FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Separable Subsurface Scattering in VR

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Lukas Fischer
Matrikelnummer 01527007

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Dipl.-Ing. Christian Freude

Wien, 18. Oktober 2018

Lukas Fischer	Christian Freude

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Separable Subsurface Scattering in VR

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Lukas Fischer
Registration Number 01527007

to the Faculty of Informatics
at the TU Wien
Advisor: Dipl.-Ing. Christian Freude

Vienna, 18th October, 2018

Lukas Fischer	Christian Freude

Erklärung zur Verfassung der Arbeit

Lukas Fischer
Kaltenbäckgasse 2/7, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Oktober 2018

Lukas Fischer

Danksagung

Anfangs möchte ich mich bei meinem Betreuer, Christian Freude, für die Unterstützung bedanken. Nicht nur ermöglichte er mir die Möglichkeit mich im Rahmen einer Bachelorarbeit mit dem ausgewählten Thema auseinanderzusetzen, sondern half mir auch während der gesamten Dauer der Arbeit mit Ratschlägen und Antworten auf meine Fragen weiter.

Weiters möchte ich mich auch bei meiner Freundin Sophie bedanken. Sie stand mir bei und bot mir Ausgleich außerhalb von Uniarbeit.

Zum Schluss möchte ich mich noch bei meinen Eltern bedanken welche mich von Oberösterreich aus unterstützten.

Vielen Dank! Ohne euch wäre diese Arbeit nicht so reibungslos zu Stande gekommen.

Acknowledgements

First off, I want to thank my supervisor, Christian Freude, for the support. Not only did he give me the opportunity to learn more about the selected topic, but he also helped me during the whole bachelor thesis with advice and answers to my questions.

Furthermore, I would like to thank my girlfriend Sophie. She stood by me and was always there for me when I needed diversion from work for university.

Finally, I want to thank my parents who supported me from Upper Austria.

Thank You very much! Without you this thesis would not been able to come together so smoothly.

Kurzfassung

Subsurface Scattering ist ein physikalisches Phänomen welches sich bei vielen Materialien beobachten lässt, wohl aber am markantesten bei menschlicher Haut auftritt. Der derzeitige Stand der Wissenschaft erlaubt die lokale Streuung rund um den Eintrittspunkt in ein Medium mithilfe von Faltungen mit separierbaren Filterkernen als Screen Space Effekt zu simulieren. Das Ziel dieser Bachelorarbeit ist es diese Technik, entwickelt von Jimenez at al., in Kombination mit stereoskopischen Rendern zu evaluieren und herauszufinden wie diese in die Videospiel-Engine Unity integriert werden kann. Unity bietet Support für Virtual Reality (VR) Anwendungen an und erlaubt das Entwickeln von eigenen Post-processing Effekten welche auf Shader beruhen. Die implementierte Subsurface Scattering Methode wird zusätzlich kombiniert mit Verfahren für Lichtdurchlässigkeit und einem physikalischem Modell für gespiegeltes Licht. User können in der fertigen Anwendung die Effekte begutachten und haben die Möglichkeit Einstellungen an diesen vorzunehmen. Performance und Qualität der Technik werden untersucht in Bezug auf die Gebräuchlichkeit in Unity, mit stereoskopischen Rendern und Bilder pro Sekunde. Die Anzahl der Bilder die pro Sekunde gerendert werden können sind besonders wichtig in VR Anwendungen um den Usern ein angenehmes interaktives Erlebnis zu ermöglichen.

Abstract

Subsurface Scattering is a physical phenomenon that appears in many materials but is most notable for human skin. Current research makes it possible to calculate the local scattering of light inside a translucent medium around the point of entry with a convolution of a separable filter in screen-space. This thesis tries to evaluate this technique by Jimenez et al. for stereoscopic rendering and how it can be implemented for the currently popular game engine Unity. Unity offers support for VR applications and allows the implementation of post-processing effects and other techniques that rely on shaders. The implemented Subsurface Scattering method is combined with an approach for translucency and a physically based specular model. In the developed application the effects can be observed with and without VR and important parameters can be changed by the user. The performance and visual quality are reviewed with respect to the viability of the effects in Unity, stereoscopic rendering and frame rate. The latter is especially important in VR applications to deliver a comfortable interactive experience.

Contents

CHAPTER 1

Introduction

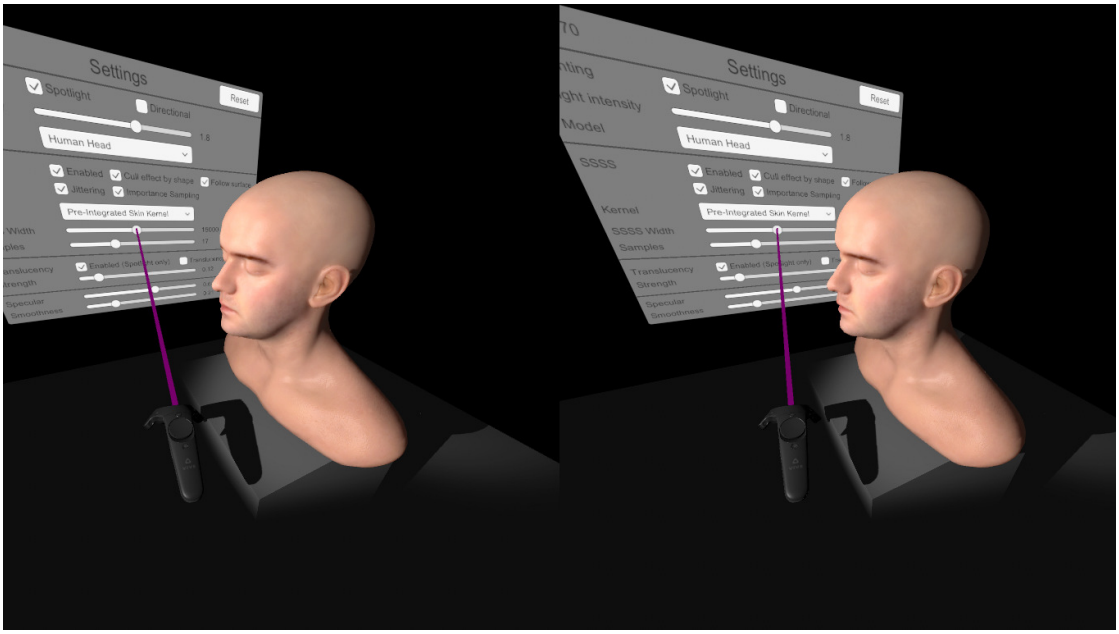


Figure 1.1: The screenshot shows the final VR application.

Realism in VR applications is an important goal if the intention of the developer is to make a believable realistic immersive world. Much research is currently done in the field of real-time rendering techniques. The results are often only examined in terms of performance and quality in an application made exclusively for the approach covered in the paper. This thesis takes a recent paper by Jimenez et al. [JZJ+15] about Separable Subsurface Scattering (SSSS) and examines how it can be implemented for stereoscopic

rendering in a VR application made with the Unity Game Engine. Figure 1.1 shows a screenshot of the resulting application.

The remainder of this chapter goes more into detail about the motivation and the chosen approach to fulfil the goals set for this thesis. The second chapter covers the research and technology this thesis is based on. The subsequent chapter will cover how the techniques have been implemented in Unity for VR. In Chapter 4 outcomes in terms of performance and quality will be discussed. The final chapter will give a summary and talk about potential extensions and improvements for the resulting application.

1.1 Motivation

Realistic rendering of materials is important to make virtual worlds more believable. Especially when it comes to human skin, the human eye is very good at recognizing if something looks unnatural. Offline rendering is a possible solution, but these rendering techniques are not suited for real-time applications like games. This is where techniques like SSSS by Jimenez et al. [JZJ+15] come into play. Subsurface Scattering appears if light enters a translucent medium, scatters inside and leaves the medium again. The consequences of this physical phenomenon are explained in Sections 2.1 and 2.2. The approach by Jimenez et al. is realized as a screen-space filter. Their method offers many variables which can be adjusted dependent on one's needs. It can also be used for materials other than skin. A current trend in technology, where realism plays a key role, is VR. Through a head-mounted display (HMD) the user should be made to believe to be in a virtual world. Realistic representation of humans is important to make a VR application more immersive.

To create VR applications, one can use existing systems like the Unity game engine. It comes with support for VR, and the editor offers many features that come in handy for creating a virtual world and technical aspects like rendering. Furthermore, it can be extended and adjusted to fit one's needs. This leads to the question how current approaches for realistic rendering of materials like human skin can be implemented in Unity. In addition to this it is interesting whether these techniques work correctly in VR and a reasonable performance can be achieved to offer a smooth interactive experience for the user. These questions led to the main goals of this bachelor thesis.

1.2 Approach

The focus of this work is SSSS and its viability for implementation in the Unity Game Engine. Current approaches for realistic rendering of human skin will be added to showcase and test them for their viability in Unity. The used techniques for translucency are explained in Section 2.2 and the specular reflection model in Section 2.3. The implementations work in VR, as well as without in the same application. This way the techniques can also be examined without the hardware needed to experience VR. In order that a person can experiment with them, a user interface exposes parameters that are

important to the user for adjustment in both variants of the application. The deliverable is executable in a Windows 10 64bit environment. SSSS is implemented first. This also involves the setup of the scene and the post-processing stack. Afterwards, the techniques for translucency and the specular model are added. Finally, the application is adapted to work with stereoscopic rendering.

The implementation of SSSS is examined for its quality in VR and whether artifacts are visible when combined with stereoscopic rendering. It works with the HTC Vive VR headset and makes use of their room-scale tracking system. Additionally, the rendering performance is reviewed since high frame rates are important for a responsive and smooth VR experience.

CHAPTER 2

Related Work And Used Technologies

The following sections describe the scientific work this thesis is based on and the technologies that were utilized to realise the mentioned techniques. This involves Subsurface Scattering, Translucency and a model for specular reflections.

2.1 Separable Subsurface Scattering

As light hits translucent matter, multiple characteristics can be observed. A fraction is reflected directly at the surface and visible as a specular reflection. Another part of the light enters the medium and spreads inside. The light scatters inside the medium and a portion leaves near the point of entry. This blurs the skin slightly, makes details such as pores less noticeable and transitions from dark to light areas appear softer. Figure 2.2 visualizes this by example with a patch of skin. Each color gets scattered differently inside different media. A fraction of the light is absorbed which adds to the red color of skin. The red wavelengths are less likely to be absorbed and can travel further inside the skin. This results in visible red fall-off at the transition areas between lit and shadowed areas. Another property that can be observed is that some light enters the matter and a fraction exits at the opposite side depending on the distance travelled. This will be explained in Section 2.2. Figure 2.1 illustrates these aspects. The terms Subsurface Scattering and Translucency refer to the same physical phenomenon but is used differently in scientific papers. Based on the papers this thesis covers, the following sections refer to the blur as Subsurface Scattering and the light that exits on the other side of an object as Translucency. The separation of these two comes from the fact that real-time rasterization approaches are not able to calculate both with one technique.

The slight blur makes the resulting material appear much more realistic. Multiple approaches exist as to how subsurface scattering can be achieved. Offline rendering

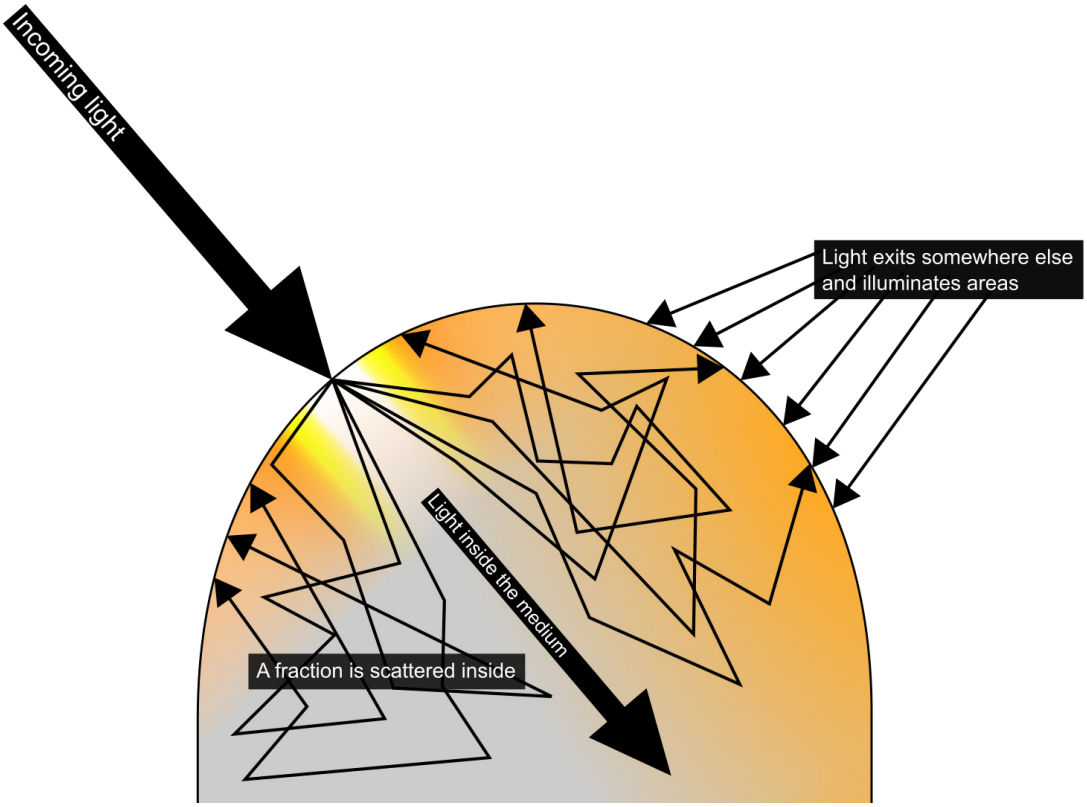
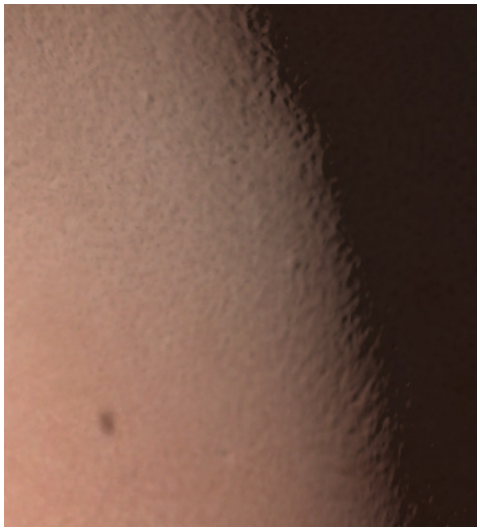


Figure 2.1: This illustration depicts Subsurface Scattering inside a medium, e.g. a human earlobe. The fraction of the light that enters the medium is scattered randomly inside.

techniques exist, but to be used for interactive applications like games the effect must be calculated fast enough to fit within the targeted frame time.

One way to achieve subsurface scattering is the texture-space approach by d'Eon et al. [dLE07]. Instead of simulating the individual rays the resulting blur can be approximated with a kernel that gets convoluted with the texture of the skin. This leads to a two-dimensional convolution operation that is very expensive. d'Eon et al. overcome this performance heavy calculation with a separable sum-of-Gaussian approach. Although a performance increase was achieved, their technique doesn't scale well with number of textures the diffusion has to be applied on. This led Jimenenez et al. [JSG09] to move the convolution from texture- to screen-space. This has the advantage of not being dependent on the amount of entities that appear on the screen. It also removes additional overhead from the rendering pipeline that was necessary for the approach from d'Eon et al. The convolution is applied only on the parts of the image that contain the translucent materials without the specular fraction. The specular fraction must be added to the final image after the convolution because the reflected light isn't part of the scattering that happens inside the translucent media. This screen-space approach is used for this thesis.



(a) Without SSS: Details are visible and nothing is blurred.

(b) With SSS: Details are less noticeable and the transition from light to shadow appears softer.

Figure 2.2: This patch of skin shows the impact of SSSS on fine details like pores.

For their convolution Jimenez et al. use a combination of two different filter applications.

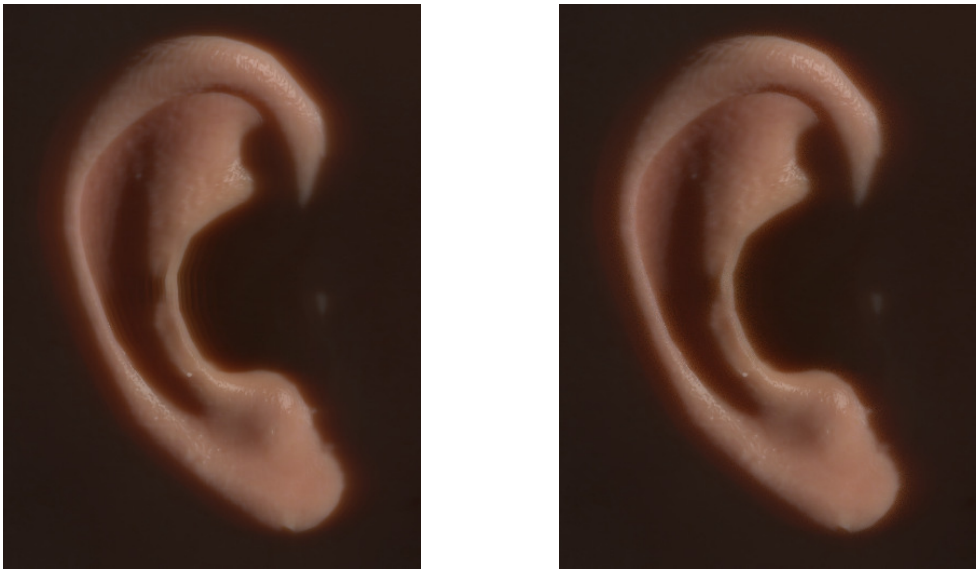
The convolution and the kernel were later further improved by Jimenez et al. with their publication "Separable Subsurface Scattering" [JZJ+15]. This scientific paper continues with the approach in screen-space, but instead of a sum-of-Gaussian convolution only two one-dimensional convolutions are applied. In their work they present a method to simplify the convolution with the original two-dimensional kernel to two one-dimensional kernel-convolutions. For the creation of the discrete kernel they present automatic methods for the simplification of the two-dimensional kernel and an artist-friendly model controlled by parameters.

The goal of the artist-friendly model is to provide a way to create separable kernels that are not based on a physically based diffusion profile, but they can still be fitted to approximate a physical profile to some extent. The following formula represents the separable 1D kernel used by Jimenez et al.:

$$a(x) = wG(x, \sigma_n) + (1 - w)G(x, \sigma_f)$$

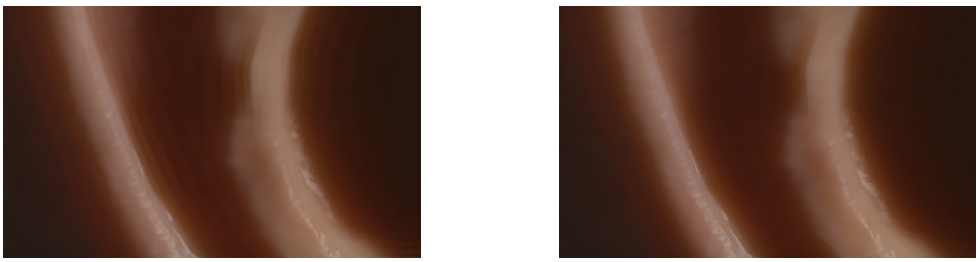
It combines two Gaussians G, for near and far scattering respectively, and combines them with a weight w. σ_f and σ_f represent the standard deviations for near and far. The values of the standard deviations are chosen differently for each color channel, whereas w remains the same. Together they form the parameters that are manageable by a user. To use the artist-friendly model in an implementation it must be sampled. The amount and locations of the sample points influence the visual result of the final image.

To be able to use less sample points for the discrete kernel, instead of sampling the kernel uniformly, more sample points are located near the area with the most energy, the entry point of the light. The kernel must be normalized so that the weights of the samples correspond with the area that each sample represents. Furthermore, the convolution itself is improved through the incorporation of random rotations for the one-dimensional kernel. This is referred to as Jittering. Rotating the kernels helps especially with artifacts that appear in areas around shadows. Figure 2.3 displays an illuminated ear with and without jittering.



(a) Without Jittering: Banding is visible.

(b) With Jittering: Banding is less noticeable.



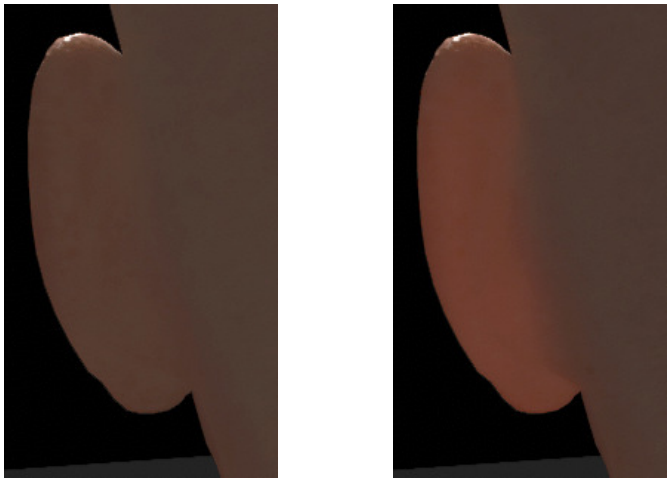
(c) Close-up view without jittering.

(d) Close-up view with jittering.

Figure 2.3: This illuminated ear showcases jittering and how it counteracts banding artifacts. Rotation is only applied to samples that are closer than half the size of the kernel.

2.2 Translucency

The scientific papers that cover real-time Subsurface Scattering generally treat the blur around the entry point differently than the light that travels through the medium and exits somewhere else on the other side with enough energy to be noticed by an observer. This occurrence is called translucency and is very notable on materials like candle wax and ears. Figure 2.4 shows a rendered ear with and without translucency.



(a) Translucency disabled (b) Translucency enabled

Figure 2.4: These two images show a rendered ear to visualize translucency. The ear is shown from behind and is illuminated from the front with a spotlight.

One approach to render translucency is the approach by Jimenez et al. [JWSG10] which was also used for this thesis. Their technique uses shadow maps in the fragment shader to calculate the distance from the fragment to the depth stored in the map. This leads to a concave estimation of shape's geometry. The calculated distance is then used as the input for a transfer function that converts the distance to a color intensity. This transfer function is different for each material. The output of the function is added to the albedo information of the fragment before the screen-space subsurface scattering blur is applied. Figure 2.5 shows two transfer functions of materials where translucency is generally prominent and noticeable.

Although this approach is simple, it uses generalisations and has some downsides. Since there is no information available about the point stored as visible to the shadow map, it is assumed that its normal is the normal of the fragment inverted. A downside of this method is that it doesn't work correctly with concave objects. The shadow map of the light that is used for translucency can only store the frontmost geometry of the object (e. g. imagine a frosted glass that is lit from the front).

(a) Skin transfer function

(b) Marble transfer function

Figure 2.5: These are two transfer functions for translucency. Both range from thickness 0 (left) to 3 cm (right). (a) is a transfer function for skin (taken from [JWSG10]) and (b) shows how light is attenuated inside marble (taken from the demo application for [JZJ+15]).

2.3 Specular model

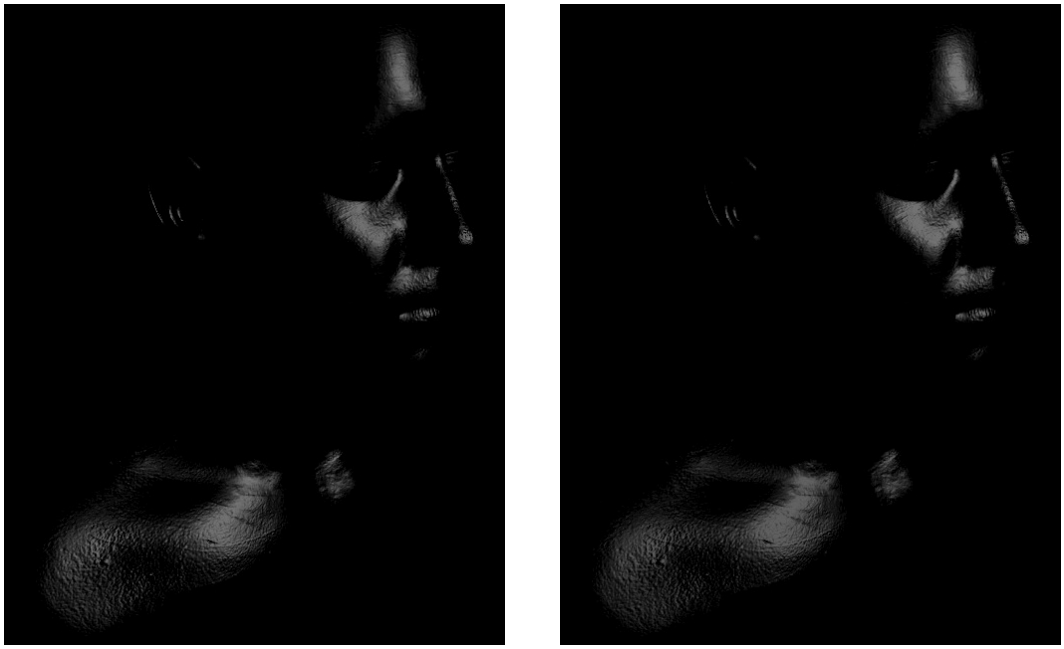
Specular reflections are an important part to make rendered surfaces more believable. For this thesis the Kelemen/Szirmay-Kalos [KSK01] physically based specular model was used. d'Eon and Luebke [dL07] take this model and process it into an efficient solution that can be easily added to other systems.

The Kelemen/Szirmay-Kalos model is a physically and microfacet based solution to calculate specular reflecions. Physical models try to take real properties of the surface and natural laws into account to create a realistic depiction of how a portion of the light is reflected directly when it hits the surface of a material. Emperical models on the other hand try to recreate the physical appearance. The most well-known empirical model is the Phong Model [Pho75]. Figure 2.6 shows a exemplary comparison between the Phong and the Kelemen/Szirmay-Kalos model. Microfacet-based means that it is implied for the calculations that every evaluated point is a mirror, and the only difference between the mirrors is a random rotation.

An important part of specular models is the resulting bidirectional reflectance distribution function (BRDF). Such a function takes a light and view vector as input. It describes how likely it is that light coming from the light direction is reflected towards the view direction. The paper by Kelemen and Szirmay-Kalos presents the following BRDF:

$$brdf(\mathbf{L}, \mathbf{V}) = P(\mathbf{H}) \cdot \frac{F(\mathbf{H} \cdot \mathbf{L})}{\mathbf{h} \cdot \mathbf{h}}$$

$\mathbf{L}$ is light vector and $\mathbf{V}$ is the view vector. $\mathbf{h}$ represents the unnormalized halfway vector of $\mathbf{L}$ and $\mathbf{V}$. $\mathbf{H}$ is the normalized one. $P(\mathbf{H})$ is a probability function that uses $\mathbf{H}$ as input. The Beckmann distribution is used for the probability function. This function controls the random rotations of the microfacets. $F(\mathbf{H} \cdot \mathbf{L})$ is a Fresnel function that needs to be adapted depending on the materials the specular model is applied on. The random distribution and the Fresnel function are the terms that determine the final look of the specular reflections.



(a) Phong model (b) Kelemen/Szirmay-Kalos model

Figure 2.6: These images display a comparison between the Phong and the Kelemen/Szirmay-Kalos specular model.

d'Eon and Luebke [dL07] use this BRDF and apply changes and optimizations to it. First off, the Beckmann distribution needed for the function is precalculated and stored in a texture. Secondly, for the Fresnel function they use Schlick's approximation and make it depend on **H** and **V**.

$$F(\mathbf{H}, \mathbf{V}) = (1 - (\mathbf{H} \cdot \mathbf{V}))^5 + F_0(1 - (1 - (\mathbf{H} \cdot \mathbf{V}))^5)$$

The reflectance parameter F_0 depends on the material's surface and can be calculated with the Fresnel equation when light hits a surface perpendicular.

$$F_0 = (\frac{n_{air} - n}{n_{air} + n})^2$$

In the equation the value n_{air} represents the refractive index of air 1 and n the refractive index of the material.

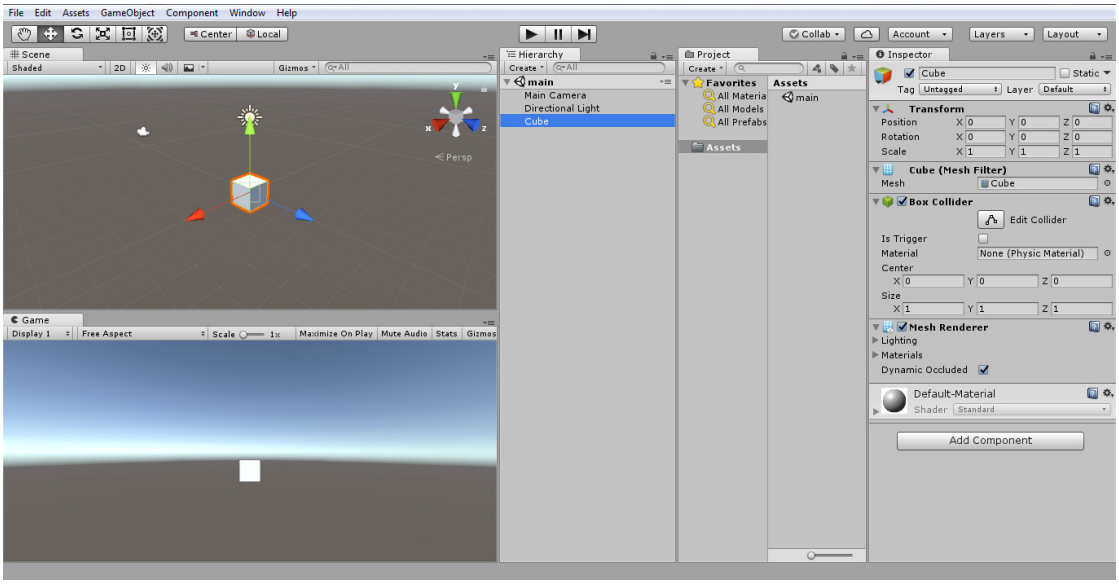


Figure 2.8: When a new project is created, the editor is opened with a default layout and a few pre-made objects in the scene.

2.4 Unity

Figure 2.7: This is the logo of the Unity Game Engine.

Unity is a cross-platform game engine developed by Unity Technologies and contains integrated development tools for the creation of 2D and 3D applications [geb]. Current supported platforms are Windows, Mac, Linux, WebGL and all current game consoles among others. Unity is free for personal use and can be downloaded at `https://unity3d.com/get-unity/download`. Unity offers many functionalities. These include an UI-System, profiling tools and support for Virtual and Augmented Reality applications. Figure 2.8 shows the interface of the Unity editor of a newly created project that only contains a cube.

A scene in unity consists of multiple GameObjects. To add functionality to the application one must create components for the GameObjects. To create a component that can be added to a GameObject the class MonoBehaviour has to be extended. As a programming

language Unity uses C#. Public variables of a class that extends MonoBehaviour are exposed in the editor and can be changed in the inspector if the currently selected GameObjects contains a MonoBehaviour. To implement functionality the user can overwrite the Start and Update functions.

Unity also offers functionalities for screen-space shaders which can be used for applications like SSSS. In Unity this is called Post-processing stack and already contains some common Post-processing effects like Bloom and Anti-Aliasing, as well as a framework to write screen-space effects. Unity currently reworks their Post-processing stack under the name 'Post-processing Stack v2'. An early version can be downloaded from Github `https://github.com/Unity-Technologies/PostProcessing`. The implementation of SSSS in this thesis was built as an effect for the new stack.

2.5 Virtual Reality

Simulating a virtual world and bringing a user into this world through technology is a definition for VR. Through a HMD the user is tricked into believing to be in this virtual world [SVS05]. The HMD tracks the movement of the person which is utilized as input for the application that creates the virtual world. While one of the main uses for VR technologies are entertainment purposes, it can also be utilised for medical (e.g. treatment of psychological disorders [MKBKR17]) or industrial (e.g. blind spot tests for cars [BV17]) intentions. Important requirements towards the application are the frame rate, the display latency and the believability of the virtual world depending on the application. This is where techniques like SSSS come into play. They offer potentially good performance and make the world more believable through realism.

To create an image for the HMD, that appears to the viewer three-dimensional, different images must be rendered for each eye. This process is called stereoscopic rendering. The main difference between the two pictures is the offset for the individual views of each eye. This gap between the eyes is recreated with different view matrices. Furthermore, a slight rotation of both cameras towards the middle between the eyes is made to simulate the convergence of the eyes. For optimizations the clip space for each view is different, which leads to different projection matrices [Gra08].

If the picture for each eye is rendered separately, the amount of draw calls is doubled as a consequence. In Unity this is referred to as multi-pass rendering. A possibility to reduce the amount of draw calls is called single-pass rendering. A single draw call renders the objects twice, once for each eye. This is done through adjustments of the shaders because the shader needs to know for each fragment whether to use the matrices for the left or the right eye [gea].

CHAPTER 3

Implementation

This chapter will cover the implementation of the aforementioned approaches and technologies. The first section will give an overview of the project, the folder structure and which parts of the project are responsible for which functionalities. The sections afterwards will go into detail about the implementations itself, starting with SSSS and concludes with the changes that were made to support VR and stereoscopic rendering. This thesis was implemented with Unity 2018.1.1f1.

3.1 Project Structure

The assets of a Unity-project contain all the project related scripts, shaders, files unity stores, configurations of components and media files like models or textures. The asset folder is located in the root of the project. The explanation of the folder structure should give an overview of the project and how it was implemented.

Materials Inside this folder the materials for objects are stored that are not related to imported models (e.g. the ground a model is placed on). A material in Unity stores the configuration of the exposed parameters for a chosen shader. A material can only be responsible for one shader. The materials are then used by a `Renderer`-component to display objects on the screen.

Models This folder contains the models this application uses, a material for each model and the accompanying textures. The models consist of a 3D-scanned head, the Stanford Dragon and a simple model of a flashlight used for the spotlight in VR (see Section 3.5).

Scenes A scene in Unity stores all the GameObjects of a scene, the attached components and the configurations of these components. This project consists of three scenes.

- `start`: The scene that is loaded when the application is started. It contains a GameObject with an attached `MonoBehaviour` that chooses between the `main` and `mainVR` scene whether or not a HMD is currently connected. If the user has a VR headset connected, the application can be forced to load the scene `main` with the command line argument `force_no_vr`.
- `main`: This scene includes all the GameObjects and attached components needed to use the application without VR.
- `mainVR`: This scene is very similar to the non-VR scene but contains additional GameObjects needed for VR support and some components are configured differently. How the project was adapted for VR support is explained in Section 3.5.

Scripts This folder holds all the files that contain the actual implementation of the application which consist for the most part of C# source code files. Additionally, files needed in the scripts and shaders are stored here.

[SubsurfaceScattering] If one wants to use the implementations for Subsurface Scattering, Translucency and the specular model in another projects, this folder needs to be copied over. It contains not only C#-files, but also all shaders, materials, RenderTextures (used in Unity to declare textures that cameras can render into), the binary kernels and the ScriptableObjects that define the kernels that can be used in the application. The following sections go into detail about the implementations.

[UI] The user interface for both the VR and non-VR version were made with the built-in user interface tools provided by Unity. Figure 3.1 shows the layout of the user interface when the application is started and displays the parameters exposed to the user. The difference to the VR-version is the minimize button in the top left and the checkbox to turn bloom on and off at the bottom. In VR the interface is presented to the user as a panel beside the model (see Figure 1.1). The parameters can be adjusted by pointing at the panel with the controller. If a Vive-controller is used, the trigger corresponds to the left button of a mouse. This folder contains three scripts that are responsible for the functionality of the UI elements and the manipulation of the respective values in the application according to the input of the user.

[VR] Sourcefiles responsible for the functionalities in VR that aren't related to rendering are in this folder. This includes scripts for the use of the VR-controllers and the user interface in VR.

PostProcessing-2 This folder contains Unity's current post-processing system that is getting reworked as mentioned in Section 2.4. This folder was last updated 20th May 2018.

SteamVR This folder contains the SteamVR Plugin from the Unity Asset Store to develop VR applications in Unity for OpenVR compatible HMDs. This project

uses version 1.2.3 and can be downloaded from the Unity Asset store `https://assetstore.unity.com/packages/templates/systems/steamvr-plugin-32647`.

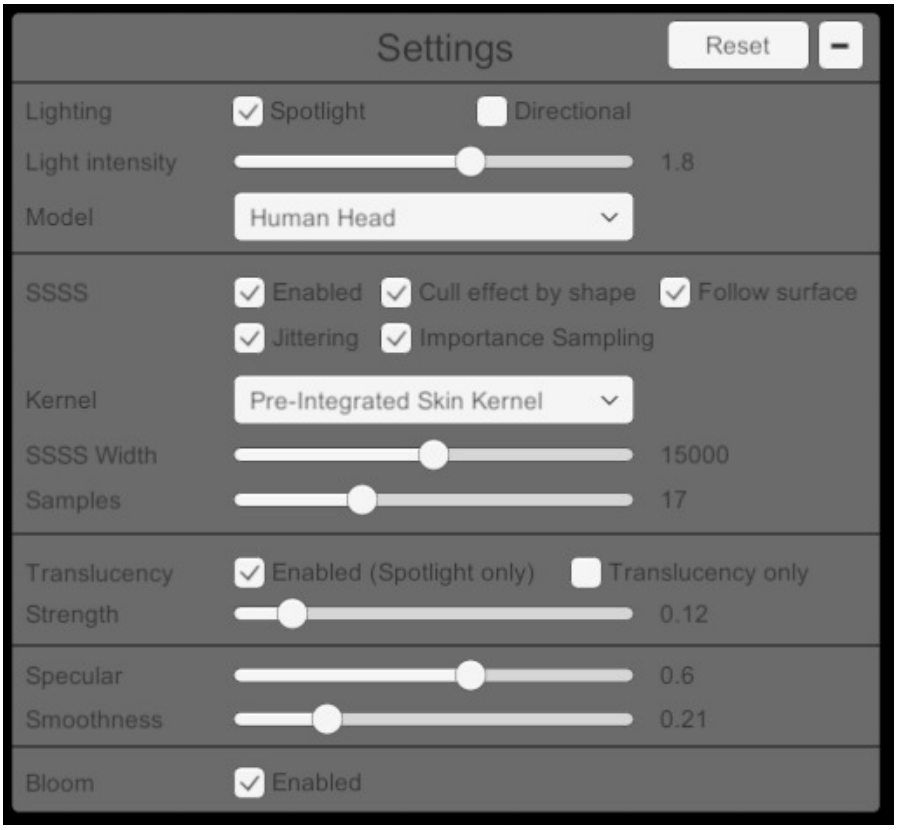


Figure 3.1: The user interface exposes multiple parameters to the user.

3.2 Separable Subsurface Scattering

To realize SSSS in Unity, a lot of tools provided by Unity can be used. Unfortunately, some aspects require workarounds and additional setup that lead to additional draw calls, like for example the need to separate the specular reflections from the screen-space effect. This section will explain the implementation of how the kernels are handled by the application and the rendering of SSSS.

3.2.1 Kernels

The class `SSSSKernelManager` extends `MonoBehaviour` and is responsible for providing the rest of the application with the currently active kernel. It also provides helper methods needed by the different types of kernels. As a `MonoBehaviour` it has to be added to a GameObject in the editor and it can only exist once in a scene, because it is

shared as a Singleton. To change kernel settings during runtime (e.g. through the user interface) the functions in the class `ChangeSSSProfile` must be called. This includes changing the kernel, the sample size of the kernel and toggling importance sampling.

The kernels in the project are represented as ScriptableObjects. ScriptableObjects in Unity are assets that contain logic and parameters. The instances of them are stored as files which comes with many advantages. They can be shared between projects, tracked via source code management and easily edited inside the Unity editor. The abstract class `SSSSKernel` extends `ScriptableObject` and provides methods that each different type of kernel needs to implement. This includes methods to set the kernel up (e.g. loading from a file) or to change the parameters of a kernel (e.g. importance sampling enabled/disabled). For this thesis three different types of kernels have been implemented.

`BinaryKernel` This extension of `SSSSKernel` loads kernels from binary files. This was done in order to support the same separable kernels as used in the demo application by Jimenez et al. for their work [JZJ+15]. A series of float values are loaded into a `Vector4` array and afterwards sampled for usage in the application.

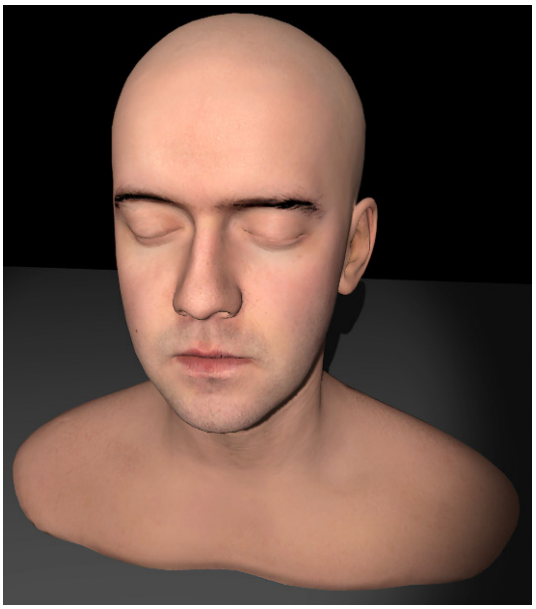
`ArtisticKernel` This class implements the artist-friendly model explained in Section 2.1. This makes different instances of artistic kernels adjustable in the editor. The kernels mentioned in the paper by Jimenez et al. are included in the project. Figure 3.2 shows the kernel with the settings 'Production' generated in Unity.

Figure 3.2: This separable kernel was generated with the artist-friendly model.

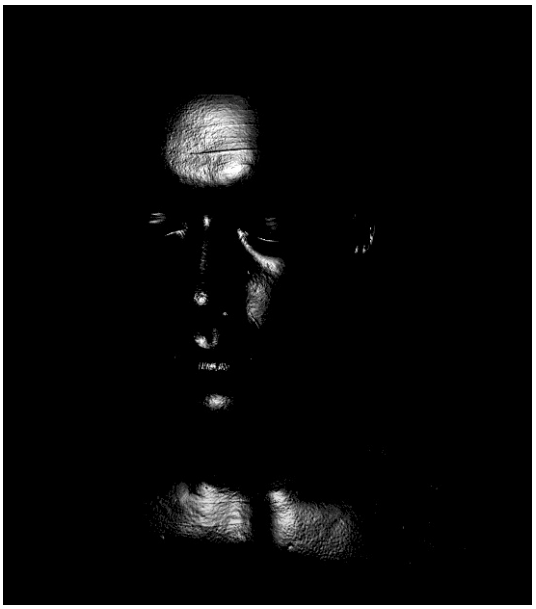
`FixedKernel` This serves the purpose to use completely self-defined kernels in the application. The values for each sample (r,g,b and offset) must be entered manually in the editor. This was done to create kernels that are only meant for debugging and to provide support for values from other sources. A drawback of these kernels is that importance sampling can't be toggled, and the sample size isn't changable during runtime.

The binary and artistic kernels both are prepared before they can be used by the application. At first the offsets are generated depending on the configured number of samples and the maximum range of the kernel. If importance sampling is enabled an array of widths is generated. This is done to multiply them afterwards with the value at the offset locations because each sample doesn't represent the same area of the overall kernel as explained in Section 2.1. For each offset the values of each color channel are interpolated from the loaded binary kernel. In case of the artistic model the values are calculated from its equation. Finally, the kernel is normalized. The result is a `Vector4` array where each entry represents one sample. The `xyz` coordinates correspond to rgb and `w` is the offset of the sample.

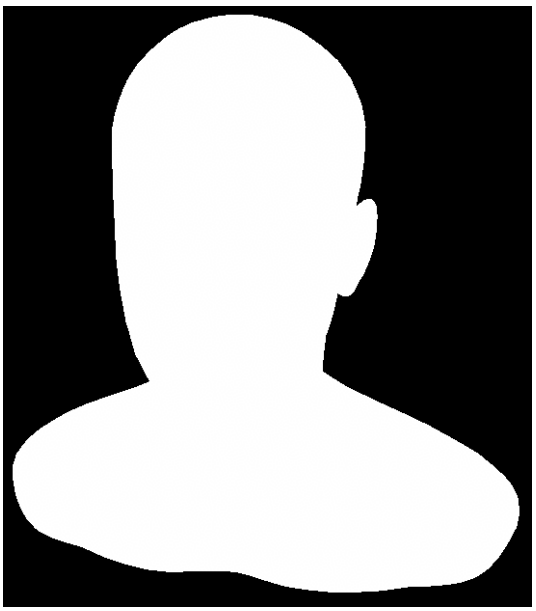
3.2.2 Rendering



(a) Image without SSSS and specular reflections.



(b) Specular reflections that are later added back to the image.



(c) Alpha mask used for culling the effect on the screen.

(d) Image after SSSS, but without specular information.

Figure 3.3: A set of images that show how SSSS is achieved in this application. The final result is shown in Figure 3.4.

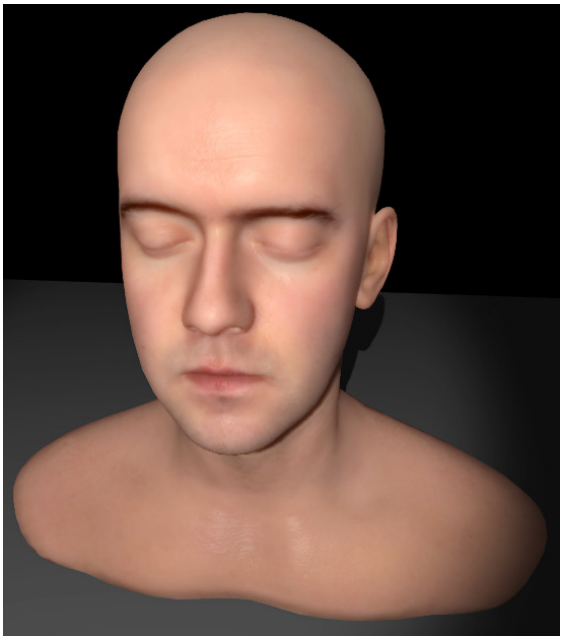


Figure 3.4: To render the SSSS effect, a screen-space shader is applied two times and the specular reflections are added to the frame.

To render the subsurface scattering effect, two cameras are used in the scene. The main camera renders to the screen and it has the Post-processing stack attached. The second camera called 'SpecularHighlightsCamera' is a child of the main camera. It is responsible for rendering the specular reflection and producing an alpha mask. The mask tells the effect on which pixels of the image the effect should be applied on. This camera is disabled in the editor because it shouldn't render automatically and is instead instructed via script to render. It doesn't render to the screen, but instead has a RenderTexture applied as target. A RenderTexture in Unity represents a texture that can be rendered into but is also exposed in the editor as an object. This makes the output of cameras usable in scripts and shaders.

The rendering of a frame is started by Unity after all the update-functions in the MonoBehaviour instances of a scene have been processed. The main camera renders all the objects in the scene. The objects, on which SSSS is applied, use their own shader, instead of the standard one provided by Unity. This was done in order to separate the specular reflections from the albedo information and to add the Translucency to the object. The implementation of Translucency will be later explained in Section 3.3. Figure 3.3a shows how an image looks like at this point.

Afterwards the effects added to the Post-processing stack are evaluated. To implement a screen-space effect the class `PostProcessEffectRenderer` must be extended and the method `Render` has to be implemented. For this project the effect was implemented in the class `SeparableSubsurfaceScatteringRenderer`. At first the method `Render`

forwards the parameters exposed in the editor (see Figure 3.5) and the kernel from `SSSSKernelManager` to the shader. The 'SpecularHighlightsCamera' is instructed to render. This is done via the class `SpecularCamera` which is attached to the camera. It begins with only rendering the SSSS-objects. To not render them with the same shaders that were used beforehand a replacement shader is set. A replacement shader on a camera in Unity is used to render objects with other shaders than the ones set on their mesh renderer components. The shader used in this project, called `SpecularModel`, calculates the specular component (see Figure 3.3b) into the rgb-channels and masking information (see Figure 3.3c) in the alpha-channel of the RenderTexture.

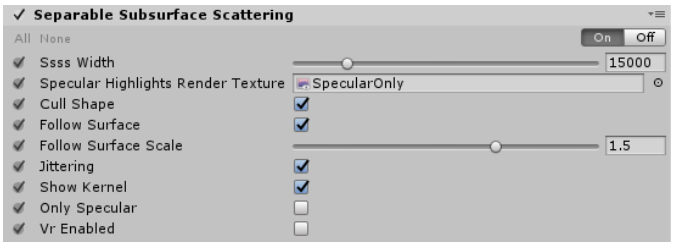


Figure 3.5: These SSSS parameters are adjustable through the editor of Unity.

Now that the shader that is responsible for SSSS has all the information of the current frame, the effect can be rendered to the screen. The shader `SeparableSubsurface-Scattering` is instructed to render two times, once for each pass, because SSSS works with a separable filter. The first pass takes the source image delivered by the post-processing stack and applies the separable filter horizontally. The second pass renders from a temporary texture into the destination which is also predefined by the stack. The second pass not only applies the convolution vertically, but it also adds the specular reflection to the final image. Algorithm 3.1 illustrates what happens in the shader. Figure 3.3d shows the result without specular information and Figure 3.4 shows the result after both passes.

3.3 Translucency

An important part of the method used for translucency in this thesis are the shadow maps used to calculate the distance from where the light hits a surface to the depth stored in the map. Unity allows to access the maps created by lights, however the internal maps weren't used for this implementation because the values stored in them aren't linear and the way they are stored is different for each type of light. Directional lights use cascaded shadow maps, point lights use cube maps and spot lights use single textures to store depth. To have more control over the shadow map created by the light, this implementation uses a camera attached to a spot light as a child object that renders the depth visible to the light into a texture. The approach has only been realized for spot lights.

Algorithm 3.1: The functionality of the screen-space shader that applies the filter kernel is explained in this pseudocode listing.

```
Get the values of the main and specular textures at the current position on the screen
if Current position isn't covered by the alpha mask then
  return the unchanged color of the main input texture
end if
for i < number of sampled points do
  calculate an offset position depending on the offset of the current sample and a
  rotation, the rotation is dependent on the current pass and jittering settings
  get the color information of the main texture at the offset position
  if Following the surface is enabled then
    Interpolate back to the original color, depending on the difference between the
    depths of the original position and the sampled offset position
  end if
  multiply the color at the sampled point with the current sample and add it to the
  resulting color
end for
if this is the second pass then
  add the specular information to the resulting color
end if
return the resulting color
```

The shadow map camera uses a simple shader called `DepthOnly` to render the depth into a texture. It calculates the distance from the world position of the fragment to the origin of the light and divides it by the distance from the far plane to the light. The class `TransmittanceSetupHelper` takes the VP-matrix of the helper camera and forwards it, together with other information needed to calculate the distances, to the shader that renders the objects translucency should be applied on. This includes the normal, intensity and position of the spot light. A shader was implemented for this purpose and to render objects without their specular reflections (see Section 3.2).

To create shaders Unity offers support for traditional vertex and fragment shaders, but they also offer a simpler solution called surface shader. In these shaders only one function needs to be implemented to determine the look of the surface. The need to implement the vertex shader and lighting calculations are taken away from the developer and handled by the engine. The surface shader for this implementation determines the Translucency with the approach described in Section 2.2 and adds it to the color information from the main texture. The specular part is set to zero. When the depth map is read the value is multiplied by the distance of the far plan to revert the values back to the actual distances.

Transfer functions for skin and marble have been added to the shader and can be chosen via a dropdown for each material in the editor (see Figure 3.6). Exposed in the editor

is also a scaling factor before the distance is used by the transfer function (Distance Multiplier) and a scaling factor after (Strength). To remove some artifacts that appear in areas with many details (e.g. see Figure 3.7), the parameter Minimal Distance Threshold sets the distance to that value if it is lower than specified. Figure 2.4 and 3.8 showcase the implemented translucency approach.

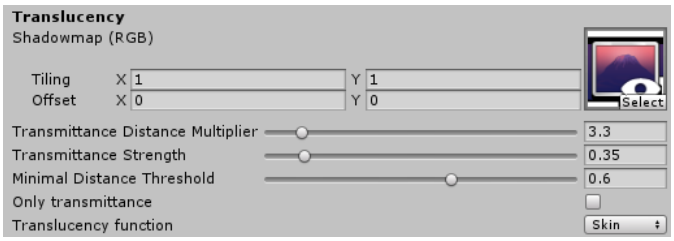
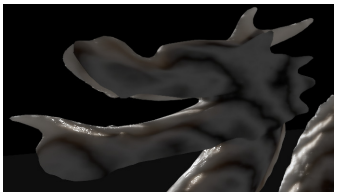


Figure 3.6: Some parameters are exposed in the editor to change translucency settings.

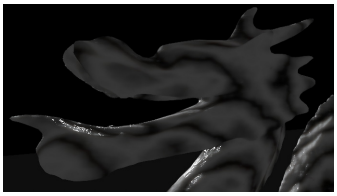
(a) Artifacts are visible when no threshold is set.

(b) No artifacts appear on this ear with a minimal distance threshold of 0.6.

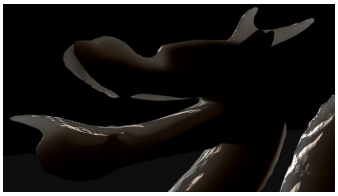
Figure 3.7: To resolve artifacts for detailed geometries that appear when translucency is enabled, minimal distance thresholds have been implemented.



(a) Translucency enabled



(b) Translucency disabled



(c) Translucency only

Figure 3.8: These images showcase the implemented translucency approach. The head of the Stanford Dragon is lit from behind.

3.4 Specular model

As mentioned in Section 3.2, a second camera exists in the project that is responsible for creating a mask to cull SSSS on the screen and to separate the specular reflections from the diffuse color information. This camera called 'SpecularHighlightsCamera' renders the objects SSSS is applied on with a replacement shader called `SpecularModel`. Unity forwards parameters from a material that has another shader attached to it to the replacement shader if the defined properties in the attached shader match the variables in the replacement shader. The editor exposes the parameters visible in Figure 3.9.

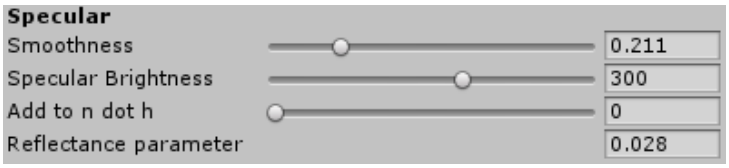


Figure 3.9: Parameters are exposed in the editor to configure the specular reflections of an object.

This thesis uses the Kelemen/Szirmay-Kalos model with the optimizations by d'Edon and Luebke [dL07]. The precomputed Beckmann-Texture is generated in the class `SpecularCamera` that is attached to the camera as a component. The replacement shader wasn't implemented as a surface shader. Although Unity allows the definition of own lighting models that can be used in surface shaders, in combination with a replacement shader this led to the problem that additive lights in the scene were exposed in the shader with the same values as the base light in the scene, which is usually the first directional light, even if it is disabled.

To provide support in Unity for directional, spot and point lights, two variants of each vertex and fragment functions must be implemented. The base pass is used for the first directional light and the additive pass is used for each additional pixel light. The difference for this implementation between the two passes is that the shadow attenuation, that is calculated through a macro provided by Unity, is additionally dependent on the world position of the fragment in the additive pass. Everything else other than the fragment function is shared between the two passes with the file `SpecularModelShared.cginc`.

The parameter 'Smoothness' controls the roughness of the surface and 'Specular Brightness' is used to scale the intensity of the specular reflection. 'Add to n dot h' is a value that is added to the dot product of the surface normal and the half vector of view and light directions. This was added for debug purposes but can also be used to increase the dot product artificially which allows reflection of light from wider angles. The reflectance parameter is a value that needs to be changed depending on the material, as explained in Section 2.3. For the human head a value of 1.4 was used which results in $F_0 = 0.028$. The Stanford Dragon model is assumed to be made out of marble, which has a refractive index of 1.486 and the corresponding $F_0 = 0.038$.

3.5 Virtual Reality Support

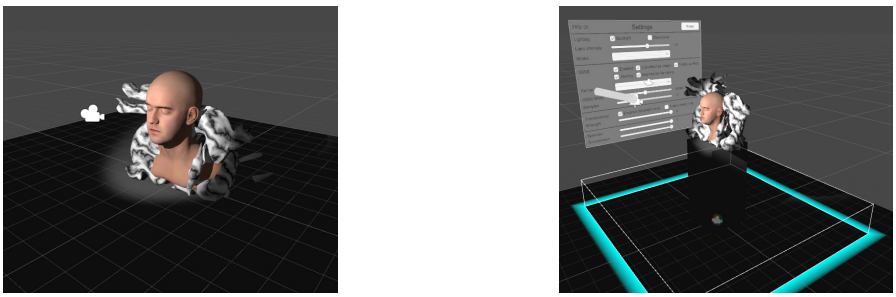
To provide support for VR devices Unity offers built-in functionalities. This includes stereoscopic rendering and head-tracking. VR systems like the HTC Vive HMD and the belonging controllers offer many more features like room-scale tracking. For this reason, the SteamVR framework was utilized to simplify the use of room-scale tracking and the controllers. This also comes with the big advantage to make the application compatible with other SteamVR compatible VR devices like the Oculus Rift. This application has only been tested with the HTC Vive but should theoretically work with any other SteamVR compatible headset. To utilize it in a Unity Project one must have SteamVR (available in from the Steam client under Tools `https://store.steampowered.com/about/`) and the SteamVR plugin for Unity (available from the Unity Asset Store `https://assetstore.unity.com/packages/templates/systems/steamvr-plugin-32647`) installed.

After the implementation of the approaches used for this thesis, as explained in the previous sections, the project has been adapted to work with stereoscopic rendering and changes were made to deliver a comfortable experience for the user.

3.5.1 General changes

To provide a smooth experience for the user, not only shaders and scripts must be changed or adapted.

Scene rearrangement The scene that is loaded if an HMD is connected is different from the scene without. For room-scale VR the scene had to be rearranged in a way so that a user can view the scene without too much effort. This includes placing the objects on a pillar so that they are higher in the scene and more comfortable to watch. The user interface exists now in world space and GameObjects provided by SteamVR have been added to enable the room-scale tracking of the HMD and the controllers. Figure 3.10 shows how they look in the scene view of the editor.



(a) Non-VR scene (b) VR scene

Figure 3.10: Which objects are in a scene and how they are arranged differs for the VR and non-VR scenes.

User interface To make the user interface interactable in VR, it is now an object in the virtual world that appears like a panel on a wall. Furthermore, the user can interact with it by pointing at the user interface with a controller. If a controller is directed towards the area of the UI, a line appears from the controller to the UI that acts as a pointer. The trigger can be used to click a button or move a slider.

Controllable spotlight If the spotlight is activated a flashlight is visible in the scene that represents the light source. If the user presses the grip button, the flashlight is attached to the controller and the user can move the light around freely around the model. Pressing the grip button again will freeze the controller in place.

Choosing a scene on application start-up When the application is started a scene is loaded that only contains a GameObject with a script that checks with the help of the functionalities provided by SteamVR whether an HMD is connected and loads the appropriate scene. If the application is started with the command line parameter 'force_no_vr', the application will load the non-VR scene even when VR hardware is connected to the computer.

3.5.2 Specular texture

The specular texture that holds the specular information and is used for culling the effect, as explained in Section 3.2, must be changed. This is the case because for VR there exist two different views of the scene, one for each eye. To provide support for stereoscopic rendering the application could be extended to use two textures or store the information of both eyes in one texture. For this implementation the latter approach was chosen. When the scene is loaded the script `SpecularCamera` sets the size of the texture to the resolution of the HMD. The value is provided by the SteamVR library. If the script is instructed to render it clears the texture, renders the scene with the replacement shader for the left eye on the left half of the texture and the right eye afterwards on the right side. Figure 3.11 shows how the specular texture looks for VR.

(a) View of the left eye

(b) Specular texture information for both eyes

(c) Culling mask for both eyes

Figure 3.11: If VR is enabled, the specular texture contains the specular reflections and culling mask for both eyes.

3.5.3 Rendering the screen-space effect

As mentioned earlier, if one uses multi-pass stereoscopic rendering, everything is rendered twice. One image is created for each eye. This also implies that the function `Render` from the class `SeparableSubsurfaceScatteringRenderer` is called twice. The object `context`, provided as a function parameter, contains information which eye is currently rendering. Unity starts rendering the left eye first. If VR is enabled, `Render` instructs the instance of `SpecularCamera` to render the specular texture and the shader `SeparableSubsurfaceScattering` reads specular and culling information only from the left half of the texture. On the second call for the right eye, the shader is instructed to use the right side of the texture. The rest of the shader stays the same.

CHAPTER 4

Results

This chapter will discuss the quality of the effect and the performance it achieves.

4.1 Evaluation

The goals of this thesis encompass finding out how SSSS can be implemented in Unity and if there are noticeable artifacts, when it is used in conjunction with stereoscopic rendering. The chosen way of the implementation proved itself flexible and combinable with the approaches for the specular model and the translucency approach. While inspecting the filter with a HMD no additional visible artifacts were perceptible. The usual ones described in the previous sections were less noticeable in VR compared to inspecting them on a normal monitor.

The most striking artifacts are made up of the banding that appears in the transition zones between lit and shadowed areas. There are two possibilities to overcome these. The jittering was added specifically to reduce the visible banding. The applied jitter itself becomes visible if one gets close to the parts were banding appears but is an improvement compared to the banding. In VR jittering is even less noticeable than the banding. Figure 4.2a displays banding artifacts where the kernel has a sample size of 17. The following image, Figure 4.2b shows how the jittering changes the appearance. An alternative to jittering is increasing the kernel's sample size. This comes with the disadvantage of increased performance. Section 4.2.2 covers the relation between performance and sample size. Figure 4.2c shows the same spot on the Stanford Dragon without Jittering but with a sample size of 51. If one enables jittering with that number of samples, Figure 4.2d shows no improvement compared to no jittering. At a certain number of samples jittering does not improve the image but rather reduces visual quality.

If one wants to use this implementation of SSSS, there are multiple things to consider. It was designed to use the layer functionalities in Unity to decide on which objects the effect

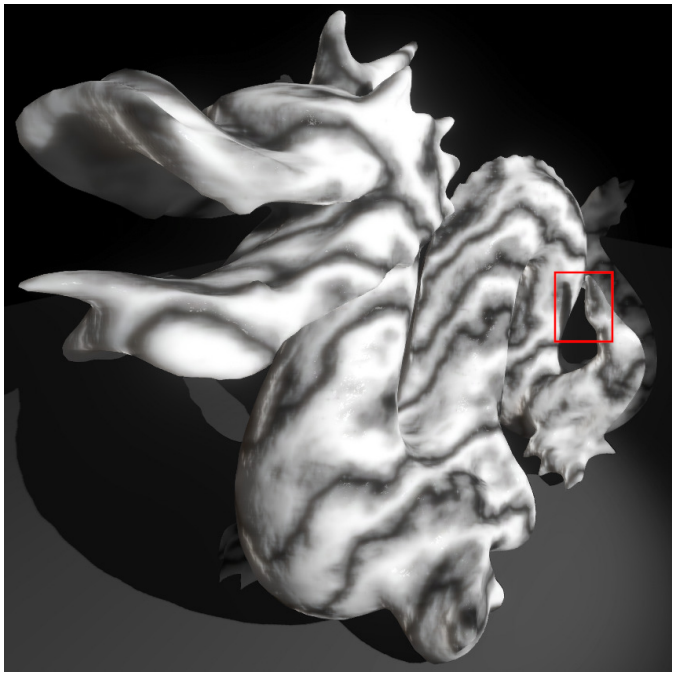


Figure 4.1: Whole image of the Stanford Dragon that shows where the snippets in Figure 4.2 are taken from.

should be applied on. This allows to easily add more objects to scenes. If one wants to use different kernels for different objects, a possible solution is explained in Section 5.2. Furthermore, the specular part and the culling mask are generated in one pass and are stored in the same texture. If another specular model should be used, one can either rewrite the shader `SpecularModel` or add it to the surface shader that draws the object itself. If one uses the latter option, specular reflections and SSSS aren't separated anymore. The custom shader needs to have the parameters of the specular model shader if one wants to use another shader for the surfaces of the objects. Otherwise, the specular reflections can't be adjusted in the editor. Adding the translucency to the objects also needs to be implemented for shaders other than the implemented surface shader. These restrictions and the need to setup additional cameras means that users need knowledge about Unity, how materials work with replacement shaders and how SSSS works.

4.2 Performance

The performance of the effect is an important part to consider especially when used in an engine. In real usage scenarios the effect itself is not the main part of the application. Therefore, this implementation of SSSS was tested with different configurations.

(a) Jittering disabled, 17 samples

(b) Jittering enabled, 17 samples

(c) Jittering disabled, 51 samples

(d) Jittering enabled, 51 samples

Figure 4.2: Jittering and increasing the sample size of the kernel are ways to overcome banding artifacts. The images show a snippet of the Stanford Dragon. The whole image is depicted in Figure 4.1.

4.2.1 Methodology

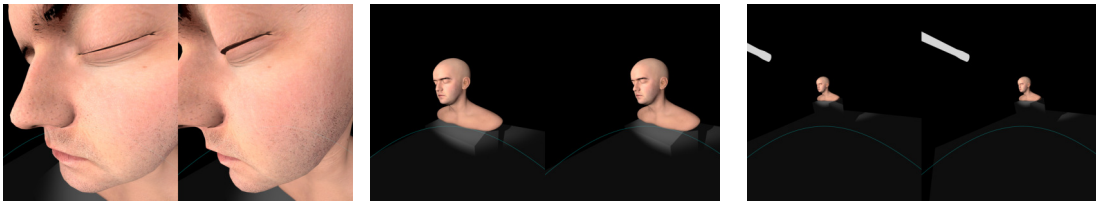
To evaluate the performance with different settings, a script was created to measure the frame time over a period of time. The script takes the time of each frame over two seconds with half a second delay between each test. After all tests have been processed the average frame time of each test is stored in a text file called `report.txt`. The actual tests have not been executed in the editor, because the editor represents overhead that affects performance. Tests have been done with SSSS enabled/disabled and with/without

CPU	AMD Ryzen 7 1800x, 8x3.60 GHz
RAM	32GB
OS	Windows 10 64bit
GPU	Nvidia GeForce GTX 1070
HMD	HTC Vive

Table 4.1: These were the specifications of the computer used for testing.

VR. The tests without SSSS were done to have a baseline reference of how much impact the effect has on performance in terms of frames per second (FPS). The SSSS tests use sample sizes of 7, 17, 25 and 51 with different distances from the model to consider how many pixels on the screen the effect is applied on. Figure 4.3 shows the different distances from the model used in the tests. This leads to twelve different scenarios for the VR and non-VR build. The window for the tests was maximized which resulted in a resolution of 1920x1017. The specifications of the computer used for testing are listed in Table 4.1.

(a) Close No-VR (b) Middle No-VR (c) Far No-VR



(d) Close VR (e) Middle VR (f) Far VR

Figure 4.3: To test how the performance of the implementation changes, depending on the amount of pixels on the screen covered with the effect, these distances were used for the different setups.

4.2.2 Results

The results of the tests are presented as graphs that show the achieved FPS in conjunction with the sample size. Figure 4.4 displays the non-VR tests and Figure 4.5 the VR tests. Both graphs show that the performance scales not well with the amount of pixels on the screen that are covered by the effect. Especially the non-VR results with sample sizes of 25 and 51 reach noticeable high frame times for the close setup. When wearing a HMD, even little differences in the frame rate can lead to uncomfortable experiences.

During normal testing the frame drops were only noticeable when getting really close to the object. For example, in a real usage scenario this can happen if a user wants to inspect how SSSS affects small details like pores. When inspecting an object in a VR application a user often gets close to observe smaller details.

It is also important to mention that the HMD caps the frame rate at 90 FPS, which is the reason why the maximum recorded frame rate in Figure 4.5 is at that value. Comparing the results for the close distance from both graphs, one can observe that stereoscopic rendering takes a big toll on the performance. The reason for this big difference in frame times is the multi-pass rendering and the bigger resolution of the HTC Vive. Twice the amount of draw calls and a resolution of 2160x1200 for both eyes lead in this case to less than half FPS for each sample size.

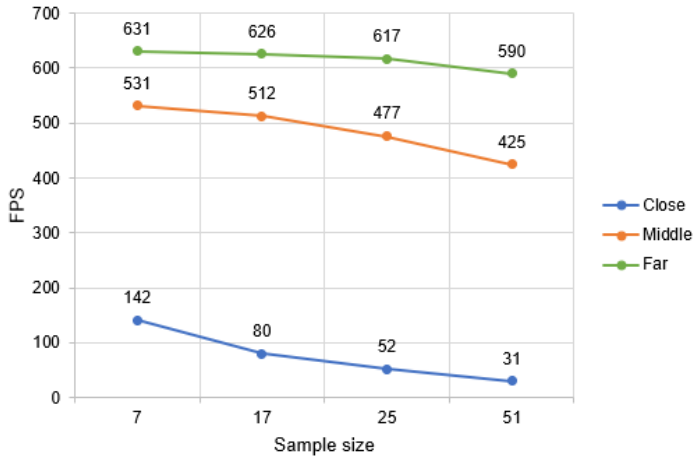


Figure 4.4: This diagram depicts the results of the non-VR tests. SSSS-Off tests achieved 621/740/790 FPS for close/middle/far.

The amount of draw calls is an important number to look at if one wants to analyze performance of an application. Algorithm 4.1 provides a listing of how an image is rendered for non-stereoscopic rendering. Since this project utilizes multi-pass VR, the draw calls of the specular texture, the rendering of the scene itself and the SSSS post-processing effect are doubled. For the setups used in this application, the non-VR scene is rendered in roughly 40 to 50 draw calls and the VR scene in 100 to 110. Both depend on what is currently visible on the screen.

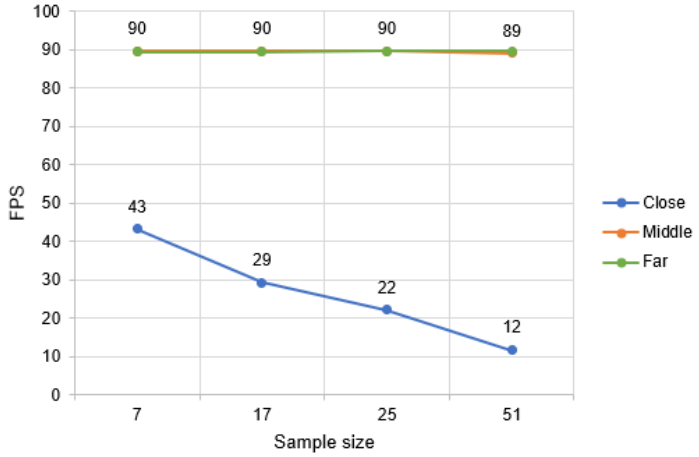


Figure 4.5: The VR results are different compared to the non-VR tests. Because the screen is capped at 90 FPS this value is not exceeded. All SSSS-Off tests were able to achieve 90 FPS.

Algorithm 4.1: This listing contains a simplified outline of the draw calls that result in the final picture.

1: Shadow map used for the spotlight that is responsible for translucency
2: **Specular texture**
3: SSSS model is rendered two times (Base and Add passes for directional and spot/point lights)
4: Other objects are drawn onto the same texture with a shader that sets the fragment black, this is done to prevent SSSS being applied on sections of the screen which are not currently obstructed
5: **Rendering the scene**
6: Each object is drawn two times (base and add passes)
7: **Post-processing**
8: AA and Bloom if enabled
9: Two passes for SSSS
10: UI is drawn

CHAPTER 5

Conclusion

5.1 Summary

This thesis presented an implementation of the technique developed by Jimenez et al. Unity offers enough functionality to add SSSS and the other implemented techniques into a project. The solution can be shared between projects and the kernel settings can be adjusted in the editor and tracked via source code management. This is a plus for developing applications in a team environment. Furthermore, it can be adjusted and extended. In terms of quality no additional artifacts were visible when the effect was inspected with stereoscopic rendering. With VR enabled the performance can dip very low when a major fraction of the screen is covered in the effect. Adjusting the quality of SSSS via the settings and the possible optimizations explained in Section 5.2 can help improve the performance. Summing up one can say that the approach is definitely viable for usage in a game or application made with Unity. As a Post-processing effect it offers possibilities to implement and adjust it in a way to fit in a project.

5.2 Possible Extensions

There are multiple ways the implementation can be extended with additional functionalities and optimized for more performance. To improve the performance for stereoscopic rendering a big improvement would be to adapt the project to support single-pass VR rendering. This could potentially half the amount of draw calls and make SSSS more viable for VR. Unity offers shader macros to help with adapting the shaders. For this implementation the rendering of the specular texture would also need changing.

The current application only supports one kernel for all pixels covered on the screen. Since SSSS can also be applied on other materials than human skin, one could extend the implementation to use different kernels for different sections on the screen. This could be

done with the specular texture that is also used to cull the convolution. If the specular part only needs one color channel the other two channels or the alpha channel can be utilized to encode which kernel should be used for which pixel. To separate the objects the layer functionality in Unity can be used to determine which kernel should be used for which object.

List of Figures

List of Tables

List of Algorithms

Acronyms

BRDF bidirectional reflectance distribution function. 10, 11

FPS frames per second. 32–34, 38

HMD head-mounted display. 2, 13, 25, 26, 29, 32, 33

SSSS Separable Subsurface Scattering. 1–3, 7, 13, 15, 17, 19–21, 24, 29–35, 37, 38

VR Virtual Reality. xi, xiii, 1–3, 13, 15, 16, 25–27, 29, 32–35, 37, 38

Bibliography

[BV17] Leif P Berg and Judy M Vance. Industry use of virtual reality in product design and manufacturing: a survey. *Virtual Reality*, 21(1):1–17, 2017.

[dL07] Eugene d'Eon and David Luebke. Gpu gems 3–advanced techniques for realistic real-time skin rendering, 2007.

[dLE07] Eugene d'Eon, David Luebke, and Eric Enderton. Efficient rendering of human skin. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 147–157. Eurographics Association, 2007.

[gea] Unity game engine. Single-pass stereo rendering. *Online][Cited: July 15, 2018.] https://docs.unity3d.com/Manual/SinglePassStereoRendering.html*.

[geb] Unity game engine. Unity game engine-official site. *Online][Cited: July 15, 2018.] https://unity3d.com/unity/*.

[Gra08] Herbert Grasberger. Introduction to stereo rendering. *Student Project, Institute of Computer Graphics and Algorithms-Vienna University of Technology*, 2008.

[JSG09] Jorge Jimenez, Veronica Sundstedt, and Diego Gutierrez. Screen-space perceptual rendering of human skin. *ACM Transactions on Applied Perception (TAP)*, 6(4):23, 2009.

[JWSG10] Jorge Jimenez, David Whelan, Veronica Sundstedt, and Diego Gutierrez. Real-time realistic skin translucency. *IEEE Computer Graphics and Applications*, 30(4):32–41, 2010.

[JZJ+15] Jorge Jimenez, Károly Zsolnai, Adrian Jarabo, Christian Freude, Thomas Auzinger, Xian-Chun Wu, Javier von der Pahlen, Michael Wimmer, and Diego Gutierrez. Separable subsurface scattering. *Computer Graphics Forum*, 34(6):188–197, 2015.

[KSK01] Csaba Kelemen and Laszlo Szirmay-Kalos. A microfacet based coupled specular-matte brdf model with importance sampling. In *Eurographics Short Presentations*, volume 2, page 4, 2001.

[MKBKR17] Jessica L Maples-Keller, Brian E Bunnell, Sae-Jin Kim, and Barbara O Rothbaum. The use of virtual reality technology in the treatment of anxiety and other psychiatric disorders. *Harvard review of psychiatry*, 25(3):103–113, 2017.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[SVS05] Maria V Sanchez-Vives and Mel Slater. From presence to consciousness through virtual reality. *Nature Reviews Neuroscience*, 6(4):332, 2005.