FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Extending LEAN Mapping to Subsurface Scattering

**Tiago Vila Verde**


Mestrado Integrado em Engenharia Informática e Computação

Supervisors: Rui Rodrigues (PhD) & Marc Olano (PhD)

June 2015

# Extending LEAN Mapping to Subsurface Scattering

**Tiago Vila Verde**

Integrated Masters in Informatics and Computing Engineering

---

June 2015

# Abstract

The goal of this thesis is to extend LEAN Mapping to incorporate the Subsurface Scattering effect which exists on translucent materials, such as marble or skin, while producing a model that is sufficiently simple in order to be easily incorporated into current art assets and pipelines.

For the success of this project it is therefore important to know what is the current state of shading models in terms of the subsurface scattering effect, as well as what is LEAN mapping and how it can be incorporated into popular models. This document provides an overview of the most important concepts for such an understanding.

Typically, a subsurface scattering effect is obtained by blurring the diffuse texture, however, previous works have found that, by having each individual RGB normal pointing in different directions, it is possible to achieve a realistic scattering effect. This means that it could, potentially, be possible to apply those blurs to normal maps and have similar results. By applying it to the LEAN maps, we can further refine the process to include more detailed specular highlights.

This can be done by having a deferred rendering approach and applying the blur to the G-Buffer. However, object curvature can be important depending on how curved the object is and how deep the light penetrates. By projecting onto a common plane, it is possible to capture that information and the blur can then be applied.

Result comparison with previous works shows the elimination of some artifacts. Nevertheless, in frame rate terms, there is a significant decrease in performance, attributed to the number of textures being blurred. On average, rendering a frame takes about four times longer.

In short, the work presented in this document is valid and shows some promise in subsurface scattering contexts, however, it can certainly be improved in the future.

# Resumo

O objectivo desta tese passa por estender *LEAN mapping* para incorporar o efeito da dispersão da luz sob a superfície (*Subsurface Scattering*) existente em materiais translúcidos como o mármore ou a pele, criando um modelo suficientemente simples de incluir nos actuais fluxos de desenvolvimento de aplicações gráficas.

Para o sucesso deste projecto é, então, importante perceber qual é o estado da arte dos modelos de *shading* em termos desse efeito, assim como o que é *LEAN mapping* e como pode ser aplicado a esses modelos. Este documento apresenta uma visão global dos conceitos mais importantes.

Tipicamente, a dispersão da luz sob a superfície de um material é obtida através da aplicação de filtros de esbatimento (*blur filters*) nas texturas responsáveis pela cor do objecto. No entanto, trabalhos anteriores concluíram que, ao ter cada normal *RGB* a apontar em diferentes direcções, é possível obter um efeito de dispersão mais realista. Isto significa que, potencialmente, é possível aplicar um filtro de *blur* aos *normal maps* e obter resultados semelhantes. Se estes filtros forem aplicados aos *LEAN maps*, pode-se ainda obter melhores resultados devido à inclusão dos detalhes de especularidade.

Este processo pode ser implementado através de uma abordagem de renderização adiada (*deferred rendering*) e aplicar o filtro ao *G-Buffer*. Porém, a curvatura de um objecto pode ser importante dependendo da profundidade a que a luz penetra a superfície e do quão curvo esse objecto é. Ao projectar a informação do objecto para um plano comum, é possível capturar a informação necessária e, então, o filtro pode ser aplicado.

Comparando com resultados de trabalhos relacionados nota-se que foi possível remover alguns artefactos mas, em termos de *frame rate*, há uma discrepância significativa pois esta nova técnica é, sensivelmente, quatro vezes mais lenta do que a técnica à qual foi comparada. Esta redução pode ser atribuída ao número de texturas a que o filtro de *blur* é aplicado.

Resumindo, o trabalho apresentado neste documento é válido e mostra alguns pontos interessantes em contexto da dispersão da luz sob a superfície. No entanto, há aspectos que podem, certamente, serem melhorados.

# Acknowledgements

I would like to thank Dr. Rui Rodrigues for accepting to supervise me while I work on this project.

I would also like to thank Dr. Marc Olano for the amazing opportunity to travel to the United States to work on this project at UMBC. Furthermore, I appreciate all the help and time he expended while mentoring me.

This project would not have succeeded were it not for these two people.

Tiago Vila Verde

# Content

# Table of Figures

# List of tables

# Notation

| | |
|---|---|
| RGB | Red, Green and Blue |
| LEAN | Linear Efficient Antialiased Normal |
| BRDF | Bidirectional Reflectance Distribution Function |
| $\vec{l}$ | Vector pointing in the incoming light direction |
| $\vec{o}$ | Vector pointing in the reflected light direction |
| $\vec{n}$ | Surface normal |
| $\vec{v}$ | Vector pointing in the direction of an observer |
| $h$ | Half angle between $\vec{v}$ and $\vec{l}$ |

# Chapter 1

# Introduction

Moore's Law states that computing power doubles every two years. When observing this rule within the realm of Computer Graphics it generally means that graphics can become more photorealistic with each passing year.

Models and techniques that would take hours to render an image can then be rendered in a fraction of a second. This opens the door to including more detail into a scene, however, if exaggerated; this raises the computational power required. This never ending cycle means that new and better techniques need to be developed to properly approximate physical based models in order to render photorealistic scenes in real-time.

However, visual fidelity is not the only important point of a good shading model. The fact is that, already, there are many models that produce a wide range of results, from realistic to outright cartoonish, but their use is scarce. The reason is the complexity of such models. It is then no surprise that the most widely used shading model was first introduced by Bui Tuong Phong (Phong, 1975). It has stood the test of time not for being the most accurate but, rather, because it produces sufficient results with an extremely simple math supporting it.

As will be explained in this document, most subsurface scattering effects are achieved by means of performing a Gaussian blur. This blur is either applied in texture space or screen space, as a post processing effect. The first case, while more accurate, is also computationally more expensive, whilst the second, while faster, loses some accuracy, especially in very curved objects, as it cannot really take into account the effects of scattered light in that curvature.

LEAN mapping (Olano & Baker, 2010) tackles a similar conundrum where, at large viewing distances, rough, bumpy but shiny surfaces seem flat, yet the lighting effect from the bumps is still there. However, lighting models interpret the surface as being flat and so, at a distance, sharp highlights appear, instead of an overall duller feel. Their solution was to project the relevant data onto a common plane, thereby preserving it even at large viewing distances.

This approach can be adapted for this particular problem in order to capture the curvature of the object by projecting onto a common plane. Once that is done, the blur filter can then be applied, followed by the standard lighting computations. Furthermore, using the LEAN maps provides an advantage, in terms of specularity, as a more realistic effect can be computed.

Finally, projecting information onto a different plane so that it can be used later is similar to deferred rendering approaches. This means we can devise a solution that is even more in line with current, modern graphic applications, which make heavy use of deferred rendering.

## 1.1 Motivation

The main motivation for this thesis is a desire to learn more about computer graphics and how scenes are rendered.

Beyond that, making a scientific contribution that helps advance the field one is interested in might be a daunting task, but the sense of achievement in the end more than makes up for it.

The prospect of coming up with a solution that could, potentially, become widely used is also a strong incentive.

## 1.2 Goal

The goal of this thesis is to extend LEAN mapping, a method for normal filtering, to include the subsurface scattering effect found in translucent materials.

Whilst LEAN mapping solves the problem of aliasing artifacts when filtering normal maps in a simple way, it does not take into account what happens when light penetrates the surface of these materials. As such, these materials are very difficult to render, as will be explained further ahead, especially when considering highly detailed and curved surfaces.

Existing models are good enough to obtain realistic results but are still somewhat complex, especially when compared with the Phong model (Phong, 1975). Therefore, by trying to tweak the terms of the LEAN mapping model, it is hoped that a sufficiently simple model can be produced that produces realistic results and is compatible with current real-time graphic technologies, such as game engines, as well as current art assets and pipelines.

## 1.3 Expected Results

Given that the LEAN maps encode the normal and microfacet distribution information and, bearing in mind Debevec's work ( 2010), it is hoped that by applying the previously mentioned sum of gaussian blurs on these LEAN parameters, a subsurface scattering effect can be achieved. Ideally, the results should be consistent with what Jimenez (2009) obtained. However, a major difference in the specular highlights should be observed, depending on the MIP level being used, due to the specularity computations from the original LEAN mapping results. Furthermore, it is intended that the halo found in that same work can be eliminated, or, at the very least, have its effect mitigated.

In short, the materials are expected to have a soft appearance, with what would, otherwise, usually be dark areas presenting a tint in accordance to the colour of the light that gets scattered the most.

## 1.4 Structure

In order to present the project in proper fashion, this report is divided into 4 main chapters, along with the introduction and conclusion: related work, LEAN mapping and subsurface scattering, implementation and result analysis.

Within the related work, the report will follow a logical train of thought in order to present some important concepts and the works that first introduced them. It will start off by explaining what lighting is and what it is for. That can be thought of as the tip of the iceberg. From there, the report will keep going deeper into the topic at hand, explaining the necessary relevant details, such as shading models and techniques, along the way.

For the third chapter, an overview of the solution will be given, coupled with a requirements list, the reasons for those requirements and a detailed explanation of the technological choices that were made.

For the implementation section, the problem at hand is described in detail. Each individual step of the process is properly explained. Furthermore, a description of the developed framework that enabled the development and testing of this project is also provided.

Finally, the last section compares the obtained results to previous techniques and discusses the advantages and limitations of this new technique.

# Chapter 2

# Related Work

In this section the most relevant work for the purposes will be mentioned and, to some extent, explained. This is done by presenting the concepts important to this thesis and explaining how previous work dealt with them. Mostly, this will focus on recent work on real-time photorealistic rendering of human skin - and the techniques involved in it - as well as methods for producing highly detailed bumpy surfaces with proper specular reflection.

## 2.1 Image Processing

Modern graphics pipelines enable the ability to render the output of the shader onto a texture. Once we have that capability, we can apply typical image processing techniques to attain the desirable outcome. In this section, the most pertinent techniques for this dissertation are presented.

### 2.1.1 Gaussian Blur

A very popular effect used for a variety of purposes is the blurring of an image. Using the well-known Gaussian function is the standard way to apply this effect in the computer graphics context. In short, we sample the nearby pixels, with the amount of pixels sampled defined by whatever kernel size is selected, and then use a weighted

average to get the final result, with the weights coming from the Gaussian function where the input is radius of the blur.

### 2.1.2 Bilateral Blur

Bilateral blurs are commonly used in edge detection software. A simple Gaussian blur will indiscriminately average out pixels. This means that it does not matter where the pixels come from, they could represent an object and the background and, with this blur, they will still be averaged together. For some purposes it is necessary to preserve the edges, hence the use a bilateral blur. This is simply multiplying the Gaussian weight of a normal blur by another Gaussian weight computed from some variable other than the radius. For example, it might not be desirable to blur between vastly different colours, so, in this case, the input to compute the additional weight could be the absolute difference between the sampled pixels.

## 2.2 Rendering Graphics

A key component of modern graphics pipelines is the ability to perform several shader passes. This capability enables the use of several strategies when rendering. These are described in this section.

### 2.2.1 Forward Rendering

Forward rendering is the traditional method of rendering images. The shaders receive their input resources, perform all necessary operation and return the final image in just the single pass. It was the only approach available before the introduction of multi pass shaders.

### 2.2.2 Post Processing

Post processing refers to any image filter applied to the final rendered scene. Examples, in 3D graphics context, include motion blur, bloom and depth of field. To apply these filters, the scene is rendered onto a texture which is then used as a resource for the next shader pass, which is responsible for the final effect.

### 2.2.3 Deferred Rendering

Modern intensive graphics applications can squeeze out better performance by deferring the lighting calculations, which, in general, are quite computationally expensive, to a later stage, using the first pass to simply gather the necessary information from the 3D objects and output it to buffers which can then be read by the final pass. In this context, the textures that store all the information, which, at very least, must contain the world position, normal, the pixel colour and the depth value, are commonly referred to as the G-Buffer.

## 2.3 Lighting and Shading Models

One of the processes through which depth perception and a sense of realism can be added to a scene what is called Lighting. The way different surfaces reflect light is what helps us determine what kind of material that surface is as well as its shape.

Shading models are what computes the way light interacts with a particular surface by figuring out the amount of light reflected and its direction.

In this section some of the concepts used in many common shading models will be presented.

### 2.3.1 Specularity

In a perfectly smooth surface, such as a mirror, the angle between an incoming ray of light $\vec{\imath}$ and the surface normal $\vec{n}$ is equal to the angle between that normal and the outgoing ray $\vec{o}$. This means that when light hits the surface it is reflected in precisely one direction.

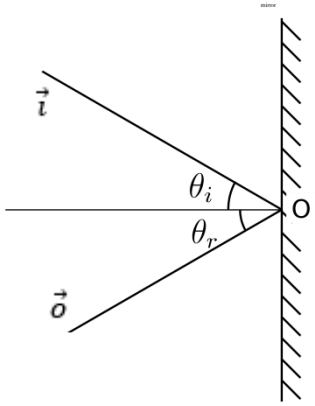

Figure 1 - An incident ray $\vec{\boldsymbol{\imath}}$ produces an angle with the surface normal equal to the one produced by the outgoing ray $\vec{\boldsymbol{o}}$ with the surface normal. This produces a specular reflection. Note the surface is a mirror.

Whenever an observer's viewing direction $\vec{v}$ is perfectly aligned with $\vec{o}$ then a specular highlight can be perceived by the observer. In a mirror-like surface the highlight's shape will be that of the incoming source of light.

However, many shiny surfaces show blurred highlights. This is due to microscopic irregularities in the surface which scatter light in different directions.

### 2.3.2 Microfacets

To represent roughness it is assumed that surfaces which are not perfectly smooth are composed of a lot of tiny surfaces, designated as microfacets, oriented in a variety of directions with each one being a perfect mirror-like specular reflector. The orientation of these facets, and, therefore, their normals, is given by a probability distribution around the normal of the overall smooth surface. In points where the direction of $\vec{h}$ is close to that of $\vec{n}$ a large number of microfacets will point in that same direction which results in a bright, intense specular highlight. As one moves further out from the centre of the highlight the direction of $\vec{h}$ starts to also move away from $\vec{n}$ which means less microfacets pointed along $\vec{h}$ resulting in a dimmer highlight, eventually falling off to zero.

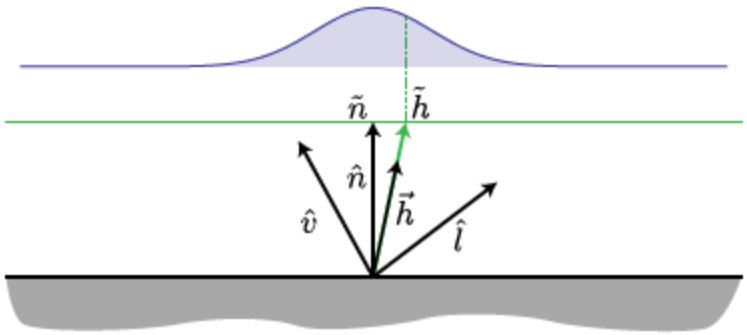


Figure 2 – Illustration of a distribution of microfacets and how the half-angle is used to figure out the intensity of the specular highlight

Phong ( 1975) first proposed this method and it has since become a staple for shading models, with many improvements and variations made over the years (Blinn, 1977; Cook & Torrance, 1981; Ward, 1992). The Beckmann distributions seem to work reasonably well in achieving realistic results as shown by Ward (1992).

### 2.3.3 Diffuse Reflection

Previously, mirrors were used as an example to explain specular reflection since they reflect all incoming light. Most surfaces do not behave like that. Some light is absorbed, some reflected in random directions and the some specularly reflected. Diffuse reflection is the scattering of light in all directions, which is more noticeable in dull surfaces. This is due to tiny, microscopic irregularities. Such surfaces produce large specular highlights with a more gradual fainting whilst shiny ones, with predominantly specular reflection, result in sharp, intense highlights.

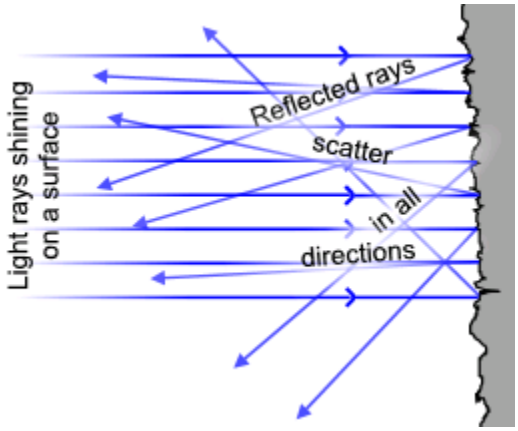


Figure 3 - Diffuse reflection from an irregular surface

Phong (1975) approximated this complex process with one variable, a diffuse constant, that would vary from surface to surface in order to accurately represent the intensity of the diffuse reflection in different surfaces.

### 2.3.4 Subsurface Scattering

Light diffusion is usually considered to occur at a material's surface but that isn't entirely correct. In fact, the process occurs *beneath* the surface, and is mostly the same

as the concept of Subsurface Scattering, which is introduced in this section. They key difference is that, for diffuse reflection, light only penetrates the surface in microscopic distances, which is why, for simplicity purposes, it is said to occur at surface level. On the other hand, in materials where light is able to penetrate more, such as human skin, wax and marble, the effect is considerably different and is referred to as Subsurface Scattering. These materials are said to be translucent.

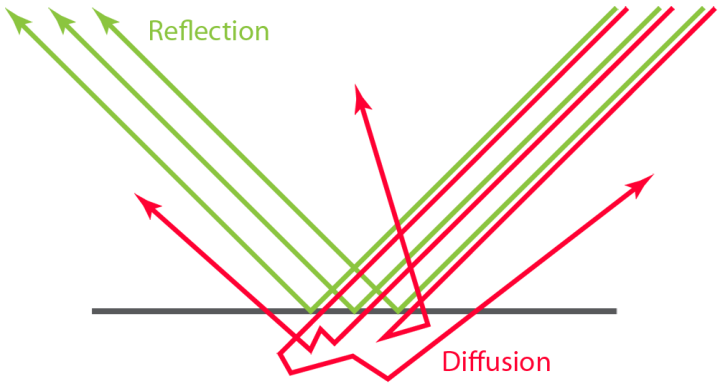


Figure 4 - Example of a translucent surface that allows light to enter the material, get scattered and then emitted back out

When light enters these materials, it gets absorbed, acquiring color, and scattered often, eventually returning to surface and exiting the material. If the material is translucent enough, light can even traverse through the whole object, exiting on the opposite side from where light is entering the material. This makes this kind of surfaces seem like they are emitting light.

Figure 5 - Example of how light scatters in a human hand. Notice the hand seems to emmit a reddish light

#### 2.3.4.1 Rendering Realistic Subsurface Scattering effects

Over the last few years significant advances have been made in rendering human skin. Work such as Jensen et al. (2001) demonstrated how lighting on translucent materials could be modelled through the use of a dipole function. Their model enables realistic rendering of human skin, however, it is too complex to be used for real-time rendering. More recently, d'Eon (2007) showed how one can approximate the previously mentioned work with a simple sum of Gaussian blur filters which results in real time rendering of photorealistic human faces. Although considered ground-breaking this process still has its flaws as the model becomes too resource intensive when rendering more than a few objects. Jimenez (2009) achieved interesting results by switching from texture space to screen space, allowing for a larger number of objects on a given frame.

This section will use the previously mentioned work to present the different aspects which need to be taken into account to properly render translucent materials, along with the challenges associated with such a task.

Note that most of these works focus on human skin. However, the same process can be applied to other materials such as marble.

#### 2.3.4.2 The appearance of skin

Skin is usually very difficult to render, particularly because of its high amount of detail. Subtle characteristics such as scars, wrinkles, pores, hair follicles and so on are very important when rendering skin because humans are particularly sensitive to them. In other materials this sort of small details can be overlooked and the rendered image will still look realistic. Whilst current scanning technology can properly capture all these details, rendering them is still a difficult process which can get a unrealistic, dry and hard-looking result, as exemplified by d'Eon et al. (2007).


Figure 6a - Improper rendering of skin
Figure 6b – Proper rendering accounting for Subsurface Scattering

Since skin is a translucent material, the missing component needed to be taken into account to prevent such results is Subsurface Scattering. The scattering and absorption of light in the skin's inner layers is what gives off its colour and warm, soft look.

Because skin is made up of many different layers, each of which absorbs and scatters light differently, it is quite complicated to accurately represent this process.

Donner & Jensen (2005) demonstrate how a single layer model is insufficient for accurately rendering skin. They also show how a three-layered model gives off satisfying results.

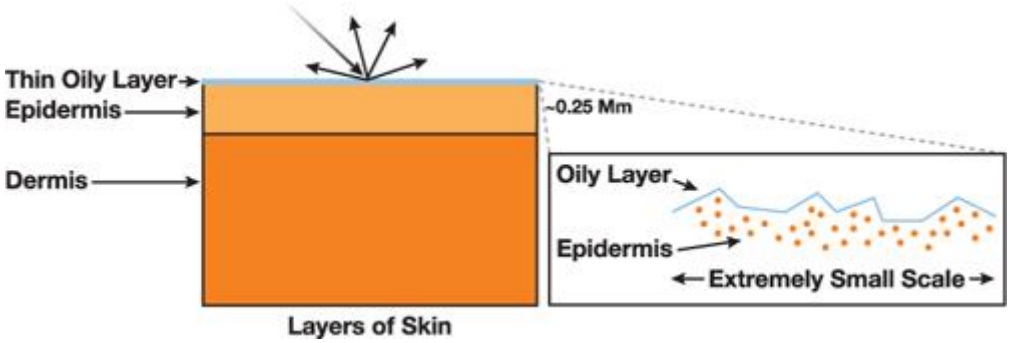


Figure 7 - A multi-layered skin model

This model consists of an upper oily surface, responsible for the specular reflection, while the two other layers scatter the light. Because light hitting tissue scatters very often and quickly, after a short amount of time light will be flowing in evenly in all directions, which is akin to what happens in diffuse reflection. Therefore, the subsurface scattering can be approximated with diffusion models, although these are still mathematically complex.

#### 2.3.4.3 Diffusion Profiles

In order to properly render a translucent material, it is required to understand the way light scatters and how far into the inner layers it is able to penetrate. A diffusion profile is an approximation of this and it is usually represented as a graph showing the amount of light emitted in regards to the radial distance from the point where incoming light hits the surface, assuming a highly scattering, translucent, flat surface. d'Eon (2007) illustrates this concept nicely by referring an experience where a light beam is aimed at a thin, flat, translucent surface in a dark room.

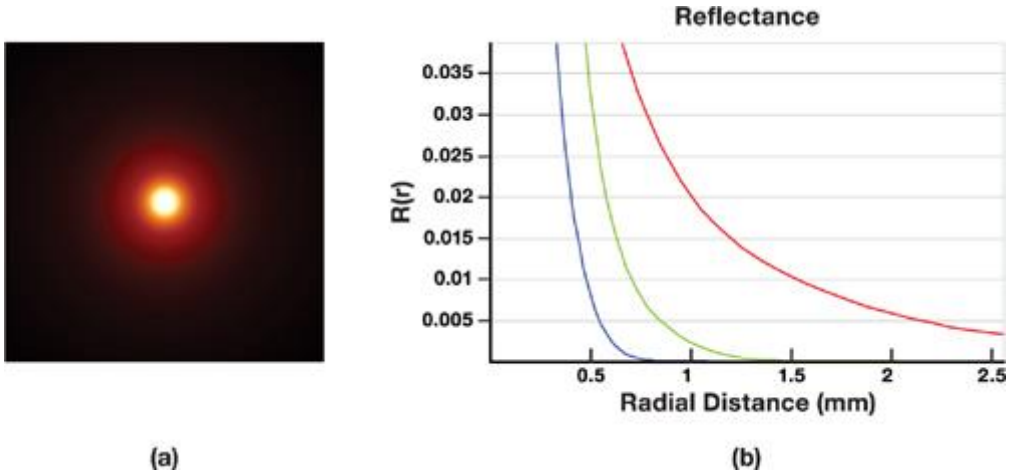


Figure 8 - d'Eon's experiment and resulting diffusion profiles

To properly render a translucent material, it is required to know the shape of the diffusion profile for that material. Jensen et al. (2001) use a dipole function to compute diffusion profiles for several materials, however, for multi-layered materials, a multipole model (Donner & Jensen, 2006) is required as the shape is too complex and the dipole function cannot capture the scattering differences of one layer on top of another.

#### 2.3.4.4 Sum of Gaussians Profile

d'Eon et al. (2007), found that the profiles computed from the mentioned experiment resemble the Gaussian function $e^{-r^2}$. He demonstrated how a weighted sum of these functions is enough to properly approximate most diffusion profiles.

To realistically render the subsurface effect the diffusion profile needs to be calculated for every location where light hits the object. The dipole and multipole function, while physically accurate, are too complex to be used for real-time purposes, which is why the sum of Gaussians approximation is so handy.

In his work, d'Eon shows how a sum of six Gaussians produces indistinguishable results from the multipole function for rendering a three-layered skin model.

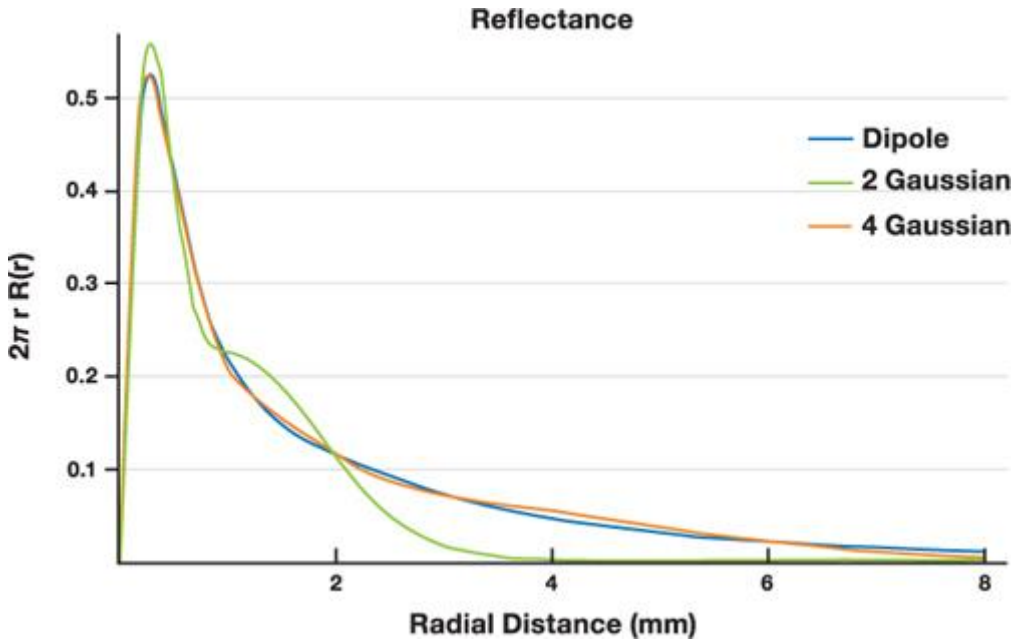


Figure 9 - A sum of Gaussians can approximate a dipole function extremely well

#### 2.3.4.5 Texture Space Diffusion

Borshukov and Lewis (2003) introduced a new technique for fast rendering of subsurface scattering effects. They essentially unwrapped a 3D mesh using the UV coordinates as the render coordinates and then modelled the scattering effect by applying a blur filter to the texture. Aligning the shape and intensity of the blur with a diffusion profile produces the expected effect.

In their work, two different terms were used for the blur to approximate the effects of different layers which introduced parameters, which an artist could tweak, that are not based on any physical model. D'Eon corrected this by substituting those terms with his sum of Gaussians approximation. Moreover, he improved on their work by extending the work of Green (2004), to incorporate transmission through thin regions where a significant amount of light can enter on one side, but exit on the other.

#### 2.3.4.6 Screen Space Diffusion

A disadvantage of Texture Space Diffusion is the increase in computational power needed when several textures are rendered in one frame. Screen Space Diffusion solves that issue by first rendering the frame without the scattering effects and then applying the required operations to the corresponding sections of the frame. This way, the heavy-lifting algorithm is run once rather that a number of times equal to the number of textures visible in one frame.

Jimenez et al. (2009) showed how moving from texture space to screen space is possible and produces almost indistinguishable results, particularly when it comes to human skin. He effectively applies the sum of Gaussians diffusion profile to a rendered diffusion texture instead of the irradiance maps used in d'Eon et al. (2007).

Figure 10 - Rendering of several faces with Jimenez's screen space technique

## 2.4 Bump mapping

Another important part of realistically rendering an object is the correctly displaying bumps, ridges and surface imperfections, which are noticeable unlike the previously mentioned microscopic imperfections.

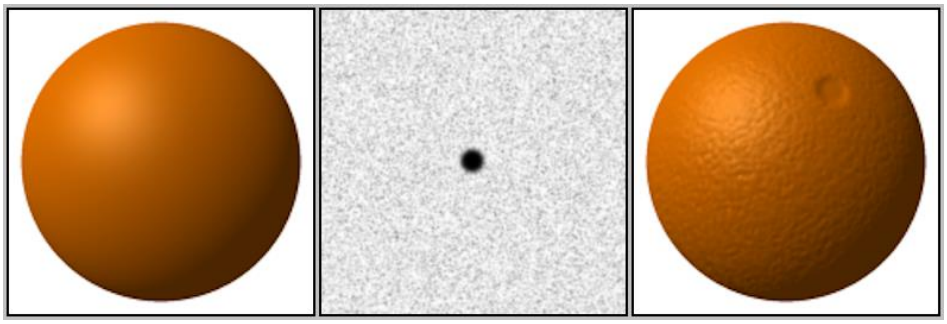

Figure 11 - The effect of a bump map applied to a sphere

Blinn (1978) first proposed Bump Mapping as a way to add detail to a surface without increasing the complexity of the 3D model. His method consists of a height texture which defines how the normals of the surface it is being applied to are modified by storing different depths as different shades of grey. Therefore, a bump map is usually an 8-bit greyscale image.

This method eventually evolved to the more commonly used Normal Mapping.

### 2.4.1 Normal Mapping

Unlike Bump maps, Normal maps store the actual normal orientation of each pixel in the texture as RGB values. This means that the renderer does not need to interpret the height map and compute the normals as that information is already there.

Due to the complexity of representing the orientation of normals, Normal maps are rarely hand-painted. Instead, a high-polygon mesh is first modelled so that the normal map can be generated. This map is then applied to the low-polygon version of the same mesh, preserving the detail of the original version.

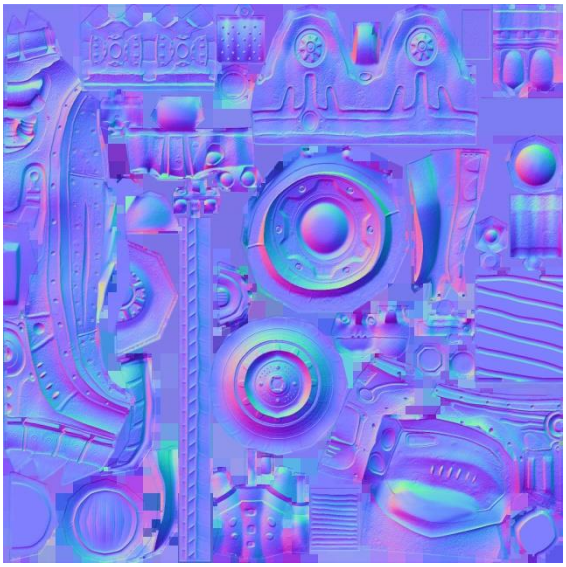

Figure 12 - A normal map taken off a high polygon 3D model. When applied to a low polygon version of the same mesh, much of the original detail will be kept.

The idea of taking information from a detailed object and applying it to a more simple version was first introduced by Krishnamurthy & Levoy (1996) and further refined and expanded by Cohen et al. (1998) and Cignoni et al. (1998).

### 2.4.2 MIP Mapping

When rendering an object on screen which can be looked at from different perspectives, angles and distances it is not ideal to have it using a single fixed resolution texture. Doing this will create aliasing artifacts, which are incorrect representation of the rendered scene. Instead, using several textures, of the same base pattern or image, but with different resolution is a much better approach. It eliminates the mentioned aliasing problems since the texture used will match the size of the rendered object so, if the object is far away from the camera, a lower resolution version will be used while when the object is close, the higher resolution one is used. Furthermore, since smaller textures are being used, the rendering speed can be increased as less texels are being processed.

Mipmaps, introduced by Williams (1983) are used for this effect. In short, they are a collage of the same texture repeated in halved proportions. So, for example, one mipmap could contain 4 textures with the largest potentially being 128x128, followed then by 64x64, 32x32 and 16x16 versions.

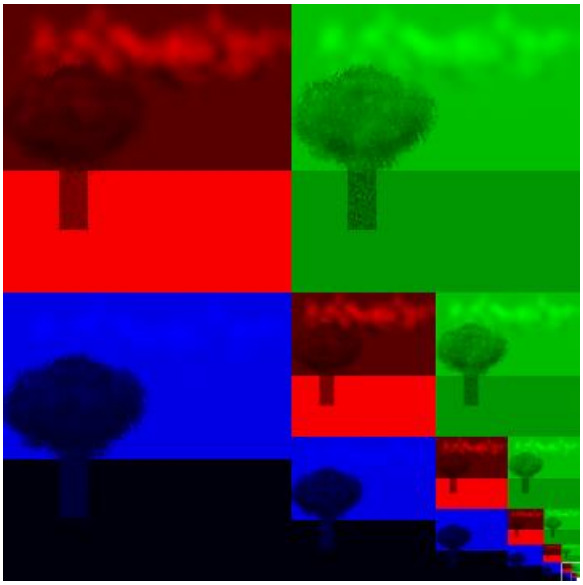

Figure 13 - The MIP map pyramid, with each texture being stored in a different color channel

Of course, if the scene is rendering the object in a 90x90 space, none of the mentioned versions will fit nicely so, to solve the issue, a texture filtering method, such as anisotropic filtering (Schilling, Knittel, & Strasser, 1996), can be used.

### 2.4.3 LEAN Mapping

One particular problem that arises from MIP mapping is the resulting effect of specular highlights. If the object being rendered is far away then a lower resolution texture will be used which means the corresponding normal map has to be scaled down so that the bumps being rendered also scale in an appropriate fashion. A shiny bumpy surface seen from a distance big enough that the bumps cannot be seen should appear to be a dull surface with large, gradual specular highlights. However, normal mapping produces a surface where the original shininess is retained. Bumps become part of the underlying microstructure and fall off to microfacets which result in a sharp, intense highlight.

Cry Engine (Crytek), one of the most popular game engines, renowned for its visual fidelity, even refers the problem in its technical documentation but provides no way to solve it.

Figure 14 - Cry engine's example of the problem with filtering normal maps

The effect is subtle but quite noticeable.

Because specular models such as Phong (1975) assume a flat surface with the normal pointing in just one direction and bump mapping works by altering the surface's normal orientation for each bump, finding a common projection plane to evaluate the normal distribution function over different viewing distances, in order to not lose any information regarding the surface's roughness, has been one of the main issues in MIP mapping normal maps.

LEAN Mapping (Olano & Baker, 2010) is a method, based of the Ward (1992) model, which solves this issue. By using the projection plane of the underlying surface as the common plane for all bump normal projections a new microfacet distribution can be computed. This new distribution will now accurately represent the specular highlight regardless of viewing distance.

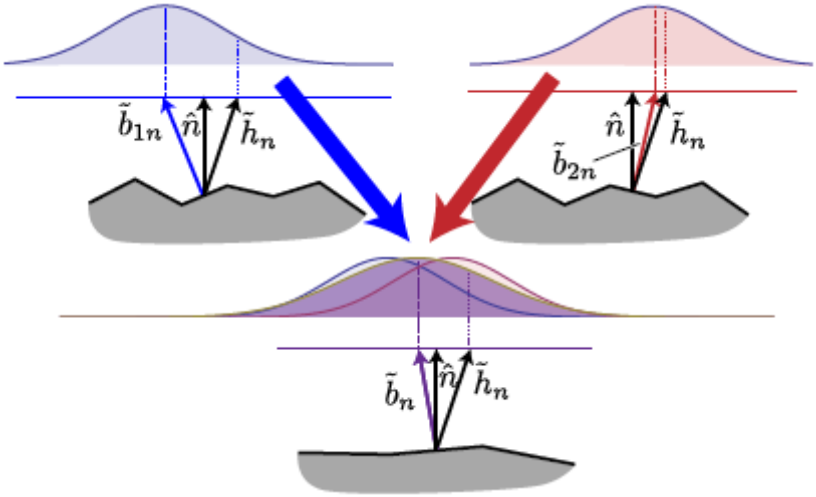


Figure 15 - Projecting bump normals onto a common plane. At different viewing distances the mipmapped normal maps provide different bump normal. By projecting onto the underlying flat surface, a new normal distribution can be computed that incorporates all these variations.

## 2.5 Conclusion

Understanding the techniques, models and concepts presented above is an important step in being able to best utilize previously existing work to produce a new, innovative paper which solves some of the problems or disadvantages found currently.

For the most part, the work of d'Eon et al. (2007) was truly ground-breaking in that it allowed, for the first time, the real-time rendering of photorealistic skin and, consequently, any other translucent material by calculating a diffusion profile via Jensen's dipole (2001) and then approximating it with a sum of Gaussian blurs.

It is based on that work, and the improvements done by Jimenez ( 2009), that this thesis will be built on, along with LEAN Mapping (Olano & Baker, 2010).

# Chapter 3

# Extending LEAN Mapping to subsurface scattering

LEAN mapping is, essentially, a mechanism to preserve the specularity effect regardless of the view distance to the object in question. To accomplish this, it extracts the bump's normal, its center point and the overall highlight shape from the normal map. Figure 16 hows the original normal map, and the resulting two LEAN maps. Note, particularly in the second LEAN map, the prominence of the bumps.

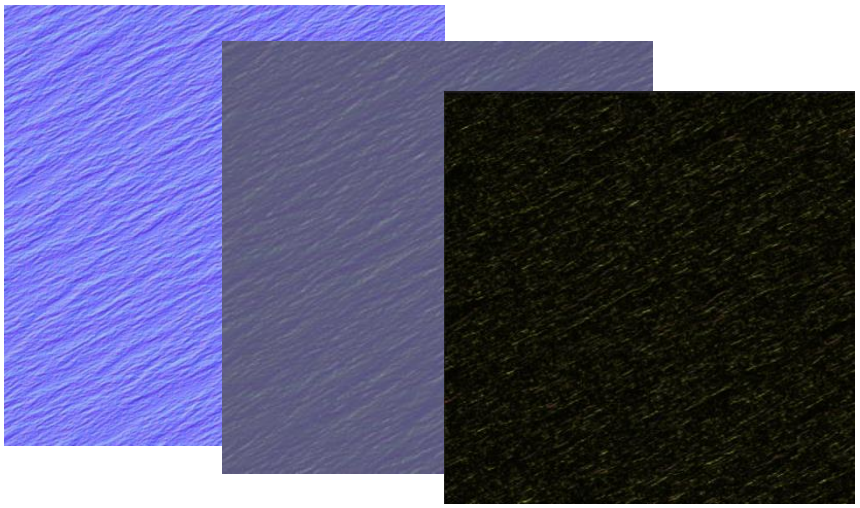

Figure 16 – From left to right: the original Normal Map, the level one LEAN map, containing the surface normal and some of the highlight shape data, the level two LEAN map, containing the bump center position and the rest of the highlight.

Debevec's work (2010), shows that the subsurface effect can be attained when the different RGB surface normals point in different directions while Jimenez's work (2009) shows the effect is also achieved by means of a Gaussian blur applied as post processing. It then stands to reason that applying the same Gaussian blur to the LEAN parameters mentioned should produce a physically accurate result while adding even more detail to the specular highlights. Furthermore, while Jimenez's work deals with how an object's curvature affects its subsurface scattering effect by widening or narrowing the blur radius depending on the depth value, by using LEAN's approach of capturing certain features by projecting them onto a common plane, it is possible to produce a more physically accurate blur.

## 3.1 Objectives

As a research project, this thesis tries to improve on several previous works by identifying their limitations or inaccuracies and correcting them.

In the case of Jimenez's work (2009), the major artifact is the halo that appears around the object. It is hoped that halo can be fully eliminated. Although not as prominent, in thin regions, pixels in the background start to bleed their colour to the object. This is somewhat counteracted by the halo effect, so it is not as noticeable. This too should be eliminated. Furthermore, Jimenez's work is already quite efficient in terms of performance. The aim is to try and maintain that performance.

In order to measure these differences, rather than relying on human observation which is always subjective, it is possible to run image differentiation software to assert whether there are vast differences between the two methods or not. Note that, in terms of the subsurface scattering effect, major differences should only appear near the edges of the object, where the bleeding effect and the halo appear. However, there are major differences in the shape and intensity of the specular highlights. In terms of subsurface scattering though, the bulk of the object should remain relatively similar, in other words, the scattering effect should be the same whether using the method explained in this document or any of the previous works.

Some margin of error is acceptable as it is not expected that every pixel will precisely match the RGB values in every implementation. However, this discrepancy should not be noticeable when run through the image differentiation software.

## 3.2 Subsurface Scattering

For the purposes of this dissertation the two more important works are d'Eon et al. (2007) and Jimenez et al. (2009). This section discusses them in more detail as they are the basis on which this thesis was developed.

### 3.2.1 D'Eon's 2007 work

The weighted sum of Gaussians was the big breakthrough that allowed real-time rendering of realistic human skin. All the equations and values used here are directly from NVidia's GPU Gems 3 book (Geiss, 2007), in particular Chapter 14.
The formula used in that work is:

$$R(r) \approx \sum_{i=1}^{k} w_i G(v_i, r),$$

*R(r)* is the diffusion profile according to radius *r*, for *k* Gaussians with weights *w* and variance *v*. The Gaussian value *G* is given by:

$$G(v, r) := \frac{1}{2\pi v} e^{-r^2/(2v)}.$$

It is important to ensure that the constant does not brighten or darken the image when the 2D blur is applied.
For his work on human skin, D'Eon found that, to accurately approximate the diffusion profile, a sum of six Gaussians was required. These were the values he used:

| | Variance (mm^2) | Blur Weights Red | Green | Blue |
|---|---|---|---|---|
| | 0.0064 | 0.233 | 0.455 | 0.649 |
| | 0.0484 | 0.100 | 0.336 | 0.344 |
| | 0.187 | 0.118 | 0.198 | 0 |
| | 0.567 | 0.113 | 0.007 | 0.007 |
| | 1.99 | 0.358 | 0.004 | 0 |
| | 7.41 | 0.078 | 0 | 0 |

Figure 17 - d'Eon's values for the weights of the sum of Gaussians. These only apply for human skin.

Another important detail to take into account is that, for any particular pixel, the amount of light that has to be calculated has to take into account the light scattered from all nearby pixels. The amount of pixels to be taken into account is determined by how far light travels through the material, in other words, by the diffusion profile. To account for this, d'Eon computed six irradiance textures and then applied the different blurs to each one. Besides these textures, to account for curvature, d'Eon used stretch textures.
The following is the complete version of the shader written by d'Eon.

```
1.    float4 finalSkinShader(float3 position : POSITION,
2.    float2 texCoord : TEXCOORD0,
3.    float3 normal : TEXCOORD1,
4.    // Shadow map coords for the modified translucent shadow map
5.    float4 TSM_coord : TEXCOORD2,
6.    // Blurred irradiance textures
7.    uniform texobj2D irrad1Tex,
8.    . . .
9.    uniform texobj2D irrad6Tex,
10.   // RGB Gaussian weights that define skin profiles
11.   uniform float3 gauss1w,
12.   . . .
13.   uniform float3 gauss6w,
14.   uniform float mix, // Determines pre-/post-scatter texturing
15.   uniform texobj2D TSMTex,
16.   uniform texobj2D rhodTex )
17. {
18.   // The total diffuse light exiting the surface
19.   float3 diffuseLight = 0;
20.   float4 irrad1tap = f4tex2D( irrad1Tex, texCoord );
21.   . . .
22.   float4 irrad6tap = f4tex2D( irrad6Tex, texCoord );
23.   diffuseLight += gauss1w * irrad1tap.xyz;
24.   . . .
```

```
25.   diffuseLight += gauss6w * irrad6tap.xyz;
26.   // Renormalize diffusion profiles to white
27.    float3 normConst = gauss1w + guass2w + . . . + gauss6w;
28.   diffuseLight /= normConst; // Renormalize to white diffuse light
29.    // Compute global scatter from modified TSM
30.    // TSMtap = (distance to light, u, v)
31.    float3 TSMtap = f3tex2D( TSMTex, TSM_coord.xy / TSM_coord.w );
32. // Four average thicknesses through the object (in mm)
33.    float4 thickness_mm = 1.0 * -(1.0 / 0.2) *
34.                          log( float4( irrad2tap.w, irrad3tap.w,
35.                                       irrad4tap.w, irrad5tap.w ));
36. float2 stretchTap = f2tex2D( stretch32Tex, texCoord );
37. float stretchval = 0.5 * ( stretchTap.x + stretchTap.y );
38. float4 a_values = float4( 0.433, 0.753, 1.412, 2.722 );
39. float4 inv_a = -1.0 / ( 2.0 * a_values * a_values );
40. float4 fades = exp( thickness_mm * thickness_mm * inv_a );
41. float textureScale = 1024.0 * 0.1 / stretchval;
42. float blendFactor4 = saturate(textureScale *
43.                                length( v2f.c_texCoord.xy - TSMtap.yz ) /
44.                                ( a_values.y * 6.0 ) );
45. float blendFactor5 = saturate(textureScale *
46.                                length( v2f.c_texCoord.xy - TSMtap.yz ) /
47.                                ( a_values.z * 6.0 ) );
48. float blendFactor6 = saturate(textureScale *
49.                                length( v2f.c_texCoord.xy - TSMtap.yz ) /
50.                                ( a_values.w * 6.0 ) );
51. diffuseLight += gauss4w / normConst * fades.y * blendFactor4 *
52.                 f3tex2D( irrad4Tex, TSMtap.yz ).xyz;
53. diffuseLight += gauss5w / normConst * fades.z * blendFactor5 *
54.                 f3tex2D( irrad5Tex, TSMtap.yz ).xyz;
55. diffuseLight += gauss6w / normConst * fades.w * blendFactor6 *
56.                 f3tex2D( irrad6Tex, TSMtap.yz ).xyz;
57. // Determine skin color from a diffuseColor map
58. diffuseLight *= pow(f3tex2D( diffuseColorTex, texCoord ), 1.0-mix);
59. // Energy conservation (optional) - rho_s and m can be painted
60.    // in a texture
61.    float finalScale = 1 - rho_s*f1tex2D(rhodTex, float2(dot(N, V), m);
62. diffuseLight *= finalScale;
63. float3 specularLight = 0;
64.   // Compute specular for each light
65.    for (<em>each light</em>)
66.     specularLight += lightColor[i] * lightShadow[i] *
67.                 KS_Skin_Specular( N, L[i], V, m, rho_s, beckmannTex );
68.   return float4( diffuseLight + specularLight, 1.0 );
69. }
```

The comments allow for an easier understanding of the procedure. The only thing requiring explanation is, on line 67, the *KS_Skin_Specular* function. This function is described in the GPU Gems 3 book, chapter 14, and its purpose is to calculate the Fresnel reflectance of the the incoming light.

Fresnel reflectance is the amount of light that is reflected as specular for a particular kind of material. Pharr & Humphreys (2004) explain the process in greater detail. The important detail to worry about is that the computations for the Fresnel term are complex so, usually, Schlick's approximation (1993) is used.

### 3.2.2 Jimenez's optimizations

As explained, the basic idea behind d'Eon's work was to blur the irradiance textures and then render the image. In contrast, Jimenez et al. (2009) allow for the image to be rendered first without the subsurface scattering effect and then applies the blurring in a post-processing effect.

Here is his shader, which is much simpler:

```
float4 BlurPS(PassV2P input, uniform float2 step) : SV_TARGET {
// Gaussian weights for the six samples around the current pixel:
//   -3 -2 -1 +1 +2 +3
float w[6] = { 0.006,   0.061,   0.242,  0.242,  0.061, 0.006 };
float o[6] = {  -1.0, -0.6667, -0.3333, 0.3333, 0.6667,   1.0 };

// Fetch color and linear depth for current pixel:
float4 colorM = colorTex.Sample(PointSampler, input.texcoord);
float depthM = depthTex.Sample(PointSampler, input.texcoord);

// Accumulate center sample, multiplying it with its gaussian weight:
float4 colorBlurred = colorM;
colorBlurred.rgb *= 0.382;

// Calculate the step that we will use to fetch the surrounding pixels,
// where "step" is:
//     step = sssStrength * gaussianWidth * pixelSize * dir
// The closer the pixel, the stronger the effect needs to be, hence
// the factor 1.0 / depthM.
float2 finalStep = colorM.a * step / depthM;

// Accumulate the other samples:
[unroll]
for (int i = 0; i < 6; i++) {
  // Fetch color and depth for current sample:
  float2 offset = input.texcoord + o[i] * finalStep;
  float3 color = colorTex.SampleLevel(LinearSampler, offset, 0).rgb;
  float depth = depthTex.SampleLevel(PointSampler, offset, 0);

  // If the difference in depth is huge, we lerp color back to "colorM":
  float s = min(0.0125 * correction * abs(depthM - depth), 1.0);
  color = lerp(color, colorM.rgb, s);

  // Accumulate:
  colorBlurred.rgb += w[i] * color;
}

// The result will be alpha blended with current buffer by using specific
// RGB weights. For more details, I refer you to the GPU Pro chapter :)
return colorBlurred;
}
```

The work developed in this dissertation uses this shader as a starting point.

Note how this shader uses static values, while the blur presented in this paper (see section 4.3.2) computes the weights on the fly via the regular Gauss equation:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

In addition, Jimenez optimized the process by summing the Gaussian blurs on the fly. The final result returned by shader above is then blended together with d'Eon's Gaussian weight values. This way, there is no need to render six different textures, all the blurring can be done iteratively on just one texture. This is the approach used by the shaders developed for this dissertation, as will be explained in section 4.3.

## 3.3 Solution Outline

Considering what was mentioned in the Related Works section and coupling it with the goal of the project, a solution for this problem can be devised. As explained in section 2.4, deferred rendering suits this particular problem very well, which means, inevitably, that the blur will have to be applied to the G-Buffer. This, in turn means that consistency in algebraic spaces for the forward and deferred pass must be maintained which, in turn, means that whatever transformations have to be applied to the forward pass to account for an object's curvature must be either inverted in the deferred pass or also applied to data unique to the deferred pass. In other words, all input that has not been relayed from the first pass to the deferred pass by means of the G-Buffer requires the application of same transformation.

With all this in mind, the solution is relatively easy to understand. The following diagram illustrates the concept in broad strokes:

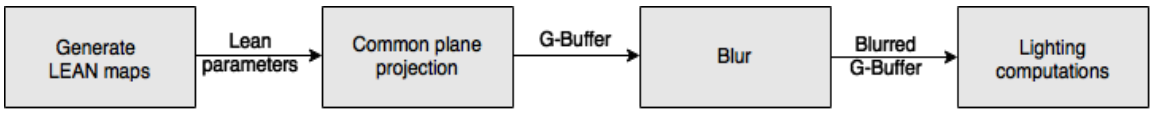


Figure 18 - An overview of the graphics pipeline

## 3.4 Technological Decision

In order to effectively carry out this project, it was necessary to choose which technology to use. This section serves to explain the reasoning behind the selection of the used technology.

### 3.4.1 Graphics API required features

In any project, before researching any potential technologies it is important to know what the requirements are. In this particular case, the chosen graphics API must allow the developer to implement several common techniques in Computer Graphics. Note that, as a consequence, the hardware must support the selected API. The following list provides that information:

#### 3.4.1.1 Programmable Graphics Pipeline

A programmable graphics pipeline is pretty much prevalent in any graphics heavy application. The ability to use shaders to control how the graphics are drawn on the screen is at the core of any 3D application. It is, therefore, a critical feature required for this work.

#### 3.4.1.2 Mip Mapping

LEAN mapping was originally developed in order to solve the issue of filtering MIP mapped Normal Maps. As LEAN mapping is at the core of this project, in fact, there is still a necessity to generate LEAN maps, it stands to reason that its requirements are also this project's.

#### 3.4.1.3 Render To Texture & Multiple Render Targets

Along the same lines of the previous item, both of these features are necessary in order to achieve the expected results. The ability to output data to Multiple Render Targets is vital in order to be able to blur the G-buffer, which, in turn, would not exist if rendering to textures was not, also, a feature.

#### 3.4.1.4 Blending

A crucial point for this project is the ability to blend two different textures. It is this way that the separable sum of Gaussians blur originally devised by d'Eon (2007) can be computed on the fly, as demonstrated by Jimenez (2009).

### 3.4.2 Platforms and Frameworks

There are currently two major technologies for developing graphical applications: OpenGL and DirectX. While both of these fulfil all of the requirements presented in the previous section, of the two, DirectX is the most common API used by the gaming industry in PC environments (StackExchange, 2011), which is where this work will be implemented on. Since one of the goals of this thesis is to develop a method that can easily be incorporated into current existing asset pipelines, it makes sense to choose DirectX over OpenGL.

Several game engines, such as Unreal Engine 4, Cry Engine or Unity3D were considered as they support both DirectX and all of the features mentioned in the requirements. However, they were all ultimately discarded as one can work directly with Direct3D and, in doing so, a more flexible framework can be developed. This way, rather than having to cope with all the rules imposed by said game engines, this framework can be specifically tailored for the intended research, which means faster iteration and prototyping.

It is also worth mentioning that the chosen DirectX version is 11, which was, at the start of this work, the most recent one.

## 3.5 Conclusion

Whilst the choice fell upon DirectX for the mentioned reasons, it is important to state that OpenGL is also a valid option and there is nothing in this work that cannot be ported to OpenGL.

DirectX simple fits best within the overall scope, theme and context of this thesis.

Given that the goal of the project is to come up with a relevant, modern subsurface scattering technique, and considering the above outlined solution, the requirements and technologies chosen, it is now possible, to move from theoretical conjecture to actual implementation.

# Chapter 4

# Implementation

In order to demonstrate how the expected result can be achieved, this section details every aspect of the solution implementation, presenting the underlying graphics pipeline configuration and the purpose of each shader pass. In addition, it provides insight into the minimum requirements to set up a functioning shader sandbox.

## 4.1 Overview

Taking advantage of Direct3D's capability to configure the graphics pipeline for deferred rendering, the following diagram illustrates how the implementation of this project was devised.

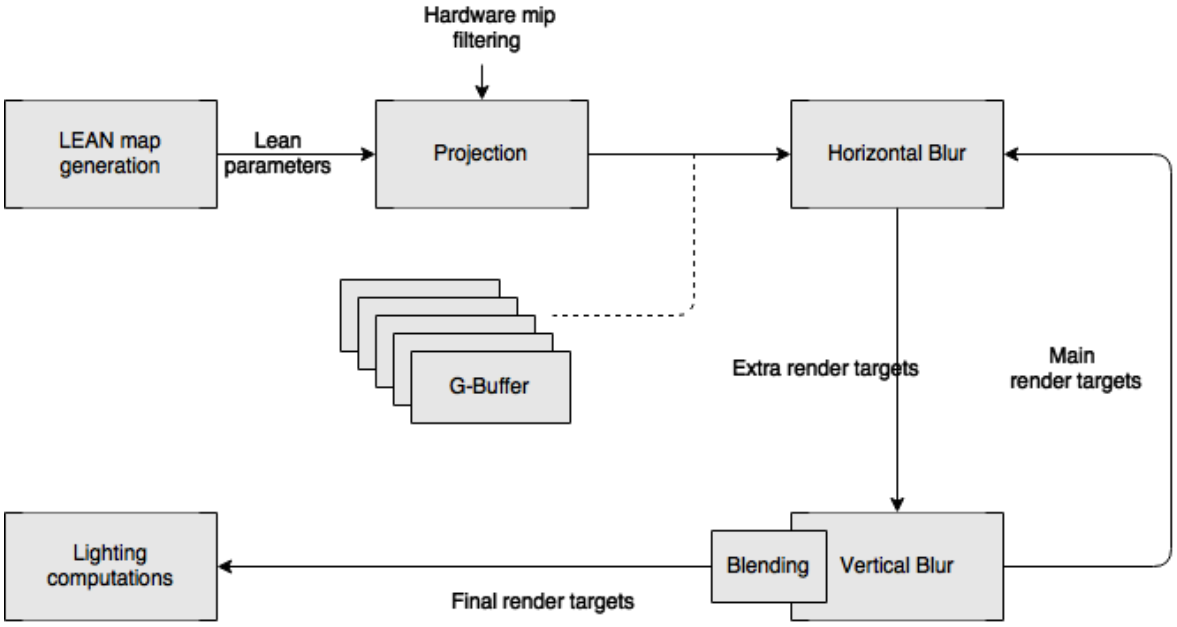


Figure 19 - A detailed look at the several passes of the configured pipeline

The first step is the generation of the LEAN maps. As these maps are based solely on the normal maps, this step could be done offline. However, including it facilitates the use of different models and textures for testing purposes.

The second step is the traditional first pass of the deferred rendering approach where we just gather the object data and output it to the G-Buffer. By sampling the LEAN values we allow the hardware MIP filtering to take place, therefore conserving the advantages of LEAN mapping. These values are then projected onto a different plane, albeit one that remains constant throughout the whole process and stored in the G-Buffer, along with the world position of the pixel in question, its depth information and colour. Since this is a screen-space technique the choice falls upon the view plane.

Next, it is necessary to apply the required blurring effects to simulate the scattering of light. The resulting output will be blended incrementally with each blur's iteration to attain the desired sum of Gaussians which approximates the material's diffusion profile.

Finally, all that remains is the relatively straightforward BRDF computations in typical deferred shading approaches. Note that, because the values were projected onto a plane in step two, it is necessary to take precautions to ensure all the computations are done in the same algebraic space.

## 4.2 Shader sandbox

To enable the development and testing of shaders it is first necessary to build a simple application which provides a reasonably efficient way of swapping out models, textures and shaders. Furthermore, it should implement all the necessary required features discussed in section 3.2.1.

The developed sandbox is a straightforward, simple, isolated application with the following structure:

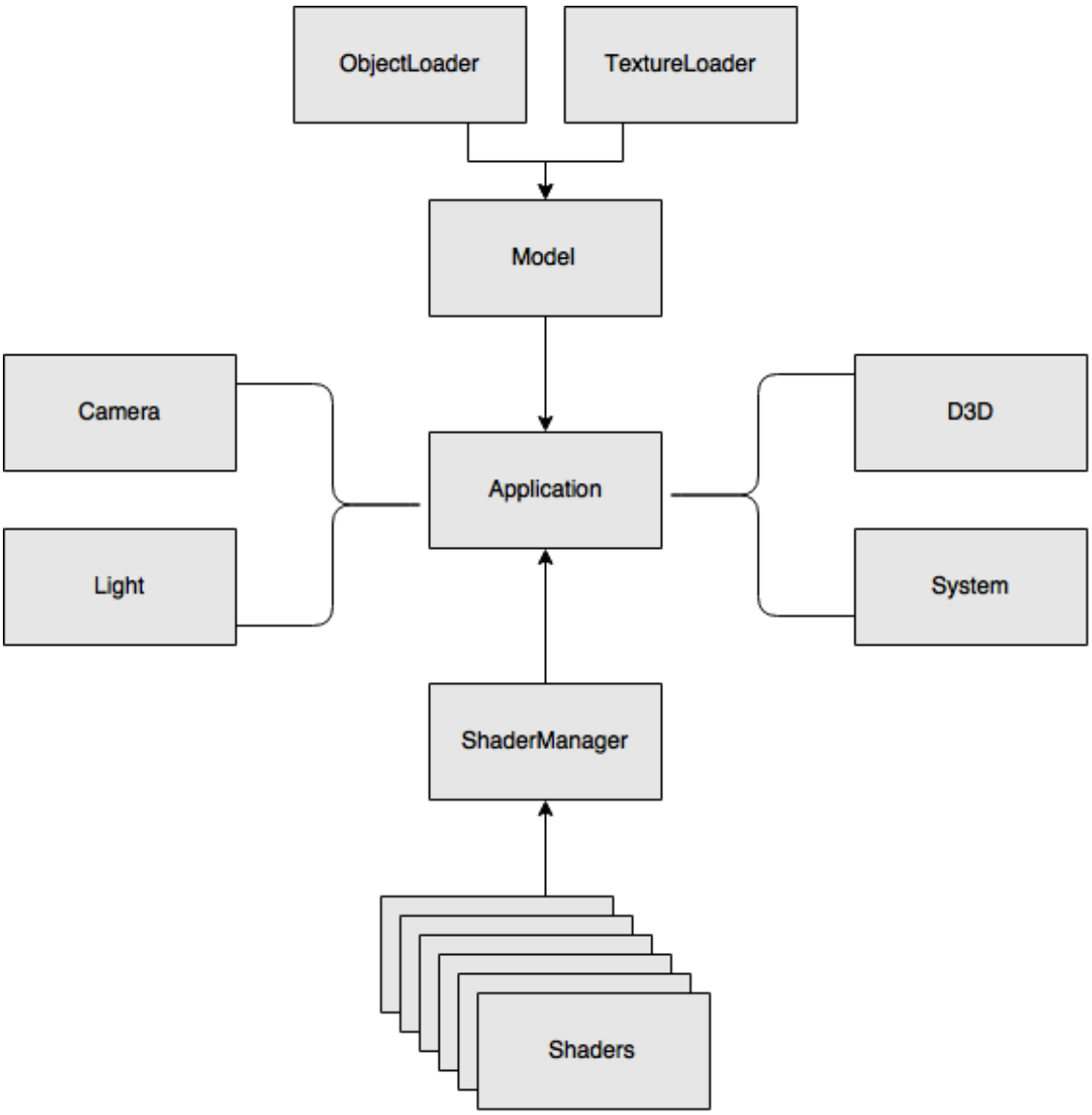


Figure 20 - The sandbox structure

The Camera and Light classes are pretty intuitive in that they hold the necessary information regarding camera and lights.

The D3D class is responsible for setting up and initializing the essential parts of any Direct3D application, such as the Swap Chain buffer, the Device and its Device Context.

The System class deals with the Windows API in order to set up the visible application window. It also figures out what hardware is being used and checks if it supports the required features.

To be able to quickly come up with a new shader that requires a different pipeline configuration there is a Shader Manager class responsible for setting the appropriate buffers for each shader and binding them correctly. This class then exposes a Render function which sets all the necessary resources and targets.

Both the Texture Loader and Object Loader are simple wrappers for the libraries used to load, optimise and manipulate both textures and 3D models. These are only used by the Model class, which is responsible for storing all the model's data, such as vertices, normals, UV coordinates and so on, while, at the same time, preparing the buffers to send them to the GPU.

The Application class is what connects all the different modules as it is responsible for selecting which models are to be rendered, which shaders to use, where with the Camera, Lights and objects are positioned and running the draw cycle.

### 4.2.1 Features

The sandbox is capable of loading Wavefront OBJ 3D models of up to 100,000 vertices as long as they are correctly UV mapped. Tangent and binormals, if not defined, will be automatically calculated. A 3D model can also be assigned four different textures. In this case these textures are almost always a colour texture, a normal map and the two LEAN maps. However, it is not necessary that the four textures fit into this description, depending on the shader being used. The textures are usually in DDS format but other formats such as JPEG and PNG are also supported.

All the requirements mentioned in section 3.2.1 are fully supported. There is an array of textures that can be used to set the shader render targets and resources and, while there is no limit on the size of this array, it is only possible to set eight render targets for any particular shader. This is a Direct3D 11 limitation. While configuring the properties of these textures it is possible to set up automatic mipmap generation. This is not exclusive to this texture array, as any loaded image can have its MIP chain generated online.

All of these features are packed into functions that are called in either the Application's class constructor, or on the Frame method, which is responsible, as the name indicates, for what happens each frame. This way, only minor tweaks in the order of a handful of lines of code, are necessary to change the sandbox's behaviour. The most complicated part is the creation of new shaders. Whilst it is easy to configure their behaviour via the available functions that deal with the above mentioned features, different shaders have different number of resources and targets. This means that, for each shader, it is necessary to edit the Shader Manager class to reflect these changes. In addition, the constructor of the Application class is responsible for the loading of all shaders and models and their file path is hard-coded, which means that, to test new models and shaders, it is necessary to edit said paths.

Below is a fragment of the Direct3D, C++ code necessary to configure the sandbox. In this case, a simple shader for LEAN mapping is being rendered.

```
m_Model->InitializeOBJ(m_D3D->GetDevice(), L"res/data/teapot.obj",
       L"res/data/marbleSwirl.dds", L"res/data/marble_normal.dds");

m_ShaderManager->InitShader("res/shaders/LEAN.hlsl", LEAN_MAP_SHADER);
m_ShaderManager->getDeferredBuffers()->SetRenderTargets(
       m_D3D->GetDeviceContext(), activeRenderTargets);
m_ShaderManager->getDeferredBuffers()->ClearRenderTargets(
       m_D3D->GetDeviceContext(), 0.0f, 0.0f, 0.0f, 1.0f);
m_ShaderManager->getDeferredBuffers()->ClearDepthStencil(
       m_D3D->GetDeviceContext());
/*.... Apply all transformations to the models ....*/

/* Put the model's vertex and index buffers on the graphics pipeline to prepare
them for drawing.*/
m_Model ->Render(m_D3D->GetDeviceContext());

// Render the model using the wanted shader.
m_ShaderManager->RenderShader(m_D3D->GetDeviceContext(),
       m_FullScreenWindow->GetIndexCount(), worldMatrix, viewMatrix,
       projectionMatrix, &m_Model, "SimpleLeanMapping");
```

The *activeRenderTargets* is an array of Booleans that indicated which textures will be bound to the 8 possible render targets. The LEAN_MAP_SHADER is a flag that instructs the ShaderManager to configure the buffers for LEAN mapping. If a new shader were to be developed, then the ShaderManager should be modified to accommodate for the new buffer configurations.

Finally, it is important to say that the sandbox is capable of running at every resolution available on the GPU, including running in full screen with VSync enabled.

### 4.2.2 Used frameworks and libraries

To facilitate the development of this sandbox, two external libraries were used, although both of them are strongly connected with DirectX. These are DirectXMesh and DirectXTexture.

It was necessary to use both of them as, recently, due to the introduction of the Windows Store in Windows 8, some of the core components of Direct3D were updated. This means that several methods and functionalities were moved out of the core DirectX package and instead left out for the developer to implement.

These two libraries are a response to that change. They are both standardised versions of a mesh loader and texture loader with several other features for both optimisation and manipulation purposes.

## 4.3 Pipeline configuration

A model sent to the GPU has to go through several stages until the final lighting computations can be done. In this section, an explanation of what these stages do is given.

Note that during the development process several obstacles required the introduction of some changes to the original outlined solution. As explained in section 3, it was hypothesized that blurring only the LEAN parameters was enough to achieve the desired effect. However, it was found that, without also blurring a diffuse map, the variation in color was not noticeable.

Also, note that the first pass explained in Figure 19, which deals with the LEAN map generation, is not included in this section as it is not in any way different from the one described in the original paper (Olano & Baker, 2010). An example of two input LEAN maps is given in Figure 16. These textures represent both the surface normal and the new microfacet distribution calculated by projecting the different bumps onto a common plane, in LEAN mapping's case, the underlying surface's plane. With these values it is now possible to figure out the highlight shape and its center, regardless of the viewing distance.

### 4.3.1 MIP filtering pass

This is essentially the first pass of any deferred rendering solution, where only the object's data is gathered and passed along to the G-Buffer.

In this case the G-Buffer is composed of the object's MIP filtered normals and LEAN parameters. This is done by sampling the LEAN maps and reconstructing the required data, in this case, the normal, the bump centre and specular highlight shape. Then, all these values are projected onto a common plane. In order to not lose any sort of information by choosing a plane that hides certain pixels behind the projected object, the chosen plane should be parallel to the view plane. Transforming into view space is then as simple as multiplying by the view matrix. These values are then rendered to the G-Buffer.

To perform the lighting operations in the final pass the world position of the pixel is also required, so it must also be projected onto the same plane and rendered to a G-Buffer texture. In addition, since the half vector is needed for the final BRDF computations, and the depth value is also required, we can render these four values onto a single RGBA texture, somewhat optimising the whole process. Note that all the textures used during this dissertation were floating point buffers, in order to be as precise as possible.

Finally, the only remaining data required to render to the G-Buffer is the colour of the pixel. However, doing so without computing the diffuse and ambient lighting does not provide the expected result (see section 6.2), so, to work around this problem, it is necessary to move the lighting computation to the forward pass.

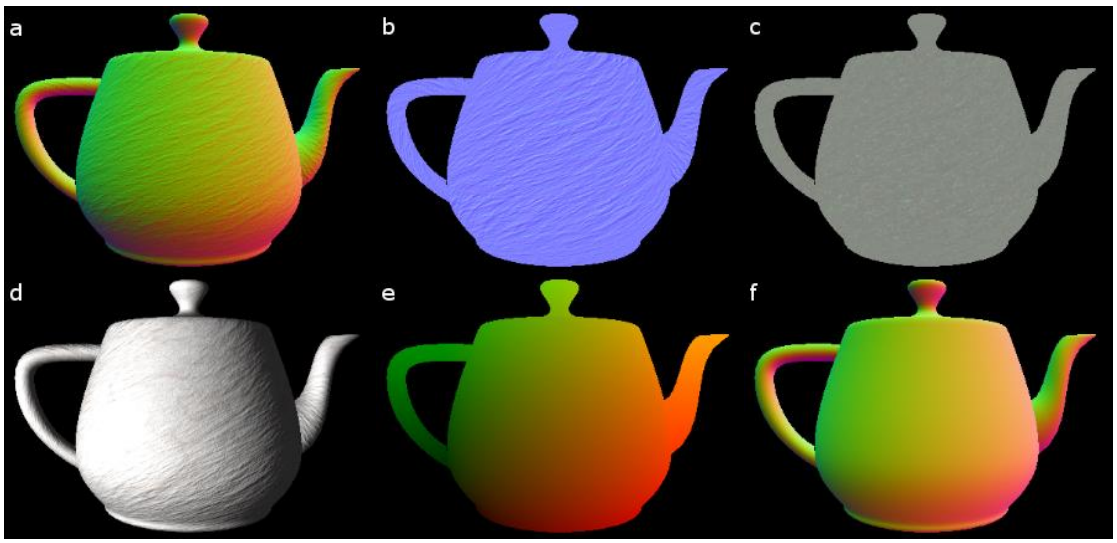


Figure 21- In alphabetical order: the normals, the bump center, the highlight shape, the colour map, the world position and the half-vector with the depth value in the alpha channel. Notice the coulour map already has diffuse and ambient lighting.

Notice that, in Figure 21, the bump center and highlight shape look different than in Figure 16. This is because, in the G-Buffer, they are both rendered to individual textures, whilst in the LEAN maps they are packed, along with the normal, into just two textures. Since the bump center only needs the Red and Green channels as a point on a plane only requires the *x* and *y* coordinates, the Blue channel is set to a neutral value, in this case, 1. This is what gives it the distinctive blue tint.

### 4.3.2 Blurring the G-Buffer

For performance reasons, the blur is separated into two different passes: vertical and horizontal. To accomplish this, a ping pong approach with extra render targets is used. Starting with a main target as input, the horizontal blur is performed and the result sent to an extra target texture. The vertical blur is then applied with the results going to two different targets, the original main target and the final one. This final target is then blended with each blur pass as Jimenez describes.

A fundamental difference between this work and previous ones is the use of a bilateral blur. Rather than narrowing the blur radius according to depth, we use the depth difference to add an additional gaussian weight. Doing so preserves the edges and

prevent the bleeding of pixel colours that do not belong to the object. A modifiable sensitivity parameter is used in order to reduce, or increase, the blurring at the edges.

In his paper, Jimenez introduces a parameter for the subsurface scattering effect's strength. He then uses it, along with the depth's value partial derivatives, to compute a stretch factor for the blur radius. Because, with this blur, the radius is constant, this subsurface scattering strength does not need to be as high. This also means that the blur is fundamentally even throughout the whole object, which would not be ideal given that objects can be curved. However, since the values being blurred are already projected onto a plane, the curvature information has already been preserved and, therefore, the end result is the same as if the blur was being stretched. The advantage here is that that stretching is now based on the actual curvature rather than the depth values.

Of the six textures in the G-Buffer, two of them, the world position and the half-vector, along with the depth value, do not require blurring as they are independent of the other values.

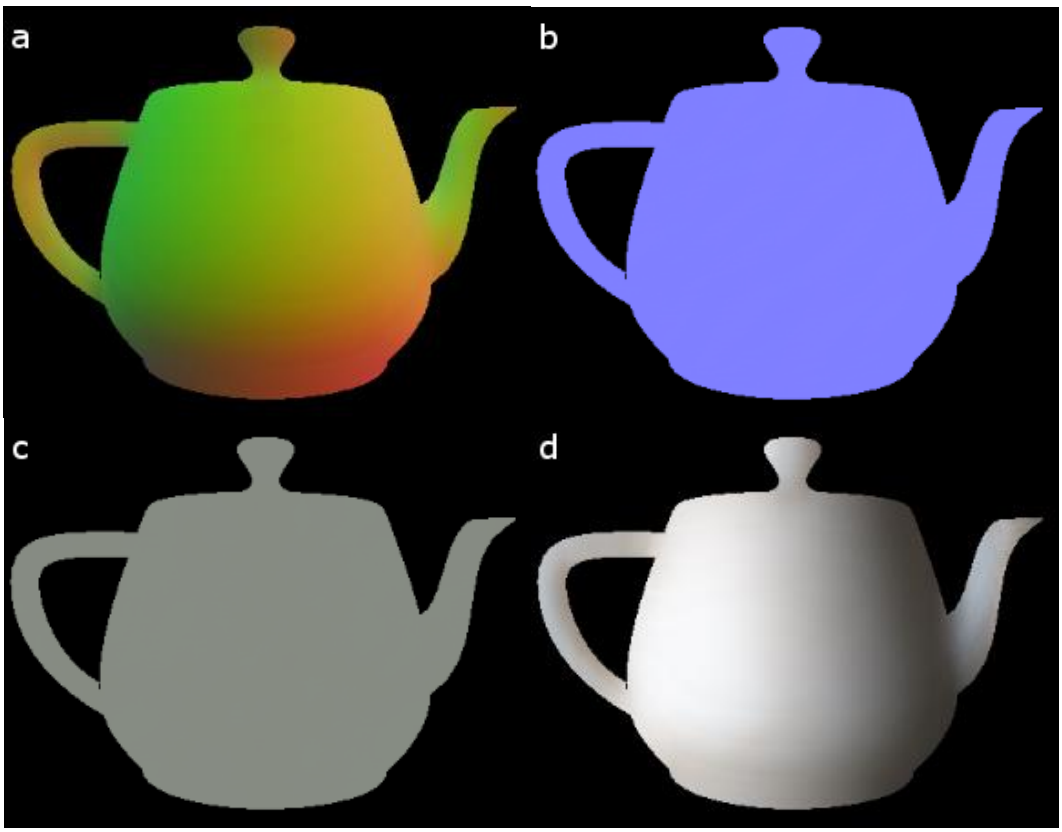


Figure 22 - The final blurred G-Buffer. In alphabetical order, the normals, the bump center, the highlight shape and the colour map.

In Figure 22, all the texture have had the last blur passed applied. Notice, in particular, the colour map, where previously shadowy areas now seem to emit some light.

### 4.3.3 Computing the BRDF

Finally, all that remains is to take the blurred values and compute the BRDF. For the specular component, standard LEAN mapping is used. For diffuse and ambient, standard techniques were used, such as computing light intensity via the dot product between $\vec{\imath}$ and $\vec{v}$.

The only detail that is of relevance in this step is that all values must be in the same algebraic space. Either transform everything value unique to this pass with the same transformation applied in the MIP filtering pass or apply the inverse of that transformation to all the values sampled from the G-Buffer.

Figure 23 - The final result after the BRDF shader pass

## 4.4 Extra tools used

Several other tools were used during the implementation process. Foremost among them is the Visual Studio graphics debugger which enables shader debugging and is, therefore, essential for this sort of work. Furthermore, sometimes it was necessary to clean or slightly alter a 3D model or a texture. For these purposes, Blender and Photoshop CS6 were also used.

## 4.5 Summary

To summarise, the implementation is based around a multi-pass shader, in accordance to typical deferred shading methods. First, the LEAN maps are generated. Then, their values are MIP filtered, projected onto the view plane and stored in the G-

Buffer along with the typical information required by deferred rendering techniques, such as the normal, the world space position and depth values. The blur is then applied to the G-Buffer and, finally, the lighting computations are completed.

Whilst, originally, it was thought that all lighting computations could be done on the final pass that was not case. In response, some of those computations were moved onto the forward pass. While this removes some advantages of the deferred shading approach, it is still a viable option. Section 6.2 has more details on what could be done to fix this issue.

# Chapter 5

# Result analysis

To analyse the results of the implementation of the previous section, it was decided to focus on a material like marble. While many of the mentioned works in this document focus on skin, there are some obstacles that rendering skin poses, such as capturing the various wrinkles and pores, and which are outside the scope of this dissertation. Furthermore, detailed 3D models would be required for a proper result analysis. Marble however, is still a translucent material and does not require as much artwork preparation. In addition, Jimenez (2009), in his paper, already provides values for a marble diffusion profile. Nevertheless, the effects seen in this section should appear in other translucent materials as well, as long as the input resources like the 3D model or the textures and normal maps are appropriate.

Also, it is worth mentioning that many artists already preintegrate a subsurface scattering effect into their normal maps by making the normals softer looking. While this is a perfectly valid option, it is not based on any physical method. These normal maps should not be used with this project or any related work as the result would be a compounded scattering effect that is not based on the actual object's surface translucency. Furthermore, because the normal maps get blurred, the scattering effect is more noticeable the bumpier the surface of the object is.

This chapter will mostly focus on comparing this dissertation with the work Jimenez did, as it is the most similar one as well as the most recent.

## 5.1 Metrics

To perform any analysis we first need to identify the metrics which are to be used. In this particular case, there are really only two metrics that are relevant, described below. For clarity purposes from hereon, Jimenez's work is the Control (C) version against which this work is compared to. The Lean Control Blur (LCB) refers to the new

method but using Jimenez's blur, whilst Lean Bilateral Blur (LBB) refers to the new method with, of course, the bilateral blur implementation.

### 5.1.1 Image Differentiation

Asking a person to try to spot differences between two images in this sort of context is not accurate as the differences might not even be perceptible to the human eye. Also, visual interpretation is subjective to the subject which means there will never be a definitive answer to each image is more realistic or even 'looks better'. Not to mention that, sometimes, what looks better may not be realistic in any sense.

For this particular reason it is best to let a computer find differentiate the images. A simple shader was written for this purpose. Its output is simply the absolute difference between the two input textures. This means that if two textures are similar then a mostly black screen should be the output, while any major difference should be easily visible. As a consequence, any artefact that is either invisible or just hard to spot can be quickly identified.

Below, in Figure 24, we can see three different versions of the same teapot. The top image corresponds to the LBB version, the middle one to the C version, whilst the bottom is, of course, the LCB one. Notice how this last one creates visible artifacts at the edge of the teapot and near the large specular highlight. Even in this version though, we can see how C's halo disappears. The LBB version fixes all the artifacts mentioned while giving off the soft and warm look of skin much like the C version.

Notably, the specular highlights are vastly different, as it is to be expected due to the advantages of LEAN mapping over Phong, which is the model used for C. Aside from the difference in shape and size, it is possible to see that the blurring of the normals, bump centres and highlight shape also gives off a very scattered feel to overall highlight, much like the C picture where the blur is applied after the highlight is computed. One other difference is the amount of external pixels bleeding into the teapot. This is especially noticeable on the C version in the teapot's handle. The bilateral blur prevents this from happening in the LBB image.

Figure 24 - The same teapot rendered with 3 different shaders. From top to bottom: C version, LCB version, LBB version.

For a better perception of just how similar or different these images are, Figure 25 shows the result of the comparison shader between LBB and C. Notice, in particular, the difference in the edges and in the halo.

Between LCB and LBB, we can see that the main difference is really in the edges.

Figure 25 - Top: Difference between C and LBB. Bottom: difference between LBB and LCB.

Figure 26 shows the effect of changing the sssStrength and sensitivity parameters. For the C version, the strength value is always 31.5, in accordance to what is described by Jimenez. For Figure 24, LBB has 31.5 and 10 for strength and sensitivity respectively, while in Figure 26 those values were dropped to 1.

With these values, the same artifacts found in the original LCB also appear on the LBB. Not only that, but the halo is also visible. Without the strength value, bumps become more pronounced, giving a harder look.

Figure 26 - The effect of strength and sensitivity parameters. On top, the strength value is the same but sensitivity is dropped to 1 while, on the bottom picture, the sensitivity is 10, while the strength has been dropped to 1.

The most prominent improvement is the removal of the halo surrounding the C version. The edges in the LBB version are much sharper and precise, while maintaining the scattering effect on the object itself and preventing any colour bleeding.

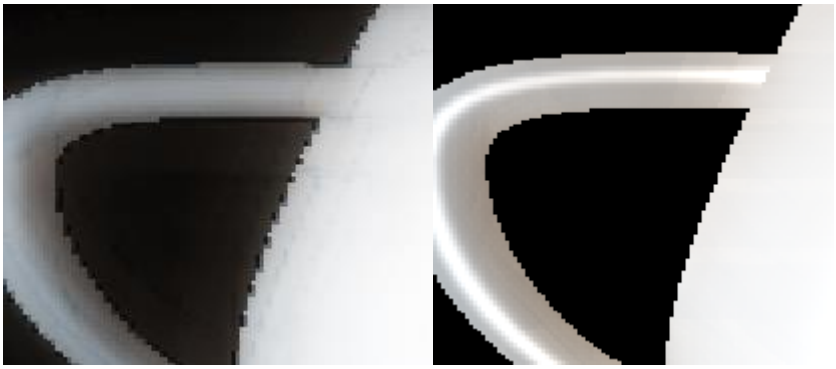

Figure 27 - Left: the C version. Right: the LBB version. Notice the elimination of the halo.

### 5.1.2 Performance

In the field of computer graphics, particularly when dealing with real-time graphics, performance is of the utmost importance. It is very likely that a particular technique's performance will end up be the deciding factor in whether or not it gets implemented in a graphics application. For this reason, performance is an obvious choice for a metric. As a consequence, an analysis on such figures has been carried out. This section describes said analysis. Keep in mind all tests were run on a system with a NVIDIA GTX770 GPU, with 8GB RAM and AMD FX-8350 processor at a frame resolution of 1024x768. Furthermore, all these measurements were made using the Visual Studio Graphics Debugger.

#### 5.1.2.1 Performance per shader pass

In order to get a better grasp on the amount of time it takes to achieve the desired subsurface scattering, it is of value to know exactly which steps take the longest and which are the fastest. In doing so, it is then possible to optimise the solution.

The following table specifies the average times captured when running this project.

Note that the first pass, the generation of LEAN maps, has been left out as, as previously mentioned, those textures can be precomputed and, therefore, the pipeline does not need to be configured with this pass. It was only included in the solution to facilitate testing. Also, the values presented here are already an average of ten different runs of this test.

| Shader Pass | Time |
|---|---|
| Projection | 0.20ms |
| Horizontal Blur #1 | 0.24ms |
| Vertical Blur #1 | 0.63ms |
| Horizontal Blur #2 | 0.36ms |
| Vertical Blur #2 | 0.63ms |
| Horizontal Blur #3 | 0.36ms |
| Vertical Blur #3 | 0.63ms |
| Horizontal Blur #4 | 0.35ms |
| Vertical Blur #4 | 0.63ms |
| Final Lighting Pass | 98.62μs |

Table 1 - Performance figures for bilateral blur

Notice how it takes roughly one millisecond per full blur pass. This means roughly four milliseconds for the four blurs. When taking into account the remaining passes we obtain, on average, a time under five milliseconds per frame. This equates to roughly 300 milliseconds per 60 frames, well under the standard of 60 frames per second.

For comparison, here is the performance data for the control test.

| Shader Pass | Time |
|---|---|
| Lighting calculations | 51.74μs |
| Horizontal Blur #1 | 0.10ms |
| Vertical Blur #1 | 0.16ms |
| Horizontal Blur #2 | 0.11ms |
| Vertical Blur #2 | 0.16ms |
| Horizontal Blur #3 | 0.14ms |
| Vertical Blur #3 | 0.15 |
| Horizontal Blur #4 | 0.14ms |
| Vertical Blur #4 | 0.21ms |

Table 2 - Performance figures for the control screen space shader

As expected, this is much faster, about four times as much. The reason for this discrepancy resides in the fact that blurring the G-Buffer means blurring more textures. In fact, it means blurring four textures against the single one in the C version.

The difference can be significant depending on the number of blurs used since, as can be seen above, the blur times are always pretty constant. Therefore, depending on exactly how accurate one intends the result to be, a different number of blurs could be used, as long as they approximate the original diffusion profile in some way.

## 5.2 Conclusion

Given that by blurring the G-Buffer, the amount of blurred texture quadruples, it was expected that the developed technique would not be as fast as the control version. However, this has shifted the effect from a post processing one to one applicable in deferred shading. This brings with it some of the natural advantages of deferred shading. On the other hand, it forced the computation of some of the lighting calculations on the forward pass, thereby also eliminating some of those advantages. Nevertheless, it is a step in the right direction and a solid foundation for future work.

# Chapter 6

# Conclusion

This dissertation has presented the most important concepts of the subject at hand, explained the goals of this work, the expected results and given a detailed description on how the developed procedure was implemented. While, visually, there are some interesting results, in particular with the specular highlights and the removal of the surrounding halo, it is also noteworthy that, performance wise, this new technique is less efficient than related works. Nevertheless, the shift from applying the subsurface scattering effect in forward rendering to deferred rendering context is an interesting point. Furthermore, the ability to incorporate the subsurface scattering effect into LEAN mapping is probably the most exciting contribution.

## 6.1 Struggles and difficulties

During the time spent working on this project there were many obstacles which presented themselves. The two most significant ones relate to how the initial results did not fit with what was expected.

Originally, due to Debevec's findings on how individual RGB normals pointed in different directions, it was hypothesised that simply blurring the normals would be enough to give skin its characteristic soft and warm appearance. This option was investigated and the results showed otherwise.

The next step was blurring the colour map as well, which should give off the distinctive reddish effect near the edges. However, because light intensity was not being computed at forward render time, the difference in colour between the unblurred and blurred versions was so subtle that no distinctive reddish effect was noticeable to the human eye. To fix this issue there were two possible options: either move the diffuse and ambient calculations to the forward pass and proceed from there, deferring only the specular highlights computation or, instead of computing the sum of Gaussians on the

fly, render each blur iteration to different textures that would only be used in the final BRDF pass. As a result, this pass would have to manually implement the blending effect after computing the ambient and diffuse components.The first option is not really in tune with a deferred rendering approach but it is more similar to the previous works this dissertation is based on, so that was the selected choice.

## 6.2 Future Work

During the course of this project, similar work into subsurface scattering was being carried out at UMBC. This dissertation focused more on developing a suitable screen space technique whilst the other was mostly investigating the same question albeit in texture space. Working in texture space assumes a flat surface as curvature cannot be captured, however it does have advantages over screen space, as it is possible to account for features that could be hidden in screen space. It is not possible to know what is on the other side of the rendered object in screen space. Combining these two projects could potentially produce very interesting results, essentially bypassing some of the limitations and difficulties encountered during this work and providing the advantage of capturing an object's curvature, which is has not been taken into account in the project in question.

As mentioned, moving the lighting computations to the forward pass means that this technique has the same limitations of typical forward rendering techniques. To solve this particular problem, rather than summing up the blurs on the fly, which was one of optimisations introduced by Jimenez, it is possible to store each individual blur and then blend them together after the deferred pass. This maintains the deferred approach at the expense of a bigger G-Buffer. While this was not tested, it is certainly an interesting possibility.

Furthermore, while this project focused more on a comparison with the more recent work done by Jimenez, it would also be of value to compare it to the original work done by D'Eon at NVidia.

# References

Alexander, O., Rogers, M., Lambeth, W., Chiang, J. Y., Ma, W. C., Wang, C. C., & Debevec, P. (2010). The Digital Emily Project: Achieving a photorealistic digital actor. *IEEE Computer Graphics and Applications*, *30*, 20–31. doi:10.1109/MCG.2010.65

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics*. doi:10.1145/965141.563893

Blinn, J. F. (1978). Simulation of wrinkled surfaces. *ACM SIGGRAPH Computer Graphics*. doi:10.1145/965139.507101

Cignoni, P., Montani, C., Rocchini, C., & Scopigno, R. (1998). A general method for preserving attribute values on simplified meshes. *Proceedings Visualization '98 (Cat. No.98CB36276)*. doi:10.1109/VISUAL.1998.745285

Cohen, J., Olano, M., & Manocha, D. (1998). Appearance-preserving simplification. *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH 98*, *32*, 115–122. doi:10.1145/280814.280832

Cook, R. L., & Torrance, K. E. (1981). Computer Graphics Volume 15, Number 3 August 1981, *15*(3), 307–316.

Crytek. (n.d.). CryEngine technical documentation. Retrieved from http://docs.cryengine.com/display/SDKDOC4/Tangent+Space+Normal+Mapping

d'Eon, E., Luebke, D., & Enderton, E. (2007). Efficient rendering of human skin. In *EGSR'07 Proceedings of the 18th Eurographics conference on Rendering Techniques* (pp. 147–157). doi:10.2312/EGWR/EGSR07/147-157

Donner, C., & Jensen, H. W. (2005). Light diffusion in multi-layered translucent materials. *ACM Transactions on Graphics*. doi:10.1145/1073204.1073308

Donner, C., & Jensen, H. W. (2006). A Spectral BSSRDF for Shading Human Skin. *Eurographics*, *2006*, 147. Retrieved from http://portal.acm.org/citation.cfm?doid=1179849.1180033\nhttp://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:A+Spectral+BSSRDF+for+Shading+Human+Skin

Geiss, R. (2007). GPU Gems 3. In *GPU Gems 3* (p. 1008). doi:10.1016/B978-0-12-384988-5.00065-6

References

Jensen, H. W., Marschner, S. R., Levoy, M., & Hanrahan, P. (2001). A practical model for subsurface light transport. *Siggraph*, 511–518. doi:10.1145/383259.383319

Jimenez, J., Sundstedt, V., & Gutierrez, D. (2009). Screen-space perceptual rendering of human skin. *ACM Transactions on Applied Perception*, *6*(4), 1–15. doi:10.1145/1609967.1609970

Krishnamurthy, V., & Levoy, M. (1996). Fitting smooth surfaces to dense polygon meshes. In *Computer Graphics and Interactive Techniques, SIGGRAPH, Proceedings of the 23rd annual conference on* (pp. 313–324). doi:10.1145/237170.237270

Olano, M., & Baker, D. (2010). LEAN mapping. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games I3D 10*, *1*, 181. doi:10.1145/1730804.1730834

Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*. doi:10.1145/360825.360839

Schilling, A., Knittel, G., & Strasser, W. (1996). Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications*, *16*, 32–40. doi:10.1109/38.491183

StackExchange. (2011). DirectX for PC gaming. Retrieved from http://programmers.stackexchange.com/questions/60544/why-do-game-developers-prefer-windows

Ward, G. J. (1992). Measuring and modeling anisotropic reflection. *ACM SIGGRAPH Computer Graphics*. doi:10.1145/142920.134078

Williams, L. (1983). Pyramidal parametrics. *ACM SIGGRAPH Computer Graphics*. doi:10.1145/964967.801126

# Annex A

## 8.1 Developed shaders

For completeness purposes, all the developed shaders are presented in this section.

### 8.1.1 Projection shader

```
//////////////
// GLOBALS //
//////////////
Texture2D colorTexture : register(t0);
Texture2D LEAN_map1 : register(t1);
Texture2D LEAN_map2 : register(t2);
SamplerState SampleType;



cbuffer MatrixBuffer
{
	matrix viewMatrix;
};

cbuffer MatrixBuffer
{
	float4 ambient;
	float4 diffuseColor;
};



struct PixelInputType
{
```

```
        float4 position : SV_POSITION;
        float2 tex : TEXCOORD0;
        float3 normal : NORMAL;
        float3 tangent : TANGENT;
        float3 bitangent : BINORMAL;
        float3 viewDirection : TEXCOORD1;
        float3 worldPos : TEXCOORD2;
        float depth : TEXCOORD3;
};


struct PixelOutputType
{
        float4 depthMap : SV_Target0;
        float4 position : SV_Target1;
        float4 projectedNormals : SV_Target2;
        float4 projectedBump : SV_Target3;
        float4 projectedMoment : SV_Target4;
        float4 diffuseMap : SV_Target6;
};



PixelOutputType LeanProjectionPS(PixelInputType input) : SV_TARGET
{
        float sc = 1.0;
        float far = 100.0f;
        PixelOutputType output;


        float4 color = colorTexture.Sample(SampleType, input.tex);

        //unpack normal
        float4 lean1 = LEAN_map1.Sample(SampleType, input.tex);
        float3 N = float3(2 * lean1.xy - 1, lean1.z);

        // Move normal to tangent space
        N = N.z * input.normal + N.x * input.tangent + N.y *
input.bitangent;
        // Normalize the resulting bump normal.
        N = normalize(N);

        //unpack BumpCenter and Moment
        float4 lean2 = LEAN_map2.Sample(SampleType, input.tex);
        float3 B = float3((2 * lean2.xy - 1)*sc, 1.0f);
        float3 M = float3(lean2.zw, 2 * lean1.w - 1)*sc*sc;

        //compute specular
        float3 lightDir = normalize(float3(-3.0f,0.0f,-10.0f) -
input.worldPos);
```

```
        float3 h = normalize(lightDir + input.viewDirection);
        h = input.normal + h.x * input.tangent + h.y *
input.bitangent;
        h = normalize(h);

        //Intensity of the diffuse light. Saturate to keep within the
0-1 range.
        float intensity = saturate(dot(N, lightDir));
        float distance = length(lightDir);
        distance *= distance;
        float4 ambient = float4(0.15f, 0.15f, 0.15f, 1.0f);
                float4 diffuseColor = float4(1.0f, 1.0f, 1.0f, 1.0f);

                float4 colorMap = ambient;


        if (intensity > 0.0f){
                colorMap += saturate(diffuseColor * intensity /
distance);
        }
        colorMap = saturate(colorMap * color);
        float3 pos = input.worldPos.xyz / far;
        output.position = float4(mul(pos, (float3x3) viewMatrix),
1.0f)* 0.5 + 0.5;
        output.depthMap = float4(h*0.5f + 0.5f, input.depth);
        output.diffuseMap = colorMap;
        output.projectedNormals = float4(mul(N, (float3x3)
viewMatrix)* 0.5 + 0.5, 1.0f);
        output.projectedBump = float4(mul(B, (float3x3) viewMatrix)*
0.5 + 0.5, 1.0f);
        output.projectedMoment = float4(mul(M, (float3x3) viewMatrix)*
0.5 + 0.5, 1.0f);

        return output;
}
```

### 8.1.2 Bilateral blur shader

```
//////////////
// GLOBALS //
//////////////
Texture2D projectedNormals : register(t0);
Texture2D projectedBump : register(t1);
Texture2D projectedMoment : register(t2);
Texture2D depthMap : register(t3);
Texture2D diffuseMap: register(t4);
SamplerState PointSampler : register(s0);
SamplerState LinearSampler : register(s1);
```

```
cbuffer SSSbuffer: register(b0)
{
        float width;
        float sssStrength;
        float correction;
        float maxdd;
        float sensitivity;
        float3 padding2;
        float2 pixelSize;
        float2 dir;
        float4 padding;

};


//////////////
// TYPEDEFS //
//////////////
struct PixelInputType
{
        float4 position : SV_POSITION;
        float2 tex : TEXCOORD0;
};


struct HBlurOutput{
        float4 normalsTarget : SV_TARGET0;
        float4 bumpTarget : SV_TARGET1;
        float4 momentTarget : SV_TARGET2;
        float4 colorTarget : SV_TARGET3;
};

struct VBlurOutput{
        float4 normalsTarget : SV_TARGET0;
        float4 bumpTarget : SV_TARGET1;
        float4 momentTarget : SV_TARGET2;
        float4 colorTarget : SV_TARGET3;
        float4 normalsFinalTarget : SV_TARGET4;
        float4 bumpFinalTarget : SV_TARGET5;
        float4 momentFinalTarget : SV_TARGET6;
        float4 colorFinalTarget : SV_TARGET7;
};


float gaussianCoef(int x){

        float PI = 3.1415926;
        return (1/sqrt(2*PI*width*width)) * exp(-0.5*x*x/
(width*width));
}
```

```
VBlurOutput VerticalLeanBilateralBlur(PixelInputType input) {

        VBlurOutput output;
        float4 normals = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 bumps = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 moment = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float4 color = float4(0.0f, 0.0f, 0.0f, 0.0f);
        float w, wZ, wC;
        float Zp = depthMap.Sample(PointSampler, input.tex).a;

        float2 finalWidth = sssStrength * width * pixelSize * dir;
        float2 offset = input.tex - finalWidth;

        int kernelSize = 7;
        for (int i = 0; i < kernelSize; ++i) {
                w = gaussianCoef((finalWidth.y / 3) * (i - 3));
                float zTmp = depthMap.Sample(PointSampler, offset).a;
                wZ = gaussianCoef(abs(Zp - zTmp)*sensitivity);
                normals += w * wZ *
projectedNormals.Sample(PointSampler, offset);
                bumps += w * wZ * projectedBump.Sample(PointSampler,
offset);
                moment += w * wZ * projectedMoment.Sample(PointSampler,
offset);
                color += w * wZ * diffuseMap.Sample(PointSampler,
offset);
                offset += finalWidth / 3;
        }
        output.normalsTarget = normals / normals.w;
        output.normalsFinalTarget = normals / normals.w;//will be
blended
        output.bumpTarget = bumps / bumps.w;
        output.bumpFinalTarget = bumps / bumps.w; //will be blended
        output.momentTarget = moment / moment.w;
        output.momentFinalTarget = moment / moment.w; //will be
blended
        output.colorTarget = color / color.w;
        output.colorFinalTarget = color / color.w; //will be blended

        return output;
}
```

Notice that this code snippet only shows the vertical pass. The horizontal is basically identical except it does not send the output to the final render targets and that the finalWidth is along the *xx* axis, not the *yy*.

### 8.1.3 BRDF Shader

```
//////////////
// GLOBALS //
//////////////
Texture2D Normals: register(t0);
Texture2D BumpCenters: register(t1);
Texture2D Moment: register(t2);
Texture2D diffuseMap : register(t3);
Texture2D positions : register(t4);
Texture2D depthMap : register(t5);
SamplerState PointSampler;


cbuffer LightBuffer : register(b0)
{
        float4 ambientColor;
        float4 diffuseColor;
        float3 lightPos;
        float specularPower;
        float4 specularColor;
};

cbuffer MatrixBuffer : register(b1)
{
        matrix viewMatrix;
        matrix invertedViewMatrix;
}

cbuffer CameraBuffer : register(b2)
{
        float3 cameraPos;
        float padding;
}


///////////////
// TYPEDEFS //
///////////////
struct PixelInputType
{
        float4 position : SV_POSITION;
        float2 tex : TEXCOORD0;
};




float4 BRDF(PixelInputType input) : SV_TARGET
{
        float far = 100.0f;
        float4 output;

        float4 worldPosS = positions.Sample(PointSampler, input.tex);
        float3 worldPos = 2 * worldPosS.xyz - 1;
```

```
        worldPos *= far;

        float3 halfVector = 2 * depthMap.Sample(PointSampler, input.tex).xyz
- 1;

        float3 N = Normals.Sample(PointSampler, input.tex).xyz;
        N = 2 * N - 1;

        float3 bump = BumpCenters.Sample(PointSampler, input.tex).xyz;
        bump = 2 * bump - 1;

        float3 moment = Moment.Sample(PointSampler, input.tex).xyz;
        moment = 2 * moment - 1;

        //convert Moment to Covariance
        float3 Covariance = moment - float3(bump.xy*bump.xy, bump.x*bump.y);
        if (Covariance.x < 0) Covariance.x = 0;
        if (Covariance.y < 0) Covariance.y = 0;
        if (Covariance.x*Covariance.y <= Covariance.z*Covariance.z)
Covariance.xyz = float3(0, 0, 0);
        Covariance.xy += 1 / (specularPower + 2);

        float DetC = (Covariance.x*Covariance.y - Covariance.z*Covariance.z)
* 1;

        //specular
        float2 halfVec_projection = halfVector.xy / halfVector.z - bump.xy;
        float e = halfVec_projection.x * halfVec_projection.x * Covariance.y
                + halfVec_projection.y * halfVec_projection.y * Covariance.x
                - 2 * halfVec_projection.x * halfVec_projection.y *
Covariance.z;
        float specular = (DetC <= 0) ? 0 : exp(-0.5 * e / DetC) /
sqrt(DetC);



        float3 lightDir = normalize(lightPos - worldPos);
        float3 viewDir = normalize(cameraPos - worldPos);


        output = diffuseMap.Sample(PointSampler, input.tex);
        //fresnel
        float3 h = normalize(lightDir + viewDir); // Half-vector.
        float fresnel = 1.0 - dot(viewDir, h); // Caculate fresnel.
        fresnel = pow(fresnel, 5.0);
        fresnel += 0.028 * (1.0 - fresnel);

        //add specular component
        output += specular*specularColor*fresnel;
        return saturate(output);

}
```

# Annex B

To provide a more technical explanation of the works presented in the document, the following paragraphs include a detailed overview of the underlying maths as well as result values.

## 9.1 Microfacet Distribuitions 5.1

When Phong (1975) first introduced his shading model he computed his specular highlights with the following equation:

$$k_{\text{spec}} = \|R\|\|V\| \cos^n \beta = (\hat{R} \cdot \hat{V})^n$$

Where R is vector oriented along the direction of the mirror-like reflected light and V is the vector oriented towards the viewer.

Blinn (1977) further refined the equation to:

$$k_{\text{spec}} = \|N\|\|H\| \cos^n \beta = (\hat{N} \cdot \hat{H})^n$$

Where N is the surface Normal and H is the the Half angle direction. The *n* value is controlled by the user and simulated the roughness of the surface.
These equations were eventually reworked and the vector L – the direction of incoming light – was incorporated with the result approximating a Gaussian Probabilistic Distribution. Since then that distribution has become commonplace when dealing with specular highlights.

Whilst the results are pleasing they are not physically based which is why the Beckmann distribution was introduced by Ward (1992):

$$k_{\text{spec}} = \frac{\exp\left(-\tan^2(\alpha)/m^2\right)}{\pi m^2 \cos^4(\alpha)}, \ \alpha = \arccos(N \cdot H)$$

Where *m* represents the roughness of the material.

## 9.2 Further reading on LEAN Mapping

Along with properly rendering specular highlights at different viewing distances, LEAN mapping also correctly render anisotropic highlights.
When a surface has ridges or grooves flowing along one direction the specular highlights should be perpendicular to that. The model LEAN mapping was based on (Ward 1992) accounted for that with the following anisotropic distribution:

$$k_{\text{spec}} = \frac{1}{\sqrt{(N \cdot L)(N \cdot R)}} \frac{N \cdot L}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{H \cdot X}{\alpha_x}\right)^2 + \left(\frac{H \cdot Y}{\alpha_y}\right)^2}{1 + (H \cdot N)}\right]$$

The $\alpha x$ and $\alpha y$ values are used to represent anisotropy. If they are equal then the highlight will be isotropic. X and Y are unit vectors, in the normal projection plane, which identify the anisotropic directions.

Furthermore, LEAN mapping can also combine several bump map layers. One example would be the ocean where, within a large enough area, there could be waves flowing in different directions. Each one would have its own bump map, reducing the amount of detail and complexity of the bump maps.

As the focus of this dissertation is human skin there will be no need for several layers of bump maps which is why this section does not go into further detail on the techniques used by Olano 6 & Baker (2010).