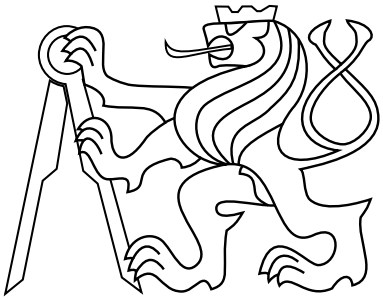
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master's Thesis

Compression Algorithms for BTF Data on a GPU

Bc. Petr Egert

Supervisor: doc. Ing. Vlastimil Havran, Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Graphics and Interaction

May 11, 2014

Acknowledgements

I would like express my gratitude to doc. Ing. Vlastimil Havran, Ph.D. for introducing me to the topic, supervising this work and providing numerous helpful advice on the way. My deepest appreciations go to my family for their love and care. My sincere thanks also go to Bc. Miroslava Květová for her endless moral support and to Bc. Iveta Kodádková for proofreading and being the best of friends.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 9, 2014 .....

Abstract

Bidirectional Texture Functions (BTFs) provide a way of capturing and accurately representing visual appearance of complex real-world materials in the world of computer graphics. Their main drawback is the amount of space required to store the material data, which can reach up to several gigabytes for a single material sample. To reduce the size to a more manageable level a compression technique needs to be applied to the BTF.

This thesis extends the Multi-Level Vector Quantization approach to BTF compression introduced by Havran et al. in 2010. The main contribution of our work is a highly parallel, highly modular, GPU-based implementation of both the compression and the decompression algorithm. Using our implementation, we were able to decrease the average compression time by the factor of 10 to less than 3 hours per material, increase the average decompression rate to about 120 million individual BTF evaluations per second in worst-case conditions and provide a much greater degree of flexibility to the compression algorithm. Our compression pipeline is fully configurable, allows the use of custom decomposition schemes and is ready for multi-spectral data processing. We demonstrate our results on a set of 14 different BTF samples compressed using 4 different compression pipeline layouts.

Abstrakt

Obousměrné texturní funkce (Bidirectional Texture Functions, BTFs) reprezentují způsob získávání a přesné reprezentace vzhledu komplexních materiálů všedního světa ve světě počítačové grafiky. Jejich hlavní nevýhodou je množství místa potřebného pro uložení dat, které se pro jeden materiál může vyšplhat až na několik gigabytů. Aby mohla být velikost BTF dat redukována na použitelnou úroveň, je nutné na ně aplikovat určitou kompresní metodu.

Tato diplomová práce rozšiřuje metodu pro kompresi BTF dat založenou na víceúrovňové vektorové kvantizaci, představenou Havranem et al. v roce 2010. Hlavním přínosem naší práce je vysoce paralelní, vysoce modulární GPU implementace komprimační i dekomprimační části algoritmu. Pomocí naší implementace jsme byli schopni snížit průměrný čas komprese cca.10× na méně než 3 hodiny na materiál, zvýšit průměrnou rychlost dekomprimace na přibližně 120 milionů individuálních vyhodnocení BTF za vteřinu (při nejhorších podmínkách) a poskytnout pro kompresní algoritmus mnohem větší míru flexibility. Naše kompresní pipeline je plně konfigurovatelná, umožňuje použití uživatelských dekompozičních schémat a je připravena na zpracování multispektrálních dat. Výsledky demonstrujeme na sadě 14 různých BTF materiálů zkomprimovaných pomocí 4 různých kompresních schémat.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The means of capturing and accurately representing realistic appearance of real-world materials are one of the key topics in the field of computer graphics. From simple shading and texturing to complex physically-based equations, the goal is to trick the viewer into believing, that what he/she sees would behave the same way in the real world. This task, being difficult by itself, can get even harder, when some external constraints, such as the amount of available processing power or memory, need to be satisfied. Proper way of representing the material appearance during rendering is therefore of key importance.

One of the more promising approaches to this problem is the use of Bidirectional Texture Functions (BTFs). Introduced by Dana et al. in [DvGNK99], a BTF is capable of capturing and representing complex material properties, such as self-shadowing, subsurface scattering or anisotropy. As a result, even materials with complex microstructure, such as fabrics, can be accurately rendered using BTFs as shown in Figure 1.1. Compared to a conventional approaches such as using two-dimensional textures, using BTFs yields much more realistically-looking results.

1.1 Motivation

The main drawback of BTFs is the amount of data required to store the material appearance. In raw form, the size of a single material represented by a BTF typically ranges from ones to tens of gigabytes of space [SSK03]. This would render the BTFs virtually useless for real-time rendering, since the amount of memory available on a current generation consumer-grade graphics card varies between 512 MB and 2048 MB [STEAM].

Several methods, as described for example in the survey performed by Filip and Haindl in 2009 [FH09], were proposed to overcome this limitation by compressing the BTF data into smaller size, preferably without sacrificing much of the visual quality of the material. One of such methods, based on Multi-Level Vector Quantization was introduced by Havran et al. [HFM10]. This method yields good compression ratios, while maintaining high visual fidelity of the material and allows direct evaluation of reflectance, as well as importance sampling, directly from within the compressed data. Its main drawback is the high compression time, which, for a single material, can take up to tens of hours. This makes it difficult to observe the outcomes of possible modifications and improvements of the algorithm.

(a) 2D texturing.

(b) BTF.

Figure 1.1: Difference between basic 2D texturing and BTF of the *Corduroy* material. Notice the lack of self-shadowing and overall flat look of the 2D texturing approach.

If the processing time of the algorithm is reduced to a more manageable level, it would simplify future research of the subject. This may provide clarifications as if some of the decision choices made in [HFM10] for the original algorithm (such as the level on which luminance gets separated) were correct and yield another possible improvements to the algorithm. By reducing the processing time, the algorithm also gets more production-ready. This may help to speed up the widespread use of BTFs to represent material appearance in both offline and real-time applications, resulting in better-looking renderings for the end users.

1.2 Subject of this Thesis

This thesis further investigates the Multi-Level Vector Quantization based approach to BTF data compression originally proposed by Havran et al. [HFM10]. Means of parallelization of the algorithm are explored and later evaluated by running the algorithm in a massively-parallel GPGPU environment using the OpenCL heterogeneous computing framework. To evaluate the performance of the parallel algorithm and to explore additional possible improvements to it, a highly modular, highly configurable implementation was created. This implementation allows the user to modify many aspects of the compression algorithm at run-time and, given its high modularity, simplifies further research on the topic.

1.3 Thesis Structure

In Chapter 2, the formal description of Bidirectional Texture Function (BTF) is provided, including the reasons why problem-specific compression methods need to be applied and what are the expected properties of such methods. In Chapter 3, the Multi-Level Vector Quantization BTF compression algorithm is described in detail. The main drawbacks of the current state of the algorithm, as well as proposed solutions to these problems are further discussed in Chapter 4. In Chapter 5 an introduction to heterogeneous computing and the OpenCL framework is provided. Details of the implementation itself are then described in Chapter 6, followed by validation of the implementation correctness in Chapter 7. The results obtained from processing 14 different BTF materials in 4 different compression pipeline layouts are discussed in Chapter 8. Finally, our conclusions are summarized in Chapter 9, including possibilities for future work on the topic.

Chapter 2

Bidirectional Texture Function Description

In this chapter, the basic concepts of Bidirectional Texture Function as a method of capturing realistic real-world material appearance are described as well as the main advantages and disadvantages of this method. The need for a problem-specific compression algorithm is explained and an overview of existing approaches to BTF compression is provided.

2.1 Formal Definition

A monospectral BTF is a six-dimensional (6D) function, which returns the amount of light reflected by an arbitrary point on the material surface ($[x, y]$ dimension), when viewed from an arbitrary direction ($[\theta_V, \phi_V]$ dimension) and illuminated from an arbitrary direction ($[\theta_I, \phi_I]$ dimension) [DvGNK99], as demonstrated in Figure 2.1. After extending the concept with color channel information, a multispectral, seven-dimensional (7D) BTF is obtained.

To clarify the concept, BTF can also be thought of as an extension to basic planar texturing, where the amount of light reflected by a single BTF texel depends not only on its position, but also on the view and illumination direction for the given texel.

2.2 Advantages and Disadvantages

Compared to other methods, BTF is successful at capturing complex visual properties given by the material microstructure, such as subsurface scattering, self-shadowing, self-occlusions etc.. To capture a single BTF, thousands of photographs are taken of a real-world material sample from different combinations of view and illumination directions. This results in much more realistically looking material appearance in the final renderings, as demonstrated in Figure 2.2.

The process of acquiring a BTF sample also indicates the main drawback of this method. Thousands of images are acquired for a single material. Even when compressed by common image compression techniques such as PNG or JPEG, the resulting data set can reach several gigabytes in size. This would make the method hard to use in offline rendering and almost impossible to use in real-time graphics.

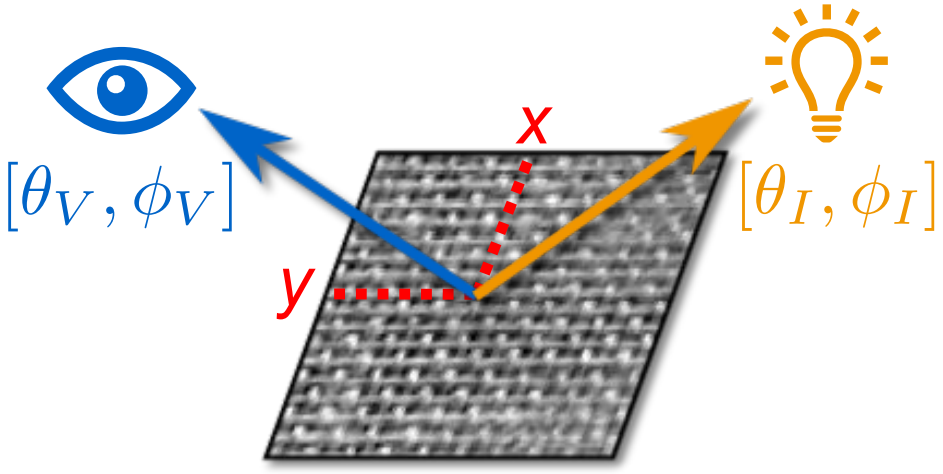


Figure 2.1: The coordinate system of a BTF

2.3 Compression Method Requirements

In order to overcome the limitations resulting from the BTF data size a problem-specific compression algorithm needs to be designed. This algorithm should be able to drastically reduce the BTF data size, while still maintaining their high visual quality. Another important requirement is the ability to perform direct decompression of a specific piece of data, without the need to decompress the whole BTF back to its original form (since then the algorithm would only be useful for storage). The decompression speed should be reasonably high, to allow the use of BTFs in both offline and real-time rendering. A direct support for importance sampling would also make the algorithm more useful in predictive rendering systems. The compression time should be kept as low as possible, ideally in a few hours range for a single material.

2.4 Overview of Existing Compression Methods

A survey of some of the BTF compression techniques can be found in [MMS+05]. A more recent comparison was presented by Filip and Haindl in [FH09]. According to [HFM10], the compression methods available can be divided into three basic groups, depending on the general approach used to process the BTF data.

The first group comprises of algorithms based on linear-basis decomposition. A matrix based approach was presented by Koudelka et al. in [KMBK03], followed by tensor based approach by Vasilescu et al. in [VT04]. The main drawback of these methods is the need to create linear combinations of a large number of components, which prevents them from achieving sufficient decompression speeds [HFM10]. Vector quantization based approach to BTF data was introduced in [MMK03] and further studied in [KM06]. In [LM01] a method based on capturing surface appearance using three-dimensional (3D) textons was presented and further extended in [TZL+02] and [LHZ+04]. More recently, a technique based on sparse tensor decomposition and K-SVD algorithm was proposed in [RK09].

Figure 2.2: Bunny model covered by *Pulli* BTF material and rendered using environment map lighting setup

Algorithms in the second group are based on the principle of representing parts of the BTF data using analytical reflectance models, such as the Lafortune reflectance lobes [LFTG97]. This approach first appeared in [DLHS01] and its fitness for BTF compression was further exploited in [MLH02]. The use of multiple Lafortune lobes is suggested in [MMK03]. In [FH05] polynomial extension to single lobe Lafortune model is proposed, combined with a clustering algorithm to further increase the compression ratio. An approach based on Phong lighting model fitting combined with a spatial-varying residual function was proposed in [MCC+04]. Representing the BTF using a stack of semi-transparent layers was later studied in [MK06]. More recently, Sparse Parametric Mixture Model [WDR11] was introduced, which decomposes the BTF data into a linear combination of multiple separate reflectance models.

Algorithms based on probabilistic BTF modeling form the final group. An example of such approach based on Markov chains was presented in [HF07]. These methods show the potential to provide very high compression ratios and can be used to synthesize BTFs of arbitrary resolution, but reach compromise results for highly non-Lambertian materials [HF07].

More recently a Multi-Level Vector Quantization (MLVQ) BTF compression method was

introduced in [HFM10], treating slices of BTF data as conditional probability density functions (PDFs). In its original version the algorithm already meets most of the requirements specified in Section 2.3. One of the remaining problems of this algorithm is the compression time, which ranges from 15 to 50 hours for a single material [HFM10].

Chapter 3

Multi-Level Vector Quantization Algorithm

This work extends the multi-level vector quantization compression algorithm proposed by Havran et al. in 2010 [HFM10]. In this chapter the basic concepts of the algorithm are summarized and a general outline of the compression and decompression algorithm is provided.

3.1 Algorithm Overview

The algorithm is based on the concept of finding similarities between the reflectance behaviour of individual texels of the BTF. If at least a partial similarity is found in the data, vector quantization (VQ) algorithm is applied to represent the similarly appearing data by only a single representative entry. This reduces the overall data size at the cost of reduced visual quality resulting from the use of quantization. To achieve better compression ratios, the reflectance behaviour is not evaluated only for the whole texel, but also for progressively smaller portions of its data, resulting in a multi-level vector quantization model.

The compression algorithm begins by transforming the input BTF data of each BTF texel to a custom, *Onion-Slices* parameterization. The parameterization allows treating the data as multidimensional conditional probability density functions (PDFs). Vector quantization is then applied to the PDFs in hierarchical way. If vector similar enough to the input PDF is present in the already compressed data, it is used to represent the PDF, thus reducing the space required if it was to be saved again.

If no such vector is found, the PDF gets decomposed by one of the input BTF dimensions into a defined number of subregions, which are then again input to the vector quantization algorithm. The decomposition can take place multiple times, performing the VQ on progressively smaller blocks of data. This allows to represent at least parts of the input BTF texel by already compressed data. Indices of entries representing the individual subregions get stored together as a single row into a compressed data codebook. As a single subregion itself can be represented by multiple smaller regions, the codebooks form a tree-like hierarchical structure. The leaves of this structure are formed by codebooks holding raw representative vectors of PDFs which do not get further decomposed.

To further increase the chance of finding a similar vector in the compressed data, both the input and the compared data are normalized to the same overall luminance level. The difference in luminance level is stored in form of a multiplicative constant referred to as a *scaling coefficient*. The coefficients are then stored along the indices into the codebooks. To achieve better compression ratios luminance and chrominance information also start to be treated separately after a given number of decompositions happen. This requires transforming the data to a more perceptually correct color model, which allows proper luminance and chrominance separation.

In the decompression part of the algorithm, the information stored within the compressed data codebooks are used to reconstruct the BTF back to its original form by means of fast chained indexing into the codebook hierarchy. Because the PDF decomposition is always performed along one of the BTF dimensions, the input BTF coordinates can be used to directly navigate through the codebook hierarchy. A single coordinate is used to find the corresponding subregion index in a codebook row. As this index points to a row in another codebook, the next input coordinate then determines the next row to access in a third codebook etc. Upon reaching the terminal codebooks, raw representative values of the BTF at the given coordinates are obtained. After applying the scaling coefficients and transforming these values back to the original color space, the resulting reflectance of the BTF is obtained.

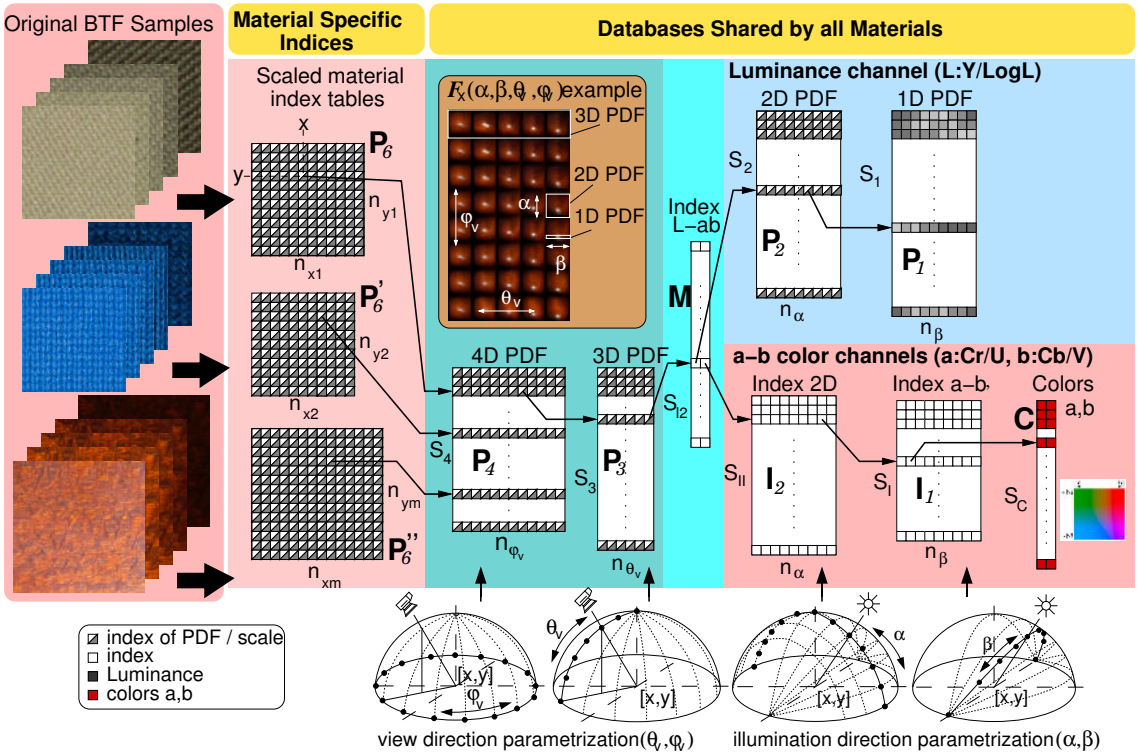


Figure 3.1: Multi-level vector quantization BTF compression algorithm scheme *Image courtesy of [HFM10]*

3.2 Onion-Slices Parameterization

The key concept of *levels* in the MLVQ algorithm is the ability to recursively split a large region of data into several smaller sub-regions and process those individually. To be able to perform the split correctly and use the resulting data directly within the MLVQ algorithm, a novel parameterization model was proposed in [HFM10], referred to as the [α, β] or *Onion-Slices* parameterization.

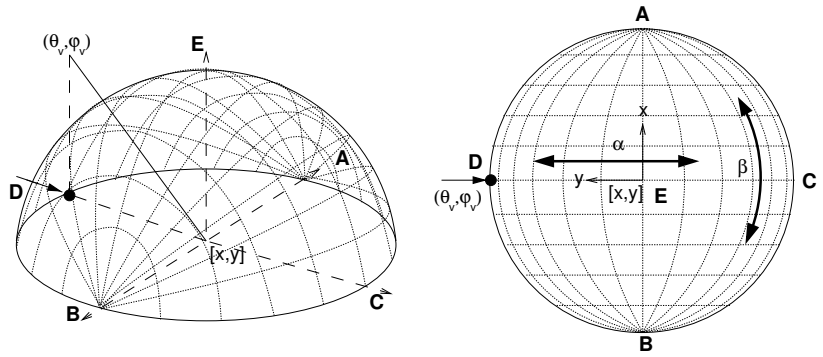


Figure 3.2: Relation between standard spherical coordinates and the *Onion-Slices* parameterization used for illumination direction parameterization. *Image courtesy of [HFM10]*

The use of this parameterization allows treating individual slices of BTF data as conditional probability density functions (PDFs). The relation between a standard spherical coordinate system and the *Onion-Slices* parameterization is demonstrated in Figure 3.2. The equations used to convert spherical coordinates into the *Onion-Slices* parameterization are shown in Equation 3.1. The relation to the Cartesian coordinate system is demonstrated using Equation 3.2. The *Onion-Slices* parameterization is only used to describe illumination direction coordinates. View direction remains in standard spherical coordinates.

$$\begin{aligned} \beta &= \arcsin\left(\sin\phi_I \cdot \cos\left(\theta_I - \theta_V\right)\right) \\ \alpha &= \arccos\left(\frac{\cos\phi_I}{\cos\beta}\right) \end{aligned} \tag{3.1}$$

$$\begin{aligned} [x, y, z] &= [\cos\theta \cdot \sin\phi, \sin\theta \cdot \sin\phi, \cos\phi] \\ [x, y, z] &= [\sin\beta, \sin\alpha \cdot \cos\beta, \cos\alpha \cdot \cos\beta] \end{aligned} \tag{3.2}$$

To convert input BTF data into the *Onion-Slices* parameterization, the raw data are first rearranged in such way, that all values for a single planar material position [x, y] are grouped together. A fixed discretization is then used for each of the four remaining dimensions (θ_V, ϕ_V, α, β). Using radial basis function (RBF) interpolation algorithm, the input data for each of the groups are resampled into the *Onion-Slices* parameterization, forming a *texel* of BTF data.

A single BTF *texel* holds all *Onion-Slices* parameterization resampled data elements for the given [x, y] coordinate pair and can be further accessed using the [θ_V, ϕ_V, α, β]

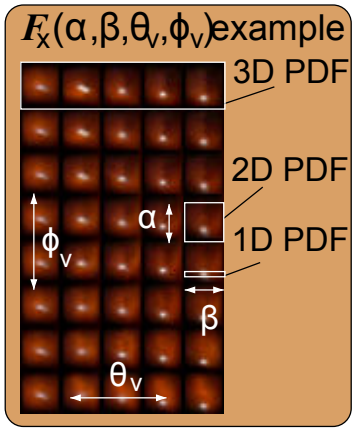


Figure 3.3: Arrangement of data in a single BTF texel resampled to *Onion-Slices* parameterization. *Image courtesy of [HFM10]*

coordinates. The information are stored in a specific layout, shown in Figure 3.3. This arrangement allows the compression algorithm to operate on progressively smaller blocks of data, forming conditional probability density function (PDFs) for the individual levels of data. The various sized PDFs are then used as input vectors to the multi-level vector quantization scheme.

3.3 Vector Quantization

The vector quantization (VQ) compression method, is based on the idea of representing a set of data vectors by only a smaller representative subset [GG91], referred to as *codebook* in this work. This is demonstrated in Figure 3.4. If two blocks of data are similar to each other, only one of them needs to be stored. For the remaining block, only a reference to the similar one needs to be kept. The original data for the second block can then be discarded. As a result, space is saved by not storing the *similar enough* data twice, but quality is also reduced due to the fact, that the data do not need to be exactly the same. This is a general concept of lossy-compression.

The multi-level vector quantization algorithm extends this concept by applying VQ to progressively smaller subsets of the original data. Instead of directly storing the input data to the codebook when a similar entry is not found, the input data can be split into several smaller regions, to which vector quantization is applied again. The process can be repeated multiple times. As a result, the representative values get only stored in the lowest-level codebooks, while on higher levels the codebooks form a tree-like hierarchy, where parts of entries in higher-level codebook point to entries in a lower-level one.

3.4 Similarity Metrics

The purpose of a similarity metric is to allow to compare two blocks of data and provide a single number in a specific range, representing the visual similarity between the two data

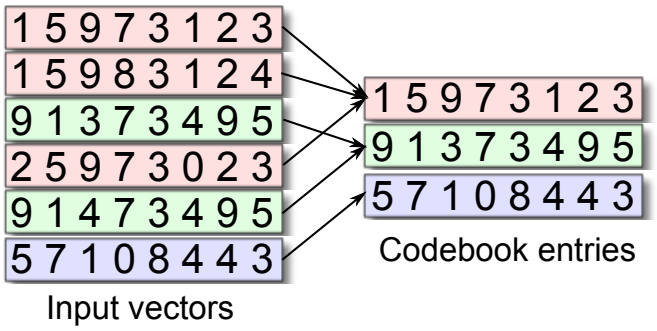


Figure 3.4: Example result of vector quantization algorithm. Multiple similar input vectors are represented by only a single codebook entry, thus reducing the overall data size.

blocks. By specifying a threshold for the returned value, a decision can be made whether the two blocks are *similar enough* to each other. This is crucial for the vector quantization algorithm to perform a search for similar entries.

Due to the facts stated above, choosing an appropriate similarity metric is of critical importance, because both the resulting compression ratio and quality depend on the choice. Numerous similarity metrics for comparing image data exist, such as Mean Square Error (MSE), Picture Signal-to-Noise Ratio (PSNR), Structural Similarity (SSIM) [WBSS04] or Visual Information Fidelity (VIF) [SB06], varying both in computational complexity and the number of visual features taken into account when computing the similarity index. In [HFM10] the Structural Similarity (SSIM) metric [WBSS04] is used (Equation 3.3).

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{3.3}$$

3.5 Color Model Transformations

To achieve better compression ratios the luminance and chrominance data of the BTF texel get compressed separately. As the input BTF data are usually stored in the RGB color model, conversion to a more perceptually uniform color model, which allows easy separation luminance and chrominance information, needs to be performed.

For low dynamic range (LDR) data, the *YCbCr* color model [ITU601] is used. Equation 3.4 shows transformation from RGB to the *YCbCr* model.

$$\begin{aligned} Y' &= 16 + \frac{65.738 \cdot R'}{256} + \frac{129.057 \cdot G'}{256} + \frac{25.064 \cdot B'}{256} \\ C_B &= 128 - \frac{37.945 \cdot R'}{256} - \frac{74.494 \cdot G'}{256} + \frac{112.439 \cdot B'}{256} \\ C_R &= 128 + \frac{112.439 \cdot R'}{256} - \frac{94.154 \cdot G'}{256} - \frac{18.285 \cdot B'}{256} \end{aligned} \tag{3.4}$$

Reverse transformation from *YCbCr* back to the RGB color model is provided in Equation 3.5.

$$\begin{aligned} R' &= \frac{298.082 \cdot Y'}{256} + \frac{408.583 \cdot C_R}{256} - 222.921 \\ G' &= \frac{298.082 \cdot Y'}{256} - \frac{100.291 \cdot C_B}{256} - \frac{208.120 \cdot C_R}{256} + 135.576 \\ B' &= \frac{298.082 \cdot Y'}{256} + \frac{516.412 \cdot C_B}{256} - 276.836 \end{aligned} \tag{3.5}$$

For high dynamic range (HDR) data, the *LogLuv* [Lar98] color model is used. To convert input RGB data to the *LogLuv* color model, Equation 3.6 is first used to transform the values into CIE XYZ [SG31] model.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.497 & 0.339 & 0.164 \\ 0.256 & 0.678 & 0.066 \\ 0.023 & 0.113 & 0.864 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.6}$$

From CIE XYZ model the data can then be converted to *LogLuv* using Equation 3.7.

$$\begin{aligned} L_e &= \lfloor 256 \left(\log_2 Y + 64 \right) \rfloor \\ u' &= \frac{4x}{-2x + 12y + 3} \\ v' &= \frac{9y}{-2x + 12y + 3} \end{aligned} \tag{3.7}$$

where:

$$\begin{aligned} x &= X/(X + Y + Z) \\ y &= Y/(X + Y + Z) \end{aligned}$$

A reverse conversion from *LogLuv* to the CIE XYZ color model is provided in Equation 3.8.

$$\begin{aligned} Y &= \exp_2 \left[\left(L_e + 0.5 \right) /256 - 64 \right] \\ x &= \frac{9u'}{6u' - 16v' + 12} \\ y &= \frac{4v'}{6u' - 16v' + 12} \end{aligned} \tag{3.8}$$

where:

$$\begin{aligned} x &= X/(X + Y + Z) \\ y &= Y/(X + Y + Z) \end{aligned}$$

The CIE XYZ data can then be transformed back to the RGB color model using Equation 3.6.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.690 & -1.276 & -0.414 \\ -1.022 & 1.978 & 0.044 \\ 0.061 & -0.224 & 1.163 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.9}$$

3.6 Compression Algorithm

The outline of the compression algorithm is provided in Figure 3.1. The algorithm begins by resampling the BTF into the *Onion-Slices* parameterization as described in Section 3.2. After resampling the input data to the *Onion-Slices* parameterization, each dimension of the data corresponds to a single level in the MLVQ algorithm. This can be demonstrated in Figure 3.1. The original monospectral BTF has 6 dimensions - (x, y, θ_V, ϕ_V, α, β) thus forming a 6D PDF. If the planar [x, y] coordinates are fixed, a 4D PDF consisting of data for the remaining (θ_V, ϕ_V, α, β) dimensions is formed. By further specifying the view azimuthal angle θ_V, a 3D PDF is obtained. Similarly, 3D PDF is decomposed into 2D PDFs etc.

In order to represent the input set of vectors by only a smaller subset of representative vectors, the codebooks need to be searched for entries similar to the input data. For this reason, a similarity metric must be applied, as described in Section 3.4, which returns the level of similarity between two sets of data. If an entry similar enough (depending on predefined conditions) to the input data is found in an codebook, it is used to represent the given input vector. If no such entry is found, a new entry needs to be added to the codebook.

The compression algorithm starts in the P_6 codebook. For each texel of the input data an entry in the P_6 codebook needs to be created. The 4D PDF representing the texel data is first matched against the 4D level codebook P_4. If a similar entry is found, its index is returned and stored in the P_6 codebook at a location given by the [x, y] coordinates of the texel.

To achieve better compression ratios, the similar entry matching in most of the codebooks is done *up to scale*, meaning the data can differ by a multiplicative constant. Normalization is performed to bring the overall luminance levels of both the input and the compared vectors to the same level. If the normalized data match, index of the corresponding codebook entry is returned along with a *scaling coefficient*, the multiplicative constant used to normalize the compared data. This means, that a single entry in the P_6 codebook consists of an index into the P_4 codebook and a floating point scaling coefficient.

If no matching entry is found in the P_4 codebook, a new entry is created. In the MLVQ algorithm, only lowest-level codebooks contain raw representative vectors of BTF data. On higher levels, codebook entries are formed by indices pointing into lower level codebooks. As a result, the individual codebooks form a tree-like hierarchy. An entry in the P_4 codebook is created by splitting the original 4D PDF into a set of 3D PDF slices. Each 3D PDF is specified by the planar position [x, y] and the view azimuthal angle ϕ_V. The 3D PDFs are then matched against entries from the P_3 codebook. An index/scale pair is returned from this codebook for each of the 3D PDFs. The returned results are then stored into a row in the P_4 codebook. The number of entries in the row corresponds to n_{θ_V}, the number of steps in which the view azimuthal direction is parameterized. Workflow similar to the one already described repeats itself on all levels during the compression. A search for similar

entries is performed on the current level. If a matching entry is not found, the PDF is split into smaller slices, which are then matched against a a lower-level codebook.

Upon reaching the M codebook, the BTF data are converted from the original RGB to a more perceptually uniform color model, as described in Section 3.5. The M codebook is needed since the luminance and chrominance channels start to be treated separately and individual sets of codebooks are used to store luminance data (P_2, P_1) and chrominance data (I_2, I_1, C). The M codebook stores the information required to merge the separate information back together. Due to the nature of the data, scaling coefficients do not bring a significant advantage for chrominance data and are therefore no longer used in favour of the reduced codebook sizes. Entries in the I_2 and I_1 codebooks therefore consist only of indices, not index/scaling coefficient pairs.

In the lowest-level codebooks P_1 and C, raw representative vectors of resampled BTF data (1D PDFs) are stored as 1D arrays. Upon reaching P_1 and C the MLVQ compression is finished for a single BTF texel (4D PDF) and the next one can be processed. This is repeated until all the texels of the material are processed. The contents of all the individual codebooks then represents the compressed data of the material.

Before saving the compressed data to a file, their size is further improved by applying a scalar quantization algorithm to floating point values [GG91] and by compacting index data to a limited number of bits. The scalar quantization algorithm converts the floating point values to a fixed precision and uses only a limited number of bits (8 bits for LDR, 16 bits for HDR BTFs) to represent thee values. Index data in the codebooks are tightly packed together using minimum number of bits required to cover the whole value range. For indices stored in codebook P_i pointing to codebook P_{i-1} of size S_{i-1} the number of bits is computed as $N = \lceil \log_2(S_{i-i}) \rceil$. After performing these optimizations, the resulting codebook data get stored into a file and the compression algorithm is finished.

As shown in Figure 3.1, a tree-like hierarchy consisting of a set of 9 codebooks is used to store a single BTF material. Multiple materials can also be compressed into the same set of codebooks if the same discretization is used for all the materials. For each new material only one additional 6D-level codebook is added. This further increases the compression ratio and allows sets of materials to be packed tightly together.

3.7 Decompression Algorithm

The decompression algorithm is based on chained indexing within the compressed material codebooks as shown in Figure 3.5. Because the *Onion-Slices* parameterization was used when compressing the data, the input coordinates first need to be converted to this parameterization using Equation 3.1. Using the $[x, y]$ coordinates a corresponding entry is found in the top-level P_6 codebook. This entry consists of a scaling coefficient and a pointer into the 4D-level codebook P_4. In the P_4 codebook, the corresponding row is determined using the pointer obtained from the P_6 codebook. This row contains n_{θ_V} scaling coefficient/pointer pairs, targeting the 3D-level P_3 codebook. Using the θ_V input coordinate, the row is indexed to obtain a single[1] scaling coefficient/pointer pair. The P_3 codebook is accessed in a similar way and ϕ_V coordinate is used to index the matching row.

[1] Although using just a single scaling coefficient-pointer pair would work, in the actual implementation, two such pairs closest to the input coordinate are used and linear interpolation is performed between the

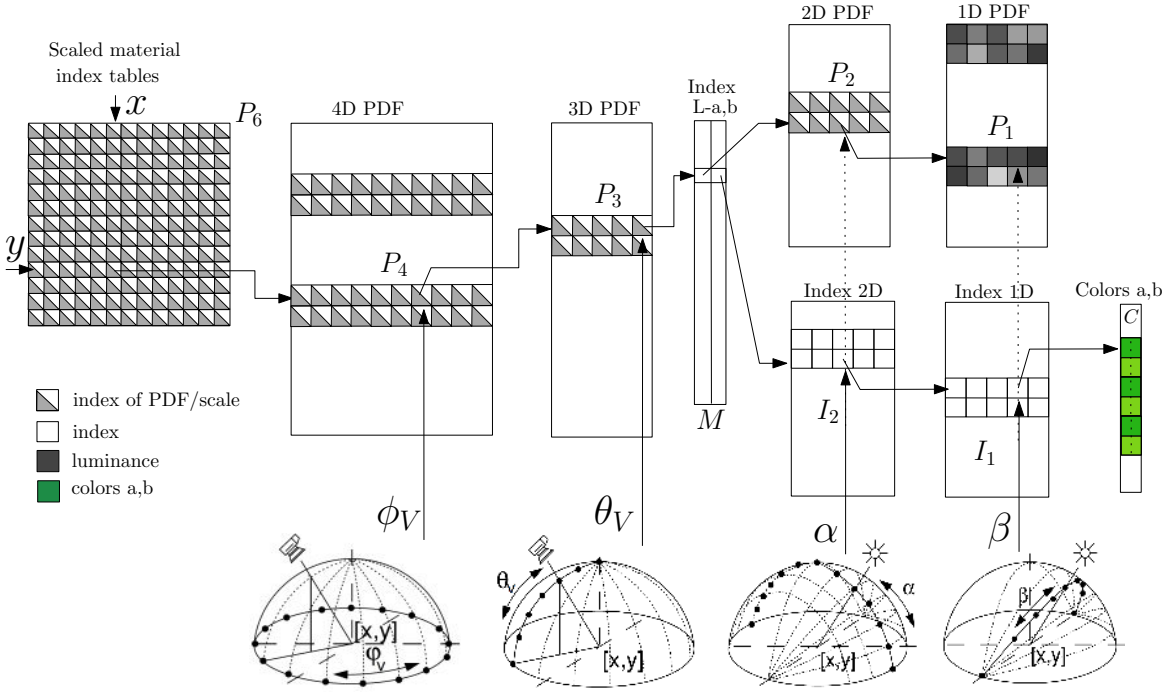


Figure 3.5: The chained-indexing algorithm used during BTF decompression

Upon reaching the M codebook, the decompression algorithm continues in two separate paths, because luminance and chrominance components of the BTF data are treated separately from this level down (as described in Section 3.6). Two pointers are obtained from the M codebook entry - one pointing into the P_2 codebook with luminance data, the other one pointing to the I_2 codebook with chrominance data.

The indexing repeats itself in the two separate paths until reaching the bottom-most level codebooks P_1 and C. Here, the raw representative values of the BTF at the given input coordinates are obtained. When returning back to the top, the scaling coefficients are applied to the raw values by the means of simple multiplication. In the M codebook, the separate luminance and chrominance are merged together and converted to the RGB color space. The type of conversion depends on the dynamic range of the material (YCbCr color model is used for LDR samples, LogLuv [Lar98] for HDR samples). Subsequent scaling coefficients are then applied to the RGB data.

After returning to the top-level codebook P_6 and applying the last scaling coefficient, the resulting reflectance of the BTF at the given coordinates is obtained as a triplet of values in the RGB color space.

obtained values to improve the resulting visual quality. A single pair is mentioned throughout the description to improve readability.

3.8 Advantages and Disadvantages

In its basic form, the MLVQ compression algorithm already meets two of the key requirements - it provides relatively high compression ratios, while maintaining high visual quality of the compressed BTF materials. From the decompression point of view, the main benefit of the algorithm is that the input coordinates are used to navigate within the compressed data without the need of any kind of preprocessing. This allows to directly evaluate the BTF at a specific set of coordinates without reconstructing any larger portion of data. This random access to the compressed data is necessary for all rendering algorithms. Because multiple BTF evaluations can be run in parallel, the decompression algorithm is well suited for GPU-based implementations. The algorithm also has a direct support for importance sampling, which makes it fit to use in predictive rendering applications.

The main downside of the compression algorithm is the time required to compress a BTF sample. According to [HFM10], the compression can take up to 50 hours for a single 256×256 sample (26.7 hours on average). As the resolution of the BTFs is expected to rise in the future, this would be a serious issue for the algorithm.

Chapter 4

Improvements of the MLVQ Algorithm

The disadvantages of the current state of the multi-level vector quantization algorithm for BTF data compression are summarized in Section 3.8. In this chapter, we describe the improvements of the original algorithm done in our implementation.

4.1 Parallelization

The compression workflow of the MLVQ algoritm is generally sequential, because the results from previous iteration (data stored within codebooks) are used as input to the next iteration (the search for already existing similar entries). This chain of operations cannot be broken without affecting the quality of the compression. It is however possible to parallelize smaller parts of the compression algorithm on several levels.

First, the search for similar entries within the individual codebooks can be efficiently done in parallel. Because typically there is more than one block of input data and more than one block of data to compare with, a single thread can be used to perform comparison of each of the combinations. The results of the comparisons do not depend on each other and can therefore be launched in parallel as an N×N workgroup.

During the compression, the input block of *Onion-Slices* parameterization BTF data gets split into progressively smaller sub-blocks. Most of the time, these sub-blocks can be also processed in parallel. This means that the search for matching entries as well as for example transformations into different color space can happen in parallel on all the sub-blocks. The problem arises when no matching entry is found for more than one of the sub-blocks and a new entry needs to be created in a codebook. If processed sequentially, one of the sub-block might form a codebook entry, to which the other block is matched. This problem can be resolved by processing the remaining sub-blocks sequentially to allow one block in the same batch to match another.

Transformations between different color spaces for a single block can also be done in parallel. For each of the elements of data to be converted a separate thread can be launched to perform the conversion.

Reconstruction of a single codebook entry back to its original representation (to a block of data in *Onion-Slices* parameterization) can be done in parallel. Similar workflow can be used to reconstruct a batch of entries, also in parallel.

Additionally, the transformation of raw input data into the *Onion-Slices* parameterization can be done in parallel, as later described in Section 6.3.5.7.

4.2 GPU-based Implementation

In this section we discuss some issues if GPU-based algorithms. The main limitation of a GPU is the amount of memory space available on the target hardware. Because this space is still quite limited (ones of gigabytes even on today's upper-class hardware), while the raw BTF data size is quite large (several gigabytes for a single material), a hybrid CPU-GPU based approach is proposed. The CPU part of the application is responsible for preparing batches of input data to be processed and feeds them to the GPU. The compression itself then runs within the GPU environment and all the required data are stored in the GPU memory, including the currently processed batch of input data, the compressed codebooks and various cache data. As the support for dynamic parallelism is still limited, the CPU is used to control the workflow of the compression. After the main compression algorithm is finished for the whole material, the resulting codebooks are downloaded from the GPU and stored to a file.

In [HFM10] the algorithm was proposed as sequential and was implemented to run on a single core CPU. Using the optimizations described in Section 4.1 it can greatly benefit from execution in the massively-parallel environment of a GPU. As a result, the time required to compress a single material is greatly reduced.

4.3 Radial Basis Function Interpolation

To transform the input BTF data into the required *Onion-Slices* parameterization, a resampling scheme based on Radial Basis Function (RBF) interpolation [CBC+01] is proposed.

The input BTF data are multidimensional and, as a result of measuring only a discrete number of view and illumination direction combinations, the data are scattered. For resampling into the *Onion-Slices* parameterization, values at coordinates different from those for which the BTF was measured need to be obtained. RBFs provide a way of interpolating such multidimensional scattered data. The value of a single RBF depends only on the distance of point $\mathbf{x}$ from the origin or some other center point $\mathbf{x_i}$. An example of Gaussian RBF is given by following equation:

$$\phi(r) = e^{-(\varepsilon r)^2} \tag{4.1}$$

where $r = \|\mathbf{x} - \mathbf{x}_i\|$.

The approximation function is then represented by a weighted sum of RBFs computed using the input data sample points as the RBF center points, as demonstrated in Equation 4.2,

$$y(\mathbf{x}) = \sum_{i=1}^{N} w_i \, \phi(\|\mathbf{x} - \mathbf{x}_i\|) \tag{4.2}$$

where w_i is the weight by which the given input sample contributes to to the resulting value. The weights can be estimated by the linear least squares method.

The *Onion-Slices* parameterization resampling algorithm is briefly described in Section 4.1. of [HFM10], where values are first interpolated for all illumination directions and a fixed viewing direction and only then for all viewing directions. Our approach is slightly different. For each of the required *Onion-Slices* parameterization coordinates, k-nearest neighbors are found within the input data. These k points are then used as center points in the RBF interpolation. Using the required *Onion-Slices* parameterization coordinates as a query point, the interpolated values are returned by the RBF. The whole algorithm can be executed in parallel on a GPU.

4.4 High Modularity

In our implementation, the compression algorithm is divided into three basic steps. Each of these steps is represented by a separate application, as later described in Section 6.1. The first step is to read the raw input BTF data in various formats and convert them into a single, strict defined format, from which the data can be easily resampled into the required *Onion-Slices* parameterization during the second step. After the data get resampled, they are input to the third and final step, which performs the MLVQ compression. The compression is handled by a highly modular and configurable processing pipeline running on a GPU.

4.4.1 Customizable Compression Pipeline

The compression pipeline represents the set of operations applied to the input BTF data during the compression. It consists of two basic elements - *nodes* and *compare units*. A *node* represents an operation with the input data, while *compare units* are used to apply similarity metrics between two data blocks. The whole pipeline is completely user configurable at run time. This means, that the user can select which types of nodes and compare units to use and how to connect them together. Each node and compare unit also has its own configurable parameters, some of which are consistent during the whole compression process, while some can be configured on a per-stage (Section 4.5.1) basis. The whole compression pipeline build-up is done in run time and different pipeline setups can therefore be achieved only by modifying the corresponding configuration files.

4.4.2 Pipeline Nodes

A node in the compression pipeline basically represents an operation performed on the input data block. The operation done is based on the type of the node. The purpose of some node type can be for example to find a similar enough entry in its codebook and if none is found, add the input data block as a new entry. The complete list of implemented node types is later provided in Section 6.3.5.

Pipeline nodes can be either *terminal* or *non-terminal* and must be interconnected to form a tree-like structure. *Terminal* nodes can only exist as leaves of the processing tree and can have no additional child nodes. *Non-terminal* nodes are on the other hand required to have one or more (depending on the node type) child nodes.

Each node operates on a fixed region of input data, which is provided to the node by its ancestor. The size of the region usually depends on the depth of the node in the processing tree. If the node uses a compare unit, it provides the compare unit with the requested data region. A single compare unit must be assigned to each node requiring it, but not all nodes need a compare unit to work. The assignment is done within the pipeline configuration.

4.4.3 Pipeline Node Categories

The types of nodes present in the pipeline can be divided into three basic categories: *basic nodes*, *nodes with codebook* and *nodes with attached compare unit*.

Basic Nodes

Basic nodes are used mainly to perform transformations of the input data or to control the data flow during the compression. Nodes in this category do not have their own codebook and therefore do not store any persistent data. A node of the basic node type can for example converts the input data between different color spaces.

Nodes with Codebook

Nodes with codebook contain one or more codebooks - memory regions in which they store some kind of persistent data. These data are typically sets of information received from their children for non-terminal nodes or vectors of representative BTF values for terminal nodes. The node writes the data to the codebook during the compression and reads from the codebook during the reconstruction. When the compression is finished, the content of all the codebooks represents the resulting compressed material data set.

Nodes with Attached Compare Unit

Nodes with attached compare unit extend nodes with codebook by specifying a compare unit, which can be used to perform comparison between an input data block and a block of data reconstructed from a codebook entry. This allows the node to first perform a search for a similar enough entry within its codebook and add new entries only when a similar enough entry does not exist.

4.4.4 Compare Units

The purpose of a *compare unit* is to compute a similarity index (described in Section 3.4) between the input data and the data reconstructed from one or more existing codebook entries. The compare units are implemented as separate entities which are common to the whole compression pipeline. This allows the user to configure which compare units should be assigned to individual nodes of the pipeline. From a developer's point of view, new similarity metrics can be implemented into new compare units without the need to modify the rest of the compression components.

4.4.5 Modifiers

In the original algorithm, individual codebooks consist of one of the three possible types of entries - indices only, indices & scaling coefficients pairs or raw representative vectors of the BTF. Also the type of entries used by each of the codebooks is static. In our implementation, the type of data stored in a codebook is generic and depends on the type of the child nodes connected to a node. This means, that if a child node uses scaling coefficients, the parent node will store them into its codebook along with the indices of the entries. If no scaling coefficients are provided, only the index data will be stored.

Additionally, properties other than the scaling coefficients can be returned from a node. If for example rotational symmetries in the input data are exploited, the number of rotation steps would be returned from the node along with the indices. The term *modifiers* is used to refer to the additional data returned by the node. According to this concept, a *scaling* modifier would be applied to the nodes representing the P_4, P_3, M and P_1 codebooks in Figure 3.1.[1] The amount of modifiers applied to a single node is not limited and combinations of different modifiers can therefore be used together.

4.5 Dynamic Configurability

Most aspects of the compression algorithm are user configurable at run time to allow the user to quickly modify the settings without the need to modify the application source code.

4.5.1 Compression Stages

To provide a more fine-grained control over the compression workflow and achieve better compression ratios, the compression algorithm can be executed in an arbitrary number of *stages*. For each stage, the configuration of individual nodes, compare units, BTF texel processing order and other properties can be specified. These can include for example the required similarity thresholds for individual nodes or the number of texels to compress during the stage. At the beginning of each stage, these properties are set and remain valid until the beginning of the next stage. The order in which the stages are executed is given by the order, in which they are specified within the configuration file.

4.5.2 Compression Pipeline Configurability

Configuration of a single node is divided in two parts. In the first part, global properties of the node are specified. These include the type of the node, its name, identifier (used to connect the nodes to form the processing tree), identifier of the compare unit to use (if required) and a list of target nodes (if any). Some node types might also require additional configuration parameters. The second part of node configuration is done on a per-stage basis. This might include required similarity thresholds for codebook-based nodes, cache configurations and other properties.

[1] The figure might look slightly confusing in this context, because the index/scaling coefficient pairs are shown for different codebooks than mentioned. This is in fact correct, because the result of a modifier is always saved in the parent node's codebook. If *scaling* modifier is applied to the P_4 codebook, the resulting scaling coefficients will be stored in the P_6 codebook.

4.6 Multispectral Data Processing

Processing of an arbitrary number of color channels is supported in the whole compression framework. This allows unified processing of both the regular RGB color model data and multispectral BTF data. Additional information such as transparency or self emissivity can also be stored as separate channels, which allows extending the data compressed beyond the scope of BTFs. As the meaning of the data is generally not known to the pipeline nodes, it is the responsibility of the user configuring the pipeline to understand the meaning of the individual channels and set up the pipeline accordingly. All types of nodes in the pipeline can operate on an arbitrary number of channels, although some operations with the data expect a fixed number of channels to be present. For example the YCbCr color model conversion (Section 3.5) requires three color channels as input. The node performing the conversion however supports an arbitrary number of input channels, since different types of conversion may require different number of channels. The number of color channels output from a node can also be different than the number of input channels.

4.7 Omission of Chrominance Lookup Codebook

In the original algorithm a separate codebook is used to store chrominance value pairs (C in Figure 3.1). In our implementation, we decided to remove the separate C codebook and store the chrominance information directly in the I_2 codebook. As one less quantization step is performed, the compression speed should be increased, as well as the resulting visual quality. The compression ratio should be the same or better, because by not performing the quantization at the C level, the chance of finding matching entries is increased on higher levels. During decompression, 16 to 64 cases (based on the interpolation method used) of indirect memory addressing would be eliminated by reading the chrominance values directly from the I_2 codebook, which should result in increased decompression speed. With minor changes, the compression pipeline can however be modified to include the separate C codebook.

Chapter 5

Heterogeneous Computing and OpenCL

The basic concepts of *heterogeneous computing* and *GPGPU* programming are summarized in this chapter, followed by the description of the OpenCL heterogeneous computing framework.

5.1 Heterogeneous Computing

The term *heterogeneous computing* refers to the process of using more than one type of computer system architecture to perform a required task [GHK+13]. The reason for using more than one architectures is to take advantages of the capabilities each of the different types offers. A CPU is for example well suited for general purpose tasks, involving program control, complex branching and operating system communication. A GPU is on the other hand well suited for performing a lot of parallel mathematical computations. Using heterogeneous computing, advantages of both the architectures can be combined together. To control the program flow a CPU can be used, while performance-demanding computations can be offloaded to the GPU.

In our work, heterogeneous computing is used to speed up both the compression and the decompression. The basic control flow and input/output handling is done on the CPU side. The CPU side is also used to schedule work for the GPU, but the compression and decompression itself is performed almost entirely on the GPU. All immediate data are stored within the GPU memory and only the final results get downloaded back to the system memory.

5.2 GPGPU

GPGPU or *General-Purpose Computing on Graphics Processing Units* is the process of using a GPU, originally intended for performing computer graphics computations, to perform general computation tasks. These can include complex calculations for example in the fields of biology, molecular dynamics or signal processing. Due to their massively-parallel nature GPUs can solve specific problems much faster than general CPUs.

GPUs are well suited for problems representable by the means of *stream processing* (SP). In the SP paradigm, a series of operations represented by a *kernel* function is applied to a set of input data, a *stream*. A resulting set of output data is generated, possibly also a stream later input to another kernel function. The units executing the kernel are usually independent and the means of communication and synchronization between individual execution units are limited. As a result, the kernel execution can be efficiently pipelined and run on multiple compute units in parallel to achieve high performance.

5.3 OpenCL

The *Open Computing Language* (OpenCL) is a programming framework providing abstraction of heterogeneous system architectures in form of a well defined standard [GHK+13]. The standard is maintained by *The Khronos Group* consortium, which also maintains the OpenGL standard. Code written in the OpenCL programming language (with a C-like syntax) can be compiled to run without change on various device types, including multicore CPUs, GPUs or field-programmable gate arrays (FPGAs). OpenCL natively supports both task- and data-level parallelism and is well suited for use in massively parallel environments. The standard is open, free to use, cross-platform and is adopted into drivers provided by many major computer hardware vendors such as AMD, Intel, NVIDIA or Apple.

5.3.1 Platform Model

OpenCL abstracts the different architectures and devices in heterogeneous computing using a platform-based model. This model is composed of the following components:

Platform
: A platform represents a group of *devices* among which resources can be shared.

Device
: A compute *device* is a collection of *compute units* and usually represents the target hardware performing the computations. Each *device* has its own capabilities, such as supported floating point precisions, memory size, limitations on *work-group* size or the amount of *compute units* available.

Context
: According to the official OpenCL 1.1 specification [OPENCL], a *context* is defined as: "The environment within which the *kernels* execute and the domain in which synchronization and memory management is defined. The *context* includes a set of *devices*, the memory accessible to those *devices*, the corresponding memory properties and one or more *command-queues* used to schedule execution of a *kernel(s)* or operations on *memory objects*."

5.3.2 Execution Model

The OpenCL execution model provides ways of exploiting both task- and data-level parallelism. The key components in this model are:

Kernel
The *kernel* function represents a set of operations which should be applied to the data during a single invocation by a *work-item*. Instruction-level parallelism can be utilized within the *kernel* code.

Work-item
A *work-item* represents a single invocation of a *kernel* by the device. It can be thought of as a single thread executing the function. Data-level parallelism can be exploited by running multiple work-items in parallel. Each *work-item* has two unique identifiers - one within the *work-group* (*local ID*) and one within the whole task (*global ID*).

Work-group
Multiple *work-items* executed on a single *compute unit* form a *work-group*. *Local* memory is shared between all *work-items* in a *work-group*.

Compute Unit
A *compute unit* is a set of processing elements (virtual scalar processors) on which *work-groups* get executed. Multiple *compute units* can be present in a single *device*, each having its own pool of *local* memory.

Command Queue
Command queue maintains the order of commands to be executed on a specific *device*. By default, the commands are executed in the same order in they were queued, but out-of-order execution is also possible with explicit synchronization.

5.3.3 Memory Model

The OpenCL memory model is composed of mutliple types of memory, as shown in Figure 5.1. Each of the memory types provides a different level of compromise between the amount of space available and the overall access performance. The following types are available:

Global Memory
Global memory is shared across the whole *context*. It is usually the largest in size, but compared to other memory types offers the slowest access speed.

Constant Memory
Constant memory is a specialized region of *global* memory, which remains unchanged during the execution of a *kernel*. The amount of available *constant* memory is usually more limited, but being read-only, the *device* executing the *kernel* can use a more efficient caching scheme and therefore increase the access speed to this memory region.

Local Memory
Local memory is shared only by *work-items* within the same *work-group*. Being closer to the actual *compute units*, the amount of space offered in this type of memory is usually quite limited, but the access performance is greatly improved.

Private Memory
Private memory is unique for each of the *work-items*. It is usually represented by

registers located directly within the processing elements. For this reason, the *private* memory provides the best access performance, but its available size is usually very limited.

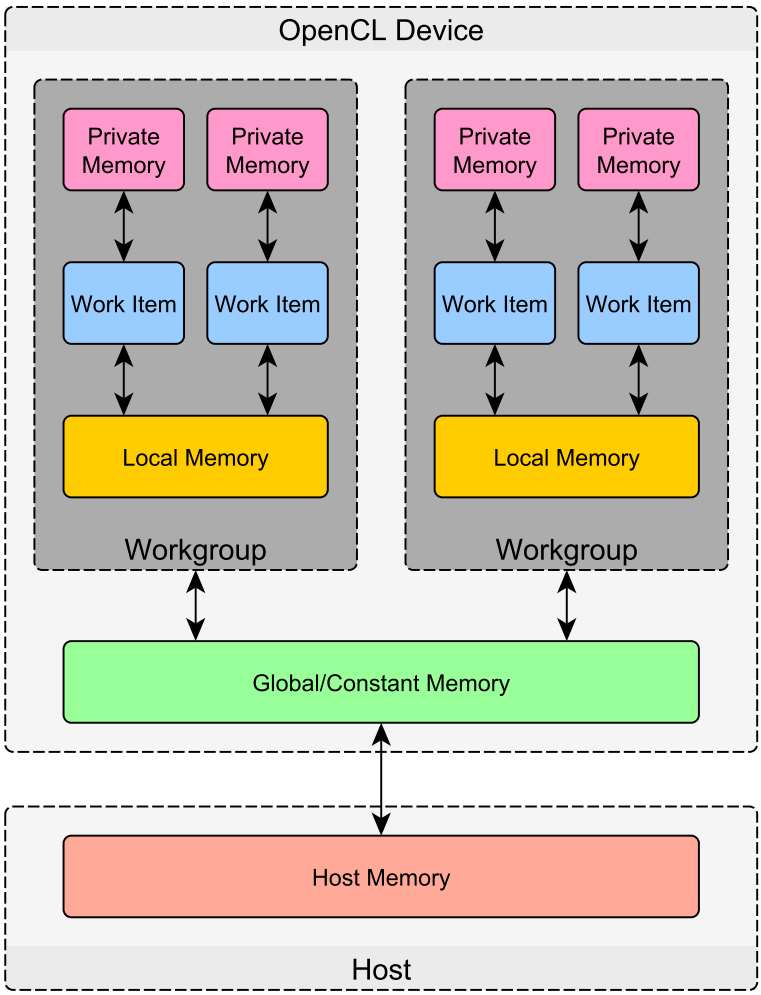


Figure 5.1: The OpenCL memory model

Chapter 6

Implementation

In this chapter, we discuss details the of our implementation of the improved MLVQ-based compression algorithm.

6.1 System Architecture

In order to offer high modularity our implementation is divided into several separate components as shown in Figure 6.1. The compression and decompression parts of the algorithm are treated separately and implemented by individual sets of components.

6.1.1 Common Functions Library

Some functionality, such as command line arguments parsing, OpenCL context management or configuration files handling is common for different components of the framework. For this reason a separate library of common functions exists which the individual components can use. Additional features covered by the library include error handling, message logging, coordinate space conversions or file system access.

6.1.2 Compression-Related Components

The compression algorithm is split into three main phases, represented by three separate applications as shown in Figure 6.1. The first application, *Preprocessor*, is used to transform the raw input data from different formats into a single common format, which is then used as input to the next phase. The second application, *Resampler*, reads the preprocessed input data and resamples them into the *Onion-Slices* parameterization. In the third and last phase, the resampled data are read by the *Compressor* application, which performs the MLVQ compression and saves the resulting compressed BTF to a file.

The reason for this workflow is to separate individual parts of the compression algorithm on high level and allow for reusing of data. For example the *Preprocessor* component is usually only used once to transform the raw input data into the required format. Once the data are preprocessed, the raw input data can safely be deleted, since no other component uses them during compression. Similarly, when the preprocessed data are resampled into

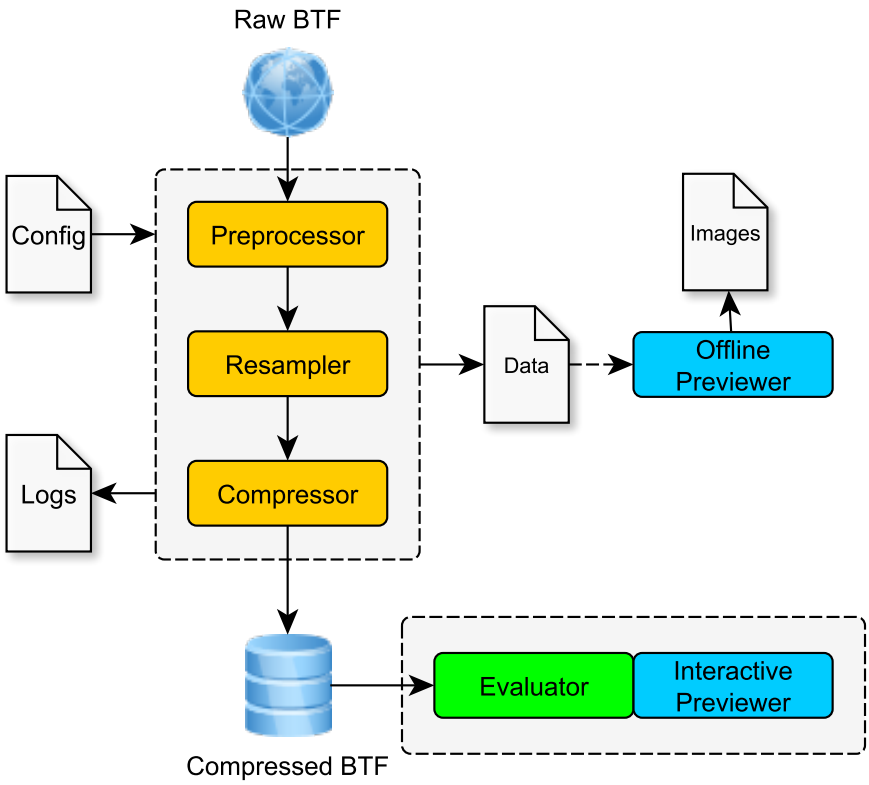


Figure 6.1: Software architecture of our implementation

the *Onion-Slices* parameterization, they can be reused by the *Compressor* component many times with different settings, without the need to resample them again.

6.1.3 Decompression Library

The decompression algorithm was implemented as a separate library named *Evaluator*. The library can be easily incorporated into custom applications to provide them the functionality needed to use the compressed BTF data. The library handles all the necessary operations, from reading the compressed material file to performing the BTF evaluation at specific input coordinates and returning the resulting reflectance. The decompression algorithm is implemented in OpenCL and allows fast parallel execution either on a multi-core CPU system or on a GPU.

6.1.4 Preview Generation Tools

To demonstrate the use of the decompression library, an interactive preview generation application was built using the library. The application is similar to the one presented in [Ege13]. The BTF evaluation is incorporated into standard OpenGL rendering workflow

and allows the user to navigate through a scene containing a BTF-mapped object in real-time.

Additionally a non-interactive preview generation tool was implemented. Compared to the interactive application, this tool can use multiple different BTF coordinate sources (predefined patterns, data read from files, OpenGL rasterization, ...) and evaluate them using multiple different techniques, including BTF evaluations from intermediate data representations (for example *tempBTF* or *onionBTF* files, Section 6.1.6). This allows the results from the intermediate steps to be visually inspected and compared to both the original and the compressed data.

6.1.5 Configuration Files

A major part of the application is user configurable, including the definition of the processing pipeline itself. The configuration for each subsystem is stored in one or more XML files, which is processed at run-time. Individual configuration files are allowed to include other files. This allows the user to split all the settings into several smaller files, some of which can be reused later on without the need to modify them. Several predefined markers can also be used within configuration files, for example the current working directory or values of command line arguments passed to the application. This allows for the same configuration file to be used during batch processing of several BTFs without the need to edit the file for each material individually. To query the configuration files from within the application, the XPath query language is used.

6.1.6 Custom File Formats

Custom file formats were created to store results from each of the basic processing steps. The file formats are binary and were designed for efficient exchange of data between the individual applications performing the processing steps as shown in Figure 6.2. Summary of the formats used is provided in Table 6.1.

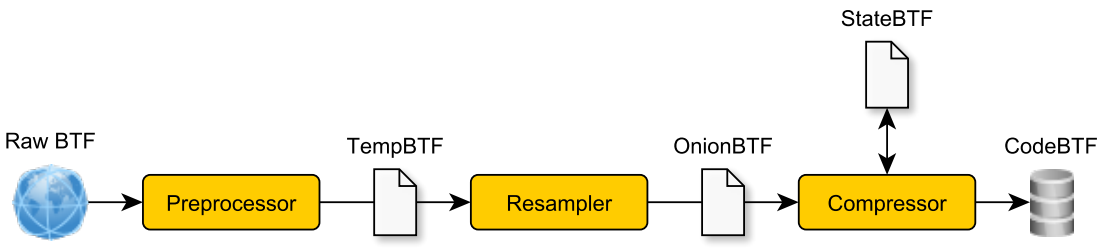


Figure 6.2: File formats used to pass data between individual components of the compression framework

Format	Generated by	Content
tempBTF	Preprocessor	Raw input data arranged in $[x, y]$ major layout
onionBTF	Resampler	BTF data in *Onion-Slices* parameterization
codeBTF	Compressor	Compressed and possibly optimized BTF data
stateBTF	Compressor	Snapshot of the compression pipeline state

Table 6.1: Custom file formats used during the compression

6.2 Input Preprocessing

Processing of the raw input data is a two step task. In the first step, the data are converted from different input formats into the strictly defined *tempBTF* format. From there, the resampling into the *Onion-Slices* parameterization is performed. The data are then ready to be compressed using the MLVQ algorithm.

6.2.1 Raw Input Preparation

Preparation of the raw input data is handled by the *Preprocessor* component. This component transforms the raw input data in various formats into a single common format, referred to as *tempBTF*, which is then used as the input to the *Onion-Slices* parameterization resampling algorithm. The raw input data can be stored in different layouts, data formats and precisions. Preprocessor is used to isolate the next compression stages from these variations by processing the raw data into a single strictly defined format.

For example the data from UBO2003 [SSK03] and UTIA [HFV12] data sets are in the raw form stored as 6561 individual images in either JPEG or PNG format, as demonstrated in Figure 6.3. Each image contains values for a *single combination* of θ_V, ϕ_V, θ_I, ϕ_I, for *all BTF texels* ($[\theta_V, \phi_V, \theta_I, \phi_I]$-major layout). For resampling into the *Onion-Slices* parameterization, values of *all combinations* of θ_V, ϕ_V, θ_I, ϕ_I, are required for a *single BTF texel* ($[x, y]$-major layout). If kept in their raw form, all images in the dataset would need to be decoded and accessed only to resample a single texel, which would make the resampling algorithm inefficient.

Preprocessor reads the raw input data from all the individual image files, transforms them into the required $[x, y]$-major layout and stores the result as a *tempBTF* file, which can then be efficiently used by the resampling algorithm.

The *Preprocessor* system consists of the following three components:

RawBTFReader
: Reads the raw input data in various formats. A reference implementation is provided which reads the raw datasets in the UBO2003-like [SSK03] format.

Reshaper
: Changes the layout of the input data into the required $[x, y]$-major parameterization.

TempBTFWriter
: Writes the resulting data to disk in *tempBTF* format.

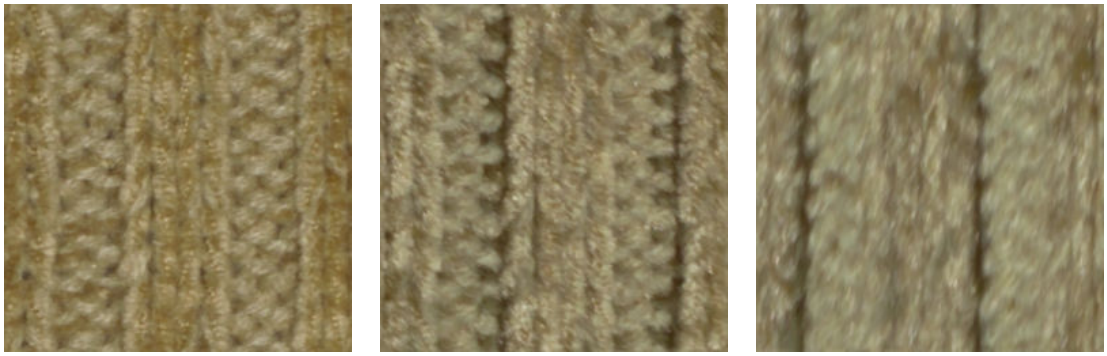


Figure 6.3: Example of 3 of the total 6561 input images (BTF measurements) captured for the *Pulli* BTF material. *Images courtesy of [SSK03]*

6.2.2 Onion-Slices Parameterization Resampling

The *Resampler* component handles the resampling of the preprocessed BTF data into the *Onion-Slices* parameterization, which is required by the MLVQ compression algorithm. The basic workflow of the *Resampler* operation starts by reading a batch of raw BTF texel data from the input *tempBTF* files. This batch of data is then processed texel-by-texel and resampled into the new parameterization using one of the interpolators provided. Finally, after all texels of the material get processed, the resulting resampled data set is stored to disk as a set of files in the *onionBTF* format.

The number of coordinates (n_{θ_V}, n_{ϕ_V}, n_α, n_β) used to discretize the BTF in the *Onion-Slices* parameterization is user configurable. Based on the configuration, a set of coordinates is generated for each possible combination. These coordinates are then used as query points for the interpolator. Using the input data as reference, the interpolator then approximates the BTF value at the requested coordinates. An example of the interpolated data for five of the BTF texels is provided in Figure 6.4.

The *Resampler* allows different interpolation methods to be used depending on configuration. Each of these methods is implemented as a separate class following a predefined interface. Three different methods were implemented – a simple nearest-neighbor search and two *Radial Basis Function* interpolators, one CPU and one GPU-based.

The RBF interpolator uses Gaussian Radial Basis Function interpolation [CBC+01] to better approximate the value at the query point. RBF interpolation is computationally expensive and its complexity depends on the number of reference points used. To reduce this computation time, a small number (i.e. 8) of data samples nearest to the query point is first found using a k-Nearest-Neighbor search algorithm. Only these data samples are then used as input to the RBF interpolator to evaluate the query. To solve the final RBF matrix, Cholesky decomposition is used. After solving the matrix, the RBF is evaluated and the interpolated value for the given query coordinates is obtained.

Because many query points can be evaluated in parallel, the algorithm is well suited for GPU use. The OpenCL implementation follows the same basic steps as the CPU implementation. The k-Nearest-Neighbor search is first performed on the CPU, because the total number of reference points is relatively low (6561 points for UBO2003 [SSK03] data set).

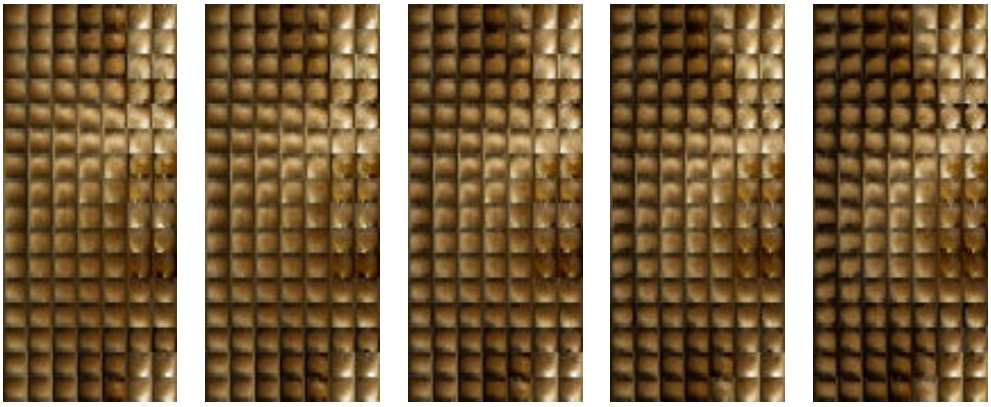


Figure 6.4: Example of five texels of the *Corduroy* BTF resampled into the *Onion-Slices* parameterization.

The search is also only performed once for the whole dataset, because the reference and query point coordinates remain the same for the whole dataset. Having the set of k nearest neighbors for each of the query points, each query can then be processed by a single thread. The thread first prepares the RBF matrix, solves it and then uses the result to obtain the final set of values for the given query point.

The *Resampler* system consists of these basic components:

TempBTFReader
Reads the input data in *tempBTF* format.

Interpolator
Performs the resampling of the input data into the *Onion-Slices* parameterization.

OnionBTFWriter
Writes the resulting data to disk in *onionBTF* format.

6.3 Compression Process

The MLVQ compression is performed by the *Compressor* application. This application is responsible for assembling the dynamic compression pipeline based on its configuration and feeding the input BTF data into this pipeline. As a result, the compressed BTF data are stored within the codebooks of individual pipeline nodes, from which they can be read and saved to disk as the resulting compressed material dataset.

6.3.1 Pipeline Node Communication

To pass data between individual nodes and compare units in the pipeline an approach based on a request-response model is used. A single request is represented by a *task*. Each task contains only a single value, which is the index of the first element the given node should operate on. Combined with the data region specified for each node, the node is provided complete information on which part of the input data to operate.

A *token* represents the data returned in response to a single task. It consists of a set of values the node will later need in order to reconstruct the data processed while executing the corresponding task. This set of values typically consists of a row index in the node's codebook and values required by the node's modifiers (for example the scaling coefficient).

This task-token model provides a unified interface to communicate between individual parts of the pipeline. This allows for the pipeline to be completely user configurable. A parent node sees the tokens of its child node only as data chunks with given size. The meaning of the data is not known to the node and in fact is not required to be known. The sole purpose of the node is to provide the data back to its child during the reconstruction phase. This allows for the use of modifiers, since only the size of the token is relevant to the parent node, not the internal structure of the token.

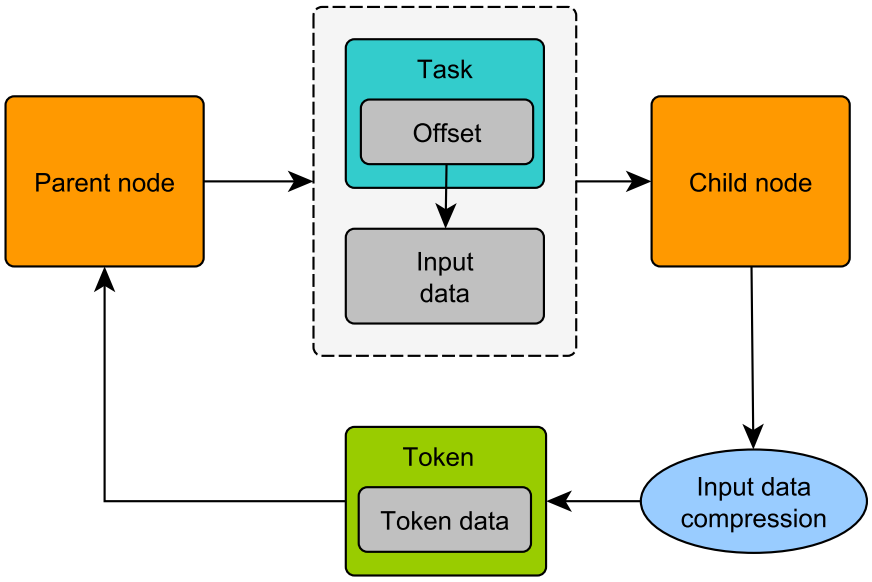
The workflow of task-token communication is illustrated in Figure 6.5. The node always receives a set of tasks as its input and is obligated to generate a matching set of tokens as the output. The actions taken to produce the tokens depend on the type of the node. Nodes with attached compare units can pass the set of tasks to the compare units and try to find matching entries within the codebooks. If a matching entry is found, its index is returned as part of the token for the given task. If no matching entry is found for the given task, the node can take other actions to produce the token, for example by creating a new set of tasks for its child node and later storing the returned tokens as a new codebook entry. For a single task, a node can generate several sub-tasks, processed by children of the node. It is then the responsibility of the parent node to process the tokens generated from the sub-tasks and return a single token as a response to the original input task.

6.3.2 Memory Layout

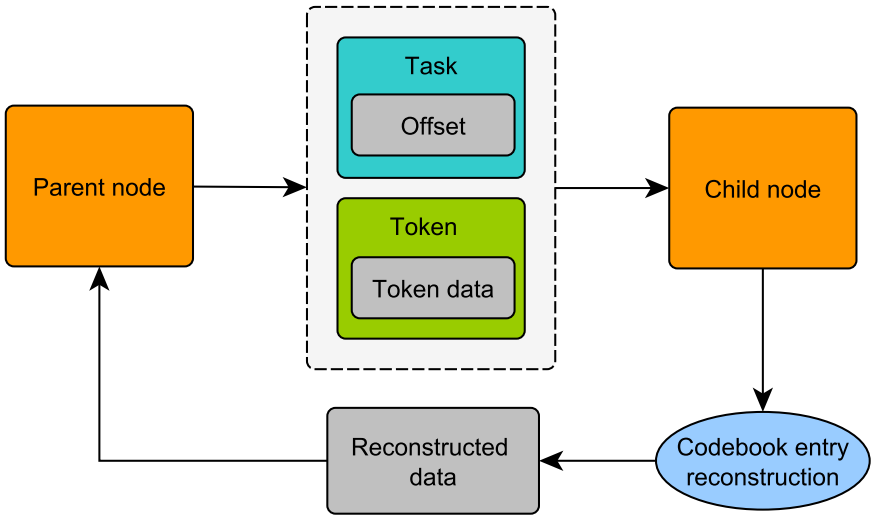
The memory layout of the *Compressor* application is demonstrated in Figure 6.6. The memory regions can be divided into three basic categories. The first category contains blocks of memory common for the whole *compressor* application. These regions are accessible by any component during the compression. The *window* area is used to store the input BTF data for the currently processed BTF texel and its neighborhood. The *reconstruct* memory region is used to store blocks of BTF data reconstructed from individual codebook entries during the search for similar entries. A compare unit compares blocks of data in the *window* region with blocks of data in the *reconstruct* region. A compare unit can also use additional blocks of memory to store its own data. These form the second category and are described in detail in Section 6.3.2.2. Finally, each of the nodes present in the compression pipeline has its own set memory regions, which form the third and final category and are discussed in Section 6.3.2.1.

6.3.2.1 Pipeline Node Memory Regions

The memory layout of a single pipeline node is shown in Figure 6.7. Each node present in the pipeline has at least three own memory regions – one to store the incoming *tasks* for the node, one to store the resulting *tokens*, and one to store the number of tasks (*tasks count*). The node also gets connected to similar memory regions in other nodes to perform the tasks-tokens based communication as described in Section 6.3.1. Each node is provided access to



(a) Compression



(b) Reconstruction

Figure 6.5: Principles of task-token communication during compression and during reconstruction phase.

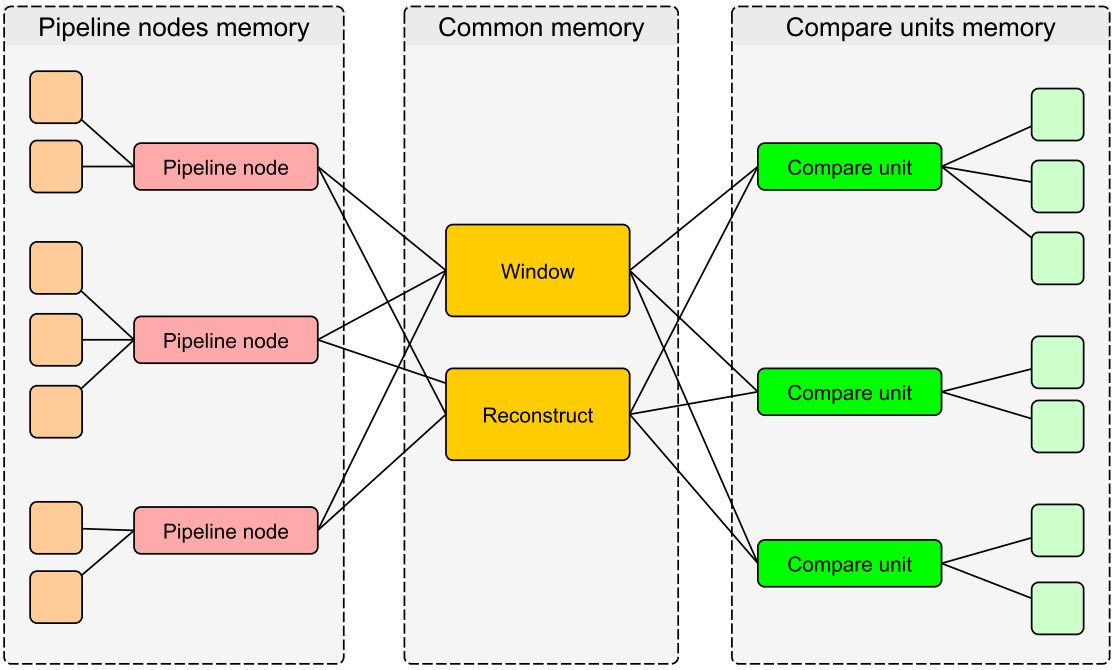


Figure 6.6: General memory layout of the *Compressor* application

the common *window* and *reconstruct* memory regions. Codebook-based nodes also contain a separate memory region for the codebook itself. If caching (Section 6.3.6) is enabled, additional memory regions get created for the node. The amount of memory allocated is not limited by the framework, so additional regions may be created if required by the given node type.

6.3.2.2 Compare Unit Memory Regions

Memory regions used by a single compare unit are demonstrated in Figure 6.8. The *tasks* and *tasks count* regions are used by pipeline nodes to communicate with the compare unit. Additionally, the information about the the data region for which the comparison should be performed must be provided to the compare unit in the *data region* memory.

After performing the comparison, the resulting similarity for each of the tasks gets stored in the *results* memory region. As comparison with multiple reconstructed entries can be performed in parallel for a single task, indices of the best matching entries are stored in the *best rows* memory. This means that for a single task an index of the best matching entry gets stored in the *best rows* memory and the similarity between the input data and this entry gets stored in the *results* memory region.

The compare unit implementation is allowed to use additional memory regions if required. For the SSIM-based compare unit implementation custom memory was used for example to store the precomputed means and variances of values within the SSIM window.

Additional memory may also be used for compare units operating with *modifiers*. For example for the *scaling* modifier, the average luminance of the the input BTF data blocks can

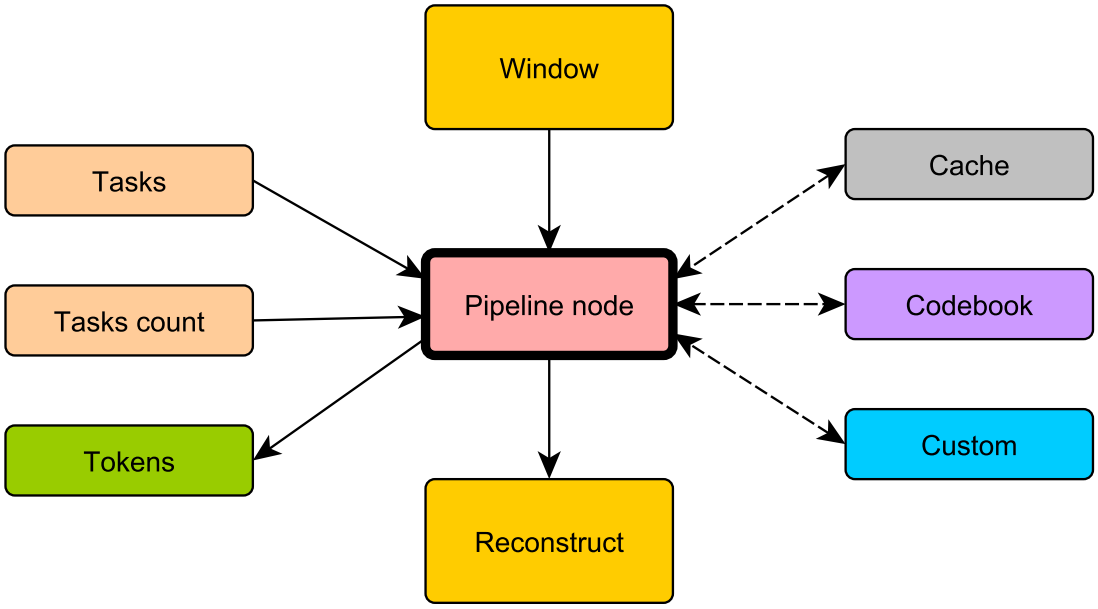


Figure 6.7: Memory regions used by a single pipeline node. The typical flow of data between the node and the memory region is indicated by an arrow.

be precomputed and stored in a custom memory region to avoid unnecessary recomputation of these values during comparison with each of the reconstructed codebook entries.

6.3.3 Compression Algorithm Workflow

The general outline of the MLVQ-based BTF compression algorithm was provided in Section 3.6. In this Section we describe how this algorithm was implemented in our application.

6.3.3.1 Pipeline Assembly

The compression algorithm begins by reading the configuration files. Using this configuration, the reader for the input *Onion-Slices* resampled data in *onionBTF* files is created and basic properties of the material, such as the discretization used, is read. In the next phase, basic initialization of the OpenCL platform is performed, followed by the initialization of the compression pipeline. The pipeline configuration is first read from the configuration files. The configuration includes the nodes which will form the pipeline, their types, basic properties and interconnections with other nodes, as well as the compare units available for the nodes to use. Based on the configuration, the nodes and compare units get created. Finally, the parent-child relationships between nodes get resolved.

In the next phase, the nodes performing split operations are informed about the input dimensions, in which they should perform the split. Knowing this information, the input regions for each node can be resolved. Beginning at the root node, the input region for the node covers the whole block of input data (the whole BTF texel). In this block, data for all

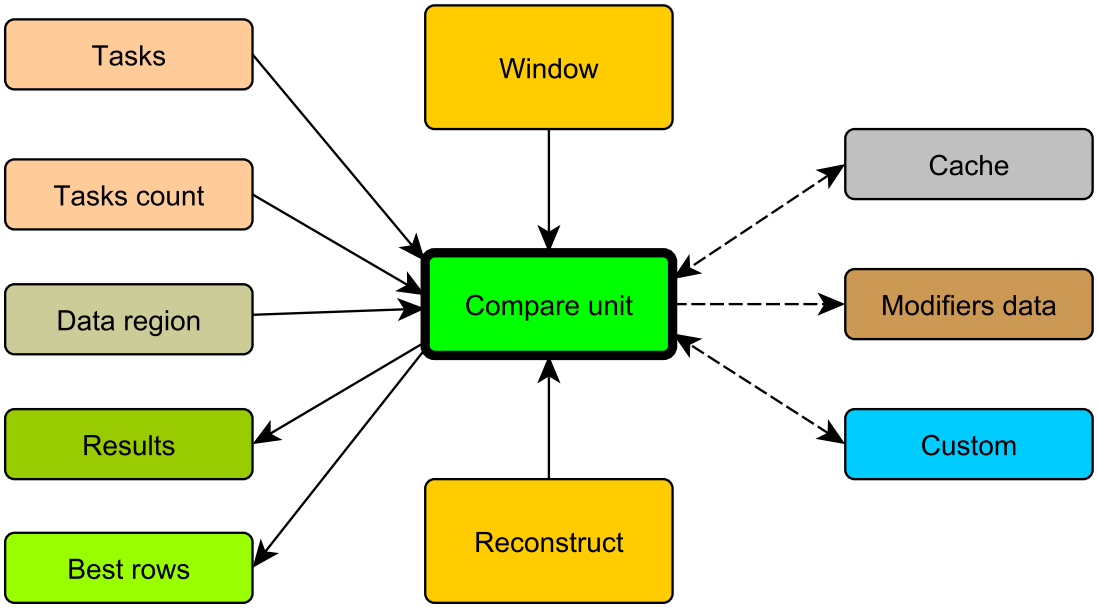


Figure 6.8: Memory regions used by a single compare unit. The typical flow of data between the compare unit and the memory region is indicated by an arrow.

four dimensions (θ_V, ϕ_V, α, β) of a single BTF texel are present. Each node performing a split operation removes one dimension from the region. This means that all nodes following the first split node will operate on three-dimensional (ϕ_V, α, β) regions. After the second split, the input region is further reduced to only two dimensions (α, β) and so on. The order of dimensions in which the region gets split is fixed as (θ_V, ϕ_V, α, β), to be consistent with the pipeline used by [HFM10]. The consequences of using different split order are currently unknown and may be an interesting topic for future work.

The pipeline initialization continues by assigning compare units to all nodes requesting them. A single compare unit may be used by multiple nodes. As the final phase, OpenCL memory regions and execution kernels are created for each node. The number of memory regions and kernels depends on the type of the node. After compiling the kernels and assigning the memory regions to them, the initialization of the pipeline is complete.

6.3.3.2 Stages Processing

After initializing the pipeline, compression stages options get read from the configuration and the first stage is set as active. Based on the properties, indices of the BTF texels to be processed during the stage are first generated and stored into the processing queue. After that, the MLVQ compression algorithm itself begins.

First a batch of BTF texel indices ([x, y] coordinates] is extracted from the processing queue. The size of the batch is user configurable and depends primarily on the resources available on the system performing the compression (mainly the available memory size). For the whole batch the input *Onion-Slices* resampled data are read. This includes the data for

the BTF texels themselves and as well as the data of the surrounding BTF texels forming the input windows.

The BTF texels in the batch then get processed one by one. First, the input window is assembled for each BTF texel and uploaded to the corresponding *window* memory region. The compression pipeline is then executed. After compressing the texel, input window for the next one in the batch gets assembled and the pipeline gets executed for the texel. These steps are repeated until all texels in the batch get compressed. After that, texel indices for the next batch are extracted from the processing queue and the next batch gets processed in the same way.

After all BTF texels are processed for the given stage, the next one begins. Configuration for the stage is first applied to all components. Then the processing queue gets filled with the coordinates of the BTF texels for the stage. The texels are then again processed in batches, until the whole queue gets processed. After all stages get processed, the main compression algorithm is finished and data for all[1] texels should be present within the codebooks of the pipeline nodes.

6.3.3.3 Similar Entry Search

The ability for a node to search and return similar entries is the key concept of the vector quantization algorithm. The search for a matching entry in the codebook can be terminated by two different methods, configurable by the user. In the first method, the first similar-enough (based on the configured threshold) entry encountered is returned. This allows the search to be terminated as soon as the entry is found, but does not guarantee, that the entry returned is the best matching from the codebook. The second method will always look for the entry best matching the input data. While this method should provide better visual quality of the result, it is also more computationally demanding, since all the entries in the codebook must always be reconstructed and compared. Because both methods have their pros and cons in different stages of the compression algorithm, the preferred method can be selected for each node on a per-stage basis.

It is worth noting, that even when an entry best matching the input data is found within the codebook, it is not guaranteed, that the entry is similar-enough based on the configured threshold.

When a similar-enough entry is found by either of the aforementioned methods, a token identifying the entry is returned and the size of the codebook remains unchanged. If a matching entry is not found, a new one is created in the codebook. The entry is then filled by the corresponding data, either a set of tokens for non-terminal nodes or a vector of representative values for terminal nodes.

The node can also be configured to *lock* the codebook on a per-stage basis. When a codebook is locked, no new entries can be added into it and only those already present in the codebook can be used. This means, that the best matching entry is returned even if it is not similar-enough based on the current similarity threshold. Because the size of a locked codebook remains constant, this allows the user to limit the resulting data size of the

[1] It is possible to configure the stages in such way, that not all texels of the BTF get processed. This is useful mainly for debugging purposes. For example only a small number of texels can be compressed in very short time to verify correct configuration of the processing pipeline.

compressed material. An opposite approach, where the node does never search for similar entries and always creates new entries in its codebook, can also be used, although it is mainly beneficial for debugging purposes.

6.3.3.4 Codebook Entry Reconstruction

In order to compare a block of input data with an entry already stored in a codebook, the entry needs to be reconstructed back to its original form (to a block of data in the Onion-Slices parameterization). The data reconstruction algorithm can be initiated from two different sources – either by the node itself during the search for a similar entry within its codebook or using a set of tasks provided by a parent node. Regardless of the source, the reconstruction algorithm follows roughly the same steps.

The index of the entry to be reconstructed is either provided to the node in form of a token by the parent node or, when the node itself is initializing the reconstruction, generated by the node. The workflow is then similar as during decompression, described in Section 3.7. The key difference is that input coordinates do not get used as indices into the codebook entries. Instead, the decompression is performed for all pieces of data in the entry. This means, that if the entry for example consists of n_{θ_V} tokens, then n_{θ_V} reconstruction tasks are created for the child node.

The process recursively repeats until reaching terminal nodes. Here, the raw representative values are copied to their appropriate locations in the *reconstruct* memory region. Returning back to the node requesting the reconstruction, the data can be further transformed for example by the means scaling coefficients or color space transformations. After returning to the source node, reconstructed BTF data will be present in the *reconstruct* memory region starting from the location given by the original task.

6.3.3.5 Workflow of a Single Node

The workflow of a non-terminal node usually consists of four main steps: *Compression forward*, *Compression return*, *Reconstruction forward* and *Reconstruction return*.

Compression Forward Step

During the *Compression forward* step, the node receives tasks to process some input data blocks. The node does the processing and, as a result, some tasks for its child nodes might be generated. For example the *Transform* node type converts the data into different color space and orders its child node to continue the compression on these newly processed data. The *Compression forward* step ends by the node ordering its child node to further process the data (the *Compression forward* step gets started for the child node).

Compression Return Step

The *Compression return* step takes place when the child node is done processing the data and returns the control back to the calling node. The tokens resulting from the child node operation are now stored in the *tokens* memory region of the child. The purpose of this step is to allow the active node to process the results generated by its child node. For example, if the active node is of type *FindOrSplit*, it will use the

Compression return step to store the tokens generated by the child node to its own codebook.

Reconstruction Forward Step
The *Reconstruction forward* step is similar to the *Compression forward* step, but is used during data reconstruction. The input to this step is a task to reconstruct a block of data. The node reacts to this task by sending reconstruction requests to its children. For example the *FindOrSplit* node uses the input request (consisting of a task and token) to look up the corresponding entry in its codebook and based on the entry produces several reconstruction tasks for its child node.

Reconstruction Return Step
The *Reconstruction return* step roughly corresponds to the *Compression return* step. It takes place when the child node has processed all the reconstruction tasks and returns control to the calling node. The calling node can use this step to further process the input data before returning control to its ascendant. For example the *Transform* node uses this step to convert the data being reconstructed back to their original color space. Also the results of modifiers (for example scaling coefficient) usually get applied during this step.

6.3.3.6 Compressed Data Output

Before saving the resulting compressed data to a file on a disk, several additional optimizations can be performed, as described in Section 6.3.7. When saving the results, first the parameterization of the *Onion-Slices* parameterization used gets stored, followed by the pipeline configuration. This includes the types of nodes, their properties and interconnections. All these information are later used to rebuild the pipeline during the decompression phase. Finally, the (possibly optimized) content of individual nodes' codebooks gets stored. As a result, a single stand-alone file is generated, which can be used by the *Evaluator* without any additional information. After writing the file, the whole compression is finished.

6.3.4 Main Compressor Components

The *Compressor* system is composed of several components, each with its specific purpose. Overview of the individual components is provided in this Section.

6.3.4.1 OnionBTF Reader

The *OnionBTFReader* component is used to read the input data in *Onion-Slices* parameterization from *onionBTF* files. It is mainly used by the *Window Manager* component to prepare the data to be compressed into the input *window* memory region.

6.3.4.2 Window Manager

The *Window Manager* component's main responsibility is to prepare the input BTF data for a BTF texel and upload them to the corresponding OpenCL memory region, from which

they can be used by the rest of the pipeline. As the similarity metric may operate not only on the currently processed texel, but also on its surrounding, the *Window Manager* creates a window consisting of data for the whole surrounding. The size of the window is user configurable. For our final measurements, we used window size of 11×11 BTF texels.

For the similarity measure to operate correctly, the importance of individual BTF texels within the window is weighted using a Gaussian kernel. The currently processed texel, situated in the centre of the window, is considered the most significant, while the texels towards the edge of the window are considered progressively less significant.

6.3.4.3 Cell Manager

The *Cell Manager* component is used to track states of individual BTF texels during compression. Tracking the state ensures that in the end, all texel are processed, but none of them gets processed twice. Each texel can at a given time be in one of the following states:

Available
: The texel needs to be compressed, but has not yet been scheduled for processing.

Reserved
: The texel needs to be compressed and is scheduled for processing in the current processing stage.

Processed
: The texel has been successfully processed and is available within the compressed data.

After starting a new compression from scratch, all BTF texels are considered to be in the *available* state. During the beginning of each compression stage, a batch of texels in the *available* state is reserved to be processed during the stage and their state is changed to *reserved*. After the texel gets successfully compressed, its state is changed to *processed*.

6.3.4.4 Cell Queue

The *Cell Queue* component is used to maintain the order, in which individual BTF texels will be compressed. At the beginning of each stage, a given number of *available* texels is selected from the *Cell Manager*. These texels are then processed one-by-one until the whole stage is finished. The number of texels processed and their order is configurable per-stage. The processing order can be either sequential (row-by-row), completely random, or quasi-random (using Halton sequence [Hal64]).

6.3.4.5 Compare Units

Each compare unit in implemented as a separate class with a defined interface. Similar to pipeline nodes, multiple types of compare units can be present in a single pipeline. A SSIM-based compare unit is provided in our code as a reference.

A compare unit allows parallel execution, which means that N blocks of original data can be compared with N blocks of reconstructed data in a single run. As in the rest of the

pipeline task-based communication is used to pass data to and from a compare unit. For each block of input data, the compare unit outputs the index of the best matching block of reconstructed data and the resulting similarity index for the given combination. Modifiers described in Section 4.4.5 can also be applied to compare units to directly produce additional data such as scaling coefficients.

To optimize the process, early termination of the comparison task can be performed. If at some point during comparison of an entry it is clear, that the resulting similarity index would be worse, than the already found best similarity index with a different entry, processing of the rest of the current entry is skipped. This also means that the better the currently-found best-matching entry is, the quicker the rest of the entries will be processed.

6.3.4.6 Pipeline Nodes

Each node type is represented by a separate class following a defined interface in the code. The classes also share a common ancestor based on the node type category. The factory programming pattern is used in the implementation to generate individual instances of pipeline nodes at run time. Overview of the implemented node types is provided in Section 6.3.5.

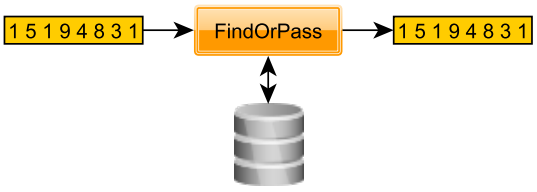
6.3.4.7 Pipeline

The *Pipeline* subcomponent is responsible for assembling and managing the compression pipeline. The *Pipeline* creates all nodes and compare units based on the configuration and maintains information about their properties and interconnections during the compression. It also keeps track of the discretization used for the material.

6.3.5 Pipeline Node Types

In this section, the overview of pipeline node types created for our implementation is provided.

6.3.5.1 *FindOrPass* Node Type



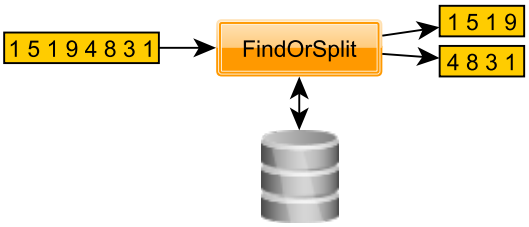
The *FindOrPass* node type belongs to the *nodes with a compare unit* category. Nodes of this type first try to find an entry within their codebook similar enough to the input data. If such an entry is found, its index is returned in the resulting token. If a similar-enough entry is not found, the whole input data block is passed to a child node. This node type is non-terminal. A single entry in a codebook of a node with this type consists of a single

token returned by its child node. *FindOrPass* nodes are mainly used to perform a search for similar entries in one color space, before transforming the input data to a different color space and processing them further by another node.

6.3.5.2 *FindOrSave* Node Type

The *FindOrSave* node type belongs to the *nodes with a compare unit* category. Similarly to the *FindOrPass* type, nodes of this type also try to first find a matching entry within their codebook. The main difference being that *FindOrSave* nodes are terminal. This means, that if a similar enough entry is not found, the block of input data on which the node operates gets stored as a new entry in the node's codebook. *FindOrSave* nodes are therefore used to store the representative vectors of BTF data.

6.3.5.3 *FindOrSplit* Node Type

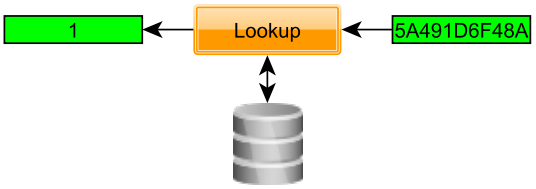


The *FindOrSplit* node type is very similar to the *FindOrPass* node type. The main difference between these two types is the action taken when a similar enough entry is not found within the node's codebook. While a *FindOrPass* node passes the whole block of input data to its child, the *FindOrSplit* node splits this data block into several smaller regions. A separate task is then generated to process each of these sub-regions by the node's child. After the child node is done processing the tasks, a token is returned for each of the sub-regions. All these tokens then get stored as a single entry in the *FindOrSplit* node's codebook.

Splitting the input region into several smaller sub-regions is the key part to the concept of levels in the MLVQ algorithm. Each *FindOrSplit* node creates a new level, in which the input data for the whole BTF texel get split into progressively smaller sub-regions. The number and size of the sub-regions is assigned to the node according to the number of steps used in the material discretization. As described in Section 6.3.3.1, the order in which the dimensions get assigned is fixed as (θ_V, ϕ_V, α, β). This means, that the first *FindOrSplit* node encountered during the compression will split the input data for the whole texel into n_{θ_V} sub-regions, effectively reducing the dimensionality of the data by one. If such a sub-region

gets processed by another *FindOrSplit* node, it will be further split into n_{ϕ_V} sub-sub-regions and so on.

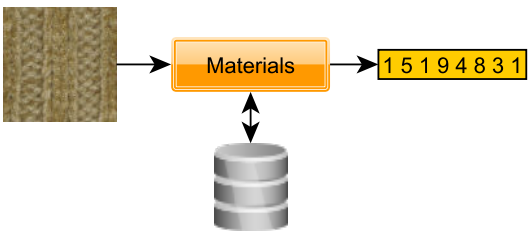
6.3.5.4 *Lookup* Node Type



The *Lookup* node type belongs to the *nodes with codebook* category. The purpose of this node is to reduce complexity of tokens obtained from its child node. Each token the node receives from its child node gets stored into the codebook. The resulting codebook entry is then identified by only its index, which gets returned as token to the parent of the *Lookup* node. This node type is transparent during the compression, meaning that it simply passes the input tasks to its child node and operates only with the resulting tokens.

The *Lookup* type of nodes has little use in a production compression pipeline configurations, because its codebook only occupies space in the compressed data, without providing any additional functionality. Attaching a node of this type to a pipeline can however provide useful debugging information. During development, it also served as a starting point for other node types.

6.3.5.5 *Materials* Node Type



The *Materials* node type is a specialized version of the *Lookup* node type. The *Materials* node is responsible for storing information about the BTF material being compressed. Each material stored within the node is identified by its index and has a name, width, height (in texels) and an offset to the beginning of the material data within the node's codebook.

At the beginning of the compression phase, the pipeline manager informs the *Materials* node about the dimensions of the material being compressed. The node allocates space within its codebook, into which a single token will be stored for each of the BTF texels. The coordinates of the texel are provided to the *Materials* node by the pipeline manager. After

the compression of the given texel is finished, the resulting token gets stored onto the given location in the *Materials* node's codebook.

A single node of this type is automatically added as a root node of the compression pipeline and cannot be used in any other part of the pipeline.

6.3.5.6 *SplitChannels* Node Type

The *SplitChannels* node type belongs to the *basic nodes* category. Nodes of this type can be used to separate processing of individual color channels of the input data block. The number of outputs and their layout is user configurable for each *SplitChannels* node. For each of these outputs, the number of channels sent to the output and a corresponding target node must be specified. The number of channels output can be greater than one, which allows using different processing paths for different parts of the data block. For example the luminance component (one channel) may be separated from the chrominance component (two channels) and processed using a separate path in the compression pipeline.

During pipeline initialization phase, the *SplitChannels* node informs its children about the color channels on which they should operate. When processing a compression task, the node first forwards the task to all its child nodes. After all the child nodes are done processing their corresponding tasks, the *SplitChannels* node concatenates all the tokens received from the children into a single group token, which then gets returned to the parent node as a response to the compression task. During reconstruction, the workflow is reversed - the *SplitChannels* node receives a single group token, divides it into individual sub-tokens, assigns the sub-tokens to the children nodes and then instructs them to process these sub-tokens.

6.3.5.7 *Transform* Node Type

The *Transform* node type belongs to the *basic nodes* category. The purpose of the *Transform* node type is to perform transformations of the processed data between different color models. During compression a *forward* transformation is performed. This transformation is applied to all BTF texels of the input window, but only in the regions the node operates on. During reconstruction a *backward* transformation is performed on the reconstructed data to convert them back to the original color model.

The type of forward and backward transformations performed is user configurable. This allows the user to define different transformation functions for example when performing compression of a HDR material. The *Transform* node type currently supports transformations in and out of YCbCr color model [ITU601] (used for LDR materials) and LogLuv [Lar98] color model (used for HDR materials) as described in Section 3.5.

The transformation is parallelized by executing a single thread to perform the required set of actions for a single value index (all values with the same coordinates in the input window or a single value with the given coordinates in the reconstruction memory).

6.3.6 Caching Mechanisms

In order to find a codebook entry best matching the input data, all entries within the codebook need to be reconstructed and compared to the input data. This gets repeated for every single BTF texel being compressed. Because the codebook entries are immutable and the same blocks of data get reconstructed for them regardless of the input texel being processed, several optimizations can be performed.

6.3.6.1 Comparison Order Cache

This optimization is based on the early termination optimization of compare units described in Section 6.3.4.5. The sooner a highly matching entry is found during the comparison process, the faster the rest of the entries get processed. The purpose of this cache is to maintain the order in which individual entries of a codebook should be processed for the early termination optimization to get used the most. The cache is based on the least-recently used paradigm, meaning that the best matching entries from the last step will be compared first. If no match is found, best entries from the step preceding the last step will be compared and so on.

6.3.6.2 Reconstructed Data Cache

This optimization is based on the fact, that the data reconstructed from an entry are always the same, regardless of the number of times they get reconstructed. Without this cache, the reconstruction algorithm involves requesting the data from a child node, which might recursively request data from its child node and so on. *Transform*-type nodes can also be encountered on the reconstruction path, always converting the same data from one color model to another. To eliminate this unnecessary overhead, the reconstructed data for each codebook entry can be cached. To perform a comparison, the cached data can then be used directly.

Since of the amount of space the cache can occupy for a single node might be high, this cache can be selectively enabled on a per-node, per-stage basis, depending on the amount of memory available on the target platform.

6.3.7 Compressed Data Postprocessing

After finishing the main MLVQ compression, each codebook-based node in the pipeline already has its codebook filled with all the required data. These data are present in their

raw form, being stored in a format, in which they can be easily used by the *Compressor*. Because the data will no longer be used by the *Compressor*, several optimizations can be used to further reduce the size of the data, before using them in the decompression.

Because the node storing the data usually does not know the meaning of the data stored within its codebook (only that they consist of tokens provided by its child node), the post-processing works by providing the stored data back to the child node. The structure of the data is known to the child node, which can therefore effectively perform the postprocessing. When finished, the child node returns the optimized data set back to its parent. During decompression, this workflow is reversed - the parent node provides the optimized data set to the child node, which transforms it back to the format the parent node can work with.

6.3.7.1 Floating-Point Values Quantization

All floating point values (raw values in terminal nodes, scaling coefficients etc.) used during compression are represented using the IEEE-754 [IEEE754] 32-bit floating point format. Because only a limited range of values is used (typically in the 0.0 - 1.0 range), these values can be optimized by quantization into a smaller width, fixed precision data type.

For each set of input values, the minimum and maximum value is found. The interval between the minimum and maximum is scalar quantized into 2^k discrete values for k bits of storage. The number of bits depends on the precision of the data type in which the quantized values will be stored. For LDR materials 8 bits are used, resulting in 256 possible values. For HDR materials 8 bits are used for all values except for the terminal node in the luminance processing branch of the pipeline, where 16 bits are used to better cope with the high dynamic range. Each of the input values is then discretized, reducing the original 32-bit width of the value to 8/16-bit width.

6.3.7.2 Minimum Required Bits Encoding

Integer data are represented by a 32-bit data type during the compression phase. Because the data produced during compression have limited range (for example codebook indices range from 0 to the number of entries), using always 32 bits to store them is not necessary, since all the values in a single set will start with the same number of 0s. Removing these insignificant bits from all values in the set and remembering the number of significant bits, the resulting values can be stored using only this limited number of significant bits.

To perform the optimization, the minimum value of the set is first found and then subtracted from all the values in the set. This reduces the number of significant bits for sets whose values range does not start at 0. The number of bits required to store all the values is then calculated from the maximal value of the set as $N = \lceil \log_2(S_{i-i}) \rceil$. Using bit-level manipulations, all the values in the set are then reduced to this number of bits and stored in a single continuous bit stream.

6.3.7.3 Huffman Coding

By studying the data present in the compressed codebooks, we observed, that some entries get used more often than other. To exploit this feature, an optimization based on Huffman

coding [Huf52] was introduced. Given a set of integral values, Huffman coding assigns each value in the set a code word based on the frequency the values occurs in the set. By assigning shorter code words to more frequently used values, Huffman coding presents a way to further compress non-uniformly distributed data.

The Huffman coding optimization was implemented as an alternative to the minimum bits coding optimization. Because Huffman coding requires storing not only the encoded values, but also the codebook by which the values can be decoded, the resulting size of the optimized data depends on the distribution of the values in the input set. This optimization is therefore only used if the resulting size of the optimized data is smaller, than by using the minimum bits encoding.

6.3.8 Modifiers

Previously described in Section 4.4.5 *modifiers* can be used to further enhance capabilities of pipeline nodes and compare units. Because a very tight coupling usually exists between the *modifier* and the subsystem using it, we were not able to find a reasonable way of isolating the *modifiers* functionality into a separate set of components.

The coupling can be demonstrated on the example of the *scaling* modifier. The compare unit used during the search for similar entries needs to know, that it should perform the search *up to scale*. This is impossible to do without incorporating the use of scaling coefficients directly into the comparison function. Components using modifiers are therefore implemented as separate node/compare unit types. If for example the *scaling* modifier is used for the SSIM compare unit, a *SSIMScaling* type needs to be used.

Individual components using modifiers are required to follow the interface of the modifier. This means, that the *SSIMScaling* compare unit can be attached to all nodes which use the *scaling* modifier. If a different similarity metric is implemented into a compare unit with support of the *scaling* modifier, all of the existing pipeline nodes supporting this modifier should be able to attach to it without modifications.

In our implementation, we experimented with the following *modifier* types:

Scaling
: This modifier allows the search for similar entries *up to scale*. Both the input and the reconstructed data are normalized to the same average luminance level. The difference between these levels gets stored in the form of a multiplicative constant, a *scaling coefficient*.

Rotating
: The goal of this modifier was to exploit rotational symmetries in the compressed data. During comparison, the reconstructed were progressively rotated along one of the input dimensions. The number of steps required for the data to match best was then stored as a *rotation index*. Due to time constraints, we were unfortunately unable to fully evaluate the potential of this modifier.

Mirroring
: The goal of this modifier was to exploit mirror symmetries in the compressed data. Instead of directly comparing the input and reconstructed data, mirrored variants were

also compared and the results stored in form of a *mirroring index*. Due to time constraints, we were unfortunately unable to fully evaluate the potential of this modifier.

6.3.9 Additional Features

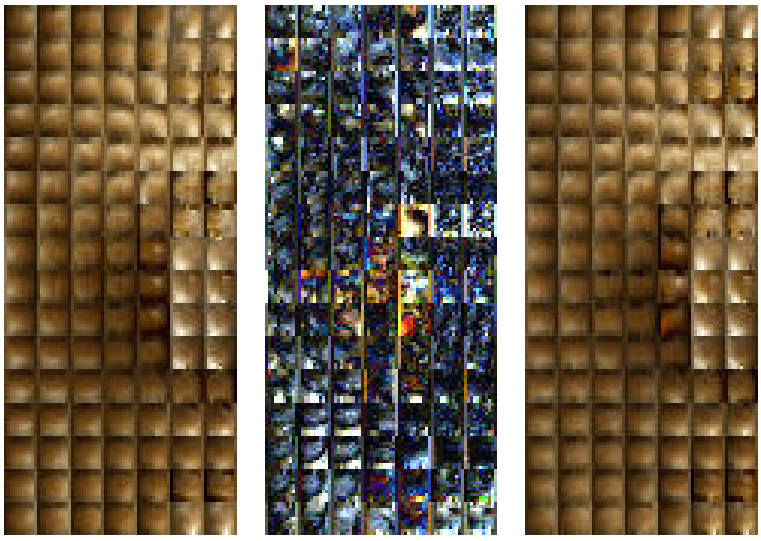
Besides the basic compression-related functionality, the *Compressor* system provides some additional features to ease the process of BTF compression.

6.3.9.1 Suspend / Resume Support

The current state of the whole compression pipeline gets regularly saved to a uniquely named *stateBTF* file. From this file, the compression can be resumed to its previous state later on. While this feature does not affect the compression itself, it provides a failsafe mechanism in the event of a power outage, system reboot or other unexpected situation.

6.3.9.2 Comparison Images Saving

For each BTF texel processed, a set of images showing the texel data before and after compression, as well as the absolute difference between these data sets can be optionally generated. An example of such images is shown in Figure 6.9. This feature allows the user to quickly verify, that the new settings used for the compression did not produce flawed results.



(a) Original (b) Difference (c) Compressed

Figure 6.9: Original and compressed BTF data for a single texel and their absolute difference (multiplied 8× for better visibility).

6.4 Decompression Algorithm

The decompression algorithm is implemented in form of a separate library named *Evaluator*, which can be easily integrated into custom solutions. Because of the dynamic nature of the compression pipeline, the implementation of the decompression algorithm needs to support this as well. As a solution to this problem, a dynamic code generation approach was selected.

The dynamic code generation approach has several benefits. The first being, that on the topmost level, only a single function handling the whole BTF decompression process exists. When provided with the requested BTF coordinates, a single call to this function returns the corresponding BTF reflectance at the given coordinates. This allows for easy integration into a custom solution. The approach also has very small overhead, because no intermediate values need to be stored and accessed in the global memory. By using different code templates, it should be also possible to generate the decompression code in language other than OpenCL, for example GLSL.

6.4.1 Decompression Algorithm Workflow

All the information needed to perform the decompression, including the BTF material discretization, the processing pipeline layout and the compressed data codebooks, are stored within the *codeBTF* file. Based on the configuration stored within the file, nodes of various types are first assembled to form a pipeline with the same layout, as the pipeline used during compression. This means, that the same types of nodes are used and connected together in the same way, as during the compression. All additional variables, such as the configuration of outputs for *SplitChannels* type nodes, are also read from the *codeBTF* file.

6.4.1.1 Dynamic Code Generation

After assembling the pipeline, the dynamic code generation phase begins. During this phase, each of the nodes used within the pipelines provides three types of OpenCL code fragments. The first fragment type is *declarations*, in which the node declares structures it uses. These include primarily token and codebook entry definitions. The second type of code fragments, *globals*, is used to describe memory regions the node uses to store its data. *Globals* typically contain a memory region in which the node's codebook is stored.

The third and final type of code fragments are *functions*. A single function, referred to as the *evaluation function*, with a name starting with the prefix *Evaluate_* , must be present for each node. This function gets called by the parent node during the evaluation and is responsible for performing the main action of the node. For example for the *Transform* node, the evaluation function first calls the evaluation function of its child, then performs the required transformation of the data and returns the transformed values back to the calling node. Apart from the main evaluation function, any number of additional support functions can be defined within the *functions* fragment.

After obtaining all the code fragments from the individual nodes, a master code template is used by the code generator to place these fragments in their final locations. In the master template, first the *declarations* fragments are inserted, followed by *globals* and finally

functions. The master template also provides some own code fragments, such as the input coordinate structure declaration or the evaluation entry point function *EvaluateBTF*.

The template also contains two variants of an OpenCL kernel used to launch the decompression algorithm. The only difference between these two variants is that the first one uses OpenCL buffer objects to store its input and output data, while the other one uses OpenCL textures. The buffer-based kernel allows for easier integration with CPU-based applications. The texture-based kernel is more suited for GPU-based application, because it allows for easier use of OpenCL resource sharing (for example with OpenGL). Additional arguments of these kernels are dynamically generated by the code generator, based on the global variables required by the individual nodes.

6.4.1.2 Codebook Unpacking

If postprocessing described in Section 6.3.7 was applied to codebooks data prior saving them to the *codeBTF* file, a reverse set of actions, referred to as codebook *unpacking*, needs to take place in order for the nodes to use the data. Although in future this can be executed on-the-fly directly during evaluation, in the current implementation, the codebook unpacking is performed before the codebook data are uploaded to their corresponding memory regions.

This is basically a reverse variant of the postprocessing used during compression. A packed codebook is first passed to the child node to revert the postprocesing done. This means, that minimal bits or Huffman coded values get restored back to their 32-bit variants and floating point values get restored back to the 32-bit IEEE 754 [IEEE754] format. The child node then returns the codebook back to the parent, which can later use it during the evaluation phase, because the data present within the codebook are now in the same format as the node originally stored them.

6.4.1.3 Pipeline Node Workflow

The main action the node performs during the evaluation process is represented by the *Evaluate_(NodeID)* function (where *NodeID* is replaced by the identifier of the node). As an input to this function, the node receives its codebooks, the input coordinates for which the evaluation is performed, a token from the node's parent and a reference to the resulting element, into which the output should be saved. The set of actions performed then depends on the type of the node.

Nodes from the *nodes with codebook category* typically use the token received to find a corresponding entry within their codebook. Using this entry and one of the input coordinates, a token for the child node is found. The main evaluation function of the child node is then called, using the token as one of its arguments. After the child node has finished the evaluation, the resulting data are returned to the active node using the reference to the resulting element. The current node is then free to further process the data, for example apply a scaling coefficient, and then return the result back to its parent using the same reference to the resulting element.

6.4.2 Input Coordinates Usage

Depending on the node type and the position of the node in the compression pipeline, one or more of the input BTF coordinates might be used to index the data stored within the node's codebook entry. The input dimensions the node should operate on are determined during the pipeline initialization process. The order in which the individual dimensions are used is fixed as $[x, y, \theta_V, \phi_V, \alpha, \beta]$ and is the same as during compression.

The $[x, y]$ coordinates are always used by the root *Materials* node. In the current implementation, only the *FindOrSplit* and *FindOrSave* nodes use input coordinates to index their data. This means, that the first *FindOrSplit* node encountered during the evaluation will use the θ_V coordinate to index the data of an entry from its codebook in order to find a single token for its child node. The next *FindOrSplit* node will use the ϕ_V coordinate and so on. Being terminal, the *FindOrSave* node will always use the β coordinate to find the representative value in its codebook entry.

6.4.3 Interpolation

The number of images using which a BTF was acquired is finite, so BTF is a discrete function. To obtain values of BTF at coordinates not directly stored within the input data, an interpolation needs to be performed.

The interpolation scheme used is based on linear interpolation in each of the six input dimensions. When indexing a codebook entry, two tokens closest to the required input coordinate are evaluated. The results obtained are then linearly interpolated together based on the distance from the actual coordinate. This is recursively repeated for each dimension. As a result, $2^6 = 64$ raw values are required to correctly reconstruct data for a single BTF query. For the x and y coordinates, the interpolation can be omitted, decreasing the number of raw values to $2^4 = 16$, but reducing the overall visual quality. The visual difference is similar to using nearest neighbor filtering instead of linear filtering for standard texturing, as shown in Figure 6.10.

6.5 Preview Generation Tools

To provide visual preview of data produced during various stages of the compression process, two utilities were created, each suited for a slightly different purpose. The first is an interactive previewer allowing real-time visualization of scenes containing objects mapped with BTF materials. The second is an offline image generation tool which provides a greater degree of configuration flexibility at the cost of reduced interactivity.

6.5.1 Interactive Previewer Tool

To verify the results of the evaluation algorithm and demonstrate its performance, an OpenGL-based application utilizing the *Evaluator* component was created. This application, referred to as *Interactive previewer*, allows interactive rendering of a scene containing an object covered with a BTF material and illuminated by a single directional light source. Properties of both the camera and the light source can be modified in realtime.

(a) 6D (b) 4D

Figure 6.10: Detail of the *Impalla* BTF showing the visual difference of performing interpolation in 6D or 4D space

The basic concept of the application is similar to the one described in [Ege13] and is demonstrated in Figure 6.11. The OpenGL part of the application is responsible for rasterization of the scene into the required BTF coordinates using GLSL shaders. This raster is then used as an input to the OpenCL-based BTF decompressor provided by the *Evaluator* subsystem. After performing the evaluation, a texture representing the resulting image is returned back to OpenGL, from which it is then displayed on screen by mapping it on a full-screen quadrilateral.

The application also allows to render a scene using environment map lighting. For this technique a predefined number of virtual light sources is created from the map using a Halton sequence [Hal64]. The contributions of each of the lights to the final scene is then evaluated sequentially and the results are summed together to form the final image.

6.5.2 Offline Image Generation Tool

A command line application, named simply *Previewer*, was created, which allows the generation of various preview and debugging images from multiple different sources and using different techniques. Compared to the *Interactive previewer*, this application operates using a predefined set of rules specified in a configuration file and is non-interactive, but provides a greater degree of configuration flexibility.

Internally, the two basic types of entities are used – *rasterizers* and *samplers*. A *rasterizer* is responsible for preparing a two-dimensional raster of BTF coordinates. The raster then gets processed by a *sampler*, which transforms each element of the raster into a resulting color. The colors then get saved as an image of the same dimensions. By using different combinations of *rasterizers* and *samplers*, images resulting from various steps of the compression can be obtained and visually compared.

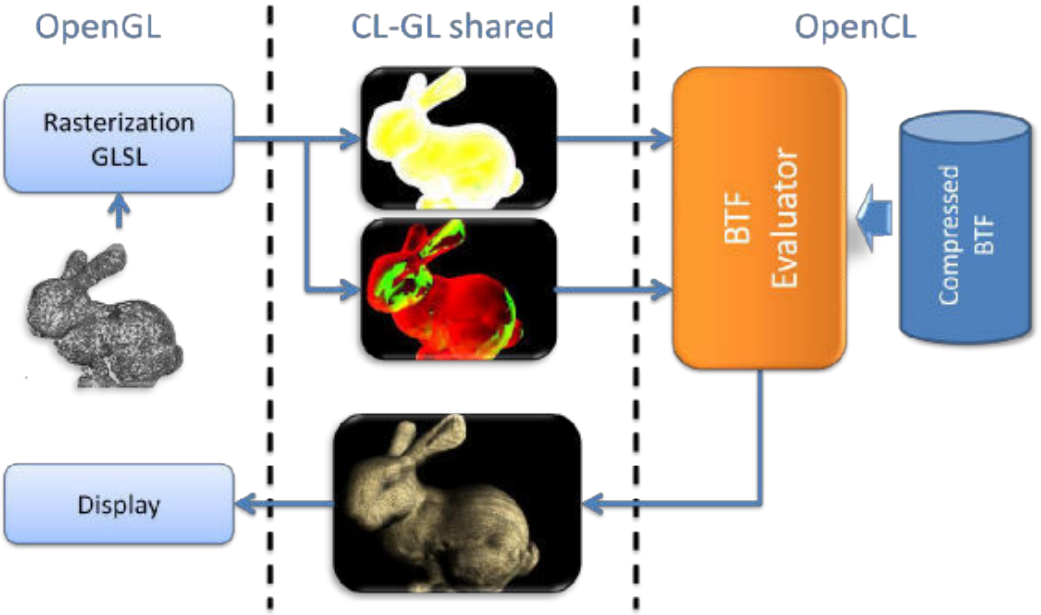


Figure 6.11: Data flow in the *Interactive previewer* application

The BTF coordinate raster generated using the current camera and light properties can be also saved to disk and later used by the non-interactive preview generation tool. This allows the user to interactively set properties of the scene, but render it using different methods later on.

6.5.2.1 Rasterizer Types

A *rasterizer* generates a two-dimensional raster of BTF coordinates. The dimensions of the raster and the method used to generate the coordinates, depend on the type of *rasterizer* used. Multiple rasters of the same size can also be generated, which, after processing, get stored as separate subimages in the resulting image. The following *rasterizer* types were implemented:

File
: This type of *rasterizer* reads the whole raster from a file and just passes it to a *sampler* for processing.

OpenGL
: This *rasterizer* uses an OpenGL application with custom GLSL shaders to rasterize a scene and provide the BTF coordinates for each pixel of the resulting image. The application allows some degree of interactivity, such as free camera movement. To the user, the scene is rendered using simple Phong shading. After the application gets closed, the scene gets rendered using the custom shaders and the result is used as the output of the *rasterizer*.

Pattern
The pattern-based *rasterizer* can generate BTF coordinate rasters using a predefined set of rules. These rules may be configured for example to recreate the same parameterization, as used in the original input data, or to render a single texel of BTF data.

6.5.2.2 Sampler Types

A *sampler* is used to transform rasterized elements of input data into colored pixels in the resulting image. The following *sampler* types were implemented:

Debug
The debug *sampler* is used to directly visualize the input coordinates. An input coordinate can be mapped to a specified color channel and optionally normalized to fit the color range of the image. Additionally, transformations of the coordinates between different coordinate systems are provided to aid understanding of the visualized data. This *sampler* is mainly used to detect anomalies in the input data and help debugging new *rasterizer* types.

TempBTF
This *sampler* type uses BTF data in *tempBTF* format to evaluate the raster. Since *tempBTF* data get created during the preprocessing stage of the compression, a *sampler* of this type can be used to verify correct parsing of the original input data.

OnionBTF
This *sampler* type uses BTF data in *onionBTF* format to evaluate the raster. *OnionBTF* data get created by the *Resampler* component and by using this *sampler* type can be visualized to verify correct resampling of the input data to *Onion-Slices* parameterization. The generated image can also serve as a reference for the before-after comparison of the compressed material.

CodeBTF
This *sampler* type uses the compressed BTF data in *codeBTF* format to perform the evaluation. An image created using this *sampler* represents the visual quality of the material after performing the compression. The same *Evaluator* component is used as in the *Interactive previewer* application.

Dump
This *sampler* type does not generate any image data, but stores the input raster into a file, which can be later read by the file-based *rasterizer* type.

Chapter 7

Verification and Validation

The steps taken in order to verify correctness of the results produced by our improved implementation of the MLVQ-based algorithm are described in this chapter.

7.1 Validation Description

In order to verify correctness of the proposed algorithm and its implementation, we compared the results produced by our implementation of the algorithm with the results presented by Havran et al. in the original paper [HFM10]. To perform the comparison, 6 LDR materials from the UBO2003 [SSK03] data set and 4 HDR materials from the ATRIUM [ATRIUM] data set were compressed and then compared in the terms of resulting data size and visual quality.

For the comparison to be fair, the properties of the compression pipeline were set to match those used in [HFM10] as closely as possible. These settings include mainly the layout of the pipeline, similarity thresholds for individual pipeline nodes and number of BTF texels to process in a single stage and their order. The same discretization was used for all dimension as in the paper [HFM10] when resampling into the *Onion-Slices* parameterization: $n_x = 256$, $n_y = 256$, $n_{\theta_V} = 16$, $n_{\phi_V} = 7$, $n_\alpha = 11$, $n_\beta = 11$.

Three compression stages were used. In the first stage, 1.25% of all BTF texels were compressed using strict settings for the required similarity thresholds. A Halton sequence [Hal64] generator was used to determine which texels should be processed. This stage was used to fill the codebooks with most of the representative entries. In the second stage, another 4% of all texels were compressed using relaxed strictness for the similarity thresholds. The sequence of texels was also created using a Halton sequence generator. The purpose of this stage was to add another set of the more important representative entries, which might have been missed during the first stage. In the third and final stage, the codebooks were locked and the remaining texels were compressed sequentially. In this stage, only the entries already present in the codebooks were used to match the input and no new entries were added. This configuration corresponds to the one used in [HFM10] to allow a meaningful comparison.

To assess the visual quality of the result, MSSIM index [WBSS04] was computed between images rendered using the original and the compressed data.

To inspect the behavior of the compressed materials in dynamic conditions, our *Interactive previewer* application (Section 6.5.1) was used. In the previewer application, illumination and camera properties were modified in real-time and the reactions of the materials were observed. The goal of this was to find error related to the use of input coordinates and transformations between different coordinate systems, which may not show in static images. To get a reference behavior, the materials were also observed using the *BTFBASEShader V1.0* [BTFBASE] application, created by the authors of [HFM10] for the original algorithm implementation.

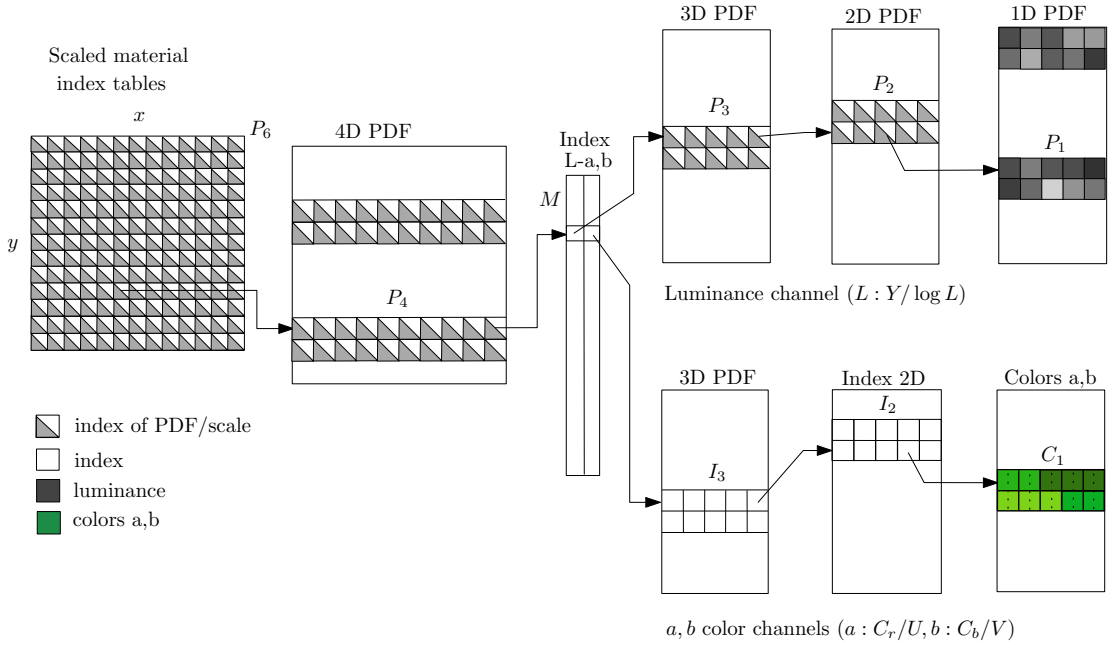


Figure 7.1: The compression pipeline used to validate the results of our implementation.

7.2 Validation Results

The results of the validation process are summarized in Table 7.1. Using the same compression pipeline and settings, the results produced by our implementation are very close to those described in [HFM10]. On average, the difference in the resulting compression ratio is about 10%. The visual quality was improved slightly, by an average factor of 7%. We believe the main reason for this is the omission of the separate C codebook (described in Section 4.7), which was expected to improve the visual quality slightly with a small compression ratio penalty. Different interpolation algorithm was also used for *Onion-Slices* parameterization resampling (Section 4.3), which might as well affect the results slightly. We also studied the number of entries present in the individual codebooks after finishing the compression. An example for the *Corduroy* material is provided in Table 7.2.

Under dynamic conditions, the behavior of materials matched those observed in the original *BTFBASEShader V1.0* [BTFBASE] application. We noticed the *BTFBASEShader V1.0* application not to correctly attenuate the illumination intensity with increasing angle of

BTF sample	Compression ratio				MSSIM		
	HFM10†	HFM10*	our	$\frac{\text{our}}{\text{HFM10*}}$	HFM10	our	$\frac{\text{our}}{\text{HFM10*}}$
corduroy	1:128	1:124	1:142	+15%	0.748	0.731	-2%
impalla	1:162	1:178	1:177	-1%	0.730	0.854	+17%
proposte	1:236	1:248	1:306	+23%	0.710	0.786	+11%
pulli	1:87	1:96	1:58	-40%	0.699	0.770	+10%
wallpaper	1:222	1:238	1:195	-18%	0.776	0.770	-1%
wool	1:77	1:71	1:87	+23%	0.684	0.763	+12%
ceilingHDR	1:235	1:244	1:291	+19%	0.711	0.839	+18%
floortileHDR	1:136	1:141	1:198	+40%	0.772	0.893	+16%
pinktileHDR	1:711	1:768	1:389	-49%	0.961	0.932	-3%
walkwayHDR	1:102	1:102	1:138	+35%	0.884	0.891	+1%
Average	1:210	1:221	1:198	-10%	0.768	0.823	+7%

† Results as described in [HFM10], $C.R.^1$ column.

* Results from [HFM10], $C.R.^1$ column, recomputed for the absence of the separate C codebook.

Table 7.1: Compression ratios and visual quality of results produced by our pipeline compared to results described in [HFM10].

incidence. This is handled correctly by our implementation. An example comparison between the images generated by our implementation and the *BTFBASEShader V1.0* application is shown in Figure 7.2. As conclusion, we believe our implementation to produce correct results.

Codebook	Entries count [-]			Basic size [B]		
	HFM10*	our	$\frac{\text{our}}{\text{HFM10*}}$	HFM10*	our	$\frac{\text{our}}{\text{HFM10*}}$
P_6	65536	65536	+0%	524 288	524 288	0%
P_4	838	880	+5%	107 264	112 640	+5%
P_3	13231	13511	+2%	740 936	756 616	+2%
M	49790	60720	+22%	398 320	485 760	+22%
P_2	43876	43514	-1%	3 861 088	3 829 232	-1%
I_2	36913	26617	-28%	1 624 172	1 171 148	-28%
P_1	11647	6151	-47%	512 468	270 644	-47%
I_1	30025	21681	-28%	2 642 200	1 907 928	-28%
Average				10 410 736	9 058 256	-13%

Codebook	Optimized size [B]			Huffman coding size [B]	
	HFM10*	our	$\frac{\text{our}}{\text{HFM10*}}$	our	$\frac{\text{Huffman}}{\text{optimized}}$
P_6	147 456	147 456	+0%	121 639	-18%
P_4	36 872	38 720	+5%	27 126	-30%
P_3	277 851	283 731	+2%	227 985	-20%
M	199 160	235 290	+18%	235 318	0%
P_2	1 327 249	1 256 467	-5%	1 039 647	-17%
I_2	761 331	548 976	-28%	487 677	-11%
P_1	128 117	67 661	-47%	65 660	-3%
I_1	660 550	476 982	-28%	416 136	-13%
Average	3 538 586	3 055 283	-14%	2 621 188	-14%

* Results from [HFM10], $C.R.^1$ column, recomputed for the absence of the separate C codebook.

Table 7.2: Number of entries in individual codebooks and their resulting sizes for the *Corduroy* material compared to [HFM10].

(a) Reference (b) Our

Figure 7.2: Comparison of the images produced by our implementation and the reference *BTFBASEShader V1.0* application [BTFBASE] for the *Wallpaper* BTF. Notice the lack of proper attenuation on the reference image.

Chapter 8

Results

In this chapter, we discuss the results obtained by using the implemented BTF compression framework to perform compression of several different BTF materials using four different compression pipeline layouts.

8.1 Hardware Setup

All the results presented were measured using the following hardware setup:

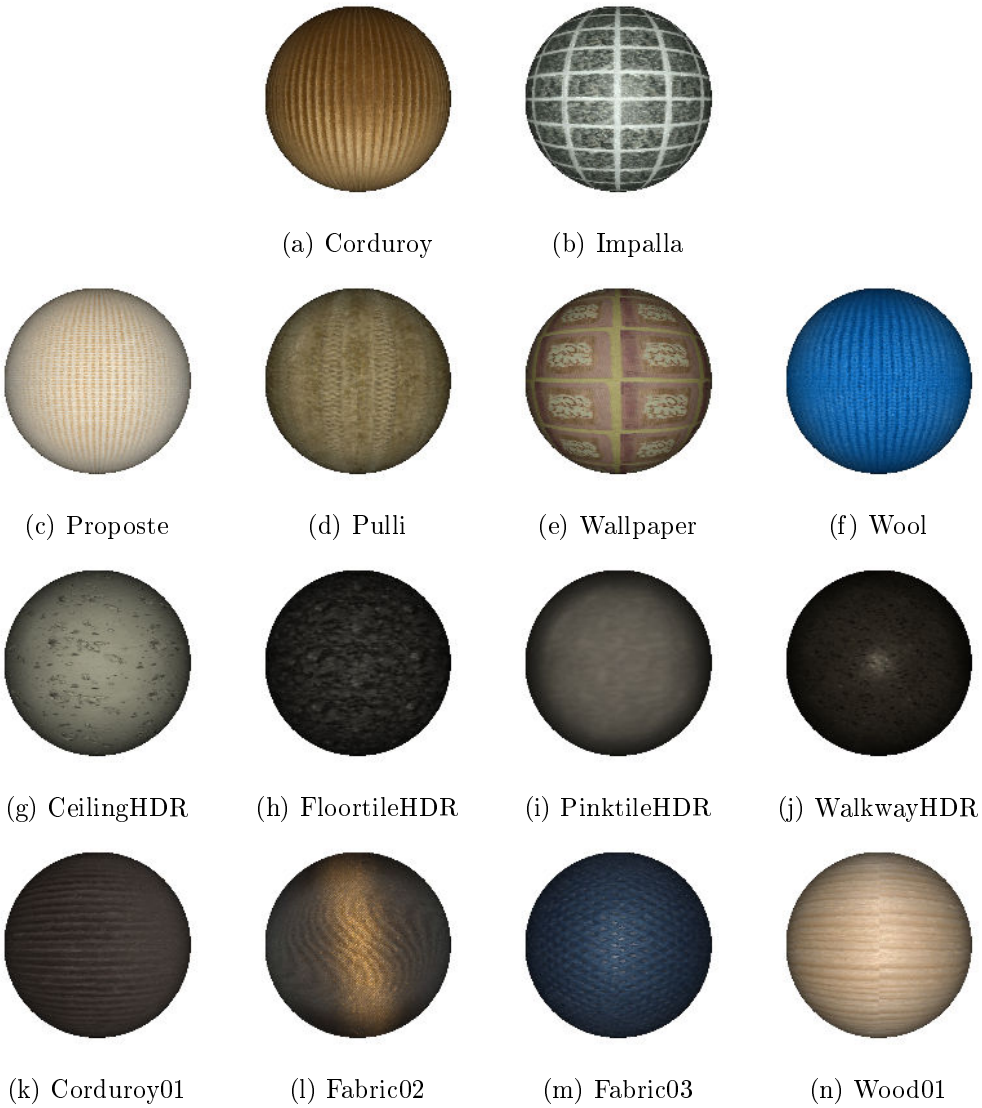
- Intel Core i5 3570K 3.40GHz quad-core processor
- 16 GB of 1333 MHz DDR3 system memory
- NVidia GeForce GTX 780 Ti (GK110) graphics card
  3GB GDDR5 384-bit @ 7000 MHz
  Core @ 1084 MHz
- Samsung 840 Pro 256GB SSD

The following software environment was used to compile and execute the framework:

- Microsoft Window 7 Professional x64 operating system
- MinGW64 compiler suite + TDM-GCC x64 4.7.1 compiler

8.2 Tests Description

For our experiments a total of 14 different BTF materials were used, four of which were HDR as shown in Figure 8.1. Six of the LDR materials, namely *Corduroy*, *Impalla*, *Proposte*, *Pulli*, *Wallpaper* and *Wool* are part of the UBO2003 data set [SSK03]. The remaining four LDR materials, namely *Corduroy01*, *Fabric02*, *Fabric03* and *Wood01* were obtained from the UTIA BTF database [HFV12]. The four HDR materials, namely *Ceiling*, *Floortile*, *Pinktile* and *Walkway*, are part of the ATRIUM data set [ATRIUM] from BTF Database Bonn.



(a) Corduroy (b) Impalla

(c) Proposte (d) Pulli (e) Wallpaper (f) Wool

(g) CeilingHDR (h) FloortileHDR (i) PinktileHDR (j) WalkwayHDR

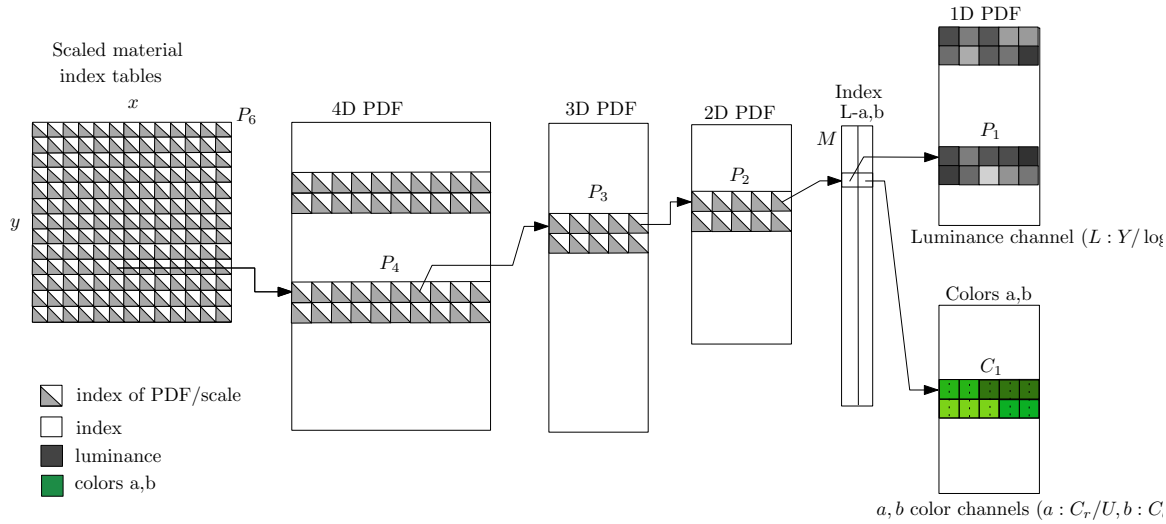
(k) Corduroy01 (l) Fabric02 (m) Fabric03 (n) Wood01

Figure 8.1: Overview of the BTF materials for which results are presented

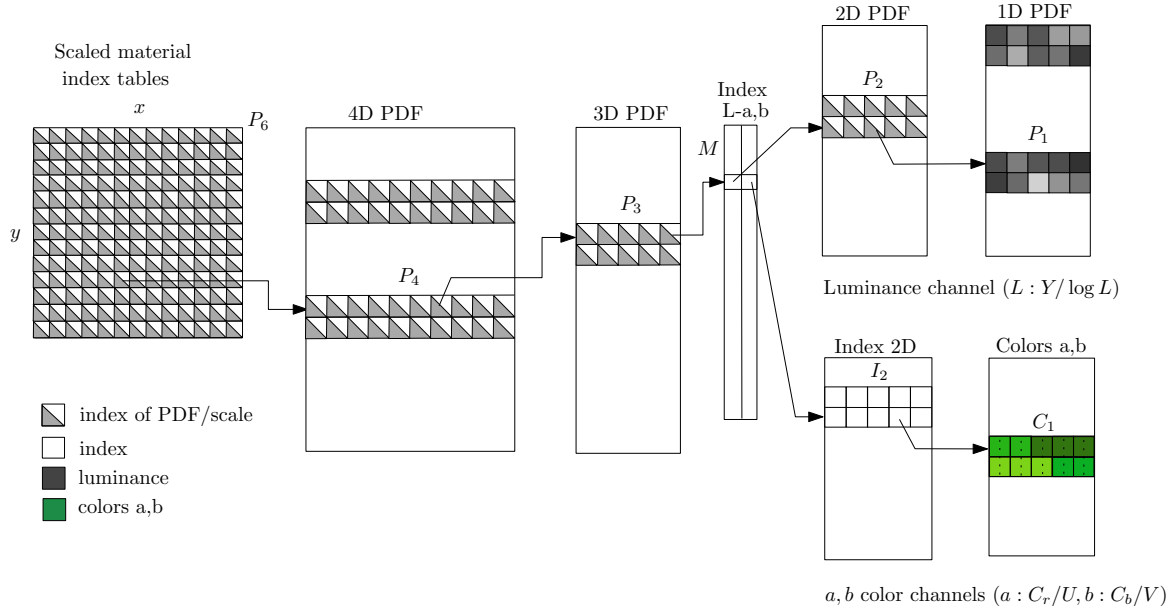
For each material, four different processing pipelines were used, as shown in Figure 8.2. The difference between these pipelines was the level, from which luminance and chrominance components get treated separately. The remaining conditions such as the discretization used and the three compression stages remained the same as during the algorithm validation described in Section 7.1.

For each material and pipeline combination, the following properties were observed:

- The total compression time.
- The number of entries in individual nodes' codebooks and the total size of these codebooks.
- The resulting data size without optimizations.

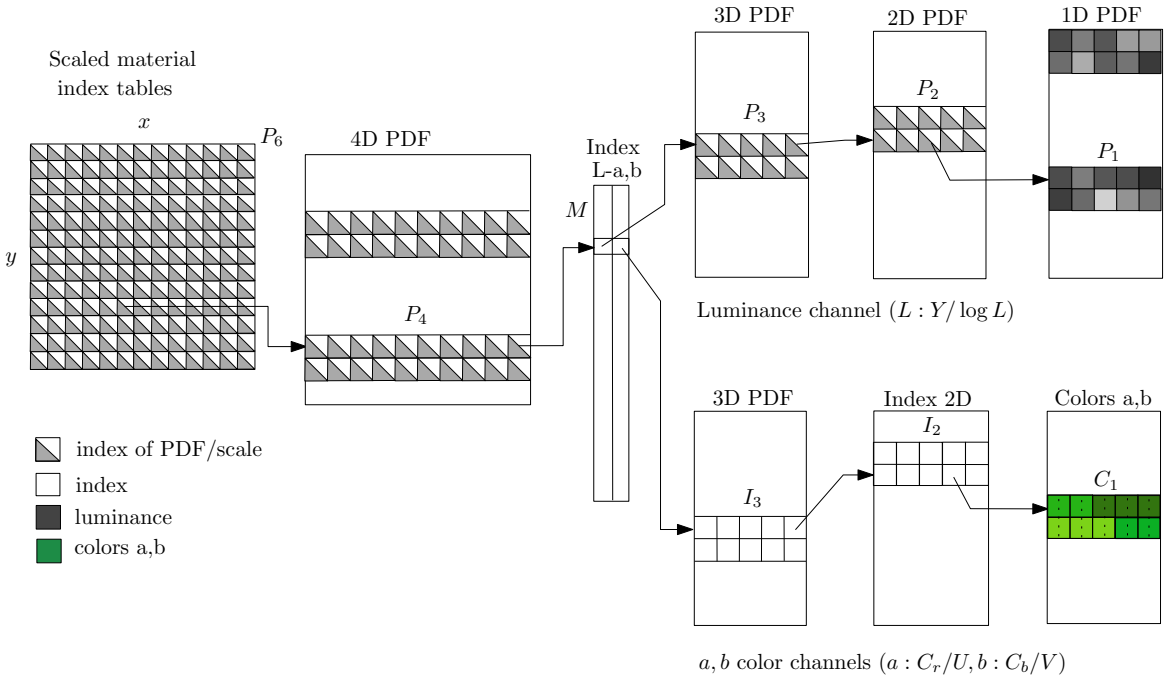


(a) Luminance and chrominance separation in 1D.

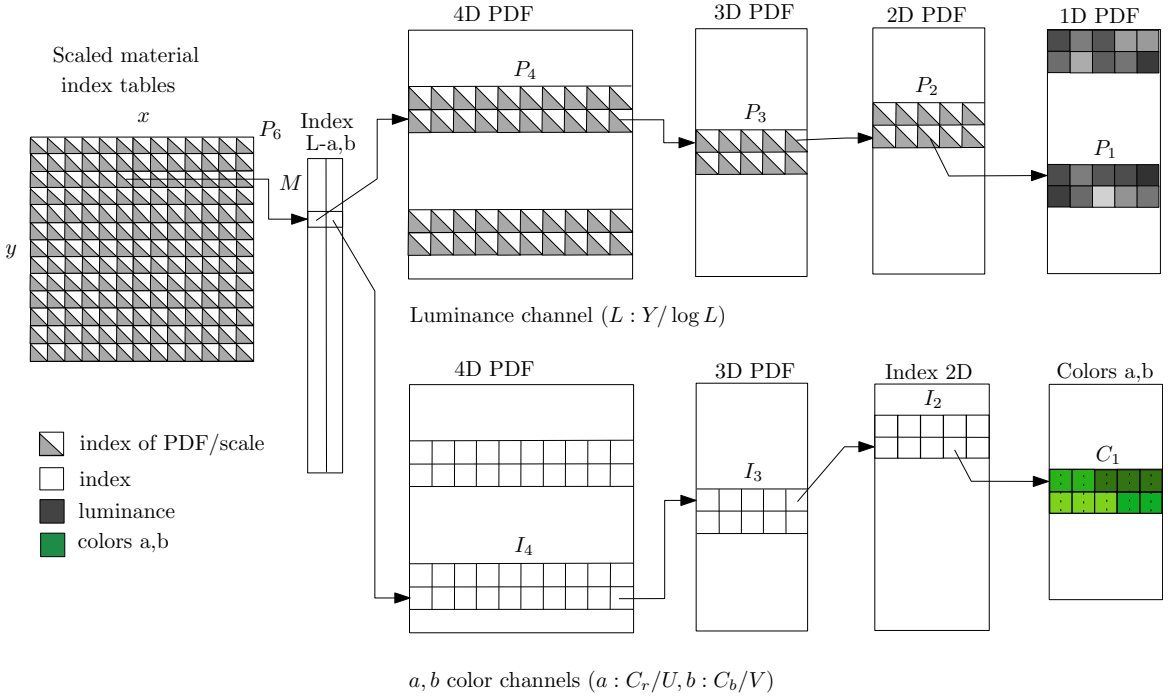


(b) Luminance and chrominance separation in 2D.

Figure 8.2: Compression pipelines with different levels of luminance-chrominance separation used during tests.



(c) Luminance and chrominance separation in 3D.



(d) Luminance and chrominance separation in 4D.

Figure 8.2: Compression pipelines with different levels of luminance-chrominance separation used during tests (*cont.*).

- The resulting data size using optimizations described in Section 6.3.7, except for Huffman coding.
- The resulting data size using optimizations described in Section 6.3.7, including Huffman coding.
- Visual quality difference using MSSIM index [WBSS04] when rendering a test scene using the original and compressed data.

For reference we also show the results presented in two of the papers most relevant to our work [HFM10, WDR11].

8.3 Compressed Data Size

The resulting compression ratios for all materials and compression pipeline layouts tested are summarized in Table 8.1. As previously shown in Table 7.1, when using conditions similar to those described in [HFM10], we were able to achieve similar compression ratios, while maintaining roughly the same visual quality.

When comparing the results for pipelines with different points of luminance-chrominance separation, we observe that there is not a single variant, which would work the best for all materials. Most of the times either the 4D or 1D split variants produce the best results, but there is no visible correlation between the compressed material properties and the answer to which variant should be used. The average compression ratio benefit of using a specific split point was 13%.

The use Huffman coding to further optimize the compressed material codebooks proved to be beneficial and resulted in average compression ratio improvement of 17%, as shown in Table 8.2. The basic Huffman coding used is not well suited for the evaluation process in rendering algorithms, because it does not allow random access directly into the optimized data. Better compression ratios resulting from the use of Huffman coding however show that the use of some more advanced entropy-based compression scheme might be beneficial.

8.4 Visual Quality of Compressed Data

The visual quality of the results was determined by computing the MSSIM [WBSS04] index of a scene rendered using the original and the compressed data. An example of such images can be seen in Figure 8.3.

According to the results shown in Table 8.3, our implementation was able to achieve slightly better visual quality than [HFM10]. It should be noted, that because the exact conditions used to compute the similarity in [HFM10] are not known, slightly different scene might have been used and a direct comparison of the results would be unfair. When compared to the results presented in [WDR11], the visual quality of materials compressed using our implementation is overall worse.

Although multiple different compression pipelines were used, the resulting visual quality remains almost identical, regardless of the level on which luminance and chrominance channels get separated. For this reason only a single similarity index is included for each material in Table 8.3.

(a) Original. (b) Compressed.

Figure 8.3: Comparison of visual quality for the *Wallpaper* material.

8.5 Compression Speed

The compression speed of our implementation for each of the materials can be seen in Table 8.4. On average, the compression times resulting from using our implementation were roughly 13.3× faster than those presented in [HFM10] and 8× faster than [WDR11]. The compression speed depends on the material being compressed, but varies only slightly with the pipeline being used. We believe, that for a single, optimized variant of the algorithm a sub-one hour average run time would be possible by sacrificing the modularity and flexibility of the current implementation.

8.6 Decompression Speed

The decompression speed of our implementation was measured using the *Interactive previewer* tool described in Section 6.5.1. The previewer was configured such that each pixel rendered on the screen results in a BTF evaluation with different input coordinates. This approach was used to minimize the amount of hidden caching within the GPU and provide results for a worst-case scenario.

The decompression rates for different materials and pipeline layouts are summarized in Table 8.5. By directly measuring the run time of an evaluation kernel we show that our implementation reaches average decompression rate of 127 million individual BTF evaluations per second in a worst-case scenario[1]. In [HFM10] an evaluation rate of 0.31 to 1.36 million BTF evaluations per second is given for a CPU implementation and a frame rate of 170

[1] The time required to read the input coordinates from the global memory and to store the results back is included in the evaluation time due to the testing method used. In production use, the evaluation function can be called directly, with the input coordinates procedurally generated and the results directly used. This eliminates the global memory access overhead and should result in even greater evaluation rates.

frames per second for rendering an 800×600 resolution scene is provided. Because the exact properties of the testing scene and measurement conditions (such as the amount of pixels for which the BTF evaluation is really performed) are not known, it is impossible to estimate the actual decompression rate and provide a fair comparison.

Unexpected results were obtained by observing the decompression rates with relation to the point of luminance-chrominance separation. According to our results, the sooner the luminance and chrominance components get separated, the faster is the decompression process. This is unexpected, because more codebooks get used when separating the components earlier, resulting in increased number of global memory fetches. We believe the increase in evaluation rate to be caused either by somehow improved caching in the GPU, or by the OpenCL compiler performing some kind of optimization dependent on the pipeline layout.

BTF sample	Ref. compression ratio			Our compression ratio - basic					Our compression ratio - optimized				
	HFM10[1]	HFM10[4]	WDR11	1D	2D	3D	4D	$\frac{\text{best}}{\text{2D}}$	1D	2D	3D	4D	$\frac{\text{best}}{\text{2D}}$
corduroy	1:128	1:418	1:71	1:170	1:142	1:142	1:141	+20%	1:485	1:422	1:420	1:417	+15%
impalla	1:162	1:522	1:105	1:184	1:177	1:169	1:172	+4%	1:536	1:512	1:491	1:501	+5%
proposte	1:236	1:806	1:144	1:284	1:306	1:305	1:309	+1%	1:838	1:931	1:939	1:954	+2%
pulli	1:87	1:264	1:138	1:55	1:58	1:61	1:62	+7%	1:143	1:157	1:166	1:170	+8%
wallpaper	1:728	1:222	1:133	1:170	1:195	1:227	1:245	+26%	1:481	1:593	1:707	1:767	+29%
wool	1:77	1:233	1:98	1:83	1:87	1:86	1:85	0%	1:220	1:239	1:233	1:232	0%
ceilingHDR	1:235	1:780	1:303	1:318	1:291	1:227	1:203	+9%	1:855	1:733	1:611	1:532	+17%
floortileHDR	1:136	1:360	1:248	1:216	1:198	1:186	1:195	+9%	1:567	1:533	1:506	1:529	+6%
pinktileHDR	1:711	1:2267	1:198	1:483	1:389	1:278	1:220	+24%	1:1286	1:968	1:691	1:551	+33%
walkwayHDR	1:102	1:257	1:194	1:123	1:138	1:158	1:162	+17%	1:305	1:370	1:435	1:446	+21%
corduroy01	n/a	n/a	n/a	1:270	1:310	1:302	1:316	+2%	1:891	1:959	1:940	1:985	+3%
fabric02	n/a	n/a	n/a	1:142	1:129	1:125	1:127	+10%	1:391	1:368	1:360	1:367	+6%
fabric03	n/a	n/a	n/a	1:82	1:82	1:85	1:88	+7%	1:222	1:229	1:244	1:252	+10%
wood01	n/a	n/a	n/a	1:174	1:205	1:230	1:224	+12%	1:492	1:603	1:692	1:738	+22%
Average	1:210	1:640	1:163	1:197	1:193	1:184	1:182	+11%	1:551	1:544	1:531	1:532	+13%

Table 8.1: Resulting compression ratios for different materials and pipeline layouts compared to [HFM10] and [WDR11]

BTF sample	Ref. C.R.	Huffman optimized compression ratio				Improvement over basic optimizations			
	HFM10[4]	1D	2D	3D	4D	1D	2D	3D	4D
corduroy	1:418	1:561	1:492	1:488	1:482	+16%	+17%	+16%	+16%
impalla	1:522	1:615	1:614	1:596	1:607	+15%	+20%	+21%	+21%
proposte	1:806	1:951	1:1063	1:1093	1:1107	+13%	+14%	+16%	+16%
pulli	1:264	1:163	1:191	1:204	1:208	+14%	+22%	+23%	+22%
wallpaper	1:728	1:545	1:681	1:810	1:875	+13%	+15%	+15%	+14%
wool	1:233	1:248	1:278	1:267	1:266	+13%	+16%	+15%	+15%
ceilingHDR	1:780	1:984	1:874	1:697	1:626	+15%	+19%	+14%	+18%
floortileHDR	1:360	1:663	1:649	1:625	1:652	+17%	+22%	+24%	+23%
pinktileHDR	1:2267	1:1455	1:1073	1:769	1:625	+13%	+11%	+11%	+13%
walkwayHDR	1:257	1:352	1:465	1:549	1:563	+15%	+26%	+26%	+26%
corduroy01	n/a	1:919	1:1096	1:1066	1:1115	+3%	+14%	+13%	+13%
fabric02	n/a	1:434	1:427	1:419	1:427	+11%	+16%	+16%	+16%
fabric03	n/a	1:252	1:273	1:292	1:303	+14%	+19%	+20%	+20%
wood01	n/a	1:563	1:732	1:834	1:890	+14%	+21%	+21%	+21%
Average	1:640	1:622	1:636	1:622	1:625	+13%	+17%	+17%	+18%

Table 8.2: Compression ratio improvements resulting from the use of Huffman coding compared to [HFM10]

BTF sample	MSSIM		
	HFM10	WDR11	our
corduroy	0.748	0.920	0.731
impalla	0.730	0.934	0.854
proposte	0.710	0.936	0.786
pulli	0.699	0.883	0.770
wallpaper	0.776	0.941	0.770
wool	0.684	0.904	0.763
ceilingHDR	0.711	0.971	0.839
floortileHDR	0.772	0.921	0.893
pinktileHDR	0.961	0.999	0.932
walkwayHDR	0.884	0.980	0.891
corduroy01	n/a	n/a	0.796
fabric02	n/a	n/a	0.777
fabric03	n/a	n/a	0.803
wood01	n/a	n/a	0.850
Average	0.768	0.939	0.818

Table 8.3: Visual quality of materials compressed using our implementation compared to [HFM10] and [WDR11] by the means of MSSIM [WBSS04] index.

BTF sample	Ref. compression time [h]		Our compression time [h]				Speedup factor	
	HFM10	WDR11	1D	2D	3D	4D	$\frac{2D}{HFM10}$	$\frac{2D}{WDR11}$
corduroy	19.2	14.7	1.1	1.1	1.3	1.1	17.5×	13.4×
impalla	21.8	11.7	2.0	2.0	2.5	2.0	10.9×	5.9×
proposte	18.0	8.6	1.0	1.1	1.1	1.2	16.4×	7.8×
pulli	27.1	10.6	7.3	6.8	7.3	5.8	4.0×	1.6×
wallpaper	28.8	12.4	1.1	1.1	1.1	1.2	26.2×	11.3×
wool	50.2	10.5	7.4	7.3	10.2	7.6	6.9×	1.4×
ceilingHDR	20.1	12.9	1.7	1.6	1.6	1.5	12.4×	8.0×
floortileHDR	28.7	15.5	2.8	3.2	3.5	2.7	9.0×	4.9×
pinktileHDR	15.6	14.7	0.7	0.7	0.7	0.7	22.9×	21.6×
walkwayHDR	37.4	21.3	4.2	5.5	5.3	4.5	6.8×	3.9×
corduroy01	n/a	n/a	1.1	1.2	1.3	1.1	n/a	n/a
fabric02	n/a	n/a	1.3	1.7	1.7	1.5	n/a	n/a
fabric03	n/a	n/a	4.8	4.4	4.3	3.5	n/a	n/a
wood01	n/a	n/a	1.2	1.3	1.3	1.1	n/a	n/a
Average	26.7	13.3	2.7	2.8	3.1	2.5	13.3	8.0

Table 8.4: Compression times for different materials and pipeline layouts compared to [HFM10] and [WDR11]

BTF sample	Decompression rate [million BTF evaluations/s]			
	1D	2D	3D	4D
corduroy	120.41	118.86	116.03	116.74
impalla	109.43	102.45	125.11	146.65
proposte	110.88	114.44	115.38	114.97
pulli	78.01	84.72	89.54	88.29
wallpaper	112.29	120.81	123.89	144.91
wool	90.19	91.68	90.25	89.67
ceilingHDR	102.14	114.12	142.00	182.78
floortileHDR	78.97	87.31	108.63	139.55
pinktileHDR	195.33	204.02	262.86	315.57
walkwayHDR	81.80	99.85	125.51	160.19
corduroy01	137.57	142.02	171.15	211.41
fabric02	109.09	110.27	106.51	108.75
fabric03	88.15	97.24	102.70	103.02
wood01	136.41	142.06	136.32	176.96
Average	110.76	116.42	129.71	149.96

Table 8.5: Decompression rates for different materials and pipeline layouts.

Chapter 9

Conclusions

In this chapter, we summarize the main contributions of our work and propose possible future work on the subject.

9.1 Summary

After studying the original MLVQ-based algorithm for BTF data compression, we proposed several ways to improve the algorithm in terms of both the compression quality and the compression speed. We described how parts of the algorithm can be efficiently parallelized and modified to run on a GPU or a multi-core CPU using the OpenCL heterogeneous computing framework. Using our implementation, we achieved an average compression time of 2.8 hours per material, which is roughly 9.5 times faster than the original single-core CPU implementation, while retaining the same visual quality.

We incorporated the improved algorithm into a highly modular and configurable compression pipeline. The pipeline allows different materials to be compressed using different compression schemes to achieve better compression ratios. We showed, that a single compression scheme is not optimal for all materials and the difference in compression ratios between the original and a modified scheme can be as high as 33% (13% on average). We also demonstrated that the compression ratio can be further increased by the use of entropy-based approaches, such as Huffman coding, which resulted in an average improvement of 17% over the previously used optimizations. The compression pipeline also has direct support for multispectral BTF data, although we did not evaluate this possibility.

We implemented the decompression algorithm as a separate library, which can be easily incorporated into a third-party application. The decompression process is fully parallel and can be efficiently run on a GPU or a multi-core CPU using the OpenCL framework. We incorporated the decompression library into a custom OpenGL-based previewer application, which allows real-time interactive rendering of a BTF-mapped scene. Using this application, we were able to achieve an average evaluation rate of 127 million individual BTF evaluations per second in a worst-case scenario. We also implemented a non-interactive preview generation tool, which allows data visualization from various stages of the compression.

9.2 Future Work

As our compression pipeline is highly modular and flexible, we believe these features can be further exploited. One of the possibilities is to study more pipeline layouts. These may include reducing the pipeline depth or treating all available color channels separately. Another possible field of study is the types of nodes available within the pipeline. These may include completely new node types or variations of the existing node type. For example the lowest-level nodes may be modified to not store the representative vectors directly, but use a curve-fitting algorithm to represent the vectors with less information to be stored. New types of modifiers may also be studied, which exploit for example axis or rotation symmetries of the data. Although incorporated into the pipeline, processing of multispectral BTF data was not evaluated and should also be an interesting topic to explore. As the role of a similarity metric used is crucial during the compression, the effects of using different metrics should be studied. Finally, the methods of optimizing the compressed material codebooks can be explored. These may include pruning of the least used entries from the codebooks, or exploiting similarities between the data of the entries stored in the codebook.

Bibliography

[BTFBASE] BTFBASE project website. <http://dcgi.felk.cvut.cz/home/havravla/btfbase/>. Accessed: 2014-04-29.

[CBC+01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and Representation of 3D Objects with Radial Basis Functions. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 67–76, New York, NY, USA, 2001. ACM.

[CMAKE] CMake - Cross Platform Make. <http://www.cmake.org/>. Accessed: 2014-05-08.

[DLHS01] K. Daubert, H. P. A. Lensch, W. Heidrich, and H.-P. Seidel. Efficient Cloth Modeling and Rendering. In *Proceedings of the 12th Eurographics Conference on Rendering*, EGWR'01, pages 63–70, Aire-la-Ville, Switzerland, Switzerland, 2001. Eurographics Association.

[DvGNK99] K. Dana, B. van Ginneken, S. Nayar, and J. Koenderink. Reflectance and Texture of Real-world Surfaces. *ACM Trans. Graph.*, 18(1):1–34, January 1999.

[Ege13] P. Egert. Efficient GPU-based Decompression of BTF Data Compressed using Multi-Level Vector Quantization. In *CESCG'13: Proceedings of Central European Seminar on Computer Graphics*, pages 1–8, 2013.

[FH05] J. Filip and M. Haindl. Efficient Image-Based Bidirectional Texture Function Model. In *Texture 2005*, pages 7–12. Heriot-Watt University & IEEE, 2005.

[FH09] J. Filip and M. Haindl. Bidirectional Texture Function Modeling: A State of the Art Survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(11):1921–1940, November 2009.

[GG91] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[GHK+13] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[Hal64] J. H. Halton. Algorithm 247: Radical-inverse Quasi-random Point Sequence. *Commun. ACM*, 7(12):701–702, December 1964.

[HF07] M. Haindl and J. Filip. Extreme Compression and Modeling of Bidirectional Texture Function. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(10):1859–1865, October 2007.

[HFM10] V. Havran, J. Filip, and K. Myszkowski. Bidirectional Texture Function Compression Based on Multi-Level Vector Quantization. *Computer Graphics Forum*, 29(1):175–190, 2010.

[HFV12] M. Haindl, J. Filip, and R Vavra. Digital Material Appearance: The Curse of Tera-bytes. *ERCIM News*, 90:49–50, 2012.

[Huf52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[IEEE754] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[ITU601] International Telecommunication Union. Recommendation BT.601 : Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios. <`http://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf`>. Accessed: 2014-05-07.

[KM06] N. Kawai and K. Matsufuji. Azimuth-rotated Vector Quantization for BTF Compression. In *ACM SIGGRAPH 2006 Research Posters*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[KMBK03] M. Koudelka, S. Magda, P. Belhumeur, and D. Kriegman. Acquisition, compression, and synthesis of bidirectional texture functions. In *In ICCV 03 Workshop on Texture Analysis and Synthesis*, 2003.

[Lar98] G. W. Larson. LogLuv Encoding for Full-gamut, High-dynamic Range Images. *J. Graph. Tools*, 3(1):15–31, March 1998.

[LFTG97] E. P. Lafortune, S.-C. Foo, K. E. Torrance, and D. P. Greenberg. Non-linear Approximation of Reflectance Functions. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 117–126, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LHZ+04] X. Liu, Y. Hu, J. Zhang, X. Tong, B. Guo, and H.-Y. Shum. Synthesis and Rendering of Bidirectional Texture Functions on Arbitrary Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(3):278–289, May 2004.

[LM01] T. Leung and J. Malik. Representing and Recognizing the Visual Appearance of Materials Using Three-dimensional Textons. *Int. J. Comput. Vision*, 43(1):29–44, June 2001.

[MCC+04] W.-C. Ma, S.-H. Chao, B.-Y. Chen, C.-F. Chang, M. Ouhyoung, and T. Nishita. An Efficient Representation of Complex Materials for Real-time Rendering. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '04, pages 150–153, New York, NY, USA, 2004. ACM.

[MK06] S. Magda and D. Kriegman. Reconstruction of Volumetric Surface Textures for Real-time Rendering. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR'06, pages 19–29, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[MLH02] D. K. McAllister, A. Lastra, and W. Heidrich. Efficient Rendering of Spatial Bi-directional Reflectance Distribution Functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 79–88, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[MMK03] J. Meseth, G. Müller, and R. Klein. Preserving Realism in real-time Rendering of Bidirectional Texture Functions. In *OpenSG Symposium*, pages 89–96, 2003.

[MMS+05] G. Müller, J. Meseth, M. Sattler, R. Sarlette, and R. Klein. Acquisition, Synthesis, and Rendering of Bidirectional Texture Functions. *Computer Graphics Forum*, 24(1):83–109, 2005.

[RK09] R. Ruiters and R. Klein. BTF Compression via Sparse Tensor Decomposition. In *Proceedings of the Twentieth Eurographics Conference on Rendering*, EGSR'09, pages 1181–1188, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association.

[SB06] H. R. Sheikh and A. C. Bovik. Image Information and Visual Quality. *IEEE Trans. Img. Proc.*, 15(2):430–444, February 2006.

[SG31] T. Smith and J. Guild. The c.i.e. colorimetric standards and their use. *Transactions of the Optical Society*, 33(3):73, 1931.

[SSK03] M. Sattler, R. Sarlette, and R. Klein. Efficient and Realistic Visualization of Cloth. In *Proceedings of the 14th Eurographics Workshop on Rendering*, EGRW '03, pages 167–177, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[STEAM] Steam Hardware & Software Survey. <`http://store.steampowered.com/hwsurvey/`>. Accessed: 2014-04-29.

[OPENCL] The Khronos Group. The OpenCL Specification, version 1.1. <`http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`>. Accessed: 2014-05-08.

[TZL+02] X. Tong, J. Zhang, L. Liu, X. Wang, B. Guo, and H.-Y. Shum. Synthesis of Bidirectional Texture Functions on Arbitrary Surfaces. *ACM Trans. Graph.*, 21(3):665–672, July 2002.

[ATRIUM] Universität Bonn - Institute of Computer Science II. ATRIUM BTF Datasets. <`http://cg.cs.uni-bonn.de/en/projects/btfdbb/download/atrium/`>. Accessed: 2014-05-08.

[VT04] M. Alex O. Vasilescu and D. Terzopoulos. TensorTextures: Multilinear Image-based Rendering. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 336–342, New York, NY, USA, 2004. ACM.

[WBSS04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Trans. Img. Proc.*, 13(4):600–612, April 2004.

[WDR11] H. Wu, J. Dorsey, and H. Rushmeier. A Sparse Parametric Mixture Model for BTF Compression, Editing and Rendering. *Computer Graphics Forum*, 30:465–473, 2011.

Appendix A

List of Abbreviations

BRDF	Bidirectional Reflectance Distribution Function
BTF	Bidirectional Texture Function
CPU	Central Processing Unit
GLSL	OpenGL Shading Language
GPGPU	General Purpose computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HDR	High Dynamic Range
LDR	Low Dynamic Range
MLVQ	Multi-Level Vector Quantization
MSE	Mean Squared Error
MSSIM	Mean Structural Similarity Index Metric
PDF	Probability Density Function
PSNR	Picture Signal-to-Noise Ratio
RBF	Radial Basis Function
RGB	Red-Green-Blue
SP	Stream Processing
SSIM	Structural Similarity Index Metric
SVD	Singular Value Decomposition
UBO	University of Bonn
UTIA	Institute of Information Theory and Automation
VQ	Vector Quantization
XML	eXtensible Markup Language

Appendix B

Image Gallery

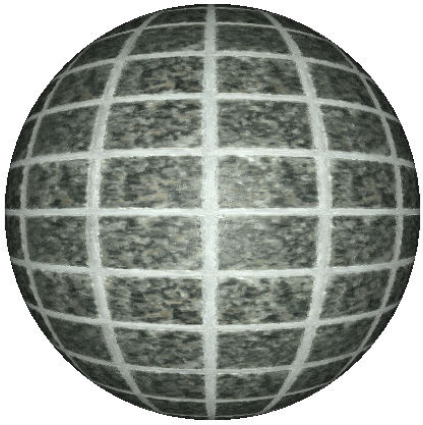
(a) Original

(b) Compressed

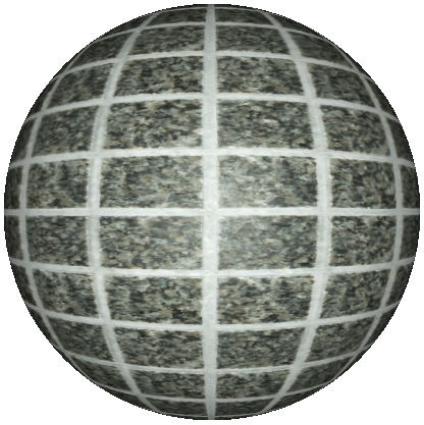
(c) Difference (×4)

(d) Environment map lighting

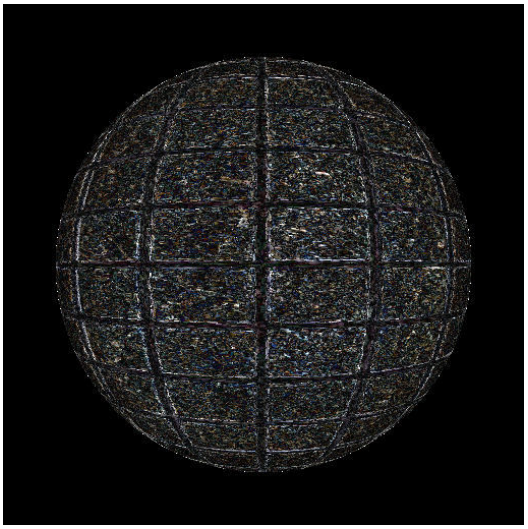
Figure B.1: Example renderings of the *Corduroy* BTF



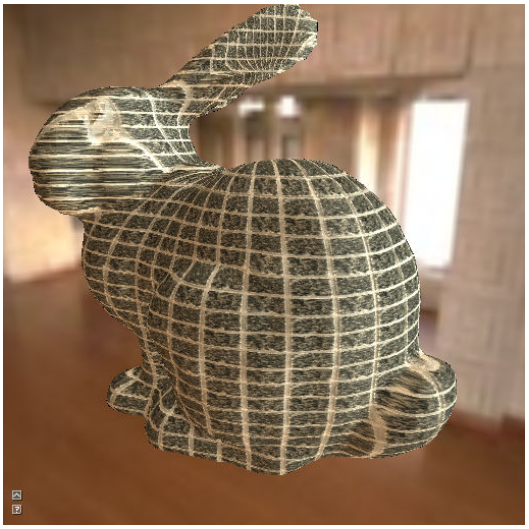
(a) Original



(b) Compressed

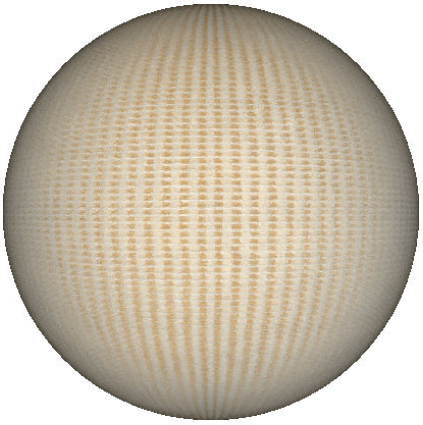


(c) Difference (×4)

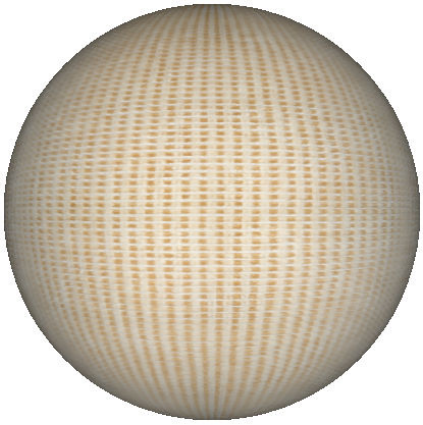


(d) Environment map lighting

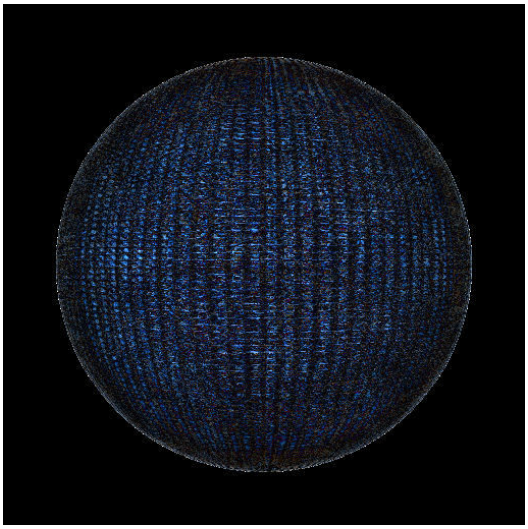
Figure B.2: Example renderings of the *Impalla* BTF



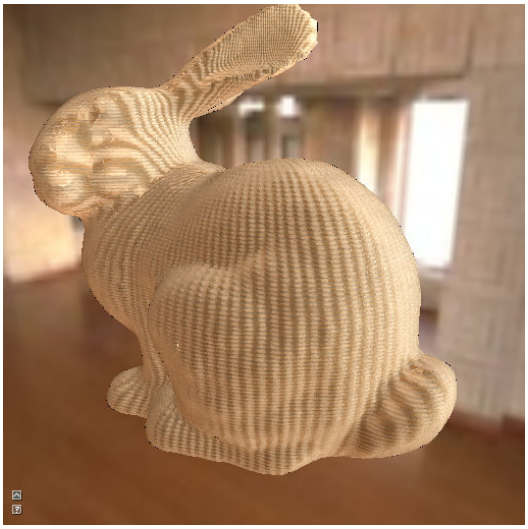
(a) Original



(b) Compressed



(c) Difference (×4)

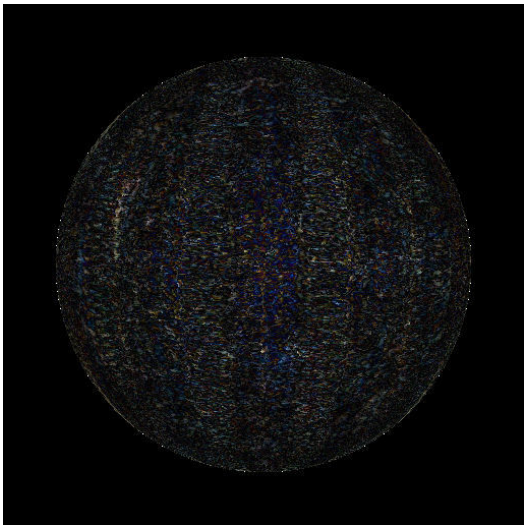


(d) Environment map lighting

Figure B.3: Example renderings of the *Proposte* BTF

(a) Original

(b) Compressed



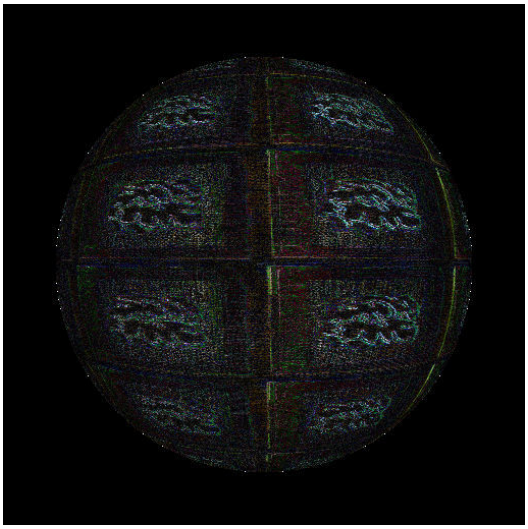
(c) Difference (×4)

(d) Environment map lighting

Figure B.4: Example renderings of the *Pulli* BTF

(a) Original

(b) Compressed



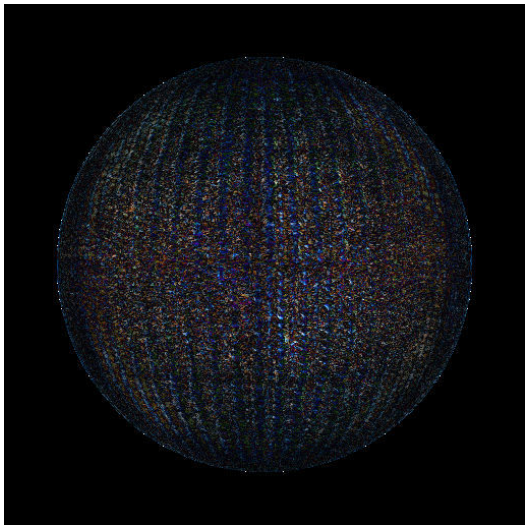
(c) Difference (×4)

(d) Environment map lighting

Figure B.5: Example renderings of the *Wallpaper* BTF

(a) Original

(b) Compressed



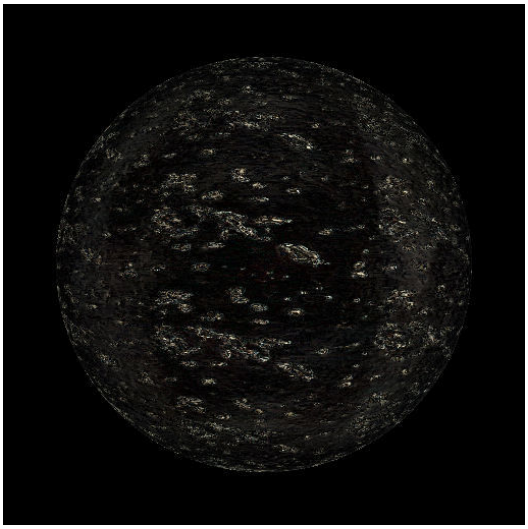
(c) Difference (×4)

(d) Environment map lighting

Figure B.6: Example renderings of the *Wool* BTF

(a) Original

(b) Compressed



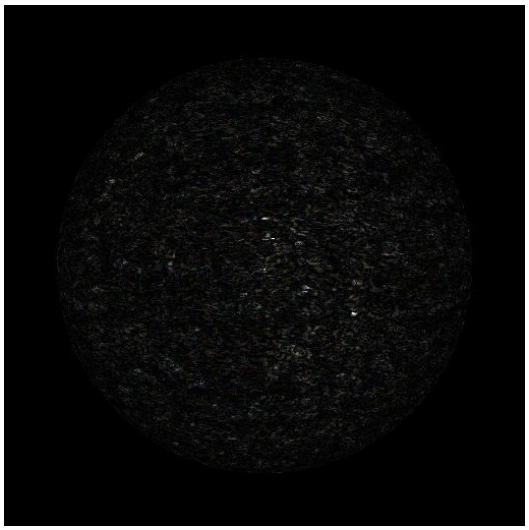
(c) Difference (×4)

(d) Environment map lighting

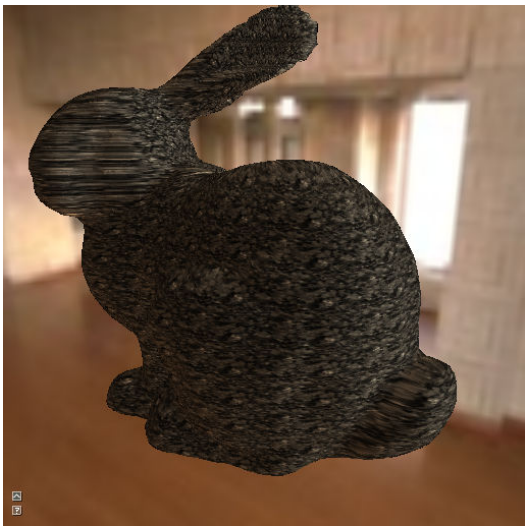
Figure B.7: Example renderings of the *CeilingHDR* BTF

(a) Original

(b) Compressed



(c) Difference (×4)

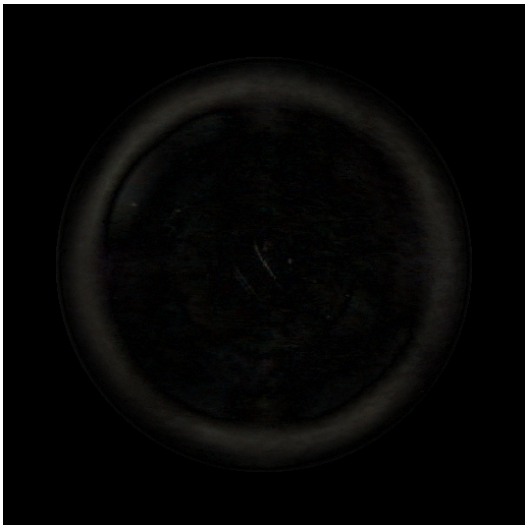


(d) Environment map lighting

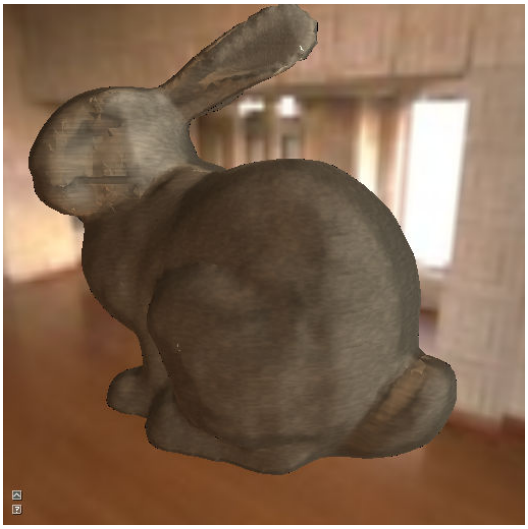
Figure B.8: Example renderings of the *FloortileHDR* BTF

(a) Original

(b) Compressed



(c) Difference (×4)

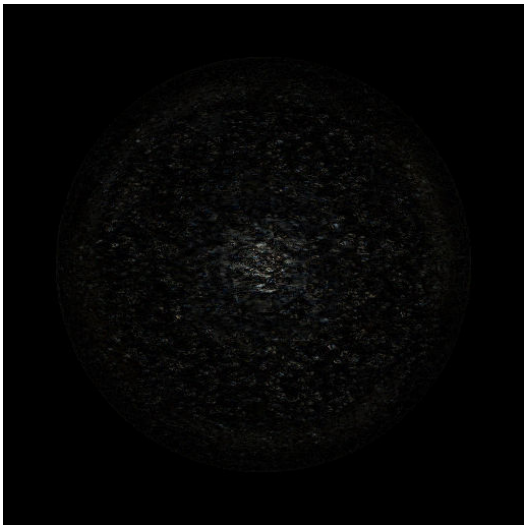


(d) Environment map lighting

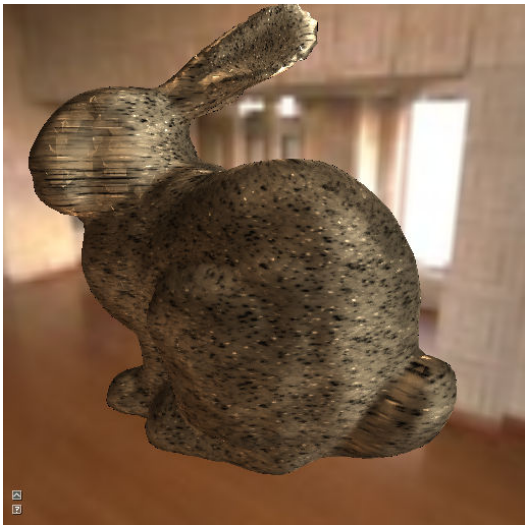
Figure B.9: Example renderings of the *PinktileHDR* BTF

(a) Original

(b) Compressed



(c) Difference (×4)

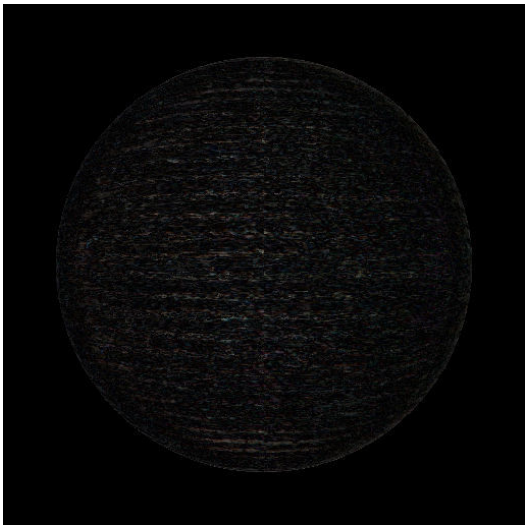


(d) Environment map lighting

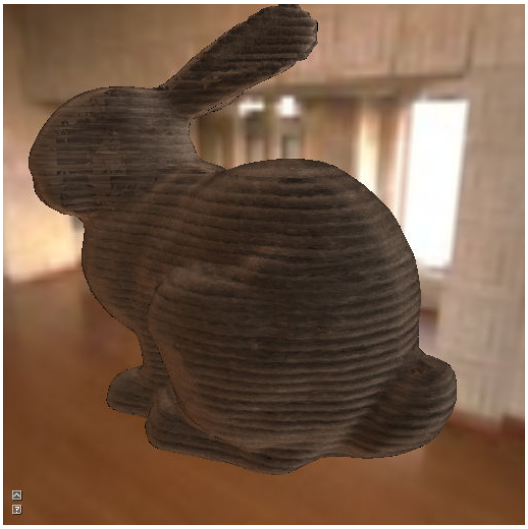
Figure B.10: Example renderings of the *WalkwayHDR* BTF

(a) Original

(b) Compressed



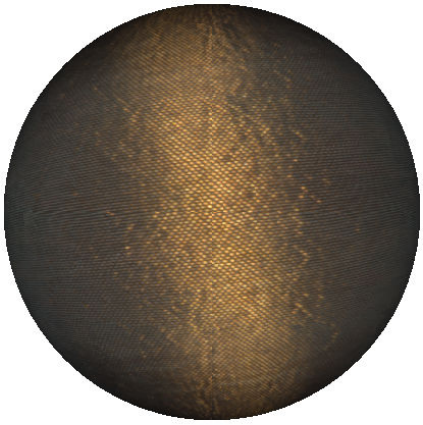
(c) Difference (×4)



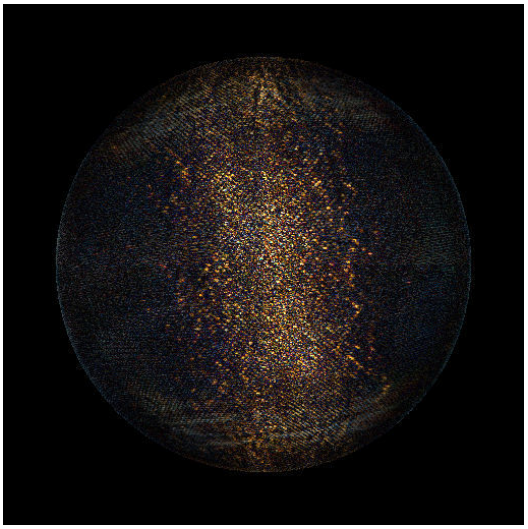
(d) Environment map lighting

Figure B.11: Example renderings of the *Corduroy01* BTF

(a) Original



(b) Compressed



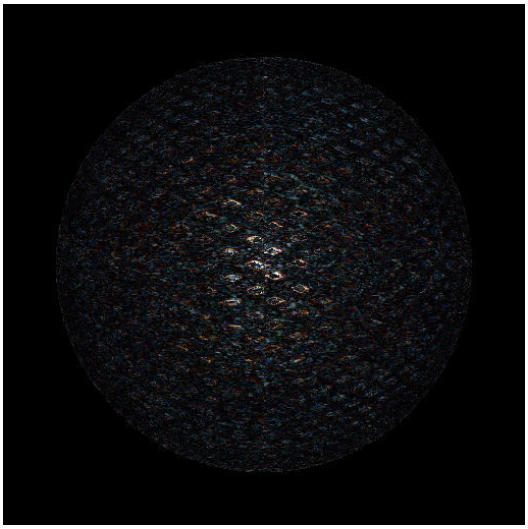
(c) Difference (×4)

(d) Environment map lighting

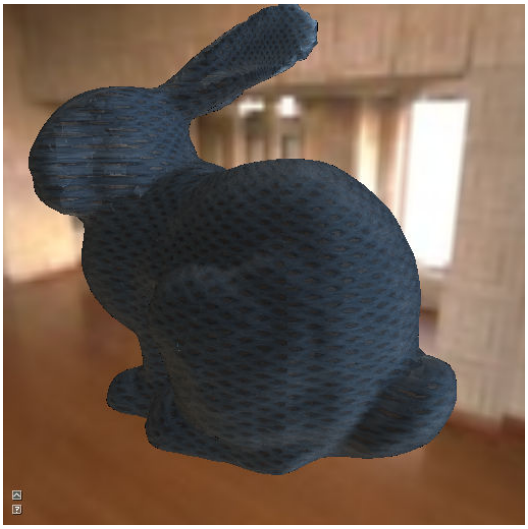
Figure B.12: Example renderings of the *Fabric02* BTF

(a) Original

(b) Compressed



(c) Difference (×4)

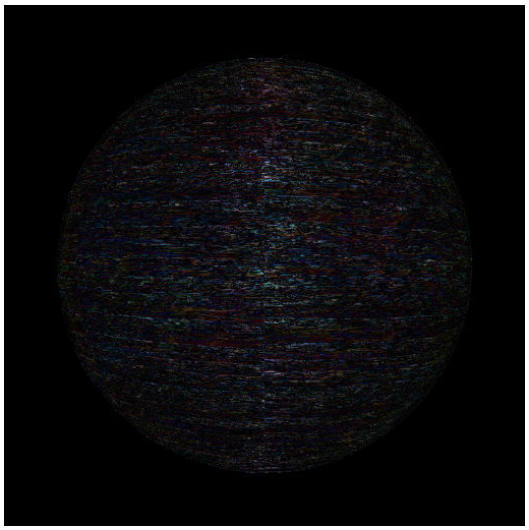


(d) Environment map lighting

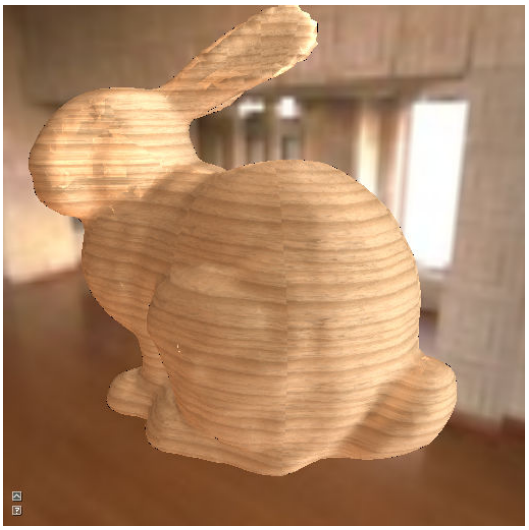
Figure B.13: Example renderings of the *Fabric03* BTF

(a) Original

(b) Compressed



(c) Difference (×4)



(d) Environment map lighting

Figure B.14: Example renderings of the *Wood01* BTF

Appendix C

Installation and User Manual

C.1 Build Instructions

The framework is written in ANSI C++ and should compile flawlessly using different compilers on different platforms. The target platform is required to be 64-bit because large amount of memory can be used by the framework. We successfully compiled the framework using Microsoft Visual Studio 2008 and MinGW64 TDM-GCC x64 4.7.1 compilers on Microsoft Windows platform and using native GCC 4.7.1 on the Ubuntu Linux platform.

Dependency	**Version**	**Homepage**
AntTweakBar	1.16	<http://anttweakbar.sourceforge.net/>
Assimp	3.0	<http://assimp.sourceforge.net/>
FLANN	1.6.11	<http://www.cs.ubc.ca/research/flann/>
FreeGLUT	2.8.1	<http://freeglut.sourceforge.net/>
GLM	0.9.4.6	<http://glm.g-truc.net/>
libjpeg-turbo	1.3.0	<http://libjpeg-turbo.virtualgl.org/>
libpng	1.6.6	<http://www.libpng.org/>
OpenCL	1.1	<https://software.intel.com/en-us/vcsource/tools/opencl-sdk>
pugixml	1.2	<http://pugixml.org/>
zlib	1.2.8	<http://www.zlib.net/>

Table C.1: Required external dependencies

We use the CMake build system [CMAKE] to manage the build process. To successfully compile the framework, external dependencies summarized in Table C.1 first need to be obtained. The easiest way to build the project is then to use the `cmake-gui` application shipped with CMake. This can be done using the following set of commands:

```
cd /path/to/project/root
cd build
cmake-gui ..
```

After executing the last command, the CMake GUI window should open and prompt for the compiler suite to use. When done selecting the preferred compiler, click the `Configure`

button to start the build configuration. If some of the required dependencies are not found, an error will be generated. The paths can then be specified manually within the CMake GUI window. When the configuration stage is passed without errors, press the `Generate` button to create the project files for the selected compiler. After running the compiler, project executables should be generated in the `bin` subdirectory of the project.

C.2 Usage

The compression algorithm implementation has three parts represented by three separate applications. Configuration templates are provided in the `data` subdirectory of the project for all components of the framework. Using these templates, specific material configuration must first be created for each of the components. The documentation of the configurable variables is provided directly within the templates.

To preprocess the raw input BTF data into a common format, use the *Preprocessor* tool by executing the following command from the project root directory:

```
bin\preprocessor.exe path\to\preprocessor_config.xml
```

When done, preprocessed data in *tempBTF* format should be created in the specified output directory (`temp` subdirectory by default). The next step is to convert the data to the *Onion-Slices* parameterization using the *Resampler* tool. This can be done by executing

```
bin\resampler.exe path\to\resampler_config.xml
```

from the project root directory.

Finally, the *Compressor* tool can be launched to perform the MLVQ compression. The compression can be started using the following command:

```
bin\compressor.exe path\to\compressor_config.xml
```

After finishing, the compressed material file should be generated in the configured output directory. All of the tools log their progress both to the system console and to a log file.

All other tools in the system are launched the same way, by providing the path to the configuration file as their first argument. The interactive previewer application can for example be launched using the following command:

```
bin\utils\previewer_int.exe path\to\previewer_int_config.xml
```

Similarly the offline preview image generation tool can be launched using the following command:

```
bin\utils\previewer.exe path\to\previewer_config.xml
```

Values of command line arguments can also be used in the configuration files. This allows for batch processing of multiple tasks without the need to modify the configuration files each time. An example of this feature is provided in the `data\configs\previewer_int_cli.xml` configuration file, which can be used to launch the interactive previewer application with the path to the material file passed using the command line, as shown on the following example:

```
bin\utils\previewer_int.exe data\configs\previewer_int_cli.xml data\btf\wool
```

Appendix D

Contents of Attached CD

```
BTFcut/                  - project root directory
|--bin/                    - precompiled Windows x64 binaries
|  \--utils/                 - binaries for non-compression related tools
|--build/                  - project build tree root directory
|--components/             - source code of project libraries
|  |--Evaluator/             - the decompression library
|  \--RasterizerOpenGL/      - OpenGL-based scene to BTF coordinates rasterizer
|--core/                   - source code of compression-related applications
|  |--Compressor/            - the MLVQ compressor
|  |--Preprocessor/          - raw input data preprocessor
|  \--Resampler/             - Onion-Slices parameterization resampler
|--data/                   - application data
|  |--btf/                   - compressed BTF materials
|  |--configs/               - configuration files and templates
|  |--OpenCL/                - source code of OpenCL kernels
|  |--scene/                 - testing scenes
|  \--shaders/               - GLSL shaders used by OpenGL rasterizer
|--depend/                 - third-party libraries should be placed here
|--doc/                    - project documentation
|  |--doxygen/               - doxygen generated source code documentation
|  \--thesis/                - source data for this thesis
|--lib/                    - precompiled binaries of the component libraries
|--library/                - source code for the common utility library
|--temp/                   - reserved for temporary data produced by the application
\--utils                   - source code of non-compression related tools
   |--Previewer/             - the non-interactive previewer applciation
   \--PreviewerInteractive/ - the interactive previewer applciation
```

A separate `readme.txt` file is provided in each of the subdirectories.

Appendix E

Configuration File Example

An example of configuration file content for the *Compressor* component is provided below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<BTFcut version="1.0">
  <Compressor>
    <OnionBTF>
      <Source>
        BTFCUT_TEMP_DIR/Corduroy_onionBTF/Corduroy.onionBTF
      </Source>
    </OnionBTF>
    <OpenCL
      platform="0"
      device="0"
      programsDir="BTFCUT_DATA_DIR/OpenCL/Compressor"
    />
    <Pipeline src="pipelines/pipeline_full.xml" />
    <Stages src="compressor_stages_test.xml" />
    <WindowManager>
      <Window size="11" />
    </WindowManager>
    <CodeBTF>
      <Target>BTFCUT_TEMP_DIR/Corduroy.codeBTF</Target>
      <Options optimizeCodebooks="false"/>
    </CodeBTF>
    <Debug saveCompareImages="true" />
    <StateBTF>
      <Target>BTFCUT_TEMP_DIR/Corduroy_stateBTF</Target>
    </StateBTF>
  </Compressor>
</BTFcut>
```