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. Kristian Sons

...

begutachtet von / reviewers

...

...

Saarbrücken, July 2014

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, July 2014 Oleksandr Sotnychenko

Abstract

Bidirectional Texture Function (BTF) is a texture that depends on light and camera views, hence BTF can be considered as a 6-D texture. The main advantage of BTF is that it can represent complex materials realistically plausible, e.g. depicting detailed surface structure with strong varying illumination. However, with such benefits comes a main drawback of BTF: big size of a data. Thus, we need to deploy good compression representation and at the same time we have to remain decent quality of BTF. Also, decompression performance of the data must be optimal in pursuance of overall solid performance of the program.

In this work we present a solution of a problem rendering BTF in a web-browser. Besides, we have one more subproblem to solve, i.e. as the compressed BTF data transmits from the server to the user it can take some minutes to wait until the data fully arrives. In order to shorten the waiting time, we employ streaming of the data using web-sockets. In fact, the user will able to see the BTF rendering progressively, while the quality of the texture will be improving during streaming. On top of that, even in a few seconds it could be possible to see the preview of the rendering object in a lower quality, while the remaining parts of BTF will be progressively streaming to the user.

Acknowledgements

I would like to express my sincere gratitude ...

To my family.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.2 Related Work

1.3 Outline

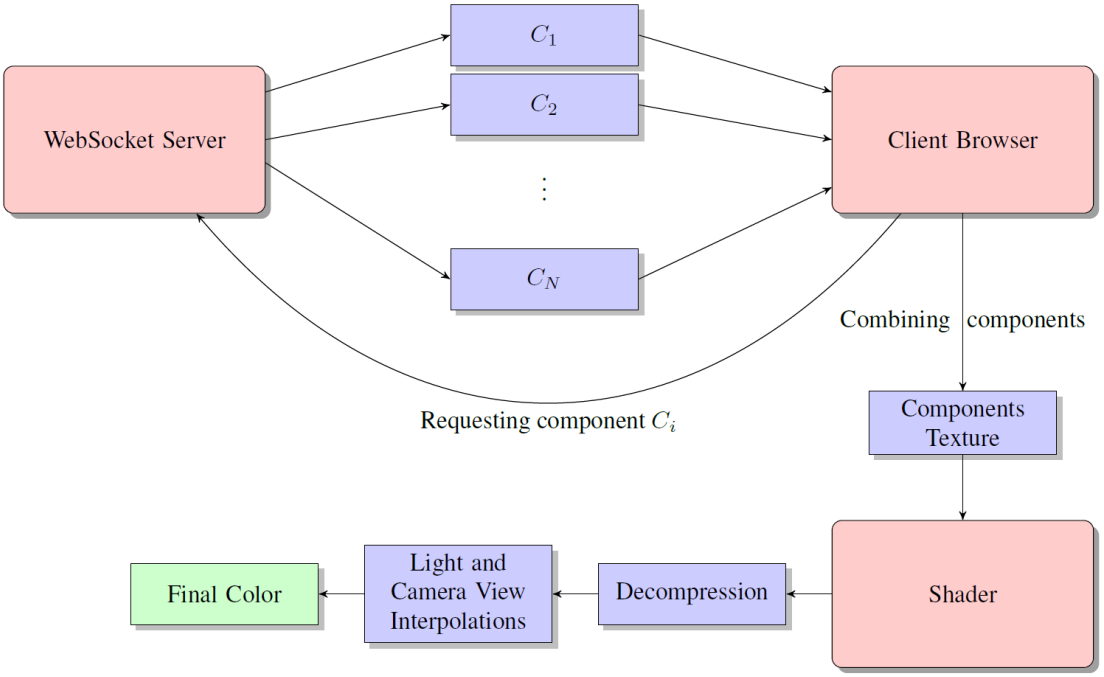

FIGURE 1.1: **Model Overview**

Chapter 2

Methods

This Chapter has the following structure. Section 2.1 describes methods of BTF acquisition and reviews some of the publicly available BTF datasets. Section 2.2 argues about the choice of compression methods, while Section 2.2.1 defines our chosen compression method. Section 2.3 defines a solution for viewing and illumination angle interpolation. The last Section 2.4 concludes the Chapter with proposing our web-socket streaming model.

2.1 BTF Acquisition

BTF acquisition is not a trivial task, since to get an accurate and reliable BTF data, it is required to have an efficient BTF measurement system. A pioneer in this task were Dana et. al. [3], who measured 61 materials with fixed light and moving around camera that photographed a flat surface of a material. Such procedure results in a set of images, i.e. BTF, which can be regarded as a 6D reflectance field function [14]

$$L = L(x, y, \theta_i, \phi_i, \theta_o, \phi_o)$$

where (x, y) is a surface point of a flat sampled material, (θ_i, ϕ_i) incoming light direction (light direction) and (θ_o, ϕ_o) outgoing light direction (camera direction).

For each measured surface Dana et. al. used 205 different combinations of viewing and illumination directions. The resulted BTF can be gigabytes in size. To achive reasonable real-time performance compression of BTF is an inevitable step. Also, Dana's et. al. BTF database is not spatially registered, i.e. images were not mutually registered for different view angles. That means one should further process the database in order to render BTF.

Based on Dana et. al. BTF measurement system, Sattler from Bonn University made his own measuring system [14]. The main difference in that system is that a camera moves on a semi-circle rail around a material. Also, such setup provides spatially registered data, with reasonable angular and spatial resolutions. Datasets of Bonn University [2] are publicly available and were used in this thesis.

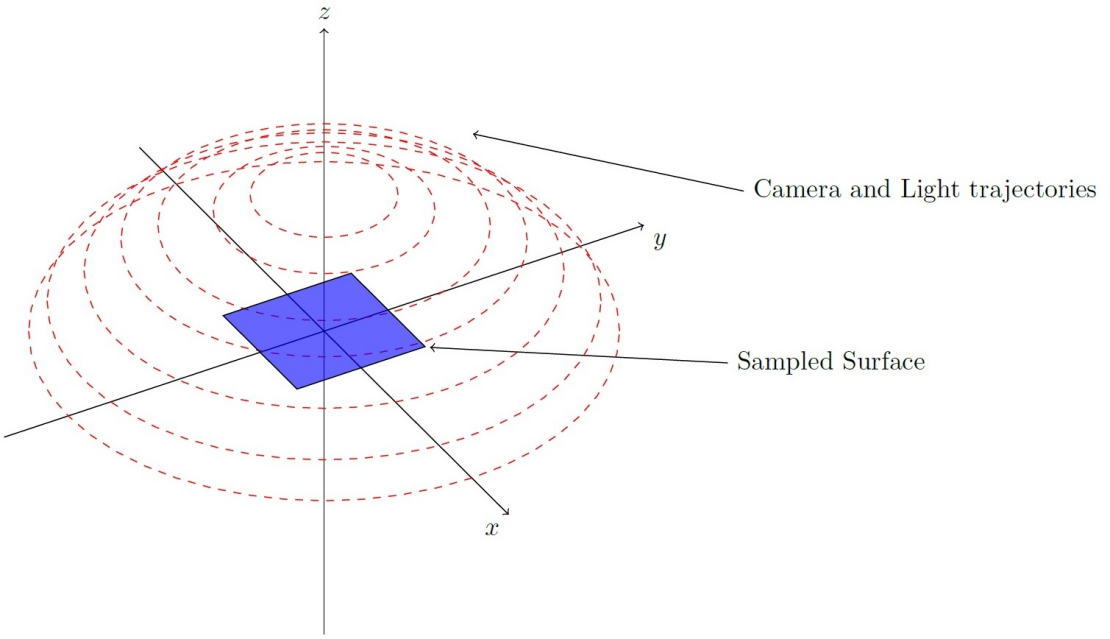

FIGURE 2.1: **Example of BTF measurement.**
Camera and light positions share the same trajectories. Red dashed circles are the sample positions on the hemisphere.

Consider Figure 2.1, which illustrates one of the possible ways of sampling the material. The measured surface is being fixed all the time on the sampler holder. Then, for each light position, a camera photographs the flat material while moving from point to point of the hemisphere. Bonn database has the same trajectory for camera and light positions, i.e. 81 positions on the hemisphere, which results in 6561 total number of acquired images.

2.2 Compression and Decompression

As we already know BTF is a high dimensional data, which makes it enormous in size. Datasets of Bonn are 8-bit PNG images with 256×256 resolution. Then, after an uncompressed PNG data loads to GPU it occupies approx. 1,2 GB of space. And that's not even large a spatial resolution in this case! Therefore, it is impractical to render BTF as a raw data even on the most modern GPUs.

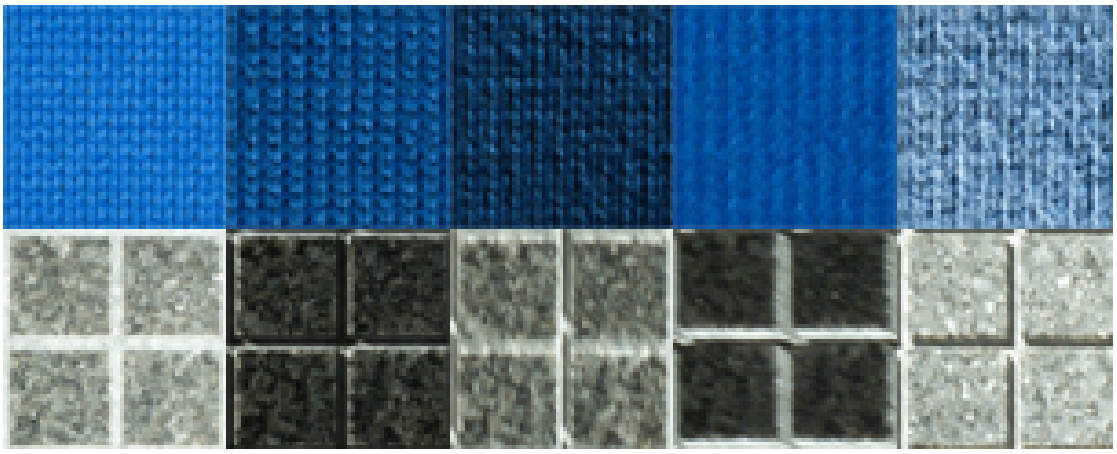

FIGURE 2.2: **BTF example of Bonn Database [2].**
Example how BTF catches rich appearance of the material due to dependencies on light and camera views. Upper row is a knitted wool, lower is a graved granite stone.

The most common technique to compress BTF is Principal component analysis (PCA), which is widely used in image compression [7, 12, Ch. 15]. PCA were used in several works of Bonn University [14, 10, 11]. Also, at some point PCA were employed as a basis for Decorrelated Full Matrix Factorization compression method [15].

The reason that PCA is the broadly used is because a user can straightforwardly change parameters of the model and control a trade between compression ratio and a reconstructed image quality. Unlikely, other methods based on probabilistic MRF models [7], even though probabilistic methods provide thousand times better compression rate than linear factorization methods. Also, one more advantage of PCA is that is more suitable for low-end GPUs, because for example MRF methods can be very heavy for GPU.

2.2.1 PCA

By a common definition Principal component analysis (PCA) is a mathematical tool that reduces the dimensions of possibly correlated variables. New reduced dimensions are called *principal components*. Principal components hold the most important variations of the data, e.g. first ones hold the biggest amount of texture details. Using principal components it is possible to reconstruct the whole data. Due to neglecting small variations of the variables at first place, the reconstructed data will differ from original one, but the most important variations of the data will be preserved. That's why PCA is good for image compression, because a human eye is not that sensitive to small variations, i.e. reconstructed image might look almost the same as original for an observer. The good part about losing small variations, is that we can reduce some noise that was present in the original images [13].

2.2.2 Data representation

First step in our analysis is to chose a suitable data representation. Basically, there are two common arrangements, i.e. as a set of images or BRDFs (Bidirectional reflectance distribution function). Both representations can be mathematically interpreted as [10]

$$BTF_{BRDF} = \{P_{(x)}\}_{(x)\in I\subset\mathbb{N}^2}$$
$$BTF_{Image} = \{I_{(v,l)}\}_{(v,l)\in M}$$

where M denotes a set of images $I_{(v,l)}$ measured for different view and light directions (v, l) accordingly. BTF_{BRDF} denotes a set of $P_{(x)}$ images, where each of them depends on a single spatial position x for all possible views, i.e. $\forall(v, l) : x \in I_{(v,l)}$. Note that such BRDFs are not fulfilling physical demanded property such as reciprocity, because they already contain a factor between an incident light direction and a normal, i.e. $n * l$. In our case with Bonn datasets the sizes of variable vectors which we want to compress are $|I| = 3 \times 256 \times 256$ and $|M| = 3 \times 81 \times 81$, where 3 stands for RGB channels.

Müller et. al. [10, 7] claim that BRDF representation works slightly better in comparison to image-wise. The advantage of BRDF is that a compression ratio can be 10 times better and reconstruction error is slightly smaller than the image-wise representation. Also, Borshukov et. al. [12, Ch. 15] chose BRDF representation, claiming that it provides better compression ratio. The reason why BRDF provides better compression ratio is because it provides better pixel-to-pixel comparison than image-wise. Each variable vector of BRDF arrangement depends only on a surface complexity [10], i.e. on a variation of reflection properties at a spatial point of the surface. On the contrary, image-wise variable depends on the whole measured image plane, thus it is logical that it may provide lower compression rate due to stronger variations. After all, we have chosen BRDF data representation based on such reasons, with which we have achieved $1 : 100$ compression ratio.

2.2.3 Algorithm

The algorithm of PCA for BTF compression and decompression is mainly borrowed from Borshukov et. al [12, Ch. 15]. Basically, this is a common algorithm of PCA that deploys computing a singular value decomposition (SVD).

In the *first* step of algorithm we built BRDF 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 number of images. The size of a_i is $W \times H$, where W and H are dimensions of the image. So, matrix A has

the following dimensions $(W * H) \times (3 * N)$, which consist of such columns a_i. Rows of matrix A are BRDF representation of BTF, 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 mean vector from each column of matrix A

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

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

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

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

2.2.4 Compression

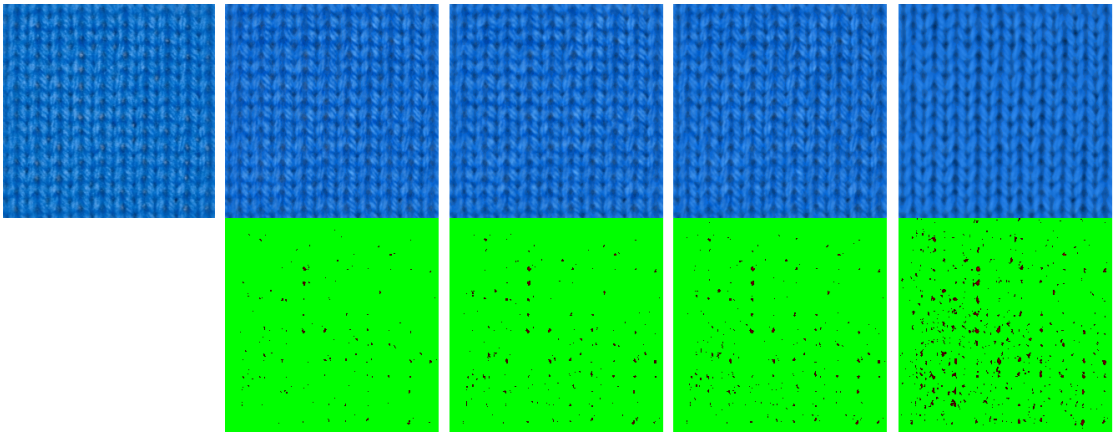

FIGURE 2.3: **Example of compression №1.**
First row from the left to the right: original image, **32** components (RMSE:7.7%), **16** components (RMSE:8.2%), **8** components (RMSE:8.5%), **4** components (RMSE:10%).
Second row: the difference between original and decompressed, *red* color denotes how big the error, *green* denotes that error is absent or very small.

In order to achieve good compression ratio and at the same time preserve a decompressed quality we need to chose suitable parameters for PCA. We perform PCA on subsets of 81 camera views. For each camera view there are 81 images sampled for different light directions. We take 3 neighbour views at once and perfrom PCA on them. With such size of subset we can achieve good trade between quality and compression ratio. For subsets of size 1 RMSE (Root mean square error) improves to 5%-7% on average. If we want to achieve even bigger compression ratio we can make the size of

the subset even bigger, but of course the quality of decompressed texture will be worse compared to smaller subset sizes. Also, performing PCA on the bigger subsets is inefficient, e.g. the time required to perform PCA on all 81 views at once can take hours and memory requirements to compute SVD are demanding.

Consider examples shown in Figure 2.3. We can see that for 4 components all images are getting blurred. In our case, 8 components is a reasonable choice, as even for 32 components the quality improvement is not that visible. Besides, smaller number of components will make decompression performance better.

The second example in Figure 2.4 has worse reconstructed quality than in Figure 2.3 due to small specularities on the surface, which are not preserved in principal components. This example demonstrates the limitation of PCA, i.e. such small details may not be reconstructed. But, an overall structure and a tone of the texture is preserved. Also, compression ration 1:100 is achieved with such parameters, i.e. from 1,2 GB we now store around 20 MB in GPU.

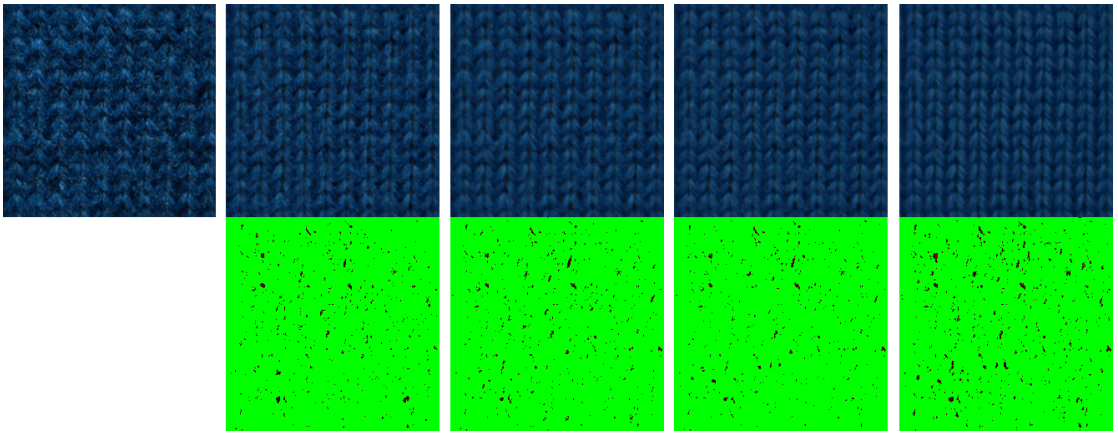

FIGURE 2.4: **Example of compression №2.**
First row from the left to the right: original image, **32** components (RMSE:8.3%), **16** components (RMSE:8.4%), **8** components (RMSE:8.2%), **4** components (RMSE:9.2%).
Second row: the difference between original and decompressed, *red* color denotes how big the error, *green* denotes that error is absent or very small.

2.2.5 Decompression

The decompression step is simply matrix operations, which require to combine 3 matrices U, Σ, V and mean vector m. To make it a bit easier to decompress on GPU we construct new matrices as Borshukov et. al.

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

Matrix A takes such form of decompression $A = LR$. In detail, if want to decompress texture with index i for first C components, we do it separately for each color channel

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

2.3 Viewing and Illumination Angle Interpolation

BTF datasets are measured for discrete number of angles, thus we would need to perform interpolation for unknown not measured angels. We will employ barycentric coordinates interpolation. However, it is very computational heavy, so the following approximation algorithm proposed by Hatka and Haindl [8] will be used.

Assume that we have found triangle $P_1P_2P_3$ which bounds our point P, which we want to interpolate. Figure 2.1 demonstrates hemisphere on which triangle $P_1P_2P_3$ lies. Y_P denotes desired pixel color. So, generally speaking linearly interpolation of that pixel will be $Y_P = w_1Y_{P1} + w_2Y_{P2} + w_1Y_{P2}$, where Y_{P1}, Y_{P2}, Y_{P3} correspond to measured pixel color of positions P_1, P_2, P_3 accordingly. Weights w_1, w_2, w_3 are normalized and sum up to 1.

Weights defined as volumes V_1, V_2, V_3 which correspond to PP_2P_3O, PP_3P_1O, PP_1P_2O tetrahedrons, where $O = (0, 0, 0)$. All volumes are normalized, which means $V_i = \frac{V_i}{\sum_{i=1}^{3} V_i}$. Volumes calculated as determinates of 4×4 vectors

$$V_1 = \tfrac{1}{6} \left| det(PP_2P_3O) \right|$$
$$V_2 = \tfrac{1}{6} \left| det(PP_3P_1O) \right|$$
$$V_3 = \tfrac{1}{6} \left| det(PP_1P_2O) \right|$$

2.4 Streaming

The size of compressed BTF data using PCA is around 10-20 Mb on a hard drive. Even though, compressed BTF data size is sufficient for plausible rendering, it is still big enough for transferring it over the internet for web rendering. The full transfer of BTF may take some minutes, especially with slow internet connection. To make this process faster and rendering a bit more interactive, we will use progressive streaming of BTF data. Deploying streaming a user will be able to see a preview of original BTF just in a few seconds. Principal components will be streamed one by one using WebSockets [17] and rendering will be refreshed whenever a new component arrives. The quality will

be progressively improved during streaming, see Figure 2.5 for an example. Also, it is possible to show an overall progress for the user, which makes the process of streaming even more interactive.

We use WebSockets for streaming the data, as it is most efficient and elegant way of communicating between a server and a client nowadays. The following advantages of WebSocket technology [17, Ch. 1] are:

- **Delivers high *Performace*** for real-time server-client connections. Usually, web developers used well known methods such as polling, long polling, and HTTP streaming. However, WebSocket saves bandwidth, CPU power, and latency compared to those methods. For example, polling method makes requests to the server and has to wait for the response. With WebSockets the client does not need to wait for the response, because WebSockets reuse the same connection from client to the server and wise-verse. This single connection reduces the latency.
- ***Saves time*** to develop web-applications. *Simplicity* of is one of the main advantage over older methods for server-client communication.

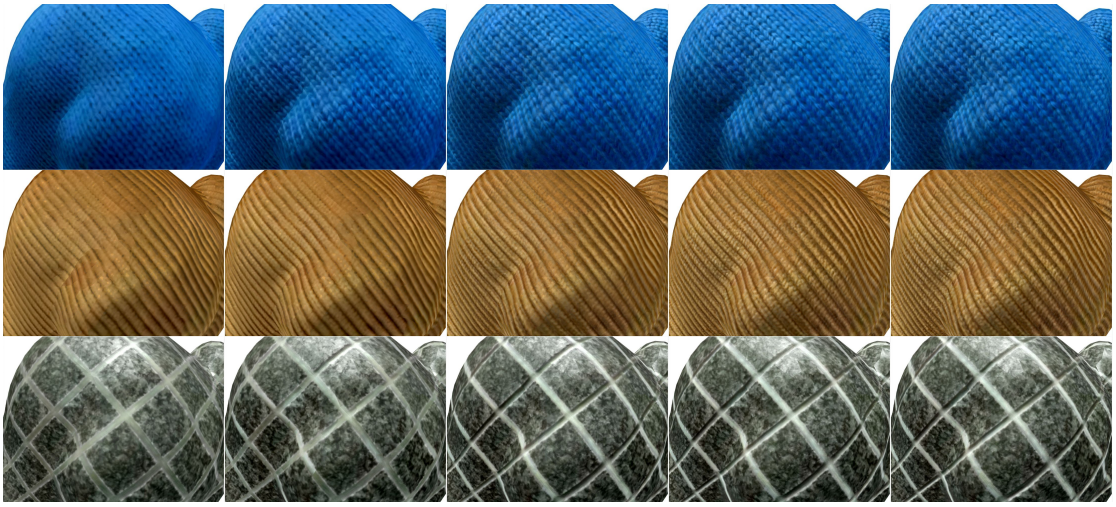

FIGURE 2.5: **Example of Progressive Streaming**
From left to right: *1*, *2*, *4*, *6*, *7* components rendered at the same time.
Note: *8th* component is a mean value m_i, which sends at the first place.

As it was described in Section 2.2.3 matrices U, Σ, V and m_i store all the needed data to reconstruct BTF. Matrices Σ and V are quite small in size, they are sent at first place. On the contrary, matrix U stores principal components of size $W \times H$ which makes matrix U the biggest in size of all. Matrix U also stores mean values m_i as first component. After a Socket connection was established, the client side requests one component at a time. Each newly arrived component saved in matrix U and the client side refreshes the shader to render BTF.

Consider Figure 2.5 that shows how the streaming works on practice. We can see that even with first components the resulted texture looks quite decent. With further components the overall quality of texture improves, e.g. specularities are increasing, small micro-structures become more visible and emphasized. To make the streaming process a bit entertaining, the client also can see the progress-bar of the streaming progress.

Chapter 3

Implementation

This Chapter describes the details of the implementation, problems and limitations that may arise using the developed work.

3.1 Compression

To compress a BTF data we used Java programming language. In our case, the main problem of PCA implementation lies in employing a singular value decomposition (SVD). To solve this problem, we used a fast linear library *jblas* [4] developed by Mikio Braun. The *jblas* library is gaining popularity in scientific computing. This library is one of the most fastest library for the Java programming that can solve various linear algebra problems.

A compressed data has to be sent to a shader. The best way to send arrays of data to a shader, is to send them as textures. So, after we perform a SVD, we save our resulted matrices U, Σ, V as textures. Matrices U and V are in range $[-1; 1]$, so we map the data to an image domain $[0; 1]$. We store each component of matrix U separately as PNG images. We save each component separately as we would need to stream the data in pieces.

Consider the examples shown in Figure 3.1, which represents first components of some materials. Matrices Σ and V are stored together in one texture as they are small enough. Note that the values of matrix Σ are not in range $[-1; 1]$, but we are still able to map them and store in the texture. We will not go in details here, as it is a trivial task.

Also, one practical consideration when using *jblas* is to scale the data for better reconstructed quality in a shader. We found out that tenths of values of U and V

matrices are zeros. So, basically we can scale matrices by multiplying them with 10 and at the same time matrix Σ by 0.1. Because, the texture stores only 8-bit values, means that the data loss during mapping is inevitable for some values. At least, with scaling we found that we improve the precision of the decompressed data.

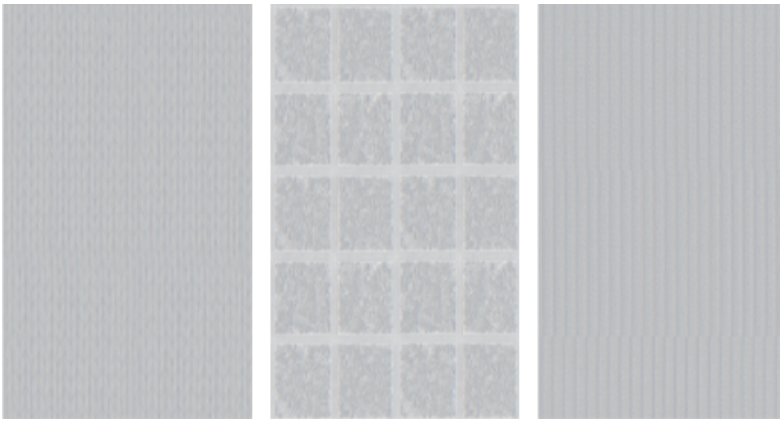

FIGURE 3.1: **Example of Principal Components**
From left to right: wool, impalla, corduroy.

3.2 Rendering

The aim of this thesis was to implement efficient BTF-shader for XML3D [16]. XML3D platform was implemented to deploy 3D graphics in web browsers. This technology is based on WebGL and JavaScript. BTF-shader is written in OpenGL Shading Language (GLSL). The rendering process of the shader is depicted in Figure 3.2.

A compressed BTF data is stored in two textures, which are passed to a shader. First texture L stores principal components, the other R stores PCA weights, which determine how the components have to be summed up. The other inputs are texture coordinates, eye and light positions. Eye and light positions are transformed to spherical coordinates. Then, we lookup three closest views A, B, C from the measured BTF data, which are will be needed for interpolation purpose. This lookup process is fairly simple. We have a static array in the shader, which stores sample intervals of the measured BTF data. Knowing intervals we can find the closest view positions. Finally, we employ barycentric coordinates interpolation for the closest eye and light positions and obtain interpolation weights w_o and w_i accordingly.

Meanwhile, we have to find appropriate principal components, i.e. to find the index of the texture that was sampled with given eye and light positions. For given eye and light positions we lookup a suitable texture index out of $3 \times 81 \times 81$ texture indices. In other words, we find which PCA weights from input texture R we need. Finally, texture

coordinates specify which exactly pixel has to be decompressed from the texture domain, i.e. which principal components from input texture L has to be used. Combining PCA weights with principal components we get uninterpolated color.

In a final step, uninterpolated colors are then combined with earlier found interpolation weights to get the final color.

The implemented shader has the it's own limitations. First of all, the number of principal components has to be fixed for a shader, because WebGL does not allow dynamic loops. Secondly, number of principal components are bound by a size of a texture L. It means not all GPUs can handle very big textures, so it must be considered beforehand. For example, the texture of size 4096MB can store 8 components with subset size of 3. Such size can handle average GPU. Last but not least, the array of sample intervals of the BTF are pre-compiled with a shader. For the BTF Bonn database all materials were sampled under the same intervals.

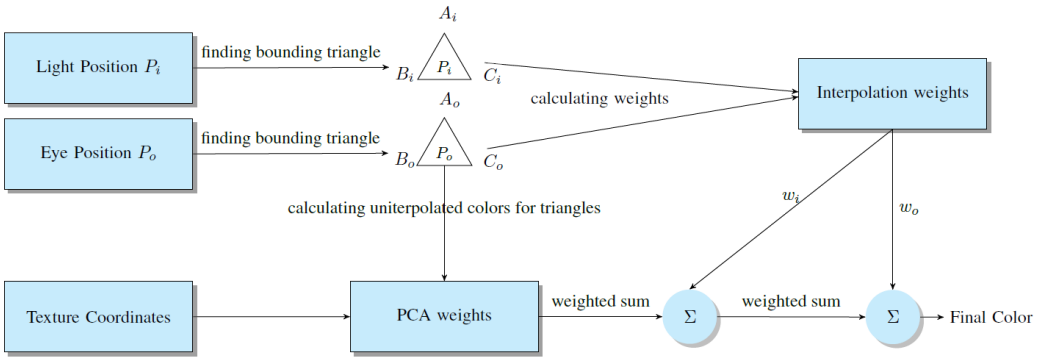

FIGURE 3.2: **Shader Design**

3.3 Streaming

As for streaming solution we used Node.js[5] platform, which is written in JavaScript. Node.js allows to write efficient web applications that is event-driven and employs a non-blocking I/O model. On top of that, we employ BinaryJS[1] library for Node.js, which provides us with binary streaming using web-sockets.

Node.js server is independent from main web server where an user browsers 3D content. After the user connects and the page completely loads, JavaScript client side requests Node.js server to start streaming BTF data. That means that all compressed BTF data are located on the Node.js server.

We also use Xflow[9] tool for a client side. Xflow is a part of XML3D implementation, which allows to process data on flow, i.e. in runtime. So, we use Xflow for saving all

newly arrived principal components in texture L on a client side. Figure 3.3 depicts the streaming process.

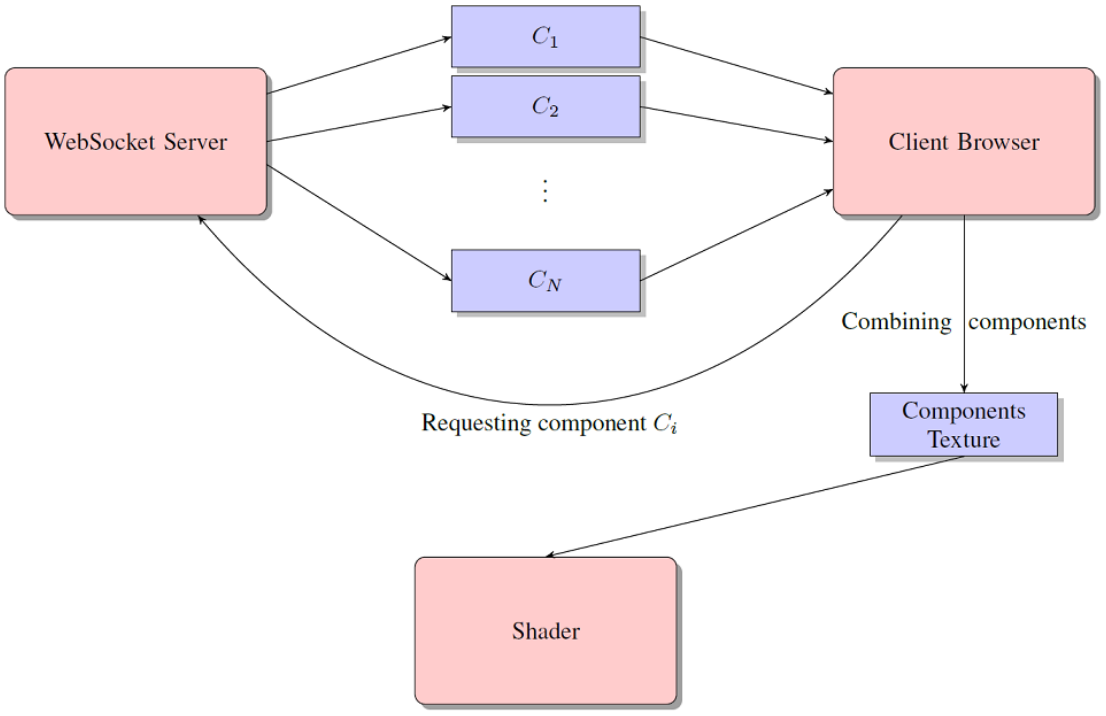

FIGURE 3.3: **Streaming process illustration**

At the start of a stream, we first send texture R and meta data to the client side. Then, on the client side we create array with the name *texData*. This array stores RGB colors, which further will be passed as texture L to the shader. As it was described in chapter 2.4, we stream principal components C_i one by one. Each principal component C_i for all possible sampled views are streamed as PNG image. Node.js server streams images in chunks, which permits client side other processing to continue before the whole transmission of PNG image is finished.

Then, on a client side we decode PNG image using PNG decoder written in JavaScript by Arian Stolwijk [6]. PNG images are decoded to pure array of RGB colors and stored in its place in *texData* array. Finally, using Xflow from *texData* array we create texture L, with which we update our BTF-shader.

Chapter 4

Evaluation

Chapter 5

Conclusions and Future Work

5.1 Summary

5.2 Future work

Bibliography

[1] Binaryjs framework. http://binaryjs.com/. Accessed on July 2014.

[2] Btf database bonn 2003. http://cg.cs.uni-bonn.de/en/projects/btfdbb/download/ubo2003/. Accessed on July 2014.

[3] Curret btf database. http://www1.cs.columbia.edu/CAVE/software/curet/index.php. Accessed on July 2014.

[4] Linear algebra for java - jblas. http://mikiobraun.github.io/jblas/. Accessed on July 2014.

[5] Node.js websocket platform. http://nodejs.org/. Accessed on July 2014.

[6] Pure javascript png decoder. https://github.com/arian/pngjs/. Accessed on July 2014.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Gero Müller, Jan Meseth, and Reinhard Klein. Fast environmental lighting for local-pca encoded btfs. In *Computer Graphics International 2004 (CGI 2004)*, pages 198–205. IEEE Computer Society, June 2004.

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

[13] Joseph Salmon, Zachary Harmany, Charles-Alban Deledalle, and Rebecca Willett. Poisson noise reduction with non-local pca. *J. Math. Imaging Vis.*, 48(2):279–294, February 2014.

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

[15] 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.

[16] 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.

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