Efficient GPU-based Decompression of BTF Data Compressed using Multi-Level Vector Quantization

Petr Egert[*]
Supervised by: Vlastimil Havran[†]

Faculty of Electrical Engineering, DCGI
Czech Technical University
Prague / Czech republic

Abstract

One of the main drawbacks of Bidirectional Texture Function (BTF), as a method of capturing realistic and accurate real-world material appearance, is the resulting size of the measured data set. Several lossy methods to compress the data were proposed over the years to cope with this problem. To efficiently use the compressed data an appropriate decompression algorithms are also needed, allowing fast random synthesis of BTF data without the need to reconstruct the whole BTF back to its original representation. One of such methods based on multi-level vector quantization and providing both good compression ratio and random access from the compressed data was proposed by Havran et al. in 2010. In this paper, we would like to share our experience with writing a GPU based implementation of the decompression part of the aforementioned method. Our goal was to evaluate the implementation difficulty, as well as the resulting performance and suitability of the algorithm for real-time use.

Keywords: BTF, bidirectional texture function, decompression, GPU, OpenCL, Multi-level vector quantization

1 Introduction

Capturing and accurately representing real-world material appearance remains one of the major challenges in computer graphics nowadays. In real-time applications, such as computer games, this challenge gets even harder, because of the limited resources available to store and apply the material to the resulting scene. The common approach is to sacrifice physical correctness for performance and achieve visually pleasing results.

As new methods to represent the material appearance have been discovered and the performance of the computer hardware increases, some also suitable for the real-time use. One such method is Bidirectional Texture Function (BTF). First introduced in [2], a monospectral BTF is a six-dimensional function returning the amount of light reflected by an arbitrary point on the material surface, when illuminated and viewed from arbitrary directions. By extending the function to a given color-space, a seven-dimensional, multispectral BTF is obtained. BTF can also be imagined as a planar texture, where the amount of light reflected from each individual texel also depends on the view and illumination directions of the texel[1].

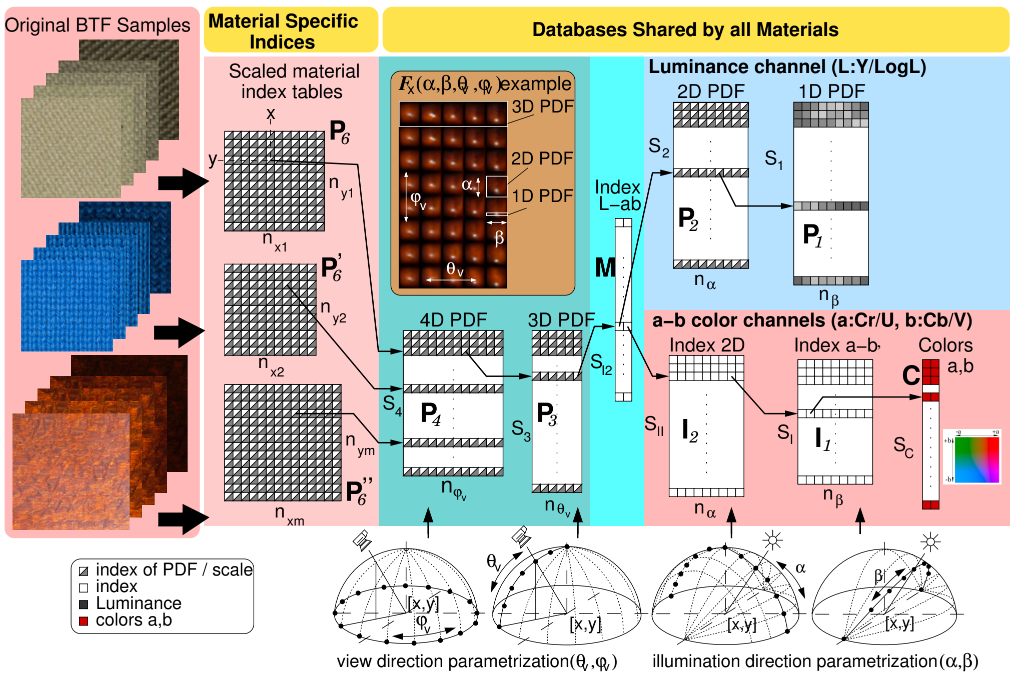
One of the main advantages of BTF over simpler methods is the ability to preserve information about the material structure, including properties such as anisotropy, masking or self-shadowing. The main drawback is the size of the measured data set, which in raw form can take up to several gigabytes of space for a single material sample. This would render the method virtually useless for real-time use. Therefore several compression schemes were proposed, some of them allowing even for real-time use.

An in-depth survey of available BTF compression schemes was provided by Filip et al. [3], including the fitness of the BTF evaluation part of the studied algorithms for fast GPU-based implementation. More recently, a novel BTF compression scheme based on multi-level vector quantization was proposed by Havran et al. [4]. In the paper, the authors state fast random access data synthesis and convenient GPU implementation as two of their design goals. They also evaluate the performance of the proposed algorithm by providing a GLSL [10] based GPU implementation.

Our goal in this paper is to summarize the lessons learned while writing our own implementation of the data synthesis part of the scheme described in [4] using OpenCL [6] as our framework of choice, instead of GLSL used in the original paper. We describe the algorithm in detail and evaluate the performance of our implementation against both a CPU and the original GLSL variant using our own OpenGL-based [1] sample application. Finally, we discuss suitability of the algorithm for real-time use.

[*]egertpet@fel.cvut.cz
[†]havran@fel.cvut.cz

[1]It is worth noting, that while for a fixed spatial position the remaining arguments of a BTF are the same as for Bidirectional Reflectance Distribution Function (BRDF), Helmholz reciprocity does not apply in this case due to structural properties of the material, which is the key difference between BTF and Spatially Varying BRDF (SVBRDF).



Original BTF Samples
Material Specific Indices
Databases Shared by all Materials
Scaled material index tables
Luminance channel (L:Y/LogL)
2D PDF
1D PDF
3D PDF
4D PDF
Index L–ab
a–b color channels (a:Cr/U, b:Cb/V)
Index 2D
Index a–b
Colors a,b
index of PDF / scale
index
Luminance
colors a,b
view direction parametrization(θv,φv)
illumination direction parametrization(α,β)

Figure 1: Multi-level vector quantization BTF compression model scheme *(Image courtesy of [4])*.

2 Compression algorithm overview

In this section, we recall the basic outline of the compression method proposed in [4]. For a more detailed description of the algorithm please refer to the original paper.

The basic concept of the compression method is shown in Figure 1. The process starts by resampling the raw measured BTF data according to another parametrization. By using the planar material position as index, the BTF can be decomposed into a set of individual 4D texels (called *apparent BRDFs*). Using a fixed number of steps to quantize each of the four dimensions, each of the texels is resampled into the 'onion-slices' parametrization described in [4]. Additionally a specific data arrangement is used, as shown in the orange area of Figure 1. This arrangement allows to treat differently sized slices of the data as conditional probability density functions (PDFs), which are then used as input vectors to the multi-level vector quantization (MLVQ) scheme.

The MLVQ scheme extends the concept of a basic vector quantization (VQ) compression method, where the whole set of input PDF vectors is represented by a smaller representative subset, a codebook. In MLVQ, the VQ process is applied to progressively smaller subsets at different levels of the original data, thus forming a set of codebooks. Each level corresponds to a single dimension of the resampled BTF data. This is shown in Figure 1; the original monospectral BTF has 6 dimensions. 4D PDF is specified by the planar material position. By further specifying the view azimuthal angle, a 3D PDF is obtained. Similarly, 3D PDF is decomposed into 2D PDFs etc.

An integral part of any VQ-based compression method is a similarity measure. A similarity measure is a function returning the level of similarity between two sets of data. In VQ, this function is used to compare the currently processed input vector with the entries already present in the codebook. If a similar enough (depending on a predefined threshold and possibly other conditions) entry is found in the codebook, it is returned as a representative vector for the given input vector. If no entry satisfying all the defined conditions is found, the input vector is added as a new entry into the codebook. Since the original and the representative vector need only to be similar to each other (not exactly the same), the VQ is considered a lossy compression method. The similarity measure and its properties are therefore of critical importance, since they define the relation between the quality of the compressed data and the overall compression ratio. In [4], the authors describe structural similarity index measure (SSIM) [8] as their method of choice.

The compression algorithm itself works by iterating over all the available texels of the resampled BTF data and performing MLVQ on each of them. For each such texel a corresponding entry in the top level 6D-level table P_6 needs to be created. As described earlier, each texel can be treated as a 4D PDF. The MLVQ process starts by finding a similar entry in the 4D level codebook P_4. If such an entry is found, its index is returned and stored in the proper location (corresponding to the planar position of the texel) of P_6.

To achieve better hit-rate and therefore better compression ratios, the search for a similar entry in the P_4 codebook is performed 'up to scale'. Both the input and the compared vectors are normalized to have the same overall luminance. If the normalized vectors are found to be similar enough, the index of the entry and scaling coefficient for the data are stored in the P_6 table.

If an entry for the given input vector is not found in the P_4 codebook, a new entry needs to be created. Contrary to the basic VQ process, in MLVQ the higher-level codebooks do not directly store the raw data. Instead, the individual codebooks form a hierarchy, where each entry in the codebook consists of a set of indices into a lower level codebook. The raw data are then only stored in the lowest levels and can be reconstructed back to the original form by the means of chained indexing through the whole hierarchy.

To create a new entry in the 4D-level P_4 codebook, the original 4D PDF is first split into a set of 3D PDF slices. Each slice is specified by the planar position of the texel and the the view azimuthal angle φ_v, thus forming a 3D PDF. The process continues by iterating over all these slices and finding their corresponding entries in the 3D-level P_3 codebook. As a result, a row of index/scale entries is obtained, where each index points to a row in the P_3 codebook. The number of entries in the row corresponds to the number of steps in which the view azimuthal direction is quantized. This row is then stored as a new entry in the P_4 codebook.

The compression algorithm remains almost the same on all levels - if a similar entry is not found in the current level codebook, the current PDF is split into smaller slices, which are then searched in a lower-level codebook. Some exceptions however apply to this process. From 2D level down, the BTF data are converted from the original RGB to a more perceptually uniform color model (YCbCr for LDR samples, LogLuv [9] for HDR samples). To achieve better compression ratios, the luminance and chrominance channels start to be treated separately on these levels. Therefore two different sets of codebooks are used - P_2, P_1 for luminance information and I_2, I_1, C for chrominance information. Additionally, scaling coefficients are only used for the luminance data. Entries in the I_2 and I_1 codebooks therefore consist only of indices, not index/scale pairs. To provide a relation between the two separate channels, an additional codebook, marked M in Figure 1, is used. By using this codebook, a matching pair of luminance and chrominance data can be obtained and converted back to the original RGB color model later on.

In the lowest-level codebooks (P_1 and C), there are no other codebooks to refer to. The representative vectors of the resampled BTF data are stored in the 1D array. This completes the codebook hierarchy and serves as the final stage of the MLVQ process. As shown in Figure 1, a minimum of nine codebooks, arranged in a tree-like hierarchy, is used in total to store a single material. The method also allows multiple materials to share the same set of codebooks, each requiring only one additional 6D level codebook. By means of chained indexing through the hierarchy, the BTF can be evaluated directly from the compressed data.

3 Decompression algorithm

The decompression part of the method is described only briefly in sec. 4.6 of [4]. In this section we would like to provide a more in-depth analysis of the problem.

The compressed BTF evaluation process is based on chained indexing among the data codebooks. The algorithm starts to look up the corresponding entry in the highest level codebook P_6 by using the planar spatial coordinates x and y. This entry consists of a row index in the lower level P_4 codebook and a scaling coefficient, which is later used to multiply the obtained value.

The algorithm then descends to the lower level, P_4 codebook. Here the index obtained from the upper level entry is used to find the corresponding row of data. Then the view direction azimuthal angle φ_v is used to find the corresponding entry in the row. As in the previous case, the entry consists of a row index in the lower level P_3 codebook and a scaling coefficient. After obtaining the entry, the algorithm descends to P_3 level. Here the same operation takes place, except that the polar angle θ_v of view direction is used to find the corresponding entry in the row. As before an index in the lower level M codebook and a scaling coefficient is retrieved.

As described in Section 2, starting from 2D level a more perceptually uniform color model is used and luminance and chrominance data start to be treated separately. The M codebook is used to merge the separate color model components back to the original representation. Each entry in the M codebook contains two indices - one used to index the P_2 codebook with luminance data and one to index the I_2 codebook with chrominance data. No scaling coefficients are stored at this level. A color conversion back to RGB color space then needs to take place after obtaining both the luminance and chrominance data from the lower level codebooks.

On lower levels, the BTF evaluation remains essentially the same as on the higher levels. The illumination direction, transformed into the 'onion-slices' parametrization, is used to index the lower level codebooks. To transform ordinary spherical coordinates of the illumination direction into the new parametrization, the following equations can be used [4]:

$$\begin{aligned}\beta &= \arcsin(\sin\,\theta_i \cdot \cos(\varphi_i - \varphi_v)) \\ \alpha &= \arccos\left(\frac{\cos\,\theta_i}{\cos\,\beta}\right)\end{aligned} \tag{1}$$

Upon reaching the bottom-most level codebooks P_1 and C, the representative BTF values are directly taken from the 1D array. Returning the codebook hierarchy back to the top, all the scaling coefficients need to be applied to the values. As stated before, a color conversion to the RGB color space takes place after returning to the M codebook. When the top-level P_6 codebook is reached, the evaluation process is complete and the resulting value is returned.

3.1 BTF coordinate interpolation

To simplify the algorithm description, we mention that only a single entry is used from each codebook. Although this approach would work in theory, it would produce visible artifacts in the resulting image, as the number of discrete steps used to quantize each dimension of BTF is relatively small. To overcome this problem, at least two entries closest to the specified coordinates need to be used at all level codebooks. Interpolation then needs to be performed on the values to minimize the quantization artifacts. A simple linear interpolation proved to be good enough to provide satisfactory results. On the 6D level, the interpolation is not necessary, but improves the overall quality of the result (similar to use of filtering with ordinary textures).

4 OpenCL specifics

4.1 Memory layout

To successfully evaluate a BTF sample, the codebooks containing the compressed data need to be accessible by the GPU. Since the data size of the codebooks is relatively small (tens of megabytes, depending on the material used), it is best to store them directly in the available device based memory (further we use the terminology used in OpenCL [6]). To reduce the number of memory regions and therefore the number of arguments in the function calls, we decided not to store each codebook in a separate memory region, but to pack all the codebooks into one continuous address space. Since the individual codebooks form a tightly connected database and should never need to be updated separately, this approach should not bring any major drawbacks.

Because two different types of information have to be stored, two separate memory regions with distinct data types are used. The first region uses a single precision floating point data type and stores the scaling coefficients of the higher-level codebooks (namely P_6, P_4, P_3, P_2), as well as the representative BTF data vectors contained in the bottom level P_1 and C codebooks. The second region is of 32-bit unsigned integer data type and is used to store data indices for the P_6, P_4, P_3, M, P_2, I_2 and I_1 codebooks.

For both memory regions, we remember offsets to the beginning of the individual codebooks. This allows us to directly index throughout the original data by only adding the corresponding offset. Another approach would be to recompute the indices to incorporate the offset directly into the codebooks, which would eliminate the need to store the offsets. Since additional information about the material (such as the number of quantization steps in individual dimensions) is still required to be known by the implementation, we decided to keep the offsets separately. This also reduces the amount of preprocessing required to load the material. A third memory region is therefore used to store all the additional information about the material, as well as the offsets of the individual codebooks. Since this memory region is only a few tens of bytes in size, it is created within the constant address space and later transferred into the private address space to achieve fast access.

Region	Data type	Size	Address space
Scales	float	by material	global
Indices	uint	by material	global
Constants	float/uint	fixed	constant/private

Table 1: OpenCL memory regions used by the evaluation process

These three memory regions contain all the information required to successfully evaluate the BTF directly from the compressed data. A quick summary is provided in Table 1.

4.2 BTF evaluation

The evaluation algorithm is based on chained indexing through all the codebooks. Upon reaching the lowest levels, a representative value is obtained, which is then multiplied by all the scaling coefficients acquired during the descent through the codebooks. A recursive approach may seem like a good solution to this problem, which however is not directly supported by the OpenCL standard. There are also several additional caveats with this approach. Luminance and chrominance components are evaluated differently in lower levels followed by color model conversion. Furthermore, interpolation between the two discrete values closest to the required coordinate must be performed in order to obtain visually appealing results. This proves even more difficult, as some of the variables need to be treated as cyclic (such as the view azimuthal angle φ_v), while other need not to (such as the view direction polar angle θ_v). As a result, the evaluation process is slightly different for each codebook.

For all these reasons, we have decided to implement access to each individual codebook by the means of separate functions. This also provides the resulting code a more human-readable structure. The basic purpose of each of these functions is to return the (partially) reconstructed BTF data at the specified coordinates and level. This process starts by indexing the codebooks at the given level and calling the lower level evaluation functions to obtain the two values closest to the required position. The values are scaled using the appropriate scaling coefficients and then a linear interpolation is performed to obtain a single resulting value. To obtain the final BTF value, this process needs to take place on all the codebook levels. Additionally, a color conversion to the RGB color model happens during access to the M codebook.

In our implementation, the BTF evaluation is computed by a single function. By providing locations of the 3 material memory regions and values for all the 6 BTF coordinate arguments, the sampling function returns the resulting BTF value at the given planar location as a color triplet in the RGB color space. A conversion from the standard spherical coordinates to the 'onion-slices' notation is handled directly inside the function. This allows easy integration of the MLVQ compressed BTF evaluation process into a custom project. We support this statement by providing an OpenGL based demonstration application.

4.3 Memory bandwidth requirements

As two closest samples need to be interpolated on each level of codebooks, the number of memory accesses increases exponentially with the increasing codebook level. To evaluate a single BTF query, a total of 47 integer and 63 float memory read operations is required [4], resulting in 440 bytes read per evaluation, if stored using 4 byte data types. If a bilinear interpolation is also performed on the 6D-level, then four times more data are read per evaluation. Additionally, when not computed on the fly, the input coordinates need to be read from a memory location and a result written, resulting in an additional increase in the required memory bandwidth.

5 Sample application

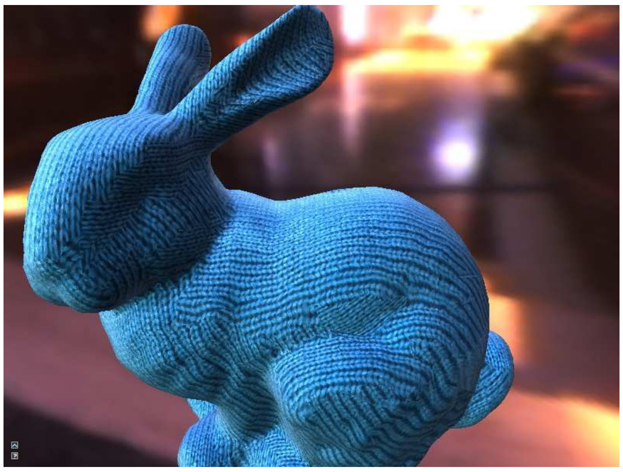


Figure 2: A geometric model covered by Wool BTF material, lit by an environment map, rendered by our sample application.

To verify the correctness and performance of our OpenCL-based MLVQ-compressed BTF evaluation code, we have designed and implemented a sample application. As shown on figure 3, the basic functionality of the application is to rasterize a scene containing a specified model, perform BTF evaluation on the rasterized data and finally to display the resulting image.

5.1 Rasterization process

To perform the BTF evaluation, all the input parameters required by the BTF are required for all the pixels. These parameters include the planar position on the material surface, view direction and illumination direction. All these six coordinates need to be specified in a local coordinate system of the material surface at rendered pixel. Programmable shaders are used to compute all these coordinates. In the vertex shader, the view and illumination direction vectors are transformed into tangent space and sent to the fragment shader. Projective transformation of the basic geometry is also handled by the vertex shader. In the fragment shader, the view and illumination vectors are reparametrized over the hemisphere, as required by the BTF evaluation function. A set of texture coordinates is used to define the planar material position.

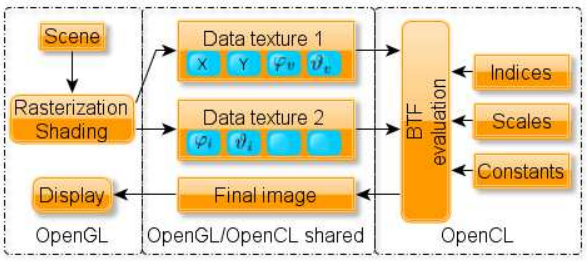


Figure 3: Basic workflow of the sample application.

5.2 Memory arrangement

To store the rasterized data, two floating point (GL_RGBA32F) render buffers are used. These buffers are of the same size as the main render buffer of the application window. In the first buffer, the planar position and view direction information is stored. In the second buffer, the illumination direction is stored. Since there are only six variables required to evaluate the BTF, two channels remain unused. We use multiple render targets (MRT) technique to fill both the buffers in a single pass.

To store the resulting image, a texture of the same size as the main render buffer is used. To correctly display the appearance of HDR materials and to be able to use this texture as an accumulation buffer for environment-map based lighting scheme, we use floating point data type for this texture as well. To display the result, we apply this texture to a full-screen quad and render it to the screen.

5.3 OpenGL-OpenCL communication

As the whole rasterization process is computed via OpenGL and our BTF evaluation function is written in OpenCL, a way to pass data between these two platforms is required. As OpenCL provides functions operating directly over OpenGL-owned data, this is the preferred way, since no copy operations is required. The OpenCL-OpenGL interoperability however is an optional feature, without guarantee of being supported by all the available OpenCL platforms. For this reason, an alternative data

Resolution	Total pixels	BTF evaluations	Coverage
800×600	480000	210848	43.9%
1280×720	921600	303636	32.9%
1920×1080	2073600	683088	32.9%

Table 2: Image resolution, count of BTF evaluations (i.e. rendered pixels), and screen coverage.

Single frame rendering time [ms]					
Material	Implementation				
	CPU	CLCPU	CLGL	RawCL	GLSL
	800×600 px resolution				
Impalla	281.2	26.3	12.5	5.1	1.9
FtHDR	483.2	28.7	14.9	6.3	2.0
Wool	295.4	29.2	13.7	5.7	2.3
WwHDR	494.7	28.3	14.8	6.4	2.0
WoolX	698.0	97.5	41.2	32.8	40.8
WwHDRX	1009.9	100.0	41.1	31.4	27.3
	1280×720 px resolution				
Impalla	408.8	47.4	22.5	7.5	2.1
FtHDR	711.5	51.5	24.7	9.6	2.0
Wool	422.4	51.4	21.3	9.1	2.5
WwHDR	708.2	52.1	26.1	11.3	2.0
WoolX	1008.8	172	55.7	51.8	46.7
WwHDRX	1428.6	179.4	54.4	45.3	31.6
	1920×1080 px resolution				
Impalla	998.5	106.3	42.3	13.9	4.2
FtHDR	1739.3	115.6	49.5	19.7	3.8
Wool	1022.9	112.8	43.5	15.1	4.4
WwHDR	1835.8	117.1	49.6	20	4.3
WoolX	2048.9	369.4	125.3	120.2	68.4
WwHDRX	3472.2	620.4	120.6	117.8	43.6

Table 3: Single frame rendering times for different implementations and image resolutions.

path, routed through the system memory, also exists. Here the prepared data first need to be downloaded from the GPU (OpenGL) to the system memory and then uploaded back to the GPU (OpenCL). After this, the same data transfer needs to take place to return the processed results from OpenCL back to the OpenGL display chain. We decided to implement both of these variants to compare the performance difference of both approaches. The system memory variant additionally demonstrates the use of the GPU-based BTF evaluation code from a CPU-based environment.

6 Results

6.1 Testing environment

To measure the performance of our implementation, we created a sample application, as described in Section 5. All the results were measured directly inside the application, using the built-in benchmark utility, with the exception of raw OpenCL kernel execution times, which were measured externally, using the gDEBugger [7] utility. In both cases, we measured the average time taken (out of 100 measurements) to fully render a single frame of a scene, consisting of a 3D geometry with surface covered by BTF. We used this approach mainly because it is the only performance metric available in the reference GLSL implementation.

We studied the performance of the following implementations of the BTF reconstruction algorithm:

- **CPU** - Original CPU-based implementation, provided by the authors of [4].
- **CLCPU** - Our OpenCL-based implementation, with the OpenGL-OpenCL communication routed through the system memory, simulating the use of the OpenCL evaluation kernel from within a CPU-based environment.
- **CLGL** - Our OpenCL-based implementation, with direct OpenGL-OpenCL communication.
- **RawCL** - Raw execution time of the BTF evaluation kernel of our OpenCL implementation (excluding the rasterization time and OpenCL-OpenGL communication overhead).
- **GLSL** - The reference GLSL-based implementation, obtained from the BTFBase project webpage [5].

The test scene we used consisted of a sphere model covered with a BTF sample, as shown in Figure 4. Both the light and the camera were positioned at a fixed location, which remained the same throughout all the tests. The number of individual BTF evaluations per frame therefore changed only when changing the resolution. The list of measured resolutions and the total number of individual BTF evaluations required per frame are shown in Table 2.

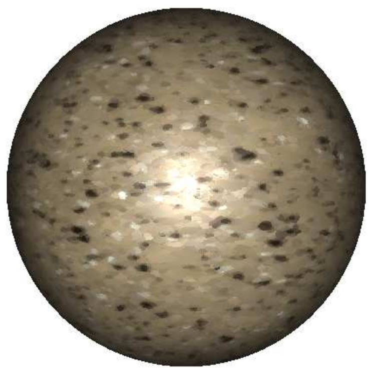


Figure 4: The test scene as rendered by our sample application, using the WalkwayHDR material.

We used two different test setups to measure the performance on different hardware. The primary test setup represented a mid-level class computer and consisted of the

Resolution [px]	Eval. time [ms]	Eval. rate [MEval/s]
800×600	19.9	24.12
1280×720	37.1	24.84
1920×1080	70.0	29.62

Table 4: Peak compressed BTF evaluation rates of our implementation for different batch sizes (resolutions).

following components: Intel Core i5 3570K @ 3,4GHZ, nVidia GeForce GTX 560 Ti, 16GB DDR3 666 MHz RAM. The secondary test setup represented a low-end computer and consisted of the following components: Intel Pentium E2160 @ 3.0GHz, nVidia GeForce 8800 GTS 512, 4GB DDR2 333 MHz RAM.

The performance was evaluated on four different material datasets - Wool (**Wool**), Impalla (**Impalla**), Walkway-HDR (**WwHDR**), FloortileHDR (**FtHDR**). On the secondary test setup, only the Wool and WalkwayHDR materials were tested, marked **WoolX** and **WwHDRX** in tables with results. The compressed datasets for these materials are a part of the publicly available GLSL-based implementation [5].

6.2 Measured performance

In Table 3, the results for all the evaluated implementations are shown, measured for the individual resolutions. We show, that highly interactive framerates can be achieved using our implementation even when working in high resolution. Further comparison with the CPU-based implementation shows an order of magnitude decrease of the time required to render a single frame. Using our second test setup, we also demonstrate, that interactive framerates can be reached even on a low-end computer hardware and that no special features available only on contemporary hardware are required by our implementation.

As visible in Table 3, comparison of rendering times for both LDR and HDR material data yields roughly the same results. This is expected, as the code for each variant differs only in the color-model conversion function used when returning from the *M* codebook. A significant drop of performance for HDR material data is only reported for the CPU-based implementation.

Comparing the performance of **CLCPU** and **CLGL** implementations on our primary test setup, we find the **CLGL** variant to be approximately 2 times faster in all cases. Highly interactive results were obtained even when using the **CLCPU** variant, which indicates that our implementation provides good performance even when supplied data from a CPU-based environment. The bandwidth of the system memory bus is of key importance in this case, as visible on the result obtained from the secondary test setup, where the performance difference between the **CLCPU** and **CLGL** implementations is more visible.

To find out the peak BTF evaluation performance of our implementation, we measured the execution time of the evaluation kernel operating on a full-screen quad fully covered by the Wool BTF material. To minimize the amount of under-the-hood caching of the GPU, the mapped BTF resolution was increased to the level, were a full BTF evaluation is required to be performed for each separate pixel. To find out the level of additional processing overhead, we performed the test using three different resolutions. As shown in Table 4, we were able to achieve peak performance of roughly 30 million compressed BTF evaluations per second. To provide a fair result, we used the same OpenCL kernel as in all the other tests, although the condition whether or not to evaluate the BTF for the given pixel could have been removed from the kernel for this specific case (resulting in a small performance increase).

Single frame rendering time [ms]				
	Material / implementation			
Mapped BTF resolution	Impalla		WwHDR	
	GLSL	CL-GL	GLSL	CL-GL
512×512	1.9	11.2	2.0	13.7
1024×1024	1.9	12.5	2.0	13.9
2048×2048	2.9	14.1	2.9	14.8
4096×4096	4.7	15.3	5.2	16.3
8192×8192	6.0	16.3	7.4	17.7

Table 5: Comparison of single frame rendering times for different mapped BTF resolutions.

6.3 Comparison of OpenCL and GLSL implementation

As separate applications are used to evaluate the OpenCL and GLSL implementations performance, we would like to point out some key differences relevant to the measured timings. First, our application executes the BTF evaluation kernel over the whole rasterized image (similar to a post-processing effect), even on the parts where no BTF evaluation is required. The decision whether or not to evaluate the BTF for a pixel can thus first be made at the beginning of the evaluation kernel. In the GLSL implementation, the decision is made directly during the rasterization stage, since shading is not performed on pixels not belonging to the object. This brings additional overhead for our application and gives the GLSL application some performance advantage.

The GLSL application can also perform the BTF evaluation directly in the fragment shader. Since it is not possible to invoke OpenCL code directly from within a shader, our application needs to store the rasterized data into an intermediate buffer, perform BTF evaluation over the data and then use another buffer to copy the computed data back to the visualization chain. This additional overhead can be seen by comparing the raw execution time of the kernel (**RawCL**) and the time required to render the whole frame (**CLGL**). Consequently this also decreases the measured performance of our application.

When comparing the performance of both applications, we have also noticed the GLSL application to have a strong dependence on the image resolution, in which the BTF material is mapped onto the object. As shown in Table 5, by increasing the BTF mapped resolution, the performance of the application decreases rapidly. This behavior can also be observed in our application, where the performance drop however is not so steep.

For these reasons, we would like to point out that our results cannot be interpreted as a direct comparison of OpenCL and GLSL, since the differences between both sample applications are significant. As the GLSL application does not provide any way to accurately measure the raw BTF evaluation performance (excluding the rasterization and display overhead), we believe that the direct performance-wise comparison would be unfair. We however decided to present the results we obtained for the two sample applications, as they show the fitness of either approach for the specific task of integrating the BTF evaluation stage into a standard OpenGL rendering pipeline.

Below we would like to comment on the results. While still maintaining interactive framerates, our sample application with OpenCL is much slower than the reference GLSL one. This is expected, as there is much additional processing required to successfully use the OpenGL-generated data within the OpenCL context and vice-versa. As a result, we find our OpenCL-based implementation not well suited for the integration as a stage into the common OpenGL rendering pipeline and recommend using the GLSL variant. We however believe our implementation is more general than the GLSL one, hopefully showing its full potential as a part of a different pipeline.

7 Conclusions

After studying the evaluation process of BTF compressed using the multi-level vector quantization method, we were able to successfully implement this method in a GPU-based environment, choosing OpenCL as a platform of our choice. Using our sample application as a test platform, we were able to achieve interactive framerates even on a low-end computer setup, thus proving the method to be well suited for GPU environment and real-time use. With our implementation, we were able to achieve a peak performance of 30 million individual BTF evaluations per second, with no preprocessing of the compressed data codebooks required. In our sample application, we also investigated the topic of OpenGL-OpenCL communication and resource sharing, finding this feature to provide significant performance benefits. Our implementation also proved to be an order of magnitude faster than a basic CPU-based variant. This holds even when feeding data to the OpenCL implementation from within a CPU context. By direct comparison, our sample application was shown to be slower than the reference GLSL-based application. We however state several points as to why a direct comparison is not appropriate. As a result of our work, we provide our implementation and all its source code to be freely used. In our codebase, the whole BTF evaluation process is covered by a single function call, which makes it very easy to integrate into an application code.

8 Acknowledgements

We would like to thank the authors of [4] for providing us with the source codes. This project has been partly supported by Czech Science Foundation under research program P202/12/2413 and Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13 (Research of Progressive Computer Graphics Methods). Finally, we would like to thank BTF database Bonn for providing their data sets.

References

[1] P. Cozzi and C. Riccio. *OpenGL Insights*. Taylor & Francis Group, 2012.

[2] K. J. Dana, B. van Ginneken, S. K. Nayar, and J. J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Trans. Graph.*, 18(1):1–34, January 1999.

[3] J. Filip and M. Haindl. Bidirectional Texture Function Modeling: A State of the Art Survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(11):1921–1940, Nov 2009.

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

[5] V. Havran, J. Filip, and K. Myszkowski. Implementation of Bidirectional Texture Function Compression based on Multi-Level Vector Quantization, 2010. Project webpage `http://dcgi.felk.cvut.cz/home/havran/btfbase/`.

[6] A. Munshi, B. Gaster, T.G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. OpenGL Series. Pearson Education, 2011.

[7] Graphic Remedy. gDEBugger - OpenGL and OpenCL Debugger, Profiler and Memory Analyzer, 2010. `http://www.gremedy.com/`.

[8] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600 –612, april 2004.

[9] G.L. Ward. LogLuv encoding for full-gamut, high-dynamic range images. *Journal of Graph. Tools*, 3(1):15–31, March 1998.

[10] D. Wolff. *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, Limited, 2011.