Vision, Modeling, and Visualization (2016)
M. Hullin, M. Stamminger, and T. Weinkauf (Eds.)

Variable Length Coding for GPU-Based Direct Volume Rendering

S. Guthe[1] and M. Goesele[1]

[1]Graphics, Capture and Massively Parallel Computing, TU Darmstadt, Germany

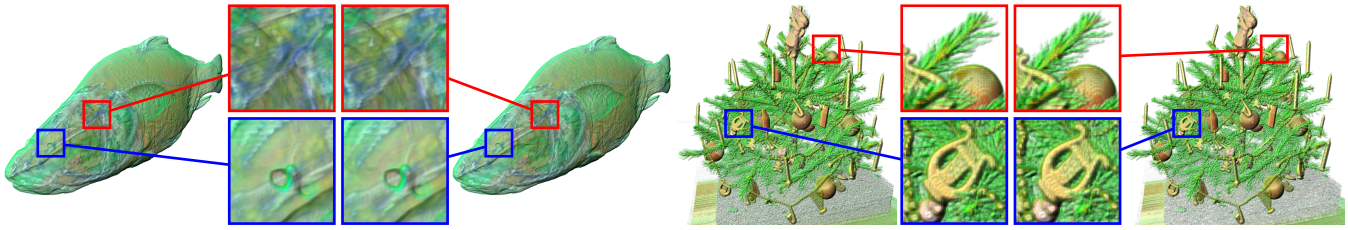


Figure 1: Carp dataset [Röt06] with lossless compression at 5.5 bit per voxel (bpv) vs. lossy at 0.5 bpv and christmas tree dataset [KTM*02] with lossless at 4.1 bpv vs. lossy at 0.5 bpv. Both datasets require 12 bpv in their uncompressed form and are rendering using on-the-fly calculation of gradients for lighting. Visually, there are some minor differences to be seen in the red regions while the blue regions are indistinguishable.

Abstract

The sheer size of volume data sampled in a regular grid requires efficient lossless and lossy compression algorithms that allow for on-the-fly decompression during rendering. While all hardware assisted approaches are based on fixed bit rate block truncation coding, they suffer from degradation in regions of high variation while wasting space in homogeneous areas. On the other hand, vector quantization approaches using texture hardware achieve an even distribution of error in the entire volume at the cost of storing overlapping blocks or bricks. However, these approaches suffer from severe blocking artifacts that need to be smoothed over during rendering. In contrast to existing approaches, we propose to build a lossy compression scheme on top of a state-of-the-art lossless compression approach built on non-overlapping bricks by combining it with straight forward vector quantization. Due to efficient caching and load balancing, the rendering performance of our approach improves with the compression rate and can achieve interactive to real-time frame rates even at full HD resolution.

Categories and Subject Descriptors (according to ACM CCS): E.4 [Coding and Information Theory]: Data Compaction and Compression— I.3.1 [Computer Graphics]: Picture and Image Generation—Graphics processors I.3.3 [Computer Graphics]: Picture and Image Generation—Viewing algorithms

1. Introduction

While volume data sampled on a regular grid is the most easy to handle data structure for representing a three dimensional function during rendering, it is also the most space consuming. Therefore, a variety of different lossless and lossy compression approaches have been presented in the past, mostly based on vector quantization [NH92, NH93, SW03], wavelets [IP98, KS99, Rod99, NS01, GS04] or other transformations [FM07, FM12, Lin14, GG16]. While fixed bit rate approaches are usually the first choice for on-the-fly decompression, they also offer the overall highest error for any given compression ratio. On the other hand, variable bit rate approaches offer overall higher quality or even lossless compression for the same number of bits per voxel. However, these approaches require additional data structures to hold pointers into the compressed data if random access during rendering is required. Since this additional indirection can also be used to implement multiple references to the same compressed data, i.e. instancing, it can efficiently store volume data that has been reduced using vector quantization. We thus propose an on-the-fly decompression algorithm based on vector quantization and lossless compression of the resulting codebook, i.e. the individual volume bricks.

In order to evaluate our approach, we based our volume renderer

on the CUDA SDK direct volume rendering example. While the performance of our approach was measured on a NVIDIA GeForce GTX 1080, it works on all Pascal, Maxwell and Kepler based architectures as we require only Kepler specific extensions for inter-thread communication. Note that the increased shared memory found in the Pascal and Maxwell architectures drastically increases the performance of our caching and therefore rendering algorithm. The simpler scheduling and memory interface design of the Pascal architecture on the other hand makes an efficient caching and load balancing approach even more important. Our contributions are:

- A lossy extension to the real-time warp-based decompression and rendering approach of Guthe and Goesele [GG16],
- an improved codebook generation for vector quantization of volume data,
- an on-the-fly gradient calculation for shading and non-photorealistic volume rendering, and
- a detailed analysis of the impact of individual components of the decompression and rendering algorithm.

2. Related Work

In contrast to remote rendering approaches [FSME14, FSE15] that render the uncompressed volume data and compress the resulting images, we provide an on-the-fly decompression algorithm for volume data, the related work can be split into the following four areas:

Hardware Based Compression: All purely hardware based texture compression approaches in existence today are based on some form of block truncation coding (BTC [DM79]). Most of these approaches however are designed with 2D textures in mind, e.g. the ETC2 compression [SP07] is not capable of directly encoding three dimensional data and is limited to at most 8 bit per channel. However, the recently proposed and implemented ASTC compression [NLP*12], which is based on PACKMAN [SAM04], supports three dimensional textures and more than 8 bit per channel natively.

Volume Data Compression: Beyond hardware based texture compression, there exists a large number of specialized volume compression algorithms for the GPU. Please refer to the surveys of Rodríguez at al. [RGG*13] and Beyer et al. [BHP14] for a more extensive review of existing approaches. The approach of Lindstrom and Isenburg [LI06] allows for lossless compression of floating point numbers with a given accuracy. However, the coding performance of this approach is very limited. Recently Lindstrom proposed an extension for fixed rate coding using the same basic algorithm [Lin14]. Similar to BTC, the bit stream representing the compressed data is truncated once the target bit rate is exceeded. Even though the algorithm was designed to be implemented on the GPU, the actual decompression performance is still too low for real-time rendering. The lossless compression of Guthe and Goesele [GG16], on the other hand, implements extensive caching of decompressed data and can be deployed into any existing rendering algorithm, achieving at least interactive frame rates for lossless compression. However, this approach only supports lossless compression.

Vector Quantization: Ning and Hesselink [NH92, NH93] proposed vector quantization (VQ) in the context of volume compression and volume rendering. Rendering approaches employing VQ are usually 3D texture based where each brick consists of n^3 voxel. In order to use the hardware texture interpolation, these bricks overlap by 1 voxel in every direction. This increases the amount of data that needs to be stored and makes traditional VQ especially prone to blocking artifacts which then have to be handled during rendering. For this, Marton et al. [MGDG14] proposed an approach that implements deblocking by using a temporary buffer prior to applying the transfer function and shading. However, the main performance bottleneck for VQ is finding the best codebook when compressing data. In order to maximize the quantization performance, we apply the optimizations of Kanungo et al. [KMN*02] combined with the insights of Hamerly and Drake [HD15] to increase the performance of distance calculations in the k-means clustering of codewords.

Rendering: Since we decompress the data during rendering and therefore allow random access and high quality filtering, we can support advanced volume rendering algorithms like interactive lighting and gradient magnitude modulation [MGS02], multi-dimensional transfer function [KKH02] or pre-integrated volume rendering [EKE01]. Mensmann et al. [MRH10] present a comprehensive overview on GPU raytracing that also includes on-the-fly gradient calculation. In contrast to their approach, we only require intra warp communication and do not require special border handling.

3. Compressed Data Representation

Our compressed data representation is equivalent to the data structures described in [GG16]. During compression the volume is split into bricks of 4^3 voxel. In a first step, a given input volume of $N \times M \times L$ voxel is split into a list of unique 4^3 voxel bricks and a volume of $\lceil \frac{N}{4} \rceil \times \lceil \frac{M}{4} \rceil \times \lceil \frac{L}{4} \rceil$ containing references to these bricks. Since these references are offsets into the array of bricks, we call the smaller volume the *offset volume*. In terms of vector quantization the list of bricks is equivalent to the *codebook*. Thus, the first observation that can be made is that each of these can be compressed individually.

3.1. Codebook Compression

Guthe and Goesele [GG16] proposed a codebook compression algorithm that is based on a selection of data transformations as well as a parallel compression algorithm called *recursive bottom up complete* (RBUC) [MA05]. The RBUC compression is a hierarchical compression scheme based on Elias codes [Eli75], that efficiently encodes a group of n small values. In case of Guthe and Goesele [GG16], the codeword contains two levels of 8 values each. With this, a compressed brick of 4^3 voxel is represented by the minimum and maximum values stored in uncompressed form and, if the minimum and maximum values are different, a single hierarchical variable length codeword consisting of:

- One byte containing the value c_2 (number of bits required to represent all values of c_1) in the lower 6 bit and the transformation id in the upper 2 bit.
- One set of c_2 bytes containing the values $[c_1(0), \ldots, c_1(7)]$ (number of bits required to represent all values of c_0)each using c_2 bits.

- Eight groups of $c_1(i)$ bytes containing the values $[c_0(8i), \ldots, c_0(8i+7)]$ each using $c_1(i)$ bits.

Each of the items in the list above is automatically aligned to byte boundaries making decompression as efficient as possible. The size of the codeword can be anywhere between 1 and $1+4+96$ byte for 12 bit input data. In order to avoid too much data expansion (the original size was 96 byte in this case), a value 255 is stored in c_2 to denote that the following bytes contain the original voxel data.

As an alternative, we also support an uncompressed codebook in our rendering approach as this allows to do a trade-off between compression rate, compression quality and rendering performance, i.e. increasing rendering performance at the same quality with lower compression rate or same compression rate with lower quality (see Section 6 for detailed performance evaluation). In this case 4^3 voxels of 12 bits per voxel are stored using bit packing, leading to 96 bytes.

3.2. Offset Volume

Each codebook entry is uniquely identified by its index ($0 \ldots k-1$ for k codewords) or its starting address which is equivalent to an offset into the codebook. However, only the starting address allows for random access to the codeword itself if the size of the codewords varies. Thus, for representing the offset volume, we first have to distinguish between two scenarios.

In case of an uncompressed codebook, the starting address of a codebook entry is always a multiple of the codeword index and the fixed brick size. We therefore simply need to store the index of the brick in the offset volume. Again, if the highest index can be represented using n bits, we only store n bits per offset volume entry and use bit packing again.

In case of a compressed codebook, we have two options. We either directly store the starting address of the compressed bricks in the offset volume (*direct mapping*) or we store the index and need an additional data structure for converting indices to starting addresses (*indirect mapping*). If we assume that we have a total of N entries in the offset volume and M codewords, where each index requires a bit and each starting address b bit, we can calculate the size of both representations as follows. The direct mapping requires $\left\lceil \frac{N \cdot b}{8} \right\rceil$ byte whereas the indirect mapping requires $\lceil \frac{N \cdot a}{8} \rceil + \left\lceil \frac{M \cdot b}{8} \right\rceil$ byte. Based on this, we pick the representation that yields the fewer number of bytes. Note that there is a slight performance penalty for the additional lookup when decompressing the brick data but on the other hand, the memory bandwidth requirement for every cache hit is lower. As can be seen in Section 6, these effects are unnoticeable in the final rendering performance.

4. Data Reduction

Prior to compressing the offset volume and the codebook, we can apply vector quantization to reduce the number of entries in the codebook. Our implementation of vector quantization uses an optimized version of k-means clustering as described by Kanungo et al. [KMN*02]. However, since the dimensionality of our data, i.e. the number of voxel per codeword, is 64, the more involved optimizations, i.e. *kd*-trees, cannot be used. However, Hamerly and Drake [HD15] note that a lot of distance calculations are unnecessary and can be avoided using the triangle inequality and storing the distance of each codeword and cluster center to the origin. In addition, since we are only interested in the order of distances rather than the actual value, we can use squared distances instead. When calculating the distance between to vectors $\vec{a}$ and $\vec{b}$ given a maximum squared distance d^2_{max}, we first check for $\left(\|\vec{a}\| - \|\vec{b}\|\right)^2 \geq d^2_{max}$. If this condition is met, the distance d between $\vec{a}$ and $\vec{b}$ cannot be below d_{max} and the actual distance calculation can be skipped. If the condition is not met, the squared distance is calculated as $d^2 = \|\vec{a}\|^2 + \|\vec{b}\|^2 - 2\vec{a} \cdot \vec{b}$ (see Algorithm 1 for the complete function).

Algorithm 1: Function for calculating the squared distance given a maximum distance to check against.

```
Function DistanceS(d²_max : max. distance², a, b : codeword)
    if (a.d − b.d)(a.d − b.d) ≥ d²_max then
        return d²_max                                   // triangle inequality
    end
    return a.d² + b.d² − 2a · b
end
```

As noted by Kanungo et al. [KMN*02], the *furthest first* initialization usually leads to both the best compression quality and least number of iterations. Starting with the codeword closest to the origin, we continue to create cluster centers at codewords that are furthest away from all existing centers. For this, we maintain a priority queue that contains the distance and the number of cluster centers taken into account so far. If the entry with the largest distance took all existing centers into account, we add it as a new cluster center. Otherwise, we update the distance put it back into the queue (see Algorithm 2). Note that for very large data sets and very conservative data reduction, we do not use the priority queue but simply update the closest cluster center for each block in parallel.

Once all cluster centers have been found, each codeword gets assigned to the closest cluster center (see Algorithm 4). In each iteration of the k-means clustering algorithm, we in turn update the cluster centers and then assign each codeword to the closest center. While updating the closest cluster center, we can encounter three different scenarios. If the distance between a codeword and the closest cluster center is 0, we do not need to check any other cluster center. If a codeword was closest to a cluster center that did not change, we know that it is further away from all other unchanged cluster centers. Thus, we only have to check against the changed cluster centers. If a codeword was closest to a cluster center that did change, we calculate the new distance and compare it against the old one. If distance decreased (which is more likely), we again only need to check against the list of cluster centers that changed. If the distance increased, we need to check against all cluster centers (see Algorithm 3). In other words, we only need to calculate the distance to all cluster center if a codeword is further away from its center after the previous cluster center update.

Once all codewords have been assigned to their corresponding cluster, we update the center and sort them into the changed and unchanged list until the list of changed cluster center is empty. For calculating the cluster center, each codeword is weighted with

Algorithm 2: Initialization of cluster center.

```
Data : codebook CB
Data : number of cluster k
Data : priority queue Q
Data : list of cluster center C
d_min ← ∞
// find codeword closest to origin
foreach cw ∈ CB do
    cw.d ← √(cw · cw)                    // distance to origin
    if cw.d < d_min then
        d_min ← cw.d
        closest ← cw
    end
end
// closest becomes first cluster center
C.append(closest)
closest.center ← C.last
// initialize priority queue Q
foreach cw ∈ CB \ closest do
    cw.p ← DistanceS(∞, cw, closest)
    cw.center ← closest
    cw.eval ← 1
    Q.insert(t)
end
while C.size < k do
    cw ← Q.top
    Q.pop()
    // loop over unaccounted cluster
    while cw.eval < k do
        d ← DistanceS(cw.p, cw, C[cw.eval])
        if d < cw.p then
            cw.p ← d
            cw.center ← C[cw.eval]
        end
        cw.eval ← cw.eval + 1
    end
    if cw.p ≥ Q.top.p then
        // cw is still furthest
        C.append(cw)
        cw.center ← C.last
    else Q.insert(cw)
end
```

Algorithm 3: Function to checking a codeword against a list of cluster center.

```
Function FindCenter(c : codeword, CL : list of cluster center)
    cw.p ← d
    center_n ← cw.center
    foreach c ∈ CL \ cw.center do
        d_n ← DistanceS(d, cw, c)
        if d_n < d then
            d ← d_n
            center_n ← c
        end
    end
    // if closest cluster center changed, remove from old
       center and attach to new one
    if center_n ≠ cw.center then
        cw.center.remove(cw)
        center_n.add(cw)
        cw.center ← center_n
        cw.p ← d_n
    end
end
```

the number of appearances in the volume. Even though we see a speedup of over two orders of magnitude, less aggressive quantizations are still very costly. A 1GB data set takes about 12 hours for a 2:1 reduction. The actual compression of the reduced data only takes a couple of seconds.

Algorithm 4: k-means clustering algorithm.

```
Data : codebook CB
Data : list of cluster center C
Data : number of cluster k
Data : list of update cluster center U
// attach each cw to the closest cluster
foreach cw ∈ CB do
    while cw.eval < k do
        d ← DistanceS(cw.p, cw, C[cw.eval])
        if d < cw.p then
            cw.p ← d
            cw.center ← C[cw.eval]
        end
        cw.eval ← cw.eval + 1
    end
    c ← cw.center
    c.add(cw)
end
U ← C
while U ≠ ∅ do
    foreach cw ∈ CB do
        if (cw.center ∈ U) then d ← DistanceS(∞, cw, cw.center)
        else d ← cw.p
        if d > 0 then
            if d > cw.p then
                FindCenter(c, C)
            else
                FindCenter(c, U)
            end
        else
            cw.p ← d
        end
    end
    clear(U)
    foreach c ∈ C do
        c.update()
        if c.changed then
            U.add(c)
        end
    end
end
```

5. Rendering

Since we use the same codebook compression as Guthe and Goesele [GG16], the decompression and caching approach stays the same. Note that both the decompression and caching operate on a per-warp basis and each warp operates on n samples per ray of a $m \times l$ group of rays. If we restrict n, m and l to be at least two, we can use warp shuffle operations to get the sample data from neighboring sample location. We can also do the same set of operations to get the corresponding sample positions. Using this data, we can derive the local gradient of the scalar volume data for shading as follows. We define a_0 to be the current sample value at position $\vec{p}_0$ and a_1, a_2 & a_3 at position $\vec{p}_1$, $\vec{p}_2$ & $\vec{p}_3$ to be the sample values returned by the warp shuffle operations. The derivatives are defined as $\vec{v}$ with $v_i = \frac{a_i - a_0}{\|\vec{p}_i - \vec{p}_0\|}$ along the direction $\vec{d}_i = \frac{\vec{p}_i - \vec{p}_0}{\|\vec{p}_i - \vec{p}_0\|}$ for $i = [1..3]$. In contrast to Mensmann et al. [MRH10], we cannot rely on the directions to be close to orthogonal. Instead we define the matrix D where each row contains one vector $\vec{d}_i$ and solve the following equation for $\vec{x}$ by matrix inversion.

$$\begin{aligned} D\vec{x} &= \vec{v} \\ \vec{x} &= D^{-1}\vec{v} \end{aligned}$$

Just like gradients calculated using central differences, the gradient $\vec{x}$ is defined in voxel space. Note that the derivatives are calcu-

data set	dimensions	bpv	voxel
Carp	$256 \times 256 \times 512$	12	33,554,432
Bunny	$512 \times 512 \times 361$	12	94,633,984
C. Present	$492 \times 492 \times 442$	12	106,992,288
C. Tree	$512 \times 499 \times 512$	12	130,809,856
Porsche	$559 \times 1023 \times 347$	8	198,434,379
Stag Beetle	$832 \times 832 \times 494$	12	341,958,656
Pawpawsaurus	$958 \times 646 \times 1088$	16	673,328,384
Flower	$1024 \times 1024 \times 1024$	8	1,073,741,824

Table 1: Data set dimensions and raw data sizes.

lated by swapping data between adjacent rays which means that two neighboring pixel will always share one common derivative. The overhead of this on-the-fly estimation is a lot lower than calculating additional samples for directly computing the derivatives along the axis of the coordinate system. As mentioned before, the whole algorithm rendering system is split into independent sections via template objects. The warp based tracing algorithm itself is unaware of how the volume is sampled and what kind of shading takes place. Thus, we can deploy any shading algorithm available today. For our evaluation, we implemented a simple post-classification renderer that allows for optional shading based on gradients and gradient magnitude modulation of the opacity, i.e. non-photorealistic rendering.

6. Results

In order to evaluate the whole algorithm, we need to analyze all parts of the decompression and rendering pipeline using real-world data. All tests were run using an Intel(R) Core(TM) i7-3930K CPU @3.20GHz, 64GB system memory, an NVIDIA GeForce GTX 1080 with CUDA 8.0, driver version 368.39, and an extension of the volume rendering example found in the CUDA SDK. We evaluated the rendering time in *ms* for every data set at two different resolutions using lossy compression and 10 different quantization settings. The compression is either with or without compressed codebook. The rendering for the uncompressed codebook has two different versions, one with per-warp caching enabled and one with direct access to individual voxel.

6.1. Data Sets

We tested a variety of data sets, including 8 bit per voxel (bpv) and 12 bpv scalar data, ranging in size from 48 MB up to about 489 MB of raw data, see Figure 2 and Table 1. Note that the volume data sets do not necessarily contain cubic voxel but usually have an actual voxel extend as defined by the data set, e.g. $1.0 \times 0.5 \times 1.28$ for the Carp data set. We use the PSNR for comparing the quality of both data sets and images in a consistent way (see [HTG08] for a discussion on the applicability of this measure).

6.2. Compression

For evaluating the compression, we compare against the lossy ASTC texture compression [NLP*12], the lossy fixed rate floating point compression of Lindstrom [Lin14], as well as against the lossless compression of Guthe and Goesele [GG16] both with compressed and uncompressed codebook. For the ASTC compression we used block sizes of $3 \times 3 \times 3$, $4 \times 4 \times 4$, $5 \times 5 \times 5$ and $6 \times 6 \times 6$, resulting in fixed bit rates of 4.74, 2, 1.024 and 0.59259 bit per voxel (bpv). For the fixed rate floating point compression (ZFP), we set a target bit rate of 4, 2, 1, 0.5, 0.25 and 0.125 bpv. Finally, for our compression, we reduce the number of unique bricks to $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ... $\frac{1}{512}$ and $\frac{1}{1024}$, producing bit rates in the range of 6 to 0.8 bpv. The compression results for all of these settings can be seen in Figure 3.

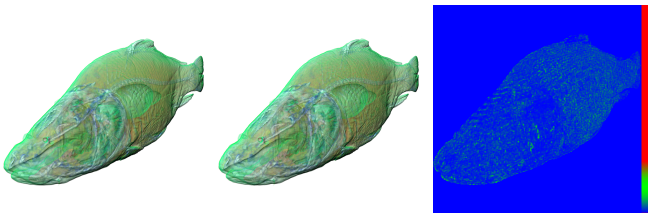
Overall, the compression results show that our approach with the compressed codebook is able to substantially outperform all other approaches. Even with the uncompressed codebook, the vector quantization performs better than ASTC in all cases except for the 8 bit Porsche data set above 0.5 bpv and better than ZFP except for 4 bpv of the Carp, Bunny and Porsche data set. For the stagbeetle data set, even the lossless compression without compressed codebook leads to fewer bit per voxel as ASTC is able to generate.

6.3. Rendering

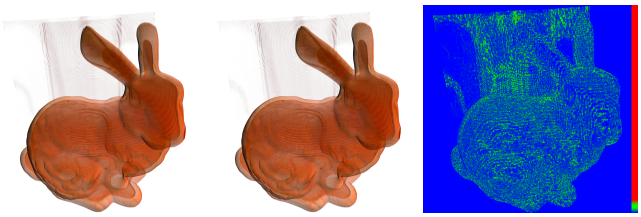
All performance measurements were done using warp-based direct volume rendering with a sample distance of a quarter of the minimal voxel extend and a transfer function that maps scalar values to opacity and color (see Röttger et al. [RGW*03] for a discussion of minimal sampling distances in this context). The threshold for early ray termination is an opacity of 0.995, i.e. only lsb errors. The transfer function used for the performance testing can be seen in Figure 2.

The renderer is designed using template classes and explicit instantiation to be able to switch to any different combination of compression, shading and warp layout at run-time. The compile-time for all variants is about 15 minutes on our test machine.

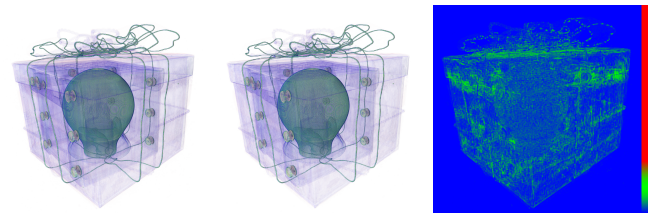
As seen in Figure 4 and Table 2, the rendering performance is at least interactive for all data sets at a resolution of 1920×1080 and real-time at a resolution 512×512 for smaller data sets and interactive at 5125×512 for larger data sets. In case of uncompressed codebooks, the direct access is faster than warp-based caching as long as the access to memory is regular enough. Once the number of codewords is small enough to scatter reading throughout the entire codebook, the caching becomes more efficient. Also the compressed codebook becomes more efficient as soon as the cache produces additional hits to cached bricks. However, for some data sets, the performance goes down initially when increasing the compression ratio. This is due to the percentage of bricks encoded with the expensive gradient predictor. There is no real explanation as to why this percentage goes up except for the observation that bricks encoded with simpler predictors are merged before bricks encoded with the gradient predictor. The maximum sampling performance for the Stagbeetle data set is 15Gsamples/s with codebook compression and 17Gsamples/s without codebook compression at a resolution of 1920×1080. Each sample is based on tri-linear interpolation of 8 voxel, leading to a bandwidth of 180GB/s or 204GB/s to the compressed voxel data. For comparison, the raw memory bandwidth of the GTX 1080 without caches is 320GB/s but the sampling performance is 61Gsample/s or 732GB/s.



(a) Carp [Röt06]: 5.48bpv vs. 0.51bpv (PNSR: 37.4 dB)

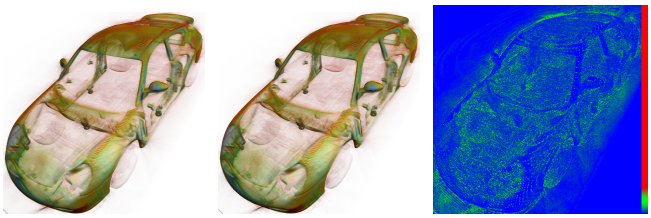


(b) Bunny [Röt06]: 5.66bpv vs. 0.50bpv (PSNR: 42.4 dB)

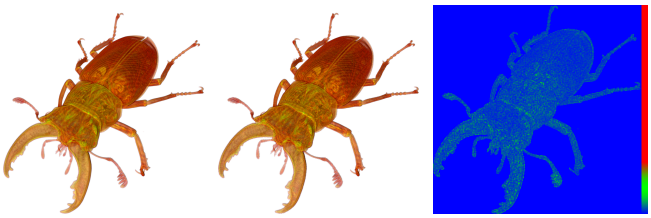


(c) Christmas Present [Hei06]: 1.75bpv vs. 0.28bpv (PSNR: 31.8 dB)

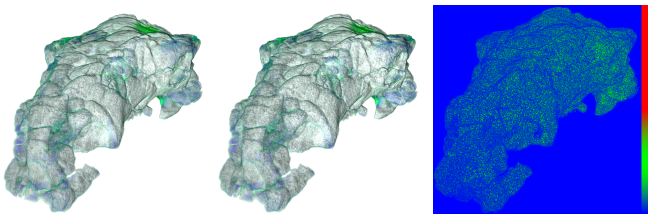
(d) Christmas Tree [KTM*02]: 4.11bpv vs. 0.47bpv (PSNR: 25.7 dB)



(e) Porsche [Röt06]: 1.48bpv vs. 0.32bpv (PSNR: 37.0 dB)



(f) Stag Beetle [GGK05]: 0.45bpv vs. 0.14bpv (PSNR: 35.9 dB)



(g) Pawpawsaurus [Dig16]: 11.12bpv vs. 0.30bpv (PSNR: 25.6 dB)

(h) Flower: 3.39bpv vs. 0.48bpv (PSNR: 43.52 dB)

Figure 2: Data sets used for evaluation rendered with lighting and with (b, c and e) or without (a, d, f, g and h) gradient magnitude modulation. All gradients were constructed on-the-fly during rendering. From left to right: lossless compression, 32:1 reduction (256:1 for Pawpawsuarus) and false color difference image. PSNR of rendered images measured at a resolution of 1024×1024.

In comparison to un-shaded rendering, calculating the gradients on-the-fly and doing the lighting calculation increases the rendering time somewhere between 5% to 15% over all performance measures. For rendering using uncompressed textures, the performance hit is substantially larger around 40% on average. However, using additional samples to calculate gradients causes an overhead of almost 300% for compressed and about 100% for uncompressed textures since every sample requires four lookups into the volume data.

7. Conclusion & Future Work

We have shown an efficient lossy compression that is able to outperform all current block based compression approaches such as ASTC and ZFP. Our codebook generation is able to handle even the largest data sets available in a feasible amount of time. We also proposed a rendering approach that calculated gradients on-the-fly without requiring additional samples of the volume which can be used for shading and non-photorealistic rendering. Finally, we did an in-detail analysis of the performance of our entire decompression, caching and rendering pipeline.

In the future, we want to look into possible multi-resolution extensions to our rendering approach. While the extension of the compression to multi-resolution is straight forward, sampling between different resolutions is not. In addition, the on-the-fly gradient calculation causes issues with the adaptive sampling steps commonly used in multi-resolution approaches such as Guthe and Strasser [GS04].

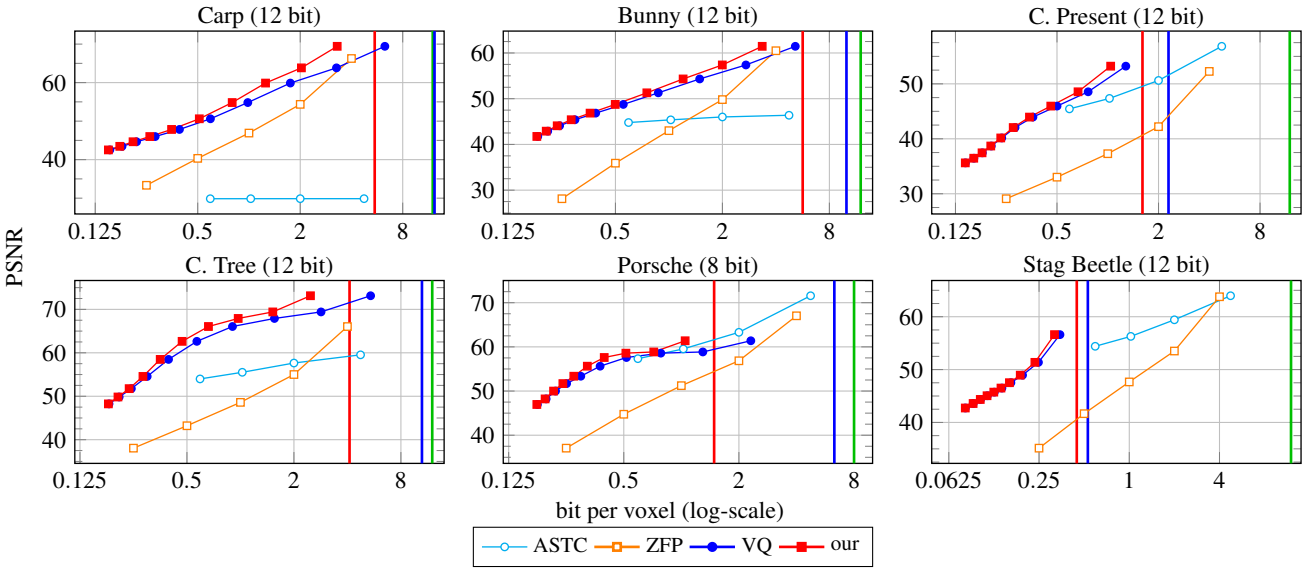


Figure 3: Lossy compression rates vs. PSNR using different compression approaches (ASTC = ASTC texture compression [NLP*12], ZFP = fixed rate floating point compression [Lin14], VQ = vector quantization only, our = vector quantization and block compression [GG16]). The red vertical line shows the lossless compression ratio with block compression. The blue line shows the ratio for VQ only. The green line shows the original number of bits per voxel.

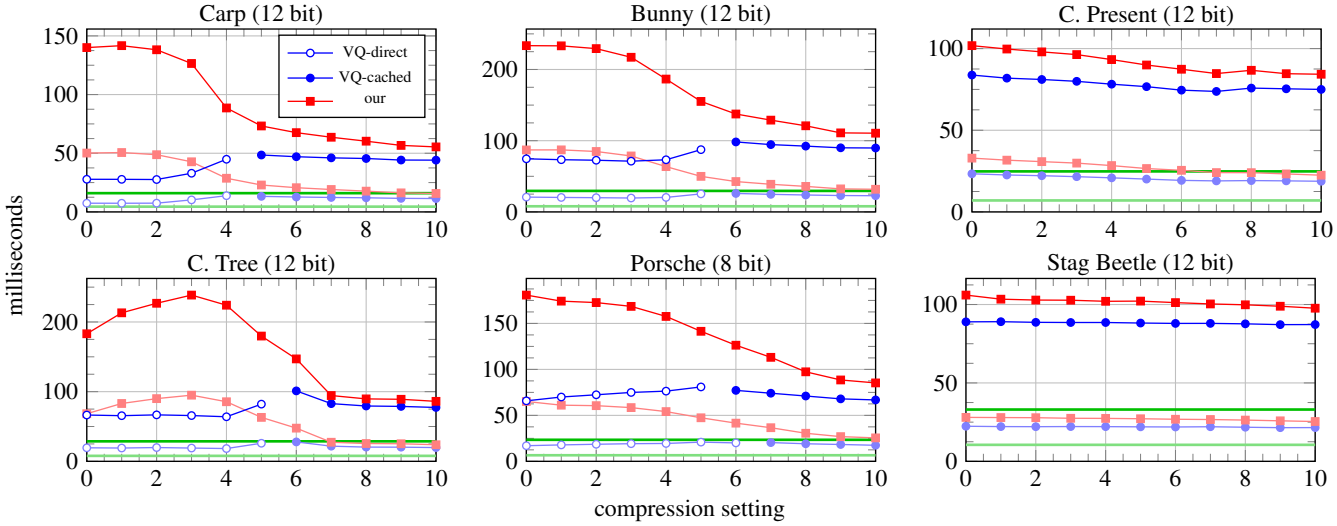


Figure 4: Quantization setting vs. rendering time @1920 × 1080 and @512 × 512 (light colors), including frame rate for lossless compression (compression setting = 0) and baseline (green) for uncompressed texture. For vector quantization without compression only the faster rendering time (caching enabled/disabled) is displayed. All times were measured using the same rendering settings as seen in Figure 2.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and suggestions. We acknowledge the Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab at University of Zurich (UZH) for the acquisition of the μCT flower data set.

References

[BHP14] BEYER J., HADWIGER M., PFISTER H.: A Survey of GPU-Based Large-Scale Volume Visualization. In *Procceedings of EuroVis)* (2014). 2

[Dig16] DIGIMORPH: Digimorph.org. http://digimorph.org/, 2016. 6

[DM79] DELP E., MITCHELL O.: Image Compression using Block Truncation Coding. *IEEE Transactions on Communications 27*, 9

data set	mode	PSNR	bpv	ms[1]	ms[2]
Pawpawsaurus	raw	Inf.	16.000	16	39
	VQ-d	Inf.	16.460	22	74
	our		11.117	235	596
	VQ-c	49.25 dB	0.314	56	185
	our		0.304	152	424
Flower	raw	Inf.	8.000	21	56
	VQ-d	Inf.	8.374	30	99
	our		3.387	205	556
	VQ-c	50.74 dB	0.547	80	266
	our		0.476	167	477

Table 2: Performance ([1]500×500, [2]1920×1080) of large data sets for lossless and lossy compression (256:1 for Pawpawsaurus and 32:1 for Flower) using the rendering settings from Figure 2 (raw = 3D texture, VQ-d = direct access, VQ-c = cached access).

(1979). 2

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2001). 2

[Eli75] ELIAS P.: Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory 21*, 2 (1975). 2

[FM07] FOUT N., M. K.-L.: Transform Coding for Hardware-Accelerated Volume Rendering. *IEEE Transactions on Visualization and Ccomputer Graphics 13*, 6 (2007). 1

[FM12] FOUT N., MA K. L.: An Adaptive Prediction-Based Approach to Lossless Compression of Floating-Point Volume Data. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012). 1

[FSE15] FREY S., SADLO F., ERTL T.: Balanced Sampling and Compression for Remote Visualization . In *Proceedings of SIGGRAPH Asia Symposium on Visualization in High Performance Computing* (2015). 2

[FSME14] FREY S., SADLO F., MA K.-L., ERTL T.: Interactive Progressive Visualization with Space-Time Error Control . *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014). 2

[GG16] GUTHE S., GOESELE M.: GPU-Based Lossless Volume Data Compression. In *Proceedings of 3DTV Conference* (2016). 1, 2, 4, 5, 7

[GGK05] GRÖLLER E., GLAESER G., KASTNER J.: Stag Beetle. http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/, 2005. 6

[GS04] GUTHE S., STRASSER W.: Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics 28*, 1 (2004). 1, 6

[HD15] HAMERLY G., DRAKE J.: Accelerating Lloyd's Algorithm for k-Means Clustering. In *Partitional Clustering Algorithms*. 2015. 2, 3

[Hei06] HEINZL C.: Christmas Present. http://www.cg.tuwien.ac.at/research/publications/2006/dataset-present/, 2006. 6

[HTG08] HUYNH-THU Q., GHANBARI M.: Scope of Validity of PSNR in Image/Video Quality Assessment. *Electronics Letters 44*, 13 (2008). 5

[IP98] IHM I., PARK S.: Wavelet-Based 3D Compression Scheme for Very Large Volume Data. In *Proceedings of Graphics Interface* (1998). 1

[KKH02] KNISS J., KINDLMANN G., HANSEN C.: Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (2002). 2

[KMN*02] KANUNGO T., MOUNT D., NETANYAHU N., PIATKO C., SILVERMAN R., WU A.: An Efficient k-Means Clustering Algorithm: Analysis and Implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 7 (2002). 2, 3

[KS99] KIM T., SHIN Y.: An Efficient Wavelet-Based Compression Method for Volume Rendering. In *Proceedings of Pacific Graphics* (1999). 1

[KTM*02] KANITSAR A., THEUSSL T., MROZ L., SRÁMEK M., BARTROLÍ A., CSÉBFALVI B., HLADUVKA J., FLEISCHMANN D., KNAPP M., WEGENKITTL R., FELKEL P., RÖTTGER S., GUTHE S., PURGATHOFER W., GRÖLLER E.: Christmas Tree Case Study: Computed Tomography As a Tool for Mastering Complex Real World Objects with Applications in Computer Graphics. In *Proceedings of IEEE Visualization* (2002). 1, 6

[LI06] LINDSTROM P., ISENBURG M.: Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006). 2

[Lin14] LINDSTROM P.: Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014). 1, 2, 5, 7

[MA05] MOFFAT A., ANH V.: Binary Codes for Non-Uniform Sources. In *Proceedings of Data Compression Conference* (2005). 2

[MGDG14] MARTON F., GUITIÁN J., DÍAZ J., GOBBETTI E.: Real-time Deblocked GPU Rendering of Compressed Volumes. In *Proceedings of Vision, Modeling & Visualization* (2014). 2

[MGS02] MEISSNER M., GUTHE S., STRASSER W.: Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In *on PC Graphics Accelerators. Graphics Interface* (2002). 2

[MRH10] MENSMANN J., ROPINSKI T., HINRICHS K.: An Advanced Volume Raycasting Technique using GPU Stream Processing. In *5th International Conference on Computer Graphics Theory and Applications* (2010). 2, 4

[NH92] NING P., HESSELINK L.: Vector Quantization for Volume Rendering. In *Proceedings of the ACM Workshop on Volume Visualization* (1992). 1, 2

[NH93] NING P., HESSELINK L.: Fast Volume Rendering of Compressed Data. In *Proceeding of IEEE Visualization* (1993). 1, 2

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive Scalable Texture Compression. In *Proceedings of ACM SIGGRAPH/Eurographics High Performance Graphics* (2012). 2, 5, 7

[NS01] NGUYEN K., SAUPE D.: Rapid High Quality Compression of Volume Data for Visualization. In *Computer Graphics Forum* (2001), vol. 20. 1

[RGG*13] RODRÍGUEZ M., GOBBETTI E., GUITIÁN J., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: A Survey of Compressed GPU-Based Direct Volume Rendering. In *Proceedings of Eurographics* (2013). 2

[RGW*03] ROETTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart Hardware-Accelerated Volume Rendering. In *VisSym* (2003), vol. 3. 5

[Rod99] RODLER F.: Wavelet Based 3D Compression with Fast Random Access for Very Large Volume Data. In *Proceedings of Pacific Graphics* (1999). 1

[Röt06] RÖTTGER S.: The Volume Library. http://lgdv.cs.fau.de/External/vollib/, 2006. 1, 6

[SAM04] STRÖM J., AKENINE-MÖLLER T.: PACKMAN: Texture Compression for Mobile Phones. In *ACM SIGGRAPH 2004 Sketches* (2004). 2

[SP07] STRÖM J., PETTERSSON M.: ETC 2: Texture Compression using Invalid Combinations. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2007), vol. 7. 2

[SW03] SCHNEIDER J., WESTERMANN R.: Compression Domain Volume Rendering. In *Proceeding of IEEE Visualization* (2003). 1