Tampere University

Tuomas Katajisto

RHOMBIC DODECAHEDRON MAPPING FOR INDIRECT LIGHTING WITH VOXEL BLOCKS

Storing path traced specular global illumination data

Master's thesis
Faculty of Information Technology and Communication Sciences
Examiners: Dr. Markku Mäkitalo
M.Sc. (Tech.) Julius Ikkala
January 2025

ABSTRACT

Tuomas Katajisto: Rhombic dodecahedron mapping for indirect lighting with voxel blocks
Master's thesis
Tampere University
Master's Programme in Information Technology
January 2025

Indirect lighting is light that hits the surface after bouncing from another surface, not directly from the light source. Mirror-like sharp reflections are a form of indirect specular lighting. Rendering sharp reflections in real time applications such as video games has many solutions with varying trade-offs. Most of the existing methods work in real-time, requiring high performance hardware or producing compromised results.

The rhombic dodecahedron mapping method proposed in this work is a new approach to storing lighting information for a voxel block, that takes advantage of the restrictions on scenes formed from axis-aligned voxel blocks to store detailed information about the environment around the block for rendering sharp reflections and fetching other stored lighting information in real time.

The method stores incoming light for a voxel block into six hemispheres, one for each side of the block. These hemispheres are stored in a pyramid shaped mapping, six of which placed on the sides of a block form a rhombic dodecahedron shape. The stored hemispheres are for the center of each side of the block, requiring depth data to be stored in the mapping to allow approximated sampling for other points in the block's surface.

Although the method is specialized for sharp reflections, it can be used generally to map light probe data for a voxel block and can also be used for low resolution diffuse lighting in addition to specular lighting.

The method results in reflections reasonably close to the path traced reference, with some cases where the method has inherent errors due to storing a single hemisphere for an entire side of the cube. The diffuse lighting gotten with the method is low resolution and can have large differences with the path traced reference, but is usable for use cases where high resolution is not required.

Keywords: rhombic dodecahedron, indirect lighting, specular lighting, reflection

TIIVISTELMÄ

Tuomas Katajisto: Epäsuoran valaistuksen varastointi vokseleille rombidodekaedreihin
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Tammikuu 2025

Epäsuora valo on valoa, joka ei osu pintaan suoraan valonlähteestä, vaan jonkin toisen pinnan kautta. Ympäristön heijastuminen sileästä pinnasta, kuten peilistä, on epäsuoran valon suuntaheijastumisen muoto. Tällaisten heijastusten reaaliaikaiseen renderöintiin, esimerkiksi videopeleissä, on useita ratkaisuja, joilla on kaikilla omat vahvuutensa ja heikkoutensa. Useimmat näistä ratkaisuista toimivat reaaliajassa, joten niissä joudutaan tekemään merkittäviä kompromisseja tuloksien laadun ja vaaditun laskentatehon välillä.

Tässä työssä ehdotettu menetelmä varastoi etukäteen lasketun valoinformaation jokaista vokselitiiltä (16 x 16 x 16 pienestä vokselista koostuvaa kokonaisuutta) kohden rombidodekaedrin tahkoihin . Menetelmä hyödyntää vokselitiilien muodostamien ympäristöjen ominaisuuksia ja varastoi valoinformaatiota jokaista tiiltä kohden. Varastoitua informaatiota voidaan käyttää terävien heijastusten renderöintiin reaaliajassa.

Menetelmä varastoi tiiltä kohti tulevan valon kuuteen pallonpuoliskoon. Jokaista tiilen kuutta sivua kohti varastoidaan yksi pallonpuolisko, jonka keskipiste on tiilen sivun keskipiste. Valoinformaation lisäksi varastoidaan syvyysinformaatiota, jota käytetään valoinformaation arviointiin pinnan muille pisteille kuin tiilen sivujen keskipisteille.

Menetelmän tuottamat tarkat heijastukset ovat melko lähellä säteenseurannalla renderöidyn referenssikuvan heijastuksia. Virheitä tuottaa menetelmän tapa käyttää yhtä pallonpuoliskoa tiilen sivua kohden. Menetelmän merkittävänä rajoitteena on se, että sen tuottamat heijastukset eivät ota huomioon muutoksia ympäristössä.

Vaikka menetelmä keskittyy tarkkoihin heijastuksiin käytettävän informaation varastointiin, voidaan sitä myös käyttää matalan resoluution hajavalaistukseen. Menetelmän tuottama hajavalaistus on epätarkkaa ja eroaa usein merkittävästi referenssikuvasta, mutta se on käyttökelpoista joissain käyttötapauksissa.

Avainsanat: rombidodekaedri, epäsuora valaistus, suuntaheijastuminen, heijastuminen, peilaus

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

- ☒ No
- ☐ Yes

PREFACE

Working on this thesis has been the largest learning journey I have undertaken so far. I want to thank both of my instructors, Markku Mäkitalo and Julius Ikkala, for all of the knowledge and feedback shared during this process and the computer graphics courses that inspired me to pursue computer graphics further. Additional thanks to Julius for the advice and discussions about my game engine project that led to the ideas in this thesis.

I want to thank my girlfriend Jenni, my parents and my grandparents for their constant support. I also wish to thank my friends Matias Järvenpää and Leevi Törölä for all the great discussions about our studies and for helping with proofreading the thesis.

Pomarkku, 7th January 2025

Tuomas Katajisto

CONTENTS

GLOSSARY OF ABBREVIATIONS

2D	Two dimensional
3D	Three dimensional
BLAS	Bottom level acceleration structure
BRDF	Bidirectional reflectance distribution function
BVH	Bounding volume hierarchies
RDM	Rhombic dodecahedron mapping
SSAO	Screen-space ambient occlusion
SSR	Screen-space reflections
TLAS	Top level acceleration structure
Voxel block	A repeatable block of 16x16x16 small voxels

1. INTRODUCTION

Voxels, small cubes that make up a larger scene, have been very popular in independent video game development for a long time. This is mostly due to them being an easy way to create 3D environments and objects, without the need for complex 3D modeling. Most computer graphics research is not focused on voxel graphics, since they are not very relevant for photorealism focused cutting-edge graphics used in productions with large budgets. By taking advantage of the limitations of voxel scenes, improved solutions to many common problems can be found that don't work with the common way of representing a 3D scene as a list of arbitrary triangles. One common problem is how to handle indirect lighting in real-time computer graphics applications such as video games.

Indirect lighting is light that does not hit a surface straight from the light source. A room can be well lit, even though direct sunlight through the window can't reach most surfaces, because the light bounces around the room from the surfaces sunlight is reaching directly. Sometimes light can bounce from a surface so consistently that the surface acts as a mirror displaying the viewer a sharp reflection of some other object.

Calculating indirect lighting takes a lot of time, due to reasons expanded upon later in this work. Real indirect lighting calculations sometimes take too much time for usage in real-time rendering, depending on the platform and the desired quality of the indirect lighting. In real-time rendering, a new image needs to be produced at a rate that appears fluid to the user and reflects user actions without a noticeable delay, e.g. 30 or 60 times a second.

One technique is calculating the indirect lighting beforehand for the entire scene and then using this data during the runtime of the program. This way the data can be calculated once on a powerful machine and stored with the program. In order for the program to utilize this pre-calculated data, the scene's lighting can't change after the lighting has been calculated. This solution usually does not allow for storage of sharp reflections of the environment, due to them being dependent on the viewing angle.

This work presents a new way of storing lighting data in scenes consisting of voxel blocks: the rhombic dodecahedron mapping. Voxel blocks provide a natural framework for storing the lighting data and the mapping will store data usable for sharp reflections of the environment.

2. THEORY

When rendering an object, the color of its surface is determined by a shading model. The shading model determines what factors affect the surface's color. There are many different shading models, that all take into account different things about the scene, they usually account for the angle between the light's direction and the surface and the angle between the view direction and the surface among other things. [1]

Shading models can be categorized into photorealistic and stylized models [1]. Photorealistic models attempt to model the behavior of light in the real world, and stylized models create a stylized appearance that matches artistic goal [1]. This work focuses on photorealistic shading models.

2.1 Photorealistic shading

The basis of photorealistic shading is the rendering equation presented by Kajiya in 1986 [2]. It states that the light traveling between two points is

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right]$$

This means that the amount light traveling from a point to the camera, $I(x, x')$ in the scene in figure 2.1 is calculated by first taking into account if there is something blocking the path of the light. In our case, where we do not account for materials that let light trough them $g(x, x')$ is 1 if there is nothing blocking the path and 0 if there is something in the way. If there is something in the way, the entire equation becomes 0. The next step is adding the light emitted from the point towards the camera: $\epsilon(x, x')$. Most materials do not emit light, but for example glowing surfaces or the surface of a light bulb do emit light into the scene. Without any light emitting points in the scene, it would be completely black. [2]

The last part of the equation is an integral which goes over all of the other surfaces in the scene and adds the light coming from the other surfaces to the original point $I(x', x'')$. This incoming light is multiplied by $\rho(x, x', x'')$, which describes how much of the incoming light to point x' from point x" is reflected towards the camera at point x. [2]

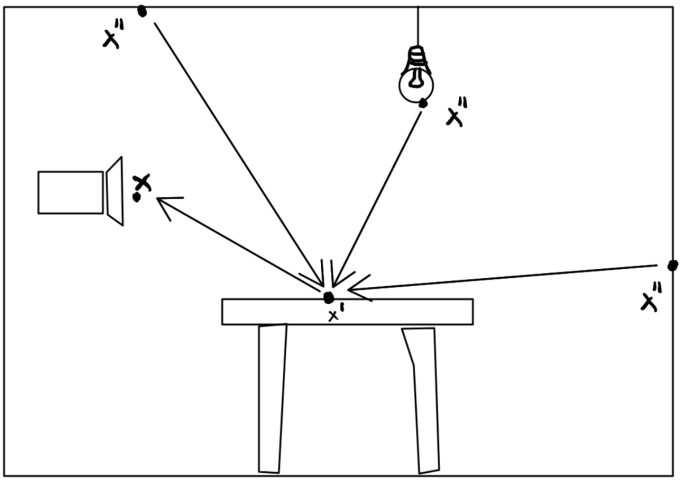

Figure 2.1. *The points x, x' and x" on the first evaluation of the rendering equation between the camera and a point on the table.*

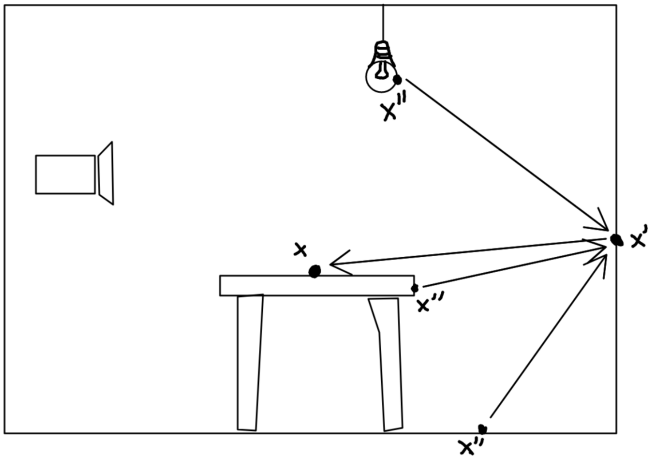

Figure 2.2. *The points x, x' and x" on one iteration of evaluating the recursive I(x', x") inside the integral.*

As we can see from the usage of $I(x', x'')$ inside of the integral, this equation is recursive. While solving $I(x, x')$, it's necessary to solve the rendering equation between x' and all other points in the scene. The new points for a recursive evaluation of the rendering equation are shown in figure 2.2. Solving $I(x', x'')$ means solving more rendering equations for all of the other points. [2]

The rendering equation is often solved using Monte Carlo methods that provide an increasingly accurate approximation of the rendering equation when more time is spent calculating the solution. These methods were designed to solve similar problems during the Manhattan Project. They work by randomly sampling the integrand and use these samples to approximate the integrals value. Path tracing is one way to apply these methods to solving the rendering equation. [1, 2]

Path tracing works by tracing the path of a ray of light in the reverse order from the

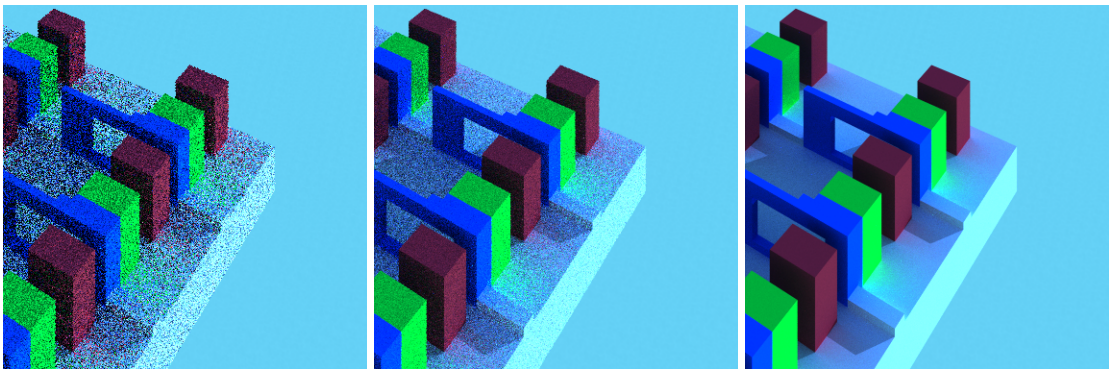

Figure 2.3. *The effect on the image of how many rays of light are sampled per pixel. Sample counts from the left are 1, 8 and 512.*

camera, until it hits the sky or another light source. This is done in the reverse order, because a very small fraction of the rays of light from a light source end up hitting the camera. By going from the camera backwards we evaluate only the rays that will hit the camera, saving massive amounts of compute time. A ray is cast into the scene for every pixel of the image being generated. Every time the ray hits an object, the bounce direction is assigned from a random distribution based on the properties of the object's material. These bounces are calculated until the ray hits a light source or we exceed some arbitrary limit of allowed bounces set to prevent the ray bouncing around in darkness forever. Taking the average of multiple rays sampled per pixel starts to provide a more accurate solution to the rendering equation. The amount of samples per pixel required depends on the scene and required fidelity. [1] Evaluating too few paths per pixel results in a bad approximation of the rendering equation and high variance. This means that two points near each other could have very different approximations of the rendering equation, even though in reality they should be very similarly lit. An example of this is visible in figure 2.3. [1]

2.1.1 Shading models and BRDFs

In computer graphics, light reflecting from the surface is often classified into diffuse and specular categories. Perfectly diffuse surfaces reflect light equally into every direction while perfectly specular surfaces reflect light exactly in one direction, the reflected version of the light's incoming direction. There are also retro-reflective surfaces that reflect the light back in the same direction that it came from, but those are outside the scope of this work. Perfectly diffuse or perfectly specular surfaces are rare in reality and most surfaces are somewhere between. [3]

A bidirectional reflectance distribution function (BRDF) is a way to describe the way light interacts with a surface. A BRDF describes how much of the light coming in from one direction is reflected in another direction. It's used in the function we represented with ρ

Figure 2.4. *Breakdown of Blinn-Phong lighting model. The images from left to right are: ambient lighting, diffuse lighting, specular lighting and combined end result.*

in the rendering equation. There are many BRDFs with varying complexity and accuracy to how light interacts with materials in the real world. They also support varying amounts of different material properties that affect the way light reflects off surfaces.

A basic BRDF, the Blinn-Phong reflection model is a common way to model the light's reflection off a surface. Figure 2.4 displays the three factors of the Blinn-Phong shading model applied to rendering the "Lucy" model [4] and the combined final image. Although the ambient factor is not a part of the BRDF, it's added in place of indirect lighting when indirect lighting is not being taken into account, because otherwise parts of the model that are not directly visible to the light would be completely black. The Blinn-Phong model offers fairly little control over the look of the material. The material's shininess value and color can be adjusted. Increasing this value makes the highlights smaller and their edges more defined. A comparison of two values 16 and 64 used for rendering the same model can be seen in figure 2.5.

A more accurate BRDF, the Trowbridge-Reitz reflection model, also known as GGX provides more accuracy to the material's properties and higher customizability. It's a microfacet model, meaning that it models the microscopic variations in the material's surface that affect the way light reflects off the surface. Light reflects perfectly mirrored off the surface, but the microscopic part of the surface that the ray of light hits might be pointing in a completely different direction than the visible surface. The microscopic variation of the surface can't really efficiently be taken into account when rendering, so we approximate it with a macrosurface that reflects light with the same distribution that the microsurface would. The rougher the material is, the more varied the microsurface is and the light is distributed more evenly everywhere around the point. A perfectly smooth surface will have a perfectly smooth microsurface and reflect light perfectly mirrored. The Trowbridge-Reitz model also allows us to have much more varied materials. [3]

Figure 2.5. *The left image is rendered with a low shininess value for the material and the right image is rendered with a high shininess value.*

2.1.2 Direct lighting

Direct lighting is light that hits a surface straight from the light source and then bounces from the surface towards the viewer. Direct light can be from different types of light sources. Light coming from point lights is light originating from a singular point in space. Directional light, which sunlight can be modeled as since it's originating from so far away, is light that is coming from the same direction regardless of the point in space. Area lights are lights that have area, and the light is originating from the area.

In path tracing, direct lighting can be evaluated for some light sources, such as the sun, by adding the amount of light reflected towards the viewer to the evaluation of lighting for a pixel. For area light sources, direct light is added when a ray ends up hitting the light source straight after bouncing off the first surface.

Direct light is easy to evaluate in real time, since we can know the light's location and we can calculate its effect on the surface's lighting based on the surface and the viewer's location. The only factor that is not trivial is whether the light from the source can reach

the point, that is, if the point is in the shadow of the light source. For area light sources, direct light evaluation is not trivial, since the light is not originating from a single point or from a single direction.

2.1.3 Shadows

Shadows are important for realistic images and also crucial for giving the viewer information about object placement. When light from a light source can't reach a point because of an object that is in the way, the point is in shadow of the light source. The object is called the occluder. The part of the shadow which is completely in shadow is called the umbra, and the part of the shadow that is only partially occluded from an area light source is called the penumbra. [1]

When path tracing, occlusion from point lights and directional lights can be determined by casting a ray towards the direction of the light source. If the ray hits something before reaching the light source, the point the ray was cast from is occluded from the light source. For area lights the ray will just not be able to bounce and hit the area light if there is something blocking it. If a part of the light is visible, less rays will go to it than if it was completely visible resulting in a less bright surface. This is often optimized by checking the light's visibility for a random point in the area light's surface.

Rendering realistic shadows in real time rendering without path tracing is hard, because it's not obvious what objects could occlude a light source and how much of the light source is occluded. Techniques for rendering shadows in real time will be discussed later.

2.1.4 Indirect lighting

Indirect light is the same light that is direct light to some other point, but instead of bouncing straight to the viewer from the surface, it bounces to some other surface or surfaces before reaching the viewer. Figure 2.6 shows an example path of indirect light. Indirect light is especially important in points that do not receive direct light. Without indirect light, these points would be completely black. In the real world a room with a window might receive direct light only in a small portion of the room's surface area, but the indirect light bouncing around still lights the entire room well.

Path tracing evaluates indirect lighting without extra complexity by just tracing the rays bouncing around the scene. Indirect lighting in real time programs is difficult and has many different solutions that will be discussed later.

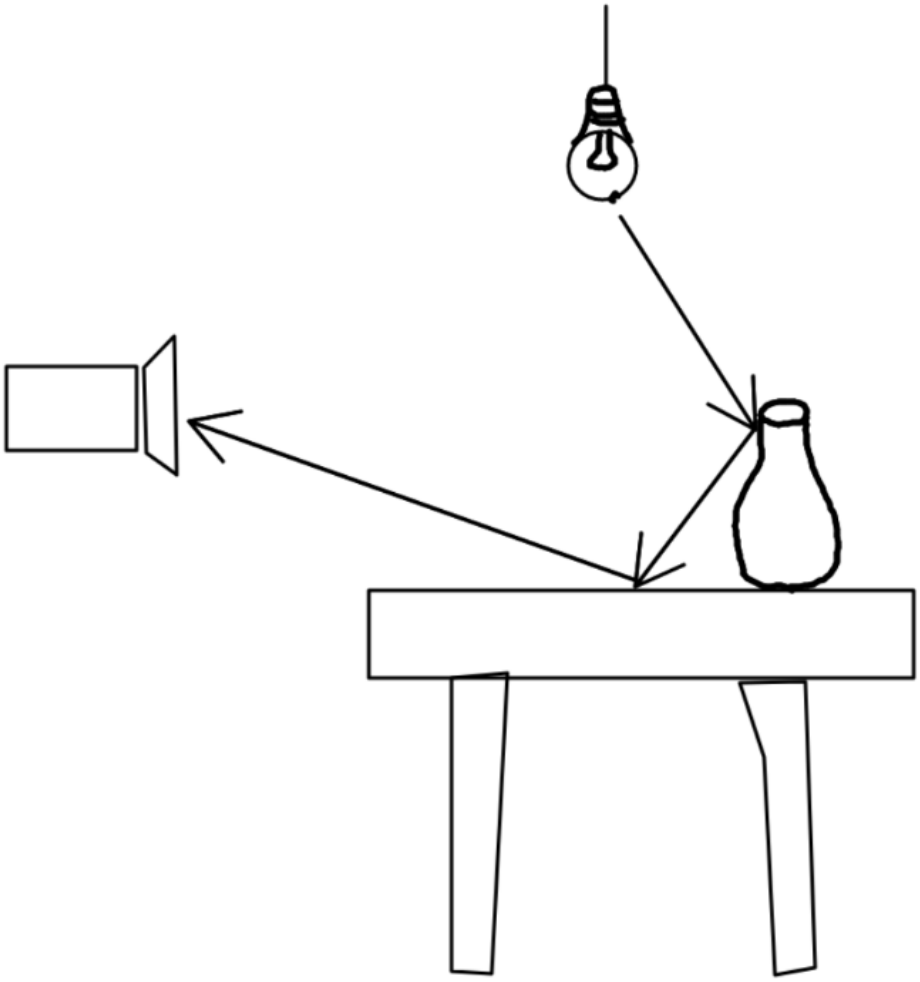

Figure 2.6. *An example path for indirect light. Light coming from the lamp is bouncing from the vase to the table's surface and bouncing from the table to the viewer. Depending on the vase's material, it could absorb some of the light, resulting in the light hitting the table after hitting it to have a different color.*

2.1.5 Reflections

Reflections are a specialized case of specular indirect lighting. Sharp reflections happen when a surface is smooth enough that it reflects light like a mirror. This happens especially with metallic surfaces but is also visible on smooth enough non-metallic surfaces. The reflection can be slightly blurred if the surface is not perfectly smooth.

Reflections work the same as indirect lighting when path tracing. If the surface is smooth enough, it's possible to do an optimization in path tracing, that reduces variance and noise, by just perfectly reflecting every ray that hits the surface instead of using a low roughness. [3]

Storing pre-calculated lighting data for reflections is often more complicated than regular indirect lighting, due to the sharp resolution of reflections and the fact that the view direction affects the reflection.

2.2 Real time approximations

For a long time, most of the light's interactions with the scene described above couldn't have been rendered with physically accurate techniques for every frame of a video game. There is a long history of increasingly accurate approximation techniques for all of them. These approximations are cheap to calculate and usually produce results that are similar looking to the result that would be achieved by doing physically based path tracing. With modern hardware and future hardware improvements there is a move towards real time path tracing, but it currently requires lots of performance and complex techniques to work in real time, making these approximations still relevant today.

2.2.1 Indirect lighting

There are real time approximations for indirect lighting that do not require any pre-calculation. The advantage of these solutions is that changes in the scene can affect the lighting, since the lighting doesn't rely on pre-calculated data calculated on a certain version of the scene. The downside is that these techniques have limited time and can't achieve the same quality that pre-calculated techniques can by spending hours or days rendering.

Screen-space ambient occlusion

Screen-space ambient occlusion (SSAO) was introduced by Crytek for their CryEngine 2 engine. It darkens areas surrounded by obstructions for light. It works by looking at how far the point, for which lighting is being evaluated, is from the camera. This is also called its depth. Then nearby points are sampled, and their depth is looked at to determine if the original point is hard to reach for light from the environment to create a darkening factor. [5]

SSAO is a screen-space method, meaning that it only uses the information that is drawn on the screen. This makes the method fast even when the scene is complex, since the method operates on the displayed image which stays at a constant size. There are multiple other screen-space methods used to approximate other things, that we will get to later. [1]

This darkening factor is only applied to the ambient part of a lighting model, like the one visualized in figure 2.4. This was partially done to make the model look less realistic, which was a goal of the new engine development. [5]

This approximation for indirect lighting is not well suited for trying to approximate photo-realistic lighting, since it only darkens the ambient lighting factor and does not take into account what kind of light sources are in the scene nor the materials of nearby surfaces. It's relatively simple to implement and computationally cheap.

Virtual point lights

Virtual point lights are approximations for indirect lighting that usually approximate one bounce of light from a surface by placing some kind of light source at the point where light from a light source hits a surface. The light from this placed light source approximates light that would have bounced from that surface. [1]

Reflective shadow maps are a real time adaptation of virtual point lights. Reflective shadow maps are images rendered from the perspective of the light source. The pixels of the rendered image (known as texels when the image is used as an texture) are used as light sources when rendering other objects. The texels are sampled when determining the lighting for a point. [6]

One bounce of light is often enough to create believable results, but virtual point light techniques have issues with occlusion of the bounces, since shadow maps, which are discussed later on, can't be generated for all of these light sources placed on surfaces to approximate a bounce of light. These issues can be mitigated by additional techniques. [1]

ReSTIR global illumination

ReSTIR GI is a relatively new technique for global illumination. It improves the efficiency of path tracing to a point where a path traced image with only 1 sample per pixel looks close to a path traced image rendered with significantly more samples per pixel. ReSTIR requires 9.3x to 166x less samples per pixel for the same quality as regular path tracing. [7]

ReSTIR GI works by sharing information about paths that contributed important lighting information across time and across the different pixels in the path traced image.

Real time path tracing is a desirable technique, due to how generalized it is. Path tracing does not need the same kinds of tricks for shadows, reflections or indirect light that rasterization requires. ReSTIR GI makes it possible to use this technique for real time rendering on powerful hardware. However, implementing ReSTIR GI is complex, and it requires high-end hardware from the user of the program.

2.2.2 Shadows

Real time approximations for shadows are more common than real time approximations for indirect lighting. This is probably a result of two major things. There are multiple viable techniques for getting believable results in real time. Shadows are also expected to be more reactive to the scene content being dynamic. The lack of changes to the indirect lighting of a scene caused by a character moving around is probably not going to be

noticed by a player. If the shadow of the character wouldn't move with the character or a character didn't have a shadow at all, that would be noticeable to the player.

Shadow mapping

Shadow maps are a technique for having arbitrary objects cast shadows on arbitrary surfaces. Shadow maps work by rendering the scene from the light's perspective. When rendering the scene, a z-buffer is formed. The z-buffer is a texture that stores the distance from the camera for every point in the rendered image. This z-buffer is stored from the light's perspective and called the shadow map. When rendering a point in the scene in the final image, it's known where it should be on the shadow map. The distance of the point from the light is also known. If there is something blocking the light reaching the point being rendered, the shadow map will have stored the distance of the object in front of the point being rendered. During the rendering of the point in the final image, there will be a discrepancy between the distance from the light to the point we are rendering and the distance to the point the light sees while looking into the direction of the point being rendered. This discrepancy means that there is something blocking light from the light source to the point being rendered. [8]

The shadows achieved by using shadow maps have hard edges. The edges can also be visibly pixelated depending on the shadow map's resolution. These issues can be reduced without a lot of added complexity by a technique called percentage closer filtering (PCF). The technique works by taking multiple samples from the shadow map to see whether the point being sampled is near non-shadowed points even though it is in shadow or vice versa. These points that are on the edge of the shadow have the shadow's effect reduced, making the edges of the shadow softer. Another technique for decreasing the issues with shadow maps is resolution enhancement. The effective resolution can be increased by rendering the shadow map at an angle that is closer to the view angle. There are also more advanced techniques such as cascading shadow maps that include rendering multiple shadow maps. [1]

There are many different PCF techniques that have different ways of sampling the shadow map. One of them is a technique developed for use in The Witness, which focuses on reducing the noise in the final image's shadows using Gaussian-weighted sampling. [9]

Shadow maps often can't be rendered at a high enough resolution to avoid certain problems. The already mentioned pixelated edges are among other problems such as shadow acne. Shadow acne is caused by the shadow map having a single pixel that needs to be used when determining if multiple different points are in shadow.

Raytraced shadows

Most problems with shadow maps would be solved by path tracing. As mentioned, real time path tracing is not a viable solution for many applications. However, only using raytracing for the shadows is more achievable than path tracing.

The method combines raytracing with traditional rasterization and uses raytracing to determine the visibility of light sources to a point being rendered. For point lights a single ray can be used to check visibility and for area lights multiple rays need to be used to determine how much of the light source is visible to the point. These checks are made in real time by storing old results of the checks when possible and by determining the number of samples that need to be taken depending on the situation. [10]

Screen-space shadows

Screen-space shadows is a method for rendering shadows that avoids the issues with shadow maps by not rendering them. The method works by rendering two z-buffers, one for the fronts of objects and one for backs of objects. [11]

Working in screen-space means that things that are not visible on the screen will not affect the result. This can lead to shadows disappearing when the object casting it leaves the screen. The method proposes rendering the buffer for screen-space shadows from a camera slightly behind the actual camera in order to capture a larger area in the scene and somewhat avoid these issues. [11]

The method is much faster than shadow maps when there are many lights in a scene, because it does not require rendering a buffer for each light. It also handles point lights that emit light in all directions better than shadow maps. It has several drawbacks relating to the screen-space nature of the method, most notably the fact that offscreen occluders are not taken into account, the shadows are not temporally consistent and very thin objects might be missed. Screen-space shadows are often used to complement shadow maps in cases where small details would not be properly occluded with shadow maps [1]. [11]

2.2.3 Reflections

There are multiple different types of real time approximations for reflections. All of them have their own benefits and drawbacks. They also support different types of reflective surfaces.

Figure 2.7. *Screen-space reflections can't display anything not present on the screen. This is visible in Cyberpunk 2077's screen-space reflections, where things disappear from the reflection when they are no longer visible on the screen due to the player looking downwards.*

Screen-space reflections

Screen-space reflections (SSR) are a common technique in many modern video games. They are reflections that are calculated from the rendered image using the z-buffer. Due to the screen-space nature of the effect, it requires small amounts of work when compared to the other techniques for real time reflections.

SSR works by tracing a single ray along the view direction reflected by the surface normal. This ray is stepped forward and by casting its position into the screen-space, the z-buffer can be compared to the ray's current position. If the ray hits the geometry in the z-buffer, we take the color from the rendered image at that position and use that as the color for the reflection at that point in the surface. If the ray is stepped forward in uniform steps, it can easily miss geometry. There are improved methods for this that only sample pixels that are on the line between the ray's start and end positions that have been cast into screen-space. [1]

SSR can provide believable results, but the major drawback is the inability to show reflections of objects that are not on the screen. In the case of water and other planar surfaces

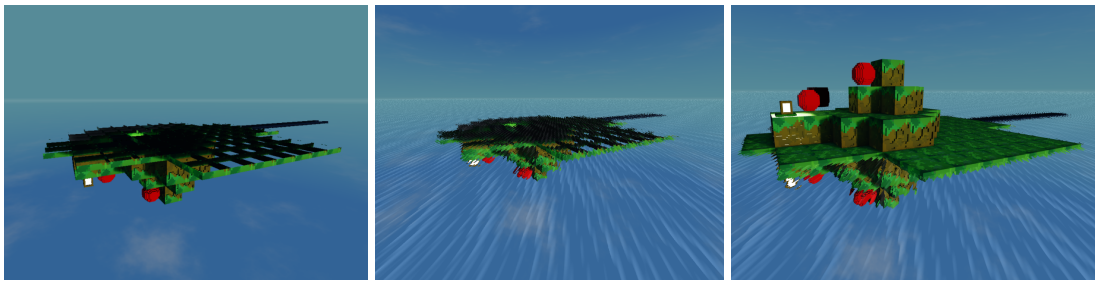

Figure 2.8. *Rendering process of planar reflections. In the leftmost image the scene rendered with the flipped camera, then in the middle image the flipped texture is applied to a plane surface and water ripple effects are added to the image in order to make it look more like water. In the rightmost image the final render with the rest of the scene rendered.*

this is very noticeable, as a tall object's or building's reflection is no longer visible when looking at the plane in a way that the building is not on the screen. In cases where the reflections are important for gameplay, SSR is rarely reliable enough to depend on. Figure 2.7 displays screenshots demonstrating this behavior with screen-space reflections from the game Cyberpunk 2077 [12] by CD Projekt RED.

Planar reflections

Planar reflections are suited for planes with reflective surfaces, most commonly water. Planar reflections can be rendered in real time by rendering almost the whole scene a second time from a different perspective. Their biggest advantage is being accurate for mirror reflections.

Planar reflections work by rendering the reflection into a buffer first and then using that buffer when rendering the reflective surface. The reflection can be rendered by flipping the camera's position and target around the reflective plane and rendering the scene using this new camera. Figure 2.8 demonstrates this process. [1]

The sharp reflections provided by planar reflections provide more accurate and reliable reflections than screen-space reflections, being able to reflect objects that are currently not on the screen. This is important when the reflections are an important part of the visuals of the game or even serve gameplay purposes. The planar reflection texture can be warped with post-processing effects to, for example, add wave ripples to the reflection texture in order to imitate water better.

Planar reflections only work when there are few reflective surfaces, since every different surface requires rendering the whole scene again. The technique also only works with planar surfaces, so it can't be used with any surfaces that aren't just even planes.

Figure 2.9. *Raytraced reflections in Cyberpunk 2077. Contrary to screen-space reflections, raytraced reflections correctly reflect the environment even when the environment is not visible on the screen, they are also sharper.*

Raytraced reflections

Raytraced reflections have the accuracy of planar reflections with the ability to work with diverse surfaces like screen-space reflections. As mentioned before, raytracing has major performance costs.

A method for raytraced reflections used in DICE's Battlefield V works by adding steps to the raytracing pipeline that improve its performance significantly. The amount of rays used is variable based on how much of an effect the ray would have on the surface's shading. This is surface dependent and also dependent on the view angle. The results are also denoised using data from nearby pixels as well as temporal data. [13]

The raytraced reflections are also hybridized with screen-space reflections. This gives the reflections the ability to show things such as bullet hole decals that are not visible in the raytracing scene without additional work but are visible on the screen. The SSR hybridization is also used to avoid tracing rays in cases where SSR would end up with an acceptable result. [13]

Raytraced reflections are more flexible than planar reflections and more accurate than screen-space reflections, but they are more complicated to implement and need more

performance, although in some cases rendering the scene twice for planar reflections would require more performance. Figure 2.9 displays raytraced reflections in Cyberpunk 2077 [12].

2.3 Pre-calculated approximations

Pre-calculating and storing lighting data for a scene enables the use of high-quality path tracing to calculate the stored data. Since the pre-calculating step has time requirements measured in hours or days instead of milliseconds for computing the lighting, the problem becomes mostly about storing the data. The biggest drawback of pre-calculating data is that it is calculated beforehand and can't react to changes in the environment. This is often an issue in video games, as scenes can change often, and objects and characters move around. In video games these pre-calculated techniques are often used only on static objects and then dynamic objects use some of the approximations

2.3.1 Light mapping

Lightmaps are pre-generated lighting textures that are used to store pre-calculated information about the lighting of a scene. They are usually very small textures, a single texel covers around 20cm x 20cm of the scene even in high quality applications. [1]

Lightmaps have downsides, such as the inability to react to changes in the scene and the maps can also have visible seams when the mesh's surface is covered by multiple light maps, although the seams can be mitigated by a number of different techniques. [1]

Lightmaps do not solve the problem of indirect lighting for objects that move within the scene, as these objects or characters can't have light maps calculated due to the fact that they are not stationary and the lighting around them is changing constantly.

Another problem light maps do not solve is high resolution specular global illumination, or more simply sharp reflections. As reflections depend on the viewing angle, light maps would have to store incoming light for a point from every direction, causing storage issues.

2.3.2 Light probes

Light probes present a solution to global illumination that also is able to accommodate objects that are not static to the scene, as long as the lighting stays static. These light probes can be manually or automatically placed in the scene.

Light probes are points in space that have stored information about how much light is coming towards the point from all directions. This information can be used to light static and dynamic objects in the scene by taking the information from nearby probes and using

it to calculate the indirect lighting that the object would receive.

Light probes are often used to store diffuse indirect lighting, but it's also possible to use light probes to store specular indirect lighting fairly accurately, especially when the probe is in the center of a small object. [1]

Cubemapping

A cubemap is an image that represents the environment rendered from the point of view of a point to all 6 directions of the cube's faces. This mapping can be used as a light probe, when only storing incoming indirect light into the cubemap. These cubemaps can be sampled in order to get lighting information about the lighting data at the location of the probe.

Cubemaps as light probes can be also used to store indirect specular lighting data. If the cubemap's sides store precise images of the environment it can be used for sharp reflections for glossy materials, although in these cases the reflections can stretch unnaturally and be imprecise due to the probes not matching the geometry and not being in the center of the object. [1]

Spherical harmonics

Spherical harmonics are mathematical functions that are used widely in different areas such as physics and quantum chemistry. In real time computer graphics, they are often used as an efficient way to store spherical functions such as the directions and intensities of light towards a point. [14]

They are a more efficient way to store lighting data for light probes. Instead of storing a cubemap of the lighting at the point of the probe, a spherical harmonic can be stored. It takes up less space allowing for a higher density of probes. Spherical harmonics are commonly used to only store low resolution data about incoming light, not usable for sharp reflections.

2.4 Scene representations

The scene is the camera, lights and a collection of objects of varying shapes and sizes that are being rendered. The objects need to be represented in memory in order to be rendered. The scene data often also contains textures and material information for the objects in the scene.

2.4.1 Triangle meshes

Triangle meshes are the most common way of storing the objects within the scene. GPUs are very good at drawing very large amounts of triangles efficiently. A triangle mesh is just a collection of triangles forming an object. If there are enough triangles and the triangles are small enough, individual triangles can't be noticed from an object.

In rasterization, each triangle is drawn to the screen at a position determined by transforming the triangle to the camera space. Then all of the pixels that contain the triangle are shaded according to the triangle's details, unless there is already something drawn on the screen that is closer to the camera.

In path-tracing, light rays are traced through the scene and triangles are checked for collisions with the ray. If the triangle is the first thing the ray collides with, the ray's next direction and its attenuation are modified based on the triangle's details and its material.

2.4.2 Acceleration structures

Checking if a light ray intersects a triangle in the scene for millions of triangles would be very bad for performance. That is why an acceleration structure is made from the triangles. Bounding Volume Hierarchies (BVH) is an example of an acceleration structure, where checking the ray against millions of triangles is avoided by grouping the triangles into groups that are also grouped into larger groups and so on. This way the ray only needs to be first checked against a few larger groups and then only the groups inside the group until the ray is finally compared against a relatively small number of triangles. [3]

Accelerations structures are often divided into top level acceleration structures (TLAS) and bottom level acceleration structures (BLAS). A BLAS is the acceleration structure for a single mesh, with its own coordinate system, while a TLAS is a collection of BLASes with transform information that can be used to rotate, scale and move a BLAS instance around the scene.

2.4.3 Voxels

Voxels are small, usually cube shaped, elements that can be composed into a three-dimensional scene. They are the three-dimensional counterpart of pixels. Voxels can be stored in a simple 3D grid, where a single cell in the grid stores information about what type of voxel is in the grid. Voxels are often generated based on triangle meshes in order to simplify the meshes for many operations. Voxels can also be sole scene representation. This kind of approach has been popularized with games like Minecraft, where the world consisting entirely of voxels enables the player to edit the environment. [1]

When path-tracing voxels, there are multiple different ways to store the voxels for checking if a ray hit a voxel in the scene. A mesh can also be generated for every voxel type. These meshes can be combined into a larger mesh that forms the entire scene and that can be formed into a BLAS for the intersection checking. The voxels could also be stored in some voxel specific acceleration structure like an octree. The voxel meshes can also be individually made into BLASes and then those structures can be instanced and composed into a TLAS where every instance of a voxel appearing in the world is added as an instance of the voxels BLAS with a transform that moves it to its position in the world. For small scenes, instancing is found to be most efficient and also allows more flexibility by allowing the voxels to not be perfect cubes, allowing for voxels with rounded corners or other additional detail [15].

The instancing approach allows for the voxels to not be perfect cubes. They actually don't really even need to be even close to being a perfect cube. Polytron Corporation, makers of a 2012 game called Fez, coined the terms "triles" and "trixels" to refer to blocks consisting of smaller 16 x 16 x 16 small voxels and the small voxels the blocks consist of [16]. This was done to bring more depth and detail into a game where the world consists of blocks when compared with a game such as Minecraft. Acceleration structures can be made for these voxel blocks or "triles" and then these acceleration structures are composed into the full scene. This approach allows blocks to contain more detail while not increasing the resolution of the voxel grid. The same generated meshes for voxel blocks that are used for the acceleration structures can be also rendered in real time using instancing.

The techniques used for general purpose real time rendering and shading are suitable for voxels. This is because a voxel scene, when turned into triangles, is just a simpler triangle mesh scene. However, the restrictions of voxels can be used to do things that are not possible when rendering a generic triangle scene without knowledge of how the triangles are placed. Voxels can also make things more efficient, since their restrictions can enable optimizations. In the case of this thesis, the most significant restrictions are the even placement of blocks on a grid and the knowledge that all surface normals will be in one of the six axis-aligned directions. Some techniques, such as voxel cone tracing, voxelize triangle meshes in order to do operations on voxels instead of triangle meshes and take advantage of the restrictions of voxels [17].

3. RHOMBIC DODECAHEDRON MAPPING FOR LIGHT PROBES

In the context of this work, a scene consists of voxel blocks that consist of 16 x 16 x 16 smaller voxels. The scene is a grid of slots for voxel blocks or empty space. These blocks can only be rotated at 90-degree angles, so that all surfaces are along the axis of the grid. These blocks are large enough that a single block forms a substantial portion of the scene. The chosen size for voxel blocks is due to visual reasons and the level of detail making the blocks feel easy to edit. The division of the block into these small voxels could be entirely different without affecting the functionality of the mapping, although surfaces that are further from the block's sides have less accurate data and with smaller voxels there is more control when it comes to the depth of the surface.

This grid of blocks provides a very natural framework for placing light probes in the scene. If a cubemap light probe or a spherical harmonic light probe was placed in the center of each block and the lighting data was calculated without the block being present, it would be easy to store incoming light to the block. This approach runs into a problem, especially with specular lighting and sharp reflections. If there are blocks on the sides of the block we are calculating the light probe for, the blocks block light to the center of the block, even though they would not block light from reaching the surface of the block. This is demonstrated in figure 3.1.

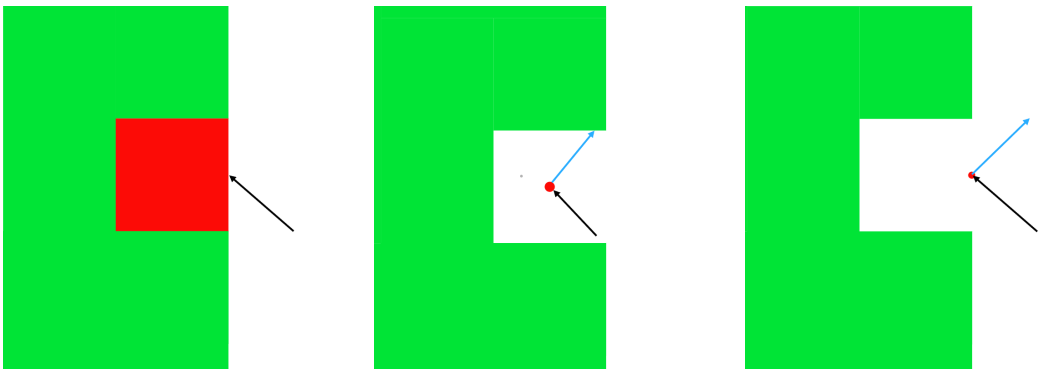

Figure 3.1. *In the leftmost image incoming lighting data is needed for the red block when viewed from the direction of the arrow. The center image shows the result if the reflection would be sampled from a light probe in the center of the block and the rightmost image shows the result being correct with a light probe located at the side of the block.*

With voxel blocks, all normal vectors of the surfaces point in 6 directions and the block only has 6 sides. This enables us to store the incoming light from the visible hemisphere of the center of every side of the block. This storage method isn't affected by blocks around the probe in the same way and results in more accurate values for incoming light to the surface.

This mapping would be even more suitable for a system where the voxel block is just a large textured cube with even sides. The fact that the voxel block consists of smaller cubes and can have depth differences on the sides, makes the stored light less accurate, since a surface further inside of the block is further from the surface where the hemisphere is stored and also the stored hemisphere is not accounting for the surface being occluded by the other small voxels in the same block.

Although the focus of the rhombic dodecahedron mapping is storing indirect light, it can be used as a general-purpose light probe for storing lighting data for a block or cube. The implementation of this thesis uses it also for direct lighting from light emitting voxels.

3.1 Storing incoming light into the rhombic dodecahedron mapping

The rhombic dodecahedron mapping (RDM) is a mapping where we store a hemisphere aligned with the surface normal for each side of the voxel block. Storing the hemisphere can be done in multiple different ways, but the RDM uses a hemi-oct encoding presented by Cigolle et al in 2014 to store a single hemisphere [18]. The hemi-oct encoding maps the quadrants of a hemisphere to the sides of a pyramid. This is visualized in figure 3.2. The name comes from the oct mapping from the same article that mapped the entire sphere to an octahedron. In the case of the RDM, only the hemisphere for each side needs to be mapped resulting in the hemi-oct encoding being used.

The oct mapping has both hemispheres in it. One mapped hemisphere square is tilted, so that its corners hit the centers of the whole square's sides. The rest of the space is used for the second hemisphere. When using the hemi-oct mapping, the inside square is rotated so it fills up the whole mapping square like in figure 3.3. The resulting stored hemisphere image looks like figure 3.4. [18]

One of these pyramid mapped hemispheres is stored for each side of the block for the center point of the side. The mappings for each side can be seen in figure 3.5. When they are combined into a uniform shape the shape formed is a rhombic dodecahedron, giving the name for the mapping. This shape is stored as 6 of these hemisphere squares in a 2 by 3 arrangement. A complete mapping for a block can be seen in figure 3.6.

When storing information, an index based on the side of the cube for which the hemisphere is being currently calculated for is used to determine the correct hemisphere

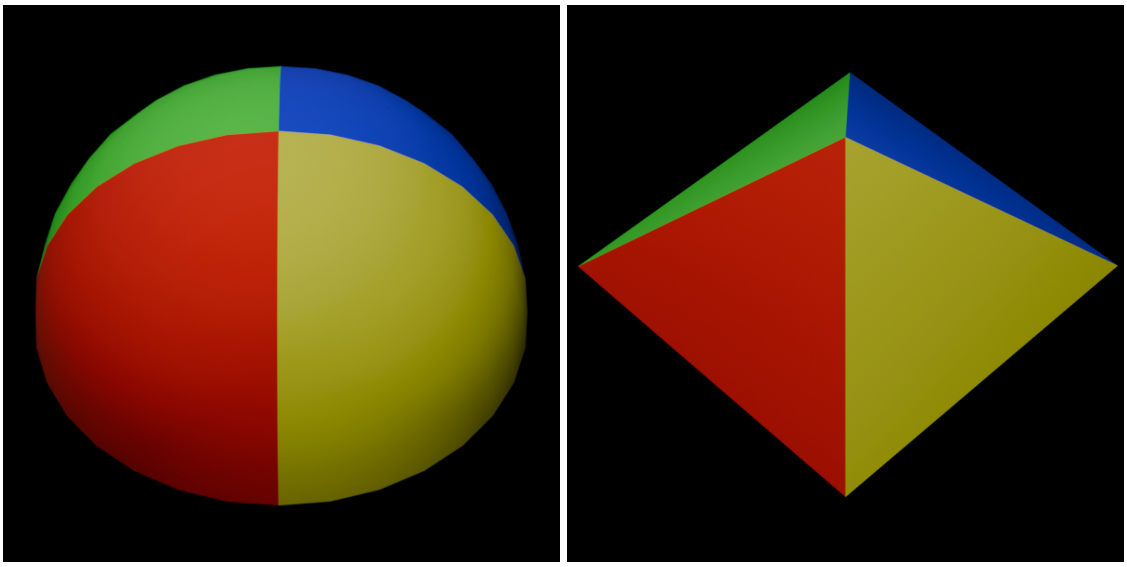

Figure 3.2. *A hemisphere and a pyramid representation of it in the hemi-oct encoding.*

square the lighting data should be stored in. Then the hemisphere square's pixels are looked at, and each pixel's coordinates are transformed into 2D coordinates in the range $0 \ldots 1$. These 2D coordinates are then transformed into a 3D unit vector into the hemisphere with the hemi-oct mapping [18]. This 3D unit vector is then transformed into being in a hemisphere aligned with the normal of the side of the block indicated by the index.

An entire RDM is calculated and stored by going through an empty RDM image with the desired resolution, finding the index based on the section of the image and then finding the unit vector. The value stored in the RDM at that position is calculated based on the unit vector (ray direction), roughness and ray origin indicated by the index. The value for incoming light is stored with the depth of the sample with the lowest depth to the first hit with a surface. The roughness is used to pre-filter the values in the RDM, in order for the values to be used with the split-sum approximation while shading [19, 1].

3.2 Reading from the rhombic dodecahedron mapping

When the scene geometry is constrained into being formed from axis-aligned cubes, we can transform the surface normal into an index indicating which of the 6 hemispheres should be used. Then, when shading the surface, we can sample the single hemisphere for the incoming light.

Diffuse lighting can be fetched by sampling the hemisphere with the maximum roughness with the surface normal. However, having only 1 diffuse lighting value for a side of the block results in low resolution diffuse lighting, but diffuse lighting is not the focus of the RDM, it's more of a convenient side effect of RDMs, when there isn't a need for higher resolution diffuse lighting by using a more advanced or appropriate technique for it.

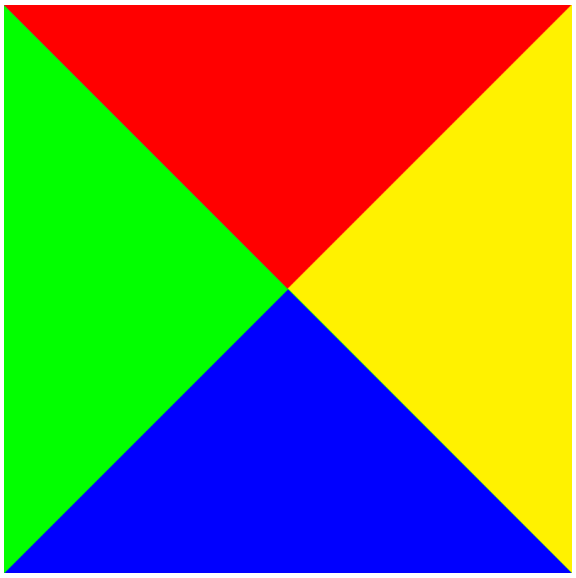

Figure 3.3. *The pyramid flattened into a square.*

If the point being shaded was actually in the center of the side of the cube, like the point for which the hemisphere in the RDM has been calculated for, the sampling would be as easy as picking the correct hemisphere with the index gotten from the surface normal, and sampling it in the reflection direction. This approach, when used for points not in the center, causes things near the block to look extremely large in the reflection.

To combat this problem, the sampling can be done with an approach that resembles the way screen-space reflections are done, where we use the depth data stored in the mapping to find the correct direction to sample the hemisphere. Figure 3.7 displays the problem of sampling the hemisphere, where the length of the blue arrow (the reflection direction) determines the direction of the red arrow (the sampling direction). Determining the length of the reflection vector also determines the sampling direction. The depth gotten from the RDM for the sampling direction can be compared to the length of the sampling direction vector, marked with d in figure 3.7. The length of the reflection vector can be approximated by stepping it forward until the sampling vector's depth is larger than the depth value stored in the mapping at the sampling direction. Then that value can be used as the result for the sample. This process is demonstrated in figure 3.8. The results are not always correct, since the data stored for the hemisphere at the center of the side can't be used to fully recreate the data for a hemisphere at the other point. Figure 3.9 displays a case where an object is visible from the point being shaded but is not visible from the center of the side. This results in incorrect data being used for the reflection. However, the results are fairly close in most cases and yield believable results.

Sampling of the hemisphere is done by first transforming the sampling direction into a unit vector in a common hemisphere where the positive Z direction is up. This is done by using the index and is a matching operation to when the hemisphere was stored. Then

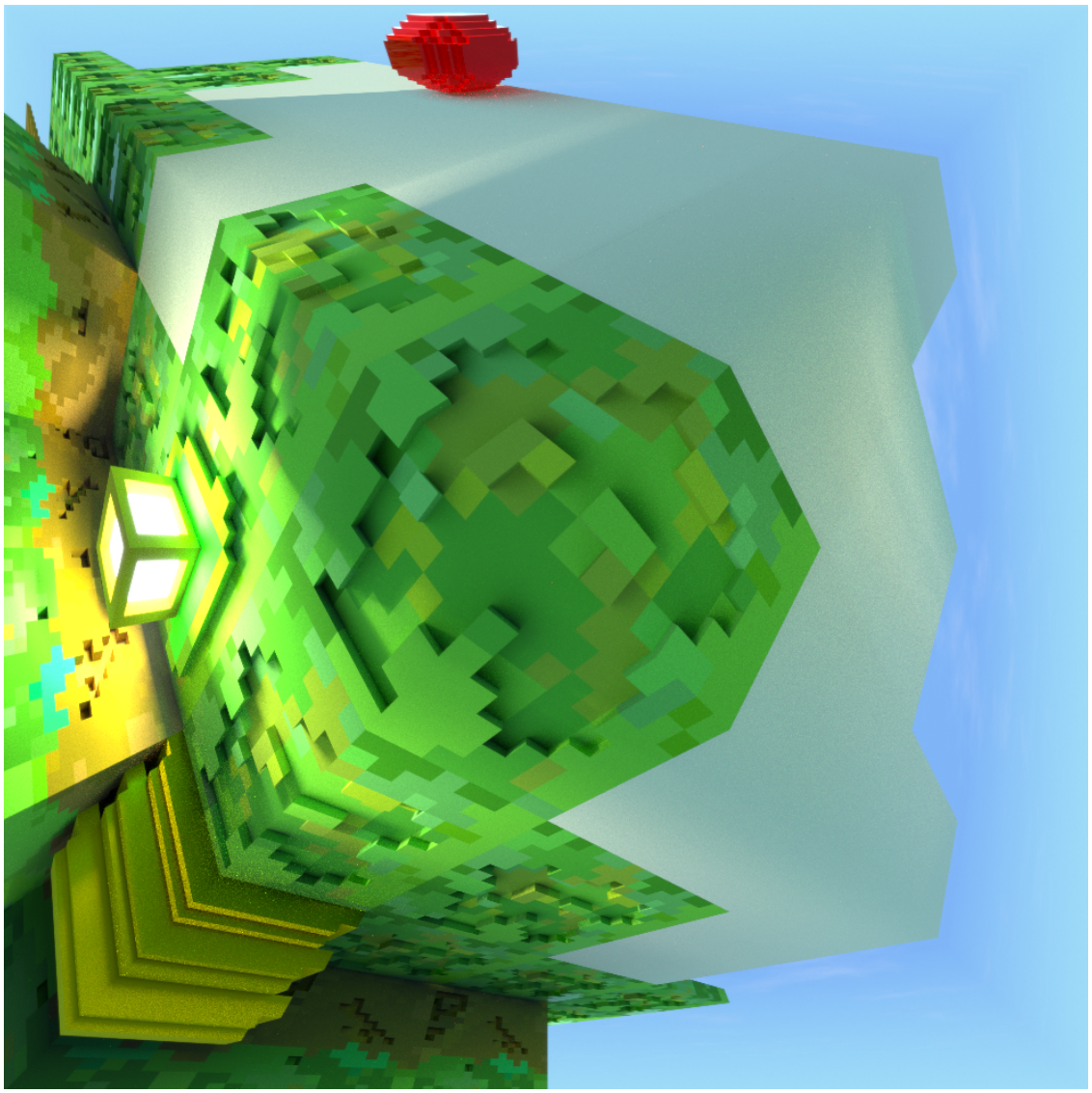

Figure 3.4. *A hemisphere image with the hemi-oct encoding. The hemisphere is from the example scene and is pointing down towards the ground.*

this vector is used to sample the correct hemi-oct mapping, determined by the index, to get the depth and light values stored in the mapping at that direction [18].

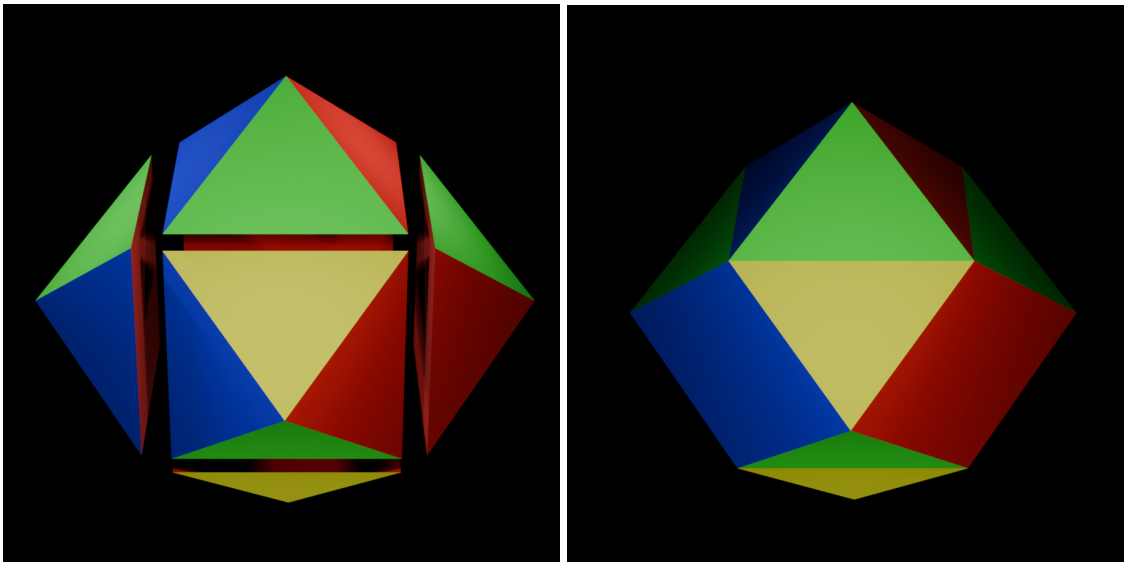

Figure 3.5. *Hemisphere pyramids arranged for each side of the block and then combined into an uniform shape.*

Figure 3.6. *A complete RDM with the hemisphere squares for every side of the block. This is a RDM created with the minimum roughness value, and it stores the incoming light at a high resolution for usage with sharp reflections.*

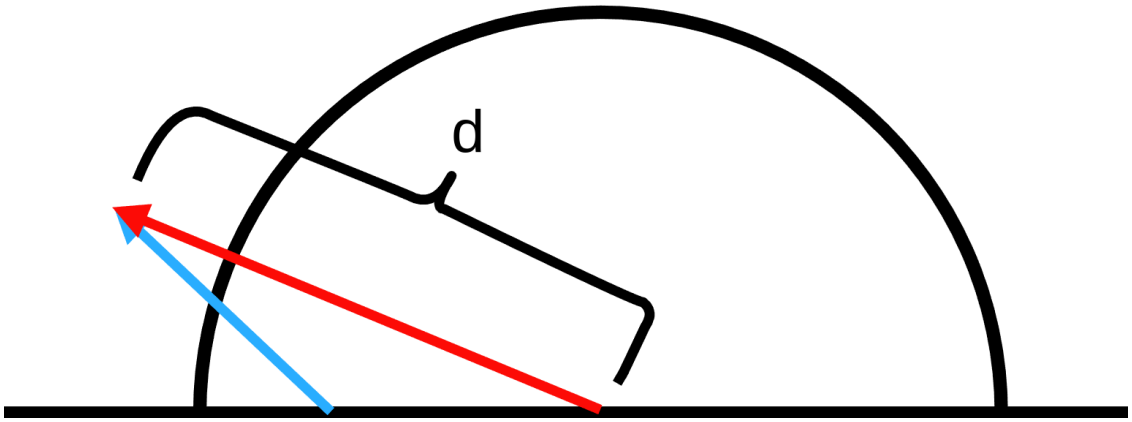

Figure 3.7. *Sampling the hemisphere located at the center of the side of the cube for a point that is not at the center, requires knowledge of how far the surface being shown on the reflection is. The length of the sampling direction vector, marked with d, can be compared with the depth value stored in the mapping. The blue arrow marks the reflection direction, and the red arrow marks the direction the hemisphere is being sampled in.*

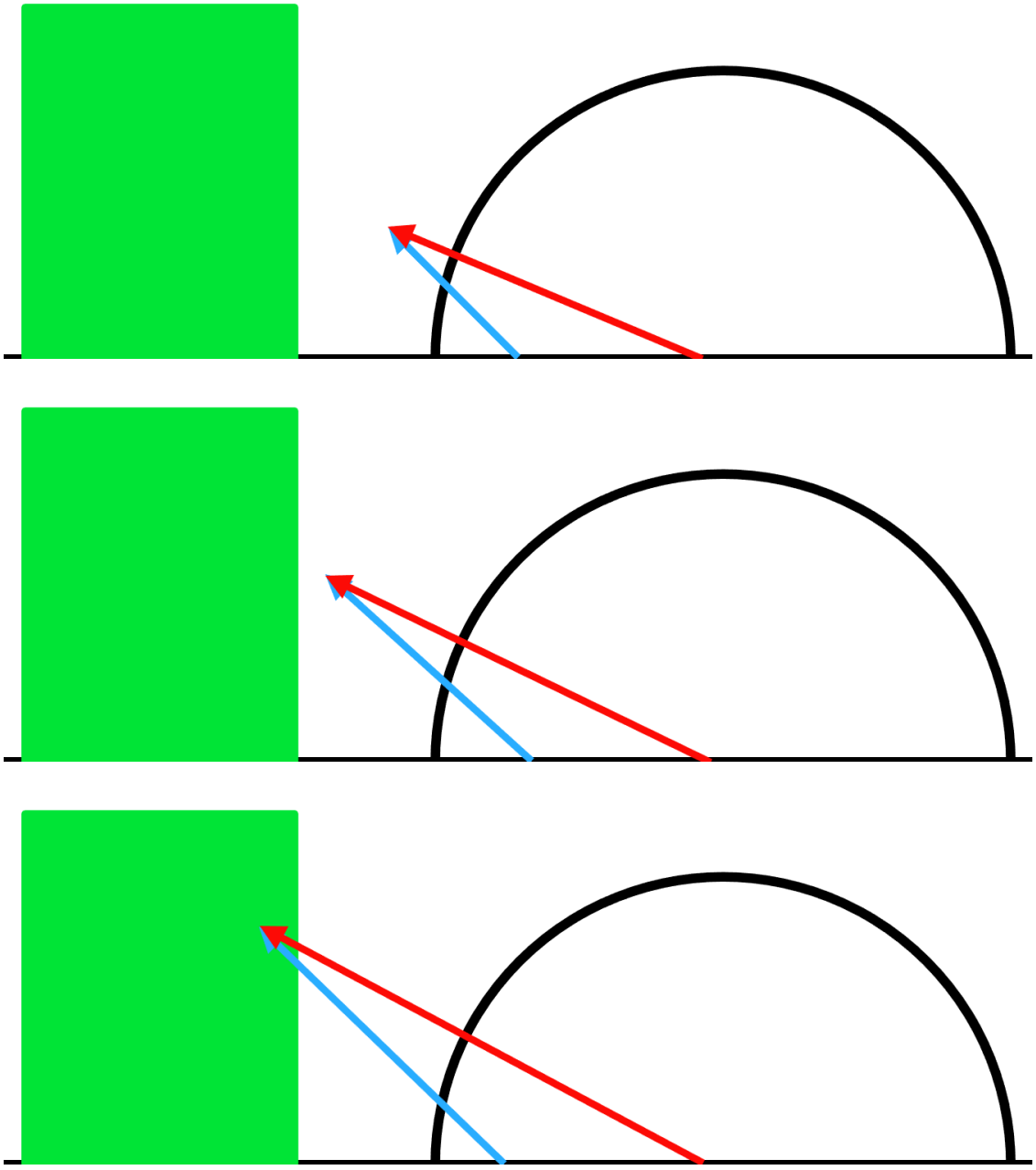

Figure 3.8. *The process of stepping the length of the reflection vector higher in order to find a point where the reflection vector hits something. This can be determined by comparing the length of the red arrow (sampling direction) to the stored depth value. In the final image, the depth value stored will be less than the length, so the lighting information at that direction of the hemisphere will be used. As the blue line gets longer, the angle of the sampling direction gets larger and approaches the reflection direction. For situations where the block is carved in a way that results in the blue line starting from underneath the hemisphere, the blue line is extended to be long enough for the sampling direction to be within the hemisphere.*

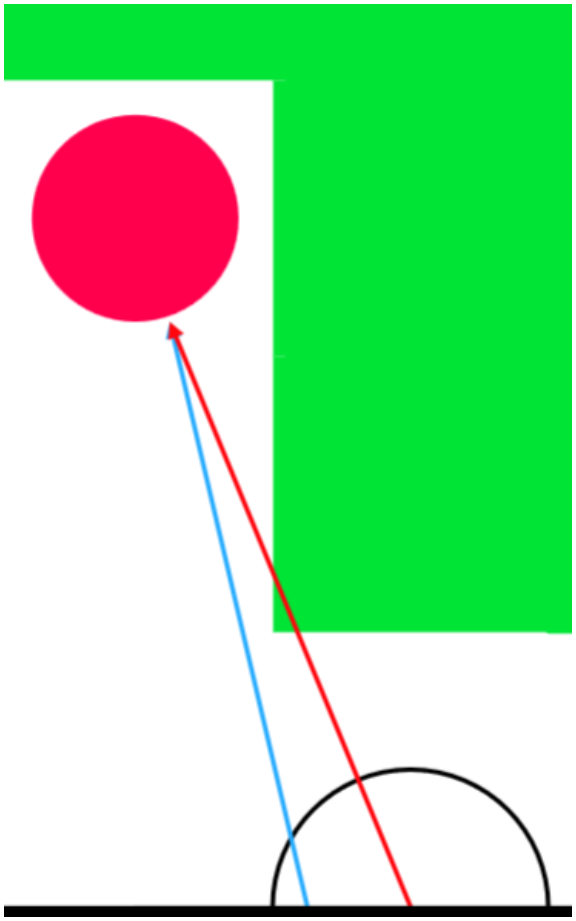

Figure 3.9. *The right side of the red ball is visible from the point that is being shaded but not visible from the center of the stored hemisphere in the center of the block's side due to other blocks blocking the view. It's not possible to display the correct reflection for the point based on the data stored in the hemisphere.*

4. IMPLEMENTATION

During this work, multiple components were implemented in order to test RDMs and measure how close they are to fully raytraced images. The components that form the entire pipeline for generating and using RDMs are a path tracer, a basic game engine and a simple packing program that packs the generated RDMs into a single atlas that can be loaded in the game engine.

4.1 Voxel blocks

The previously mentioned 2012 game called Fez, served as an inspiration for the approach to voxels in this implementation. In Fez, the game world consists of blocks of voxels, where each block consists of 16 x 16 x 16 small voxels [16]. These blocks are made in an editor, and they serve as three-dimensional tiles from which the scene is formed. In figure 4.1, there is a basic scene which is clearly built from blocks of small voxels. The ability to carve depth detail into the block by removing small voxels allows for better representation of objects that are not cube shaped or are smaller than the default voxel block. Additionally carving depth details into voxel blocks can improve the overall look of the game, especially with good lighting.

4.1.1 Voxel mesh generation

Even though voxel blocks are edited and stored as a three-dimensional array of small cubes, when rendered in the path tracer or in the game engine, the small cubes are not being actually rendered. Rendering a single block with even sides where all 4096 small voxels are in place, would result in rendering $4096 \cdot 2 \cdot 6 = 49152$ triangles, since each small voxel has 6 sides and rendering a single square side of a voxel takes 2 triangles.

Instead, the surface of the voxel block is combined into a set of triangles, so that only the visible sides of the small voxels are rendered, and the surfaces of the small voxels are combined into larger squares and then turned into triangles. The example above would only require 2 triangles for each side of the full voxel block, resulting in only 12 triangles being rendered. This set of triangles for the outside surface of the voxel block is called its mesh. Figure 4.2 displays the differences in triangle count between rendering the small

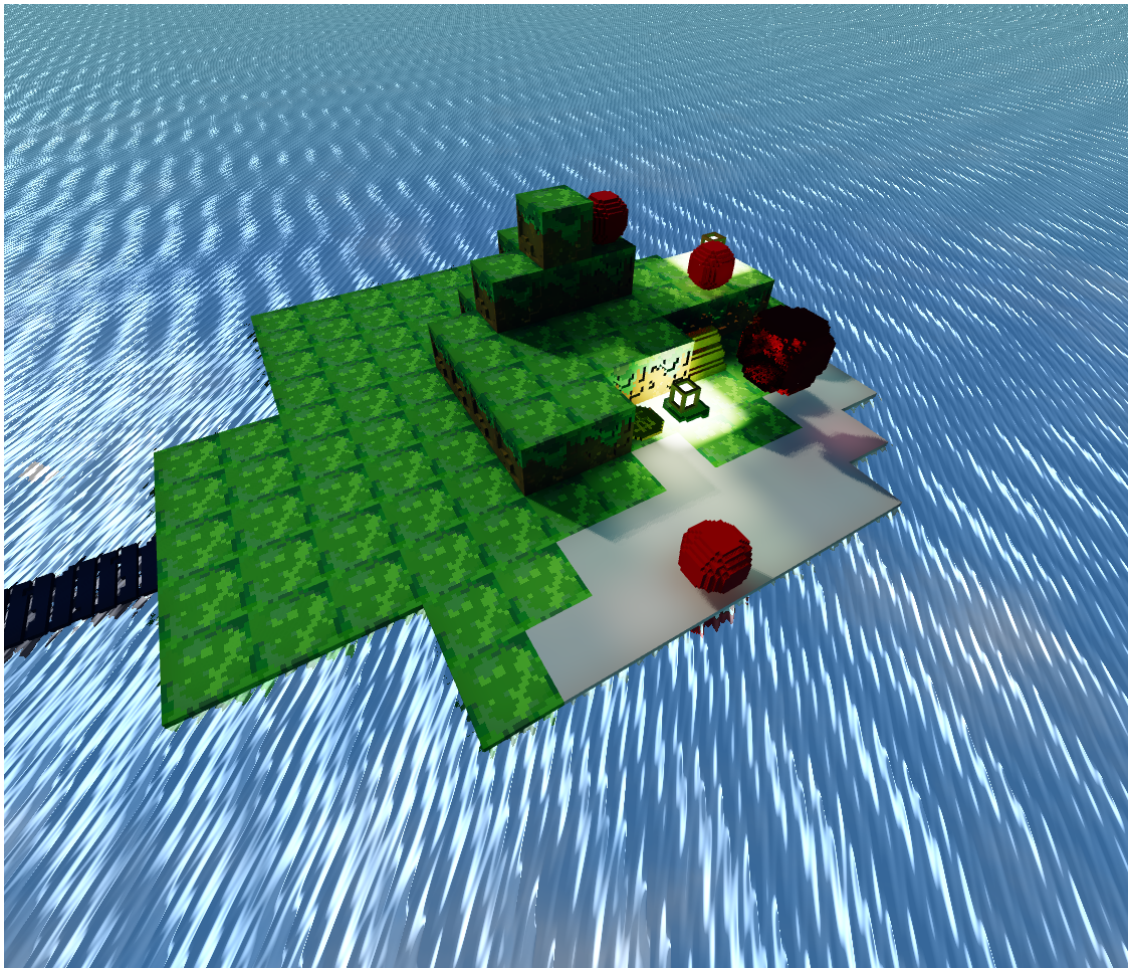

Figure 4.1. *A simple scene formed from voxel blocks. The repeating block elements are formed of small voxels.*

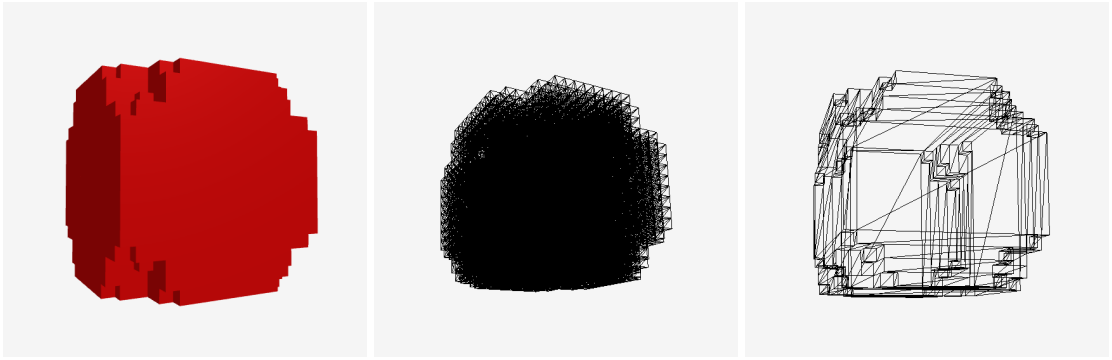

Figure 4.2. *Visualization of differences in triangle count between a mesh generated from the surface and rendering all of the small voxels. On the left, the voxel block, in the middle the triangles of all of the small voxel blocks and on the right the triangles of a mesh generated from the surface.*

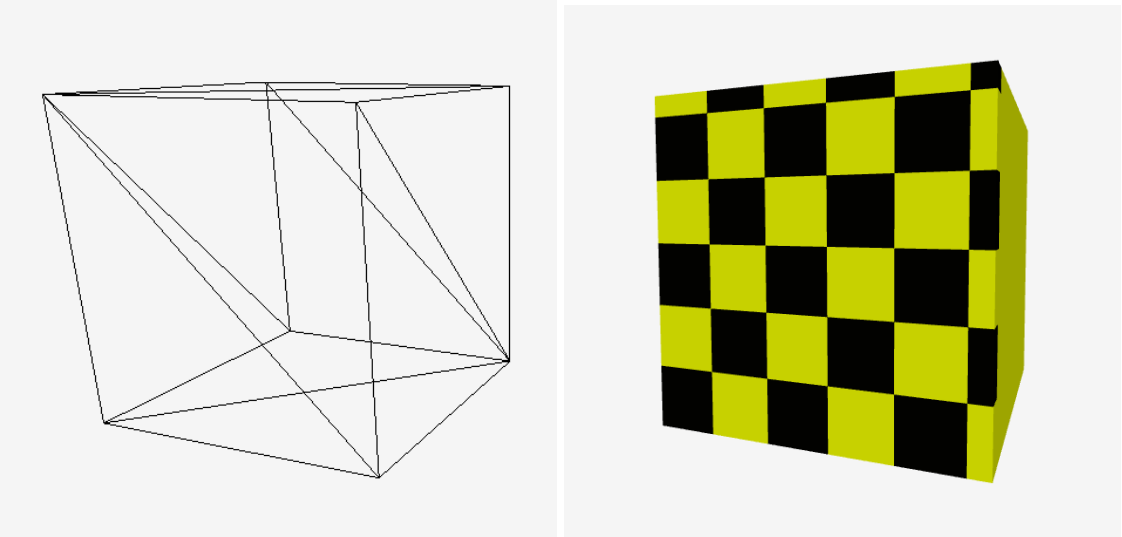

Figure 4.3. *Multiple different colors present inside a single triangle. As seen on the left image, the actual triangles of the mesh do not correlate with the small voxels, that determine the material for a point.*

voxels and the generated mesh, and how rendering the generated mesh is preferable.

This approach of combining the small voxels into a mesh was used in Fez and their approach to generating the mesh is detailed in a series of blog posts, although the mesh generation and storage detailed by them is more advanced and better optimized than the current basic implementation [20].

4.1.2 Finding the material for a point in the mesh

Fez had a texture for each side of the block and the color for each side of a small voxel was taken from the voxel's position in the 2D texture for the side [16]. Unlike Fez, this implementation's voxel blocks don't have textures, but each small voxel has a color and a material. This approach offers more flexibility in some cases where 2 voxels are in the same place when viewed from the side but should have different colors. It also ensures that every small voxel is consistently the same color, and its color does not depend on which side you are viewing it from. This can also be viewed as a negative effect depending on the use case.

However, this approach to coloring the small voxels based on 3D information runs into issues when the small voxels are combined into a mesh. As is visible in figure 4.3, surfaces no longer are associated with a single small voxel, which makes it harder to find out the correct color for a point on the surface. For example, with an even cube, a single triangle can have many different colors inside of it. The color for a single small voxel inside of the block can be easily fetched, but that requires knowledge of which small voxel the current point in the surface belongs to.

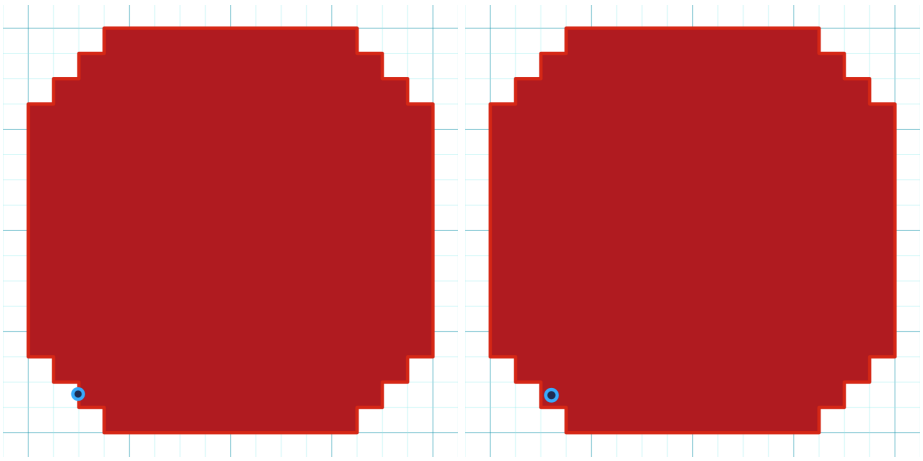

Figure 4.4. *2D image demonstrating how moving the sampling point by subtracting the normal multiplied with a small multiplier from the point moves it clearly into a small voxel position. The first image is the original point and the second image is the point after the operation.*

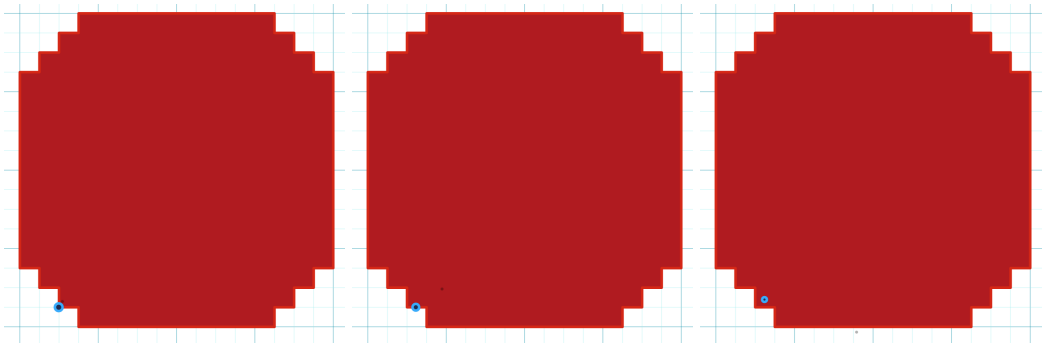

Figure 4.5. *2D image showing how the approach described above fails for cases where the point is in the corner of the surface and how moving the point towards the triangle's center solves the issue.*

Finding out to which voxel block a pixel on the surface belongs to is surprisingly problematic. This is mostly due to the surface always being between small voxel positions. As can be seen in figure 4.4, it's not clear whether the blue point is inside the red small voxel or the empty slot. This can be remedied by moving the point in the opposite direction of the surface's normal, moving it clearly inside the voxel that is supposed to be sampled.

A special case where the point is in a corner of the surface requires more than the above-mentioned case. In figure 4.5, the point on the corner is still a part of one of the surfaces, and when moved opposite to the direction of the surfaces normal, is still between small voxel positions. This can be remedied by moving the point also slightly towards the triangle's center, which gets the point clearly inside a small voxels position that we can sample from. In practice, having a special null value for the materials of empty small voxel positions helps, by allowing checks whether sampling a position would result in being on the wrong side of the surface.

Figure 4.6. *The material information of each small voxel is stored in a texture. This texture stores this data for a red voxel ball. 3 channels are used for color and 1 channel is used for the material data. The data is stored in vertical slices of the voxel block, where empty voxels have a special value for the material and color.*

Material data for a voxel block is stored in a texture. Figure 4.6 displays an example of one such texture for a red voxel ball. The material of a single small voxel is stored in the texture in a single 4 channel color, where there are 8 bits per channel. 3 of these channels are used for the red, green and blue values for the color of the material, and the last channel's bits are used to store the roughness, metallic and light emission values for the material. There are 8 different values for the roughness and 4 different values for both light emission and metallic properties of the material. These values are mapped to the $0 \ldots 1$ range in rendering.

4.2 Pre-calculation

The pre-calculation steps result in a single large image that has the RDMs of the scene called an RDM atlas and also a file that contains information about where each RDM is stored in the atlas. The current implementation stores the RDMs of the whole scene in the image but could be easily modified to store them in one image per section of the scene.

The path tracer is used for calculating the RDM for a voxel block and also used for rendering the reference images. The path tracer is a C++ program using the Vulkan graphics API and a shader written in the GLSL shading language. All of the path tracing work is done on the GPU in order to leverage the raytracing hardware in modern GPUs to render images faster than rendering them on the CPU would be. Another advantage of doing hardware raytracing on GPUs is that the GPU and graphics API handle the ray intersection checking instead of having to write code to do this on the CPU.

When rendering the RDMs for a scene, the path tracer loads the scene and forms a bottom level acceleration structure (BLAS) from every unique voxel block. Then the BLASes matching these blocks are composed into the top level acceleration structure (TLAS) by adding instances of the block's BLAS in every place where the block exists in the scene. After these are loaded to the GPU, we can use Vulkan's ray query API to check collisions for rays leaving a point in a direction and get information about the voxel block that was hit. This instanced mesh approach has been found effective and performant, especially for use cases where memory isn't a limiting factor [15].

Once the BLASes and TLASes are loaded, the scene is looped trough for each voxel block. The voxel block's materials are checked for which of the 8 possible roughnesses are present within the block. For each roughness present, a RDM is rendered for the block. The maximum roughness RDM is rendered for every block, since it's also used for the diffuse indirect lighting, so it's required for all blocks. The RDMs size is determined based on the roughness, since rougher surfaces need less detail. In the current implementation the width and height of a single hemisphere inside the RDM is determined using the roughness scale $1 - 8$ as $2^{\text{roughness}}$. The resulting RDM is $3 \cdot 2^{\text{roughness}}$ pixels high and $2 \cdot 2^{\text{roughness}}$ pixels wide.

The roughness is also used while path tracing to pre-filter the RDM image. Pre-filtering the image by roughness allows for taking only one sample of the result even with rough surfaces instead of having to sample multiple points in the RDM for a rough surface to determine the incoming light. Rough RDMs are blurry. Figure 4.7 of packed RDMs shows how rougher RDMs are also getting blurrier while getting smaller. [19, 1]

The current path tracer renders a single RDM at once. A layout for the RDM containing the ray directions, ray origins and roughnesses for each pixel in the RDM is uploaded to the shader and the path tracer renders the requested number of samples for that layout resulting in the RDM image. The depths of the first collision of the ray are also stored for later use. This result is saved to a directory created by the path tracer. Then the RDM specific data is unloaded, while scene specific data is kept. Then the path tracer moves on to rendering another RDM from the same block with a different roughness of a RDM for the next block in the scene.

The images from the path tracer are saved in image files containing 3 channels for color

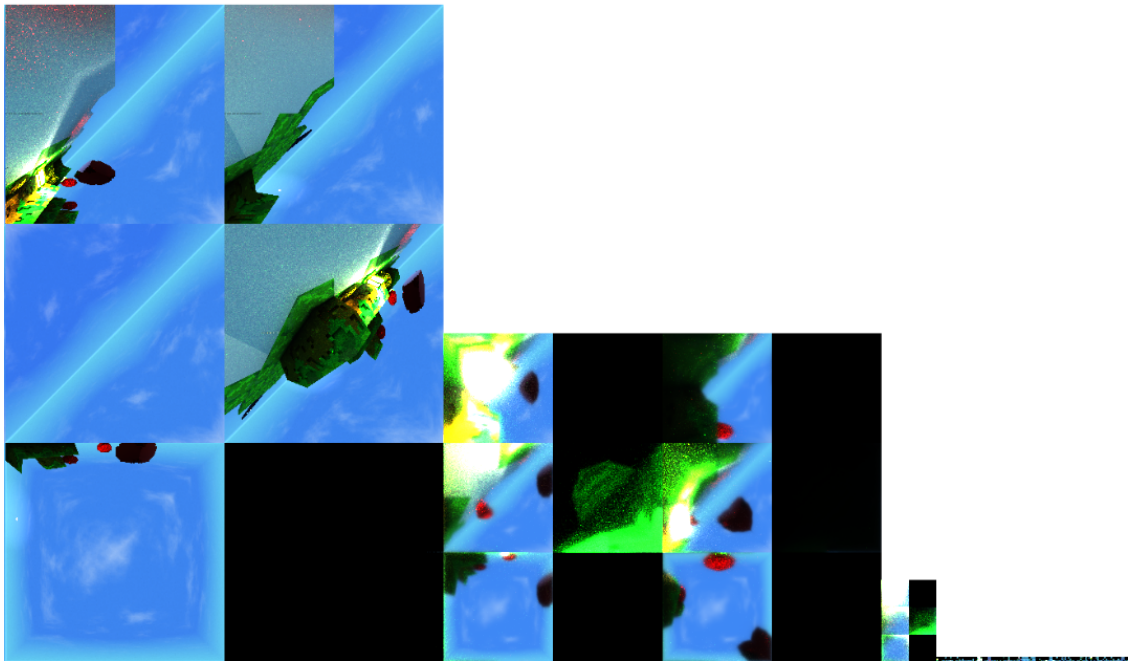

Figure 4.7. *Zoomed in image of a part of the RDM atlas, displaying the large RDM stored next to smaller RDMs. On the bottom right corner, there are the smallest RDMs that are 4 pixels wide and 6 pixels tall. Only the 3 color channels are displayed in this image, discarding the depth channel.*

and 1 channel for the depth. Saving the results in HDR images is important, since they are used for storing lighting. HDR images allow for larger brightnesses than LDR images.

Once all of the images are complete, the next step is forming the atlas out of them. The current implementation's atlas packing is simple, and it results in a lot of needlessly unused space on the atlas. The size of the atlas is determined before the packing is tried and if the packing doesn't succeed a bigger atlas is tried. The images are sorted in descending order by height, and then the images are added from left to right to the atlas until the width of the image is no longer enough for the next image and the next row is started. This packing could be improved massively, especially given the uniform shape of the images. The final packed atlas is saved in the same format as the original images. Figure 4.7 displays a zoomed in image of an atlas with different sizes of RDMs stored next to each other. The wasted space of suboptimal packing of the atlas is also visible.

In addition to the atlas image, a text file is saved, where the first row of the file stores information about the amount of RDMs in the image and the size of the atlas image. After that, each row contains the information about the location of the block the RDM is for, the roughness of the RDM and its location and size on the actual atlas image. This data can be combined with the scene data in the engine while loading the scene.

4.3 In-engine lighting

The pre-calculated data can be combined with real-time lighting techniques in the game engine, in order to try to achieve visuals as close as possible to the path traced reference image. The shader needs access to the RDM information in order to evaluate pre-calculated lighting and then add the real time lighting.

4.3.1 Accessing RDM information from the shader

In order to make use of the stored lighting information, the game engine needs access to the data while shading. The access is easily arranged by uploading the RDM atlas texture to the shader. The shader also needs to know what locations in the atlas are significant for the current point being shaded.

The voxel blocks in the engine are drawn with instancing. That means that a single block is being drawn in multiple positions at the same time. This is done by providing a list of transformation matrices with the block, which tell the shader what position, scale and rotation the particular instance of the block should be drawn at. In addition to the list of transformation matrices, another list of matrices can be provided. The matrices on this list provide information about the positions in the RDM atlas relevant to the current voxel block. The contents of an instance of the matrix are presented below.

$$\begin{bmatrix} x_1 & y_1 & x_2 & y_2 \\ x_3 & y_3 & x_4 & y_4 \\ x_5 & y_5 & x_6 & y_6 \\ x_7 & y_7 & -1 & y_{\text{texture}} \end{bmatrix}$$

In the matrix x_r and y_r are the x and y pixel positions in the RDM atlas texture for the RDM for the voxel block with roughness r in the roughness range $1-8$, although coordinates for roughness 8 are missing from the matrix, since the maximum roughness RDM positions are stored in another texture generated in the engine and uploaded to the shader. The y_{texture} is the pixel row in that texture for the current block. They are stored in the other texture, because the diffuse lighting uses the maximum roughness RDMs and diffuse lighting evaluation also needs access to blocks around the current block.

The lookup texture in figure 4.8 is used to get the locations for the maximum roughness RDMs around the current voxel block. The y_{texture} row number is used to get a row of pixels with coordinate values from the lookup texture. The row contains the coordinates for every maximum roughness RDM near the current voxel block, which means it stores coordinates for $3 \cdot 3 \cdot 3 = 27$ RDMs. The texture is 27 pixels wide, and each pixel uses

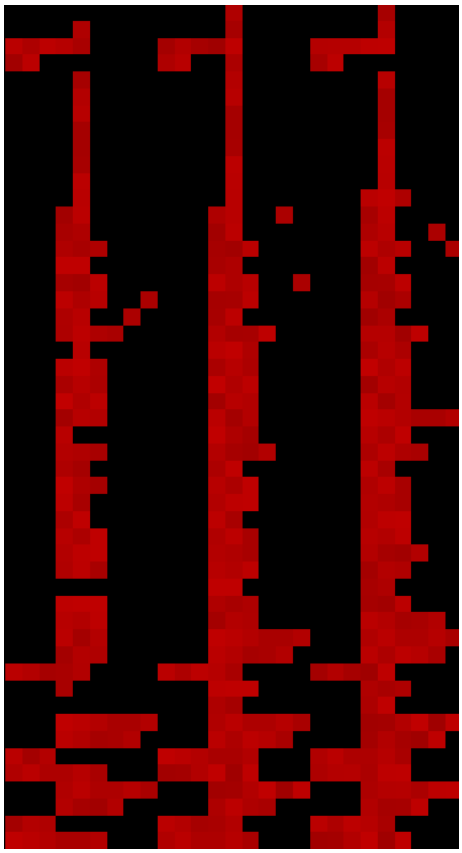

Figure 4.8. *Part of a lookup texture where every row has the positions of the highest roughness RDMs for all the 27 voxel blocks that are around the block, including the block itself. This image's values are displayed so that 0 is black and 5000 is fully colored. The x coordinate is in the red channel and the y coordinate is in the green channel. All of these RDMs are on the first row, so there is no green. -1 value is used for non-existing RDMs.*

2 channels to store the x and y coordinates for the RDM. The middle pixel always has a coordinate value, since it's the location for the block's own maximum roughness RDM.

4.3.2 Shading

With access to the RDM information, the shader can evaluate its impact on lighting for a point. Figure 4.9 displays a high-level overview of lighting evaluation in the implementation. The effects are separately evaluated for diffuse and specular lighting. The RDMs are used for both indirect and direct lighting in the implementation. The implementation's light sources are either the sun, which is evaluated in real time and discussed later, or materials with non-zero light emittance values. The lighting from these light emitting materials is not evaluated at all in real time, but their lighting is stored in the RDMs. The major downside is that only the sun can provide lighting to dynamic objects and the dynamic objects can't cast shadows, and the upside is that the lighting implementation is significantly simplified and can handle complex situations with a voxel block having many different light emitting materials in different places on the block.

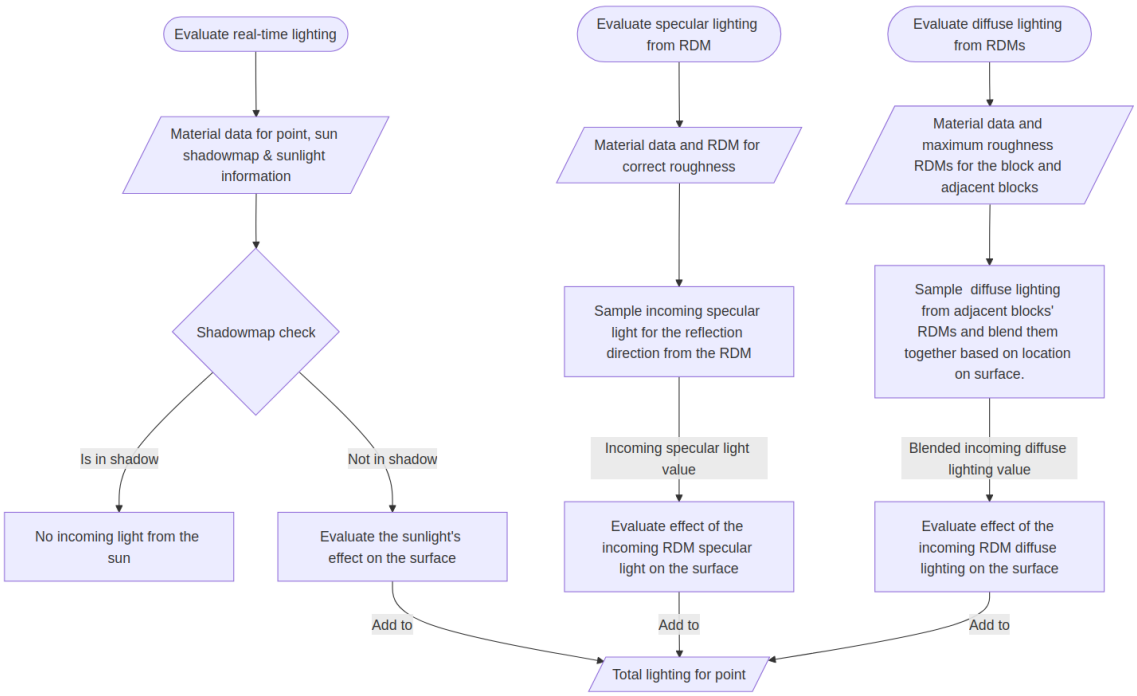

Figure 4.9. *High-level overview of evaluating light for a point.*

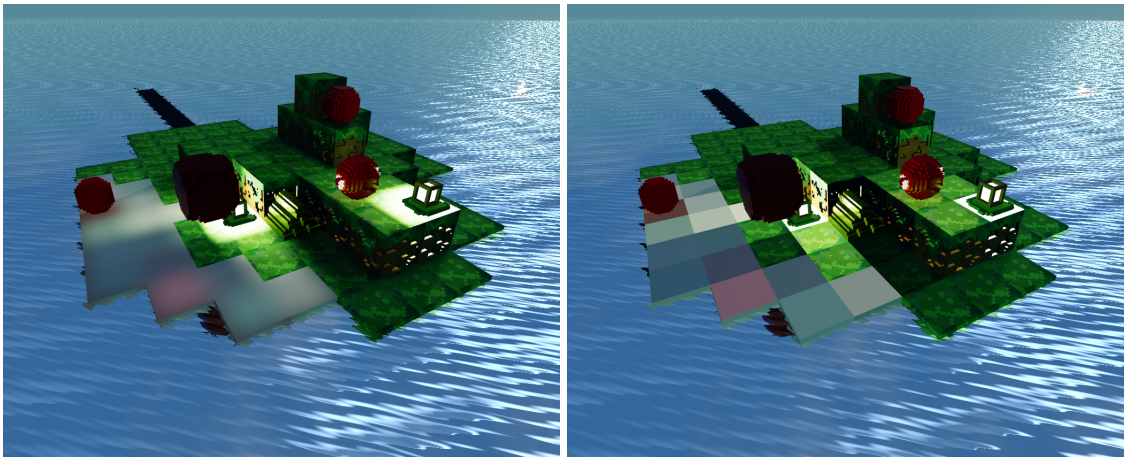

Figure 4.10. *Scene with diffuse lighting with and without blending together nearby RDM diffuse lighting values.*

Diffuse lighting is evaluated by accessing the maximum roughness RDM for the block and the relevant adjacent blocks based on the point's normal and what quadrant of the block's side it's on. The RDMs are sampled in the direction of the point's normal, which means that every hemisphere is being sampled straight upwards. The values are bi-linearly blended together resulting in smooth, but low-resolution diffuse lighting. Where adjacent RDMs do not exist, due to the lack of an adjacent block that a RDM would have been created for, a static ambient light value is used in the blending. Figure 4.10 shows the importance of having access to adjacent RDMs and the effect of blending them together on the rendered image.

Specular lighting can be evaluated from the RDM by sampling the RDM for incoming light from the reflected view direction as specified earlier. This incoming light value is then used in shading the surface with the split-sum approximation method, where the pre-filtered value for the incoming light is combined with values fetched from a pre-calculated lookup texture using the materials roughness and the angle between the view direction and the surfaces normal [19]. The pre-filtered texture for the incoming light includes indirect light and direct light from light emitting materials.

In addition to stored light from the RDMs, the implementation evaluates direct specular and diffuse light from the sun in real time. The evaluated lighting from the sun is multiplied by a value gotten from a shadowmap check, where a previously mentioned shadowmap filtering method is used to provide soft edges to shadows [9]. The shadowmap is rendered for the area near the camera's target and surfaces outside of the shadowmap's area are assumed not to be in shadow. This approach is simple and works well for situations where things shown on camera are near each other, but for situations where far away things are displayed a more sophisticated approach with multiple shadowmaps rendered would be a large improvement.

Lastly, the implementation uses planar reflections to render the reflection of the water surface in the scenes. This is done because the RDM is not suitable to store reflections from a large plane that is not formed from voxel blocks. This is also done so that the water reflection can react to dynamic objects in the scene, such as a player character.

5. RESULTS

The rhombic dodecahedron mapping is evaluated in performance and quality. Quality evaluation is done by comparing the real time rendered images using the RDMs to path traced reference images and to real time rendered images without the RDMs. This comparison is done using the ꟻLIP metric, a tool for evaluating perceived differences between images developed by Nvidia [21]. Performance is evaluated by comparing the frametimes of real time rendering with the RDMs to real time rendering without the RDMs. This is so the impact on performance and the quality results achieved can be put in perspective of their performance costs.

Since the scenes RDM is suitable for have specific restrictions, no pre-existing scenes will be used. The measurements are done with two different scenes, made for the comparison using the implemented game engine's block and level editors: the island scene and the chess scene. The island scene is a scene with a couple of emitting voxel light sources, a large metallic reflective cube and smooth non-metallic spheres made out of the small voxels. The island scene has diffuse lighting in a big role in the quality comparison. The chess scene is more focused on the sharp reflections the RDM is meant for. It doesn't have emitting materials, and all light comes from the sun, it has a large number of blocks that reflect sharp reflections of the environment.

5.1 Quality comparison

The quality comparison is done with two scenes, the island scene and the chess scene. Both scenes are rendered in the path tracer to produce a reference image. Then both scenes are rendered in the real time renderer of the game engine without the lighting data from the RDMs and then with the RDMs. Both of the real time images are compared to the reference and the differences from the reference image are compared between them. Comparing the real time image with RDMs to the image without them is done to see how much closer to the reference the details from the RDMs bring the real time image and if details that are not present in the reference image are introduced. This comparison does not compare the RDM approach to another approach that would realistically be used in a real time renderer, since the details gotten from RDMs would be realistically substituted with details from some other method instead of just flat ambient lighting.

Table 5.1. *Measurement results for the quality comparison.*

Measurement	Mean ꟻLIP metric to reference (lower is better)
Island scene, with RDMs	0.209
Island scene, without RDMs	0.322
Chess scene, with RDMs	0.128
Chess scene, without RDMs	0.285

All of the images are rendered without the water, to avoid the differences between rendering the water in the real time and path tracing renderers introducing error to the measurements. In the real time images without the RDMs, a constant ambient lighting value replaces the indirect lighting gotten from the RDMs. The reference images are rendered at 5000 samples per pixel (spp) and the RDM images are calculated with the same sample count. All of the images' pixel values are tone mapped to the $0 \dots 1$ range with the Reinhard tone mapper, where a color value in the image is transformed with $C_{\text{tonemapped}} = \frac{C_{\text{original}}}{1+C_{\text{original}}}$ [22].

For the island scene, figure 5.1 displays the reference image, path traced with 5000 spp. Figure 5.2 displays the image produced by the real time renderer with RDMs and figure 5.3 without the RDMs. The ꟻLIP comparison between the reference image and the real time renderer with RDMs is visible in figure 5.4 and the comparison between the reference and the real time renderer without RDMs is displayed in figure 5.5. Table 5.1 contains the mean ꟻLIP values for both comparisons. The island scene render produced by the real time renderer moves significantly closer to the reference when the details and lighting from the RDM are added, with a 34.9% reduction to the ꟻLIP metric.

For the chess scene, figure 5.6 displays the reference image, path traced with 5000 samples per pixel (spp). Figure 5.7 displays the image produced by the real time renderer with RDMs and figure 5.8 without the RDMs. The ꟻLIP comparison between the reference image and the real time renderer with RDMs is visible in figure 5.9 and the comparison between the reference and the real time renderer without RDMs is displayed in figure 5.10. Table 5.1 contains the mean ꟻLIP values for both comparisons. The chess scene render was improved even more significantly by the RDMs, decreasing the ꟻLIP metric by 55.2%.

In both cases, using the RDMs brought the real time image significantly closer to the path traced reference. Visual inspection of the images also confirms the result, where both scenes look very flat without the additional lighting information from the RDMs.

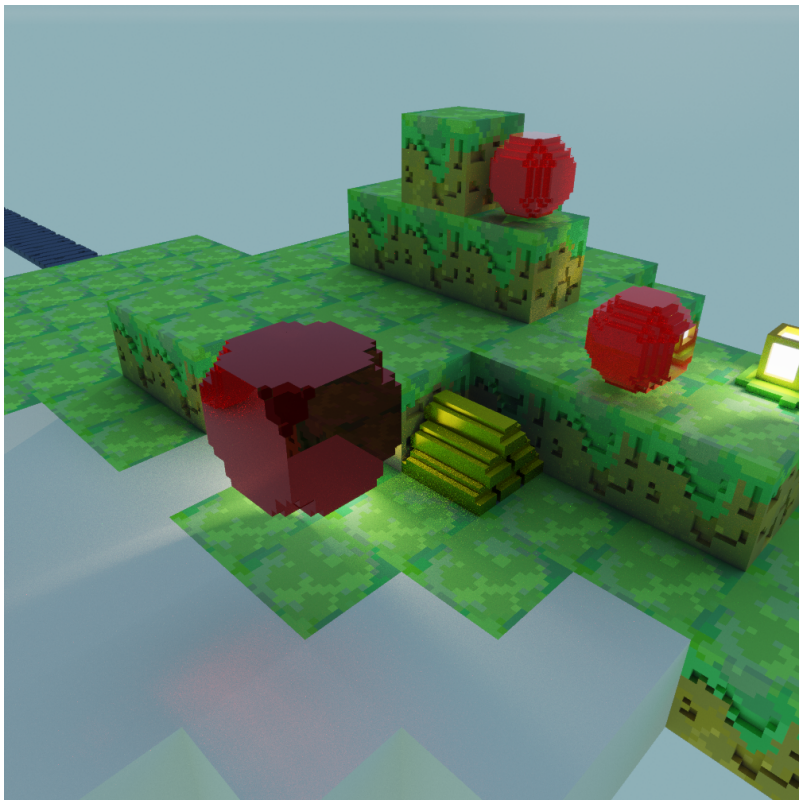

Figure 5.1. *The island scene rendered with the implemented path tracer at 5000 spp.*

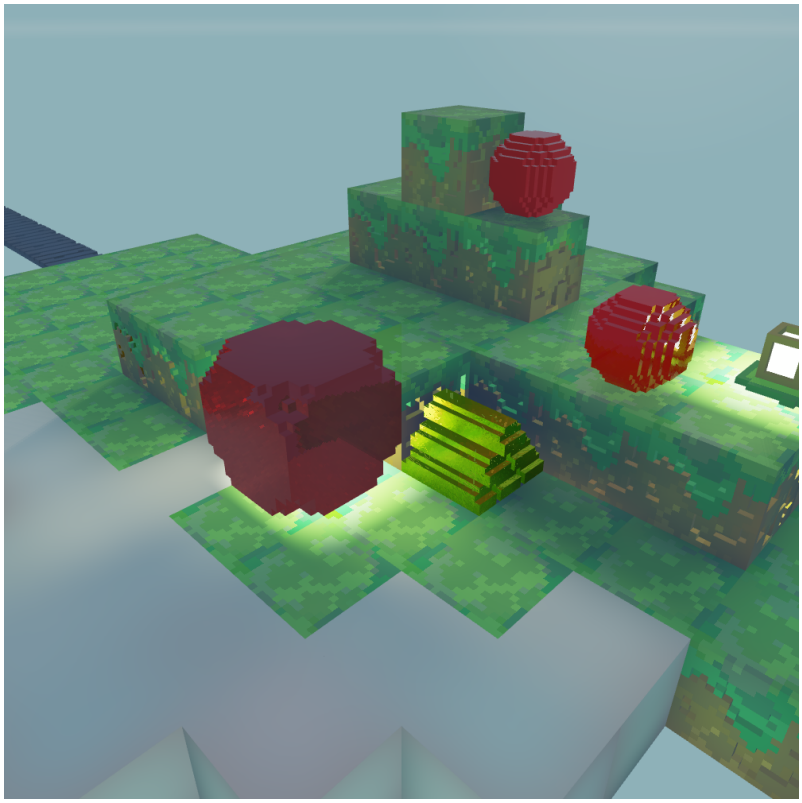

Figure 5.2. *The island scene rendered with the real time renderer using the data stored in RDMs.*

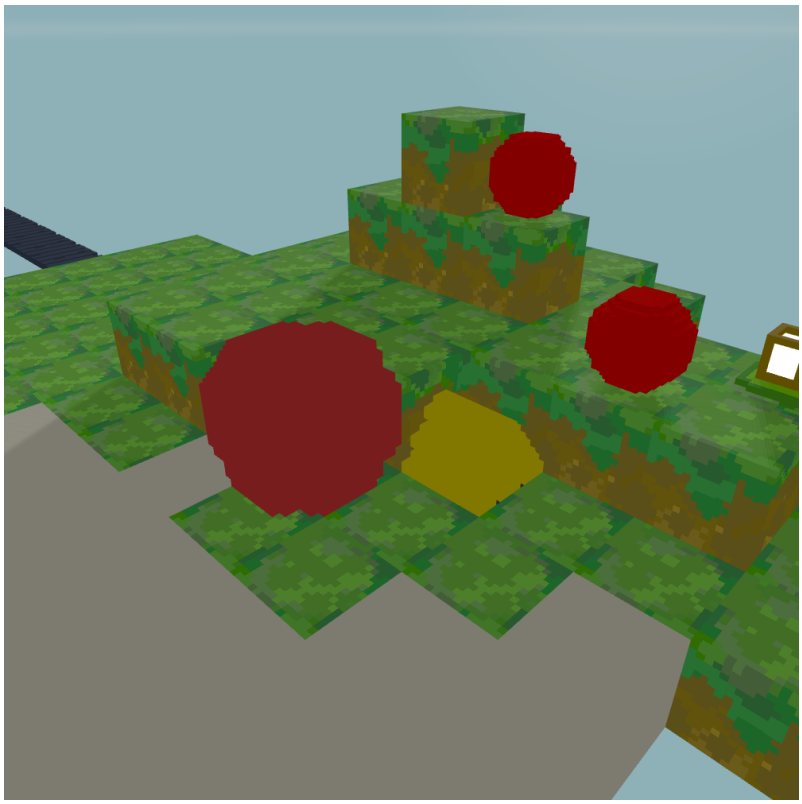

Figure 5.3. *The island scene rendered with the real time renderer without using precaclculated lighting data*

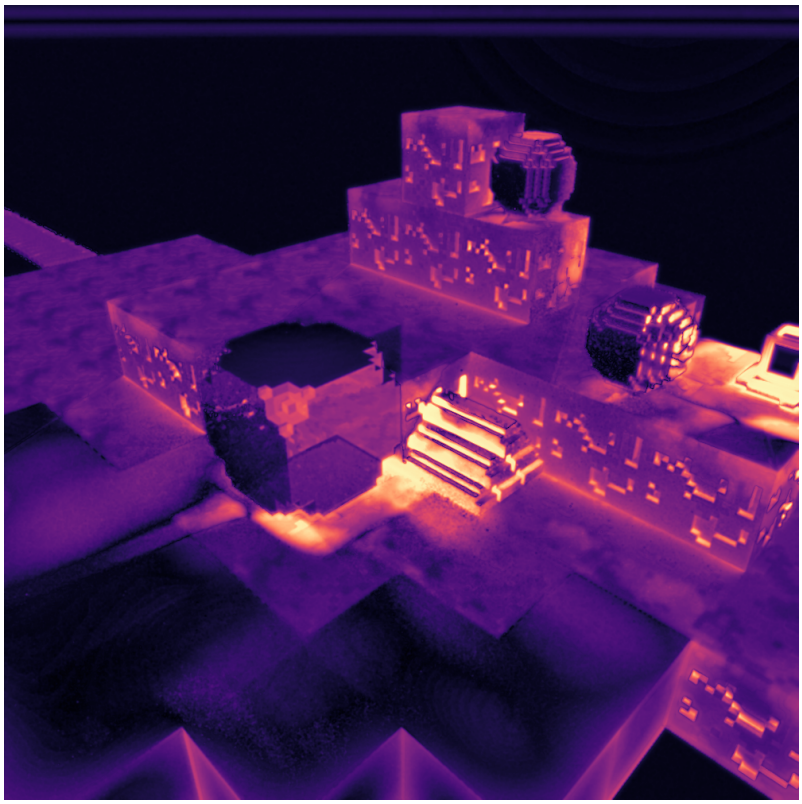

Figure 5.4. ꟻLIP *image displaying differences between the reference and the real time renderer with RDMs for the island scene.*

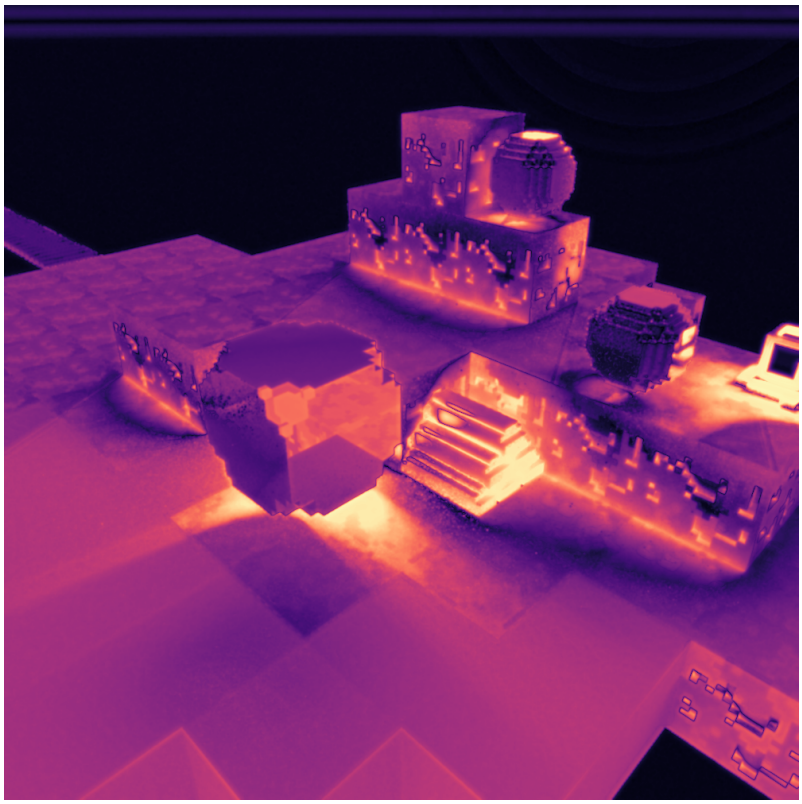

Figure 5.5. *FLIP image displaying differences between the reference and the real time renderer without RDM data for the island scene.*

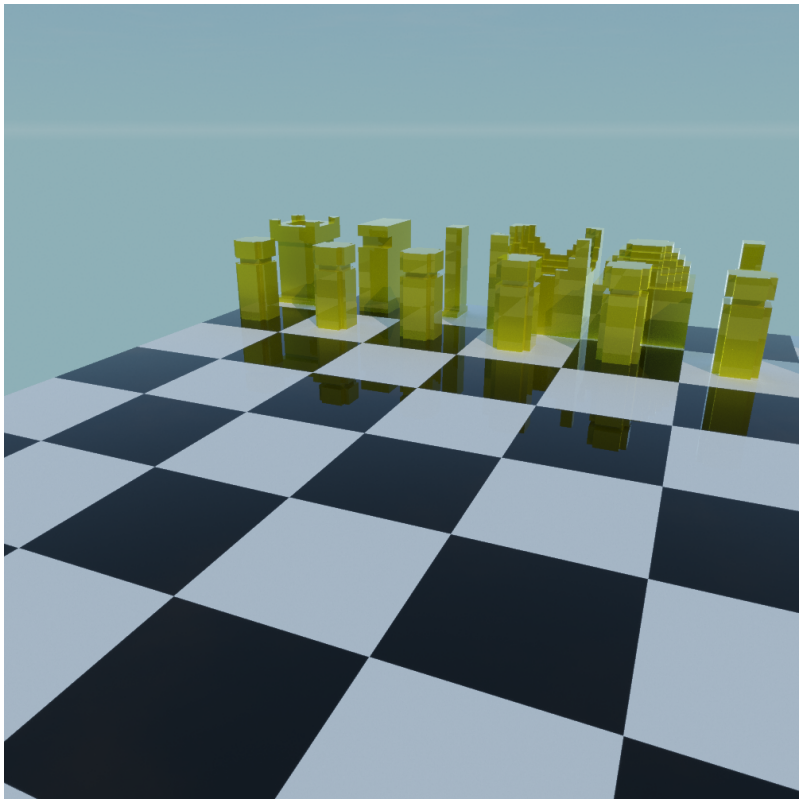

Figure 5.6. *The chess scene rendered with the implemented path tracer at 5000 spp.*

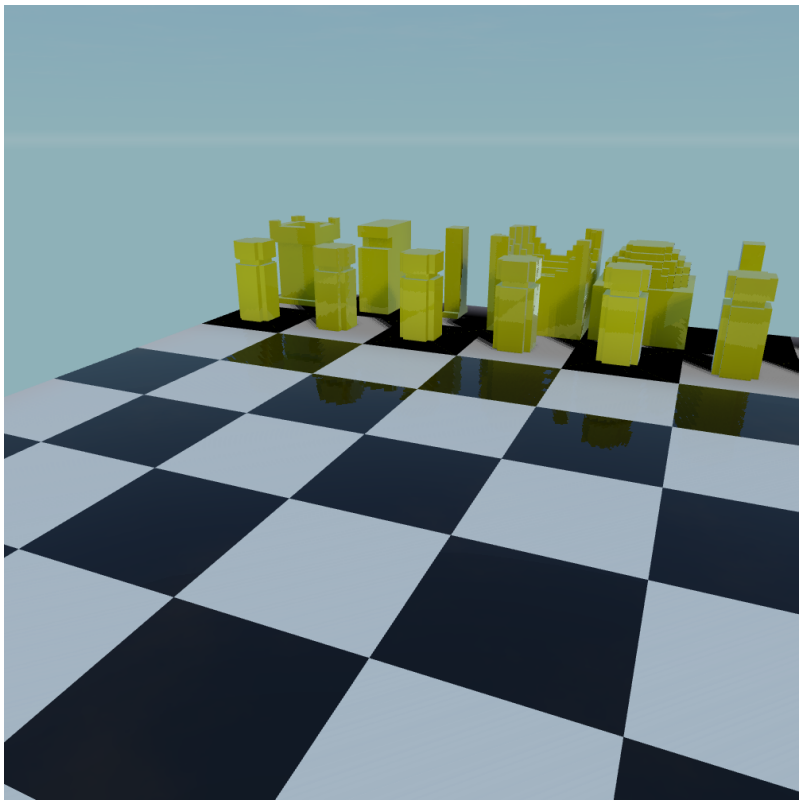

Figure 5.7. *The chess scene rendered with the real time renderer using the data stored in RDMs.*

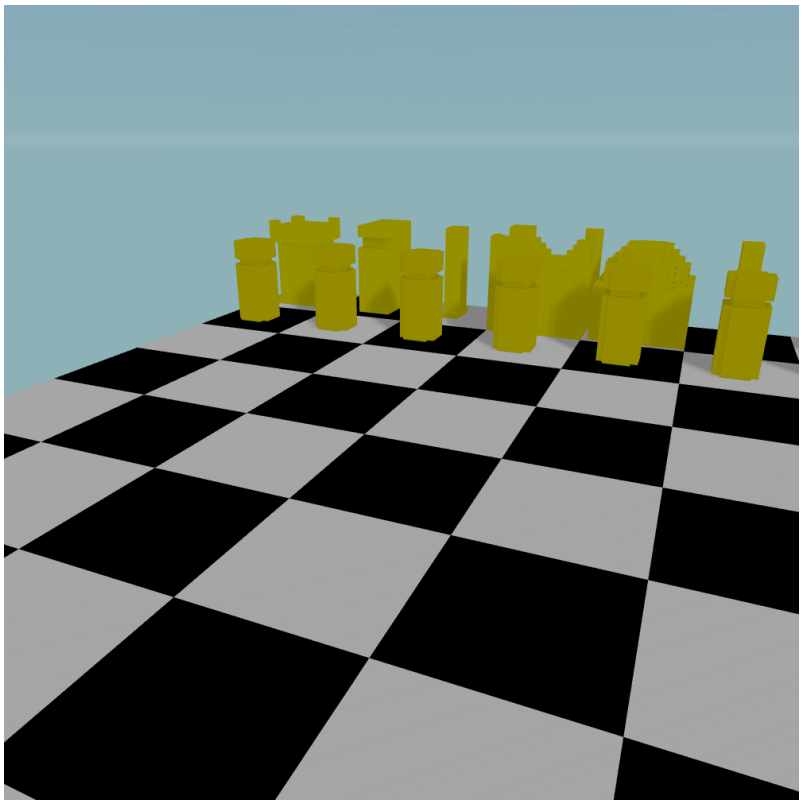

Figure 5.8. *The chess scene rendered with the real time renderer without using pre-calculated lighting data*

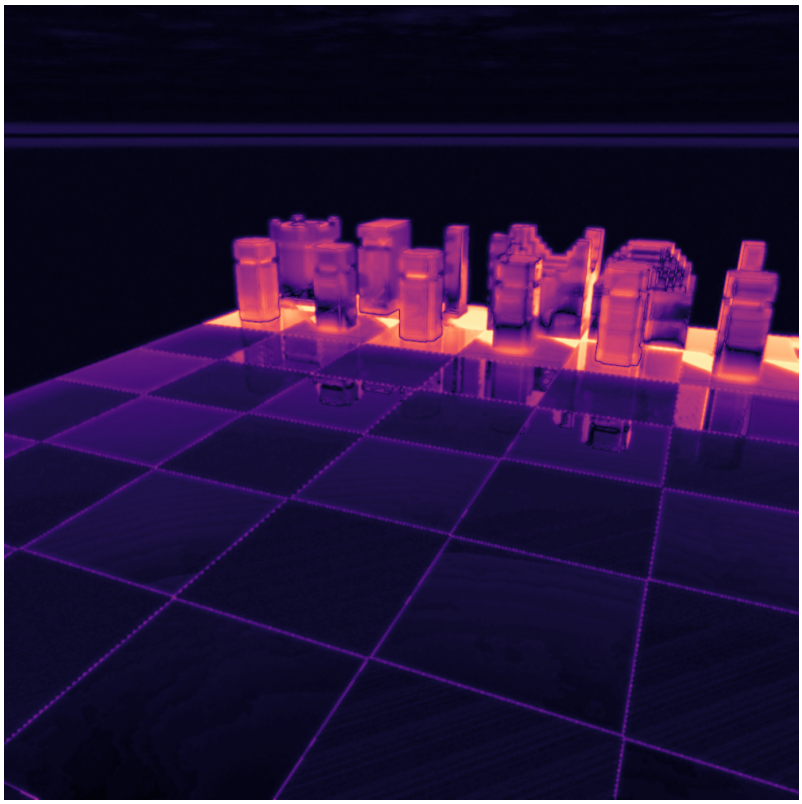

Figure 5.9. ꟻLIP *image displaying differences between the reference and the real time renderer with RDMs for the chess scene.*

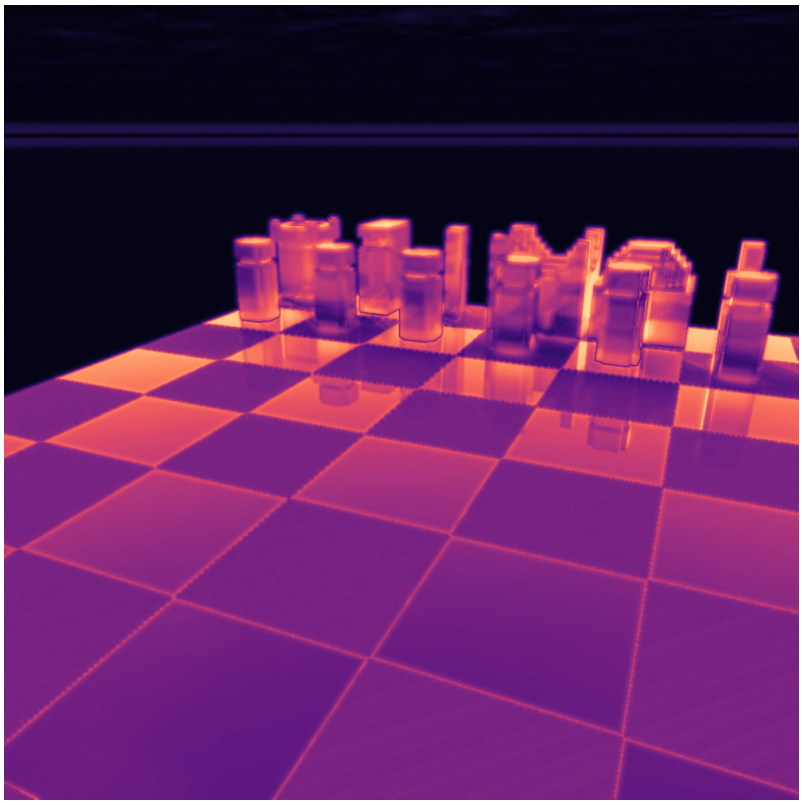

Figure 5.10. ꟻLIP *image displaying differences between the reference and the real time renderer without RDM data for the chess scene.*

Table 5.2. *Measurement results for the performance comparison.*

Measurement	Average frametime
Island scene, without RDMs	0.219 ms
Island scene, with RDMs	0.536 ms
Chess scene, without RDMs	0.167 ms
Chess scene, with RDMs	0.418 ms

5.2 Performance comparison

The performance comparison is with the real time renderer comparing its average frametimes with and without using the RDMs. This is done to evaluate the performance costs of the visual gains gotten from using RDMs. Performance is not compared to the path tracer, since it is not optimized for producing images in real time, so comparisons with it would not be indicative of the performance compared to a production grade real time path tracer. The performance measurements are done on an AMD Radeon 6700XT GPU by averaging the frametimes of running the scene for 10000 frames.

Table 5.2 shows the results of the performance comparison. In the island scene, the RDMs cost 0.32 milliseconds of additional rendering time. In the chess scene, the RDMs cost 0.25 milliseconds of additional rendering time. The island scene has 161 blocks and the chess scene has 80 blocks. The chess scene had all 80 blocks with large resolution RDMs, and the island scene had only a few high resolution RDMs.

5.3 Analysis

The performance impact of RDMs is large when compared to the frametimes without it, but the absolute time the RDM lighting took was fairly short. It seems that the performance cost rises with the number of blocks. The resolution of the RDMs doesn't seem to matter much for performance, which is in line with the fact that diffuse lighting requires sampling more RDMs to blend the values together and the specular lighting evaluation is a single sample regardless of the RDM's resolution.

The RDM approach's specular lighting has multiple different inherent problems that are visible in the comparisons against the reference images. Figure 5.11 displays one of these inherent problems, where the block's geometry is in the way of the reflection, but due to how the sampling works that is not taken into account. Figure 5.12 demonstrates another problem, where a block with detail can block the sampling point at the center of the block's side. This leads to the RDM information being missing, even though some of the surface is visible. Both of these issues are partly caused by the blocks consisting of small voxels instead of being perfect cubes.

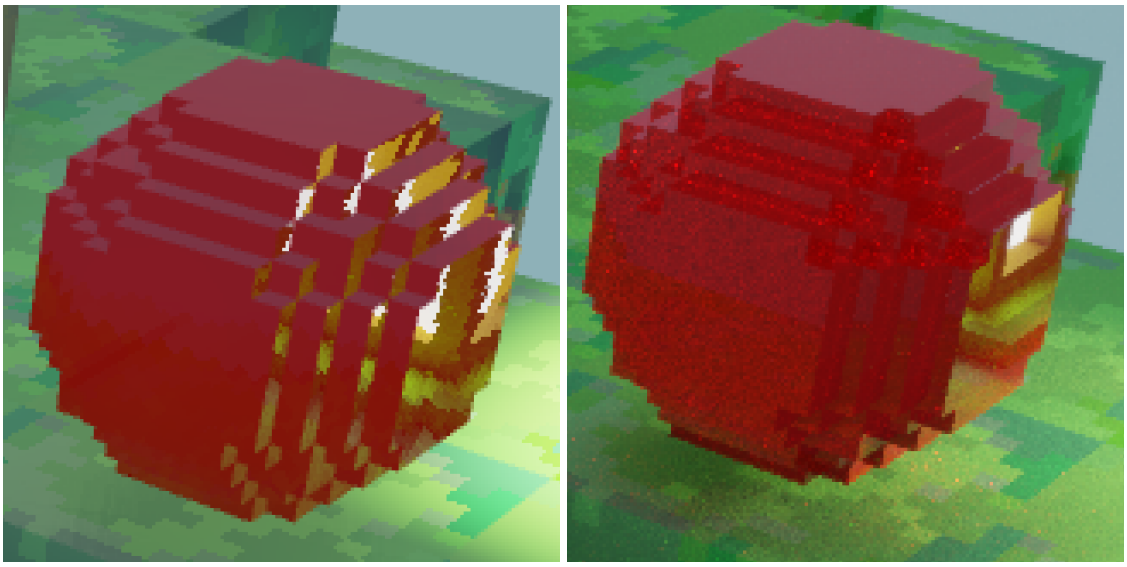

Figure 5.11. *In the left image using the RDM the surface of the sphere is reflecting light as if the reflection was not blocked by its own surface. The reference image is on the right.*

There are also issues with the diffuse lighting inherent to how it's implemented. Figure 5.13 highlights how the fact that diffuse lighting is always sampled from the hemisphere with the matching normal to the surface results in surfaces far away from the side that's receiving light receiving light they should not receive. Figure 5.14 shows how the low resolution of diffuse RDM data results in missing detail in the lighting.

Overall, the RDM method's performance is acceptable, and it adds details that bring results closer to the path traced reference and enhances the visuals of the images considerably, both visually and also according to the measurements. It has several key shortcomings, which can be worked around when designing scenes and blocks. The usage of RDMs would require clever scene design in video games anyways, since they are precalculated and can't react to scene changes or characters, so thoughtful placement of blocks with mirror-like surfaces is required in order to avoid glaring errors in the reflections.

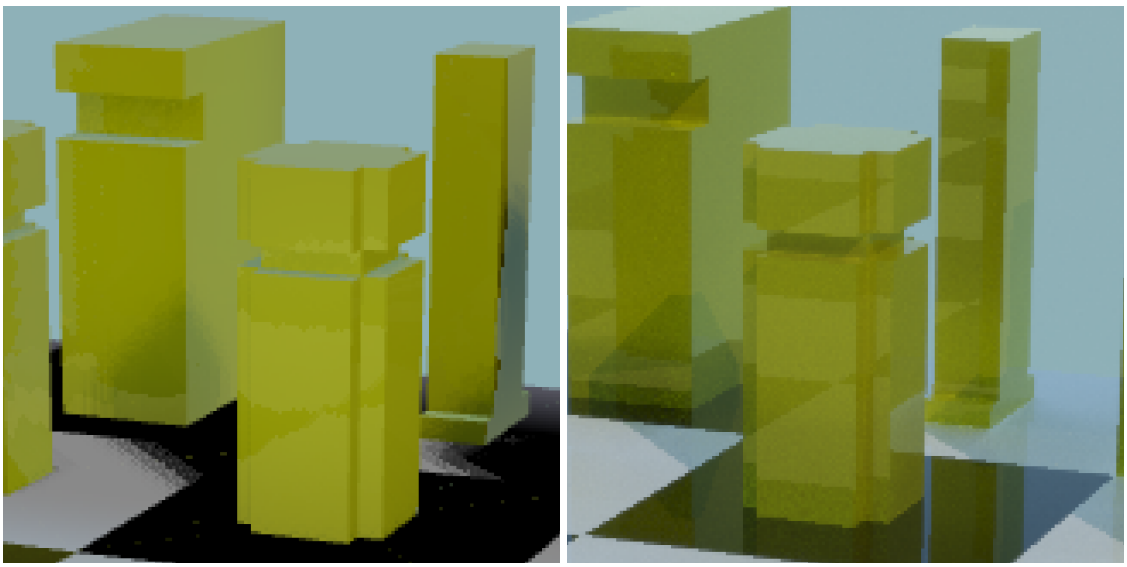

Figure 5.12. *In the left image using the RDM the tiles under chess pieces miss diffuse lighting and reflections, since the hemisphere sampling point is blocked by the chess piece. The reference image is on the right.*

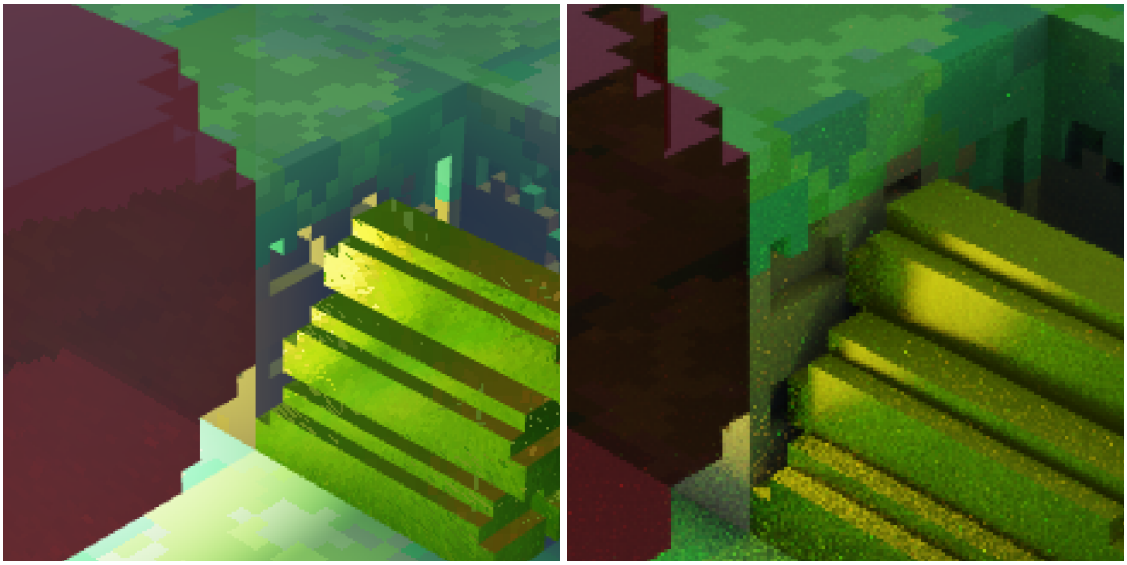

Figure 5.13. *In the left image using the RDM the details on the ground block are lit wrong due to them taking light values reaching the left side of the block and assuming the same amount of light is reaching the left side of the small voxels in the details. The reference image is on the right.*

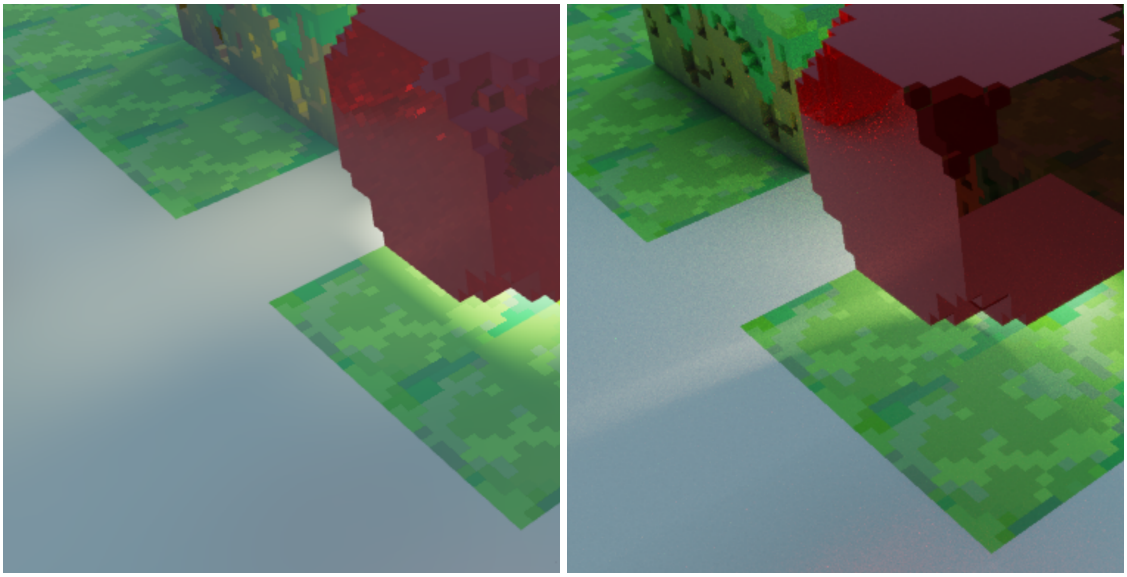

Figure 5.14. *In the left image using the RDM the lighting on the ground does not have detail due to the way diffuse lighting is done using RDMs. The reference image is on the right.*

6. RELATED WORK

The rhombic dodecahedron mapping is focused on rendering sharp reflections of the environment. This is a common problem in modern video games with large budgets and it has state of the art solutions used in these games. An example of a game with state of the art solutions to this problem is the previously mentioned Cyberpunk 2077 by CD Projekt RED [12]. It uses different techniques based on the user's graphical settings, adjusted based on their hardware and preferences for visuals and performance.

Cyberpunk 2077 employs at least three different already discussed techniques for sharp reflections of the environment. Figure 2.7 displays screen-space reflections in Cyberpunk 2077. Figure 2.9 displays raytraced reflections, enabled with a graphics quality setting. The indoor mirrors in the player's apartment use planar reflections to render a reflection of the player and the background. ReSTIR global illumination was also added to the game in a patch to improve the quality of path traced lighting [23].

As seen in the figures, the screen-space reflections result in inconsistent reflections with the core problem of not being able to show things that are off-screen. The ray traced reflections produce visually great results, but even at a relatively low resolution of 1920 x 1080, rendering with the top graphics settings, including raytracing for reflections and other lighting, is very performance intensive. Currently, reaching an average framerate above 60 fps is possible only with the 4 most powerful consumer-grade GPUs [24]. Currently the most popular GPU among users of the popular PC video game platform Steam, the NVIDIA GeForce RTX 3060 with a 5.76% share of Steam users, can only reach an average framerate of 24 fps [25, 24].

The rhombic dodecahedron mapping wouldn't be a suitable technique for these kinds of games for multiple reasons. The scenes in these games are not constrained to a grid of voxel blocks. Even if they were, the fact that the pre-calculated rhombic dodecahedron mappings can't adjust to dynamic changes in the scene would possibly be unacceptable for this kind of game. Also, storing the rhombic dodecahedron mappings for a massive environment such as Cyberpunk's Night City could prove too storage intensive. The rhombic dodecahedron mapping's advantages are the visual quality of its results in combination with its performance and the relative simplicity of its implementation.

7. CONCLUSION

The goal of this thesis was to present a new method for rendering indirect lighting focused on sharp reflections in scenes where the scenes are formed from voxel blocks. The method's performance was fast enough to be usable in real time applications such as video games.

The method presented had performance characteristics suitable for real time applications, where the method's performance impact was determined mainly by the number of blocks in the scene. The method's resulting visual quality was good and was close to the path traced reference.

A lot of the difference to the path traced reference was from inherent characteristics of the method. The method was more accurate for sharp reflections and indirect specular light than it was for diffuse lighting, which is expected given the method's focus on specular lighting. The method is best suited to situations where the blocks are as close to perfect cubes as possible, due to its design and the sampling points for the stored hemispheres. Many of the methods problems could be worked around by thoughtful design of the voxel blocks and scenes. Using the method for diffuse lighting needs to be carefully considered, since the lack of detail can result in low lighting quality. The method is fairly simple to implement and use. The largest restriction of the method is its inability to react to changes in the scenes, making it not viable for many use cases.

The method was not compared to state of the art solutions, since a comparison would have required either implementing state of the art lighting solutions in the implementation's game engine or implementing the game's scenes and materials in some other engine in a comparable way. Both of these options were scoped out of this work due to the time and effort they would have required.

The method proved suitable for simple use cases with the given scene restrictions and produced believable reflections. The method had performance characteristics suitable for low-end hardware and it successfully fulfilled the goals of this thesis.

7.1 Future work

There are many interesting potential ways to further develop the method and combine it with other techniques to produce better results and add more flexibility.

Adding some sort of ambient occlusion, such as SSAO or some other ambient occlusion technique to the blocks could help compensate for some of the errors the method produces, especially with diffuse lighting, where there are issues caused by only having 1 value for incoming diffuse lighting from a direction. This causes places that would normally receive less light, such as corners, to have no difference to other parts of the same block.

Storing data about whether a part of a block is visible when viewed from a certain side would also help mitigate issues in cases where a block receiving lots of light from the top can have a small voxel within it that is not supposed to be receiving light from the top of the block. In the current implementation, that small voxel's top side would still be lit according to the light coming to the top of the entire block. If a small voxel is not visible from a side, that side's data could be discarded when shading that voxel.

Many of the weaknesses of the RDM method are present, especially with diffuse lighting. Combining the RDMs with a light mapping technique for diffuse light could enable the scene to have accurate diffuse lighting and RDMs could be rendered and stored only for the blocks for which specular lighting or sharp reflections are important.

REFERENCES

[1] Tomas Möller et al. *Real-time rendering*. eng. Fourth edition. Boca Raton: CRC Press, 2018. ISBN: 0-429-22540-7.

[2] James T. Kajiya. “The rendering equation”. eng. In: *Computer graphics (New York, N.Y.)* 20.4 (1986), pp. 143–150. ISSN: 0097-8930.

[3] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. MIT Press, 2023.

[4] *The Stanford 3D Scanning Repository*. (visited 14.10.2024). URL: http://graphics.stanford.edu/data/3Dscanrep/.

[5] Martin Mittring. “Finding next gen: CryEngine 2”. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH ’07. San Diego, California: Association for Computing Machinery, 2007, pp. 97–121. ISBN: 9781450318235. DOI: 10.1145/1281500.1281671.

[6] Carsten Dachsbacher and Marc Stamminger. “Reflective shadow maps”. In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D ’05. Washington, District of Columbia: Association for Computing Machinery, 2005, pp. 203–231. ISBN: 1595930132. DOI: 10.1145/1053427.1053460.

[7] Y. Ouyang et al. “ReSTIR GI: Path Resampling for Real-Time Path Tracing”. eng. In: *Computer graphics forum* 40.8 (2021), pp. 17–29. ISSN: 0167-7055.

[8] Lance Williams. “Casting curved shadows on curved surfaces”. In: *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’78. New York, NY, USA: Association for Computing Machinery, 1978, pp. 270–274. ISBN: 9781450379083. DOI: 10.1145/800248.807402.

[9] Ignacio Castaño. *Shadow Mapping Summary*. 2013. URL: https://www.ludicon.com/castano/blog/articles/shadow-mapping-summary-part-1/.

[10] “Part III: Reflections, Refractions, and Shadows”. eng. In: *Ray Tracing Gems*. Apress L. P, 2019. ISBN: 9781484244265.

[11] Brent Cowan et al. “Screen space point sampled shadows”. eng. In: *2015 IEEE Games Entertainment Media Conference (GEM)*. IEEE, 2015, pp. 1–7. ISBN: 1467374520.

[12] CD Projekt RED. *Cyberpunk 2077*. 2020.

[13] Johannes Deligiannis and Jan Schmid. *It Just Works: Ray-Traced Reflections in Battlefield V*. 2019. URL: https://www.youtube.com/watch?v=ncUNLDQZMzQ.

[14] Peter-Pike Sloan. “Stupid Spherical Harmonics (SH) Tricks”. In: *Game Developers Conference* (Jan. 2008).

[15] Dylan Lacewell. “Ray-Tracing Small Voxel Scenes”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs,

Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 599–609. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_37.

[16] Renaud Bédard. *Behind Fez : Trixels (and why we don't just say voxels)*. (visited 15.10.2024). 2007. URL: https://theinstructionlimit.com/behind-fez-trixels-and-why-we-dont-just-say-voxels.

[17] Cyril Crassin et al. "Interactive Indirect Illumination Using Voxel Cone Tracing". eng. In: *Computer graphics forum* 30.7 (2011), pp. 1921–1930. ISSN: 0167-7055.

[18] Zina Cigolle et al. "A Survey of Efficient Representations for Independent Unit Vectors". In: *Journal of Computer Graphics Techniques* 3 (Apr. 2014), pp. 1–30.

[19] Brian Karis. "Physically based shading in theory and practice: Real Shading in Unreal Engine 4". In: *ACM SIGGRAPH 2013 Courses*. SIGGRAPH '13. Anaheim, California: Association for Computing Machinery, 2013. ISBN: 9781450323390. DOI: 10.1145/2504435.2504457.

[20] Renaud Bédard. *Behind Fez : Trixels (part two)*. (visited 9.11.2024). 2009. URL: https://theinstructionlimit.com/behind-fez-trixels-part-two.

[21] Pontus Andersson et al. "ꟻLIP: A Difference Evaluator for Alternating Images". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), 15:1–15:23. DOI: 10.1145/3406183.

[22] Erik Reinhard et al. "Photographic tone reproduction for digital images". eng. In: *ACM transactions on graphics* 21.3 (2002), pp. 267–276. ISSN: 0730-0301.

[23] *Cyberpunk 2077: UPDATE 2.1 PATCH NOTES*. (visited 27.11.2024). 2023. URL: https://www.cyberpunk.net/en/news/49597/update-2-1-patch-notes.

[24] Steve Walton. *Cyberpunk 2077: Phantom Liberty GPU Benchmark*. (visited 27.11.2024). 2023. URL: https://www.techspot.com/review/2743-cyberpunk-phantom-liberty-benchmark/.

[25] *Steam Hardware & Software Survey: October 2024*. (visited 27.11.2024). 2024. URL: https://store.steampowered.com/hwsurvey/videocard/?sort=pct.