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

...

...

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.

Sperrvermerk

Blocking Notice

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 would like to express my sincere gratitude ...

To my family.

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 materials 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 renderings 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 becoming more and more popular. BTFs are rarely used 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 for on mobile-devices.

And the 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 possibly, while steadily increasing 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. This promises fast and reliable solution for real-time performance in *WebGL*-based application.

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

We implemented WebGL demo web-application and *Web-Sockets* server for streaming purposes.

1.1 Related Work

Nowadays, Web technologies are able to support WebGL [12], which is a JavaScript API for rendering 3D content. WebGL provides vertex and fragment shaders for creating highly realistic rendering results. New standards for 3D graphics in context of HTML domain are evolving, for instance such as XML3D [39]. XML3D allows for embedding 3D graphics into HTML pages. XML3D uses JavaScript and WebGL. It allows for declaring 3D scenes elements in the context of HTML pages. Web programmers can apply their existing knowledge (i.e. about DOM, XML, CSS) to include 3D graphics in the HTML content by using XML3D. Currently XML3D is a polyfill, because browsers do not support XML3D natively.

Most of previous work 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.* [37] 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.* [19, 17]. A trade-off has to be made between the rendering quality and compression rate depending on the intended application. Our compression method is related to the PCA RF (Principal Component Analysis Reflectance Field) method, which was introduced by Sattler *et. al.* [29]. 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 [19] 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 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 [14]. 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. Depending on the material there were introduced less computational functions without losing rendering quality. In order to chose suitable scattering function for capturing certain scattering effects for a particular type of material, it is important to know which functions already 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[44]. 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 may 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 originated. Light *absorption* is a process when a light is being taken up by a material and transformed into internal energy of the material, for instance thermal energy. When a material is transparent, light *transmittance* can occur.

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 [44]

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[29]. 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.

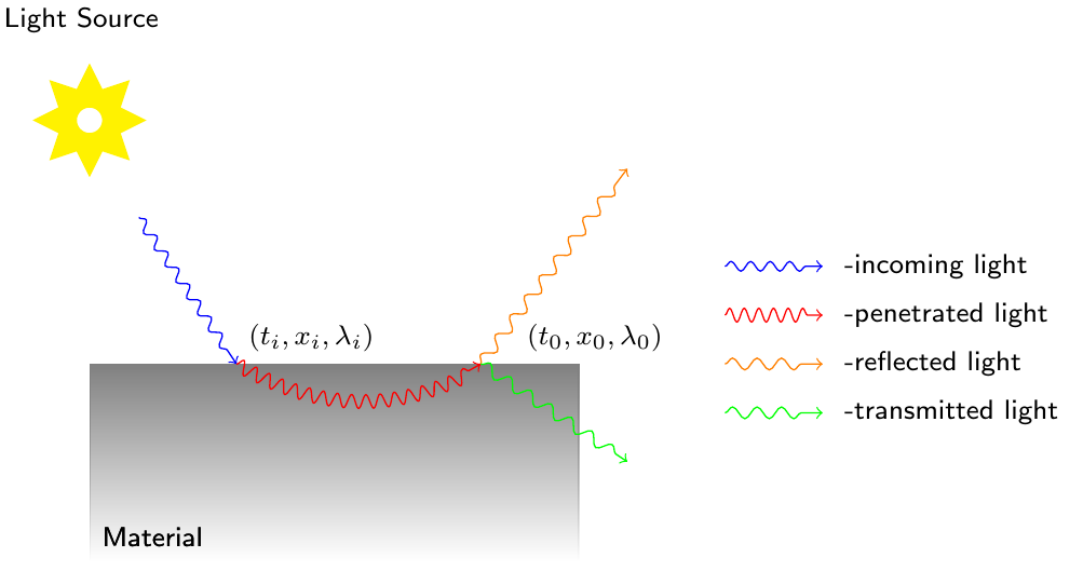

FIGURE 2.1: **Example of Light-Material interaction.**

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. Such function compromise 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. Usually such assumption are made [29]:

- 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 [29]:

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

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 [14], 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 the following 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. [32] 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 [44]. *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) [40].

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 [34] is the most used shading model in computer graphics. The traditional Phong model satisfy 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. The Phong model is an empirical model and is designed to fit the original function, often based on simple formulas which were derived from observations.

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 [17].

A SVBRDF represents structures at micro-scale level, which corresponds to near flat opaque materials. On the other side, a BTF capture 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. [17]

2.8 Surface Light Field and Surface Reflectance Field

Consider BTF and assume

- fixed light direction $\theta_i = const$, $\phi_i = const$

we will get a 4-D Surface Light Field model (SLF)

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

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

If we fix camera directions for BTF, we end up with SRF(Surface Reflectance Field). SRF is another very popular variant of image-based rendering and is also a subset of BTF. In this case many images are taken under varying light directions with fixed view point[29]. For instance, Debevec et al. [13] recorded the appearance of human face 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 directions.

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 [17]. 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 textures 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 not a non-trivial task as it requires special acquisition setup. There are only a few measurement systems [29, 38, 11, 21, 25, 30] 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. 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 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 [17].

Digital cameras are used as capturing devices. Depending on the setup the number of cameras can vary. If there is only one camera [29, 30, 11], it usually moves on a robotic arm or rail-trail system around the hemisphere above the sample[29]. 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 as a result.

There are approaches which do not involve camera and light source movement at all. Schwartz et al. [38] 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 setup provide a high angular and spatial resolution.

Also, Ngan et al. [30] made a setup which does not involve camera movement by placing the planar slices of the material in a form of the pyramid. Thus, such setup captures the material appearance for 13 different camera views at once. Light directions are sampled by manually-moved electronic flash. The disadvantage of such setup is that it provides sparse angular resolution, but depending on the application such approach can be plausible. For example, a material which does not posses high illumination changes can be sparse sampled, thus the reduction of redundant data is achieved.

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.

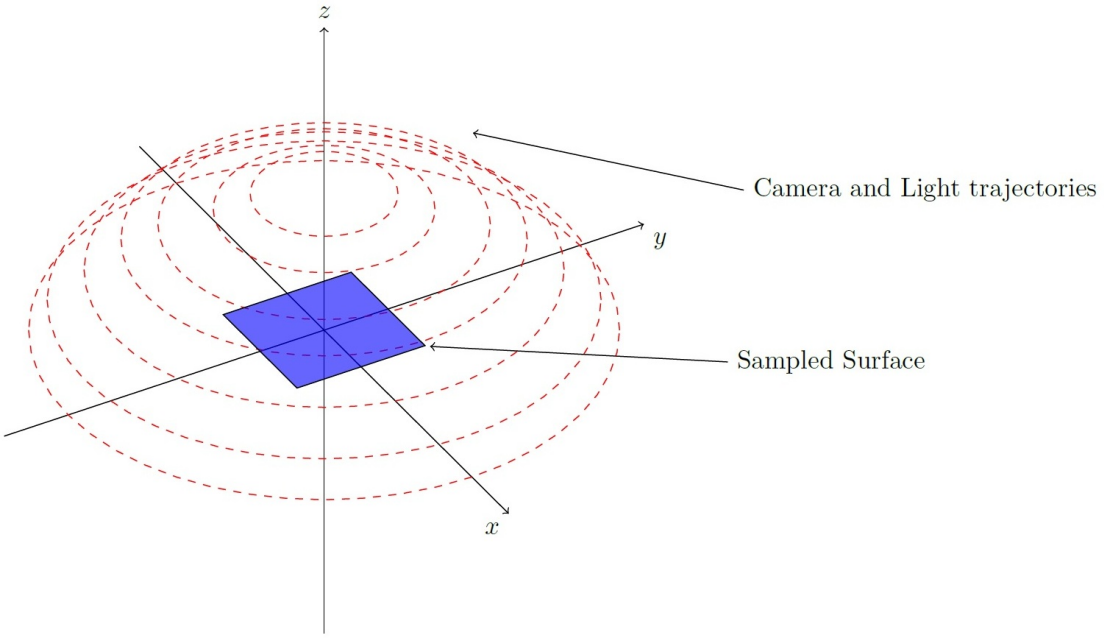

FIGURE 3.1: **Example of BTF measurement.**
Camera and light positions share the same trajectories. Red dashed circles are the sample positions on the hemisphere.

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 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 [35]. 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 [17]. 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 align. Also, registration errors can 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 [35]. 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 is highly depended on the quality of the acquired data, especially for BTF. 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.* [3], who measured 61 materials with fixed light and 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 [3]. 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

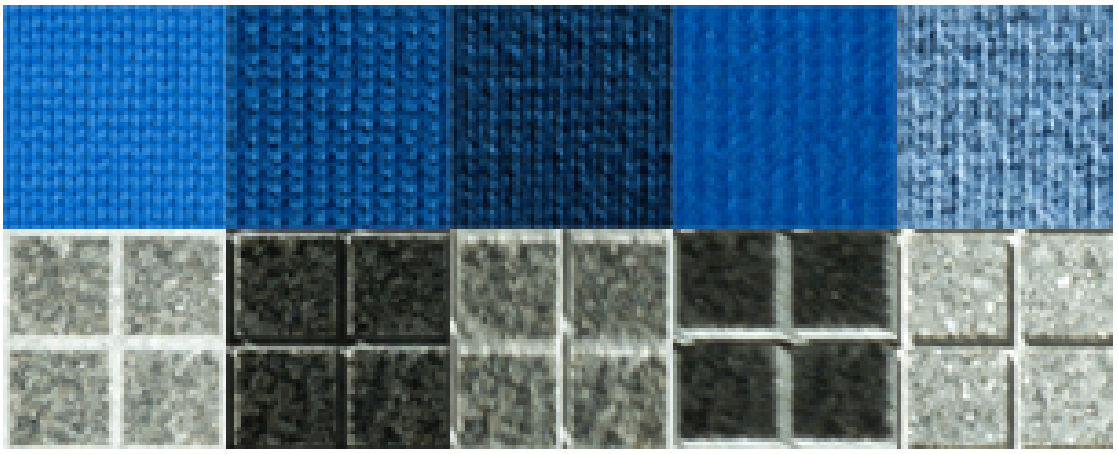

FIGURE 3.2: **BTF example of Bonn Database [2].**
Example how BTF catches rich appearance of the material due to dependencies on light and camera directions. Upper row is a knitted wool, lower is a graved granite stone.

rectified, but the authors provide image coordinates to enable rectification. Because of this limitations such BTF dataset are usually used for computer vision purposes, i.e. texture classification [17].

Based on this BTF measurement system, Bonn University created their own measuring system [35]. 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 [2] are publicly available and are used in this thesis.

Consider Figure 3.2, which illustrates one of the sampled 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 $81 \times 81 = 6561$ total 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 BTF rendering and 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 representation is a set of ABRDFs, one ABRDF for each sample position of the material plane. ABRDF was defined in Chapter 2.6. 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. 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 [19], 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 [2] a BTF consists of 8-bit PNG images with a resolution of 256×256 sampled for 81×81 different camera and light directions. Uncompressed BTF occupies 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 utmost importance. For our scenario it is important to chose a compression method, that one

that allows for real-time decompression, high compression rate 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 and at some point it is important to have an option to render the preview of BTF based on partial data.

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. Such functions few parameters, thus real-time performance is easily achieved. However, only these groups of methods can suffer from a decreased rendering quality [19]. 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 [37]. (See Chapter 3.3.2.)
- *Probabilistic models* can extremely compress the original BTF. However, the resulted image quality usually suits for flat surfaces. Also, there are problems of implementing these models on the GPU for real-time rendering. (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 allows for a compact representation of BRDFs, which in general can also applied to ABRDFs. Each texel of a BTF, i.e. spatial position on the surface are represented as 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.* [27] used this approach which allows for high compression rates and generally 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 [19].

Haindl *et. al.* [19] applied PTM on reflectance fields (RF). General formula looks the following way (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.* [27] claim that this method produce considerable errors for high grazing angles.

Another model which produces slightly better visual quality is the polynomial extension of the one-lobe Lafortune model (PLM) [19].

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

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 [15]. Filip and Haindl [15] 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 [19].

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) [9, 36, 19, 37, 35, 28, 31]. PCA is a linear transformation of the data to a new basis, where each new coordinate (variable) is called a *principal component* [9]. 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 are used for the better decompression error, and vise-verse, a smaller number of components gives better compression ratio, but worse decompression error. This allows for an easy regulation if 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% [36, 19]. Thirdly, PCA is suitable for real-time decompression on average GPUs [36, 19]. 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) [35, 36, 28, 19]. Sattler *et. al.* [35] 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 error around 2%- 4%. On the other hand, PCA BTF method, which does PCA on all camera directions at once, requires on average 41 components [19]. Müller *et. al.* [36] improved PCA BTF by exploiting vector quantization techniques and applying PCA on the resulted clusters. Local PCA (LPCA) requires 19 components on average [19]. Even though 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 [18, 23, 19] provide much higher compression rate than most others methods and allow for a seamless enlargement of textures of any size [19]. Usual compression compression ratio achieved by such models are $1 : 6 - 8 \times 10^4$. The visual quality of the rendered results are best suited for smooth materials, while materials with high frequencies can appear flat using probabilistic methods [19]. 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 [23, 19]. Even though solutions exists to handle this problem, e.g. take advantage of Framebuffer Objects, which allow access to previously rendered results [19], solution are time-consuming for low-end GPUs.

A common algorithm that employs probabilistic model combines a material range map and synthetic smooth texture produced by a probabilistic model. Range map is a monochrome image that stores the depth of each spatial position of the material surface [23]. If the synthetic texture is larger than range map, it can be enlarged using the Roller technique[20]. Currently there are available several probabilistic models, i.e. Gaussian Markov Random Field (GMRF) and Causal Autoregressive (CAR) models [19, 9]. Commonly these models are 3D models which further factorized and modelled by less dimensional models, i.e. by a set of 2D models. 2D models requires less parameters to store. Such models are very complex, because they can store up to thousand 2D models [17]. The learning process of such models is not easy as well, because it requires large training data set. Thus, probabilistic models can be applied to some simpler materials, e.g. smooth materials. Materials with high frequencies can result in a flat look, as shown by Haindl *et. al.* [19]. Also, the set of 2D models has to be trained simultaneously, thus it demands high computational effort.

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 belong to statistical methods, see Chapter 3.3.2. PCA compression method allows for easy regulation of trade-off between rendering quality and compression ratio. While probabilistic methods have better compression ratio, however they are generally suited for flat materials [19]. Analytical methods can produce errors at gazing angles and generally have bigger decompression error than PCA [19]. Also, the PCA is suitable for the streaming very well, because the components are sorted in the order of the importance. This means there is no need to do additional analysis to decide in which order to send the components, as it was done by Schwartz *et. al.* [37]. Based on the above reasons, we chose PCA over other methods.

4.1 Derivation of PCA

PCA can be defined through a maximum variance formulation, as done by Bishop [9]. 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:

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

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

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

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, 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 derivative will be the following equation [9]:

$$S u_1 = \lambda_1 u_1$$

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 rest 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 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 [43]:

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

where U and V are orthogonal matrices and S is a diagonal matrix consisting only of singular values. The diagonal values of Σ are the square roots of the eigenvalues of $X_c X_c^T$ [26].

$$X_c X_c^T = (U\Sigma V^T)(U\Sigma V^T)^T$$
$$X_c X_c^T = (U\Sigma V^T)(V\Sigma U)$$
$$V^T V = I$$
$$X_c X_c^T = U\Sigma^2 U^T$$

From the eigen decomposition theorem [42] 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.* [31, Ch. 15]. We apply PCA per k neighbour camera directions. If $k = 1$ proposed method becomes equal to the PCA RF method [19]. If we take the whole number of camera directions it becomes PCA BTF method [19]. 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 $i = 0..(3 * N)$. 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 $(W * H) \times (3 * N)$, which consist of columns a_i. Rows of the matrix A are ABRDF representations of BTFs, 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 $W \times H$ and Σ is a diagonal matrix and holds the "importance" value of each principal component, and matrix V stores weights that are needed for reconstruction of B.

4.3.2 Decompression

The decompression only involves matrix operations, which combine 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.* [31, Ch. 15]

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

The matrix A 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 using barycentric weights [17].

The algorithm involves finding closest directions and then applying 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 [19].

The Bonn database [2] provides data with uniform sampling per longitude. Depending on the longitude position, different quantization step is applied for latitude. For areas closer to the bottom of the hemisphere quantisation step for latitude gets increased. This is done to have relatively equal distance between the directions for any longitude position.

Assume, a set of quantisation steps $S = \{(\Delta\theta * n, \Delta\phi_n) \mid n \in M \subseteq \mathbb{N}\}$, where M specifies the number of steps. For $\theta = 0^\circ$ only one image is taken, i.e for $(0^\circ, 0^\circ)$ 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 $n = \frac{\theta_L}{\Delta\theta}$. And, 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 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 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 belong to ABC or CBD.

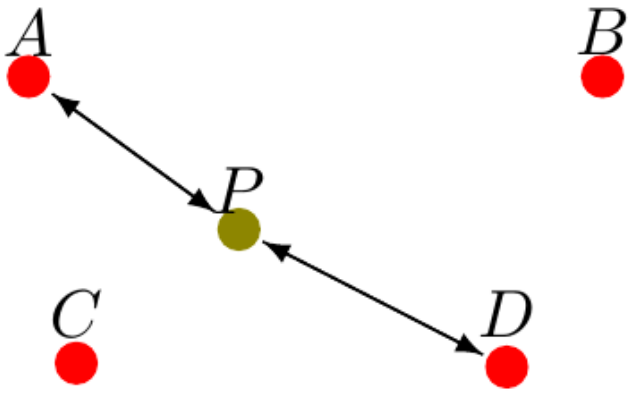

FIGURE 4.1: **Closest Directions**

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

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

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, because there is negative sign in the formula d. 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 measured directions, i.e. $A = (\theta_L, \phi_L^1)$ and $B = (\theta_L, \phi_U^1)$ 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 [22] 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

$C_P = w_1 C_{P1} + w_2 C_{P2} + w_1 C_{P2}$, where C_{P1}, C_{P2}, C_{P3} correspond to color values of the found triangle $P_1 P_2 P_3$. Weights w_1, w_2, w_3 are normalized and sum up to 1.

Weights w_1, w_2, w_3 defined as volumes V_1, V_2, V_3 which correspond to PP_2P_3O, PP_3P_1O, PP_1P_2O tetrahedrons, where $O = (0, 0, 0)$. Volumes calculated as determinates of 4×4 matrices

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

Last step is the normalization of found volumes, i.e. $V_i = \frac{V_i}{\sum_{i=1}^{3} V_i}$.

4.4.3 Algorithm

Computing interpolation weights for input direction P, it is required to find the closest measured directions $P_1P_2P_3$ to P. After these three directions are known for the input direction P, interpolation weights can be computed. Chapter 4.4.2 explains how to compute barycentric coordinates, i.e interpolation weights.

To compute the final interpolated color we combine interpolation weights of light and camera directions. Assume that I and O are input light and camera directions. Then, bounding triangles for both of them are $I_1I_2I_3$ and $O_1O_2O_3$ accordingly. Corresponding barycentric weights then are $b_i = [b_{i1}, b_{i2}, b_{i3}]^T$ and $b_o = [b_{o1}, b_{o2}, b_{o3}]^T$. The final color is the linear combination of b_i, b_o weights and known 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 for a light direction I_u and a camera direction O_v.

Chapter 5

Streaming

When rendering BTF in a web-browser, all the data has to be transferred before the rendering. Even the compressed BTF data can be tens of megabytes in size take considerable amount of time for full transmission. We propose to stream the BTF data to reduce the delay before rendering can start, by implementing a streaming technique. The user will be able to see a low quality preview of the original BTF just in a few seconds. In our case, principal components are streamed one by one using WebSockets technology [41]. Each principal component cover full angular domain, so the BTF can be rendered for any camera and light direction at certain point of the streaming process. The rendering quality is enhanced whenever a new component arrives. Also, we provide the information about the overall progress of the stream to give an instant feedback to the user.

5.0.4 WebSockets

WebSockets technology is an efficient and an elegant way of communicating between the server and the client. WebSockets is a brand new technology and already supports natively by web-browsers, including mobile devices.

This technology has several advantages over HTTP Polling, Long Polling, Streaming approaches. Main properties of WebSockets are [41, Ch. 1]:

- **Single connection** using WebSockets allow the client to reuse the same connection with the server. Whenever the new data becomes available the server sends a message to the client. Unlike Polling method, which makes requests at regular intervals to the server to check whether the new information is available. This

single connection reduces the latency. HTTP Streaming method also keeps single connection as WebSockets, but the drawback of HTTP Streaming is that it never signals if the HTTP response is finished. This makes harder for the client to understand if the data transmission is finished. Also, the client's firewall may buffer the HTTP response, which can result in increased latency. WebSockets save bandwidth, CPU power, and latency compared to HTTP Streaming, Polling methods.

- **Native binary data** transmission support. For instance, using binary format can decrease the size of the data compared to Base64 representation. This may reduce the load on the network by transferring less data.
- **Messaging** support after the connection is established. When the connection is open, the server and the client can send messages to each other without additional overhead. So, for example, the client can message the server that he needs a new principal component of the BTF.
- **The dedicated server** for WebSockets may perform better than traditional HTTP server. Traditional servers designed for request/response cycle and may not handle properly many connections at the same [4]. On the other hand, WebSockets server can be developed in a efficient way to handle many clients at the same time and providing so called non-blocking IO. For instance, NodeJs server side implementation [6].

In summary, WebSockets is quite new and innovative technology. It provides a big step forward in developing real-time web-applications. WebSockets can yield reduction in HTTP header traffic and reduction in latency compared to HTTP Polling, Streaming methods [41, Ch. 1]. Overall, it is a promising technology for web-applications.

5.0.5 Transmission

As it was mentioned above, for WebSockets technology the dedicated streaming server is required. It means that the implementation code is required for the server part, in order it can properly communicate with clients and send them the requested data. For example, if the client requests certain BTF data, then the server should send principal components one by one. The client can message the server if the component is already transferred and the server can send the next one. Also, the streaming server can store all additional parameters needed for the BTF decompression.

In Section 4.3 matrices U, Σ and V store all the data needed to render the BTF. Matrices Σ and V are small in size and they are sent in the first place. On the contrary,

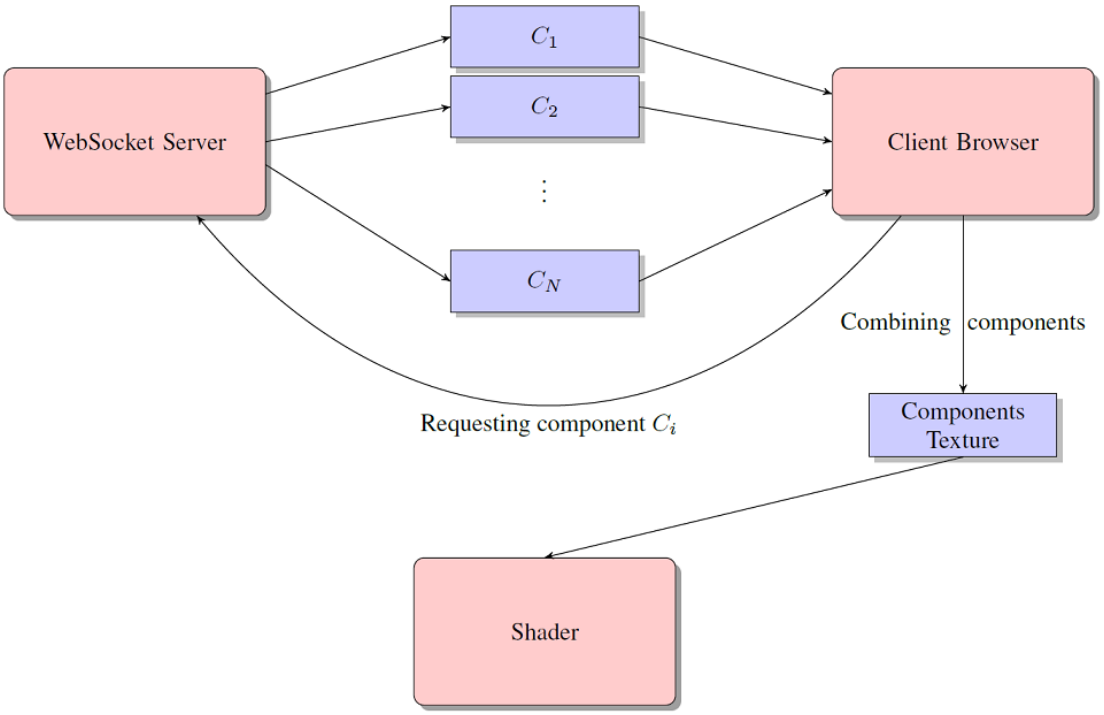

FIGURE 5.1: **Streaming process illustration**

most of the information is stored in the matrix U, which stores principal components. On the client side before streaming, matrix U is initialized with blank values, for example zeros. After the Socket connection is established, the client messages the server to send one component at a time. Each component on the WebSockets server is stored as a PNG image, which provides further compression for the BTF. When a new component fully transferred to the client, it saved to the matrix U and the rendering of the BTF is refreshed. With each component the quality of the resulting image is progressively enhanced. Now, with streaming the client is able to see rendered image just in a few seconds. All matrices on the client are sent to the shader as PNG images.

The Figure 5.1 illustrates the described streaming process. Components Texture in the Figure 5.1 is the matrix U, which saves the transferred principal components.

Chapter 6

Implementation

This chapter describes details, problems and limitations of the thesis implementation part.

The implementation of the demo-application 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 [2]. It is possible to adapt any other BTF databases for our demo-application by resampling BTFs [16]. For streaming we use Node.js [6] platform. To integrate 3D graphics into HTML page we use XML3D [39]. The overall model of the demo application is depicted in Figure 6.1.

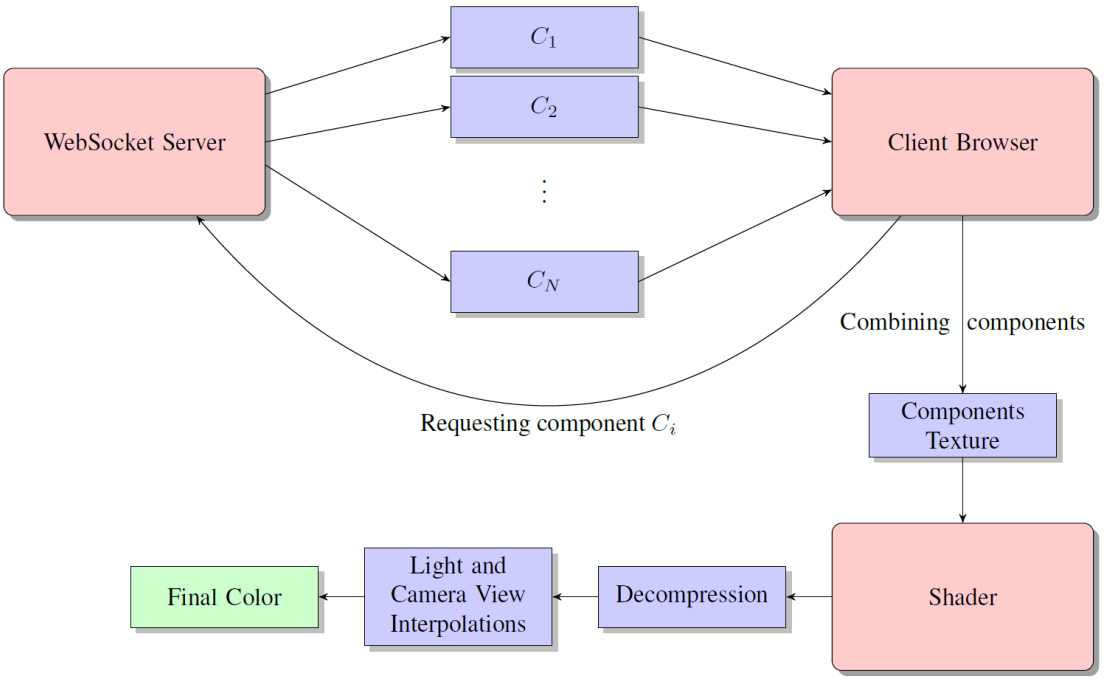

FIGURE 6.1: **Model Overview**

6.1 Compression

The implementation of the singular value decomposition (SVD) is the main part of the PCA implementation. We use *jblas* [5] library developed by Mikio Braun. This is a linear algebra library written in Java language, which provides very high performance [5].

The compressed BTFs have to be sent to the shader. OpenGL Shading Language (GLSL) version 1.0 support uniform arrays, however, they do not fully support dynamic indexing [7]. Fortunately, it is possible to send arrays of data which are mapped to textures. After performing SVD, resulted matrices U, Σ, V are saved as PNG images. (See Chapter 7.2). Values of matrices U and V are in the range of $[-1; 1]$. Those values have to be mapped to the image domain, i.e. $[0; 1]$. The following function is used to map the values:

$$f(x) = (x + 1)/2$$

Each component of the matrix U is stored as the PNG image. For example, if compressed BTF data has C principal components, then this would result in C separate images. This is done for streaming purpose, so then principal components can be sent one by one from the server to the client. We use RGBA color space to save four components into the one pixel, because it is possible to do efficient multiplication in the shader by using vector multiplications. Matrices V and Σ are saved in the single 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 [2] are not bigger than 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 the 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 the similar manner.

We noticed that in case of Bonn Database [2] the *jblas* [5] library produces relatively sparse values for U and V, i.e. close to the zero. We scale the data to improve the floating point error caused by mapping of U and V back-and-forth. The scaling improved the overall decompression error approximately by 5%. Using the following function we find the scaling factor:

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

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 each time Σ values by the factor $\frac{1}{factor(M)}$ if we scale either U or V.

6.2 Streaming

We use Node.js [6] as a server side platform, which enables realtime streaming using WebSockets. On top of that, we use BinaryJS [1] for the binary streaming. BinaryJS is a framework that uses WebSockets to stream the binary data to the client from the Node.js server. The server and the client applications are written in JavaScript.

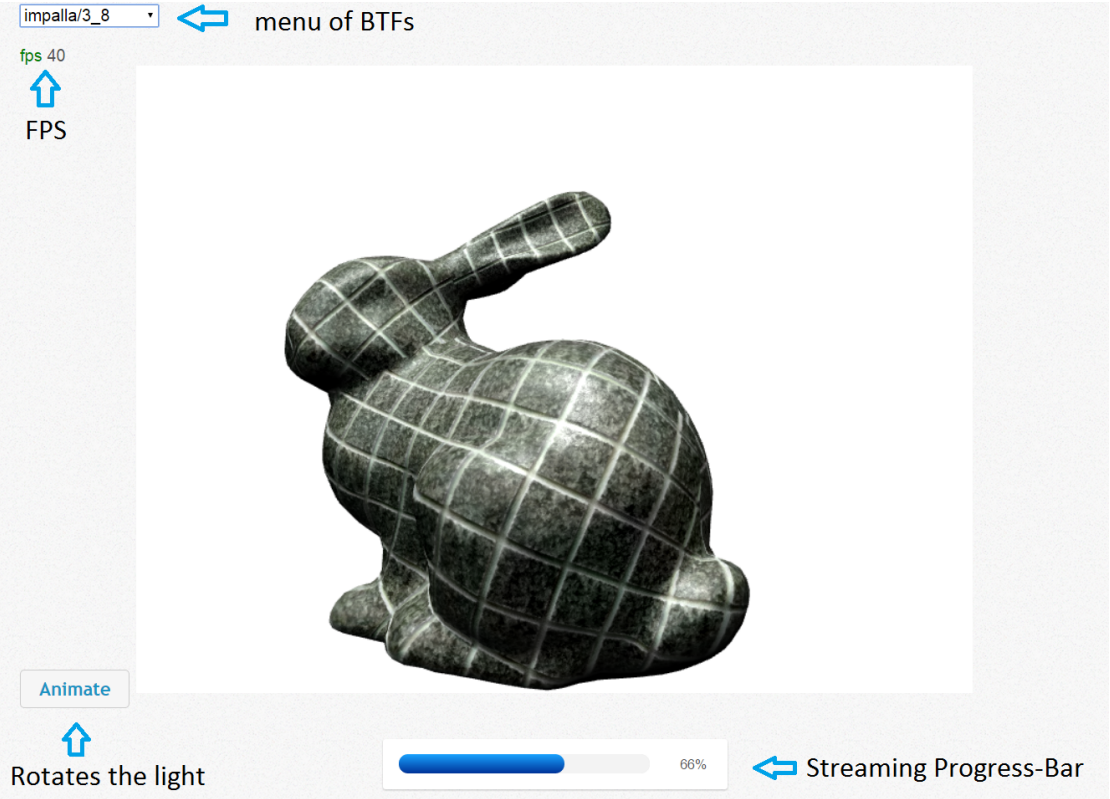

FIGURE 6.2: **The streaming progress on the client side**

We also use Xflow [24], which is a declarative language for data flow processing in realtime. Xflow is a part of XML3D implementation. We use Xflow for gathering all the transferred principal components and storing them in one common texture L. (See Chapter 4.3.2).

Consider Figure 5.1 from the Chapter 5 that depicts how the streaming works on practice. When the user connects to the streaming server and the HTML page loads completely, i.e. all the 3D objects are 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 texture R and meta data. (See Chapter 4.3.2). Afterwards, each principal component C_i are streamed in chunks one by one as PNG images. Each C_i covers all the angular domain, i.e. all the possible camera and light directions.

We decode PNG images using PNG decoder written by Arian Stolwijk [8]. PNG images are decoded to the array of RGB colors and saved to the common texture L in canvas format using Xflow. Each time the texture L updates the rendering also refreshes. Even with the first principal component the resulted image looks plausible. With further components the overall quality of the image improves, i.e. specularities are increasing, small micro-structures become more visible and emphasized. The user also able to see the progress-bar of the streaming progress as shown in Figure 6.2.

Currently the limitation of such streaming approach is the drop of the frame rate when the assembling of the principal components occurs on the client side. This is caused by the update of the canvas element (texture L) each time new component is transmitted to the client. Also, we did not test the performance of multiple streaming, i.e. if there are several 3D objects that request BTFs at the same time. Our current implementation does not support this. This can be included to the future work.

6.3 Rendering

We use XML3D [39] platform to embed 3D graphics into the HTML page. XML3D based on XML and allows for declaring your own 3D scenes and shaders. XML3D is based on WebGL and JavaScript.

The shader design is depicted in Figure 6.3. The compressed BTF data comes to the shader in the form of two textures. The first texture L stores principal components and the second R stores PCA weights. (See Chapter 4.3.2). First, we find 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.

In the next step we sample the input textures to decompress the needed values for found closest 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

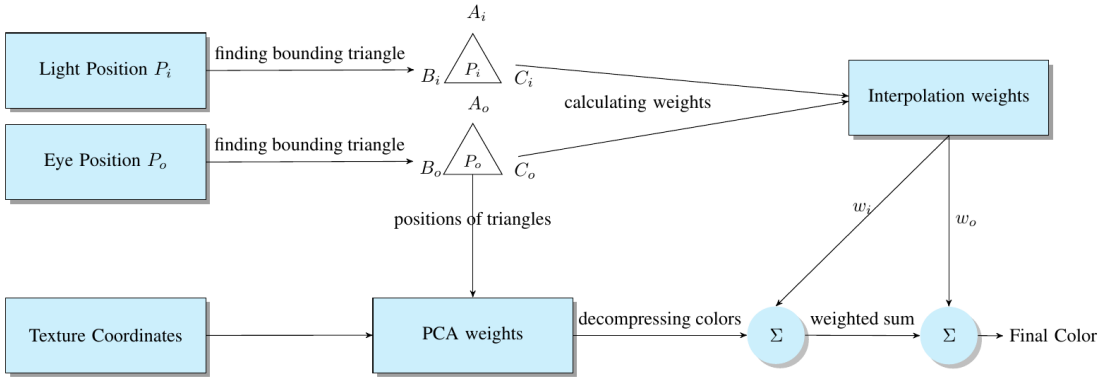

FIGURE 6.3: **Shader Design**

be possible to start the decompressing. All the principal components are written linearly in texture L, i.e. one by one. 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 size 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, we lookup the index of suitable compressed image. 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 early found interpolation weights to get the final color. (See Chapter 4.4.3).

The implemented shader has it's own limitations. First of all, the number of principal components has to be fixed for a shader, because GLSL version 1.0 does not allow for dynamic looping [7]. Secondly, the number of principal components are bound by a size of the texture L. It means that not all GPUs can handle very big textures. The problem can be fixed by using multiple textures. But anyway these limitations does not directly influence the performance of the shader, which provides real-time frame rate and performs well even on the mobile devices. (See Figure 6.2).

Chapter 7

Evaluation

In this chapter we evaluate the decompression error, compression ratio and image quality during the streaming.

7.1 Compression

We evaluate our compression approach on the Bonn's BTF Database [2] and compare the results to other related PCA methods [19]. We tested different configurations for BTF's compression and decided 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 7.1 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 [33]. MAE is calculated for each of the channels separately and then the final result is averaged among them.

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

Figure 7.1 shows that the decompression error is not improved enormously for 16 or even for 32 components. That is why the reasonable choice is 8 components, while for 4 components the image is getting blurred. Haindl [19] also claims that 8 components is the optimal number of components for the PCA RF method, which is related to our method.

Table 7.1 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 Root Mean Square Error to measure the variance in the errors

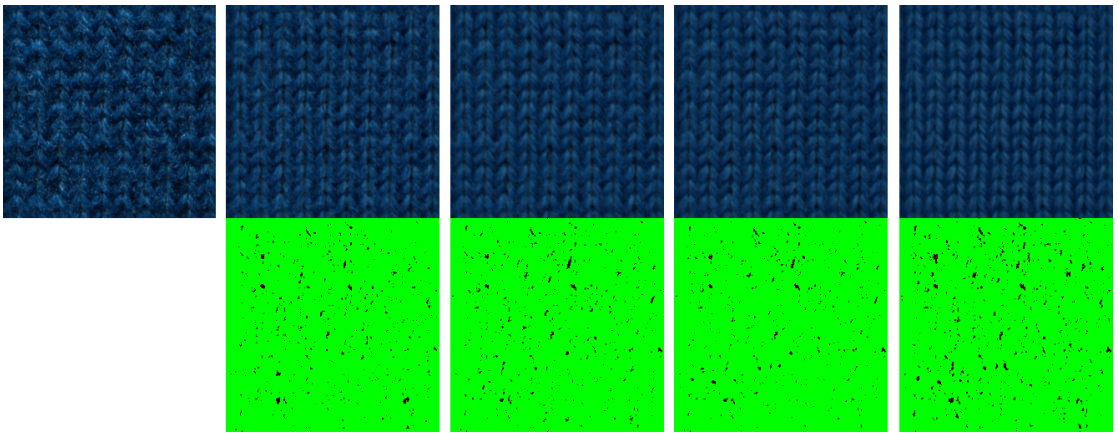

FIGURE 7.1: **Example of decompression errors**
First row from the left to the right: original image, **32** components (MAE:2.17), **16** components (MAE:1.83), **8** components (MAE:2.21), **4** components (MAE:3.04). **Second row**: the difference between the original and the decompressed images, *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 7.1: 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 [10]. 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 7.1 where $k = 1$ our method becomes equal to the PCA RF [19] method. Haindl [19] claims to have the MAE equal to 3.16 in CIELAB color space for the same sample. Our method produces a bit better result, i.e. 2.14. The second row shows that for $k = 3$ MAE stays practically the same. However, the compression ratio improves in three times! We can see that even for 32 components the decompression errors improves only a bit, but the compression ratio becomes worse. If we use k bigger than 3, then we would need to use more components to have good decompression error. For intance, our method becomes equal to the PCA BTF [19] method when $k = 81$ (all camera directions). In this case we would need approximately 41 components [19], which is the intensive computational task for the GPU.

Also, the LPCA BTF [19] method has the MAE equal to 2.42, which is a bit higher than ours method. The LPCA BTF uses 19 components, while our method uses only 8.

7.2 Real-time performance

We evaluate the rendering quality during the streaming and the real-time performance. Our demo viewer is available at http://btf.mywebcommunity.org. Figure 7.2 depicts the 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 $C = 8$.

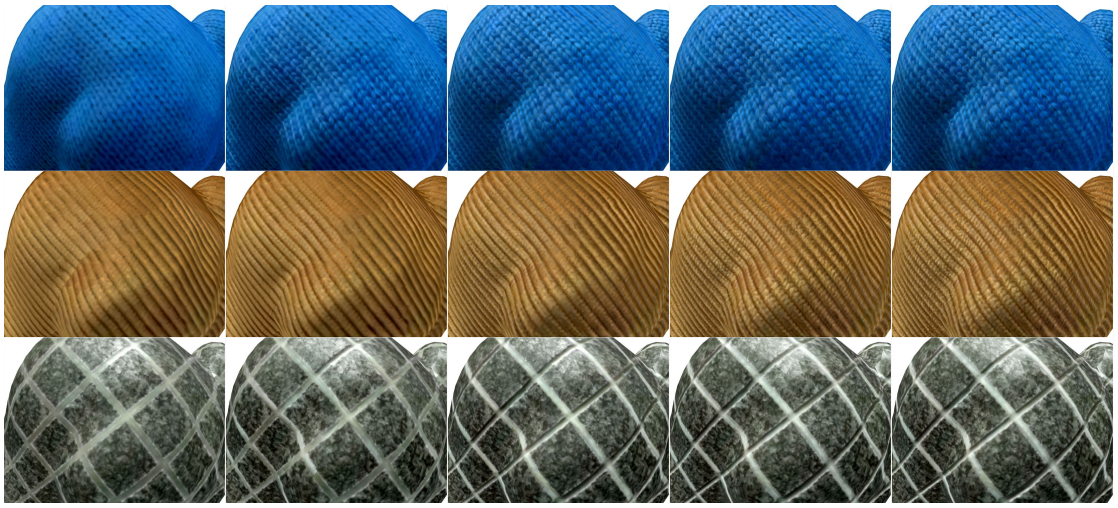

FIGURE 7.2: **Example of Progressive Streaming**
From left to right: *1*, *2*, *4*, *6*, *8* components rendered at the same accordingly.

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 the component, which has the size on average about 0.7Mb. With further components the overall quality of the image improves, i.e. specularities are increasing, small micro-structures become more visible and emphasized. After all components are transmitted, the demo-viewer hits the maximum available frame-rate of the monitor, i.e. 60Hz or even 144Hz. The tests were done on the NVIDIA GeForce GTX 480 videocard. Also, the demo-viewer works smoothly on other average GPUs.

We performed a test of the demo-viewer on the mobile device Sony Xperia SL. The real time frame rate was approximately 10-15 FPS (frames per second), which is relatively good result. With newer mobile devices it possible to achieve even better results.

7.3 Comparison with the Phong Shader

Figure 7.3 shows the tremendous difference between the BTF shader and 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, while the BTF makes the material look very bumpy and shiny. The BTF shader shows the correct inner-reflections, sub-surface scattering and shadows under varying light conditions.

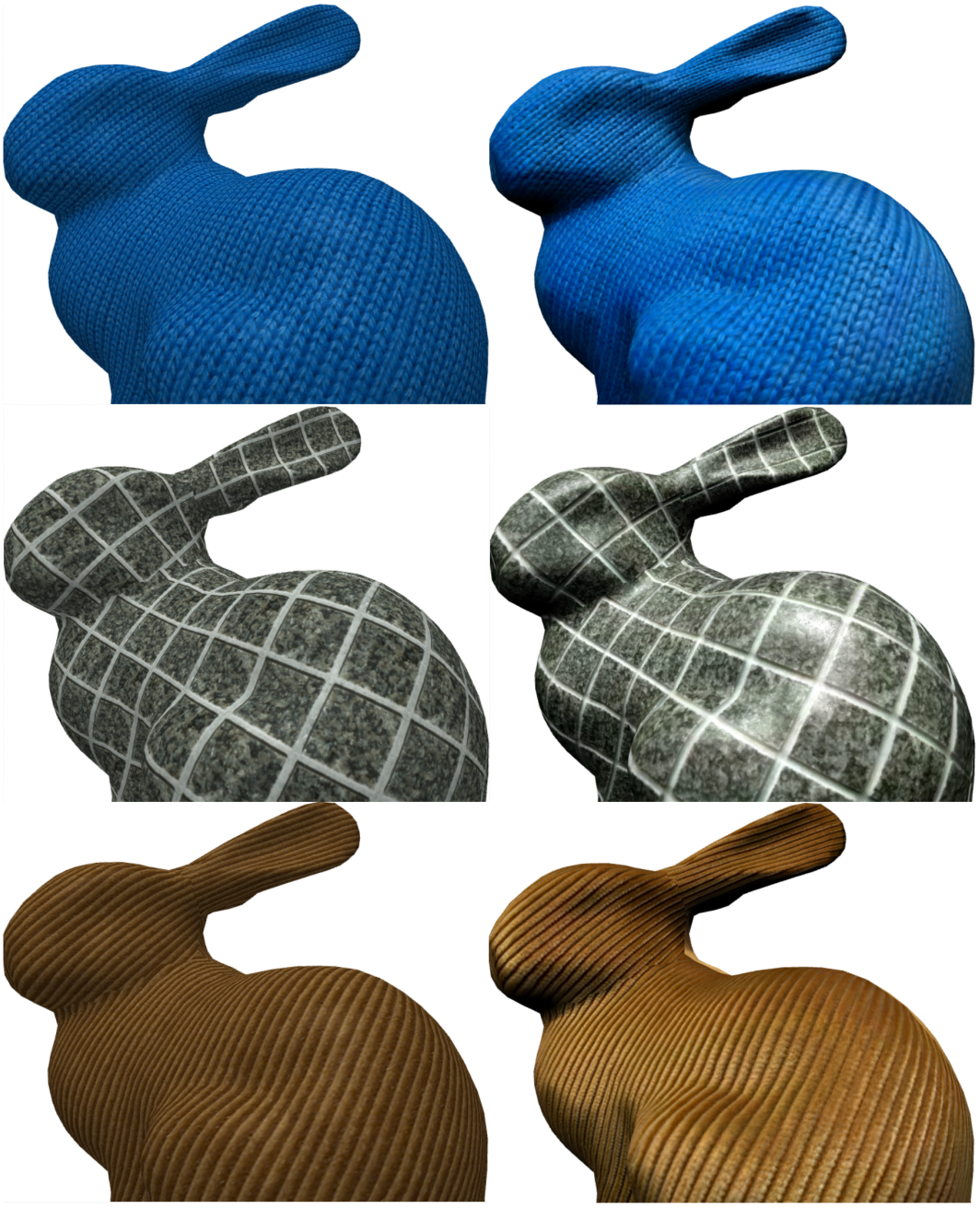

FIGURE 7.3: **The Phong shader vs The BTF shader**

Chapter 8

Conclusions and Future Work

8.1 Summary

8.2 Future work

Bibliography

[1] Binaryjs library for realtime binary streaming. http://binaryjs.com/, note = Accessed on January 2015.

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

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

[4] Introducing websockets: Bringing sockets to the web. http://www.html5rocks.com/en/tutorials/websockets/basics/, note = Accessed on January 2015.

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

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

[7] 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.

[8] Pure javascript png decoder. https://github.com/arian/pngjs. Accessed on January 2015.

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

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

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

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

[37] 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.

[38] 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.

[39] 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.

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

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

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

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

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