DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Object Space Shading in Unity

Tim Kaiser

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Object Space Shading in Unity

Objektraum Shading in Unity

Author:	Tim Kaiser
Supervisors:	Dr. Matthäus G. Chajdas
	Marcus Rogowsky
Advisor:	Prof. Dr. Rüdiger Westermann
Submission Date:	15.02.2019

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.02.2019 Tim Kaiser

Acknowledgments

I dedicate this to my parents, who supported me through all my life, and who I will always admire for their strength. I would also like to thank everyone that helped me with this thesis.

Abstract

Object Space Shading is a rather new technique in real-time rendering, only used in one commercially available game so far, called "Ashes of the Singularity". This shading method differs from previous approaches like Forward Rendering or Deferred Shading by the fact that the actual shading does not happen during or after rasterization, but before. The purpose of this thesis is to research the performance benefits and extra computational cost Object Space Shading causes. Additionally, it wants to show that this approach can be easily integrated into an already existing environment, like the Unity3D Game Engine.
To show this, a render pipeline capable of performing Object Space Shading was implemented in a new Unity project. This was possible due to Unity's new "Scriptable Render Pipeline" feature, which allows the user to take full control over the rendering process via a C# script. This allowed us to perform two render passes during each frame, one to determine which part of the object is visible, and one to render the textures, that were shaded between the render passes, back to the screen.
The tests performed in this project, which measure the performance benefits and costs of Object Space Shading, show promising results. Up to 95% of already shaded pixels can be reused each frame and less than 5% have to be shaded new. The costs vary a bit more depending on screen size, but are always less than twice the costs of a Forward Renderer, if the parameters of the Object Space Shading renderer are chosen correctly. This, of course, is still a lot but is acceptable thanks to the before mentioned savings.

Kurzfassung

Objektraum Shading ist eine relativ neue Echtzeit-Rendermethode, die bisher nur in einem einzigen Computerspiel, nämlich „Ashes of the Singularity“, verwendet wird. Diese Rendermethode unterscheidet sich dadurch von bisherigen Ansätzen wie Forward Rendering oder Deferred Shading, dass sie nicht während oder nach der Rasterisation shaded, sondern davor. Ziel dieser Arbeit ist es, die Performance Ersparnisse und Kosten von Objektraum Shading zu messen. Zusätzlich soll gezeigt werden, dass ein solcher Ansatz sich ohne größerer Probleme in eine bereits existierende Umgebung wie die Unity3D Game Enginge einbauen lässt.
Um dies zu zeigen wurde eine eigene Render Pipeline, die in der Lage ist Objektraum Shading durchzuführen, in Unity implementiert. Die war möglich danke Unitys neuem „Scriptable Render Pipeline“ Feature, dass dem Nutzer erlaubt den Render Prozess über ein C# Script zu kontrollieren. Dies ermöglicht es uns zwei Render Durchläufe pro Frame zu haben. Einen Ersten, der uns zeigt, welche Teile eines Objekts sichtbar sind, und einen, der die Texturen, die in der Zwischenzeit geshaded wurden, auf den Bildschirm rendert.
Die Tests, die anschließend durchgeführt wurden, um Performance zu messen, zeigen vielversprechende Ergebnisse. Bis zu 95% der bereits geleisteten Arbeit kann im nächsten Frame wiederverwendet werden und nur 5% müssen neu berechnet werden. Die genauen zusätzlichen Kosten sind etwas schwerer zu bestimmen, da sie je nach Bildschirmauflösung sehr variieren können, liegen aber unter dem Doppelten der Kosten eines Forward Renderers. Das klingt zwar erstmal nach viel, ist aber dank der guten Ersparnisse zu verkraften.

Contents

1. Introduction

1.1. Motivation

The way we render three-dimensional objects did not change fundamentally in the last decades, since the early days of computer graphics. Forward renderers, which were already used in "Quake" in 1996 [Abr97], are still commonly used in computer games and Deferred rendering too is over a decade old[1]. Sure, incremental improvements and optimizations happened over the years, making both render techniques very efficient and allowing them to create lifelike images, but in their foundation, they are still the same. What both of them have in common is, that they shade in the mathematical space of the screen [Bak16], which helps to avoid unnecessarily shading occluded parts of the scene, but is rather unintuitive. Object Space Shading, a rather new render method, at least in real-time rendering, on the other does shading centered around the object and not the camera, hence the name. It also stores all shading results in textures, allowing them to be used again later, which helps to avoid doing the same calculations twice. While the idea of precomputing shading and storing the results in textures has also been around for a long time, doing it during rendering is kind of new. So far only one commercially available game called "Ashes of the Singularity" is using this technique. Oxide Games, the developer, claims to get good performance out of this approach, but scientific research about this has not been done, which is where this thesis comes into play.

1.2. Contribution

To contribute to the current research, this thesis quantifies the additional computational costs and possible improvements of the somewhat new method of "Object Space Shading" over more traditional shading methods like Forward Rendering. This thesis also wants to show that such an approach can be integrated relatively easy in already existing game engines like Unity. The key questions this thesis focuses on are as follows:

- What are the added extra costs of Object Space Shading compared to other methods?
- How much of the results of the previous frame can be reused in the next frame and how big are the possible performance improvements from that?
- Do the benefits outweigh the extra costs and is this a feasible approach to shading?

[1] https://sites.google.com/site/richgel99/home (visited on: 12.02.2019)

To answer these questions I implemented a render pipeline capable of using Object Space Shading. For that I used Unity's new Scriptable Render Pipeline feature, that lets its users manipulate the way Unity renders game objects. This also allows us to add new shader passes, a hard requirement for Object Space Shading. After that, I set up two scenes, that are commonly used in testing and showcasing render methods, and prepared them to be used in my render pipeline. I implemented three tests, one to measure costs and two to measure performance in both scenes, and used Python to evaluate the resulting data. I use these statistics to answer the posed questions.

1.3. Thesis Outline

This thesis consists of six chapters, including this one. Chapter 2 defines the theory that forms the foundation of this thesis. This includes basic principals of what shading is in general and how it differs from the Object Space Shading that is done in the project accompanying this thesis. This chapter also gives information about the Unity Game Engine and some of its for this thesis important features. Chapter 3 then briefly discusses "Ashes of the Singularity", a commercially available game developed by Oxide Games, that already uses Object Space Shading. Subsequently, chapter 4 outlines the implementation of the project, its integration into Unity and the limitations this specific code poses. The chapter after that, chapter 5, explains and evaluates the statistics that were taken from the project. Lastly, chapter 6 tries to answer the questions posed in the introduction and draws a final conclusion for this thesis.

2. Theory

In this chapter, I will outline the basis of theory required to understand this thesis. At first, I will introduce a few basic terms and concepts. Afterward, I will explain what Object Space Shading is and how it differs from previous approaches. And finally I will take a look at Unity, the game engine this project will be implemented in, with special focus on its new "Scriptable Render Pipeline" feature.

2.1. Basic Terms and Concepts

This section will define some of the basic terms used in this thesis as well as explain some of the fundamental concepts of computer graphics associated with shading and rendering.

2.1.1. Shading

In computer graphics shading is the process of assigning colors to points on a surface based on outgoing radiation [Ola+02, p.5]. These can be influenced by multiple things, like the color and intensity of the incoming light, the reflective properties of the surface point or the object's color itself, represented by its texture. This makes the objects in the scene look three-dimensional and gives them depth by modeling light and shadows, which is where shading gets its name from. There are multiple different approaches to shading. While some of them try to look photo-realistic by imitating real-world physics, others want to go a more artistic and stylized way and create cartoonish images, with highlighted edges and very few nuances in color. The formula to calculate outgoing reflection used in the project accompanying this thesis is based on the Phong Illumination Model, a shading approach first described and named after Bui Tuong Phong in 1975 [Pho75]. This model splits the reflection into three parts: ambient, diffuse and specular. The ambient reflection is a constant that tries to account for all the indirect light an object would receive. While the diffuse reflection only depends on the color and intensity of the incoming light and the angle in which this light hits the surface, the specular reflection needs additionally the position of the viewer. All three of these together combined with the texture of the object result in the final color visible on the screen.

2.1.2. Rendering

Rendering is the process of creating a two-dimensional image from given data [Han+12]. This data typically describes a three-dimensional scene with multiple objects in it. These objects are usually represented by a 3D meshes which consist of numerous polygons, typically

triangles. The color of the object is either uniform for each triangle, a gradient between the points of a triangle, or described by an image, the so-called texture. The scene also often includes light sources and a virtual camera, which represents the position and the view direction of the viewer. The rendering process involves multiple steps, including geometric transformations, culling, which is the act of removing objects and faces not visible in the final image, and shading. These steps are described in more detail in the next subsection.

2.1.3. Render Pipeline

The Render Pipeline is a combination of processes whose main purpose is to generate a two-dimensional image by rendering a given scene [McS13]. The pipeline can be roughly divided into three conceptual stages: application, geometry and rasterizer. Each of them typically consists of multiple substages. [AHH08, p.12-14]

Application Stage

This is the first stage in the pipeline. It's also the stage the developer has the most control over, since it executes on the CPU, and not on dedicated graphics hardware. This stage contains all the code that is not directly related to graphics. In a computer game, this would include player movement, collision detection, user input, game AI and more. The most important job of the application stage, with respect to the render pipeline, is to pass on the rendering primitives, like points or triangles, to the next stage. [AHH08, p.14-15]

Geometry Stage

The next stage is the geometry stage. It handles most of the geometric transformations applied to the scene, hence the name. This stage can be divided further into the following steps: model and view transform, vertex shading, projection, clipping, and screen mapping. [AHH08, p.15]
During the model and view transform stage, all coordinates in the scene get converted to camera space. Camera space is a coordinate system with the camera at the origin, rotated in a way that the camera's view direction is aligned with the z-axis. This transformation can be done with simple matrix multiplications [AHH08, p.16].
Next is the vertex shading stage. In a typical render pipeline, some of the shading work would be done here. Some of the data, like normal vector, position and texture mapping coordinates would be assigned to every vertex during this stage, and then be interpolated during the rasterization stage. Most of this can be skipped for this project since I try a different approach to shading. [AHH08, p.17]
The next two steps are projection and clipping. During the projection phase, the scene's geometry is transformed again, this time into normalized device coordinates. During this transformation, everything that is visible for the camera is mapped between (-1,-1,-1) and (1,1,1). Everything outside this gets removed during clipping. [AHH08, pp.18-19]
In the last step, the normalized device coordinates get mapped to the actual screen, which

means that the x- and y-coordinates get scaled to the width and height of the screen. The z-coordinates stay unaffected. The data then gets passed on to the rasterizer stage. [AHH08, p.20]

Rasterizer Stage

All primitives that enter this stage are rasterized, which means they get converted into individual pixels. Pixels within a triangle get interpolated data, like normal or position, from the vertices of the triangle. Then, a pixel shader, also called fragment shader, is used on each individual pixel. This shader maps a texture to each pixel, if needed. In a Forward Renderer it would also do the final lighting and shading calculations. In case of this project, no additional shading is added. For Object Space Shading, this is done at a different part of the pipeline. Here only the already shaded texture gets applied to the pixel during the fragment shader stage. The final image can than be displayed on the screen or stored into a texture. This completes the render pipeline. [AHH08, p. 26]

2.1.4. Texture and Object Space

Different parts of the scene and the rendering process are defined by different coordinate systems. The most common is the world space, also known as global space. This is a complete three-dimensional representation of the scene, with no special origin. But there is also object space, which is also called local space, camera space, clip space, screen space, texture space and so on. The two coordinate systems most interesting for us here are texture and object space.
Texture space is used to describe points in the texture of an object. It is a two-dimensional system with a horizontal u-axis and a vertical v-axis. The axes are named that way to avoid confusion with other coordinate systems. Coordinates in this system are called Texture- or UV-coordinates and are usually between (0,0) and (1,1). Texture coordinates describe a position in a texture independent of its dimensions. (0,0) is always the upper left corner, (1,1) always the lower right corner and (0.5,0.5) always the center of the texture, no matter its size. The mip level could be regarded as a third coordinate. More about that is written in the next subsection. [Lun12, p.314]
Object space is a three-dimensional coordinate system which is unique for every object in a scene. The respective object is placed at the origin. The coordinate system moves and rotates with the object. 3D meshes are represented in object space. Object space coordinates can be converted to world space coordinates by using the object's local-to-world-matrix. [Gei13, p.26]
It is called Object Space Shading, even though we use UV coordinates most of the time, since calling it "Texture Space Shading" would lead to the impression that we shaded directly into the texture, which would affect all objects with the same texture. Instead, we create a copy of the texture for every object and shade them individually, thus making it unique for each object, which is why we call it Object Space Shading. The exact process is described in section 2.2.

2.1.5. Mipmaping

Mipmapping is a popular method of anti-aliasing, a way to make distanced objects look better and avoid visual artifacts. This is done by generating a series of images, each getting filtered down to a quarter of the size of the previous one, starting with the original texture [AHH08, p.163]. During rendering the image which corresponds best with the object's size on the screen is chosen as texture for each surface point of an object. This sounds as if it would use a lot of resources, but it actually only increases the memory space used by a third, as illustrated in figure 2.2. This technique was invented by William Lance and first presented in his paper "Pyramidal Parametrics" in 1983 [Wil83].
In this project, mipmaping is used to drastically reduce the number of pixels we have to shade, since we shade in object space. Smaller objects and objects that are far away from the camera have a smaller visible texture thanks to mipmaping, which reduces the amount of work we have to do.

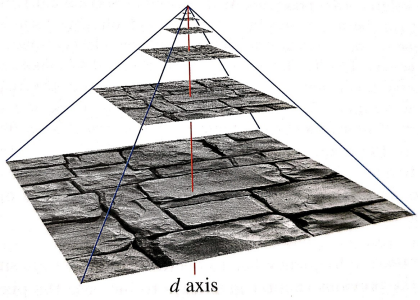

Figure 2.1.: Image Pyramid of progressivly smaller Mip Maps
[AHH08, p. 164]

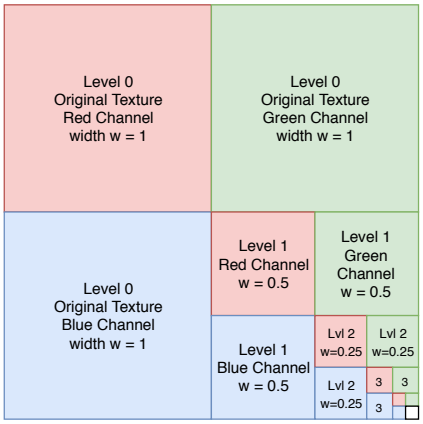

Figure 2.2.: Space allocation for Mip Maps

2.2. Object Space Shading

2.2.1. Basics

"Texture shading is a [...] shading implementation technique that precomputes a complex lighting model (or parts of a complex lighting model) and stores it into one or more texture maps." [Ola+02, p.113]. This is essentially also what Object Space Shading does, and it is not a new technique. Even "Quake", from the early days of 3D computer games, involved some kind of texture space lighting [Abr97]. What Object Space Shading does differently is that instead of creating light maps once before the start, the shaded textures are continuously updated during the runtime of the program.
In a more traditional, forward rendering approach, the shading would happen during the

rasterization process, as already mentioned before. The fragment shader gets the rasterized fragments and computes the light reflection at that point. Normal and world position are already part of the fragment data, interpolated from the vertex buffer, and do not have to be computed at that part. This is the standard method used for decades now. This is a very efficient way of rendering, thanks to years of performance optimization.
Another common approach is deferred shading. This render method is in most parts not really different from Forward Rendering. Where they differ is, that a deferred renderer does not shade during, but after rasterization. This technique uses intermediate buffers as render targets for the first fragment shader pass, to store screen-space, per pixel information. This includes diffuse color, normal vector and depth value. These buffers are then given to a shading algorithm, which computes the final image, without needing the original geometry. This approach simplifies the rendering pipeline and allows us to use more advanced lighting techniques. Both Forward and Deferred Rendering are visualized in figure 2.3. [Eng10, pp.561-562]
Now that we have seen two techniques that shade during and after rasterization, we have to ask ourselves: Can we shade before rasterization? This is exactly what Object Space Shading does. This render technique takes the texture of each object, shades it, and stores the result in a texture atlas. The fragment shader only reads the already shaded texture from the texture atlas during rasterization and outputs the result. This, of course, poses many challenges. The first one is, that we need to know which parts of the objects are actually visible. To determine this, a first render pass renders the scene and returns the visible UV coordinates of each object. These are used in a compute shader to create a binary mask, that contains the information of which part of the texture is visible. A 1 in the mask means that this part of the texture can be seen on the screen, a 0 means it can not. The mask does not have to have the same size as the texture. Each pixel in the mask contains information for one tile that covers multiple pixels of the same texture. If one of these can be seen on screen, the whole tile is shaded. Then, a second compute shader iterates the mask and shades the respective part of the texture, if that part of the mask is set to 1. Here, the second challenge arises. We do not have all the data we need for shading. In a forward or deferred renderer, we would already have the normal and position of each fragment, since this gets passed on by the vertex shader and interpolated by the rasterizer. But with Object Space Shading, we do not have this information, since rasterization did not happen yet. Instead, we have to compute this at the beginning of the program before the actual rendering starts. In case of this project, the vertex IDs of the triangle a pixel is located in and the respective barycentric coordinates get stored into a texture at the beginning. These can then be used to interpolate the necessary data from the vertex buffer, which in turn can be used for shading. The final result gets rendered to the screen by the second render pass.
Lastly, Object Space Shading requires that all UV coordinates of every object are chartable, which means that no object can use any part of its texture more than once. Otherwise, we would have to shade this part twice, which makes no sense. It is not an uncommon require for rendering techniques. [Bak16]
The different stages of an Object Space Shading render pipeline can also be seen in figure 2.3.

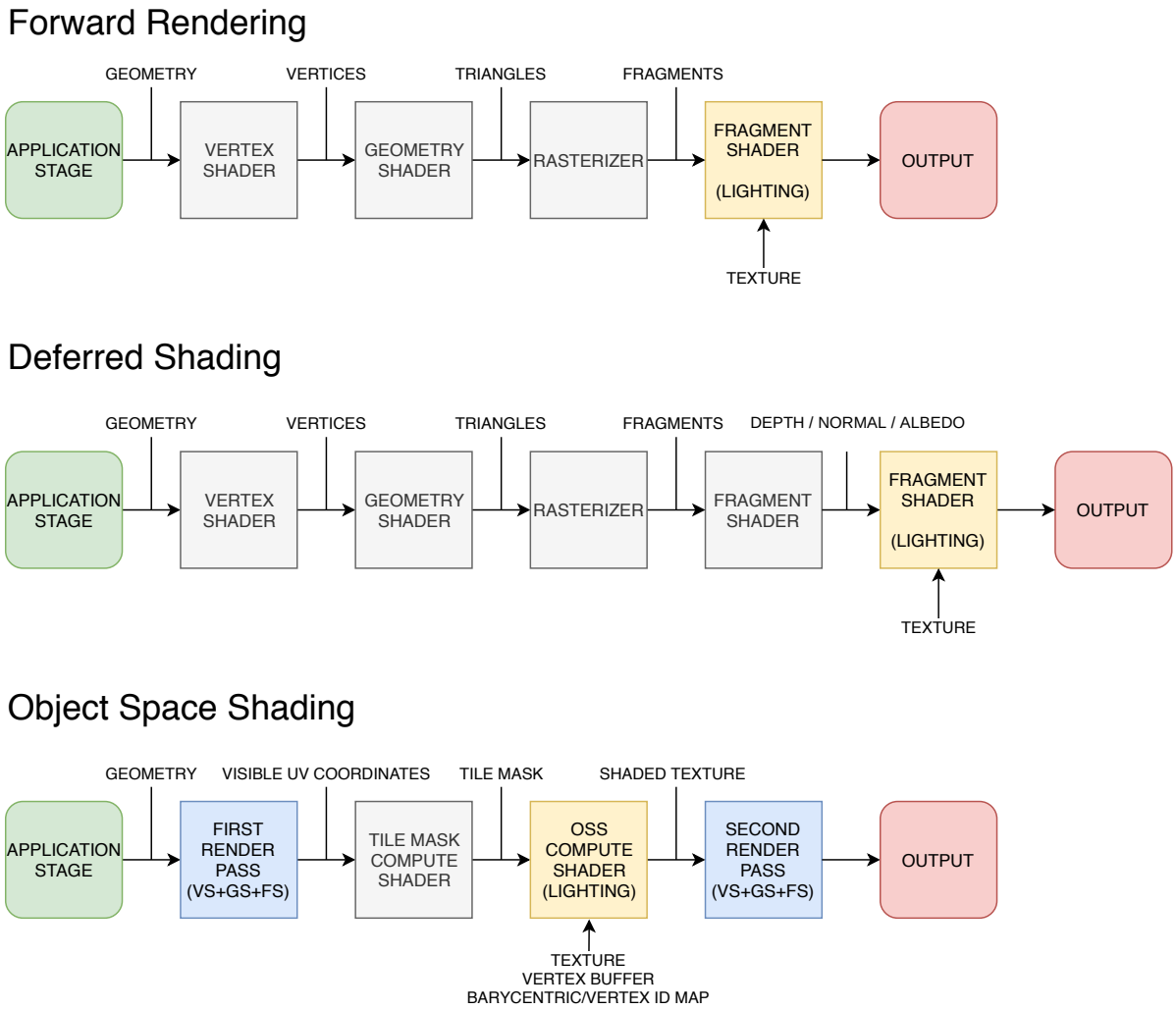

Figure 2.3.: Comparison of the different stages of forward, deferred and object space rendering based on: `http://www.alexisbreust.fr/2018-deep-deferred-renderer.html` (visited on: 11.02.2019)

2.2.2. Disadvantages of Object Space Shading

Of course, this method is not without drawbacks. Most of them are a result of the extra work we have to do before we shade.
The first disadvantage is that we add an extra render pass, which reduces the per frame performance. In a more advanced render pipeline, this might be cleverly combined with the second pass, reducing the cost slightly, but for now, we have to accept the performance loss. [Hil16]
There is also the problem of the additional memory, that is needed to store all the extra information. Per fragment data would be calculated right before shading and not stored for a long time since we do not need it after that anymore. But since we do not work with per fragment data, we have to use a lot of video memory for data we need for shading. The memory usage could be reduced by a more advanced memory allocation algorithm, but will probably still be considerably higher than the memory usage of other more common render methods. [Hil16]
A Forward or Deferred Renderer would only shade what can be seen on screen. This is one advantage of shading in the mathematical space of the screen. Parts of objects that are occluded, and therefore not visible, are not shaded. By doing a first render pass, we try to shade as few occluded pixels as possible, but since we work with tiles we will always do a bit extra work. We try to measure this in section 5.2. [Hil16]

2.2.3. Advantages of Object Space Shading

But of course, we would not try Object Space Shading if it would also have numerous advantages, that hopefully outweigh the drawbacks.
For starters, the easiest way to gain back some of the performance we lost is to decouple the shading rate from the render frequency. Other shading methods are depended on the rasterizer for their shading, and therefore have to calculate it new every frame. Object Space Shading, on the other hand, could theoretically reuse results for one or two frames without having a significant impact on the image quality. This would drastically reduce the computational costs. How much of each frame can be reduced is evaluated in section 5.4. [Hil16]
The same could be done for the screen resolution. Shading does not have to happen at the same resolution as the screen. The shading resolution could be lower and then scaled up, to increase performance, or higher and scaled down, to improve the image quality. [Bak16]
Another advantage is that the shading is decoupled from the scene's geometry. This is helpful because other render methods have wastage problems with small triangles. This does not apply to Object Space Shading since we shade the texture and do not really care about triangles. [Hil16]
We can also reuse shaded results across multiple virtual cameras. This is especially useful for Virtual Reality or multi-user applications. Other renderers would have to compute everything again, but since we already store every result in a texture, we can just use that when needed. [Hil16]

The last important advantage I want to mention is that it avoids some of the visual artifacts, that are caused by shading in screen space. This was a major reason why Pixar's Renderman Software uses a similar approach to rendering. [Bak16]

2.2.4. View-dependent and View-independent Shading

One more important thing to talk about is how to handle view-dependent shading. While view-independent shading, like diffuse and ambient shading, stays the same from every camera angle, other shading, like specular, does not. This means that, in order to enjoy many of the advantages of Object Space Shading, an Object Space Shading render pipeline would have to handle both of them separately. Otherwise, it could not be used for something like VR or multi-user applications. Even using it for multiple frames in a row is difficult for view-depended shading since the camera is constantly moving in most cases. It might be possible to use it for one or two frames without noticeable effect, but then it has to be calculated new. View-independent shading, on the other hand, can be used as long as the lighting or object position does not change. There are multiple solutions for this. For one, we could use Object Space Shading only for view-independent shading. We could then combine it with a Forward Shading in the second render pass for other shading like adding specular highlights. We could also store both results in different textures, and update the specular one more frequently. Either way, we have to do something in order to get maximum performance out of this.

2.3. Unity

2.3.1. Basics

Unity, also named Unity3D, calls itself the "the world's leading real-time creation platform"[1]. That may be a bold claim, but they have the numbers to support it. In their "Games by Numbers Report" from Q1 2016, they announced that there are over 220,000 games made with Unity, running on more than 1.7 billion devices [2]. This, of course, also includes a huge number of mobile devices, such as iPhones and Android phones.
So what is Unity? "'Unity' is an umbrella term referring to a suite of technologies and tools whose collective purpose is to help create interactive software products, most notably games [...] but also augmented reality applications and simulators. Unity, in short, is game-making software." [Tho14, p.8]. This does not mean that Unity already includes every tool that is needed to create a game. It is not an asset creation tool. Rather it uses assets created by other software, such as Blender, and combines them to a full game. These assets can include 3D models, sound effects, textures, sprites, objects, and many more [3]. Unity mainly consists of three parts: a visual editor, an underlying game engine, and publishing modules [Tho14,

[1] `https://unity3d.com/de/unity` (visited on: 11.02.2019)
[2] `http://response.unity3d.com/games-by-the-numbers-q1-2016-report` (visited on: 11.02.2019)
[3] `https://gamedevelopment.tutsplus.com/articles/how-to-fund-your-games-by-creating-and-selling-game-assets--cms-24380` (visited on: 11.02.2019)

p.9]. The visual editor is a quick and easy way to create and manipulate game scenes and objects, without having to know any coding language. But in contrast to many other game engines, Unity does not limit its scripting support with its visual editor [Hoc15]. Everything can still be controlled via C# or JavaScript code. The engine does all the actual work. It is the logical core in any game made with Unity. This includes rendering, object management, player physics, input handling, and so on. Everything that is needed to run a game. The last part, the publishing modules, allow a game made with Unity to be deployed on many different platforms, such as Windows, Mac, Linux, Android, iOS and many more, following Unity's motto 'Build once, deploy everywhere' [Tho14, p. 10].
Unity's high customisability and easy C# integration make it perfect for proofing that Object Space Shading can be implemented into an already existing game environment without too many obstacles. Its simple access to custom shaders for each game object and its new Scriptable Render Pipeline give us the tools we need for this project.

2.3.2. Scriptable Render Pipeline

In early May 2018 Unity released its version 2018.1.0 [4] . This version includes a new feature, called "Scriptable Render Pipeline" (SRP). The SRP allows the user to freely customize the rendering process via a simple C# script and replace Unity's already build in renderer with a custom pipeline. The user can decide when and how to do cull, set or replace shaders for each individual object, call multiple different shader passes, change or manipulate object textures, add post-processing effects, and so on. That way the render pipeline can be fitted to the needs of the intended platform, using only simple features for low-performance hardware or adding more compute-intensive effects for high-end systems. This also allows us to implement our own render method, which is useful for the project that is part of this thesis. The Scriptable Render Pipeline already includes two render pipeline assets, that can be used as a template for one's own pipeline, the Lightweight Render Pipeline (LWRP) mostly for mobile devices and the High Definition Render Pipeline (HDRP) for more powerful systems. Alternatively, a custom render pipeline can be written from scratch. This, however, requires a dedicated graphics programmer in most cases.[5]

[4] `https://unity3d.com/de/unity/roadmap/archive` (visited on: 11.02.2019)
[5] `https://blogs.unity3d.com/2018/01/31/srp-overview/` (visited on: 11.02.2019)

3. Related Work

In this chapter, I take a look at already existing implementations of Object Space Shading. This includes "Ashes of the Singularity", which is a computer game using this render technique, and a short paper about Object Space Shading.

3.1. Ashes of the Singularity

Object Space Shading has already been successfully implemented into one commercially available game. This game is called "Ashes of the Singularity" and was released in 2016 by Oxide Games and Stardock Entertainment. It is a futuristic real-time strategy game based on Oxide Games own Nitrous Game Engine. This engine is what allows the game to use Object Space Shading. Dan Baker, one of the founders of Oxide Games, described their implementation of this render technique closer in his presentation during the Game Developers Conference in San Francisco in 2016 [Bak16].
The studio knew relatively early that they want to try this approach, even though they had concerns about not being able to make it efficient enough. They were inspired by CGI movies, that used similar techniques already 20 years ago. With the difference, that CGI movies do not have to be rendered in real-time and are therefore not as dependent on performance as computer games. But since graphics hardware improved significantly during the last two decades, it is now possible to also try such an approach for real-time applications like games. Pixar used a similar technique, which they called "REYES", in their Renderman rendering software. The reason this was used is to avoid any kind of artifacts that are caused by anti-aliasing, a technique that makes faraway objects look better on the screen. [Bak16]
Even though the Nitrous Engine uses in most parts the Object Space Shading algorithm that was described in section 2.2, it does a few things differently. For starter, they do not check which parts of the textures are visible on the screen, at least at the time of Dan Baker's presentation. Instead, they shade the whole texture. They do, however, cull objects that are not visible, but this only affects objects which are completely outside of the screen space. This caused a lot of problems for large objects, like terrain, for which they had to find a workaround. Also, they do not calculate the vertex position and normal vector during shading, using the vertex buffer and texture maps containing vertex IDs and barycentric coordinates, like in the project accompanying this thesis. Instead, they calculate them before the beginning of the render process, usually during asset creation, and store them in textures. This increases performance by reducing the number of necessary computations during shading. Lastly, instead of having one texture atlas per object, they use one large, combined texture sheet for all of them. This has a number of benefits. First of all, it greatly reduces the amount of

video memory needed. It also acts as a hard upper limit for the amount of shading that is done each frame. If more objects have to be shaded, everything gets scaled done and shaded at a lower resolution. This was not implemented in this project, because it requires a very complex allocation algorithm for the texture atlas. [Bak16]
Oxide Games claims to get excellent performance out of their engine while still having many benefits thanks to Object Space Shading. First of all, the texture sheet limits the amount of shading samples each frame, which help to keep performance stable. They can also reduce the shading frequency, without noticeable effect on the image quality. The frequency could, for example, be reduced to 30 times per second, while still rendering at 60 frames per second, which would further increase the performance. This is especially useful for VR where the frames per second usually have to be higher. Another benefit they found is, that they can shade far away objects at a higher resolution than they would be shaded normally, and then use anti-aliasing techniques to deliver a better result. Lastly, in case Object Space Shading is not the right choice for some objects in the scene, they can bypass it and use normal forward rendering techniques.[Bak16]
This thesis takes the method described by Dan Baker and adds an algorithm to determine the parts of the objects that are actually visible, and therefore reduces the amount of unnecessary shading work done. It also tries to measure and quantify the benefits this gives us, and what extra costs arise.

3.2. "Texel Shading" Paper by K. Hillesland and J. C. Yang

"Ashes of the Singularity" might be the only commercially available game so far that uses this render method, but there are scientific papers about Object Space Shading. Karl Hillesland and J. C. Yang wrote such a paper in 2016 [HY16], roughly at the same time "Ashes of the Singularity" was released. In this paper, they describe a different approach that is closer to what I am doing in this thesis. For one, they also do a first render pass to determine the visible parts of the scene and avoid shading occluded objects. And they too shade tiles and not single pixels. They also shade across different mip level, something "Ashes of the Singularity" does not. This is possible thanks to the additional render pass performed at the beginning of each frame. Otherwise, it would be hard to determine the correct level. Shading across different mip levels gives us additional performance, as we do not need to shade faraway objects at such a high resolution. This would be very inefficient, especially considering that we always shade whole tiles. The paper focuses on using a variable shading rate and tests this among three, smaller scenes with multiple lights. They only tried a tile size of 8x8 pixels. Their results show that an Object Space Shading render pipeline can be more efficient than regular fragment shading if a variable shading frequency is used, especially if there are many lights in the scene. This thesis, on the other hand, will focus on the influence of the tile size and use more complex scenes. It also wants to show that this approach can be integrated into a Game Engine like Unity.

3.3. "A Lazy Object Space Shading Algorithm With Decoupled Sampling" by C. A. Burns, K. Fatahalian and W. R. Mark

Another paper researching Object Space Shading is "A Lazy Object Space Shading Algorithm With Decoupled Sampling" written by C. A. Burns, K. Fatahalian and W. R. Mark in 2010 [BFM10]. At that time computer hardware was not efficient enough to perform Object Space Shading at real-time, but offline implementations like Pixar's Renderman Engine already existed. The paper tried to improve some of the flaws of this Engine by using pixel sized samples of the surface instead of using micro-polygons and shading at the vertices. The main goal of this was to bring accurate depth-of-field and motion blur image effects to real-time render engines. They furthered this goal, by drastically reducing the amount of polygons they have to shade compared to the Renderman Engine, but could not reach it due to hardware limitations at that time.

4. Project Implementation

This chapter describes my implementation of Object Space Shading. In the following sections, I first explain how my own render pipeline is integrated into the Unity Game Engine via its Scriptable Render Pipeline feature. Then I write about how game objects and assets have to be prepared in order to be used in this implementation, and what requirements they have to fulfill. Thereafter I illustrate how my pipeline works, what has to be done at the beginning of the program, and what happens during each frame.

4.1. Scriptable Render Pipeline

This project was built in Unity 2018.2.10f1, as this was the newest version when I started. In a previous version, this project would not have been possible, since Unity's "Scriptable Render Pipeline" was only introduced earlier in 2018, in version 2018.1.0. This feature can simply be added via the integrated package manager, even though it is still in an experimental phase.
Once the package is imported one can either choose to use one of the already given render pipelines and use them as a template to build upon, or to build a new pipeline from scratch. In this project, I went for the latter, following a very helpful tutorial from a website called "Catlike Coding" [1], that explained the basic functionality of the Scriptable Render Pipeline very well. The SRP is essentially a C# script that derives from the class "RenderPipeline". It can control the rendering process by overwriting the inherited render method. Once a custom pipeline is created it can be used to generate something called a "Render Pipeline Asset" which then can be chosen in Unity's graphics setting as the current render pipeline. This might, however, break parts of Unity's visual editor, like the scene preview window, which is more of an inconvenience than a real problem.

4.2. Asset Requirements

The object in the game scene have to meet a few requirements to be used here, mostly in regards to their 3D meshes and their textures. Some of these are due to the unoptimized state of the project's implementation, but some of them are due to the limitations of Object Space Shading.
One requirement, that could easily be solved if the project would be continued is that every texture has to have the same size. This size is currently set to 4096 pixels in height and width to account for more complicated objects, but could quickly be changed if needed. This was

[1] `https://catlikecoding.com/unity/tutorials/scriptable-render-pipeline/custom-pipeline/` (visited on: 11.02.2019)

done to simplify the render pipeline and focus on more important features. Also, the textures have to have the same height and width for a similar reason.
Another limitation, that is caused by the lack of performance and memory usage optimization, is the upper bound for the number of objects that can be in a scene. The exact number depends on the available video memory and the already mention fixed texture dimensions. The reason for this is that the textures and other data stored during the rendering process fill up the video memory and the renderer has to use other, slower memory if there are too many objects. This could be solved with a better, more efficient use of memory by allocating only the space needed during each frame.
The last requirement is that no object can use any part of its texture twice. Every UV coordinate has to be unique. This is due to limitations in Object Space Shading, since it is done in texture space, and no part of the texture can or should be shaded more than once. This might be solvable, but would require a complex texture allocation algorithm.
If an object fulfills all of these requirements it can be used in the render pipeline described below.

4.3. Render Pipeline

4.3.1. Preparations at the Beginning of the Program

One problem with Object Space Shading is, that vertex normals and positions are not easily available when they are needed during the shading process, as these are usually interpolated from the vertex buffer during rasterization. Since we do not shade during rasterization, as we would with other shading techniques, that data needs to be prepared before the first frame can be rendered. This can either be done beforehand and saved as part of the asset at the cost of storage space, or, like in this project, calculated at the beginning of the program. This makes it easier to add new objects to the scene but requires a few additional calculations before the actual rendering can start.
The information we need to calculate for each individual pixel in the object's texture are the ids of the vertices of the triangle the pixel is located in and the pixel's barycentric coordinates within this triangle. With this additional data, position and normals can be interpolated for each pixel, using the formula below:

$$P_{ij} = V[ID_{ij}.x] * B_{ij}.x + V[ID_{ij}.y] * B_{ij}.y + V[ID_{ij}.z] * B_{ij}.z$$

P	Position or Normal
(i, j)	Position of pixel in texture
V	Vertex Buffer containing information about vertex positions or normals
B	vector containing barycentric coordinates
ID	vector containing vertex ids

To get this data we first have to render each object individually, using a replacement shader.

This shader unwraps the object's mesh during the vertex shader stage so that any information about the object can be rendered into UV space. Then, during the geometry shader stage, each of the vertices of the triangle gets assigned a vector of size three containing the vertex ids of each of the three vertices, and a vector containing the barycentric coordinates of that vertex. These coordinates get interpolated during the rasterization and stored into a texture together with the vertex ids during the fragment shader stage. The resulting image for the barycentric coordinates of a standard sphere in Unity can be seen in figure 4.1.
After this has been done for each object the newly calculated data can be used to interpolate the normal and the position using the available information in the vertex buffer. This will be done during each frame for the actual shading.
Also, each object in the scene gets assigned its unique object id during the preparation step.

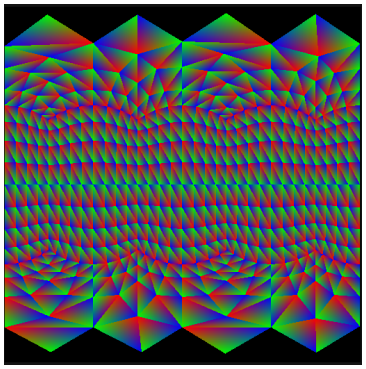

Figure 4.1.: Barycentric coordinate texture for a sphere

4.3.2. Rendering during each Frame

The rendering that happens during each frame can be split into four parts. In the first step, the first render pass, the scene's visible UV coordinates and mip level are rendered into a texture. In the second step, a compute shader uses these coordinates to create a mask for each individual object that displays which parts of the object's texture are visible in the scene. Each pixel of the mask represents one tile that covers multiple pixels of the object's texture. The next step uses this mask to only shade the visible parts of the textures. In the last step, the scene is rendered again in a second render pass using the shaded texture, delivering the final image that will be displayed on the screen.

First Render Pass

As already mentioned the first shading pass renders the scene using a special shader, with the resulting image displaying the UV coordinates of each object that is visible on screen. The image is stored in a dual channel floating point texture with 32 bits per channel, as precision is necessary to avoid visual artifacts in the end result. That shader also outputs an image containing the mip level and ids of each object. The id is needed to match the rendered UV

coordinates to the correct object during the next step. The mip level is used to shade the texture later at the correct level. It is calculated using the code below.

```
float2 dx = abs(ddx(i.uv));
float2 dy = abs(ddy(i.uv));
float mipLevel = log2(max(max(dx.x, dx.y), max(dy.x, dy.y)) * TextureSize);
```

Figure 4.2.: Mip Level Code

That code is called in the pixel shader of the first shading pass, and then later reused during the second pass. It needs the current UV coordinates, which are provided by the rasterizer, and the size of the object's texture as input to compute the correct mip level. In the current version of the project, the texture dimensions are fixed to a certain size and have to be changed in code if the texture size changes, but that is not a requirement for Object Space Shading.
The results of the first stage can be seen in figure 4.3. This image would then be passed on to the next stage, where a mask for each object's texture will be created. The final results of the render pipeline can be seen in figure 4.4.

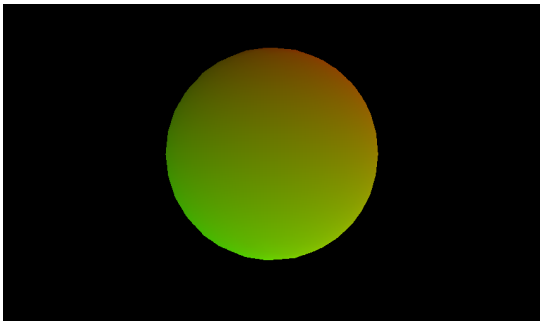

Figure 4.3.: UV Coordinates Earth Sample Scene

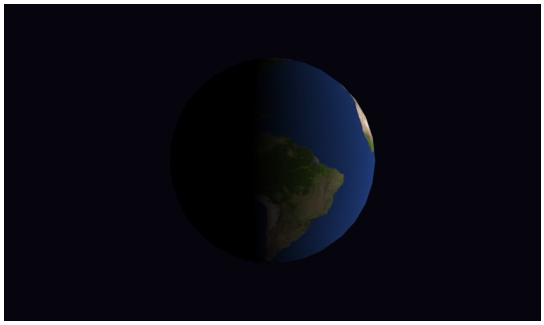

Figure 4.4.: Shaded Earth Sample Scene

Tilemask Shader

The results of the first pass are used in this step of the pipeline to create a mask for the texture of each object. These masks contain information about which part of the texture can be seen on screen. The mask doesn't have to be the same size as the texture. In fact, each pixel of the mask usually covers a texel or tile, containing multiple pixels of the texture. The size of the tiles can be chosen before the program starts. The performance variations between different tile sizes will be evaluated in chapter 5. A single channel 8-bit texture is used for the mask, as this was the smallest texture format available in Unity and every pixel of it only needs to contain either a 0, if that part of the texture cannot be seen on screen, or a 1, if it can be seen. Since the mask also needs to contain this information for each individual mip level, it is

essentially a texture atlas, with sections allocated for each level. Figure 4.5 shows how space in the mask is used. The formula for the size of the tile mask can be seen below.

$$mask\,width = \frac{texture\,size}{tile\,size} * 2$$

$$mask\,height = \frac{texture\,size}{tile\,size}$$

An example for a finished tile mask can be seen in figure 4.6.

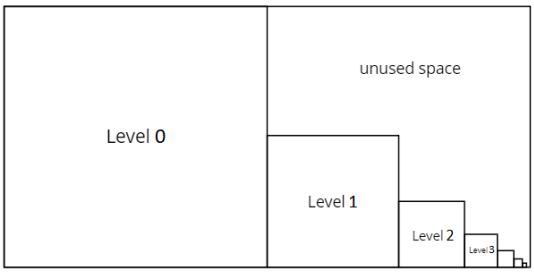

Figure 4.5.: Space use in texture atlas/tile mask

Figure 4.6.: Tile Mask Earth Sample Scene

To create these masks, we first have to clear them each frame, because they might still contain the information from the last frame. After that we let a compute shader run for every object, each of them iterating over every pixel in UV image we created in the first render pass. If a pixel contains the same id as the object we are currently looking at we read the UV coordinates from the image and set the respective part of the mask to 1. If not we do nothing and continue with the next pixel.
When every tile mask has been completed we can continue with the next step.

Object Space Compute Shader

After a tile mask for an object is finished a new texture atlas is created. This atlas is used to store the shaded textures after this step. The size is double the size of the texture, as it also has to contain the shading results of each mip level. It follows the same space allocation rules for the different mip level as the tile mask. These can be seen in figure 4.5.
After the atlas has been created another compute shader is called for each object. This shader does the actual shading work. It iterates over every pixel of the atlas. If the respective pixel in the mask is set to 0 it skips that pixel and continues with the next one. If the pixel is set to 1 the compute shader starts shading that part of the texture. Before the shader can do that it first needs the world position and object normal at that point of the texture. For that, it uses the barycentric coordinate texture and the vertex id texture that were created during the preparation stage of the render pipeline. These are passed to the computer shader together with the object's vertex buffer, containing vertex positions and normals, and the object's local-to-world matrix. Using this information the shader can interpolate the pixel's

local position and normal and then transform them to world space.
With this data available the actual shading can start. This project only uses a very simple ambient and diffuse shading after the Phong Lighting Model [Pho75]. The formula used for that can be seen below.

$$I_{ambient\,object} = k_{ambient} * I_{ambient\,global}$$

$$I_{diffuse} = k_{diffuse} * (\vec{N} \cdot (\vec{P}_{light} - \vec{P}_{object})) * I_{light}$$

$$I_{out} = I_{ambient\,object} + I_{diffuse}$$

I	Light Intensity
k	Reflection factor
N	Object Normal
P	Position

Figure 4.7.: Phong Lighting Formula (without specular reflection) [Pho75] and [Lyo93]

A more sophisticated shading could be done, but is not necessary here, as this project's goal is not to produce the prettiest image, but rather to measure the performance costs and improvements of Object Space Shading.
The light intensity calculated with the Phong Lighting Model is combined with the color value of the object's texture and saved into the texture atlas. What the finished texture atlas for a single object might look like can be seen in figure 4.8.

Figure 4.8.: Shaded Texture Atlas for Earth Sample Scene

Read-Back Shader

After the shading is done the only thing that is left to do is to render the scene again in a second render pass. This is done by using a very simple unlit shader that only renders the now shaded texture back to their respective objects. The shader uses the same code to determine mip level as the first render pass. The code is displayed in figure 4.2. The final

image is not rendered directly to the screen since this produced undesirable visual artifacts. Instead, it is rendered to a separate texture that is then copied to the screen. The artifacts seem to be caused by DirectX 11 rendering textures upside down, which results in slightly different mip levels.
When this is done the render pipeline is finished and can start rendering the next frame.

5. Evaluation

In this chapter, I evaluate the performance costs and improvements of my approach to Object Space Shading. For that I set up three different tests, one outlining the added computational costs, one measuring the amount of shading work that can be reused between different camera angles, and one showing what can be reused between two frames of the same moving camera. I will perform each of these tests in two different scenes, which I pre-processed for my render pipeline. At last, I will evaluate the resulting statistics to answer the question if Object Space Shading is a feasible approach to shading.

5.1. Preparation of Example Scenes

To evaluate the performance of the Object Space Shading render pipeline implemented in the project accompanying this thesis, test scenes of sufficient size were needed. Two already existing sample scenes commonly used for testing and showcasing render methods were chosen. The first one is a replica of the atrium of the Sponza Palace in Dubrovnik, Croatia[1]. It was made by the company Crytek for testing their own game engine, but is available for others too[2]. The scene contains about 200,000 vertices that form roughly 250,000 faces. The other scene that was used is called "Lost Empire", and contains a 3D model of a map taken from the game "Minecraft" [3]. It was built on the Minecraft server "Vokselia". The model contains almost 450,000 vertices that form over 220,000 faces.
Both of them had to be modified in order to be used for this project, as they did not conform to the requirements explained in section 4.2, and could therefore not be shaded with this render pipeline. Most of the required changes were done using the free and open source graphics software Blender. At first, the number of objects in each scene had to be reduced, because the amount the render pipeline can handle is limited by the available video memory, as already explained. This was done in two ways: first, by removing some of the detail in both scenes, and second by joining together multiple smaller objects into one big object. The next thing that had to be done was to give each face in the scene a unique UV coordinate, which is, as mentioned before, a necessary requirement for Object Space Shading. This was also done using Blender. Unfortunately, this meant that the textures of both scenes could not be used anymore, at least not without considerable effort. The textures had to be replaced with something else, as they are needed for the render pipeline. For the "Sponza" scene a map of the scene's UV coordinates, automatically generated by Blender, was used. For the

[1] `https://github.com/trsh/Armory3DSponza/tree/master` (visited on: 11.02.2019)
[2] `https://www.cryengine.com/marketplace/sponza-sample-scene` (visited on: 11.02.2019)
[3] `https://casual-effects.com/g3d/data10/` (visited on: 11.02.2019)

"Lost Empire" scene new textures containing very simple checkered patterns in the colors green, brown and grey were created, using a simple, self-written code.
With these modifications done the scenes were imported into the Unity project, and each object was assigned the correct shader and texture. Images of the scenes can be seen below. In the pictures, they are already shaded using the render pipeline implemented in this project.

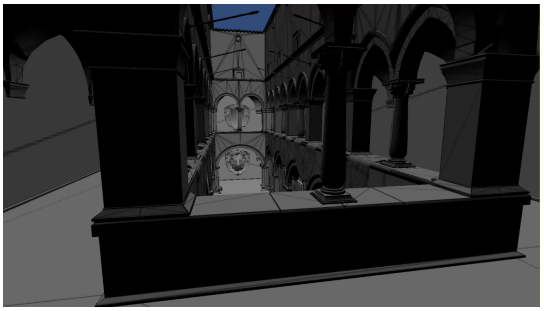

Figure 5.1.: Shaded Sponza Sample Scene

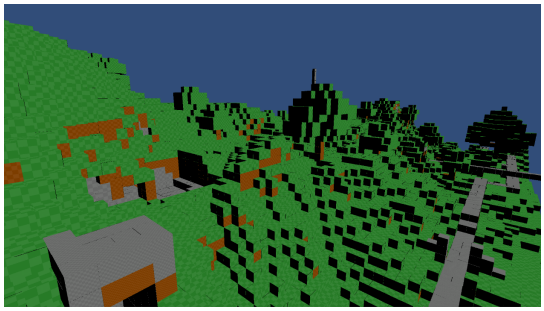

Figure 5.2.: Shaded Earth Lost Empire Sample Scene

5.2. First Test: Measuring extra cost

5.2.1. Setup

The first test measures the extra costs that arise through the use of Object Space Shading. As the render pipeline in this project is not performance optimized we cannot determine all the overhead costs. We can, however, count the amount of pixels shaded by the Object Space Shader, and put them in relation to the number of pixels we would have to shade, if we would use a more traditional shading technique, like Forward Rendering. This gives us a good idea of the scale of the extra costs. The amount of pixels shaded by the pipeline is obtained by counting the 1s in the binary tile mask of each object after the tile mask shader step, adding them all up and multiplying them by the amount of pixels a single tile covers. Each tile covers an area of its size squared. The number of pixels a traditional renderer would have to shade is the same as the number of pixels on the screen. The formula that was just described can be seen below.

$$ratio = \frac{pixels\,shaded\,in\,OSS}{pixels\,shaded\,normally} = \frac{\sum_{o \in Objects} t_o * tilesize^2}{screenwidth * screenheight} \tag{5.1}$$

t number of 1s in tilemask

This calculation is performed for every frame of a moving camera. This camera movement consists of about 600 frames in each of the two scenes.
For the "Sponza" scene, the movement starts on the upper floor of the palace, with the camera

looking towards the atrium. The camera then follows the hallway, still looking towards the center of the scene, with pillars moving in and out of the camera's view. In the end, the camera turns toward the hallway.
In the "Lost Empire" scene, the movement starts with the camera centered above the scene, flying downwards. After it has reached the ground, the camera enters a very narrow hallway, just to emerge on the other side in a giant cave. The movement finishes with the camera looking towards the center of the cave.
This test was performed multiple times. For each of the tile sizes 8, 16, 32 and 64 pixels in different screen resolutions. The resolutions used were 800x600, 1024x768, 1920x1080 and 3840x2160, covering a broad range.

5.2.2. Results

Two exemplary graphs with the results of this test can be seen below.

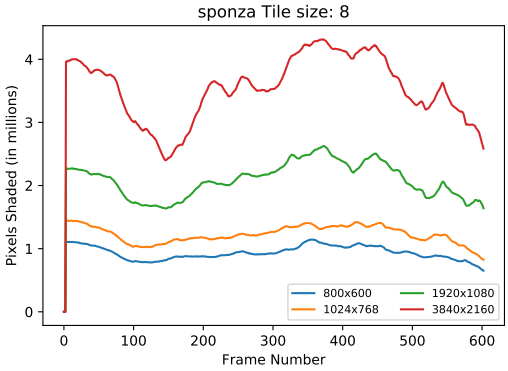

Figure 5.3.: First Test, Sponza, Tile size: 8

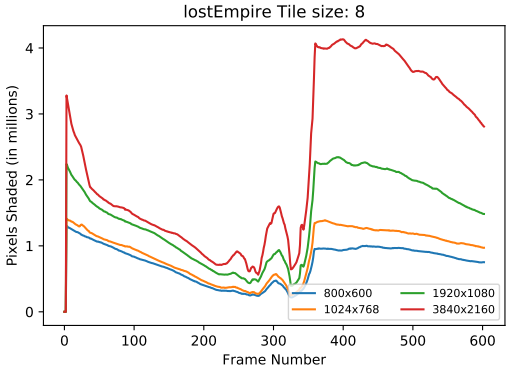

Figure 5.4.: First Test, Lost Empire, Tile size: 8

These graphs show the amount of pixels the Object Space Renderer shades each frame. The scenes were shaded with a tile size of 8 pixels in four different resolutions. Graphs for other tile sizes are not shown here, as they look very similar, with just different labels on the y-axis. They can however be found in appendix A.1.
These graphs already show that the number of pixels shaded can vary widely when performing Object Space shading. This can be seen in figure 5.3 around frame 150, where a decline is visible, which was caused by a huge pillar moving in front of the camera. But it gets especially clear when looking at figure 5.4, where in the beginning there is a slight decline in the number of pixels shaded that is caused by the camera getting closer to the ground and therefore seeing less of the scene, and then a sudden drastic increase around frame 350, when the camera enters the cave, where a considerable amount of objects can be seen. This means that an Object Space Shading renderer may be very well suited in some cases, where the visible surface area is small and uniform, but could potentially cause problems in cases where are a lot of small, delicate objects have to be rendered.
The results shown in the graphs can now be used to calculate the ratio of pixels shaded with

an Object Space Shading renderer to the number of pixels that would be shaded with another render technique for each frame. This is done by applying the formula that can be seen in equation 5.1. After this, the median of all frames corresponding to one tile size and one screen size is taken for both scenes. The results can be seen in figure 5.1.

		sponza		lost empire	
tile size	screen size	OSS/Standard	std. deviation	OSS/Standard	std. deviation
8	800x600	192%	7%	172%	36%
	1024x768	156%	4%	130%	22%
	1920x1080	102%	2%	73%	9%
	3840x2160	43%	0%	21%	2%
16	800x600	315%	21%	274%	94%
	1024x768	244%	12%	188%	55%
	1920x1080	153%	5%	98%	20%
	3840x2160	60%	1%	27%	4%
32	800x600	630%	85%	487%	342%
	1024x768	469%	50%	307%	199%
	1920x1080	278%	18%	152%	67%
	3840x2160	101%	3%	41%	11%
64	800x600	1393%	340%	901%	1430%
	1024x768	1019%	203%	562%	822%
	1920x1080	572%	66%	263%	267%
	3840x2160	199%	12%	72%	40%

Table 5.1.: Results of the first test

'OSS/Standard' shows the amount of pixels shaded by a OSS Renderer in relation to what a standard render would have to shaded.

It is already very clear from these results that both the tile size and the screen size have a huge influence on the performance costs. The results also differ between scenes, as already discussed above.

Influence of the Tile Size

As expected from the very beginning, the tile size impacts the results a lot. To measure how big the influence of the tile size on the result is was one of the reasons the project was implemented. The extra costs that arise during shading with a small tile size might, depending on the results of the other tests, still be reasonable, with the costs not more than doubling for a size of 8 pixels. The extra costs of bigger sizes on the other hand seem to be

too high, with the costs for a size of 64 pixels being up to 14 times that of other shading techniques. The reason for that enormous rise in unnecessary shading work is that a lot of the tiles are only partially visible on screen. Bigger tiles of course are more likely to not be completely displayed, and cause more additional work, as they cover a bigger area, and therefore add more unnecessary pixels that have to be shaded, even though only a small part of the tile is actually visible. Small tiles on the other hand produce a smaller overhead cost. But without the additional results from the other tests it cannot be definitively said that smaller tiles perform better than bigger tiles.

Influence of the Screen Resolution

The other factor that influences the costs is the screen size. The data depicted in figure 5.1 clearly indicates that the extra performance costs is far higher for low-resolution screens. This is because the number of tiles visible grows at a much smaller rate than the amount of pixels on screen. Also very noticeable is that the Object Space Shading renderer is actually more efficient than a normal renderer for very high resolutions. But this is only because the rendered scene is heavily undersampled in that case, meaning that the resolution of the object textures are too low, which results in multiple pixels on the screen displaying the same pixel of one texture. This may be helpful in some cases, but ideally the textures should have a high enough resolution, to produce the best possible image.

Impact of the Scene Geoemtry

The scene itself has obviously also some influence in the performance. Figure 5.1 shows that the additional costs for the "Sponza" scene is slightly higher. On the other hand, the actual performance in this test fluctuates more heavily in the "Lost Empire" scene, at least in this test. This is especially clear in figure 5.4.

Conclusion

In conclusion, it can only be said that the additional performance costs vary widely depending on screen size, tile size and the objects that are currently visible on screen. Whether this render method is a feasible approach has to be decided for the specific use case, and depends on the results of the next tests, that will be discussed in the next sections.

5.3. Second Test: Comparing different camera angles

5.3.1. Setup

In the second test, the scenes are rendered from numerous angles, using multiple cameras. This is done to measure the overlap of tiles between both cameras. Each of them is placed at a different position in different parts of the scene. All of them look roughly at the same area, otherwise, there would not be much overlap. The purpose of this test is to show that Object

Space Shading might be useful in situations where there is more than one camera in a scene. Tiles that are visible through multiple cameras could be shaded only once and then be used for all of them, saving unnecessary shading work. Situations, where this could be used, are applications including multiple users, like online games that let the server do some of the shading, to run more efficient on lower-end devices, or games allowing several players to play together locally on one machine. That does also includes Virtual Reality (VR) games, where the image has to be rendered twice from a very similar position, to create that 3D effect. This, however, was not specifically tested here, as all cameras used in this test are further away from each other than they would be in VR applications. But the overlap would most likely be very high since both pictures in VR look very similar.
The placement of the cameras in the "Sponza" scene is depicted below in figure 5.5. This figure also includes the IDs of the cameras and arrows indicating the direction each of them is looking at. Five cameras were used in total. All cameras in this scene except camera number 4 are pointed towards the center of the atrium. Number 1 is the only one that is not located on the upper level of the scene. Instead, it is placed on the bottom of the atrium looking up towards the upper floor. Since many of the cameras are placed within the arcade, most of their view is blocked by pillars.

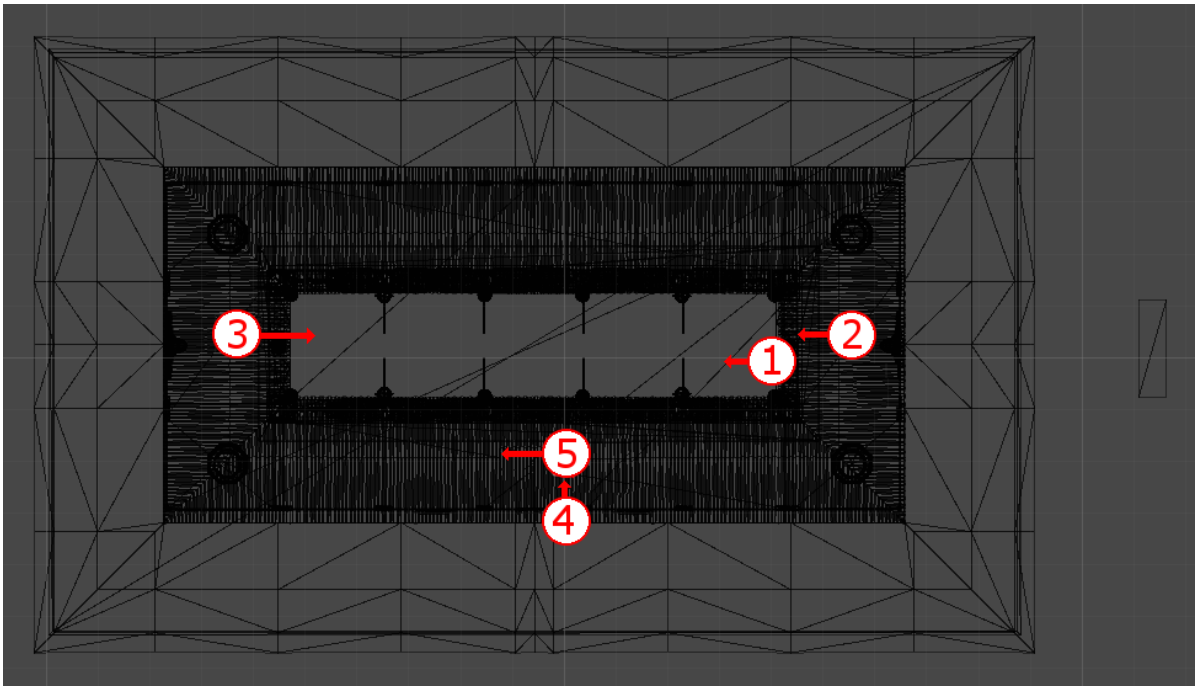

Figure 5.5.: Placement of the cameras for the second test in the "Sponza" scene

For the "Lost Empire" scene all of the cameras were placed somewhere on the edge of the scene, also pointing towards the center, as can be seen in figure 5.6 below. Again this figure includes the camera IDs and arrows indicating the direction. Five camera positions were used, similar to the "Sponza" scene. Camera 1 is placed above the scene looking downwards,

capturing an aerial view. All other cameras are placed somewhere close to the ground, on points where there is not much obstructing their view.
This test was also performed multiple times using the same tile sizes and resolutions as in the first test. The tile sizes used are 8, 16, 32 and 64 pixels. The resolutions used are 800x600, 1024x768, 1920x1080 and 3840x2160 pixels.
No actual calculations for this test were done during the rendering process. Instead, all tile masks were saved and evaluated later. This resulted in over 2000 masks for all tile sizes, all screen resolutions, both scenes and all objects in them. Comparing the masks and evaluating the result was then done using Python, as this is the easiest way to work on the data and visualize it.

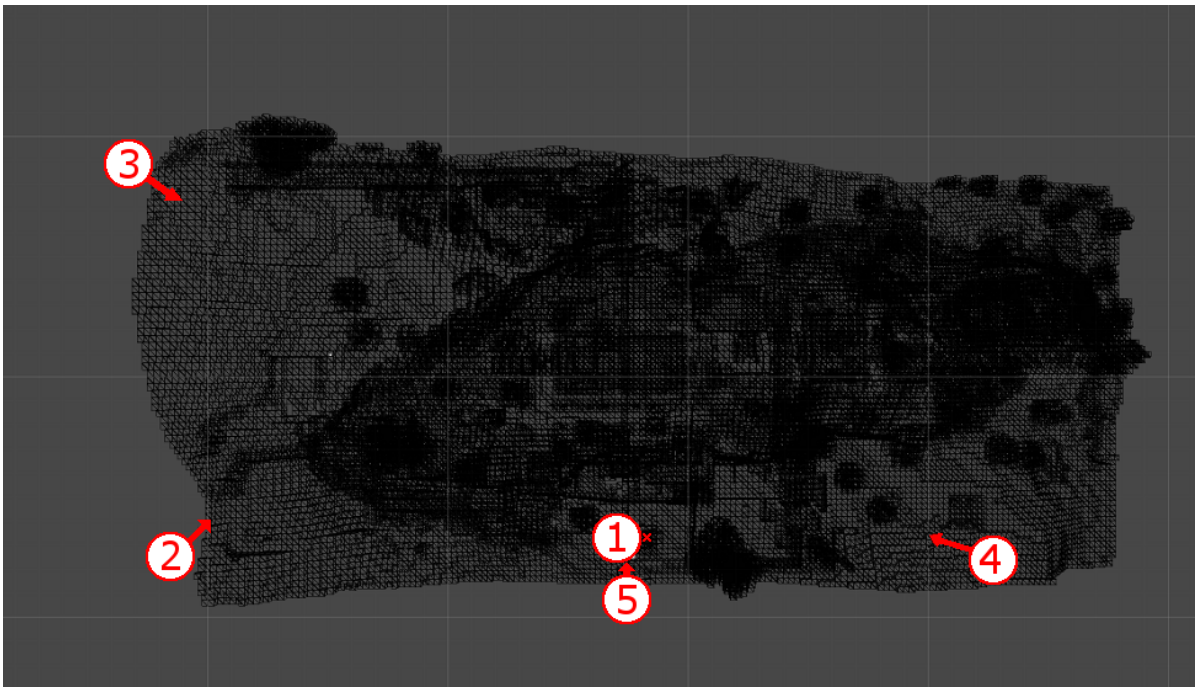

Figure 5.6.: Placement of the cameras for the second test in the "Lost Empire" scene

5.3.2. Results

The data this test resulted in is depicted in figure 5.2 for the "Sponza" scene and in figure 5.3 for the "Lost Empire" scene. Both tables show the data being partitioned by tile sizes. A further dived by screen resolution was omitted for reasons of clarity, since its influence was rather marginal. The results seen here are the average over all screen sizes. Tables showing the extended results broken down by tile and screen sizes can be found in appendix A.2.

tile size	cam ids									
	1&2	1&3	1&4	1&5	2&3	2&4	2&5	3&4	3&5	4&5
8	18%	7%	2%	1%	10%	5%	1%	6%	3%	7%
16	23%	14%	4%	2%	17%	8%	3%	8%	4%	7%
32	31%	24%	9%	5%	27%	15%	6%	14%	7%	10%
64	40%	37%	17%	11%	41%	25%	14%	23%	13%	18%

Table 5.2.: Results of the second test for the "Sponza" scene

This table shows how much of the shading can be reused between two camera angles

As the table in figure 5.2 clearly shows, the overlap for this scene is rather small in most cases. This is especially true for camera number 5 since it is not even looking at the center of the scene, like all the other cameras. The reason why so few of the already shaded tiles can be reused in this scene is, that here most of the view is blocked by pillars close to the cameras. Also, all cameras see very little of the floor or the ceiling, they only see the walls on the opposite site, which means that only cameras looking in the same direction could see similar parts of the scene, and therefore share some tiles. Most cameras here see completely different parts of the scene.

tile size	cam ids									
	1&2	1&3	1&4	1&5	2&3	2&4	2&5	3&4	3&5	4&5
8	37%	36%	18%	21%	57%	22%	20%	23%	22%	50%
16	40%	38%	21%	23%	58%	23%	21%	25%	22%	52%
32	45%	41%	27%	28%	60%	27%	25%	28%	25%	56%
64	51%	45%	38%	38%	66%	35%	32%	34%	31%	60%

Table 5.3.: Results of the second test for the "Lost Empire" scene

This table shows how much of the shading can be reused between two camera angles

It does look a bit better for the "Lost Empire" scene, as the table in figure 5.3 clearly shows. The overlap here is almost always at least 20%, with some even being even 60% and higher. The reason why it works so much better in this scene is, that all cameras are placed on an open landscape, where there is not much blocking their view, like in the "Sponza" scene. They also see a lot of flat ground and not many walls. Walls can only be seen from the direction they are facing, whereas ground can be seen from all angles. This results in a much higher overlap between cameras, and therefore more shared shading work.
The data here is not broken down by screen sizes, since the influence of the resolution, as already mentioned, was rather small. This makes sense since the resolution does not change the area two cameras overlap in. It only scales down what can be seen, but keeps the ration untouched. The difference in the results for all screen sizes was only a few percents with only

a few outliers, as can be seen in appendix A.2.
Tile size has a bigger influence, but even here the amount of tiles two cameras share is only really different for big tile sizes. Probably because with bigger tiles each camera has more unnecessary work to do, and is therefore more likely to shade parts of the scene that are also visible for other cameras. The first test already showed this. It gets especially clear in the "Sponza" scene, depicted in figure 5.2, where there is only a noteworthy overlap for a tile size of 64 pixels. This is probably because with tiles that big both sides of the pillars get shaded, and are therefore more visible for other cameras.
Overall, it can be said that in some cases Object Space Shading might be useful for some multi-user applications. These are mostly cases where players are in small areas close together, like for example multiplayer shooters with small arenas. It is also useful for games where users see the game from a similar perspective like in racing or sport games, or have a kind of top-down view, like in many strategy or MOBA[4] games. It might also be useful for VR games, even thought this was not specifically tested here.

5.4. Third Test: Comparing consecutive frames

5.4.1. Setup

The third and last test is performed similarly to the second test. But instead of comparing the images from two different cameras we compare consecutive frames of the same moving camera. This is done to measure the number of tiles that can be reused each frame of a normal moving camera, similar to what we would have in most computer games. This tells us how much of the already done work we can use again each frame, and how much we have to shade new. We need this to improve the performance of the renderer, and counterbalance the costs measured in the first test. If the savings here are not high enough, Object Space Shading might not be worth the effort, at least not for normal applications.
For the measurement the exact same camera movement as in the first test was used. Walking through the arcades of the upper level around the atrium for the "Sponza" scene, and hovering over the scene, flying downwards and into the cave for the "Lost Empire" scene. But this time, not all of the frames were used. This would have been too much data to evaluate. Instead, 60 frames from each scene were picked, from three different moments in the camera movement. Similar to the second test, the calculations for the comparisons were not done during the render process. Instead, all tile masks for every object in both scenes were saved and for later evaluation. This resulted in over over 28.000 tile masks, from two scenes, four resolutions, four tile sizes, 60 frames and 14 objects having each one tile mask. These where then compared using Python, for easy evaluation and visualization.
Again, this test was done for the same tile sizes as the two previous test, which are 8, 16, 32, and 64 pixels. Also the screen resolutions this test was performed with stayed the same as before. These are 800x600, 1024x768, 1920x1080 and 3840x2160.

[4]"MOBA" stands for "Multiplayer Online Battle Arena" and includes games like DotA 2 and League of Legends

5.4.2. Results

The results of the test can be seen in figure 5.4. The table shows the median reusability for each test performed. The table is partitioned by tile size and screen resolution. The first column for both scenes shows how much of the already shaded tiles can be reused for the next frame, the second column shows how much has to be shaded new.

		Sponza		Lost Empire	
		reusable	new for	reusable	new for
tile size	screen size	old frame	next frame	old frame	next frame
8	800x600	95,47%	4,28%	97,11%	2,78%
	1024x768	96,08%	3,79%	97,69%	2,18%
	1920x1080	96,87%	2,98%	98,13%	1,70%
	3840x2160	97,52%	2,41%	98,65%	1,33%
16	800x600	95,47%	4,58%	97,96%	1,98%
	1024x768	96,09%	3,99%	98,33%	1,62%
	1920x1080	96,78%	3,19%	98,56%	1,25%
	3840x2160	97,47%	2,44%	98,86%	1,06%
32	800x600	96,03%	4,05%	98,28%	1,55%
	1024x768	96,49%	3,74%	98,48%	1,31%
	1920x1080	97,08%	3,07%	98,73%	1,00%
	3840x2160	97,51%	2,45%	99,00%	0,88%
64	800x600	97,33%	2,84%	98,58%	1,26%
	1024x768	97,46%	2,69%	98,58%	1,33%
	1920x1080	97,83%	2,31%	98,91%	0,87%
	3840x2160	97,87%	2,20%	99,14%	0,82%

Table 5.4.: Results of the third test

This table shows how much of the shading can be reused between consecutive frames of the same moving camera.

Just by taking a quick look at the table it gets very clear that the number of tiles that can be used again for the next frame is very high for both scenes, independent of tile size and screen resolution. In every test performed the reusable amount was always above 95% of the total visible tiles of the frame before. The new frame never had to shade more than 4.6% of its visible tiles. This means, that a once shaded tile could theoretically be used for at least 20 frames on average. This of course is only possible if the shading does not change during that time, which would only be true if it is not effected by the cameras position or moving lights. But even in theses cases, shading could be done at a lower frequency than the frame rate, like for example only every second frame, without noticeable difference.
The screen resolution does influence the results. For higher resolution more of the old frame

can be reused. This means that in some cases only have as many tiles have to be shaded new, if you compare the results for 3840x2160 pixels with the results for 800x600 pixels.
The same can be said about the influence of the tile size. For bigger tile sizes more of the old tiles can be reused and fewer have to be shaded new. Here too, the work that has to be done for a size of 64 pixels is only half of that for 8 pixels in some cases.
Also noticeable is the difference between both scenes. The results for the "Lost Empire" scene are a bit higher in every case. This means that the actual result is dependant on both the movement speed of the camera and whether there are objects close to it, like in the "Sponza" scene. Closer objects move faster through the visible field of the camera, and are therefor visible for a shorter time, which results in more tiles that have to be shaded new. All in all the results are pretty good. They are considerably higher than originally expected.

5.5. Summary of the Results

After having done all these tests, we can now try to answer the question asked at the beginning: Do the benefits outweigh the costs? Considering the excellent results of the third test, the answer is yes, they do, at least if it is done right. With the possibility of reusing 95% of our already done work, we could improve the performance substantially. Even though most likely the real world improvements that could be gained from Object Space Shading are not as high as 95%, since the lighting in the scene and the camera's perspective, and therefore the shading, do change during the runtime, and updating the tiles once in 20 frames is not enough. This could nonetheless be used to make the shading rate independent of the frame rate. A game running at 60 frames per second does not need to be shaded at the same rate. Instead, it could be shaded only 30 times per second, and therefore reduce the shading work by half, without having a noticeable effect on the image quality. Considering that the additional costs were up to double that of a classic approach for a tile size of 8 pixels, as we have seen in the first test, this improvement is the minimum amount necessary to make Object Space Shading worthwhile. A bigger tile size is not advised, as this test also showed us that the additional costs rise drastically with the tile dimensions, while the third test revealed that the positive influence on the performance is not as big.

6. Conclusion and Future Work

Object Space Shading can improve the performance significantly if it is done right. The excellent results of the tests that were performed for this thesis support this. With the possibility to reuse 95% of already done shading work, there is a lot of room for performance improvements. Even though the real world performance gains might be lower than that, Object Space Shading is still worth the effort.
Only parts of the extra costs this method causes were measured. Namely how many pixels were unnecessarily shaded. This showed us that the costs can vary widely between different screen resolution, which makes it hard to give a final result. Still, we can say that the cost does not more than double if the tile size is chosen right, which is still in an acceptable range, thanks to the excellent performance gains. Other overhead costs were not measured since this would require a more performance-optimized pipeline.
The results also showed that a bigger tile size does not improve the performance as much as it raises the costs. For a tile size of 64 pixels, the costs increased up to sevenfold, while the performance improvements only doubled. Therefore a smaller tile size is advised.
The project accompanying this thesis also showed, that such a shading approach can easily be implemented into an already existing environment, like the Unity3D Game Engine. Its new "Scriptable Render Pipeline", which allows the user to take complete control over the rendering process via a C# script, made this thesis possible.
Future research could try to implement a performance-optimized pipeline, to evaluate the overhead costs of Object Space Shading further. This would most likely include some advanced memory allocation algorithm, that gives memory space only to tiles visible on the screen and does not allocate memory for parts of the texture, that are not visible. Another way to improve performance might be to calculate and store the normal and local position in a texture and not calculate them each frame from the vertex buffer. This would not take more memory space than the pipeline that was implemented here already does since it would not have to store barycentric coordinates and vertex IDs in textures.
It could also be researched on how much Object Space Shading improves the performance in VR applications. I would expect that it might boost the performance significantly since for VR the same scene has to be rendered twice from almost identical angles. This is a big advantage for Object Space Shading. Every point in the scene has to be rendered at most once, no matter how many virtual cameras look at it.
Conclusively, it can be said that while this thesis already showed with very promising results that Object Space Shading might greatly improve rendering performance, more research about the additional costs and other use cases needs to be done.

A. Figures

A.1. First Test: Additional Figures

The figures that can be seen here are an addition to the first test, the results of which are discussed in section 5.2.2. They show the number of pixels the Object Space Renderer shades every frame of a moving camera.

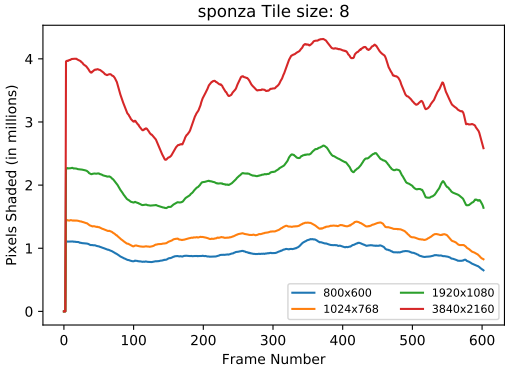

Figure A.1.: First Test, Sponza, Tile size: 8

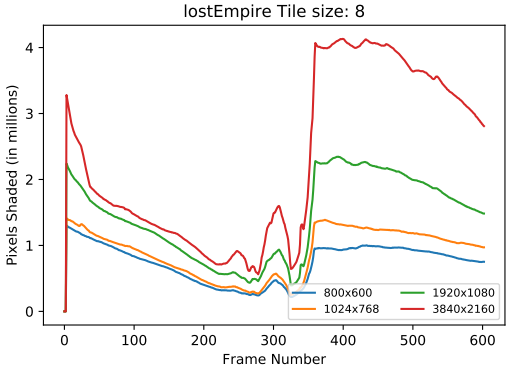

Figure A.2.: First Test, Lost Empire, Tile size: 8

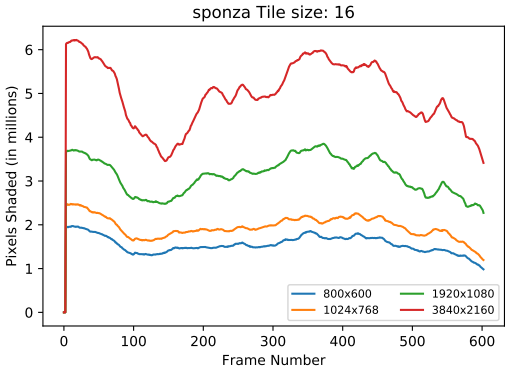

Figure A.3.: First Test, Sponza, Tile size: 16

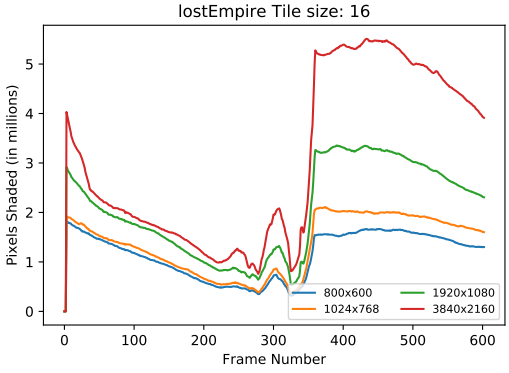

Figure A.4.: First Test, Lost Empire, Tile size: 16

Figure A.5.: First Test, Sponza, Tile size: 32

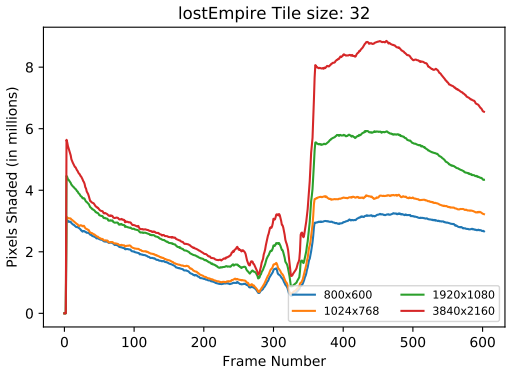

Figure A.6.: First Test, Lost Empire, Tile size: 32

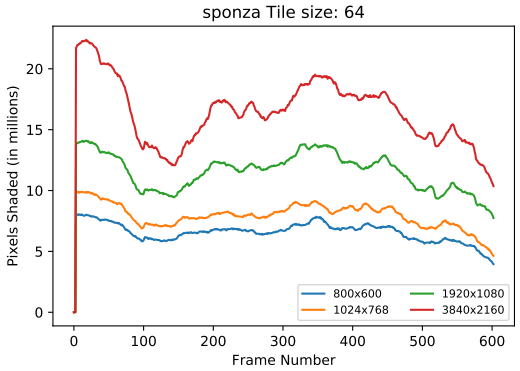

Figure A.7.: First Test, Sponza, Tile size: 64

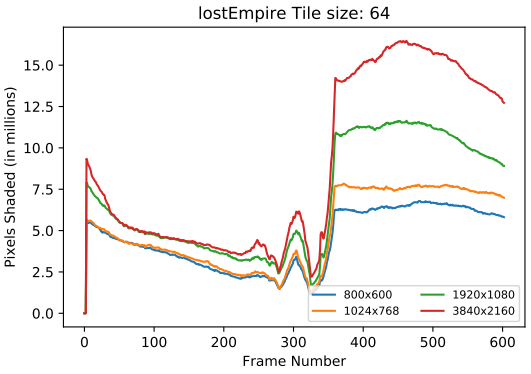

Figure A.8.: First Test, Lost Empire, Tile size: 64

A.2. Second Test: Additional Results

The tables that can be seen here show the extended results of the second test. A more compact version can be found in section 5.3.2, where the results of the second test were discussed. The tables show how much overlap there is between two cameras, placed at different points in the scene, looking roughly at the same area from different angles. Pictures showing the placement of the cameras in the scene, including the direction each of them is looking at, can be found in section 5.3.1.

tile size	screen size	cam ids									
		1&2	1&3	1&4	1&5	2&3	2&4	2&5	3&4	3&5	4&5
8	800x600	20%	8%	2%	1%	11%	5%	1%	5%	0%	2%
	1024x768	18%	8%	2%	1%	11%	4%	1%	5%	0%	4%
	1920x1080	15%	6%	2%	1%	9%	5%	1%	6%	4%	12%
	3840x2160	19%	6%	3%	2%	10%	7%	2%	7%	7%	10%
16	800x600	26%	17%	5%	2%	19%	9%	3%	8%	2%	4%
	1024x768	24%	15%	4%	2%	18%	7%	2%	8%	2%	5%
	1920x1080	20%	12%	4%	2%	15%	8%	2%	8%	5%	12%
	3840x2160	23%	11%	4%	3%	15%	9%	2%	8%	6%	10%
32	800x600	35%	29%	12%	6%	31%	17%	8%	15%	7%	8%
	1024x768	31%	26%	9%	5%	29%	14%	6%	14%	5%	8%
	1920x1080	28%	21%	8%	5%	25%	15%	6%	14%	8%	14%
	3840x2160	29%	18%	7%	4%	23%	14%	5%	13%	7%	11%
64	800x600	44%	42%	20%	13%	44%	27%	16%	24%	15%	19%
	1024x768	40%	39%	18%	12%	42%	24%	14%	23%	12%	16%
	1920x1080	38%	35%	17%	11%	39%	26%	14%	24%	15%	22%
	3840x2160	38%	31%	14%	9%	37%	24%	11%	21%	12%	17%

Table A.1.: Extended version of the results from the second test for the "Sponza" scene

tile size	screen size	cam ids									
		1&2	1&3	1&4	1&5	2&3	2&4	2&5	3&4	3&5	4&5
8	800x600	40%	40%	20%	22%	59%	25%	23%	27%	24%	48%
	1024x768	40%	40%	21%	22%	56%	23%	21%	25%	23%	51%
	1920x1080	37%	35%	18%	20%	56%	21%	21%	23%	22%	52%
	3840x2160	32%	28%	14%	18%	56%	17%	18%	18%	18%	51%
16	800x600	42%	42%	23%	25%	60%	27%	24%	28%	25%	51%
	1024x768	42%	42%	23%	25%	57%	24%	22%	26%	23%	53%
	1920x1080	40%	37%	21%	23%	57%	23%	21%	25%	22%	53%
	3840x2160	35%	30%	18%	21%	56%	19%	19%	20%	19%	52%
32	800x600	46%	46%	29%	30%	63%	30%	27%	31%	28%	55%
	1024x768	46%	45%	29%	31%	60%	28%	26%	29%	26%	56%
	1920x1080	45%	40%	28%	28%	60%	27%	24%	28%	25%	56%
	3840x2160	41%	34%	24%	25%	58%	24%	22%	23%	21%	55%
64	800x600	50%	48%	39%	41%	67%	39%	35%	38%	35%	58%
	1024x768	51%	48%	39%	41%	66%	35%	33%	34%	32%	59%
	1920x1080	55%	45%	38%	36%	65%	34%	30%	33%	30%	61%
	3840x2160	50%	40%	35%	34%	67%	33%	29%	29%	27%	61%

Table A.2.: Extended version of the results from the second test for the "Lost Empire" scene

List of Figures

List of Tables

Bibliography

[Abr97] M. Abrash. *Michael Abrash's Graphics Programming Black Book, Special Edition*. 1997. URL: http://www.jagregory.com/abrash-black-book/#the-quest-for-alternative-lighting (visited on 08/02/2019).

[Bak16] D. Baker. *Advanced Graphics Techniques Tutorial Day: Object Space Rendering in DirectX 12*. Oxide Games. 2016. URL: https://www.gdcvault.com/play/1023511/Advanced-Graphics-Techniques-Tutorial-Day (visited on 11/02/2019).

[Ola+02] M. Olano, J. Hart, W. Heidrich, and M. McCool. *Real-Time Shading*. A K Peters/CRC Press, 2002. ISBN: 1568811802.

[Pho75] B. T. Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: http://doi.acm.org/10.1145/360825.360839 (visited on 11/02/2019).

[Han+12] C. Hansen, E. W. Bethel, T. Ize, and C. Brownlee. "Rendering". In: (2012).

[McS13] M. McShaffry. *Game coding complete*. Boston, Mass: Course Technology PTR, 2013. ISBN: 978-1-133-77657-4.

[AHH08] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. A K Peters/CRC Press, 2008. ISBN: 9781568814247.

[Lun12] F. Luna. *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning & Information, 2012. ISBN: 1936420228.

[Gei13] M. Geig. *Sams Teach Yourself Unity Game Development in 24 Hours*. Sams Publishing, 2013. ISBN: 0672336960.

[Wil83] L. Williams. "Pyramidal Parametrics". In: *SIGGRAPH Comput. Graph.* 17.3 (July 1983), pp. 1–11. ISSN: 0097-8930. DOI: 10.1145/964967.801126. URL: http://doi.acm.org/10.1145/964967.801126 (visited on 11/02/2019).

[Eng10] W. Engel. *GPU Pro: Advanced Rendering Techniques*. A K Peters/CRC Press, 2010. ISBN: 1568814720.

[Hil16] K. Hillesland. *Texel Shading. Blog Post*. 2016. URL: https://gpuopen.com/texel-shading/ (visited on 13/02/2019).

[Tho14] A. Thorn. *Unity 4 Fundamentals. Get started at making games with Unity*. Focal Press, 2014. ISBN: 978-0-415-82383-8.

[Hoc15] J. Hocking. *Unity in Action. Multiplatform game development in C# with Unity 5*. Manning Publications, 2015. ISBN: 978-1-61729-232-3.

[HY16] K. Hillesland and J. C. Yang. *Texel Shading. Short Paper*. 2016. URL: http://developer.amd.com/wordpress/media/2013/12/TexelShading_EG2016_AuthorVersion.pdf (visited on 14/02/2019).

[BFM10] C. A. Burns, K. Fatahalian, and W. R. Mark. "A Lazy Object-space Shading Architecture with Decoupled Sampling". In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 19–28. URL: http://dl.acm.org/citation.cfm?id=1921479.1921484 (visited on 14/02/2019).

[Lyo93] R. F. Lyon. "Phong Shading Reformulation for Hardware Renderer Simplification". In: (1993).