Universität des Saarlandes

Rendering and Streaming of Bidirectional Texture Functions

Masterarbeit im Fach Informatik
Master's Thesis in Visual Computing
von / by

Oleksandr Sotnychenko

angefertigt unter der Leitung von / supervised by

Prof. Dr. Philipp Slusallek

betreut von / advised by

M. Sc. Jan Sutter

begutachtet von / reviewers

Prof. Dr. Philipp Slusallek

Prof. Dr. Karol Myszkowski

Saarbrücken, January 2015

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, January 2015 Oleksandr Sotnychenko

Abstract

Bidirectional Texture Functions (BTF) are 6-dimensional functions that depend on the spatial position on a surface, light and camera directions. Due to changes either of light or camera directions the appearance of real-world materials can severely change. BTF can represent such materials by capturing important material properties for a wide range of illumination changes.

In this work we present a technique to render BTFs at real-time within a web-browser using WebGL. The ever-growing number of mobile devices that use web-browsers imply constraints on the hardware capabilities. Thus, rendering has to be efficient to be possible on such devices and the fact that all data has to be transferred to the client make compression inevitable. We employ a principal component analysis to compress the data, which allows for rendering of BTFs with interactive frame rates. To provide immediate feedback to the user we additionally use a streaming approach that allows for a progressive enhancement of the rendering quality while transferring the remaining data to the client.

Acknowledgements

I am deeply grateful to my adviser Jan Sutter for giving me motivation, for sharing his experience with me and for his continuous support in writing my thesis. I am thankful to my supervisor Prof. Dr. Philipp Slusallek for inspiring me to write the thesis in the field of Computer Graphics and providing me with valuable pieces of advice. Special thanks go to Kristian Sons and Felix Klein for giving me a continues support and for providing me with directions.

I would like to express my sincere gratitude to Saarland University for providing excellent studying possibilities. I am thankful to the German Research Center for Artificial Intelligence for very pleasant working atmosphere.

Last but not least, I am especially thankful for my parents and friends, who supported me during my studies.

Contents

List of Figures

Chapter 1

Introduction

One of the main goals in computer graphics is realistic rendering. Even though computer graphics is constantly improving, we are still quite away from reality because material representation in a traditional way lack important realistic properties. A 2-D texture in conjunction with a shading model is a conventional way to represent material appearances in rendering. Real-world material surfaces on the other hand consist of surface meso-structures, i.e. intermediate in between micro and macro structures. Meso-structures are responsible for fine-scale shadows, self-occlusions, inter-reflection, subsurface scattering and specularities. Also, reflectance and the look of real-world materials can drastically change when camera and light directions vary.

One of the possible solution to represent such material attributes is to use sophisticated light functions, for instance a Bidirectional Texture Function (BTF). A BTF is a 6-dimensional function that depends on camera and light directions as well as on spatial texture coordinates. The BTF conceptually extends traditional 2-D textures by the dependence on light and camera directions. This function is usually acquired as a data-set of thousands of images that cover discrete light and camera directions. Due to the enormous size of that data direct rendering, even on modern hardware, without any compression is impractical. Fortunately, many techniques exist to deal with the huge size of BTFs, i.e. compression methods.

In this thesis, we use *WebGL* for real-time rendering of BTF in the Web. Even though 3-D graphics for the Web is becoming more and more popular, BTFs are rarely used for realistic rendering, due to their huge size and the overall computational effort needed for rendering, including decompression of the data and interpolation of camera and light directions. Such demanding computational effort is time consuming, especially on mobile-devices.

Another problem which arises when rendering BTFs in a Browser is the transmission of the data. Before rendering on the client side, the transmission of the data has to be finished. Even the transfer of a compressed BTF data can take a considerable amount of time. The compressed BTF can be tens of megabytes in size. Because users are eager to see a result, especially on the Web, we use a streaming solution to deliver the data as quickly as possible, while steadily increasing the quality of the resulting image. *Web-Sockets* are the state of the art technique for streaming and are supported by most of today's browsers. With *Web-Sockets* a full-duplex communication is available for the server and the client, which is faster than traditional HTTP-based methods.

In this thesis we use PCA for compression, which allows for a good compression ratio with low decompression error. It is also well suited for real-time rendering and streaming.

1.1 Related Work

Rendering of 3D content in the browser is possible using WebGL [9], which provides vertex and fragment shaders for realistic rendering results. In contrast to WebGL, declarative 3D approaches, for instance XML3D [37], enable web developer to embed 3D scenes directly inside the HTML DOM. This enables a developer to apply existing knowledge in the context of 3D graphics. The prototype implementation in this thesis uses XML3D seamlessly integrate 3D content into the web. Due to the lack of native browser support, the XML3D.js polyfill is used. It allows for a direct definition of custom GLSL vertex and fragment shaders which are necessary for the rendering of BTFs.

Previous works that used BTFs for realistic rendering were primarily intended for offline applications. However, their approach is not suitable in the context of the Web, due to the data transfer between the server and a client. Schwartz *et. al.* [35] presented a work in which compressed BTF data is streamed from the server to a client. The streaming over the Internet is done by means of HTTP streaming, i.e. the web-application requests the data in small chunks. With each new chunk of the BTF data the rendering quality of the 3D object is progressively enhanced. We on the other hand, use *Web-Sockets* to improve application performance compared to HTPP streaming, by reducing the network latency and streaming the data in a binary form.

Existing BTF compression methods are reviewed by Haindl *et. al.* [16, 14]. A trade-off has to be made between the rendering quality and compression rate depending on the application. Our compression method is related to the PCA RF (Principal Component Analysis Reflectance Field) method, which was introduced by Sattler *et.*

al. [27]. This method allows for real-time performance with realistic rendering quality. Sattler performs PCA on each of the n camera directions separately. We on the other hand, perform PCA on k neighbour camera directions at once. In Haindl's review [16] PCA based methods achieve better or at least equal reconstruction results than most other methods. Compression rates of PCA methods are not as high as those of other methods. However, those methods, which produce better compression rates, have worse visual quality than PCA based methods.

Chapter 2

A Brief Review of Scattering Functions

In this chapter we will review a hierarchy of scattering functions. Scattering functions describe how incoming and outgoing directions of the light are related for a surface at which light-material interaction occurs [11]. Such functions are possible to measure for a given object. After obtaining the data, scattering functions can provide all the necessary information to render the material appearance. Because different materials exhibit different light interaction phenomena, a variety of scattering functions exist. These functions differ in their computational complexity and in their ability to capture specific appearance properties. In order to choose a suitable scattering function for capturing certain scattering effects for a particular type of material, it is important to know which functions exist.

2.1 Light-Material Interaction

Generally speaking, when light hits a material's surface, a sophisticated light-matter process happens. Such process depends on physical properties of the material as well as on physical properties of light[42]. For instance, an opaque surface such as wool will reflect light differently than a smooth surface with high specularities such as metal.

When light makes contact with a material, three types of interactions occur: light *reflection*, light *absorption* and light *transmittance*. Light *reflection* is the change in direction of light at an interface between two different media so that light returns into the medium from which it originates. Light *absorption* is the process when light is being taken up by a material and transformed into internal energy, for instance thermal

energy. When a material is transparent, light *transmittance* occurs. It means, that the light travels through the material and exits on the opposite side of the object. Figure 2.1 demonstrates these 3 types of interactions.

Because light is a form of energy, conservation of energy says that [42]

incident light at a surface = light reflected + light absorbed + light transmitted

2.2 General Scattering Function

To define the general scattering function(GSF) imagine the light-wave hitting the surface at time t_i and position x_i and with wavelength λ_i[27]. With a given local coordinate system at a surface point, the incoming direction of light can be defined as (θ_i, ϕ_i). Light travels inside the material and exits the surface at position x_o and time t_o, with a possibly changed wavelength λ_o in the outgoing direction (θ_o, ϕ_o). Figure 2.1 illustrates the process.

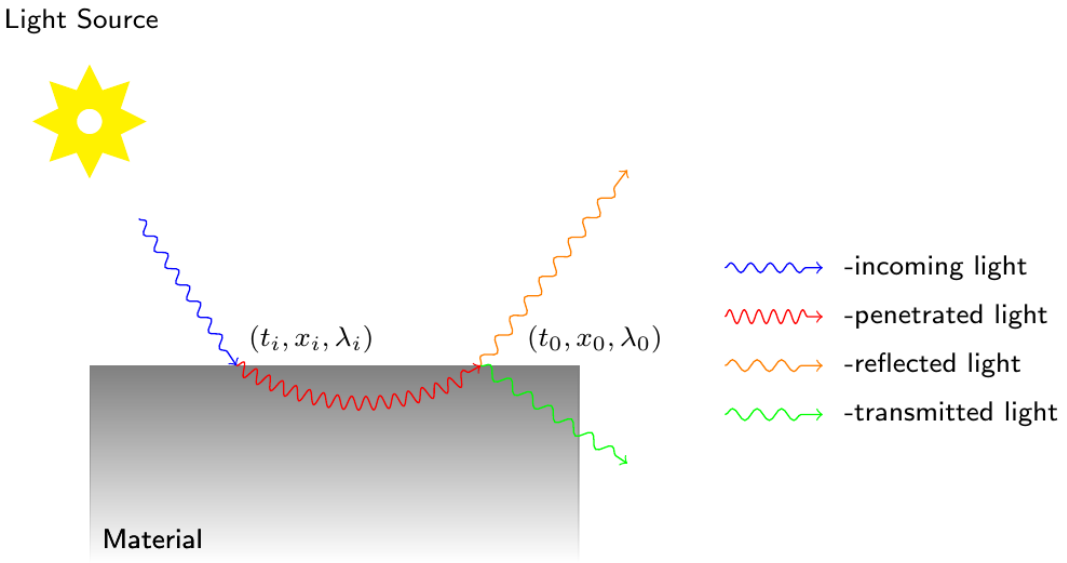

FIGURE 2.1: **Example of Light-Material interaction.**

According to the description we get a GSF

$$GSF(t_i, t_o, x_i, x_{0,}\theta_i, \phi_i, \theta_o, \phi_o, \lambda_i, \lambda_o)$$

in which spatial positions $x_{i,o}$ are 2-D variables. This function describes light interaction for each surface point for any incoming light and outgoing direction at certain time. This function comprises 12 parameters Also, note that we neglected light transmittance, which would even further complicate the function.

2.3 Bidirectional Scattering-Surface Reflectance Distribution Function

Since the measurement, modeling and rendering of a 12-D GSF function is currently not practical, additional assumptions have to be made to simplify the function [27].

- light interaction takes zero time ($t_i = t_o$)
- wavelength is separated into the three color bands red, green and blue ($\lambda_{r,g,b}$)
- interaction does not change wavelength ($\lambda_i = \lambda_0$)

After mentioned assumptions we get a 8-D bidirectional scattering-surface reflectance distribution function (BSSRDF)

$$BSSRDF(x_i, x_o\theta_i, \phi_i\theta_o, \phi_o)$$

BSSRDF describes various light interactions for heterogeneous both translucent and opaque materials. That is why BSSRDF can be used for rendering materials such as skin, marble, milk and other objects which do not look realistic without subsurface scattering. Subsurface scattering is a process when light penetrates an object at an incident point, travels inside the object and exists at a different point of the object.

2.4 Bidirectional Texture Function

If we simplify further and assume that

- light entering a material exits at the same point $x_i = x_o$, while internal subsurface scattering is still present

we will get a 6-D bidirectional texture function (BTF).

Subsurface scattering, self-occlusion, self-shadowing are still present, now it just comes pre-integrated, i.e. it can be defined through BSSRDF [27]:

$$BTF(x, \theta_i, \phi_i, \theta_o, \phi_o) = \int_S BSSRDF(x_i, x, \theta_i, \phi_i, \theta_o, \phi_o)dx_i$$

The assumption that $x_i = x_o$ simplifies measuring, modeling and rendering of the scattering function. As we can see the BTF integrates subsurface scattering from neighbouring surface locations.

2.5 Bidirectional Subsurface Scattering Distribution Function

Another possible reduction of the 8-D BSSRDF is to assume that we deal with a homogeneous surface [11], i.e.

- subsurface scattering depends only on relative surface positions of incoming and outgoing light $(x_i - x_o)$

Simply saying it means that scattering do not vary over a surface. With such assumption we get a 6D function that known as a bidirectional subsurface scattering distribution function (BSSDF).

$$BSSDF(x_i - x_o, \theta_i, \phi_i, \theta_o, \phi_o)$$

BSSDF represents homogeneous materials for which subsurface scattering is a significant feature of their overall appearance. For instance, BSSDF accounts for objects such as water, milk, human skin, and marble.

2.6 Bidirectional Reflectance Distribution Function

If we assume for a BSSDF that

- there is no spatial variation
- no self-shadowing
- no self-occlusion
- no inter-reflections
- no subsurface scattering
- energy conservation
- reciprocity $BRDF(\theta_i, \phi_i, \theta_o, \phi_o)$=$BRDF(\theta_o, \phi_o, \theta_i, \phi_i)$.

we get a 4-D bidirectional reflectance distribution function (BRDF)

$$BRDF(\theta_i, \phi_i, \theta_o, \phi_o).$$

Nicodemus et al. [30] was the one who proposed the BRDF. Two principal properties of the BRDF were introduced, i.e. *energy conservation* and *reciprocity*. *Energy conservation* law states that the total amount of outgoing light from a surface cannot exceed the original amount of light that arrives at the surface [42]. *Reciprocity* says that if we swap incoming and outgoing directions the BRDF stays the same. If either of these conditions are not satisfied then such BRDF is called *apparent* BRDF (ABRDF) [38].

It is quite difficult to create a mathematical model for a BRDF that satisfies reciprocity, energy conservation and at the same time produces realistic images. However, most BRDF models do not satisfy these conditions and still get plausible rendering results. For instance, the Phong model [32] is the most used shading model in computer graphics. The traditional Phong model satisfies neither energy conservation nor reciprocity, but can still render many materials realistically plausible. Usually, such materials are of opaque and flat nature, for example plastic materials. Many existing BRDFs, such as Phong, are imperial models that are often based on simple formulas derived from observations.

In contrast to analytic BRDF models, BRDFs can be also represented and modeled as a set of measurements [25]. Measurements result in a set of images. Each acquired image represents a set of BRDFs. Each pixel of the image is treated as a separate BRDF measurement. The concept of modeling and measurement of the BRDF is related to the BTF, because BTFs can be treated as set of ABRDFs. (See Chapter 3.2.2.)

2.7 Spatially Varying Bidirectional Reflectance Distribution Function

If spatial dependence for BRDF takes place, we get a 6-D spatially varying BRDF (SVBRDF)

$$SVBRDF(x, \theta_i, \phi_i, \theta_o, \phi_o).$$

Assumptions are the same as for the BRDF, except now spatial dependence is present.

A SVBRDF is closely related to a BTF. The difference is in the scattering process. Changes in scattering at local position x for the BTF are influenced from neighbouring 3D surface geometry, as a result the self-shadowing, masking and inter-reflections are captured by the BTF. On the other side, the spatial dependence of a SVBRDF describes variations in the optical properties of a surface [14].

A SVBRDF represents structures at micro-scale level, which corresponds to near flat opaque materials. On the other side, a BTF captures structure both at macro and micro scales. That means that the BTF takes into account influences from local neighbourhood structures. Even though measurement, compression and rendering are more efficient for the SVBRDF, BTFs can produce better visual results. [14]

2.8 Surface Light Field and Surface Reflectance Field

Consider BTFs and assume a fixed light direction $\theta_i = const$, $\phi_i = const$, we will get a 4-D Surface Light Field (SLF) model

$$SLF(x, \theta_o, \phi_o).$$

SLF model is a simple a textural model and is a subset of a BTF. SLF is used when illumination direction is not varying in the scene, but the view direction varies. This model is favoured for its greater computational efficiency in such cases.

Similarly, if we fix the camera, we end up with a Surface Reflectance Field (SRF). SRF is another very popular variant of image-based rendering. In this case many images are taken under varying light directions with a fixed camera direction [27]. For instance, Debevec et al. [10] recorded the appearance of human faces while a light source was rotating around the face. Rendering the human face realistically is always a struggle in computer graphics. Due to complex reflectance characteristics of the human face, common texture mapping usually fails under varying illumination. Debevec et al. acquired approximately two thousand images under different light positions and could render the face under arbitrary lighting for the original camera direction.

2.9 Summary of Scattering Functions

In practice, the advantage of a simpler scattering function is the computational efficiency, while the disadvantage is the reduction of visual quality. The constant improvement of graphics hardware, however, encourages the use of more sophisticated scattering functions, which provide even more realistic material renderings.

However, a complex material representation requires sophisticated data measurement and modeling. For instance, untill now a GRF has not been measured and still stays as a state-of-the-art problem [14]. In practice, the appropriate scattering function depends on the specific application. For instance, a scene with various textures can be

rendered with different scattering functions. Simpler materials that do not have complex scattering features can be rendered with a 2-D texture in combination with BRDF models, such as the Phong model. If the material cannot be represented realistic without subsurface scattering, masking, self-reflections then typically such materials require a high quality representations, e.g. BTF. However, these advanced material representations are very complex. In practice, a tradeoff between visual quality and rendering cost is inevitable.

Chapter 3

Bidirectional Texture Functions

3.1 Acquisition

BTF acquisition is a non-trivial task as it requires a special acquisition setup. Only a few measurement systems [27, 36, 8, 18, 22, 28] exist, but as the interest in realistic material rendering using BTFs is growing, measurement systems are developing. In this chapter we will review how in general BTF data acquisition is done, which post-processing steps are made, and advantages and disadvantages of existing measurement systems are discussed. We will also review publicly available BTF datasets.

3.1.1 General Acquisition Methods

All the mentioned BTF acquisition systems share the same idea in the data acquisition, i.e. capturing the appearance of a flat square slice of the material surface under varying light and camera directions. The material surface is usually sampled over a hemisphere above the material slice, as shown in Figure 3.1. Depending on the material's reflectance properties the sampling distribution may vary, e.g. the sampling distribution can be dense in regions where specular peaks in the light reflection occur. Then, if needed, a uniform distribution can be calculated in a post-processing step using interpolation [14].

Digital cameras are used as capturing devices. Depending on the setup the number of cameras can vary. If there is only one camera [27, 28, 8], it usually moves on a robotic arm or rail-trail system around the hemisphere above the sample[27]. The advantage of this approach is that it is less expensive and can suit low-budget applications. The disadvantage however is the positioning errors that can arise, which influence the overall measurement error. Depending on the application, light sources can be fixed or moveable.

There are approaches which do not involve camera and light source movement at all. Schwartz et al. [36] developed a novel measurement system which uses a dense array of 151 cameras, which are uniformly placed on a hemispherical structure. Flashes of the cameras are used as light sources. Such a setup provide a high angular and spatial resolution.

Ngan et al. [28] made a setup which does not involve camera movement by placing the planar slices of the material the form of a pyramid. Thus, such setup captures the material appearance for 13 different camera views at once. Light directions are sampled by a manually-moved electronic flash. The disadvantage of this setup is that it provides sparse angular resolution, but depending on the application such an approach can be plausible. For example, a material which does not exhibit high illumination changes can be sampled sparsely to reduce the amount of redundant data.

The material sample is commonly a flat and squared slice, which is placed on the holder plate. To conduct automatic post-processing borderline markers are placed on the holder. Those markers provide the important information for further post-processing steps such as image rectification and registration.

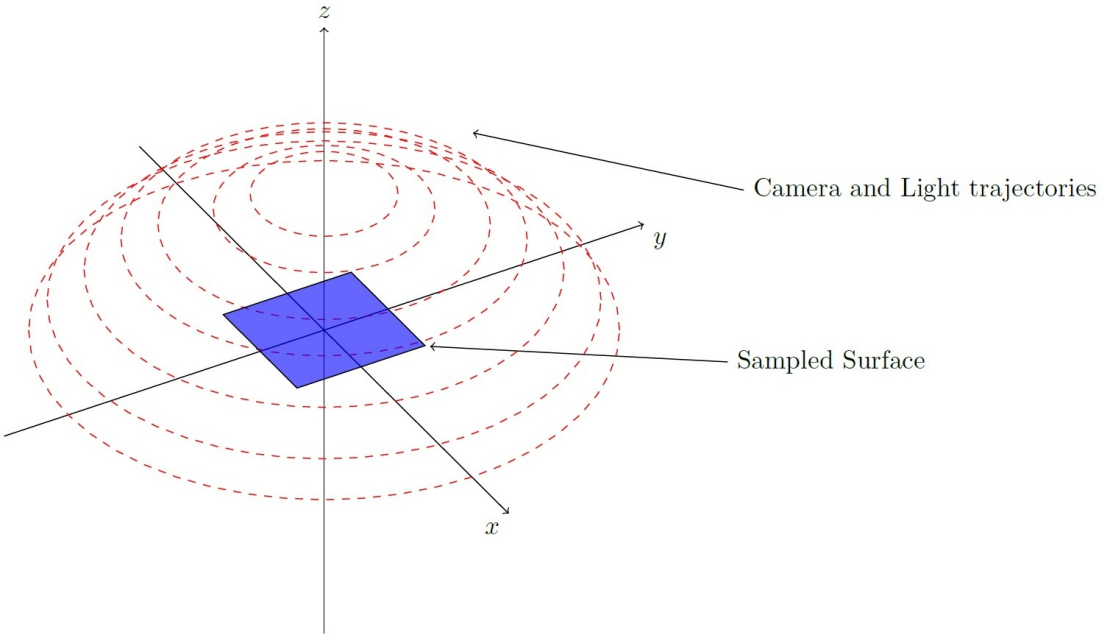

FIGURE 3.1: **Example of BTF measurement.**
Camera and light positions share the same trajectories. Red dashed circles are the sample positions on the hemisphere.

3.1.2 Post-processing

After the measurement is done, the raw data has to be further post-processed, because typically such data is not ready for compression. Raw data consists of a set of images

that are not aligned with each other and are not mutually registered.

When these raw images are obtained under different camera angles (θ, ϕ) they are perspectively distorted [33]. Thus, all sample images have to be aligned with each other and spatially registered to be further used. Firstly, borderline markers that were placed around the material sample on the holder plate aid the automatic detection of the material slice. Then, after the material slices are detected and cropped, they are ready for mutual alignment. This process is called *rectification*. *Rectification* is a process which involves projecting all sample images onto the plane which is defined by the frontal view, e.g. $(\theta_o = 0, \phi_o = 0)$. In other words, all normals of the sample images have to be aligned with their corresponding camera directions, i.e. as if all sample images were taken from frontal view $(\theta = 0, \phi = 0)$. The last step is image *registration*, a process of getting pixel-to-pixel correspondence between the images. Finally all images have to be scaled to an equal resolution.

Even after the proper rectification and registering of the measured data, registration errors can still be present between individual camera directions [14]. This happens due to structural occlusion of the material surface. Because of such self-occlusion some geometries structures are not captured by certain camera directions. That is why even after rectification images captured from completely different directions are not correctly aligned registration errors can also be caused by both inaccurate camera and material sample positions happened during the measurement processes.

If needed, further processing steps can be done, for instance *linear edger blending* to reduce tiling artifacts [33]. Also, typical image processing steps may be employed, e.g. noise reduction filters.

3.1.3 Publicly Available BTF Datasets

The accurate rendering of the material surface highly depends on the quality of the acquired data, especially for BTFs. There are several properties that are vital for reproducing high quality rendering results. BTF datasets can be distinguished by how well image post-processing were done and how good spatial and angular resolutions are.

A pioneer in the BTF acquisition was Dana *et. al.* [2], who measured 61 materials with fixed lighting and a moving camera aided by a robotic arm. This procedure resulted in a set of images, which can be seen as a subset of a BTF, which is called surface light field (SLF), see Chapter 2.8. Dana *et. al.* *CUReT* database is publicly available [2]. For each measured surface Dana *et. al.* used 205 different combinations of camera and light directions, which resulted in relatively sparse angular resolution. These datasets are not

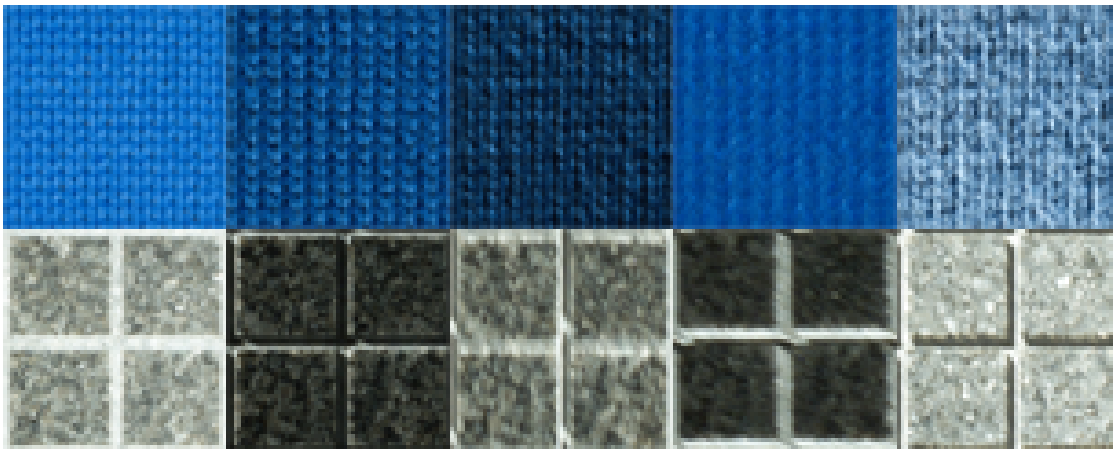

FIGURE 3.2: **BTF example of Bonn Database [1].**
Example how BTF catches rich appearance of the material due to dependencies on light and camera directions. Upper row is a knitted wool, lower is a graved granite stone.

rectified, but the authors provide image coordinates to enable rectification. Because of these limitations such BTF dataset are usually used for computer vision purposes, i.e. texture classification [14].

Based on this BTF measurement system, Bonn University created their own measuring system [33]. The main difference of this system is that the camera moves on a semi-circle rail around the material sample. Such a setup provides spatially rectificated and mutually registered data, with reasonable angular and spatial resolutions. Datasets of the Bonn University [1] are publicly available and are used in this thesis.

Consider Figure 3.2, which illustrates sample materials from the Bonn database. The measured surface is being fixed all the time on the sampler holder as shown in Figure 3.1. For each light position, a camera takes a shot of the material while moving from point to point on the hemisphere. Bonn University datasets have the same trajectory for camera and light positions, i.e. 81 positions on the hemisphere, which resulted in $81 \times 81 = 6561$ number of acquired images. After that the sample images were rectificated and registered, resulting in a set of images with a spatial resolution of 256×256. Typically, the size of one uncompressed BTF is around 1.2 Gb.

3.2 Data Representations

Before doing any further operations on the acquired BTF data it is important to chose a suitable representation for the data. A suitable presentation can influence the final quality of the BTF rendering and the compression ratio enormously.

3.2.1 Texture Representation

The first straightforward representation as a set of textures can mathematically represented as

$$BTF_{Texture} = \left\{ I_{(\theta_i,\phi_i,\theta_o,\phi_o)} \mid (\theta_i, \phi_i, \theta_o, \phi_o) \in M \right\}$$

where M denotes a set of images $I_{(\theta_i,\phi_i,\theta_o,\phi_o)}$ measured for different light and camera directions (i, o) accordingly.

Basically, $BTF_{Texture}$ is used for compression methods that do their analysis on the whole sample plane.

3.2.2 ABRDF Representation

ABRDF (Chapter 2.6) representation is a set of ABRDFs, one for each sample position of the material plane. In this case it is called *apparent*, because BTF includes effects such as self-occlusions and sub-surface scattering which violate two basic properties of a BRDF, i.e. *energy conservation* and *reciprocity*.

$$BTF_{ABRDF} = \left\{ P_{(x)} \quad \mid (x) \in I_{(\theta_i,\phi_i,\theta_o,\phi_o)} \subset \mathbb{N}^2 \right\}$$

BTF_{ABRDF} denotes a set of $P_{(x)}$ images, i.e. ABRDF for spatial position x.

BTF_{ABRDF} representation allows better pixel-to-pixel comparisons, which can give a big advantage for methods that employ pixel-wise compression, e.g. BRDF based models.

Also, such arrangement provides images with lower variance [16], which can allow better compression results in certain scenarios, e.g. when the material surface is not smooth.

3.3 BTF Compression Methods

A measured BTF consists of thousands of images, which require lots of storage space. In the Bonn Database [1] a BTF consists of 8-bit PNG images with a resolution of 256×256 sampled for 81×81 different camera and light directions. An uncompressed BTF is approximately 1.2 GB in size. To render a scene with several BTFs and to achieve an acceptable frame-rate becomes practically impossible, especially for low-end hardware. Also, as we intent to render BTFs in a web-browser, the data has to be transferred from the server to the client, which means the compact representation of the BTF is of utmost importance. For our scenario it is important to chose a compression method,

that allows for real-time decompression, high compression rates while preserving good quality, and separability of compressed BTF data. Separability is needed for real-time streaming in a web-browser. As the data is streamed in small chunks, partial data has to be renderable to produce a preview of the BTF.

There are different methods for BTF compression. Those methods can be categorized as follows:

- *Analytic methods* represent BTFs through analytic BRDF models. These analytic BRDF models are fitted separately to each texel of the BTF. These functions only have a few parameters, thus real-time performance is easily achieved. However, these groups of methods can suffer from decreased rendering quality [16]. In addition, it is hard to change parameters in order to control the visual quality. (See Chapter 3.3.1).
- *Statistical methods*, which belong to methods that reduce dimensionality based on statistics, for instance based a linear basis decompression method such as PCA (Principal component analysis). PCA based methods are frequently used, because its parameters directly correspond to the trade-off between compression ratio and reconstruction quality. Also, PCA is frequently the basis for some more sophisticated methods [35]. (See Chapter 3.3.2.)
- *Probabilistic models* can achieve high compression rates, but the resulted image quality is generally only suitable for flat surfaces. Additionally, the implementation on a GPU is problematic. (See Chapter 3.3.3.)

3.3.1 Analytic methods

This group of methods take advantage of the ABRDF representation of a BTF. There is a large number of techniques which allow for a compact representation of BRDFs, which in general can also be applied to ABRDFs. Each texel of a BTF, i.e. spatial position on the surface are represented as a ABRDF. Each of these ABRDFs can be modeled and compressed by any BRDF model.

One of the possible ways to model ABRDF is to use *Polynomial Texture Mapping* (PTM). Malzbender *et. al.* [24] used this approach which allows for high compression rates and good quality. However, PTM requires to compute specular and diffuse effects separately. The PTM model assumes that the input surface is either diffuse or their specular contribution has been separated beforehand. For BTFs it can be difficult to separate specular highlights [16].

Haindl *et. al.* [16] applied PTM on Reflectance Fields (PTM RF)

$$R_o(r,i) \approx a_0(r)u_x^2 + a_1(r)u_y^2 + a_2(r)u_x u_y + a_3(r)u_x + a_4(r)u_y + a_5(r)$$

where R_o is the approximated RF for a fixed camera direction o and u_x, u_y are projections of the normalized light vector into the local coordinate system $r = (x, y)$. The set of all possible R_o is the number of all camera positions, i.e. n_o. Coefficients a_p are fitted by the use of *singular value decomposition* (SVD) for each R_o and parameters are stored as a spatial map referred to as a PTM.

This method enables fast rendering and is generally suited for smooth material surfaces. However, Malzbender *et. al.* [24] claim that this method produces considerable errors for grazing angles.

Another model which produces slightly better visual quality is the polynomial extension of the one-lobe Lafortune model (PLM) [16].

$$Y_o(r,i) = \rho_{o,r}(C_{o,r,x}u_x + C_{o,r,y}u_y + C_{o,r,z}u_z)^{n_{o,r}}$$

where $w_i(\theta_i, \phi_i) = [u_1, u_2, u_3]^T$ is the unit vector pointing from surface to light. Parameters ρ, C_x, C_y, C_z, n can be computed with a Levenberg-Marquardt non-linear optimisation algorithm [12]. Filip and Haindl [12] claim that the one-lobe Lafortune model produces unsatisfactory results for complex ABRDFs. Thus, the polynomial extension of the one-lobe Lafortune was introduced (PLM RF):

$$R_o(r,i) \approx \sum_{j=0}^{n} a_{r,o,i,j} Y_o(r,i)^j$$

PLM RF solves the problem of bad quality for grazing angles and improves the rendering quality compared to PTM RF. However, statistical based methods produce even better quality compared to above methods but with lower compression rates [16].

3.3.2 Statistical methods

This group of methods is mostly based on a dimensionality reduction of the data, while preserving the most important information. The goal of dimensional reduction is to find a new basis which would represent the data with less dimensions and at the same time preserving the important features of the original data. The size of the data with the new basis will be decreased.

One of the popular methods used for BTF compression belongs to this group, i.e. *Principal component analysis* (PCA) [6, 34, 16, 35, 33, 26, 29]. PCA is a linear transformation of the data to a new basis, where each new coordinate (variable) is called a *principal component* [6]. All new coordinates are mutually orthogonal, i.e. uncorrelated. Components are sorted by decreasing variance, i.e. the first component posses

the greatest variance compared to other components. The last components which hold small variations are discarded, because they do not contribute important information.

PCA is a popular approach for compressing BTF. Firstly, PCA allows for a strong correspondence between the number of components and the decompression error. The correspondence lies in the amount of components used. The more components used the better the decompression error is, and vise-verse, a smaller number of components gives better compression ratio, but worse decompression error. This allows for an easy regulation of the trade-off between rendering quality and compression ratio. Secondly, a reasonable compression ratio along with a low decompression error can be achieved, e.g. a compression ratio of 1 : 100 with an average decompression error of 2%- 4% [34, 16]. Thirdly, PCA is suitable for real-time decompression on average GPUs [34, 16]. It is suitable, because decompression of a single pixel does not depend on other pixels. Computation of a single pixel for PCA decompression on the GPU is done by means of linear combinations of estimated PCA parameters, so the speed of decompression depends on the number of components used.

There exist different methods based on PCA, for instance PCA *Reflactance Field* (RF), PCA BTF and local PCA (LPCA) [33, 34, 26, 16]. Sattler *et. al.* [33] developed PCA RF. This method ensures a pixel position coherence, by employing PCA per camera direction. The advantage is that on average 8 components are enough to get good results with decompression errors around 2%- 4%. On the other hand, PCA BTF, which does PCA on all camera directions at once, requires on average 41 components [16]. Müller *et. al.* [34] improved PCA BTF by exploiting vector quantization techniques and applying PCA on the resulted clusters. Local PCA (LPCA) requires 19 components on average [16]. Even though the compression ratio of PCA RF is lower than PCA BTF and LPCA, it is less computational demanding.

In general, any PCA method is suitable for streaming, as it is possible to stream components separately and progressively enhance the rendering quality.

3.3.3 Probabilistic models

Compression methods based on probabilistic models [15, 20, 16] provide much higher compression rates than most others methods and allow for a seamless enlargement of textures of any size [16]. Usual compression ratios achieved by such models are $1 : 6-8\times10^4$. The visual quality of the rendered results are best suited for smooth materials, while materials with high frequencies can appear flat using probabilistic methods [16]. Also, these type of models can be problematic for implementation on low-end GPUs. The problem arise in shaders, when probabilistic models require information of spatial

neighbours to synthesize the texture [20, 16]. Even though solutions exists to handle this problem, e.g. take advantage of Framebuffer Objects, which allow access to previously rendered results [16], these solutions are time-consuming for low-end GPUs.

Chapter 4

PCA

In this chapter we explain the derivation of PCA and the algorithm for compressing and decompressing the BTF. We have chosen PCA for BTF compression, which belongs to statistical methods, see Chapter 3.3.2. PCA compression methods allow for easy regulation of the trade-off between rendering quality and compression ratio. Analytical methods can produce errors at gazing angles and generally have bigger decompression error than PCA [16]. Also, PCA is suitable for streaming, because the components are sorted in order of importance. This means there is no need to do additional analysis to decide in which order to send the components.

4.1 Derivation of PCA

PCA can be defined through a maximum variance formulation, as done by Bishop [6]. Consider a set of variables $\{x_n\}$, where $n = 1..N$. x_n are D-dimensional vectors. The goal is to project this data to an orthogonal basis while maximizing the variation of each new variable. For the sake of simplicity, consider the projection to a one-dimensional space, i.e. to a new basis which consist of one vector u_1. As the magnitude of the vector is not important in this case, let it be a unit vector, i.e. $u_1^T u_1 = 1$. Then, each variable x_n is projected onto the new basis, i.e. a scalar $u_1^T x_n$.

To compute the variance of the projected data, first we need to define the mean of projected data:

$$\frac{1}{N}\sum_{n=1}^{N} u_1^T x_n = u_1^T(\frac{1}{N}\sum_{n=1}^{N} x_n) = u_1^T \overline{x}.$$

Then, the variance of the projected data can be expressed as:

$$\sum_{n=1}^{N}(u_1^T x_n - u_1^T \overline{x})^2 = u_1^T S u_1$$

where S is the covariance matrix defined as:

$$S = \sum_{n=1}^{N}(x_n - \overline{x})(x_n - \overline{x})^T.$$

The next step is to maximize the variance $u_1^T S u_1$ with respect to u_1. In order to avoid $\|u_1\|$ growing to infinity, an additional constrain has to be used, i.e. normalization constrain $u_1^T u_1 = 1$.

Then, the problem can be expressed as an optimization problem:

$$\begin{aligned} &maximize \quad u_1^T S u_1 \\ &subject\ to \quad u_1^T u_1 = 1 \end{aligned}$$

This can be solved with a Lagrangian multiplier:

$$u_1^T S u_1 + \lambda_1(1 - u_1^T u_1).$$

By taking the derivative with respect to u_1 and setting it to zero, the local extremum can be found:

$$\frac{\partial}{\partial u_1} u_1^T S u_1 - \lambda_1 \frac{\partial}{\partial u_1} u_1^T u_1 = 0.$$

The result of taking the derivative will be the following equation [6]:

$$S u_1 = \lambda_1 u_1$$

This implies that this is an eigenvector and an eigenvalue problem, where u_1 is the eigenvector and λ_1 is the eigenvalue. So, the maximum will be when each of the eigenvector u_1 will have the largest eigenvalue λ_1. The vector u_1 will be the *principal component.*

In the same way it is possible to define the other of principal components, which maximize the variance of the projected data with the condition that all new components are orthogonal to each other.

4.2 SVD as PCA

Consider a set of variables x_n to be the columns of matrix X. To compute the covariance matrix S, the matrix X has to be centred, i.e.

$$X_c = X - 1\overline{x}$$

where $\overline{x}$ is the vector of column-averages of matrix X and matrix 1 is the matrix of ones. Then, the covariance matrix can be calculated in the following way:

$$S = X_c X_c^T$$

In practice to find eigenvectors and eigenvalues of covariance matrix S can be done by means of a *singular value decomposition*(SVD). SVD does not require S to be computed,

instead it is enough to perform an SVD on the matrix X_c. For any real matrix X_c there exists a decomposition [41]:

$$X_c = U\Sigma V^T$$

where U and V are orthogonal matrices and Σ is a diagonal matrix consisting only of singular values. The diagonal values of Σ are the square roots of the eigenvalues of $X_c X_c^T$ [23].

$$\begin{gathered} X_c X_c^T = (U\Sigma V^T)(U\Sigma V^T)^T \\ X_c X_c^T = (U\Sigma V^T)(V\Sigma U) \\ V^T V = I \\ X_c X_c^T = U\Sigma^2 U^T \end{gathered}$$

From the eigen decomposition theorem [40] follows that matrix U holds orthogonal eigenvectors of $X_c X_c^T$ and Σ contains the square roots of the eigenvalues. Thus, the decomposition of $X_c = U\Sigma V^T$ contains the eigenvectors and eigenvalues needed for a PCA.

4.3 Algorithm

The PCA algorithm used for BTF compression and decompression is equal to the one described by Borshukov *et. al.* [29, Ch. 15]. We apply PCA per k neighbour camera directions. If $k = 1$ this method becomes equal to the PCA RF method [16]. If we take the whole number of camera directions it equals the PCA BTF method [16]. PCA RF requires less principal components to store per camera direction and performs better in real-time rendering. While PCA BTF has better compression ratio, but is slower than PCA RF in real-time rendering (See Chapter 3.3.2). Our proposed method is flexible and gives the possibility to chose in between of these methods, i.e. regulate the trade-off between compression ratio and real-time computational performance.

4.3.1 Compression

In the *first* step of the compression algorithm we built a ABRDF representation of the BTF data. Let matrix A denote the stored BTF data. We consider each image I as three column vectors a_i (red, green, and blue channels), where $i = 0..(3N)$. N is the total number of images in the BTF dataset and 3 is the number of channels per image. The size of a_i is $W \times H$, where W and H are the dimensions of the image. The matrix A, thus, has the following dimensions $(W * H) \times (3N)$, which consist of columns a_i. Rows of the matrix A are ABRDF representations, which will be dimensionally reduced.

The *second* step is called "centring" of the data. We compute the average value of each row of matrix A

$$m_i = \frac{1}{3N} \sum_{i=1}^{3N} A_{i,j}$$

Then, we subtract the mean vector from each column of A

$$B_{i,j} = A_{i,j} - m_i.$$

At last, we compute the singular value decomposition (SVD) of the matrix B. The result of which will be the following decomposition:

$$B = U\Sigma V^T$$

where U holds the *principal components* of size $W \times H$, Σ is a diagonal matrix and holds the "importance" value of each principal component, and matrix V stores weights that are needed for the reconstruction of B.

4.3.2 Decompression

The decompression only involves matrix operations, which combine the three matrices U, Σ, V and the mean vector m. To easier decompress the data on a GPU we construct new matrices following Borshukov *et. al.* [29, Ch. 15]

$$L = \left[m \mid U\Sigma \right]$$
$$R = \begin{bmatrix} 1 \ldots 1 \\ V^T \end{bmatrix}.$$

The matrix A is then expressed as $A = LR$. To decompress the image at index i we separately reconstruct each color channel as follows:

$$red(x,y) = \sum_{k=1}^{C} L_{xy,k} R_{k,3i+0}$$
$$green(x,y) = \sum_{k=1}^{C} L_{xy,k} R_{k,3i+1}$$
$$blue(x,y) = \sum_{k=1}^{C} L_{xy,k} R_{k,3i+2}$$

4.4 Angular Interpolation

BTF data is measured for a discrete set of light and camera directions, thus it is necessary to perform interpolation to find the color values for unmeasured directions. We use a 2-D linear interpolation utilizing barycentric weights [14].

The algorithm involves finding the closest directions and then calculating barycentric weights for interpolation.

4.4.1 Finding Closest Directions

Depending on the material sampling, uniform or non-uniform, different strategies can be applied for finding the closest directions for an input angle. One of the strategies is to compute it each time or use a precomputed cubemap [16].

The Bonn database [1] provides data with uniform sampling for latitude. Depending on the latitude position, different quantization steps applied for longitude positions. For areas closer to the bottom of the hemisphere the quantisation for longitude gets smaller. This is done to have relatively equal distances between the directions for any longitude position.

Assume, a set of quantisation steps $S = \{(\Delta\theta * n, \Delta\phi_n) \mid n = 0..M\}$, where M is the number of quantization steps. For $\theta = 0°$ only one image is taken, i.e for $(0°, 0°)$ direction.

First, we find the four closest directions, and then decrease it to three closest directions. Let the direction for which we need to find the closest directions be denoted as P=(θ^p, ϕ^p). To find closest points for θ^p, we use the following functions accordingly, which compute lower and upper bounds for the input angle:

$$f^l(x, \Delta) = \Delta * \left\lfloor \frac{x}{\Delta} \right\rfloor,$$
$$f^u(x, \Delta) = f^l(x, \Delta) + \Delta.$$

First we find lower and upper bounds for latitude, i.e. $(\theta_L, \theta_U) = (f^l(\theta_p, \Delta\theta), f^u(\theta_p, \Delta\theta))$. Then, for (θ_L, θ_U) we compute bounds for longitude. i.e. lower and upper bounds at lattide θ_L: $(\phi_L^1, \phi_U^1) = (f^l(\phi_p, \Delta\phi_n), f^u(\phi_p, \Delta\phi_n))$, where $n = \frac{\theta_L}{\Delta\theta}$. Finally for θ_U we get that $(\phi_L^2, \phi_U^2) = (f^l(\phi_p, \Delta\phi_n), f^u(\phi_p, \Delta\phi_n))$, where $n = \frac{\theta_U}{\Delta\theta}$.

The resulting four closest directions to the direction P are: $A = (\theta_L, \phi_L^1)$, $B = (\theta_L, \phi_U^1)$, $C = (\theta_U, \phi_L^2)$ and $D = (\theta_U, \phi_U^2)$, as shown in Figure 4.1.

We, then reduce to the three closest directions, so less interpolation weights are to be computed.

One possible way to find the closest three directions is to compute the distance between P and all other four directions and to discard the furthest one. However, this is computational heavy for real-time applications. The less computational demanding way is to approximately find the triangle to which P belongs, i.e. ABC or CBD. The following method produces unnoticeable differences of visual results compared to the method which tests if the point P belongs to ABC or CBD.

We compute to which direction P is closer, i.e. whether to A or D. The distance is computed as follows:

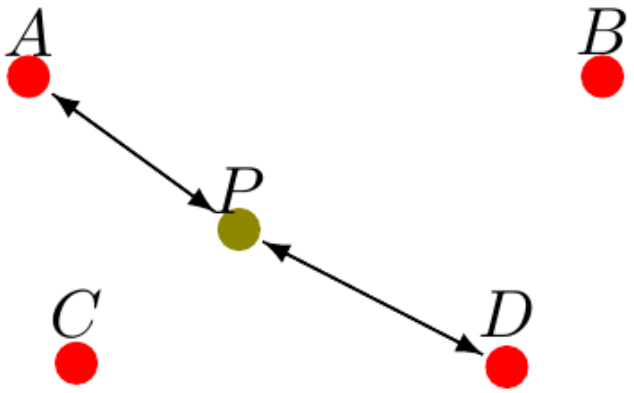

FIGURE 4.1: **Closest Directions**

$$d = \sqrt{(x-x')^2 + (y-y')^2 + (z-z')^2} =$$
$$\sqrt{r^2 + r'^2 - 2rr'(sin(\theta)sin(\theta')cos(\phi)cos(\phi') + sin(\theta)sin(\theta')sin(\phi)sin(\phi') + cos(\theta)cos(\theta'))} =$$
$$\sqrt{r^2 + r'^2 - 2rr'(sin(\theta)sin(\theta')cos(\phi - \phi') + cos(\theta)cos(\theta'))}$$

Note that, in practice r and $r^{'}$ are equal 1. As we are interested in comparing distances it is enough to compare this term:

$$d^{'} = sin(\theta)sin(\theta^{'})cos(\phi - \phi^{'}) + cos(\theta)cos(\theta^{'})$$

The bigger the term $d^{'}$ the smaller the overall distance. If P is closer to A, the resulting three directions are A, B, C. If P closer to D then B, C, D.

If the input direction P is beyond the measuring directions, i.e. $\theta_p > \Delta\theta * n$ for any n. We take the two closer directions, i.e. $C = (\theta_U, \phi_L^2)$ and $D = (\theta_U, \phi_U^2)$ and perform linear interpolation between these two directions.

4.4.2 Barycentric Coordinates

A common interpolation technique for the BTF is barycentric coordinates interpolation. However, as it is computational heavy, the approximation algorithm proposed by Hatka and Haindl [19] will be used.

Assume that a triangle $P_1P_2P_3$ bounds input point P, for which we want to compute interpolation weights. Figure 3.1 demonstrates the hemisphere on which triangle $P_1P_2P_3$ lies. C_P denotes desired pixel color. In general, linear interpolation of that pixel will be $C_P = w_1C_{P1} + w_2C_{P2} + w_1C_{P2}$, where C_{P1}, C_{P2}, C_{P3} correspond to color values of the found triangle $P_1P_2P_3$ and w_1, w_2, w_3 are normalized and sum up to 1.

The weights w_1, w_2, w_3 are defined as volumes V_1, V_2, V_3, which correspond to the tetrahedrons PP_2P_3O, PP_3P_1O, PP_1P_2O, where $O = (0, 0, 0)$. These volumes can be calculated as determinants:

$$w_1 := V_1 = \frac{1}{6} |det(PP_2P_3O)|$$
$$w_2 := V_2 = \frac{1}{6} |det(PP_3P_1O)|$$
$$w_3 := V_3 = \frac{1}{6} |det(PP_1P_2O)|$$

The last step is the normalization, i.e. $V_i = \frac{V_i}{\sum_{i=1}^{3} V_i}$.

4.4.3 Algorithm

Computing interpolation weights for the input direction P, requires the closest measured directions $P_1P_2P_3$. After these three directions are known for the input direction P, interpolation weights can be computed as explained in Chapter 4.4.2.

To compute the final interpolated color we combine interpolation weights of light and camera directions. Assume that I and O are light and camera directions. Then, bounding triangles for both of them are $I_1I_2I_3$ and $O_1O_2O_3$ accordingly. The corresponding barycentric weights are $b_i = [b_{i1}, b_{i2}, b_{i3}]^T$ and $b_o = [b_{o1}, b_{o2}, b_{o3}]^T$. The final color calculated as the linear combination of b_i, b_o weights and the measured color values

$$C_f = \sum_{u=1}^{3} b_{iu} \sum_{v=1}^{3} b_{ov} C_{uv},$$

where C_{uv} is a color value that corresponds to a light direction I_u and a camera direction O_v.

Chapter 5

Implementation

The implementation of the prototype is divided into three main parts: compression, streaming and rendering. The compression is implemented as a standalone Java application, which compresses BTFs from the Bonn University [1]. It is possible to adapt any other BTF databases for our prototype by resampling the BTFs [13]. For the streaming server implementation Node.js [4] is used. The client side 3D rendering is done using XML3D [37], a declarative 3D approach that integrates 3D graphics into the DOM. An overview of the architecture of the prototype is depicted in Figure 5.1.

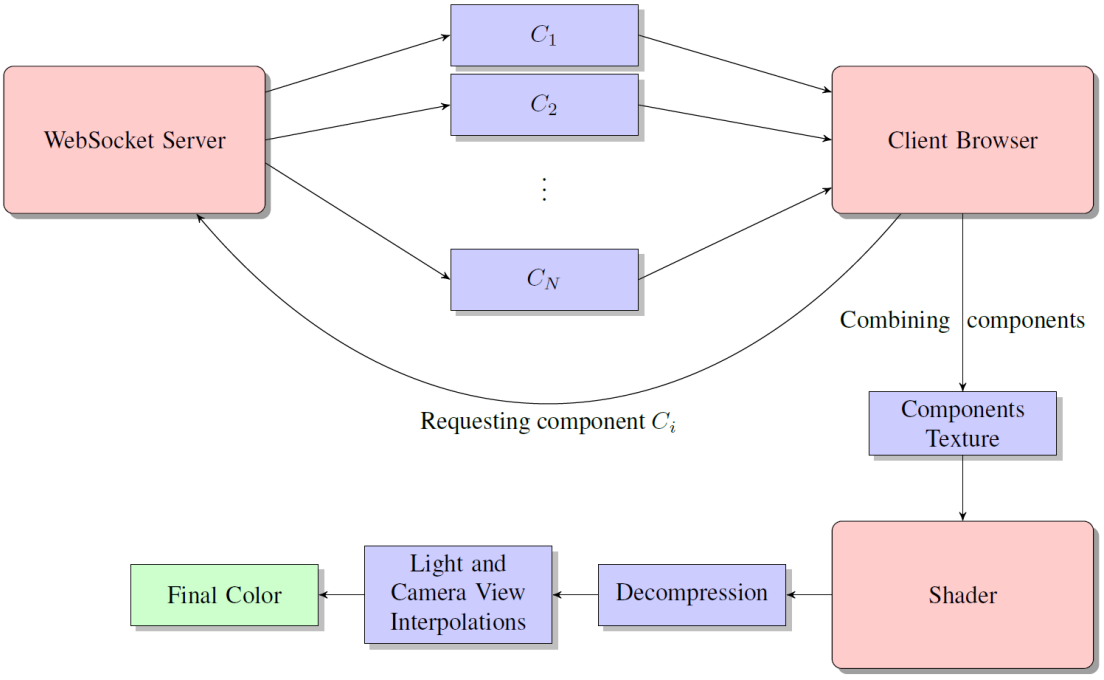

FIGURE 5.1: **Model Overview**

5.1 Compression

The implementation of the singular value decomposition (SVD) is the main part of the PCA implementation. We use *jblas* [3] developed by Mikio Braun, is a BLAS (Basic Linear Algebra Subprograms) implementation for Java.

The compressed BTFs have to be sent to the shader. OpenGL Shading Language (GLSL) version 1.0 supports uniform arrays, but does not fully support dynamic indexing [5]. Fortunately, it is possible to transform arrays of data into textures. After performing SVD, the matrices U, Σ, and V are saved as PNG images. (See Chapter 6.2). The values of U and V are in the range of $[-1; 1]$. Those values have to be mapped to the image domain, i.e. $[0; 1]$.

Each component of the matrix U is stored as a separate PNG image. For example, if the compressed BTF data has C principal components, then this would result in C separate images. This is done for streaming purposes, to transfer each component one by one from the server to the client. We use RGBA color space to save four components into one pixel, because it is possible to do efficient calculations in the shader by using vector multiplication. The matrices V and Σ are saved in a shared texture, as they are small in size.

The matrix Σ is a diagonal matrix. The values of Σ can be bigger than the image color value, i.e. an 8-bit value. In practice the values of Σ for Bonn University BTFs [1] are not bigger than a four digit number $a_4a_3a_2a_1$. We split the value into two values, i.e.

$$\underbrace{a_4a_3}_{R}\underbrace{a_2a_1}_{G}$$

It means that two values of Σ are mapped into one pixel, i.e. one value to RG channels and the second to BA channels. If the value of Σ exceed the four digit number, then it is possible to adapt this technique further in a similar manner.

We noticed that for BTFs from the Bonn Database [1] the *jblas* [3] library produces relatively sparse values for U and V, i.e. close to zero. Due to floating point imprecisions this sparsity results in decompression errors. To compensate for this precision limitation we scale the values of the matrices U and V. This improves the overall decompression error by approximately 5%. The scaling factor is determined by the following equation:

$$factor(M) = 10^{floor(min[log10(min(M)),log10(max(M)])}$$

where M is the matrix U or V. The term $floor(min[log10(min(M)), log10(max(M))])$ calculates the degree with which it is possible to multiply the matrix and preserve the original values range, i.e. $[-1; 1]$. If it is not possible, the resulting factor will be 1.

As the decompression is computed as multiplication of $U\Sigma V$, we have to remain the original decompressed BTF values. We do this by multiplying all Σ values by the factor $\frac{1}{factor(M)}$ if we scale either U or V.

5.2 Streaming

When rendering BTFs in a web-browser, all data has to be transferred before rendering. Even compressed BTF data can be tens of megabytes in size and can take a considerable amount of time for transmission. We stream the BTF data to reduce the latency before rendering can start. The user will be able to see a low quality preview of the original BTF in just a few seconds. In our case, principal components are streamed one by one using WebSockets [39]. Each principal component covers the full angular domain, so that the BTF can be rendered for any camera and light direction at any time during the streaming process. The rendering quality is enhanced whenever a new component arrives.

WebSockets are an efficient way for real-time client server communication. They are widely supported by desktop as well as mobile browsers. WebSockets have several advantages over HTTP Polling, Long Polling, and Streaming approaches [39, Ch. 1]. As they provide a Full-Duplex connection, the client and the server can reuse the same connection for communication. In contrast Polling requires the client to regularly check for new information. This single connection dramatically reduces the latency. HTTP Streaming also uses a single connection, but only from the server to the client and is currently only available for ASCII data. Firewalls may additionally buffer the HTTP response, which can result in increased latency. All in all WebSockets save bandwidth, CPU time and reduce latency compared to HTTP streaming or polling approaches.

We use Node.js [4] as a server side platform, which enables realtime streaming using WebSockets. On the client side we use Xflow [21], a declarative data flow processing language integrated into XML3D [37]. Xflow is used to gather and store all already transferred principal components in a texture. (See Chapter 4.3.2).

Consider Figure 5.1 that depicts how the streaming works in practice. When the user connects to the streaming server and the HTML page loads completely, i.e. the 3D object is on the client side, the JavaScript client side sends the message to the Node.js server to start streaming the BTF data. All the compressed BTFs are located on the Node.js server. At the start of the stream, we first send the texture R and meta data. (See Chapter 4.3.2). Afterwards, indiviual principal components C_i are streamed as PNG images.

Received PNG images are decoded and stored into the texture L. (See Chapter 4.3.2). This update in turn refreshes the rendering. Even with the first principal component the rendering looks plausible. With further components the overall quality of the image improves, i.e. specularities are increasing and small micro-structures become visible.

Because JavaScript is single threaded every time the principle components are assembled into the texture L the framerate and responsiveness of the overall application is affected. This could be fixed by sending each principal component as an individual texture to the shader. However, the number of textures are limited by the specifications of the GPU.

5.3 Rendering

We use XML3D [37] to embed 3D graphics into the HTML page. The XML3D.js polyfill uses WebGL for rendering and enables the user to define custom GLSL shaders.

The shader design is depicted in Figure 5.2. The compressed BTF data is accessed in the shader in the form of two textures. The first texture, L, stores principal components and texture R stores PCA weights. (See Chapter 4.3.2). First, we find the three closest directions for the camera and light directions. (See Chapter 4.4.1). Then, we compute barycentric coordinates for interpolation as described in Chapter 4.4.2.

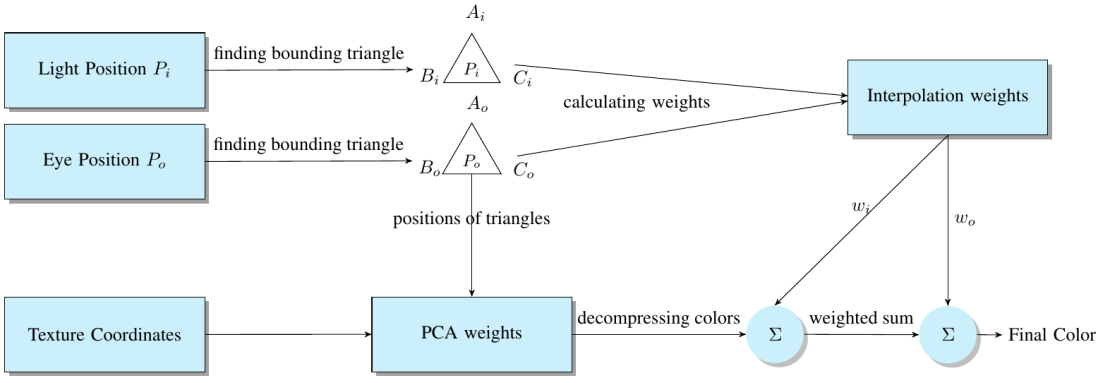

FIGURE 5.2: **Shader Design**

In the next step we sample the input textures to decompress the needed values for the found directions. We have three closest directions per camera and light directions. This means for the interpolation we need to have nine color values. For a given texture coordinates u, v we lookup the index of the first principal component, with which it will be possible to start the decompression. All the principal components are written linearly in the texture L. We have the following mapping to get the needed index:

$$indexL(i,j) = (b * N^2 + (i * N + j)) * C$$

where $i = \lfloor u * N \rfloor$, $j = \lfloor v * (N) \rfloor$, b is the number of subsets, N is the dimension of the compressed image and C is number of components. The parameter b depends on the number of subsets on which PCA is separately done. (See Chapter 4.3).

Texture R stores PCA weights. The mapping depends on the camera direction $v = (\theta_v, \phi_v)$ and light direction $l = (\theta_l, \phi_l)$. It is defined as:

$$indexR(v,l) = offset(\theta_v, \phi_v) + offset(\theta_l, \phi_l)$$

where $offset(\theta, \phi) = (\frac{\theta}{\Delta\theta} + \frac{\phi}{\Delta\phi})$, where $\Delta\theta$ and $\Delta\phi$ are quantization step sizes.

When all the indices are computed and textures L and R can be sampled, we decompress the colors as defined in Chapter 4.3.2. In a final step we combine nine decompressed colors and the early found interpolation weights to get the final color. (See Chapter 4.4.3).

The implemented shader has some limitations. First of all, the number of principal components has to be fixed and known beforehand, because GLSL version 1.0 does not support dynamic looping [5]. Secondly, the number of principal components is bound by the maximum dimensions a GPU allows for the texture L. This problem could be fixed using multiple textures. However, the number of possible active textures is limited, too.

Chapter 6

Evaluation

In this chapter we will evaluate the presented solution in regard to the decompression error, compression ratio, image quality and real-time performance during the streaming.

6.1 Compression

We evaluate our compression approach using the Bonn's BTF Database [1] and compare the results to other related PCA methods [16].

We tested different configurations for the BTF compression and found that the optimal configuration for our method is when $k = 3$ and $C = 8$, where k is the number of neighbour directions and C is the number of principal components. (See Chapter 4.3). Figure 6.1 shows an example of the wool material decompressed with various number of components. The Mean Average Error (MAE) in CIELAB color space is computed for each of them. The CIELAB metrics accounts for the human visual sensitivity, i.e. is consistent with human perception [31]. MAE is calculated for each of the channels separately and then the final result is averaged over them.

$$MAE = \frac{1}{N} \sum_{i=1}^{N} (|y_i - \hat{y}_i|)$$

Figure 6.1 shows that the decompression error is not significantly improved for 16 or even 32 components. 4 Components on the other hand result in a blurred image. This justifies our choice for 8 components. Haindl [16] also shows that 8 components is the optimal number of components for the PCA RF method, which is related to our method.

Table 6.1 provides the evaluation for the whole BTF space of the wool sample, i.e. for all possible camera and light directions. The MAE is also computed in RGB color space along with the Root Mean Square Error to measure the variance in the errors

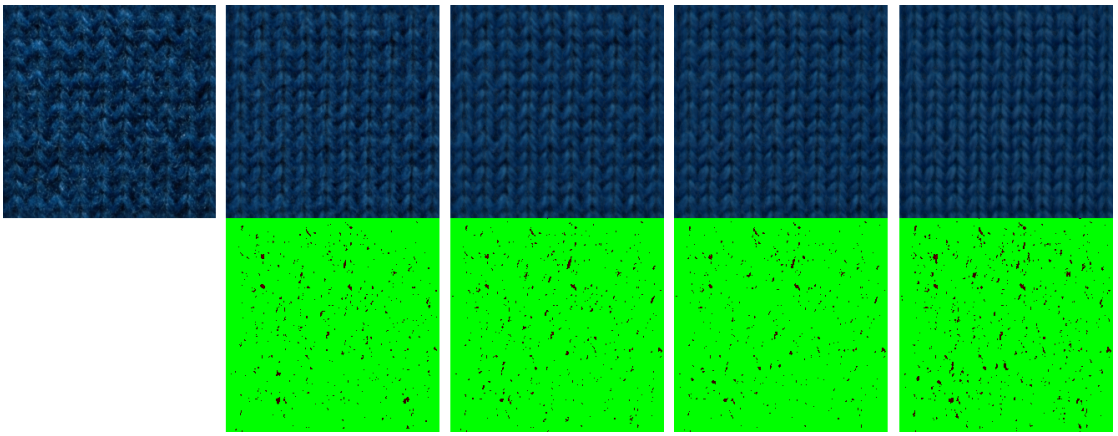

FIGURE 6.1: **Example of decompression errors**
First row from the left to the right: ground truth, **32** components (MAE:2.17), **16** components (MAE:1.83), **8** components (MAE:2.21), **4** components (MAE:3.04). **Second row**: the difference between the original and the decompressed images, *red* color denotes how big the error, *green* denotes that the error is absent or very small.

	k	C	MAE CIELab	MAE RGB	RMSE RGB	Compression Ratio	Parameters Size / Storage Size (PNGs)
wool	1	8	2.14	5.86	7.5	1:23	53Mb / 20 Mb
wool	3	8	2.19	6.83	8.68	1:70	18Mb / 5Mb
wool	3	32	2.22	5.92	7.5	1:17	70Mb / 25Mb

TABLE 6.1: Evaluation of BTFs

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2}$$

RMSE gives large weights to big errors, thus we can evaluate if big errors are present [7]. The bigger the RMSE the bigger the variance of the errors. We can see that in our case the RMSE is close to the MAE, which means that the variance of the errors is relatively small.

In the first row of the Table 6.1 where $k = 1$ our method becomes equal to the PCA RF [16] method. Haindl [16] has a MAE equal to 3.16 in CIELAB color space for the same sample. Our method produces better result, i.e. 2.14. The second row shows that for $k = 3$ MAE stays practically the same. However, the compression ratio improves by a factor of three. We can see that even for 32 components decompression errors improve only insignificantly, but the compression ratio becomes worse. If we use $k > 3$, more components would be necessary to reduce decompression errors. For intance, our method becomes equal to the PCA BTF [16] method if $k = 81$ (all camera directions). In this case approximately 41 components are necessary [16].

Also, the LPCA BTF [16] method uses 19 components on average and has the MAE equal to 2.42, while our method uses 8 components.

6.2 Real-time performance

We evaluate the rendering quality during the streaming and the real-time performance. Figure 6.2 depicts intermediate images during the streaming. We tested our approach on three materials: wool, impalla and corduroy. The parameters for the compression are $k = 3$ and $C = 8$.

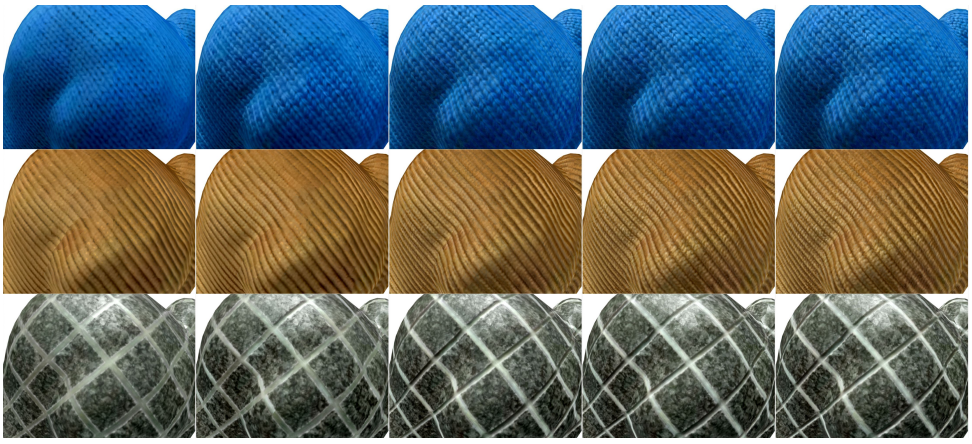

FIGURE 6.2: **Example of Progressive Streaming**
From left to right: *1*, *2*, *4*, *6*, *8* components rendered accordingly.

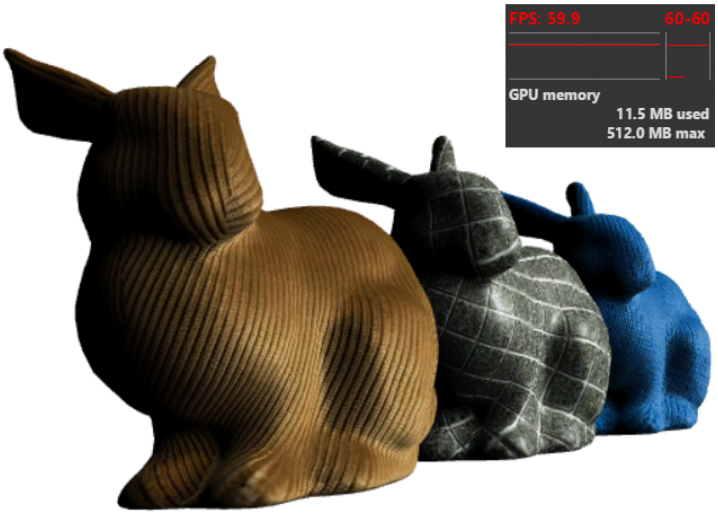

FIGURE 6.3: **Example of 3 BTFs rendered at the same time**
From left to right: corduroy, impalla and wool materials are rendered under dynamic light.

Rendering starts as soon as the first component transfers to the client side. The overall appearance of the the material is already visible even with the first component, which has an average size of about 0.7 Mb. With further components the overall quality of the image improves, i.e. specularities are increasing, small micro-structures become more visible and emphasized. On a desktop computer with an NVIDIA GeForce GTX

480 graphics card 3 BTFs can be rendered at 60 frames per second. Figure 6.3 shows an example of this.

On a mobile phone Sony Xperia SL, 15 frames per second on average are achieved for one BTF at a time.

6.3 Comparison with the Phong Shader

Figure 6.4 shows the tremendous difference between the BTF and a conventional 2-D texture in combination with the Phong shader. Both images were taken under the same camera and light directions. The BTF introduces high varying specularities and the realistic depth in the images. The Phong shader with the combination of the 2-D texture provides the impression that the material looks flat. The BTF shows the correct inner-reflections, sub-surface scattering and shadows under varying light conditions.

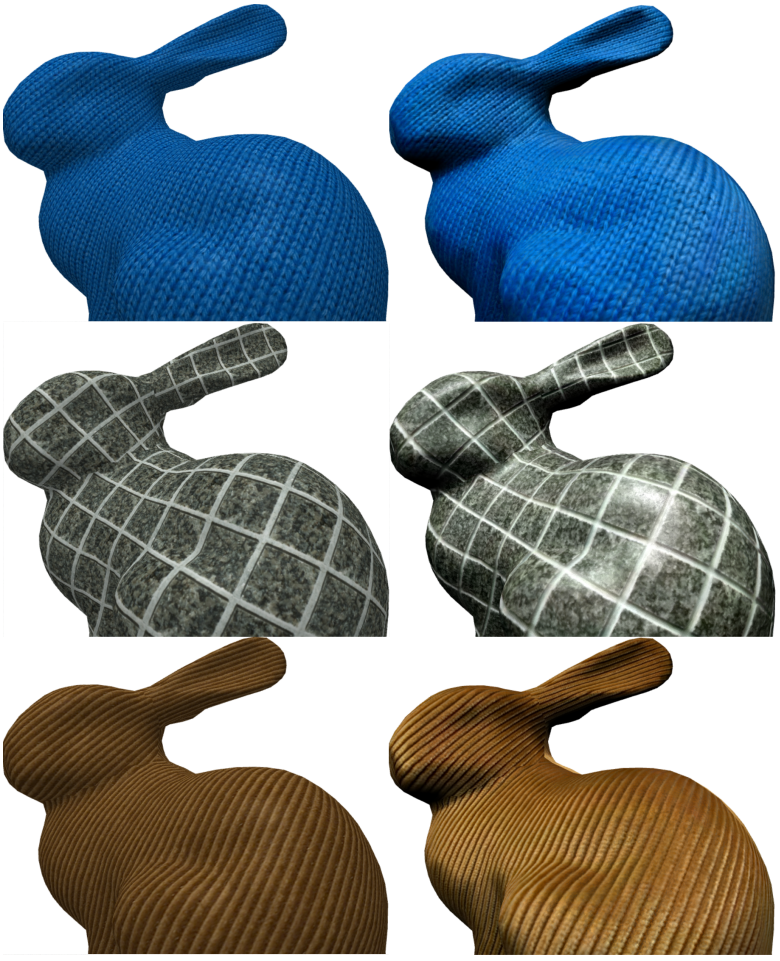

FIGURE 6.4: **The Phong model in comparison to the BTF**

Chapter 7

Conclusions and Future Work

Bidirectional Texture Functions are currently the best texture representation for materials that contain high frequencies in both the angular and spatial domain [26]. Thousands of images have to be taken to sample the appearance of such materials. to the huge size of an acquired BTF real-time rendering is impossible to achieve without suitable compression. Thus, BTFs are still rarely used and staying in a state of the art in computer graphics.

7.1 Conclusions

In this work we achieved the realistic and real-time rendering for a scene with several BTFs. We showed that real-time rendering is possible even on mobile devices. With constantly improving hardware BTFs have a bright future in computer graphics.

Considering that we render BTFs in a browser, we managed to reduce the latency that is caused by the data transmission, i.e. by streaming principal components individually. We managed to improve the decompression error that is caused by floating point imprecisions. The scaling of the compressed BTF data before converting it into the textures decreased the decompression error approximately by 5%. As a result this also improved the real-time frame rate and the compression ratio due to reduction in the number of used components. Last but not least, our method produces better decompression errors compared to other implemented PCA methods [16].

Finally, our method is flexible and allows for balancing between the visual quality, memory usage and computational effort.

7.2 Future work

Based on our results several directions for future work are possible. First of all, several optimizations of the BTFs performance. For instance, to reduce some of the calculations such as computation of the interpolation weights a precomputed cube map can be used [16]. Other calculations such as computation of the bidirectional normals can be precomputed and stored along with a 3D mesh.

Furthermore, there is room for further compression. For instance, PCA parameters can be further compressed using wavelet compression [35] or entropy encoding [29], if further memory savings are necessary.

Bibliography

[1] Btf database bonn 2003. http://cg.cs.uni-bonn.de/en/projects/btfdbb/download/ubo2003/. Accessed on January 2015.

[2] Curret btf database. http://www1.cs.columbia.edu/CAVE/software/curet/index.php. Accessed on January 2015.

[3] Linear algebra for java - jblas. http://mikiobraun.github.io/jblas/. Accessed on January 2015.

[4] Node.js websocket server implementation. http://nodejs.org/, note = Accessed on January 2015.

[5] The opengl es shading language 2013. https://www.khronos.org/registry/gles/specs/3.0/GLSL_ES_Specification_3.00.4.pdf. Accessed on January 2015.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] T. Chai and R. R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? arguments against avoiding rmse in the literature. *Geoscientific Model Development*, 7(3):1247–1250, 2014.

[8] K.J. Dana, B. Van-Ginneken, S.K. Nayar, and J.J. Koenderink. Reflectance and Texture of Real World Surfaces. *ACM Transactions on Graphics (TOG)*, 18(1):1–34, Jan 1999.

[9] Brian Danchilla. *Beginning WebGL for HTML5*. Apress, Berkely, CA, USA, 1st edition, 2012.

[10] Paul Debevec, Tim Hawkins, Chris Tchou, Haarm-Pieter Duiker, and Westley Sarokin. Acquiring the reectance field of a human face. In *SIGGRAPH*, New Orleans, LA, July 2000.

[11] Y. Dong, S. Lin, and B. Guo. *Material Appearance Modeling: A Data-Coherent Approach: A Data-coherent Approach*. SpringerLink : Bücher. Springer, 2013.

[12] J. Filip and Michal Haindl. Non-linear reflectance model for bidirectional texture function synthesis. In J. Kittler, M. Petrou, and M. Nixon, editors, *Proceedings of the 17th IAPR International Conference on Pattern Recognition*, pages 80–83, Los Alamitos, August 2004. IEEE, IEEE.

[13] Jiří Filip, Michael J. Chantler, and Michal Haindl. On optimal resampling of view and illumination dependent textures. In *Proceedings of the 5th Symposium on Applied Perception in Graphics and Visualization*, APGV '08, pages 131–134, New York, NY, USA, 2008. ACM.

[14] M. Haindl and J. Filip. *Visual Texture*. Advances in Computer Vision and Pattern Recognition. Springer-Verlag, London, 2013.

[15] Michal Haindl and J. Filip. A fast probabilistic bidirectional texture function model. In A. J. C. Campilho and M. Kamel, editors, *Image Analysis and Recognition*, pages 298–305, Heidelberg, September 2004. Springer, Springer.

[16] Michal Haindl and Jiri Filip. Advanced textural representation of materials appearance. In *SIGGRAPH Asia 2011 Courses*, SA '11, pages 1:1–1:84, New York, NY, USA, 2011. ACM.

[17] Michal Haindl and M. Hatka. Btf roller. In M. Chantler and O. Drbohlav, editors, *Texture 2005: Proceedings of 4th Internatinal Workshop on Texture Analysis and Synthesis*, pages 89–94, Edinburgh, October 2005. Heriot-Watt University, Heriot-Watt University.

[18] Jefferson Y. Han and Ken Perlin. Measuring bidirectional texture reflectance with a kaleidoscope. *ACM Trans. Graph.*, 22(3):741–748, July 2003.

[19] Martin Hatka and Michal Haindl. Btf rendering in blender. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 265–272, New York, NY, USA, 2011. ACM.

[20] Michal Havlíček. Bidirectional texture function three dimensional pseudo gaussian markov random field model. In *Doktorandské dny 2012*, pages 53–62, Praha, 11/2012 2012. České vysoké učení technické v Praze, České vysoké učení technické v Praze.

[21] Felix Klein, Kristian Sons, Dmitri Rubinstein, and Philipp Slusallek. Xml3d and xflow: Combining declarative 3d for the web with generic data flows. *Computer Graphics and Applications, IEEE*, 33(5):38–47, 2013.

[22] Melissa L. Koudelka, Sebastian Magda, Peter N. Belhumeur, and David J. Kriegman. Acquisition, compression, and synthesis of bidirectional texture functions. In *In ICCV 03 Workshop on Texture Analysis and Synthesis*, 2003.

[23] Feifei Li. Advanced topics in data management. University Lecture, 2008.

[24] Tom Malzbender, Tom Malzbender, Dan Gelb, Dan Gelb, Hans Wolters, and Hans Wolters. Polynomial texture maps. In *In Computer Graphics, SIGGRAPH 2001 Proceedings*, pages 519–528, 2001.

[25] Wojciech Matusik, Hanspeter Pfister, Matthew Brand, and Leonard McMillan. A data-driven reflectance model. *ACM Trans. Graph.*, 22(3):759–769, 2003.

[26] Gero Müller, Jan Meseth, and Reinhard Klein. Compression and real-time rendering of measured btfs using local pca. In T. Ertl, B. Girod, G. Greiner, H. Niemann, H.-P. Seidel, E. Steinbach, and R. Westermann, editors, *Vision, Modeling and Visualisation 2003*, pages 271–280. Akademische Verlagsgesellschaft Aka GmbH, Berlin, November 2003.

[27] Gero Müller, Jan Meseth, Mirko Sattler, Ralf Sarlette, and Reinhard Klein. Acquisition, synthesis and rendering of bidirectional texture functions. In Christophe Schlick and Werner Purgathofer, editors, *Eurographics 2004, State of the Art Reports*, pages 69–94. INRIA and Eurographics Association, September 2004.

[28] Addy Ngan and Frdo Durand. Statistical Acquisition of Texture Appearance. pages 31–40.

[29] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, August 2007.

[30] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Radiometry. chapter Geometrical Considerations and Nomenclature for Reflectance, pages 94–145. Jones and Bartlett Publishers, Inc., USA, 1992.

[31] Marius Pedersen and Jon Yngve Hardeberg. Full-reference image quality metrics: Classification and evaluation. *Found. Trends. Comput. Graph. Vis.*, 7(1):1–80, January 2012.

[32] Bui-Tuong Phong. Illumination for Computer Generated Pictures. 18(6):311–317, 1975.

[33] Mirko Sattler, Ralf Sarlette, and Reinhard Klein. Efficient and realistic visualization of cloth. In *Eurographics Symposium on Rendering 2003*, June 2003.

[34] Martin Schneider. Real-time btf rendering. In *The 8th Central European Seminar on Computer Graphics*, pages 79–86, April 2004.

[35] Christopher Schwartz, Roland Ruiters, Michael Weinmann, and Reinhard Klein. Webgl-based streaming and presentation of objects with bidirectional texture functions. *Journal on Computing and Cultural Heritage (JOCCH)*, 6(3):11:1–11:21, July 2013.

[36] Christopher Schwartz, Ralf Sarlette, Michael Weinmann, and Reinhard Klein. Dome ii: A parallelized btf acquisition system. In Holly Rushmeier and Reinhard Klein, editors, *Eurographics Workshop on Material Appearance Modeling: Issues and Acquisition*, pages 25–31. Eurographics Association, June 2013.

[37] Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. Xml3d: interactive 3d graphics for the web. In *Web3D '10: Proceedings of the 15th International Conference on Web 3D Technology*, pages 175–184, New York, NY, USA, 2010. ACM.

[38] Frank Suykens, Karl vom Berge, Ares Lagae, and Philip Dutr. Interactive rendering with bidirectional texture functions. *Comput. Graph. Forum*, 22(3):463–472, 2003.

[39] V. Wang, F. Salim, and P. Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apressus Series. Apress, 2012.

[40] Eric W. Weisstein. Eigen decomposition theorem. From MathWorld—A Wolfram Web Resource.

[41] Eric W. Weisstein. Singular value decomposition.From MathWorld—A Wolfram Web Resource.

[42] Chris Wynn. An introduction to brdf-based lighting. *NVIDIA Corporation*.