and the functionality of each pass is very application specific. Some examples of common rendering passes are:

- **Geometry pass**. Receives scene geometry data as input and outputs the properties of the geometry at each pixel. The output of this pass is often referred to as the

geometry buffer (G-buffer). Some information that is usually stored in the G-buffer are normal vectors, color and depth. More information can be added to the G-buffer depending on the requirements of the other rendering passes.

- **Lighting pass**. Uses the G-buffer and the properties of a light source to compute how that light source affects the scene. This pass typically runs once for each light source in the scene.
- **Resolve pass**. Combines the output of previous rendering passes and produces a single texture that is displayed on the screen.

1.1.2 OpenGL and GPUs

Graphics cards are present in virtually all personal computers, either integrated in the central processing unit (CPU) or as a standalone hardware component. Graphics cards, or graphics processing units (GPUs), are processors designed specifically to render images quickly. They focus heavily on parallelization which is important because when rendering, each pixel can often compute its color without interacting with other pixels. This allows the GPU to process many pixels at once. While a CPU might have a few cores that can do computations simultaneously, a GPU can have hundreds or thousands. In return, an individual GPU core is not nearly as complex as a CPU core and some operations, like branches, are much slower on a GPU than on a CPU. Because of these differences it is a good idea to do as much rendering work as possible on the GPU and avoid doing rendering computations on the CPU.

To communicate and issue commands to the GPU we will use OpenGL [2] in the implementation of this thesis. OpenGL acts as an interface between the program and the GPU. All commands or operations we can do on the GPU are defined by OpenGL. This allows our program to run on any graphics card as long as it supports OpenGL.

1.1.3 Coordinate spaces

When we create and render a scene, several different coordinate systems are used. When we create a scene, objects are placed in a virtual 3D world and each object has a 3D coordinate. However, when we render the scene the coordinate of each object needs to be transformed from a 3D virtual world coordinate into a 2D screen coordinate. The three main coordinate systems we use when rendering are:

- **World space**. This is the coordinate system that defines the 3D virtual world we place scene objects in. The origin is an arbitrary 3D point and all objects are positioned relative to the origin.
- **View space**. This is a 3D coordinate system where the origin is located at the camera. It is also often referred to as camera space or eye space. In this coordinate space, all objects are positioned relative to the camera.
- **Screen space**. This is a 2D coordinate system where the position of each object directly corresponds to a position on the screen. This is also referred to as clip space or pixel space.

To transform positions between different coordinate systems, transformation matrices are used. The matrix that transforms positions from world space to view space is called the view matrix. The matrix that transforms a position from view space to screen space is called the projection matrix. It is also possible to invert the transformation matrices to reverse the transformations. For example, an inverted projection matrix will transform positions from screen space to view space.

To be able to correctly perform projective transformations, we convert coordinates to homogeneous coordinates before applying transformations. This is done by adding another dimension to the coordinates. The world space position (x, y, z) becomes (x, y, z, w), with $w = 1$. After transforming this position into view space using the view matrix, we are left with (x', y', z', w'). To convert this homogeneous coordinate back to a 3D cartesian coordinate, we divide by w' and remove the fourth dimension, leaving $(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'})$.

1.1.4 Shaders

Shaders are special programs that are executed on a GPU instead of a CPU. This allows programmers to customize many parts of the rendering process. There are different types of shaders that are executed at different stages of the rasterization process. The two most commonly used shader types are the vertex shader and the fragment shader.

The vertex shader receives individual vertices as its input. The primary task of the vertex shader is to transform the vertex position from world space to screen space. It can also be used to compute per-vertex lighting or other per-vertex effects. The vertex shader is executed once for each vertex in the scene.

The fragment shader is executed once for each pixel in the image being rendered. It is sometimes referred to as the pixel shader. The fragment shader can make use of results produced by the vertex shader to compute its own result. The output of the fragment shader is a depth value and zero or more color values. The number of colors produced by the fragment shader is decided by the programmer. Fragment shaders can be used to compute per-pixel lighting, texturing, and many other per-pixel effects.

Another shader type that is seeing more and more use is the compute shader. While vertex shaders and fragment shaders have existed for a long time, compute shaders are a relatively new type of shaders. Compute shaders are extremely versatile and commonly used to perform arbitrary computations on a GPU. All inputs and outputs of a compute shader are defined by the programmers, they do not have any pre-defined inputs or outputs. The number of times a compute shader is executed is also decided by the programmers.

OpenGL shaders are written in a special programming language called GLSL (OpenGL Shading Language) [4]. GLSL is in many ways similar to C. It has support for vectors and matrices and several built-in math operations that are commonly used for rendering.

1.2 Introduction to reflections

Reflections occur when light hits a surface and bounces in a new direction instead of being absorbed. This is the reason we can see anything at all. Light bounces on a surface and enters our eyes, allowing us to create images of the world. In this thesis however, we will be ignoring reflections caused by the light of a light source directly hitting an object. We will

instead focus on reflections that occur when the reflected light of one object is reflected again by another object. These kinds of reflections are not nearly as common as reflections from direct light. You can only see them on certain surfaces such as mirrors, windows, water, shiny metals, polished floors etc. because of how relatively weak they are.

While there are several well known and well researched ways of approximating direct reflections, these methods do not work well for approximating secondary reflections. In direct reflections, you can compute a good approximation using only the properties of the light source and the properties of the surface being hit by the light. But the color of secondary reflections depends on potentially all other objects in the scene. Light sources also have well defined behavior and are often represented as zero-volume points while ordinary scene objects have arbitrary shapes and colors. We cannot use the same method to approximate secondary reflections as we do with direct reflections, since it is too expensive.

1.3 Motivation

Real-time graphics applications such as games strive to render an image that is as realistic as possible in a limited amount of time. As hardware grows more and more powerful, more and more rendering can be done in the same amount of time. In a game running at 60 frames per second (FPS), the application has about 16.67 milliseconds of time to render an image as realistic as possible, and reflections are one way to enhance the realism of the image.

1.4 Methods of rendering reflections

There are a few different ways of rendering reflections in scenes. In general, accuracy is inversely proportional to speed. The more time you spend on rendering the higher the quality of the reflections will be.

1.4.1 Ray tracing

Ray tracing is the most accurate and the most expensive way to produce reflections. With this method, a number of individual photon paths are simulated for each pixel in the rendered image. Interactions between each photon and the scene geometry are simulated, which allows for physically correct rendering. This produces very accurate images, but it is more suitable for rendering non-interactive scenes such as videos rather than interactive real-time applications.

1.4.2 Additional cameras

A second way to compute reflections is to render the scene multiple times from different points of view. As an example, to render the reflection shown in a mirror on a wall, the scene could be rendered from the point of view of the mirror. This rendered image would then be used as the reflection in the mirror.

Another way to use this method would be to strategically place cameras at different points in the scene. These cameras will render their surroundings and store them. When reflections are to be rendered, the information stored in the closest camera is used to render them. This method is available in several 3D game engines with different names like Reflection Probes in Unity [5] or Reflection Capture Actors in Unreal Engine 4 [6].

1.4.3 Screen space ray tracing

A third way to render reflections is to do ray tracing in screen space. In the deferred rendering process we insert an additional rendering pass after geometry and lighting is finished. In this pass we use the G-buffer information to compute the reflection color on each pixel by finding the origin of the light that was reflected by the surface at that pixel. Once we find the origin of the light, we retrieve the color at the light origin and use that color as the reflection color.

Because of the limited amount of information used (one depth value per pixel) the rays can be traced fairly quickly. It also works very well on a GPU because each pixel can do its ray tracing independently of other pixels. There are of course some limitations. Information about the scene that is not present in the G-buffer cannot be used in the ray tracing and can therefore not be reflected. So objects that are obscured by other objects or parts of the scene that are outside the screen bounds cannot be reflected at all.

This method is fast enough to be usable in real-time graphics applications and this is the method this thesis will focus on.

1.5 Deep G-buffers

One of the major limitations of doing reflections with screen space ray tracing is that objects that are obscured cannot be reflected. Examples of this limitation can be seen in Figures 1.1 and 1.2. A way to get around this limitation is to use multiple layers in the G-buffer. The scene geometry is rendered multiple times with each rendered image containing information from different depths of the scene. The information from all these layers is then combined when doing the ray tracing which allows obscured objects to be reflected, provided that these objects are visible in one of the layers. The downside of this is that it is expensive to render the scene multiple times. We are going to evaluate the performance impact of using multi-layered G-buffers relative to the increase in image quality in this thesis.

1.5.1 Depth peeling

The method that will be used for creating multi-layered G-buffers is depth peeling [7]. First the scene is rendered normally and stored as the top layer of our G-buffer. Then the scene is rendered again, once for each layer, and on each pixel, a depth comparison is performed with the same pixel on the previous layer. If the depth value of a pixel at layer $i + 1$ is less than or equal to the depth value of the same pixel at layer i, this pixel is discarded. The end result is that each layer only contains information that was obscured in the previous layer. A demonstration of this effect can be seen in Figure 1.3.

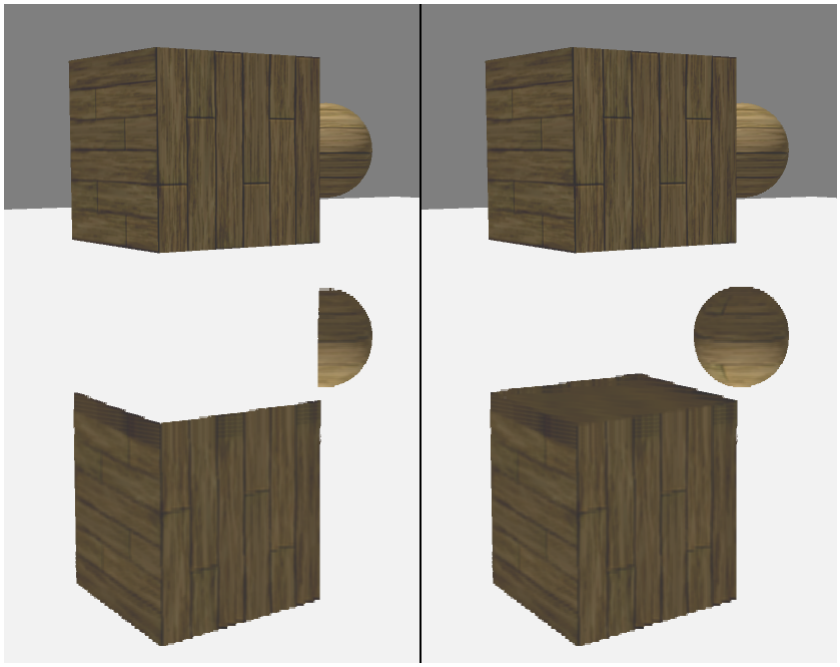

Figure 1.1: Scene with a cube and a sphere floating over a reflective surface. The left image is rendered with a single G-buffer layer and obscured parts of the image are not reflected. The right image is rendered with multiple layers.

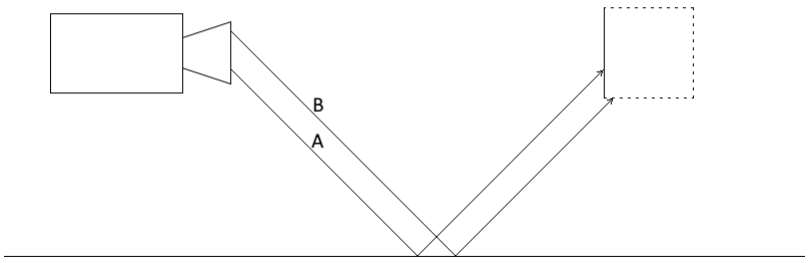

Figure 1.2: Only one side of the square is visible to the camera. Because of this, ray A will properly intersect with the square but ray B will not. If we used a multi-layered G-buffer, the sides of the square that are not visible in the camera would be still be stored in the G-buffer. This would enable both rays to properly intersect with the square.

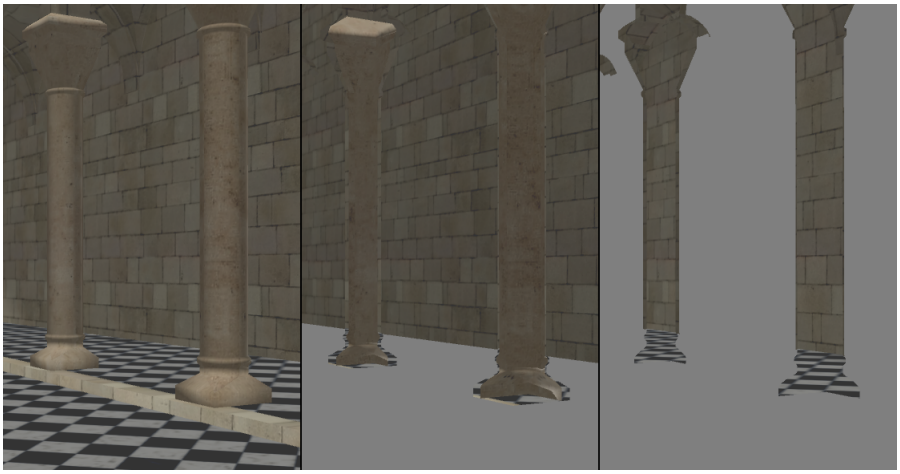

Figure 1.3: Demonstration of depth peeling. Scene rendered with a G-buffer with three layers. First layer is shown to the left, second layer in the middle and third layer to the right. Each layer contains information that was obscured in the previous layer.

1.6 Related work

We base our screen space ray tracing algorithm on previous work by Morgan McGuire and Michael Mara presented in the paper "Efficient GPU screen-space ray tracing" [14]. Their paper presents a fast screen space ray tracing algorithm. We modified this algorithm to fit into our program and extended it to support arbitrary G-buffer depth.

We also use ideas based on a presentation by Tomasz Stachowiak titled "Stochastic Screen-Space Reflections" [17]. Many ideas used in our implementation is based on their presentation. However, their presentation does not explain why or how the ideas were used and implemented, which is something we do in this thesis. Their version also uses a single-layered G-buffer while we use a multi-layered G-buffer in this thesis.

Previous work on screen space reflections was presented in a talk by Peter Sikachev and Nicolas Longchamps titled "Reflection System in Thief" [15]. Their solution to approximate rough surfaces is different from the one implemented in this thesis. Their implementation uses the distance traveled by the ray to approximate contact hardening, but it still assumes that surfaces are perfectly flat. Our method is more physically accurate but also more expensive because we need multiple rays per pixel.

Related work in screen space algorithms using deep G-buffers is presented by Michael Mara, Morgan McGuire, Derek Nowrouzezahrai and David Luebke in a paper titled "Fast global illumination approximations on deep g-buffers" [13]. In their paper a different method is used to construct the deep G-buffer. Instead of depth peeling, which requires multiple render passes, their method constructs a deep G-buffer in a single rendering pass. Their method uses a geometry shader to render each triangle into multiple texture layers in one rendering pass. Their method also predicts the depths of each layer using the depths from the previous frame. With depth peeling, the depth value of a pixel in layer i is not known until layer i has been rendered. But with their method, the depths of a layer is predicted which allows layers i and $i + 1$ to be rendered in the same rendering pass.

1.7 Contributions

In this thesis we implement real-time screen space reflections with a deep G-buffer and show that it is possible to obtain an acceptable reflection quality while maintaining a decent frame rate. We provide implementation details and motivation for a method to approximate reflections on rough surfaces. We also provide an example of how a deferred rendering pipeline can be structured when implementing screen space reflections.

We present several ray tracing parameters that can be modified to trade quality for performance, and we provide data on how these parameters affect the reflections. While the changes in quality and performance depend on the scene rendered, these measurements provide an idea of what performance to expect for different parameter values.

Chapter 2
Implementation

We implement reflections using screen space ray tracing. Different techniques are added to improve the quality and performance of the reflections.

The implementation uses deferred rendering to render the scene. Rendering is split up into several passes: geometry, ray tracing, ray resolving, and temporal anti-aliasing. An overview of the deferred rendering pipeline can be seen in Figure 2.1.

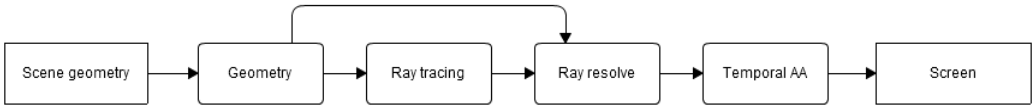

Figure 2.1: Overview of rendering passes in our implementation. Each rounded box represents a rendering pass. The arrows show how the results of each pass are used by other passes.

2.1 Tools

The Bonobo Framework from the course EDAN35 - High Performance Computer Graphics at LTH was used as a basis for the implementation of reflections. This framework is a wrapper around OpenGL and contains very useful functionality such as asset loading, window creation, input handling and debugging tools. OpenGL version 4.3 was used so that we could make use of compute shaders.

2.2 Geometry pass

The geometry is rendered by iterating through all objects in the scene and rendering them one at a time. Each pixel in the G-buffer stores the following information:

- Diffuse color (4 bytes)
- World-space normal vector (3 bytes)
- Reflectivity (1 byte)
- Depth (4 bytes)

The G-buffer is made up of three separate texture arrays: one to store the color of the object, one to store normal vector and reflectivity, and one to store depth. The number of layers in each texture array is equal to the depth of the G-buffer, since each texture array layer stores one G-buffer layer.

Depth peeling is performed in the pixel shader of the geometry pass. The entire scene geometry is rendered once for each layer and each layer makes use of the depth buffer of the previous layer. For each pixel, if $depth_i \leq depth_{i-1}$ where i is the current layer, that pixel is discarded.

2.3 Ray tracing pass

The ray tracing pass uses the G-buffer to trace a number of rays for each pixel. The results of these ray traces are stored in texture arrays and used in the next pass.

2.3.1 Ray tracing algorithm

Implementation of the screen space ray tracing algorithm was done based on previous work by Morgan McGuire and Michael Mara [14]. The algorithm simulates a ray moving in view space and screen space space simultaneously. At each step, the depth at the current pixel is read from the depth texture and a hit test is performed using the position of the ray and the depth of the geometry at the current pixel. If the hit test is successful, the ray is considered to have intersected with the geometry and the algorithm terminates.

The algorithm also uses a thickness value. The thickness represents how far the geometry at each pixel extends away from the camera. This is used as an interval during the hit test. If the difference in depth between the ray and the geometry at a pixel is smaller than the thickness of the geometry, the hit test succeeds.

A low thickness value may cause the ray to pass through objects, especially objects that are far away from the camera. But a high thickness value will produce artifacts. A demonstration of this can bee seen in Figure 2.2.

Using a variable thickness value produces better results than using a constant thickness in most cases. The difference in world position between two adjacent pixels with large depth values is greater than the difference between two adjacent pixels with small depth values. Because of this, in our implementation we store the position of the ray at the previous step. In each step, the thickness is scaled by the difference between the current ray position and the previous ray position.

A stride parameter is also used in the algorithm. This determines how far the ray travels between each hit test, measured in pixels. A high stride length will reduce the amount of hit tests the algorithm performs. Increasing the stride length greatly improves performance but reduces the quality of the reflections.

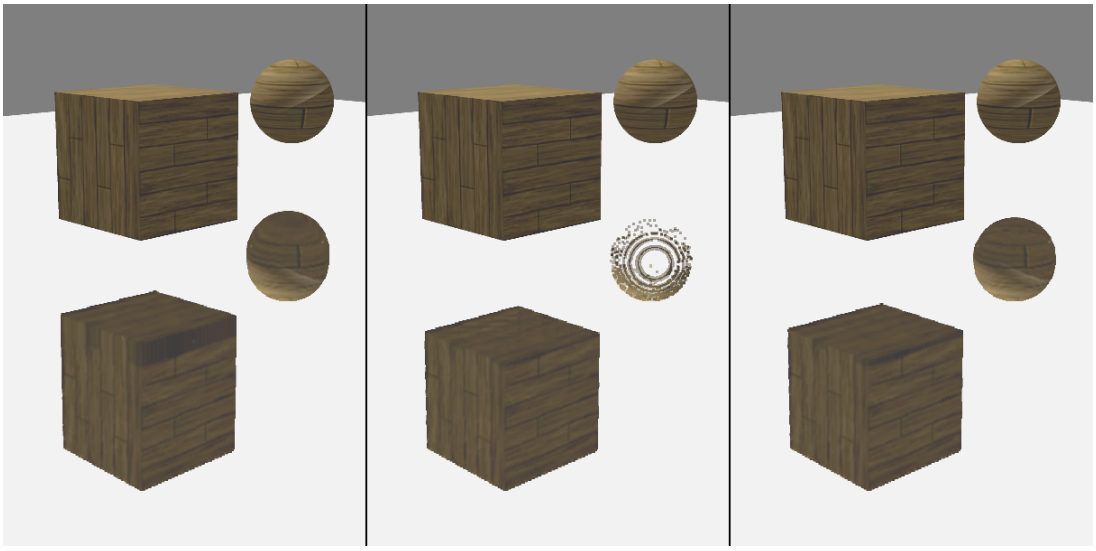

Figure 2.2: Demonstration of different thickness values. A high thickness value is seen to the left. Here, all rays intersect with the objects but the reflection of the cube close to the camera is stretched. A low thickness value is seen in the middle. Many rays completely miss the sphere because of the low thickness. To the right, a variable thickness is used. Objects that are far away from the camera are considered to be thicker while objects close to the camera are considered to be thinner.

On each step in the ray tracing algorithm, a hit test is performed on each layer in the depth texture array. This means that using multiple layers not only slows down the G-buffer creation, it also slows down the ray tracing.

2.3.2 Normal vectors on rough surfaces

In mirrors most light will be reflected as if the mirror was a completely flat plane. In this case, the direction of the ray can be calculated with

$$R = I - 2N(N \cdot I)$$

where I is the normalized vector between the ray origin and the camera, R is the direction of the ray and N is the surface normal where the ray originates. But most surfaces do not behave like this. Real surfaces are not perfectly flat and the surface normal is not constant across the surface. Instead it is constantly varying because of imperfections in the surface. To approximate reflections on rough surfaces we will need to take this into account. An illustration of this can be seen in Figure 2.3.

First we will need a way to correctly generate varying surface normals on rough surfaces so that we can use these normals to compute the reflection direction. We will use methods described by Bruce Walter, Stephen R Marschner, Hongsong Li and Kenneth E Torrance [18] to generate these. In this paper, different Bidirectional Scattering Distribution Functions (BSDF) are described. A BSDF is a function that models how light scatters when it hits a surface. A BSDF can be split up into two functions: one function describing

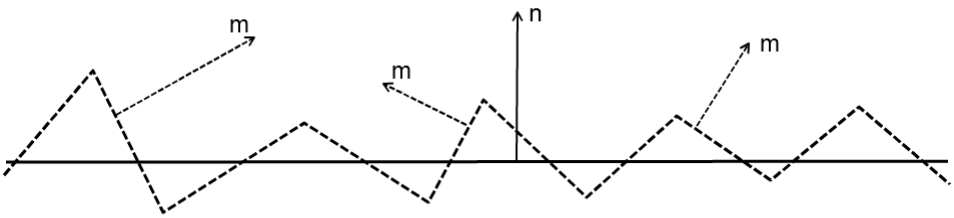

Figure 2.3: Illustration of microsurface and macrosurface. The solid line represents the macrosurface, which is an approximation of a real surface, where we assume that it is completely flat. The macrosurface normal n is constant across the entire surface. The dashed line represents the microsurface, which approximates a surface with imperfections due to roughness. The microsurface normals m vary across the surface.

how light is transmitted through a surface and the other function describing how light is reflected from a surface.

Because we only care about reflections in this thesis, we will ignore the transmission part of the BSDF. This leaves us with only the reflection part of the BSDF, which is normally referred to as a Bidirectional Reflectance Distribution Function (BRDF). There are several different BRDFs and they all attempt to model how light is reflected. The one we use in our implementation is called GGX.

To generate the normals, we sample a microfacet distribution function, which is included in the definition of each BRDF. A microfacet distribution function describes how the normal vector of a surface varies across the surface due to imperfections. Given two random numbers ζ_1 and ζ_2, both in the range $[0, 1)$, we can sample the GGX microfacet distribution with

$$\theta = arctan\left(\frac{\alpha\sqrt{\zeta_1}}{\sqrt{1-\zeta_1}}\right)$$

$$\phi = 2\pi\zeta_2$$

The α value is the roughness of the surface, with $\alpha = 0$ being a perfectly flat surface. As the roughness increases, the chance of sampling a large θ increases. We can convert these spherical coordinates to a vector (x, y, z) with

$$x = cos(\phi)sin(\theta)$$

$$y = sin(\phi)sin(\theta)$$

$$z = 1 - cos(\theta)$$

The equation for z may look unusual. The reason we subtract $cos(\theta)$ from 1 is that we want to have $(x, y, z) = (0, 0, 0)$ when $\theta = 0$. This vector (x, y, z) is then added to the original surface normal and used to compute the ray direction.

2.3.3 Monte Carlo integration

To compute the final reflection color of a rough surface we cannot simply sample a single ray direction and use that as the result. We can however look at the rendering equation described by James T Kajiya [12] to figure out how to proceed. The rendering equation states that the outgoing light of a point is the sum of the emitted light and the reflected light of that point. We are only concerned with reflected light so we can rewrite the equation as

$$L_o(\omega_o) = \int_\Omega f_r(\omega_i, \omega_o) L_i(\omega_i)(\omega_i \cdot n) d\omega_i$$

where

- ω_o is the direction of reflected light
- $L_o(\omega_o)$ is the color and intensity of reflected light in direction ω_o
- Ω is the unit hemisphere centered around the surface normal n
- f_r is a BRDF
- ω_i is the direction of incoming light
- $L_i(\omega_i)$ is the color and intensity of incoming light from direction ω_i
- n is the surface normal

To compute the final reflection color we need to integrate over all light incoming from all possible directions ω_i. A visualization of this can be seen in Figure 2.4. Since there are an infinite number of possible directions we will have to approximate this somehow. This will be done by performing Monte Carlo integration. In Monte Carlo integration, a random value is chosen and the integral is evaluated for this value. This process repeated and as you evaluate more values your result converges to the true result.

Our values in this case are ray directions, which means that the more rays we trace, the closer our reflection color will be to the true reflection color. However, if we choose uniform random directions and trace rays using those directions, we will need a large amount of rays to get a good result. Since it is expensive to perform ray tracing, we want to minimize the amount of rays we trace and still get a good result.

Instead of picking a large number of uniformly distributed ray directions, we will pick a smaller number of ray directions from another distribution. The idea behind doing this is that not every ray direction is equally important, and we can make our Monte Carlo integration converge faster by sampling important ray directions, also known as importance sampling. So we will perform the Monte Carlo integration with values sampled from the GGX microfacet distribution function that we selected previously.

By sampling from the GGX distribution instead of a uniform distribution, we ensure that the directions we evaluate in our Monte Carlo integration are likely to be important directions, even though they are still random. The average difference between the microfacet normals we sample and the original surface normal will depend on the roughness of the surface which can be seen in Figures 2.5 and 2.6. This is exactly the behavior we

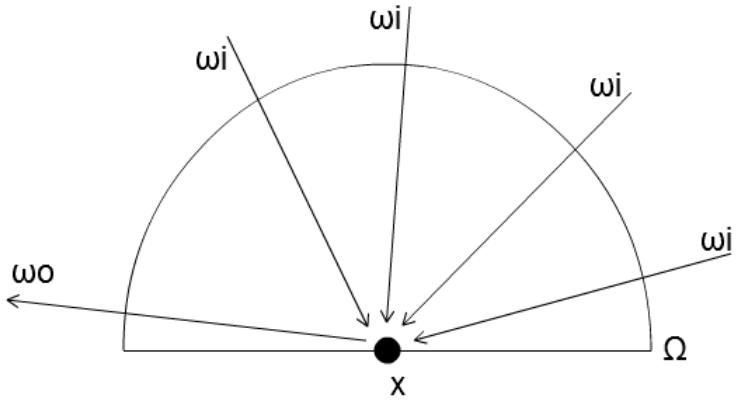

Figure 2.4: The light reflected from point x in direction ω_o is equal to the sum of all incoming light from all possible directions ω_i in the unit hemisphere Ω, weighted by a BRDF $f_r(\omega_i, \omega_o)$

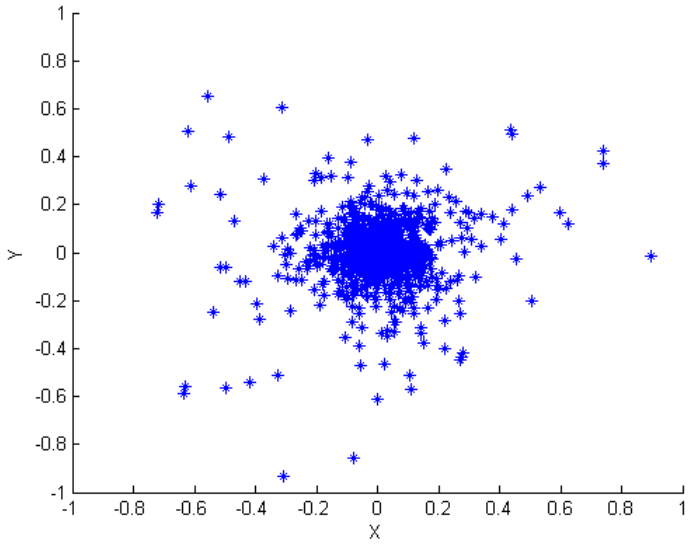

Figure 2.5: The x and y components of 1000 normal vectors sampled using the GGX microfacet distribution function with roughness $\alpha = 0.1$. Most values are located around $(0, 0)$ and most sampled normal vectors will point in almost the same direction as the original surface vector.

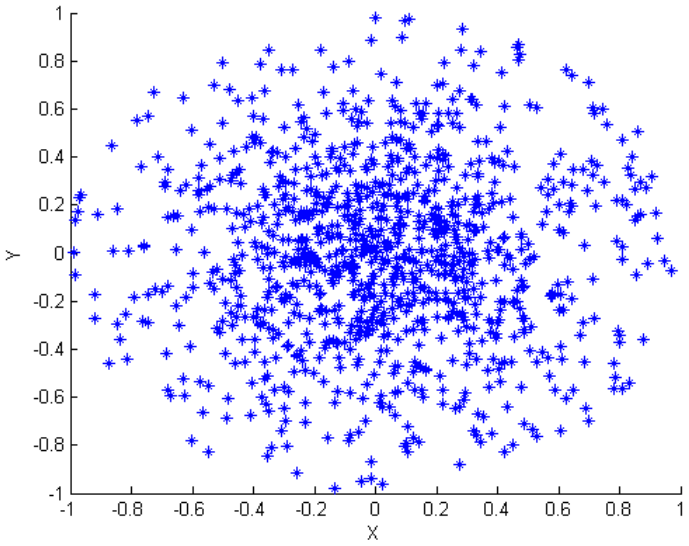

Figure 2.6: The x and y components of 1000 normal vectors sampled using the GGX microfacet distribution function with roughness $\alpha = 0.5$. While the values are more densely packed around $(0, 0)$, many of the normals will point in a completely different direction than the original surface normal.

desire. A low roughness should mean that most microfacet normals are similar to the original surface normal and a high roughness should mean that the sampled normals are more spread out.

The number of rays traced per pixel largely depends on the reflection quality desired for the application. More rays result in nicer but more expensive reflections. A comparison between different numbers of rays per pixel can be seen in Figures 2.7, 2.8 and 2.9.

Since we established that our sampled ray directions differ in importance and probability of being chosen, we need a way to weigh them differently when computing the result. To do this, we make use of the weighing formula described by Bruce Walter, Stephen R Marschner, Hongsong Li and Kenneth E Torrance [18]

$$weight(o) = \frac{|i \cdot m| G(i, o, m)}{|i \cdot n||m \cdot n|}$$

where

- o is the direction of the scattered light. In our case this is the normalized vector from the camera position to the ray origin.
- i is the direction of incoming light. In our case this is the negative direction of the ray. This is because even though we are tracing the ray from the origin and outwards, the light we are approximating is traveling from a scene object to the ray origin.
- m is the microsurface normal. This is the normal we sample from the GGX distribution.

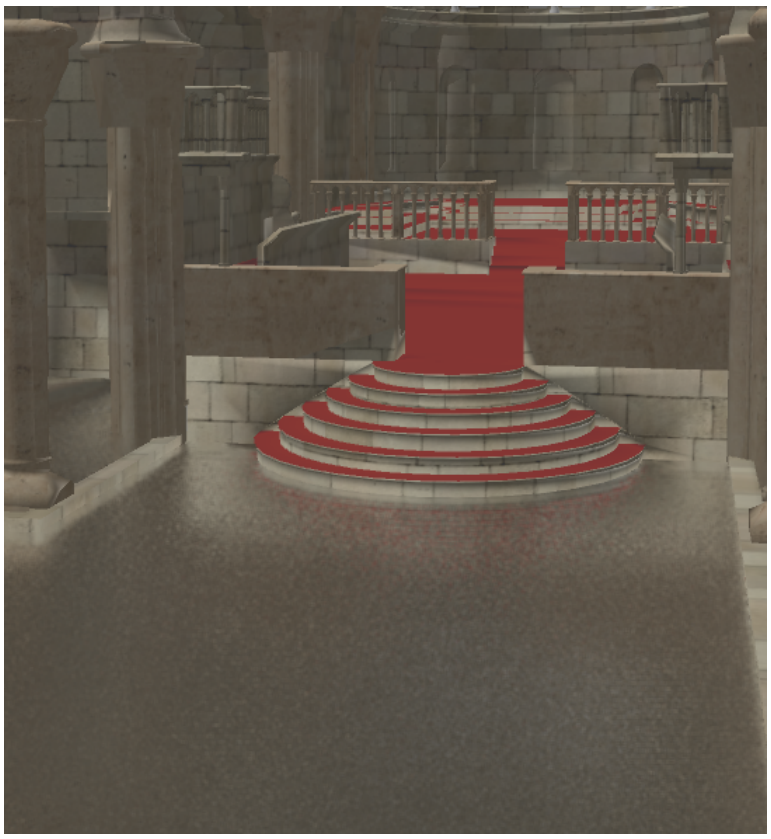

Figure 2.7: Sibenik[1] scene with reflections using two rays per pixel. The floor in the scene is reflective with a uniform roughness $\alpha = 0.08$. Each pixel effectively has 18 ray trace results to use because of neighboring rays being reused. The reflections are very noisy and rough, which is due to the small amount of samples for the Monte Carlo integration.

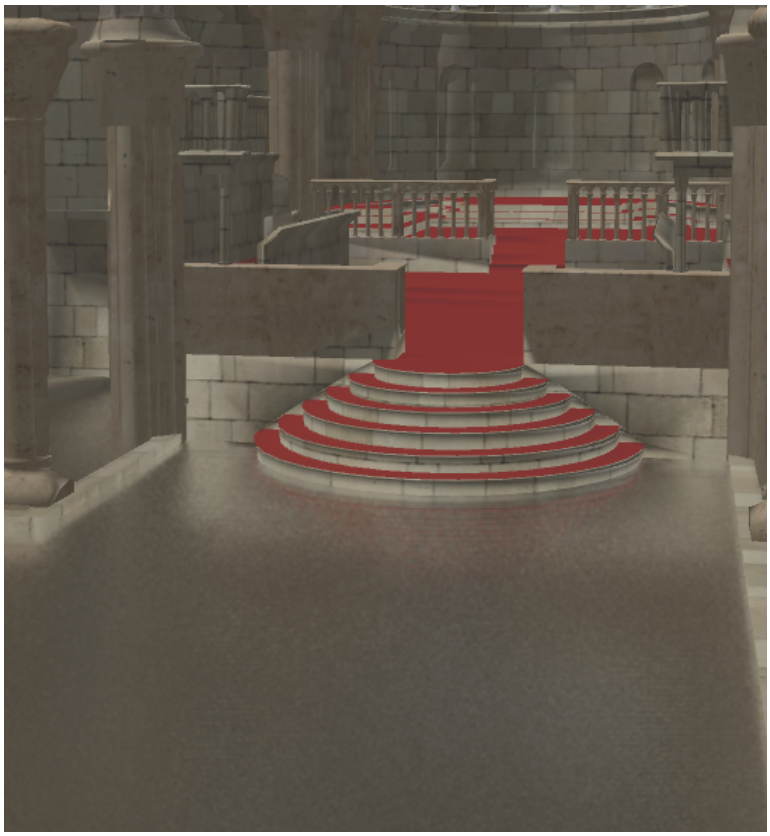

Figure 2.8: Sibenik scene with reflections using four rays per pixel. The floor in the scene is reflective with a uniform roughness $\alpha = 0.08$. Each pixel effectively has 36 ray trace results to use because of neighboring rays being reused. The reflections are still a bit noisy but smoother than in Figure 2.7.

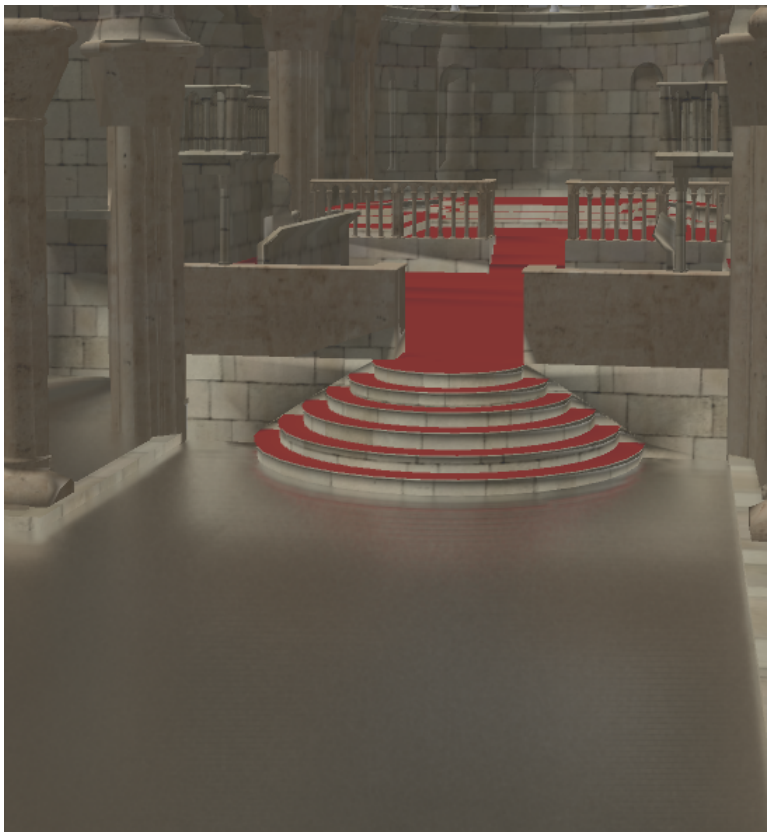

Figure 2.9: Sibenik scene with reflections using 16 rays per pixel. The floor in the scene is reflective with a uniform roughness $\alpha = 0.08$. Each pixel effectively has 144 ray trace results to use because of neighboring rays being reused. Reflections are much smoother than in Figures 2.7 and 2.8 but also much more expensive to compute. The reflections in this image were rendered in about 38 milliseconds, while the reflections in Figure 2.8 took 11 milliseconds to render and the reflections in Figure 2.7 took 6 milliseconds to render.

- n is the macrosurface normal, or the original surface normal we would use if the surface was completely flat.
- $G(i, o, m)$ is a Bidirectional shadowing-masking function. This function describes how much of the microsurface with normal m is visible in the directions i and o. By studying Figure 2.3 it is easy to imagine that parts of the microsurface may be blocked by other parts of it, and this is what $G(i, o, m)$ represents.

We use the G function as described by B Smith [16], which is $G(i, o, m) \approx G_1(i, m)G_1(o, m)$. The $G_1(v, m)$ function is defined as a part of a BRDF. For GGX it is

$$G_1(v, m) = \left(\frac{v \cdot m}{v \cdot n}\right) \frac{2}{1 + \sqrt{1 + \alpha^2 tan^2 \theta_v}}$$

where θ_v is the angle between v and n, and α is the roughness.

2.3.4 Ray tracing pass implementation

The ray tracing pass is implemented as a pixel shader. It receives the G-buffer data as input and uses this data to trace rays. For each ray, the following is computed:

- Screen space position of the pixel that was hit
- Which G-buffer layer the hit occurred in
- The weight of the result

We use 32 bits to store the screen space position. This gives us 65536 possible values for each coordinate in the position. Since modern screens usually have a few thousand pixels on each axis, there should be no problem with accuracy. We use 16 bits to store the weight and 16 bits to store the index of the layer that was hit. We also increment the layer index so that the first layer has index 1. Index 0 is used to indicate that the ray did not intersect with any geometry.

A texture in OpenGL can store 32 bits of data for each pixel, but we need 64 bits to store the result of a traced ray. Additionally, there can be multiple rays traced per pixel. Therefore, we use two texture arrays to store the result of this rendering pass. The number of layers in these texture arrays is equal to the number of rays we trace per pixel.

The steps to compute the results for a single ray are

1. Select two random numbers R_1 and R_2
2. Sample a normal vector n using the GGX distribution function, the random numbers R_1 and R_2, and the roughness a of the surface
3. Compute the ray direction; the `reflect` function in GLSL can be used for this
4. Find the intersection point using the screen space ray tracing algorithm
5. Compute the weight of the result
6. Store the result information in the texture arrays

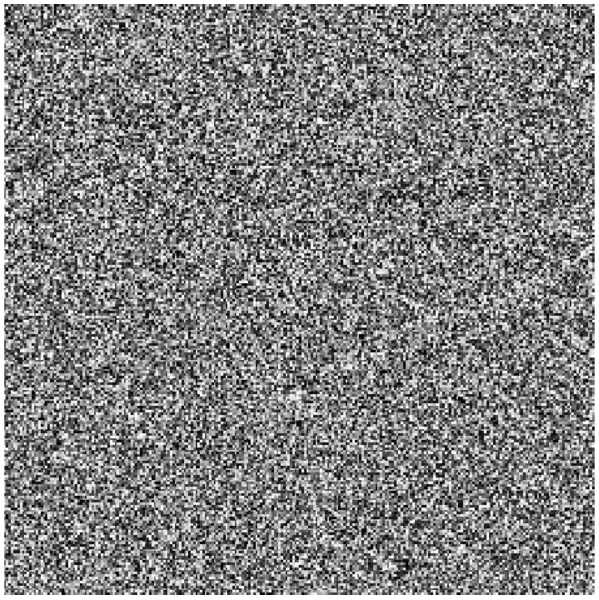

Figure 2.10: Output of hash function used to generate random-looking numbers in our implementation. The image looks like random noise with no visible pattern, which is what we want.

To generate random numbers we use a simple hash function

$$R(s) = fract(sin(s \cdot (12.9898, 78.233)) * 43758.5453)$$

where the seed s is a two dimensional vector. `fract` is a function built into GLSL that returns the fractional part of its argument. This ensures that the number we generate is in the range $[0, 1)$. The output of this function can be seen in Figure 2.10. The advantage of this method is that it is fast. Other methods, such as a lookup texture or more a complex noise function, would be slower. For the seed we use a combination of camera position, pixel position and surface normal at the pixel.

Because our number of rays is relatively low, the ray directions might become clustered if we are unlucky. This will appear as noise in the reflection. Instead of just using the random numbers $R1$ and $R2$ we generate, we add a quasi-random low discrepancy sequence to them. A quasi-random low discrepancy sequence is a sequence of values that appear to be random even though they are not actually random. These values also have the property of being evenly distributed in the interval $[0, 1]$. This increases the likelihood that the ray directions we generate are spread out, which reduces noise. The low discrepancy sequence we used in the implementation was the Hammersley sequence [9]. A comparison image showing reflections with and without the low discrepancy sequence can be seen in Figure 2.11

The depth values we read from the G-buffer depth texture have been transformed by a projection matrix. Because of this, they are not linear in view space, and retrieving the view space depth is not as easy as multiplying by a scale factor. Instead we will have to reverse the projective transformation. A projection matrix in OpenGL looks like this:

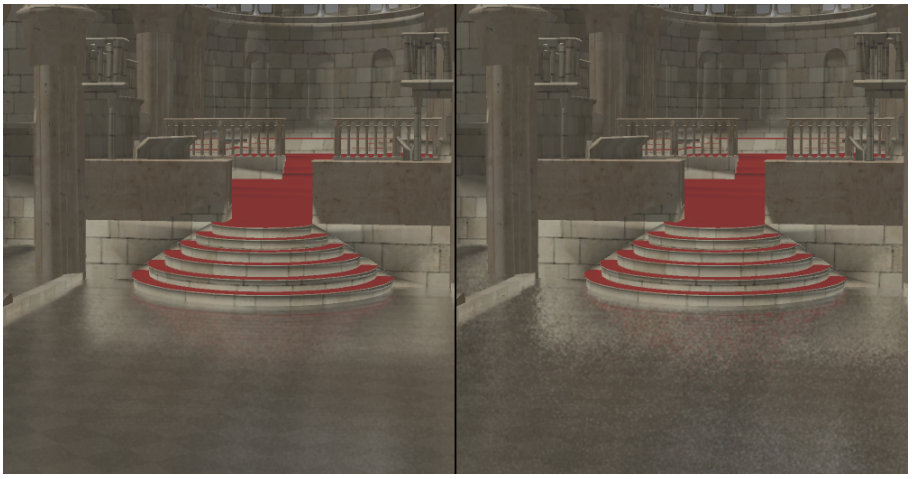

Figure 2.11: Comparison of rough reflections with and without adding the Hammersley sequence to the random numbers before sampling a ray direction. Left side uses the Hammersley sequence while the right side does not. The result is that the reflections on the right side contain much more noise.

$$P = \begin{pmatrix} \frac{g}{A} & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where $g = cotangent(\frac{fovY}{2})$, $fovY$ is the field of view of the camera in the y direction, A is equal to screen width divided by screen height, f is the distance between the camera and the far clip plane, and n is the distance between the camera and the near clip plane. These values are specified when the camera is created and passed to the pixel shader.

This matrix transforms a homogeneous 3D point $(x, y, z, 1)$ in view space to a point in screen space (x', y', z', w'). In our pixel shader, we read the value $z_s = \frac{z'}{w'}$ from the depth buffer and we wish to calculate z. To reverse the transformation we first set

$$A = \frac{f+n}{n-f}$$

$$B = \frac{2fn}{n-f}$$

Using the projection matrix we get

$$z_s = \frac{Az + B}{-z}$$

This can be rewritten as

$$z = \frac{B}{-z_s - A}$$

By replacing A and B with their original values we get

$$z = \frac{2fn}{n-f} \cdot \frac{1}{-z_s - \frac{f+n}{n-f}}$$

Which can be rewritten as

$$z = \frac{2fn}{z_s(f-n)-(f+n)}$$

This allows us to quickly convert a screen space depth value to a view space depth value.

2.4 Ray resolve pass

The purpose of this pass is to take all the information gathered in the ray tracing pass and combine it with the G-buffer contents to compute the final color of each pixel.

2.4.1 Reusing neighboring rays

A common assumption in graphics is that the properties of neighboring pixels are similar to the properties of the current pixel. We can make use of this assumption to greatly increase the number of effective rays per pixel. Each pixel will reuse the results of all rays traced by its neighboring pixels. In this implementation we chose to reuse only results from immediate neighbors, which means each pixel has nine times as many effective ray trace results compared to what they would have with no reuse. A larger reuse radius can be used, but the risk of the above assumption being false increases, which may lead to undesirable artifacts.

2.4.2 Ray resolve pass implementation

The ray resolve pass is implemented as an OpenGL compute shader instead of a pixel shader for performance reasons. If we implemented it in a pixel shader, each pixel would have to extract the results of nine pixels to compute the final color of one pixel. The total number of result extractions would be $9WHR$ with W and H being the width and height of the image being rendered and R being the number of rays per pixel. To reduce this number, we process larger blocks of pixels in a compute shader instead. Each block of pixels in this case corresponds to a compute shader work group. Each block is of size $M * N$ and in each block all pixels that are not on the edge of the block will be able to compute their final color. The number of pixel colors computed in a block of size $M*N$ will be $(M-2)(N-2)$. The number of blocks that need to be executed is

$$n_{blocks} = \left(ceil\left(\frac{W}{M-2}\right)\right)\left(ceil\left(\frac{H}{N-2}\right)\right)$$

To minimize the number of ray trace results we need to extract, the values of M and N must be as large as possible. However, the OpenGL wiki [3] states that the minimum required value in OpenGL implementations for compute shader work group size is 1024. To ensure that the implementation is supported by all OpenGL implementations, we must choose M and N so that $M * N \leq 1024$.

As an example, let's say that our application is rendered with resolution 1600×900 pixels. We also assume that we use four rays per pixel. We choose $M = N = 32$ because

$32^2 = 1024$. With these parameters, a pixel shader implementation requires $9WHR = 51,840,000$ extractions. A compute shader implementation requires

$$MNRn_{blocks} = MNR\left(ceil\left(\frac{W}{M-2}\right)\right)\left(ceil\left(\frac{H}{N-2}\right)\right) = 6,856,704$$

extractions. Since the result of the ray tracing pass is stored in two texture arrays, each extraction requires at least two texture reads. A third texture read is potentially required to retrieve the color of the pixel that was hit by the ray. This means that the pixel shader implementation requires many more texture reads than the compute shader implementation. Through experimentation we found that switching from a pixel shader implementation to a compute shader implementation reduced the time needed to render a frame by around 10%.

The compute shader is split into two parts, one part where all data is extracted and one part where this data is combined. OpenGL Compute shaders are multi-threaded so the performance will depend on the number of threads the GPU supports. Each invocation of the compute shader is assigned a single pixel to work on. The shader will do a bounds check to make sure its pixel is within the bounds of the screen, because some assigned pixel coordinates will be outside the screen. This is because the pixel coordinates are shifted by $-(1, 1)$ so that the screen edges do not correspond to work group edges (which would prevent the screen edges from being colored). Also, the screen size may not be a multiple of the work group size which will cause even more coordinates to be outside of screen bounds. When this happens, the shader will simply set the weight of the out-of-bounds pixel coordinate to zero to prevent it from affecting the output.

The shader iterates through the layers of texture arrays produced by the ray tracing pass, and computes a weighted sum of colors together with the combined weight of all rays from the assigned pixel. These values are stored in shared memory arrays, one array for the total reflection color from each pixel and one array for the total weight from each pixel.

The GLSL function `barrier()` is used to ensure that the results of all pixels in the block are available to all threads before continuing. The shader invocations that were assigned pixels on the edge of the work group will immediately return after the barrier. They do not have data from all neighboring pixels available and therefore can not compute their final color. The other shader invocations will iterate through their immediate neighbors and sum up all colors and all weights. The final reflection color of each pixel will then be calculated as the total color of all pixels in a 3×3 area around the pixel divided by the total weight of all pixels in the same area.

The OpenGL wiki [3] also states that the minimum required shared memory size is 32 kilobytes. In our implementation, each shader invocation will store five 4-byte floating point values in shared memory, one floating point value for the weight and four for the color. The total amount of shared memory used by our implementation is $32^2 * 5 * 4 = 20480$ bytes or approximately 20 kilobytes. This is below the minimum size and we are guaranteed to not run out of shared memory.

The output of this pass is a single texture containing the final color of each pixel. The reflection colors and the colors from the G-buffer are combined linearly with

$$c = rc_r + (1 - r)c_d$$

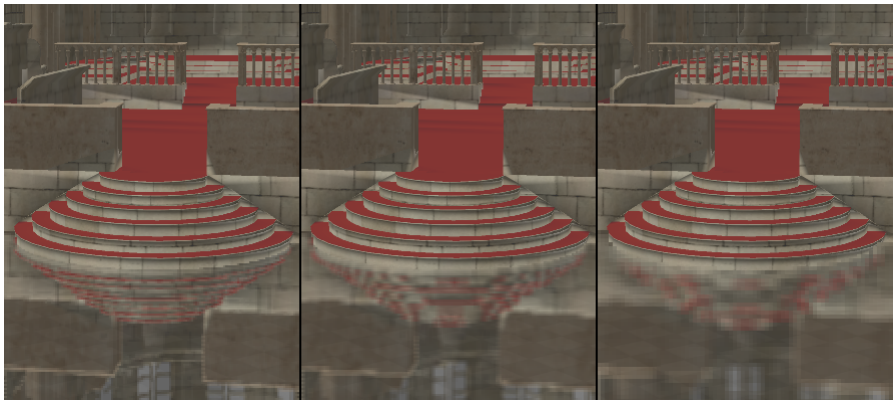

Figure 2.12: Comparison of ray tracing at reduced resolution. First image shows full resolution followed by half resolution and quarter resolution ray tracing. Artifacts are very noticeable, especially at quarter resolution. However, the performance gains are large and it is probably worth using a lower resolution for many applications. It is worth noting that on rough surfaces the differences between full resolution and half resolution are harder to see.

where r is the reflectivity of the surface, c_r is the reflection color and c_d is the G-buffer color.

2.5 Temporal anti-aliasing pass

To further smooth out the reflections, a very simple version of temporal anti-aliasing was implemented based on a talk by Tiago Sousa [11]. A jitter is added to the projection matrix every frame, so frames $f_0, f_2, f_4, ...$ have identical jitter and $f_1, f_3, f_5, ...$ also have identical jitter added. Just before the final image is rendered to the screen, a rendering pass is inserted that linearly combines the current frame with the previous frame, and this combination is rendered to the screen. This does make the image, especially the reflections, look much smoother. The disadvantage is that at low frame rates or fast movement, artifacts such as ghosting may occur.

2.6 Downscaled raycasts

A way to trade quality for performance is to execute the ray tracing pass at a lower resolution than the other rendering passes. Specifically, by downscaling the dimensions of the ray tracing result textures we can drastically reduce the number of rays we trace. A comparison between different downscaling ratios can be seen in Figure 2.12.

Chapter 3
Evaluation

3.1 Experimental Setup

To evaluate the performance and quality of the reflections, we executed the application we implemented and studied the output. This was done on a stationary computer with an AMD R9 290 graphics card. It is likely that the relative performance increases and decreases that occur when modifying parameters would be similar on other graphics cards. However, the actual frame times would vary depending on how powerful the card is.

3.1.1 Scenes

Two different scenes were used to test the different implementation parts and to measure the results. One scene contains a model of the Sibenik Cathedral [1] while the other scene contains procedurally generated spheres and boxes that are created when the application starts. In both scenes the floor is made reflective in order to test and measure reflection properties. A full view of the Sibenik scene can be seen in Figure 3.1.

3.1.2 Measuring error and time

To measure the error of an image, we compare it to a ground truth image. The ground truth is simply an image rendered with an unrealistic amount of resources to achieve a result that is as good as possible. The ground truth uses 32 rays per pixel, a G-buffer with 16 layers and a stride set to 1.

The actual error value is calculated by transferring both the ground truth image and the image being measured to the CPU, and then summing the absolute difference between the color value of each pixel in the images. This value is then divided by the total amount of pixels in the image to obtain resolution-independent error values. The absolute value of the error varies greatly depending on scene complexity and camera orientation. Therefore,

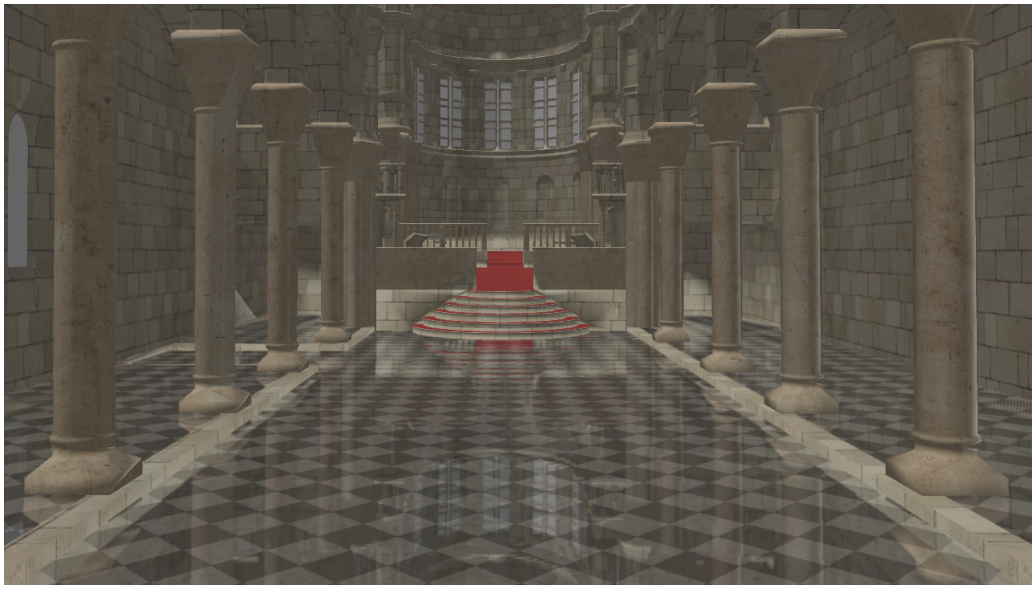

Figure 3.1: Full view of the Sibenik Cathedral scene used for testing and measurements

we do not make any conclusions based on the absolute error value. Instead, we study the relative change in error that happens when we modify parameters.

To measure the time required to render the reflections, we simply store the time before and after the ray tracing pass and ray resolve pass are executed. The OpenGL function `glFinish` is used both before and after the passes to ensure that all computations are finished before storing the times.

A common frame time target of real-time graphics applications such as games is 33 milliseconds per frame. With this frame time, the application can render about 30 frames each second, which is enough to be considered real-time. There are many things that need to be rendered in these 33 milliseconds, such as scene geometry, lighting and shadows. The amount of time we can spend on reflections depends on how much time is left over after the more important parts of the scene has been rendered.

3.2 Results

3.2.1 G-buffer depth

To measure the effect of different numbers of layers in the G-buffer, three different scene setups were used. The first measurement was done with the scene shown in Figure 3.2 and the results can be seen in Figure 3.3. This scene is constructed so that one layer is enough to render most, but not all, of the reflections.

The second measurement was done in a scene identical to the first one, except that the elephant was removed. The results of this can be seen in Figure 3.4. In this scene, almost all reflections can be rendered with just one layer.

The third measurement was done in the scene shown in Figure 3.5. This scene is constructed to require many layers to render correctly. The results can be seen in Figure

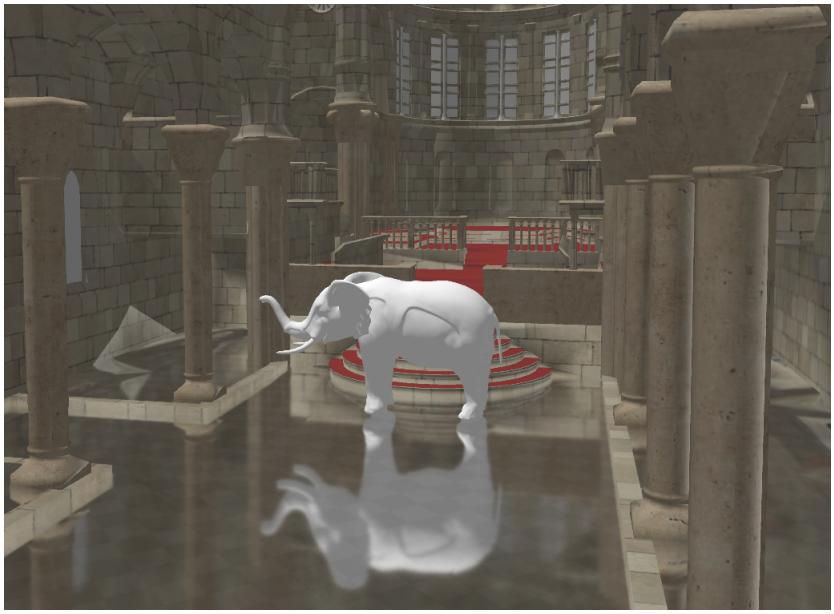

Figure 3.2: Ground truth rendering of scene used for measuring the effect of a deep G-buffer

3.6. The settings for all the measurements were four rays per pixel, a stride length of four, temporal anti-aliasing on, half-resolution ray tracing and a maximum of 250 steps for each ray.

The rendering time increases linearly as the number of layers increase. This is to be expected, since depth peeling is a very expensive process. All geometry in the scene must be rendered once for each layer. Additionally, the GPU cannot perform early z-testing while depth peeling is used, and the ray tracing algorithm must perform more texture reads and depth comparisons.

We can see that the second and third layers are the most important in terms of decreasing the error in all measurements. In the first and second measurements, the third layer causes the error to drop faster than the second layer. This is because in general, the second layer enables the parts of the scene objects that are facing away from the camera to be reflected, and the third layer causes the parts of the scene that are blocked by the closest scene objects to be reflected.

In the third measurement, the impact on the error of each layer slowly decreases as the number of layers increases. This is likely because the earlier layers add more information than the others, and the information they contain is closer to the camera. Objects that are further away from the camera are smaller when rendered because of the perspective projection. This further supports the statement that the first 2-3 layers are the most important.

3.2.2 Downscaling

Effects of downscaling can be seen in Figure 3.8. The measurement was done in the scene shown in Figure 3.7. This scene is constructed so that reflections can be properly rendered using only one layer. The settings used were four rays per pixel, a stride length of four

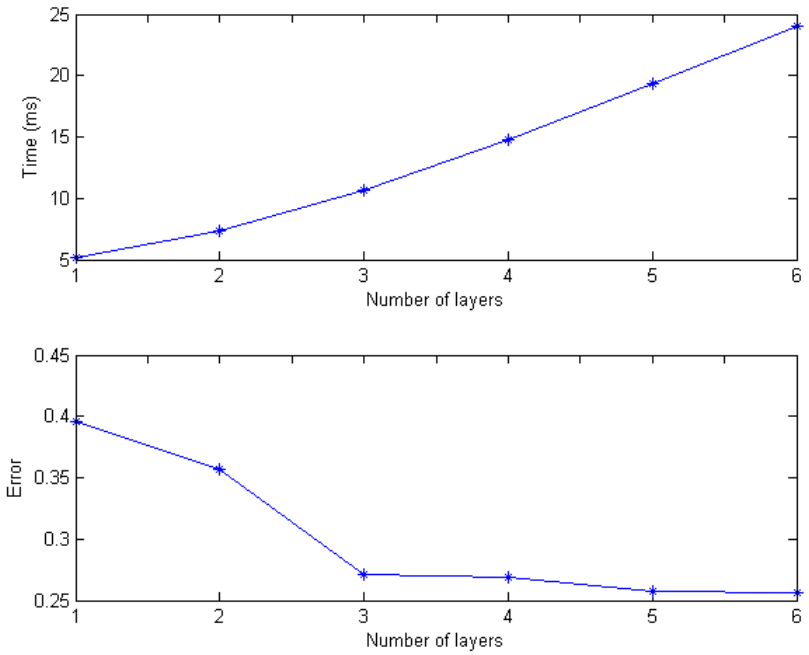

Figure 3.3: Measurement of error and rendering time with a different number of layers in the G-buffer. Measurement was done in the scene shown in Figure 3.2.

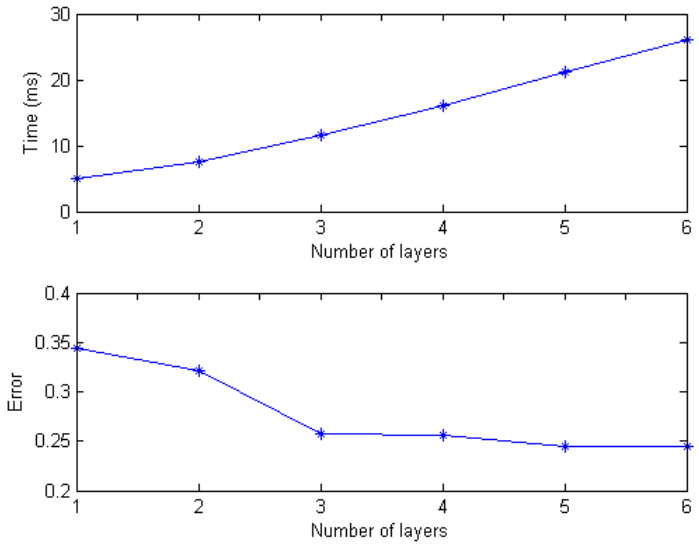

Figure 3.4: Measurement of error and rendering time with a different number of layers in the G-buffer. Measurement was done in the scene shown in Figure 3.2 without the elephant.

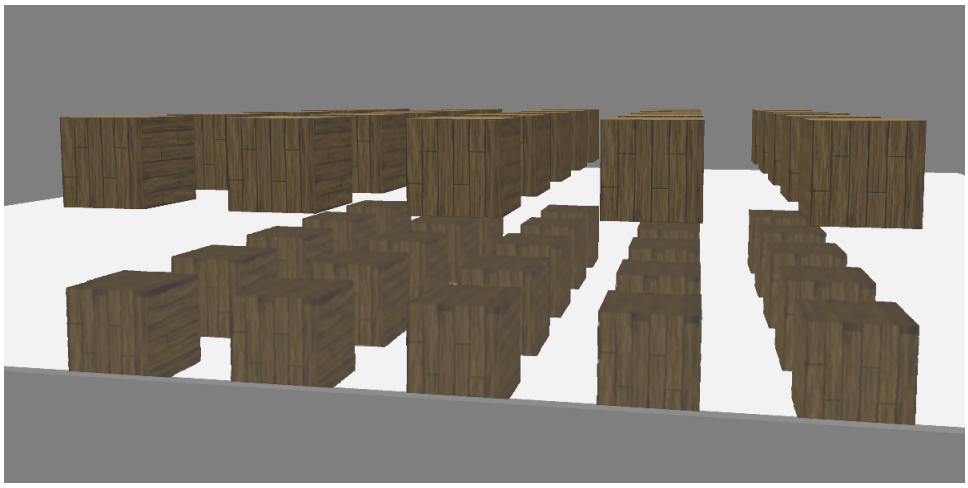

Figure 3.5: Ground truth rendering of box scene used for measuring the effect of a deep G-buffer

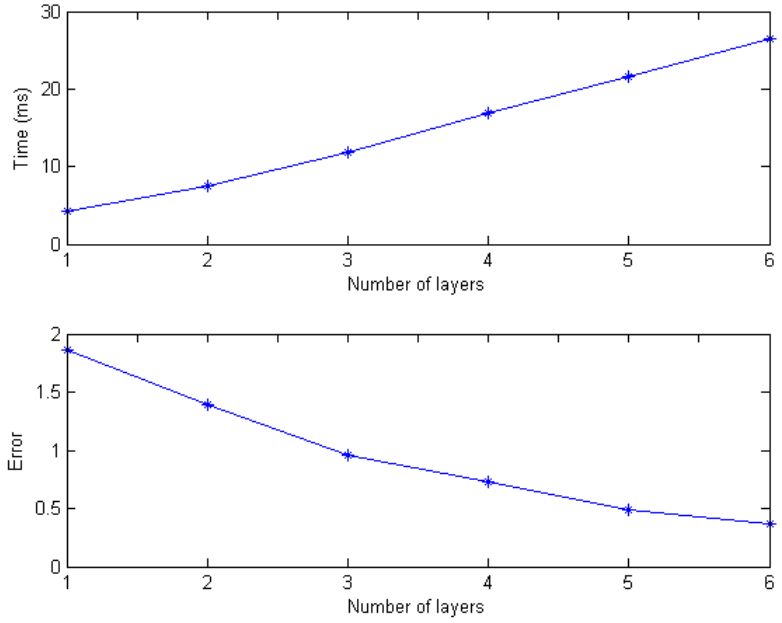

Figure 3.6: Measurement of error and rendering time with a different number of layers in the G-buffer. Measurement was done in the scene shown in Figure 3.5

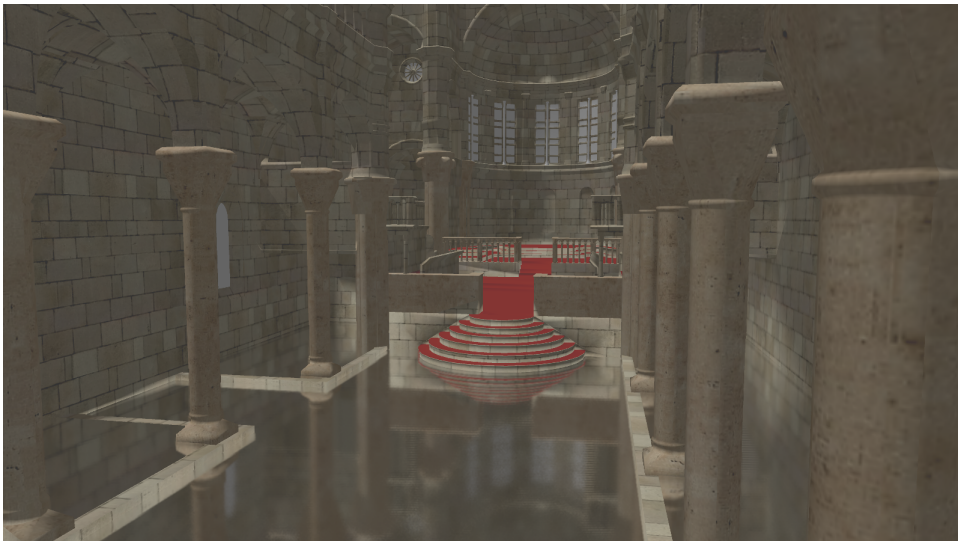

Figure 3.7: Scene used for measuring the effects of downscaling, different number of rays per pixel, and different stride lengths.

pixels, temporal anti-aliasing enabled, single-layer G-buffer and a maximum of 250 steps for each ray.

The increase in error when going from full to half resolution is much smaller than the increase that occurs when going from half to quarter resolution. This may be because of the neighbor reuse radius. When using results from the eight neighboring pixels, each individual pixel will still get a unique mix of ray trace results at half resolution. However, at quarter resolution, some pixels will have completely identical sets of ray trace results, which leads to a sudden degradation of image quality.

The results suggests that ray tracing at half resolution is the option that results in the best quality relative to time spent. The increase in error when going from full resolution to half resolution is minimal compared to the large reduction in rendering time.

3.2.3 Rays per pixel

The effect of modifying the number of rays per pixel can be seen in Figure 3.9. This measurement was done in the scene shown in Figure 3.7. The settings used were half-resolution ray tracing, a stride length of four pixels, temporal anti-aliasing enabled, single-layer G-buffer and a maximum of 250 steps for each ray.

The rendering time increases linearly as the number of rays per pixel increases. Using four rays per pixel seems to be a good idea in this scene, since the difference in error between 2 and 4 rays per pixel is much larger than the difference between the other samples. Two rays per pixel have very noticeable noise in the reflections while four rays per pixel give much smoother results. This also depends on the roughness of the surface. A roughness of zero is only going to require one ray per pixel, for example.

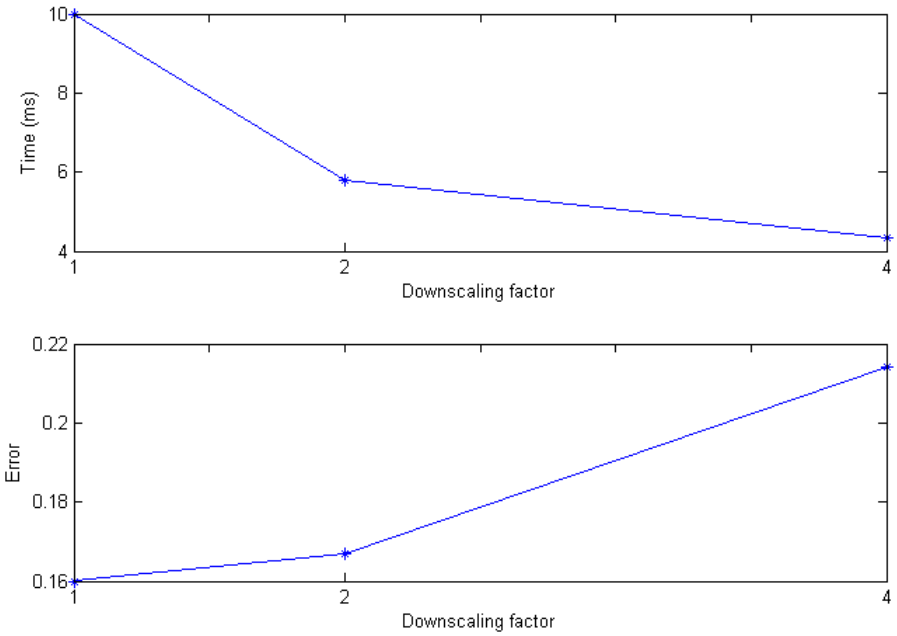

Figure 3.8: Measurement of error and rendering time with different downscaling factors.

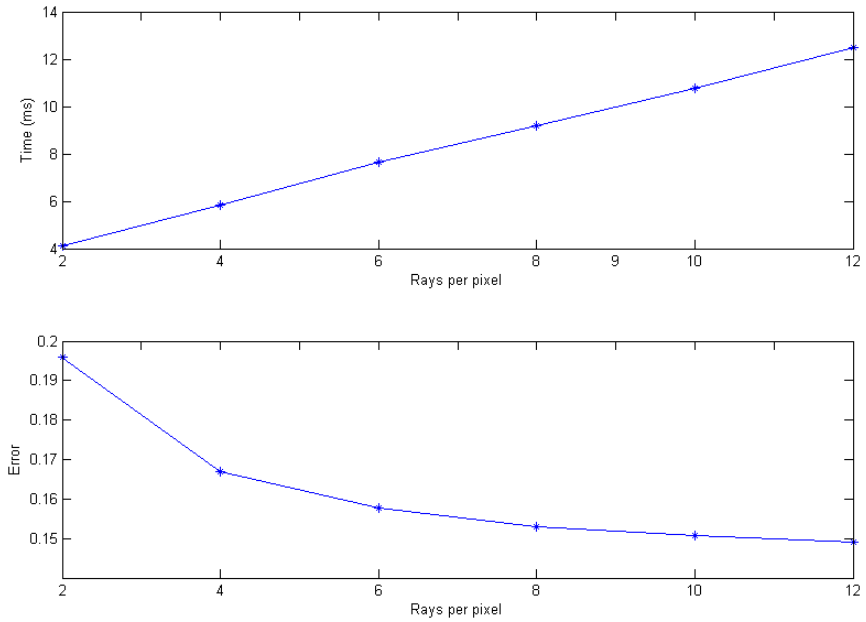

Figure 3.9: Measurement of error and rendering time with different numbers of rays per pixel.

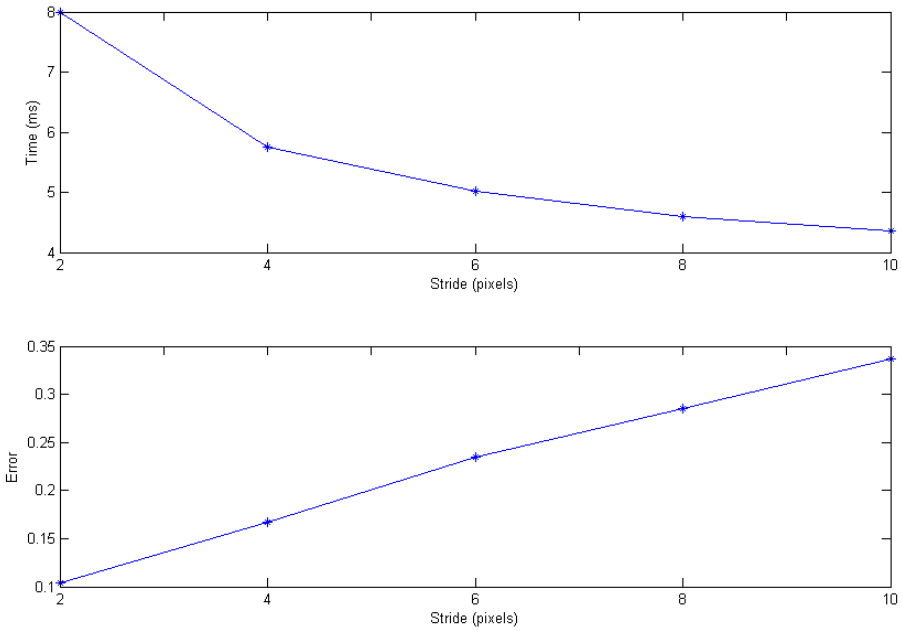

Figure 3.10: Measurement of error and rendering time with different stride lengths for the ray.

3.2.4 Stride

The stride length of the ray being traced is a very important factor in the ray tracing process. A long stride will cause the ray to travel much quicker, requiring fewer steps to potentially hit an object. But it can also cause the ray to pass through objects that it in reality should have hit, especially thin objects.

To evaluate the effect of modifying the stride length, we performed measurement in the scene shown in 3.7. The settings used were half-resolution ray traces, four rays per pixel, temporal anti-aliasing enabled, single-layer G-buffer and a maximum of 250 steps for each ray.

As can be seen in Figure 3.10, the rendering time quickly decreases as the stride length increases. Going from a stride length of one pixel to two pixels means that almost all rays will finish in half the time. The error increases steadily as the stride length increases.

Chapter 4
Conclusion

The results indicate that it is definitely possible to render decent quality reflections in screen space for real-time graphics applications. There are many variables to tweak when using this technique and most of them can be used to trade performance for quality or vice versa.

4.1 Evaluation of reflections

The degree of improvement in realism that reflections provide is fairly scene dependent. Screen space reflections work best when there are lots of reflective surfaces that are part of the scene background and not the focus of the scene. If, for example, the players of a game with screen space reflections have a lot of time to study the reflections, it will be easy to notice in most cases the unavoidable artifacts screen space reflections produce. Therefore it is a good idea to avoid relying on screen space reflections for very smooth surfaces such as mirrors or calm, large bodies of water. It is better suited for surfaces such as polished floors, smaller puddles of water, metal, windows and shiny objects like cars. In these cases it will enhance the image quality and increase the realism of the rendered images, and the limitations will be harder to notice.

The performance also greatly depends on the percentage of surfaces in the scene that are reflective. If only small parts of the scene contain reflective surfaces, then only a few rays need to be traced to render the reflections.

As long as the reflections are not the main focus of the scene, some of the optimizations that lower quality and increase performance can be safely used. In many cases, the effects of performing ray tracing at half resolution will be hard to notice unless the reflections are studied closely. It is even harder to notice in rough reflections because those reflections do not contain much detail.

Increasing the stride length provides a large increase in performance. As we saw in Figure 3.10, increasing the stride length from two pixels to four pixels reduced the ren-

dering time by 25%. The loss in quality that occurs when increasing stride length will in many cases not be noticeable, since reflections usually contain less detail than the actual geometry. Therefore it is a good idea to use a stride length longer than one pixel. The actual stride length value will depend on the performance requirements of the application.

4.2 Evaluation of deep G-buffers

While using a deep G-buffer will increase the amount of scene geometry that can be reflected, it has a very large performance cost. Because of this, we believe that the time spent constructing the deep G-buffers is better spent on other parts of rendering. In the future, when hardware is more powerful, using a deep G-buffer is likely a good way to increase the quality of screen space reflections, but with current hardware it just takes too much time.

As an example, let's say we have an application that needs to render at least 30 frames per second and spends half of the frame time, about 16 milliseconds, on rendering geometry, lighting and shadows. We can easily afford to add screen space reflections with a single layer to this application. Based on figures from the previous chapter, the reflections might cost around 6 milliseconds of frame time. However, if we wish to add reflections using two layers, we would have to render geometry, lighting and shadows twice. In this case, the cost of adding reflections would cost 22 milliseconds of frame time, which we cannot afford. Applications need to have a very large amount of time to spare to afford rendering the scene twice in one frame.

It is however important to note that having a deep G-buffer provides additional scene information for all screen space algorithms, not just reflections. If there are several screen space algorithms that take advantage of a deep G-buffer, the total improvement of scene quality relative to the performance cost of rendering a deep G-buffer will increase. In this case it might be a worthwhile trade-off.

In our implementation and measurements the scene was been rendered without any backface culling. This enabled the G-buffer layers to capture the parts of each scene object that were facing away from the camera. In some cases, it might be a better idea to ignore these parts of the objects, especially in scenes where the likelihood of them being reflected is not great. A scene where most rays will travel away from the camera would likely benefit from using backface culling, as it is likely that most rays will intersect with a surface that is facing towards the camera. In a scene where most rays travel towards the camera, the rays are more likely to intersect with surfaces facing away from the camera, and not using backface culling should yield the best results. As shown in Figures 3.3 and 3.4, the third layer has the largest impact on the error. If backface culling was enabled, the second layer would have the largest impact instead.

4.3 Artifacts and limitations

The greatest limitation of screen space reflections with a multi-layered G-buffer is that a lot of information needed to render the reflections are simply outside the bounds of the screen. As an example, if a player character in a game is standing on a reflective floor and

looking straight down, it will be impossible to render any screen space reflections. The necessary information will be unavailable. To mitigate this, a pre-rendered environment map of the scene can be used as a fallback method for when the screen space reflections cannot be rendered. This will avoid sharp differences between areas where screen space reflections cannot be rendered and areas where they can, but the environment map will only contain static geometry and any moving objects will not be reflected by it. A more complex method for rendering real-time reflections outside of the view frustum is described by Per Ganestam and Michael Doggett [8]. Their method uses a bounding volume hierarchy to store information about geometry close to the camera, and a cube map to store information about objects further away. These data structures are updated each frame which enables them to capture moving objects. Some of these ideas could possibly be used to complement our screen space reflections.

4.4 Future development possibilities

In the current implementation, all reflective pixels trace the same amount of rays. However, if the pixels have varying roughness, we could potentially improve the overall result by doing a varying number of ray traces per pixel. A pixel with a very low roughness does not require many rays to produce an accurate result, since all rays will bounce in nearly the same direction. A pixel with high roughness will require more rays because the directions they bounce in are spread out.

The temporal anti-aliasing in the implementation could likely be improved. Right now we store the previous frame and linearly combine it with the current frame. One potential way to improve this would be to instead store the ray trace results of the previous frame. These results could then be reused in the ray resolve pass, which would double the amount of effective rays per pixel.

Bibliography

[1] Sibenik Cathedral 3D mesh. Original by Marko Dabrovic, with holes corrected by Kenzie Lamar at Vicarious Visions and high-resolution texture and bump maps painted by Morgan McGuire. Downloaded from `http://graphics.cs.williams.edu/data/meshes.xml`. Retrieved 2016-04-26.

[2] OpenGL. `https://www.opengl.org/`. Retrieved 2016-06-19.

[3] OpenGL. Compute Shader. `https://www.opengl.org/wiki/Compute_Shader`. Retrieved 2016-05-20.

[4] OpenGL Shading Language. `https://www.opengl.org/documentation/glsl/`. Retrieved 2016-06-19.

[5] Unity Manual. Reflection Probe. `https://docs.unity3d.com/Manual/class-ReflectionProbe.html`. Retrieved 2016-05-20.

[6] Unreal Engine documentation. Reflection Environment. `https://docs.unrealengine.com/latest/INT/Resources/Showcases/Reflections/`. Retrieved 2016-05-20.

[7] Cass Everitt. Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7, 2001.

[8] Per Ganestam and Michael Doggett. Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer*, 31(10):1395–1403, 2015.

[9] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

[10] Shawn Hargreaves. Deferred shading. *Game Developers Conference*, 2004.

[11] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*, 2011.

[12] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[13] Michael Mara, Morgan McGuire, Derek Nowrouzezahrai, and David Luebke. Fast global illumination approximations on deep g-buffers. *NVIDIA Corporation*, 2014.

[14] Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, 2014.

[15] Peter Sikachev and Nicolas Longchamps. Reflection system in Thief. *SIGGRAPH 2014 Advances in Real-time Rendering in Games course*, 2014.

[16] B Smith. Geometrical shadowing of a random rough surface. *IEEE transactions on antennas and propagation*, 15(5):668–671, 1967.

[17] Tomasz Stachowiak. Stochastic screen-space reflections. *SIGGRAPH 2015 Advances in Real-time Rendering in Games course*, 2015.

[18] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.

DEPARTMENT OF COMPUTER SCIENCE | LUNDS TEKNISKA HÖGSKOLA | PRESENTATION DAY 2016-06-03

MASTER'S THESIS Multi-layered G-Buffers for Real-Time Reflections
STUDENT Mattias Simonsson
SUPERVISOR Michael Doggett (LTH)
EXAMINER Flavius Gruian (LTH)

Screen space reflections with multiple layers

POPULAR SCIENCE SUMMARY **Mattias Simonsson**

Real-time reflections can be rendered quickly with decent quality in screen space. We investigate and evaluate the performance of a method that improves the quality of the reflections

Real-time graphics is all about approximating the look of the world in a limited amount of time. However, as computers grow more and more powerful, the amount of computations that can be done in the same amount of time increases.

Reproducing the behavior of reflective surfaces, like mirrors and water, is expensive but still possible to do thanks to the power of modern hardware. To render, or draw, reflections on a surface, we need to find out where the light that bounced on the surface into our eyes came from. Because of the complexity of scenes in many modern applications, this takes a lot of time to do accurately.

By reducing the complexity of the scene in some way, we can render the reflections faster. A way to do this is to render the reflections in screen space. This can be thought of as taking a photograph of the scene and then drawing reflections on top of this photograph later.

Reducing the amount of available information before rendering reflections improves performance but reduces accuracy of the reflections. As an example, if a player is standing in front of a mirror in a third-person game, the mirror will not be able to reflect the face of the player. The information is simply not available.

In my thesis I combine screen space reflections with a technique known as depth peeling. With depth peeling, the scene is rendered into multiple 2D images, or layers, instead of just one, with each layer containing unique information. This will improve the accuracy of the reflections but also reduce performance because the scene needs to be rendered multiple times. I evaluate and compare the performance cost relative to the increase in image quality in my thesis.

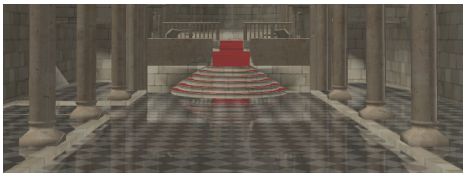

Figure 1: Example of screen space reflections

The results suggest that while using multiple layers does improve image quality, this improvement is in most cases too small to justify the very large performance cost of depth peeling. The results also suggest that if multiple layers are used, the amount of layers should be two or maybe three. In general, each successive layer adds less information than the previous.