Compressing Color Data for Voxelized Surface Geometry

Dan Dolonius[*] Erik Sintorn[†] Viktor Kämpe[‡] and Ulf Assarsson[§]
Chalmers University of Technology

(a) (b) (c) (d)

Figure 1: *The* EPICCITADEL *scene, with precomputed illumination, voxelized at resolution* 32768^3. *a) Reference, 24-bit colors, 2.4GB . b) Our BC7 compression, 800MB (30%), MS-SSIM: 0.99. c) Our variable bitrate block encoding, 371MB (15.3%), MS-SSIM: 0.98. d) [Dado et al. 2016]] 456MB (18.8%), MS-SSIM: 0.92.*

Abstract

We explore the problem of decoupling color information from geometry in large scenes of voxelized surfaces and of compressing the array of colors without introducing disturbing artifacts. First, we present a novel method for connecting each node in a sparse voxel DAG to its corresponding colors in a separate 1D array of colors, with very little additional information stored to the DAG. Then, we show that by mapping the 1D array of colors onto a 2D image using a space-filling curve, we can achieve high compression rates and good quality using conventional, modern, hardware-accelerated texture compression formats such as ASTC or BC7. We additionally explore whether this method can be used to compress voxel colors for off-line storage and network transmission using conventional off-line compression formats such as JPG and JPG2K. For real-time decompression, we suggest a novel variable bitrate block encoding that consistently outperforms previous work, often achieving two times the compression at equal quality.

Keywords: voxelization, compression

Concepts: •**Computing methodologies** → *Texturing; Graphics file formats; Image compression;*

[*]e-mail:dolonius@chalmers.se
[†]e-mail:erik.sintorn@chalmers.se
[‡]e-mail:kampe@chalmers.se
[§]e-mail:uffe@chalmers.se

I3D '17,, February 25 - 27, 2017, San Francisco, CA, USA
ISBN: 978-1-4503-4886-7/17/03…$15.00
DOI: http://dx.doi.org/10.1145/3023368.3023381

1 Introduction

Sparse Voxel Octrees (SVOs) have become increasingly popular, e.g., for raytracing indirect illumination and glossy reflections [Crassin et al. 2011]. In 2013, Sparse Voxel DAGs were introduced, which heavily compress voxelized *geometric* information [Kämpe et al. 2013]. Only recently, it has been investigated how to connect the DAG's geometric data with material data, and how to compress the material data separately [Dado et al. 2016; Williams 2015].

Our first contribution in this paper is a novel method for connecting material information to DAGs, which in practice does not increase the size of the DAG by more than 1%. Next, we concentrate on the compression of voxel color data. We focus on colors, although any other similar data should be compressible with our methods, such as surface roughness or normals.

While the voxel-color data certainly corresponds to colors in a 3D spatial domain, algorithms intended for compressing traditional 3D textures, or other volumetric data, will perform poorly since the information is very sparse. The colors are actually distributed over two-dimensional surfaces, but traditional 2D compression methods do not directly apply. Instead, after decoupling the geometry and color data, we are left with a compact one dimensional array which (depending on how the decoupling is done) may still have ample coherence.

Our second contribution enables efficient compression of voxelized surface colors using conventional image compression methods. By mapping the one-dimensional array to a two-dimensional image, using a space-filling curve, much of the coherency can be retained in the image and we can therefore apply standard 2D image compression methods. We first demonstrate that modern, hardware accelerated, texture compression formats (BC7 and ASTC) can compress the data to 33% with very little loss in quality. This data can still be immediately accessed on the GPU with no extra performance cost. Next we show that conventional off-line image compression techniques can compress the data down to around 10%, with reasonable quality, in cases where the data shall be stored to disk or transmitted over a network.

Our third contribution is a novel compression format where we instead attempt to compress the array of colors immediately, without transforming it to an image. In the spirit of many 2D-block based algorithms, this algorithm divides the array of colors into blocks of sizes such that each block can be represented by two endpoint colors and one weight per original color which interpolates between these, without introducing an error higher than a specified threshold. This data can still be accessed in realtime, on the GPU, and, in most experiments, compresses the data almost twice as well as previous work, for similar quality.

2 Previous Work

Octrees have been used to represent 3D scenes for over three decades [Rubin and Whitted 1980; Jackins and Tanimoto 1980; Meagher 1982]. Despite octrees being a sparse format in itself, very high resolutions are non-trivial to fit in memory and render [Museth 2013]. We will only cover the most related methods that use voxels as the representing primitive rather than points [Elseberg et al. 2013] or triangles [Gobbetti and Marton 2005].

Sparse Voxel Octrees store voxelized objects in an octree format, where each node represents a non-empty voxel at that hierarchical level and, potentially, also stores its associated material information [Gobbetti et al. 2008; Crassin et al. 2009]. Laine and Karras [2011] introduce *Efficient Sparse Voxel Octrees*, which improve on the geometric shapes by storing contour data in each voxel. They also compress color and normal blocks of 2^3 voxels using DXT-based compression. Crassin et al. [2011] use cone tracing in an SVO to compute real-time ambient occlusion and indirect lighting.

Merging common subtrees Webber and Dillencourt [1989] compress binary cartographic images by using quadtrees and common-subtree merging, and Parsons [1986] use cyclic quadgraphs to represent 2D straight lines. Parker et al. [2003] extend to using common subtree merging for voxel octrees and achieve compression for axis-aligned regular structures, such as flat electrical circuits.

Sparse Voxel DAGs are based on the important observation that by removing the material information from the voxel data, common subtree merging often becomes up to three orders of magnitude more efficient [Kämpe et al. 2013]. Apart from direct visualization of extremely high-resolution models ($128K^3$) lacking colors, DAGs with only geometry can for instance be used for ambient occlusion and shadows [Sintorn et al. 2014; Kämpe et al. 2015]. Jaspe Villanueva et al. [2016] significantly improve on the compression by also searching for reflection symmetry of subDAGs in the x, y, and z directions. Furthermore, they use a frequency-based pointer compaction per hierarchy level and in total reduces the memory consumption up to two times. These optimizations can be used orthogonally with our suggested technique.

Decoupling geometry and material data A problem with DAGs is that they can only efficiently represent the *geometric* information in the DAG. Material information for the models is often desired, and an efficient connection between the DAG nodes and per voxel colors can be non-trivial. The reason is that simply inserting color indices into the nodes will destroy the subtree-merging opportunities.

An early work in this direction is the Perfect Spatial Hashing suggested by Lefebre et al. [2006]. Their method allows a lookup from any 3D point in space to a 2D image using hash tables. While that

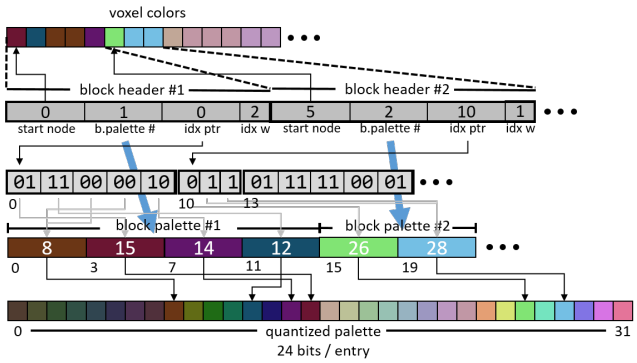


Figure 2: *In the format of Dado et al., each block stores the starting node, the index of a block palette, the first bit in a compact list of offsets to the block palette, and the width of each entry in that offset list.*

method may well be applicable to decoupling voxel geometry and colors, the new position for a voxel color in the lookup texture will inherently be random. This is not good for large data-sets since it means that caching will work poorly, but more importantly, for our purposes, it means that any coherence existing in the original voxel colors will be lost, which complicates subsequent compression of the data.

Very recently, Williams and Dado et al. presented two separate approaches to connect voxels with colors [Dado et al. 2016; Williams 2015] by inserting index information that does not harm the merging possibilities. Each node's color index is defined by the node's order according to a fixed-order full tree traversal of the corresponding SVO. Both methods insert a pre-computed value per child pointer in the DAG, while our approach allows using only a value per node, which is an important difference, since the former roughly doubles the amount of data in the DAG node, leading to nearly a doubling of the DAG memory consumption [Dado et al. 2016].

In short, Williams stores, per pointer, the number of empty SVO voxels in a corresponding full subtree. These values reach 10^{15} for scenes of $128K^3$ and heavily influence the node sizes. They also need an indirection table that grows exponentially, requiring hundreds of MB even for small resolutions of $1K^3$. That solution is therefore infeasible for large resolutions. The solution suggested by Dado et al. will be explained and further discussed in the next section.

Compressing voxel colors In addition, Dado et al. suggest a method for compressing voxel attributes. They first quantize all colors that exist in the scene to obtain a subset of colors which are chosen as the global palette. Next, they divide the original array of colors into blocks, where each block will be associated with a smaller *block palette*, which in turn points into the global palette. If possible, several blocks can share the same block palette. Each block also points into a compact list of offsets to the block-palette with one entry per original color. The complete data structure is illustrated in Figure 2. Obtaining good compression rates with high quality requires that the color space can be quantized to a sufficiently small subset, and that there is a possibility for many blocks to share block-palettes.

Volume visualization of semi-transparent data in grids is used in, for instance, medical visualization [Guthe et al. 2002], and a complete overview is outside our scope. However, Balsa Rodriguez

et al. [2014] provide a detailed state-of-the-art report on real-time GPU-based compressed-volume rendering. What differs these methods from our approach is that they compress three-dimensional voxel structures (e.g., using cosine transforms and wavelets), while we target compressing voxelized surface data. Also, these methods typically target grid resolutions of up to about $1K^3$, while we target resolutions of, e.g., $32K^3$, i.e., 5 orders of magnitude higher.

3 Decoupling Voxel Geometry and Attributes

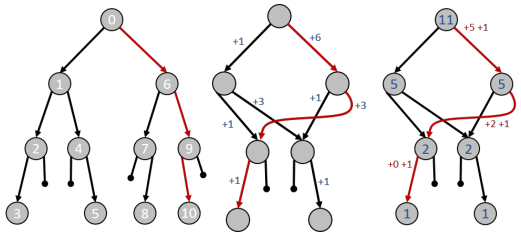
Whether the geometry information in the voxel data is compressed using a DAG [Kämpe et al. 2013] or stored as an SVO, it can be beneficial to decouple geometry information from voxel-attribute (e.g. color) information. In many cases, only the geometry information is required for querying (e.g. ray tracing) the data-structure, and isolating the geometry information can lead to better cache coherency. Additionally, if geometry and color information is stored at the same resolution, the geometry information will require much less memory and might, for instance, fit in GPU memory while the color data does not. By separating geometry and colors, raytracing of the data structure can be performed on the GPU while the color lookup can be done on the CPU.

When the geometry is stored as a traversable SVO, decoupling colors is trivial. Since nodes are fixed size, the index pointing to where the node's children can be found can also be used as an index into a separate array of node colors. When the geometry is stored as a DAG, however, the index is used to point out a node that may be shared by several different SVO subtrees with different color content and so a direct index can not be stored in the DAG.

Dado et al. [2016] achieve their voxel-to-color-index connection by storing, for each child pointer, the difference in color index for the child and parent. The actual voxel index can then simply be computed during traversal by summing all the offsets along the current path, from the root to the node (see Figure 3). Since the offsets will be identical for identical subtrees, the DAG still compresses as well as without this information. Unfortunately, the pointers make up the vast bulk of the information required to store a DAG so, by adding a 32-bit offset to each 32-bit pointer, the size of the DAG is effectively doubled. Dado et al. alleviate this problem by noticing that most offsets will be small, and only require a few bits of storage, but this complicates the way the DAG is stored and can be very detrimental to the performance of, e.g., ray tracing the DAG.

In this paper, we note that by storing per DAG node a *voxel count* (i.e. the number of voxels represented in the node's subgraph), the number of voxels preceding a specific node in a full-tree traversal can be computed using a running sum of the voxel counts during traversal. We start with a zero-initialized index and when traversing from any node, $\mathbf{p}$, to the next node, $\mathbf{n}$, along a path from the root to a leaf, the voxel counts of all $\mathbf{n}$'s preceding siblings plus one (for the color occupied by $\mathbf{p}$) are added to the index. Consequently, the index will continuously represent the voxel index for $\mathbf{p}$. Figure 3 illustrates the index computation for a specific node.

Thus, with our method we only need to store an additional value *per node*, which is much more memory efficient in a DAG. In our DAG implementation, for alignment purposes, we use a 32-bit word to store the 8-bit child mask and then up to eight 32-bit child pointers (one for each non-empty child). For resolutions up to around $1K^3$, the voxel-count value typically fits in the 24 unused bits, leading to no increased storage requirements. For larger resolutions, at the upper levels, we store the voxel count in a separate 32-bit word. These nodes are, however, so few that the memory overhead is typically far less than 0.1%. Thus, we effectively need 24 bits on average per node for the color connection, compared to 8-12 bytes on average using Dado et al.'s method. We do not use any pointer nor value compression [Jaspe Villanueva et al. 2016; Dado et al. 2016], although that could be added orthogonally.



(a) *SVO* **(b)** *[Dado]* **(c)** *Ours*

Figure 3: *Transforming an SVO to a DAG with color index information. a) A simple SVO with nodes labeled with their depth-first order. b) The method of Dado et al. [2016]. To each pointer is appended the offset in index from the parent node to its subnodes. c) Our method. With each node, we store the number of voxels (alternatively leaf-voxels) contained in the subtree.*

The array of colors can be generated by traversing the original SVO depth first such that the voxel colors will appear in the order of a Morton curve. Then, much of the existing coherency between colors will be retained. This is important for the compression algorithms that will be discussed in the next section. In order to retain even more of the coherency, the array can be reordered to follow a Hilbert curve instead. Then, when traversing the DAG to find the voxel index, we simply must make sure to consider the Hilbert ordering of each node's children when deciding which children precede the one we traverse to.

Regardless of which method is used to calculate the voxel index, we have a choice of storing only leaf-voxel colors, or the color of all nodes, in our array. In the former case, only the number of leaf nodes contained in the subtrees of $\mathbf{n}$'s siblings are added to the index as we traverse the tree. Unfortunately, when all nodes' colors are stored, the colors of internal nodes will be interleaved with nodes at lower levels, which leads to poor memory access patterns and might negatively affect color compression. In the remainder of this paper, we only consider leaf-node colors.

4 Attribute Compression

Having decoupled voxel colors from the geometry information, we now search for a means of compressing the color information without introducing too disturbing artifacts. Since the geometry information can be very efficiently compressed using a DAG, the color information will usually consume much more memory, even if the geometry is stored at higher resolutions. In this section we will discuss a number of novel approaches to compressing the color information, as suitable in different scenarios.

4.1 Compression using a 2D Mapping

There are an abundance of 2D image compression algorithms, and modern GPUs even have specific hardware support for decompressing 2D images, so naturally it would be desirable to be able to use such algorithms on colors of voxelized surfaces. Therefore, we first consider transforming our one-dimensional array of colors into a two-dimensional image. Image-compression algorithms rely on there being coherence in two dimensions so the chosen mapping must attempt to retain the coherence existing in the array when transformed to an image. We chose to map our array onto an image by following a 2D space-filling curve and then applying conven-

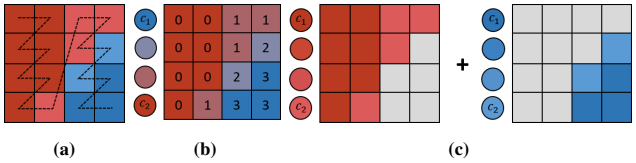


Figure 4: *a) The original block of colors. b) With BC1, sharp contrasts are blurred. Potentially bleeding across separate surfaces. c) Formats with partitioning can have separate endpoints for separate surfaces.*

tional image-compression algorithms to it. Specifically, we either generate the image using a Morton or a Hilbert mapping.

4.1.1 Hardware Texture Compression

Most modern GPUs contain fixed-function hardware designed to decompress textures during lookup at virtually no performance cost. Being able to utilize this hardware for looking up the color of a voxel is highly desirable, and therefore, we have evaluated the suitability of three significantly different such formats.

BC1. Perhaps the simplest, and certainly most supported form of texture compression is the BC1 (also called DXT1 or S3TC) format. Here, the image is divided into blocks of 4x4 pixels and, for each block, two 16-bit colors c_1 and c_2 are stored, along with a 2-bit weight, w_{ij} per pixel. To decompress the color c_{ij} of the pixel (i, j) in the block, the dedicated hardware will calculate $c_{ij} = (w_{ij}/3)c_2 + (3 - w_{ij}/3)c_1$. Thus, an approximation of 16 24-bit colors can be achieved in 64 bits (compression ratio is 1:6).

BC7 is a more recent format, supported by most recent GPUs. For us, the most important difference from BC1 is that with BC7 each block can be divided into two or more *partitions*, each with its own color end points. To specify how the blocks will be partitioned, the compressed block contains a few bits choosing a partitioning from a fixed set. This allows for much better quality in the decompressed image when the original image is not well described by an interpolation between two colors. The block size for BC7 is 4x4 pixels and the compression ratio is 1:3.

ASTC is similar to BC7 but much more flexible. It is only supported by some recent GPUs. With ASTC, the block size can be chosen quite freely and the partitioning is done by a random number generator, rather than a hardware table, allowing for very different partitionings than BC7.

The images we compress are very different from the natural images these formats were designed to handle, which is very evident when using BC1 compression. Consider the example in Figure 4a. The colors of the original voxel-color array describe two different surfaces and are laid out in a Morton curve. There is a distinct jump in the color space as we move from one of the surfaces to the other. With BC1, the whole block will be approximated by linearly interpolating between two colors and, while the result may work acceptably for a natural image, in our case it results in the red surface being tainted with blue hues, and vice versa, resulting in objectionable artifacts. This problem is greatly alleviated by the partitioning mechanism available in the BC7 and ASTC formats. Two surfaces that happen to occupy the same block in the 2D image will be compressed using two separate partitions and the decompressed colors will be much closer to the original.

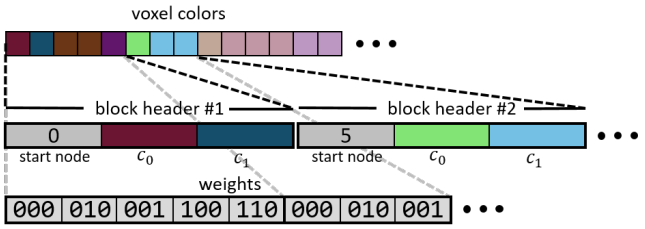


Figure 5: *In our suggested format, the array of voxel colors are divided into blocks of varying length that can be described with two endpoint colors, c_0 and c_1 and a weight per color that interpolates between these.*

4.1.2 Conventional Off-line Image Compression

Contrary to hardware-accelerated texture-compression formats, conventional off-line image-compression techniques do not have a requirement of being randomly accessible. With these formats, the entire image is usually decompressed into raw format for display or modifications. Therefore, much higher compression ratios are often achievable. We have explored whether compressing an array of voxel data with off-line image compression, by mapping it to an image using a space filling curve, is viable when the data shall be, e.g., stored to disk or transferred over a network. In our experiments, as detailed in section 5.3, we have evaluated three well known, and distinctly different, formats. We will briefly overview their characteristics in the remainder of this section.

JPEG is a common format for heavily compressing photographs and natural images. It works by first transforming the RGB data to $Y'C_bC_r$ data and then transforming blocks of 8x8 pixels into the frequency domain using the discrete cosine transform. In this domain, the data is quantized which will cause the image to be compressed in the final entropy-coding stage. Thus, a lot of the high-frequency information in the image is discarded, potentially causing the same kind of artifacts as we expect from BC1.

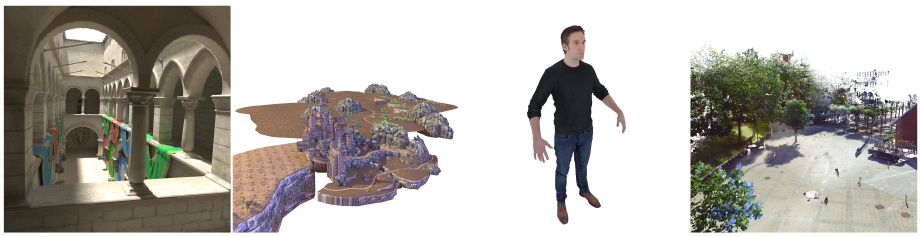
JPEG2000 is a more recent format where the image is instead wavelet transformed hierarchically as a first step. The resulting coefficients are then quantized to reduce the number of bits required to store them and facilitate entropy coding. While this approach would seem better suited to avoid the problem described in Figure 4b, quantizing a single coefficient can affect a large region of the image, which in turn can affect large volumes containing separate surfaces in the voxel data.

PNG itself is a lossless format, but encoders often provide the option of preprocessing the image before compression so as to achieve a smaller file size. In our experiments we have used an encoder that first reduces the number of colors in the image by clustering in color space.

All of these formats naturally run the risk, at high compression ratios, of blurring together surfaces that are actually separate and the pros and cons of each format will be discussed in Section 5.

We would like to mention that while it is tempting to simply pad the voxel-color array so that two surfaces of different colors do not occupy the same block, we have not yet found a way of achieving this without destroying either the decoupling information in the DAG or the potential for merging common subtrees.

Table 1: *The scenes used in the evaluation of our algorithm. The SVO size reported is what would be obtained with a traversable SVO where each internal node is two 32-bit words (mask and pointer), and leaf nodes are 4x4x4 blocks described by a 64 bit word.*



Scene	SPONZA	EPIC	BODY	CAMPUS
Resolution	4096^3	32768^3	16384^3	32768^3
Leaf Voxels	147M	848M	167M	97M
Color Data Size	420MB	2.4GB	477MB	277MB
SVO Size	100MB	719MB	168MB	139MB
DAG Size	9MB	189MB	96MB	118MB

4.2 Variable Bitrate Block Encoding

When the compressed voxel data is to be queried in realtime, the off-line image compression algorithms just described are not an option. Instead, for traditional texturing, the fixed bitrate block encodings described in Section 4.1.1 are used. However, as will be revealed in Section 5.1, for high-quality decompressed images, the compression ratio is fairly low for these formats. As reviewed in Section 2, Dado et al. [Dado et al. 2016] suggest a compression format that lies somewhere in the middle ground between off-line image compression formats and fixed bitrate block encodings, requiring a binary search to locate the containing block but still being accessible in realtime environments. The novel format that will be described in this section similarly lies in this middle ground.

4.2.1 Data Structure and Decompression

In our data structure, the array of colors is divided into blocks that can have any length. Just like the BC1 format, a block carries two endpoint colors, c_0 and c_1, and each color in the block is described as an interpolation between these two colors. Thus, our entire data structure consists of an array of B block headers, b_i, and an array of N weights, w_j, where N is the total number of voxel colors. The block headers contain one index specifying the voxel that begins this block and the endpoint colors. The weights are all of constant bitwidth, W, (usually 2-4 bits) and so are directly indexable. Our data structure is illustrated in Figure 5

When the voxel index, j, has been found, e.g. by raytracing a DAG, the decompressed color is calculated by first performing a binary search through the block headers to find the block that contains this voxel color. Then, the voxel color is decompressed as:

$$c_j = (w_j/W)c_1 + (W - w_j/W)c_0. \quad (1)$$

Thus, decompressing the datastructure is very simple and intuitively it should be able to fit our specific data quite well. In the next section we will discuss the method with which we choose the block division such that the decompressed colors will be sufficiently close to the original.

4.2.2 Compression

We initialize our compression algorithm with the maximum accepted error allowed for a decompressed color. This error measure can be chosen arbitrarily, but we simply look at the distance, e, (in sRGB or CIELAB space) between the original and the decompressed color and supply a specific error threshold, e_t. The objective now is to find the smallest set of blocks for which this error is sufficiently low. This is obviously a very difficult problem and we make no attempt at finding the optimal solution. Instead, we present in this section a heuristic that works well.

First, each color is assigned its own block, and the blocks are stored as a doubly linked list. We then greedily merge blocks in several iterations, until no more blocks can be merged. In each iteration, we start at the *second* block in the list and calculate a score for merging with either the left or the right block. We simply use the mean square error of all compressed colors in the potential new block as our score, or a negative number if any color was below the error threshold. The block is then merged with the highest scoring neighbor. If neither neighbor could be merged within the error threshold, the block is left as is. We then move *two* blocks to the right and repeat the merging procedure until we reach the end of the list. As long as *any* block was merged with another, we then start a new iteration at the second block in the list.

The algorithm is detailed in Algorithm 1. We additionally maintain which blocks were modified in the last pass, so that we can skip redundant calculations when neither it nor its neighbors have changed.

4.2.3 Variable Bitwidth For Weights

In our implementation, the number of bits used to store the interpolation weight per color is fixed. It may be that even better results could be obtained by carefully choosing the number of bits per block so that, for instance, a large block that only contains two colors could be encoded with a single bit per weight. This comes at the cost of a more expensive block header, however. Since the weight array would no longer be directly indexable, the block header would have to carry an index to the first element and the bitwidth for this block. The compression algorithm would also have to be more complex in order to decide which bitwidth is most suitable for a block. A thorough evaluation of using variable bitwidths for weights is left as future work.

We have implemented a small subset of this idea, however. In one version of our algorithm, we first locate blocks that can be described by a single color without surpassing the error threshold. This is done with a single greedy sweep over the voxel color data, and then, the compression algorithm continues as before.

Algorithm 1: Find the set of compressable blocks

input : e_t- the error threshold
blocks- a linked list of blocks, initially one per color

Procedure `Eval` (block.start, block.end)
  $(c_0, c_1, w) \leftarrow$ `LeastSquaresFit` (block.start, block.end) ;
  if *all color errors* $< e_t$ **then**
    return mean square error;
  else
    return -1;

Procedure `Compress` (e_t, blocks)
  do
    block $\leftarrow$ second element in blocks;
    while *(not at end of list)* **do**
      leftmse $\leftarrow$ `Eval` (block.left.start, block.end) ;
      rightmse $\leftarrow$ `Eval` (block.start, block.right.end) ;
      if *(*leftmse > 0*)* OR *(*rightmse > 0*)* **then**
        if leftmse $<$ rightmse **then**
          `Merge` (block,block.left)
        else
          `Merge` (block,block.right)
      block $\leftarrow$ block.right.right;
  while *(any block was merged)*;

5 Results

To evaluate the compression algorithms, we have chosen a set of four very different types of scenes, shown in Table 1. SPONZA and EPIC, are voxelized video-game scenes with path-traced colors. BODY is a high-resolution mesh with high-resolution color textures obtained by 3D scanning. CAMPUS is a university campus captured by a laser scanner, where the original point cloud is approximately 8GB of data (500M points) and has been voxelized without any surface reconstruction.

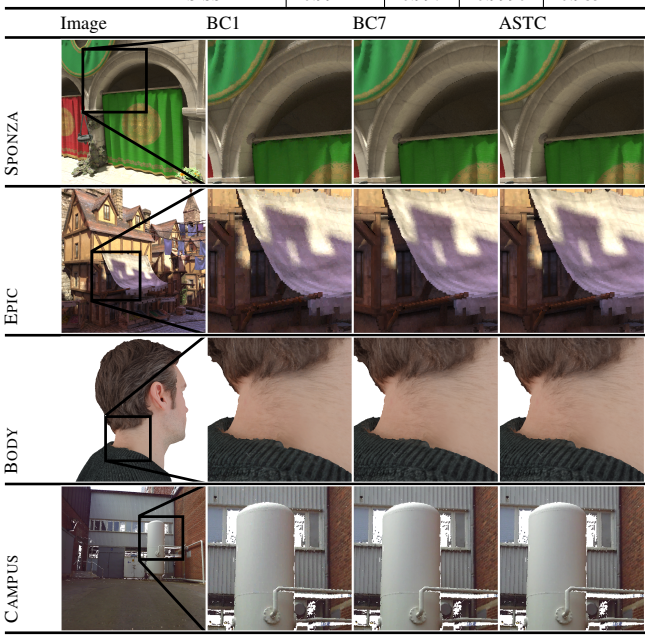
All scenes have been converted to a geometry DAG as described in [Kämpe et al. 2013] with voxel-color connections inserted as described in Section 3. Once the one-dimensional array of colors has been obtained, we compress that using a number of different algorithms listed below. We evaluate the quality of the compressed data first globally by calculating the *Global Mean Square Error* (GMSE) of all compressed colors compared to the ground truth. The error is the distance, e, (in sRGB space) between the original and decompressed color. The GMSE provides some notion of the quality of the compressed data as a whole, but if the scene contains, for instance, large areas that compress very well, the GMSE may not be a very good measurement.

Therefore, we also chose a number of viewpoints in each scene and render an image for each. For these images, we calculate the *Mean Squared Error* (MSE) and *Multi Scale Structural Similarity Index* (MS-SSIM). These numbers correspond reasonably to how the difference in quality of the rendered images are perceived. However, especially at lower quality settings, which compression method to prefer can be highly subjective. Our complete results occupy too much space to fit in the paper and are instead supplied as supplementary material.

To inspect the scene, we have implemented a real-time raytracer in CUDA, which calculates the primary hit point per pixel by intersecting a ray with the DAG. The raytracer outputs a voxel coordinate per pixel, and in a second pass the color is obtained from the voxel color data. In the two Variable Bitrate Block-Encoding algorithms, the compressed data is stored and evaluated on the GPU,

Table 2: *Comparison of hardware texture formats.*

		SPONZA	EPIC	BODY	CAMPUS
BC1	*Compression*	16%	16%	16%	16%
	Global MSE	3.9	2.4	3.3	18.4
	MSE	10.7	22.4	1.8	4.1
	MS-SSIM	0.953	0.974	0.996	0.942
BC7	*Compression*	33%	33%	33%	33%
	Global MSE	0.42	0.3	0.3	3.3
	MSE	1.1	1.8	0.2	0.6
	MS-SSIM	0.992	0.997	0.9995	0.983
ASTC	*Compression*	33%	33%	33%	33%
	Global MSE	0.5	0.3	0.3	4.0
	MSE	1.2	1.7	0.2	0.7
	MS-SSIM	0.992	0.997	0.9996	0.983



but for the other formats we decompress the data on the CPU before storing it on the GPU. While we could have read the hardware texture formats in an OpenGL compute, or fragment, shader, our CUDA implementation does not support that. In the cases where the uncompressed data is too large to reside on the GPU, our implementation falls back to an identical CPU path for the color lookup.

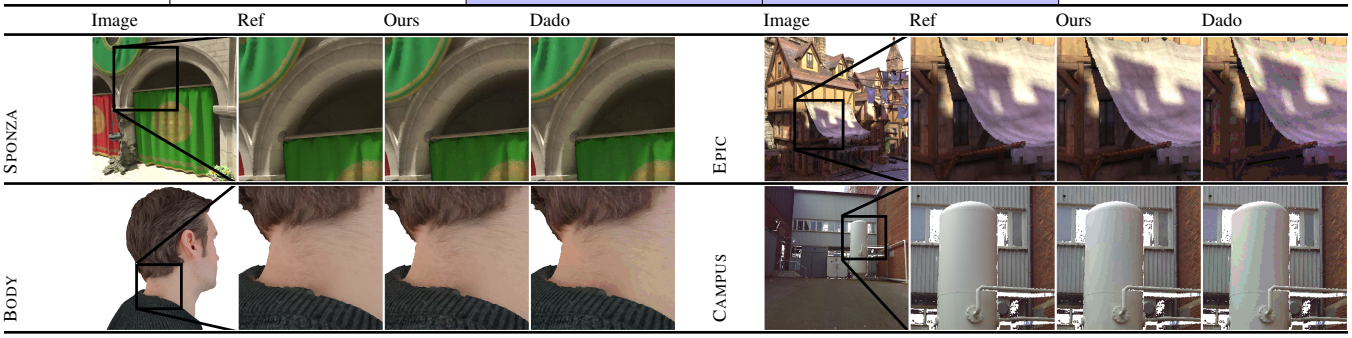
For the hardware texture formats and the conventional image compression algorithms, the voxel-color array is first transformed into an image, as described in Section 3, and then compressed using off-the-shelf software. Since our voxel data is often very large, we have split it into partitions that each make up one 2048x2048 image, and compress these separately. If the compressed textures were to be accessed on the GPU, each compressed image could be read into a slice of an array texture. This also lets us control the maximum amount of padding required to fit our data into a square texture.

- **BC1, BC7** AMD Compressonator.
- **ASTC** ARM-software ASTC-encoder.
- **Ours** Our implementation of the algorithm described in Section 4.2.
- **Dado** Our implementation of the algorithm described in the paper by Dado et al. [2016].
- **JPG, JPG2000** Image Magick.
- **PNG** pngout and pngquant [Silverman 2016; Lesiński 2016].

The error, e, has been calculated in sRGB space. We have also run

Table 3: *Comparison of variable bitrate block-encoding formats. Our and Dado et al.'s formats compared at varying quality settings. The images show results at the highlighted settings.*

		SPONZA				EPIC				BODY				CAMPUS			
		Comp.	GMSE	MSE	MSSSIM	Comp.	GMSE	MSE	MSSSIM	Comp.	GMSE	MSE	MSSSIM	Comp.	GMSE	MSE	MSSSIM
Ours	e_t: 0.025	18.8%	1.0	2.0	0.976	20.2%	0.4	1.8	0.994	17.2%	1.4	1.3	0.995	44.2%	2.1	1.7	0.952
	e_t: 0.038	15.5%	2.1	4.5	0.951	15.3%	0.8	3.8	0.988	13.3%	3.7	3.3	0.990	29.0%	4.2	3.6	0.914
	e_t: 0.050	14.2%	3.5	8.4	0.914	13.9%	1.3	6.5	0.981	12.7%	5.2	5.5	0.985	23.2%	7.2	7.0	0.823
	e_t: 0.100	13.1%	9.1	29.8	0.820	12.7%	3.9	27.4	0.943	12.5%	17.1	16.7	0.951	15.0%	20.9	25.4	0.633
Dado	lossless	37.2%	0.0	0.0	1.000	89.9%	0.0	0.0	1.000	65.2%	0.0	0.0	1.000	103%	0.0	0.0	1.000
	colors: 16K	21.2%	0.6	1.2	0.980	31.9%	0.2	1.4	0.994	35.0%	0.3	0.3	0.998	39.3%	1.9	2.0	0.939
	colors: 4K	17.1%	2.4	3.3	0.959	27.9%	0.7	3.8	0.986	31.7%	0.6	0.6	0.997	31.7%	6.7	5.5	0.859
	colors: 2K	15.2%	3.3	5.4	0.939	25.6%	0.9	6.0	0.980	30.6%	1.0	0.9	0.995	29.1%	9.0	9.5	0.819
	colors: 256	11.2%	19.5	31.6	0.800	18.8%	6.7	30.0	0.923	24.0%	4.7	4.9	0.977	21.4%	58.9	64.2	0.606



our experiments using the distance in CIELAB space, but found the results to be slightly worse in all cases, both for our algorithms and that by Dado et al.

All experiments have been performed for both the Hilbert and Morton order of both the depth-first traversal and the 2D space-filling curves. However, using a Hilbert-order was found to consistently provide very minor improvements. Since the Morton order is very common due to its simplicity, we therefore choose to only present results using the Morton order.

5.1 Hardware Texture Formats

Table 2 shows the results obtained when transforming the voxel color data to a 2D image, using a space-filling curve as suggested in Section 4.1, and compressing these images using hardware accelerated texture compression formats. With the simpler BC1 format, we obtain a compression of 16%, but compression artifacts are clearly visible. One source of artifacts are the discretization and color shifts that are inherent in the format, but we also see, especially in the SPONZA scene, that unexpected voxel colors turn up on otherwise smooth surfaces as explained in Section 4.1.1. A notable exception is the BODY scene, where the BC1 algorithm performs very well. This is most likely due to the scene having few thin or overlapping features, so that the vast majority of 2D blocks will contain only colors from one surface.

The BC7 and ASTC formats both generate high quality images and very low global errors and the obtained compression is identical at 33%. In Table 6, we show how different formats degrade with lower quality settings. The BC1 and BC7 formats do not allow for different settings, but with the ASTC format, we are able to choose to use even fewer bits per texel. While the quality of results quickly degenerates, the data can become very small and there may be scenarios, such as when the voxel data is used for glossy indirect reflections, where these settings are viable.

5.2 Variable Bitrate Block-Encoding Formats

In Table 3, we compare our novel compression scheme, described in Section 4.2, with our own implementation of the algorithm suggested by Dado et al. [2016]. For our format, the results are similar between the first three scenes. The compressed data is 17-20% of the original with virtually no perceptible error, 13-15% with very high quality, and we can push it down towards 12% (optimal with the chosen bitwidth for weights) with quality that can still be acceptable in some cases.

All results presented for our format use the 16-bit RGB565 format to store the color endpoints in the blocks. This gives slightly better compression results while achieving the same quality as if we use 24-bit color endpoints. We use three bits per weight; This gives the best result in almost all cases (the exception being a few of the highest quality experiments, where the size of the block headers strongly outweigh the size of the weights array).

The last scene, CAMPUS, is very challenging for both our and Dado et al.'s algorithms. There are two main reasons for this; First, the color information in the scanned data is merged from different cameras at different viewpoints and the real-world materials are often highly view-dependent. Thus, points on the same surface often have highly irregular colors even though they appear smooth in reality. Also, the resolution of the data is relatively low, so thin features (e.g., the many trees that are part of the scan) will cause very noisy colors to begin with. Thus, at the current resolution, our format with lower quality settings can be used if some error is acceptable, but otherwise, it is preferable to use our BC7 or ASTC formats described above.

On the EPIC and BODY scenes, our format is clearly the better choice, offering very high quality at compression ratios that cannot be reached with the other algorithm. In the SPONZA scene, however, Dado et al.'s format performs almost as well as ours, and much better than in any of the other scenes. We have investigated this further to demonstrate some important differences between the two formats.

Table 4: *Comparing the quality of voxel data compressed using conventional off-line image-compression formats.*

		SPONZA				EPIC				BODY				CAMPUS			
quality setting		Comp.	GMSE	MSE	MSSSIM	Comp.	GMSE	MSE	MSSSIM	Comp.	GMSE	MSE	MSSSIM	Comp.	GMSE	MSE	MSSSIM
JPG	95	11.7%	2.3	3.8	0.956	14.8%	1.0	3.6	0.987	14.9%	2.3	1.8	0.996	21.0%	5.8	3.0	0.924
	85	6.4%	8.3	21.1	0.869	7.3%	4.5	24.6	0.945	7.5%	10.4	7.4	0.988	10.4%	23.7	11.4	0.847
	75	4.7%	15.4	42.6	0.809	5.0%	8.3	55.4	0.909	5.1%	17.5	11.1	0.983	7.1%	41.7	19.4	0.801
	50	3.1%	35.2	89.8	0.727	3.0%	16.3	137.6	0.853	3.1%	25.5	15.5	0.975	4.1%	80.0	32.5	0.718
JPG2K	x5	20.0%	0.7	1.3	0.980	20.0%	0.9	5.8	0.982	20.1%	1.1	0.5	0.997	20.4%	5.8	3.8	0.893
	x10	10.1%	3.2	6.0	0.920	10.0%	3.6	29.3	0.938	10.0%	4.5	2.1	0.991	10.3%	18.0	11.8	0.750
	x20	5.0%	14.3	32.2	0.777	5.0%	11.1	130.4	0.823	5.0%	15.7	7.3	0.976	5.2%	47.1	27.3	0.607
	x40	2.6%	58.2	124.0	0.593	2.5%	25.5	326.3	0.692	2.5%	33.0	16.1	0.956	2.6%	96.4	57.1	0.491
PNG	lossless	29.8%	0.0	0.0	1.000	52.7%	0.0	0.0	1.000	50.2%	0.0	0.0	1.000	67.9%	0.0	0.0	1.000
	100	12.4%	0.8	4.6	0.967	21.5%	1.0	5.6	0.985	25.6%	0.6	0.5	0.998	21.7%	7.2	3.1	0.925
	70	4.9%	6.6	32.4	0.788	10.4%	3.8	11.2	0.970	7.6%	9.5	5.5	0.978	18.6%	9.4	4.4	0.892
	30	3.1%	18.8	71.7	0.641	6.4%	10.2	32.7	0.930	4.8%	23.2	16.5	0.932	11.6%	25.9	13.8	0.736
	10	2.3%	27.8	127.4	0.515	4.9%	17.0	49.7	0.897	3.6%	36.0	27.4	0.889	8.8%	42.5	21.1	0.639

Table 5: *A detailed evaluation of the* SPONZA *scene.* ORIGINAL *is the original scene and in* MODIFIED *we have removed the redundant geometry.*

		ORIGINAL		MODIFIED	
		Comp.	GMSE	Comp.	GMSE
Ours	e_t: 0.025	18,8%	1.0	20.3%	1.9
	e_t: 0.038	15.5%	2.1	16.0%	3.8
	e_t: 0.050	14.2%	3.5	14.7%	6.6
	e_t: 0.100	13.1%	9.1	13.1%	15.7
SCB	e_t: 0.025	16.3%	1.2	22.6%	2.0
	e_t: 0.038	12.2%	2.5	16.5%	4.0
	e_t: 0.050	9.8%	3.3	13.6%	5.2
	e_t: 0.100	7.7%	7.1	9.7%	12.0
Dado	lossless	37.2%	0.0	65.3%	0.0
	colors: 4K	17.1%	2.4	26.2%	3.7
	colors: 2K	15.2%	3.3	23.7%	5.3
	colors: 256	11.2%	19.5	17.0%	29.5

The main reason for these results, it turns out, is that the SPONZA scene contains one large box that lies *inside* the walls of the model and is invisible from any reasonable viewpoint. This box comprises almost 50% of all the voxels in the scene but, as it receives no light, all of them are completely black. When a large block can be described by a single color, our algorithm will find the block but will still require a fixed number of bits per color in the block to store the weight. With Dado et al.'s format, such blocks will find that they can use a block palette with *one entry*, and so they do not have to store any per-voxel information. We demonstrate that this is indeed the cause of the anomalies in Table 5, where we have removed the hidden box from the scene and re-run the experiments.

In Section 4.2.3, we suggest a modification to our algorithm that should handle cases such as this and these results are also given in Table 5. While the modified algorithm produces better results on the original SPONZA scene, it can only be considered a modest success, as the overhead introduced in the block headers mean that the modified scene (which has very few blocks that can be described by a single color) actually cause slightly worse compression at high quality settings. It does however provide a way of pushing compressed sizes even lower if quality is not the main concern.

5.3 Off-line Image Compression Formats

We have also compressed our voxel data using off-line image compression formats as described in Section 4.1.2 and the results are available in Table 4. Table 6 shows what type of artifacts are introduced and many more examples are available in the supplementary material. JPG and JPG2K can both produce fairly high quality results at compressed sizes from 15%-20%. Which of these two formats is preferable at lower quality settings is highly subjective, but we note that, as expected, JPEG2K introduces noise and color shifts on large continuous surfaces, while JPG introduces disturbing artifacts where surfaces are nearby and the voxels fall in the same 8x8 block. PNG compression of color-quantized data appears surprisingly efficient when only considering the GMSE numbers in Table 4, but in the second row of PNG images in Table 6, we can see that the compression comes at the cost of some areas having completely incorrect hues.

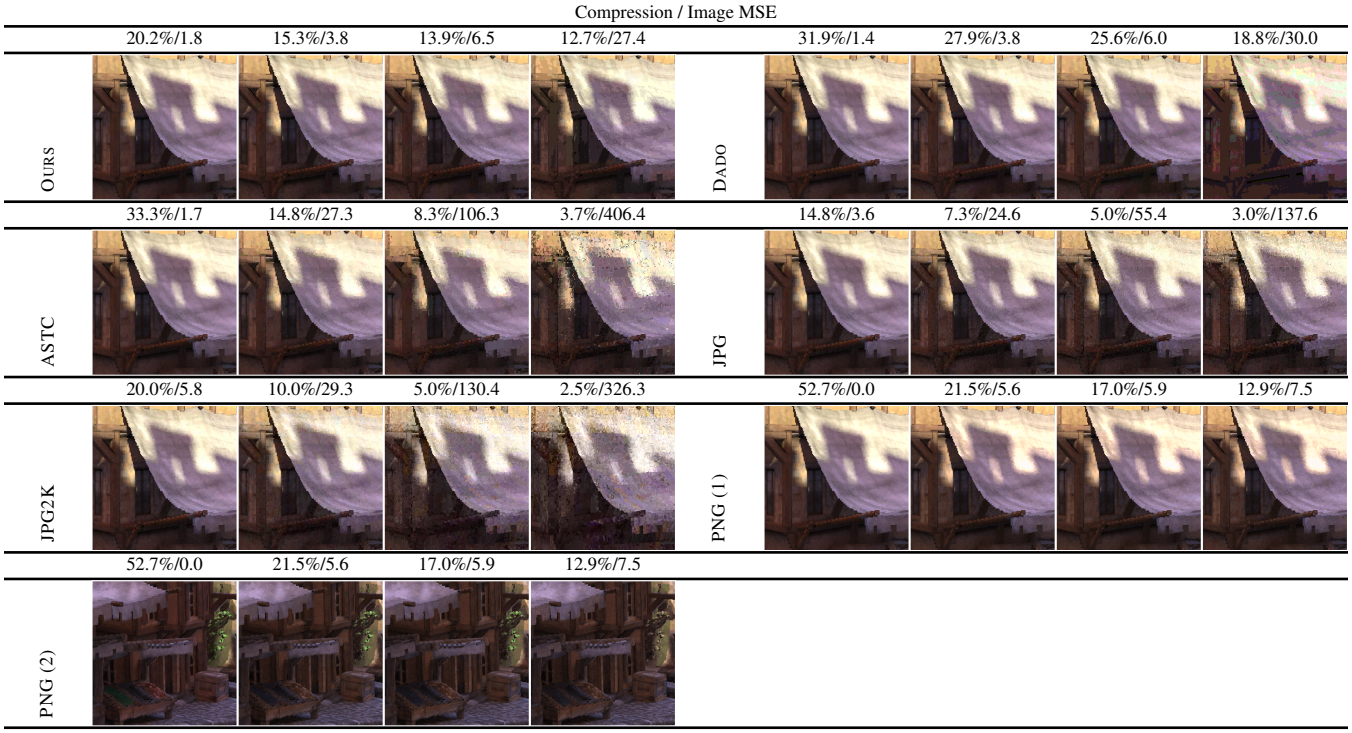
The compressed data is not directly accessible from, e.g., a shader or raytracer, but due to the simplicity of implementation and ready availability of software for 2D image compression, we believe these formats could be a good choice for compressing voxel data that is to be transferred over a network or stored to disk. It should be noted, however, that while we have not done a full analysis, voxel data compressed with either of the algorithms discussed in Section 4.2 can be further compressed by approximately 50% using off-the-shelf compression software (e.g., zip).

5.4 Performance.

Compression The BC1, BC7, ASTC, JPG, JP2K and PNG formats have all been compressed using external software. For the BC7 and ASTC formats, we have compressed using exhaustive search to find the optimal block configurations, and this is very time consuming (approximately 20h for the EPIC scene). We have also tried using faster settings where heuristics are used to improve speed and the results have been almost as good at a fraction of the time. For the rest of the image compression formats, compression time has at worst been a few minutes. For our implementation of Dado et al.'s algorithm, no effort has been put into optimization of the code, and on a single core of an Intel Core i7 3930K, compression took approximately 2h. The version of our own implementation that was used for all measurements has about the same performance as that of Dado et al., but we have subsequently optimized this algorithm by moving parts to the GPU. With that version, the EPIC scene took only 7 minutes to compress.

Lookup. In the table below, we present the time taken to render the images shown in Table 3 at 1024x1024 resolution using Our method and that of Dado et al., both when using per-pointer and per-node offsets to calculate the voxel-color index.

Table 6: *For each compression method, show how image quality changes with compression rate.*



		SPONZA	EPIC	BODY	CAMPUS
Raytracing		2.5ms	3.8ms	4.1ms	3ms
Lookup, Ours	per-pointer	0.6ms	0.6ms	0.6ms	0.6ms
	per-node	1.0ms	1.0ms	1.0ms	1.1ms
Lookup, Dado	per-pointer	0.6ms	0.6ms	0.8ms	0.6ms
	per-node	1.0ms	1.0ms	1.0ms	1.1ms
DAG size	per-pointer	15MB	336MB	168MB	184MB
	per-node	9MB	189MB	96MB	118MB

As expected, the lookup performance is somewhat faster when using per-pointer offsets, but the resulting DAG-sizes are also significantly larger (1.5x - 1.8x).

6 Conclusion and Future Work

We have described a method for decoupling DAG geometry and attribute data that has a very small impact on the final size of the DAG. We have also described a number of methods for lossy compression of the voxel attribute data. With our method for decoupling colors from geometry, the voxel-color data is ordered according to a 3D space-filling curve and contains much color coherency. We have shown that by transforming the color data to an image using a 2D space-filling curve, much of that coherency is retained, and conventional image compression formats, both off-line formats and hardware accelerated texture formats, can be used to achieve high quality results for compressed voxel data. In particular we have shown that, with the BC7 and ASTC formats, we can effortlessly provide 3x compression with very little loss in quality, enabling extremely fast color lookups from GPU shaders. Finally, we have suggested a novel real-time format, and compression algorithm, that consistently outperforms previous work and often achieves around twice the compression for equal quality.

We believe much higher compression ratios should be obtainable for an off-line format, and will, in the future, explore whether off-line image compression algorithms can be modified to better suit voxel-color data. We would also like to further explore the idea of compressing different blocks with different bitwidths for color endpoints and weights.

Acknowledgements

This work was supported by the Swedish Research Council under Grant 2014-4559. The EPIC scene is distributed with the Unreal Development Kit by Epic Games. The BODY scene is a freely available scene from Ten24. SPONZA is created by Frank Meinl at Crytek. The CAMPUS scene is courtesy of Jonathan Berglund, Erik Lindskog and Björn Johansson, and was obtained as described in their recent paper [Lindskog et al. 2013].

References

BALSA RODRIGUEZ, M., GOBBETTI, E., IGLESIAS GUITIÁN, J., MAKHINYA, M., MARTON, F., PAJAROLA, R., AND SUTER, S. 2014. State-of-the-art in compressed gpu-based direct volume rendering. *Computer Graphics Forum 33*, 6 (September), 77–100.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. ACM I3D*, ACM.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Pacific Graphics 2011) 30*, 7 (sep).

DADO, B., KOL, T. R., BAUSZAT, P., THIERY, J.-M., AND EISEMANN, E. 2016. Geometry and attribute compression for voxel

scenes. *Computer Graphics Forum (Proc. Eurographics) 35*, 2 (may).

ELSEBERG, J., BORRMANN, D., AND NÜCHTER, A. 2013. One billion points in the cloud – an octree for efficient processing of 3d laser scans. *Journal of Photogrammetry and Remote Sensing 76*, 76 – 88.

GOBBETTI, E., AND MARTON, F. 2005. Far voxels: A multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (July), 878–885.

GOBBETTI, E., MARTON, F., AND IGLESIAS GUITIÁN, A. J. 2008. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7, 797–806.

GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. 2002. Interactive rendering of large volume data sets. In *IEEE Visualization*, 53–60.

JACKINS, C. L., AND TANIMOTO, S. L. 1980. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing 14*, 3, 249 – 270.

JASPE VILLANUEVA, A., MARTON, F., AND GOBBETTI, E. 2016. SSVDAGs: Symmetry-aware Sparse Voxel DAGs. In *Proc. ACM i3D*.

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2013. High resolution sparse voxel dags. *ACM Transactions on Graphics 32*, 4.

KÄMPE, V., SINTORN, E., AND ASSARSSON, U. 2015. Fast, memory-efficient construction of voxelized shadows. In *Proc. ACM i3D*.

LAINE, S., AND KARRAS, T. 2011. Efficient sparse voxel octrees. *IEEE TVCG 17*, 1048–1059.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. *ACM Trans. Graph. 25*, 3 (July), 579–588.

LESIŃSKI, K., 2016. pngquant.

LINDSKOG, E., BERGLUND, J., VALLHAGEN, J., AND JOHANSSON, B. 2013. Visualization support for virtual redesign of manufacturing systems. *Procedia CIRP 7*, 419 – 424.

MEAGHER, D. 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 2, 129–147.

MUSETH, K. 2013. Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (July), 27:1–27:22.

PARKER, E., AND UDESHI, T. 2003. Exploiting self-similarity in geometry for voxel based solid modeling. In *Proc. Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, 157–166.

PARSONS, M. S. 1986. Generating lines using quadgraph patterns. *Comput. Graph. Forum 5*, 1 (Mar.), 33–39.

RUBIN, S. M., AND WHITTED, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph. 14*, 3 (July), 110–116.

SILVERMAN, K., 2016. Ken silverman's utility page.

SINTORN, E., KÄMPE, V., OLSSON, O., AND ASSARSSON, U. 2014. Compact precomputed voxelized shadows. *ACM Transactions on Graphics 33*, 4.

WEBBER, R. E., AND DILLENCOURT, M. B. 1989. Compressing quadtrees via common subtree merging. *Pattern Recognition Letters 9*, 3, 193 – 200.

WILLIAMS, B. R. 2015. Moxel DAGs: Connecting Material Information to High Resolution Sparse Voxel DAGs. Master's thesis, California Polytechnic State University.