Compressing Color Data for Voxelized Surface Geometry

Dan Dolonius, Erik Sintorn, Viktor Kämpe, and Ulf Assarsson

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. In this extension of our I3D paper with the same title [1], we first 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.

Index Terms—Voxel, sparse voxel octree, directed acyclic graph, space filling curve, color compression, ASTC, BC, JPEG, PNG

1 Introduction

Sparse Voxel Octrees (SVOs) have become increasingly popular, e.g., for raytracing indirect illumination and glossy reflections [2]. In 2013, Sparse Voxel DAGs were introduced, which heavily compress voxelized *geometric* information [3]. 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 [4], [5].

Our first contribution in this paper is a novel method for connecting material information to DAGs, which does not increase the size of the DAG by more than 0.1 percent compared to the work by Kämpe et al. [3]. This is a significant improvement over previous work [4], where the additional data roughly doubles the size of the DAG or, at the expense of complicating addressing logic and reduced rendering performance, causes an overhead of 30 percent.

Next, we concentrate on the compression of voxel color data. Although many types of material properties, such as surface normals or roughness, should be compressible with our methods, we focus on diffuse colors, since a fair evaluation of the quality of all types of material properties would be out of scope for this article. 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, for the first time, 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 percent 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 percent, 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 varying 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. We compare two versions of this algorithm. In the first, the number of bits per weight is fixed globally, and in the second, the number of bits per weight is chosen per block to minimize the memory consumption. Compared to previous work [4], our compressed data is usually less than half the size at similar quality, is simpler to implement, and scales to extremely high compression rates, as can be seen in Fig. 1. The colors can be decompressed in less than a millisecond at 1024^2 resolution on a GTX1080.

- *The authors were with the Department Computer Science and Engineering, Chalmers University of Technology, Gothenburg 412 58, Sweden. E-mail: {dolonius, erik.sintorn, kampe, uffe}@chalmers.se.*

Manuscript received 31 Mar. 2017; revised 14 July 2017; accepted 26 July 2017. Date of publication 18 Aug. 2017; date of current version 31 Dec. 2018. (Corresponding author: Dan Dolonius.)
Recommended for acceptance by C. Wyman.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TVCG.2017.2741480

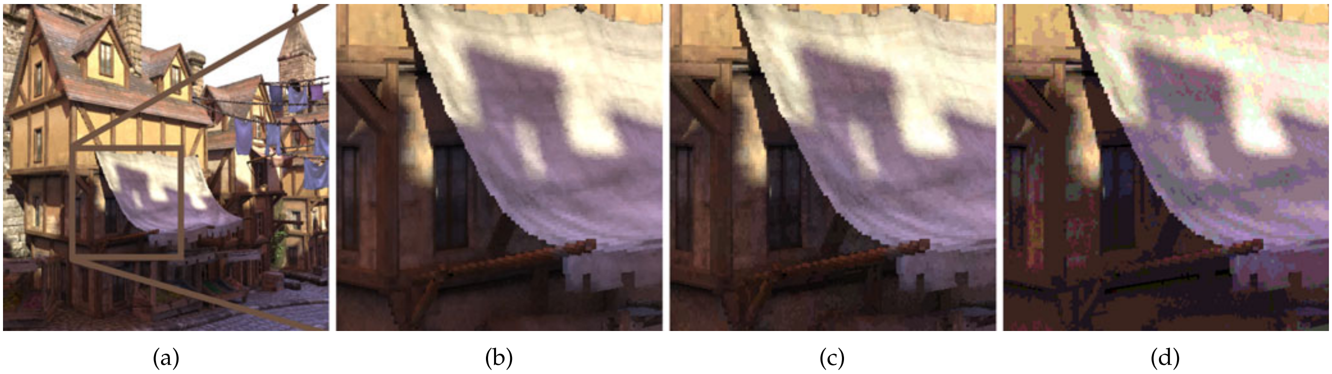


Fig. 1. The EpicCitadel scene, with precomputed illumination, voxelized at resolution 32768^3. a) Reference, 24-bit colors, 2.4GB . b) Our BC7 compression, 800 MB (30 percent), MS-SSIM: 0.99. c) Our variable bitrate block encoding, 219 MB (9.03 percent), MS-SSIM: 0.97. d) The method by Dado et al. [4] 456 MB (18.8 percent), MS-SSIM: 0.92.

2 Previous Work

Octrees have been used to represent 3D scenes for over three decades [6], [7], [8]. Despite octrees being a sparse format in itself, very high resolutions are non-trivial to fit in memory and render [9]. We will only cover the most related methods that use voxels as the representing primitive rather than points [10] or triangles [11].

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 [12], [13]. Laine and Karras [14] 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. [2] use cone tracing in an SVO to compute real-time ambient occlusion and indirect lighting.

Merging Common Subtrees. Webber and Dillencourt [15] compress binary cartographic images by using quadtrees and common-subtree merging, and Parsons [16] use cyclic quadgraphs to represent 2D straight lines. Parker et al. [17] 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 [3]. 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 [18], [19]. Jaspe Villanueva et al. [20] 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. [21]. Their method allows a lookup from any 3D point in space to a 2D image using hash tables. While that 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 [4], [5] 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 [4].

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 Fig. 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 [22], and a

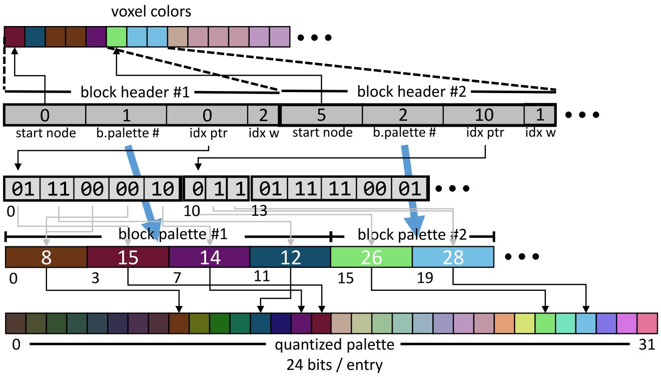


Fig. 2. In the format of Dado et al., each block stores the starting node, the index of a block palette, the first bit in a compact list of offsets to the block palette, and the width of each entry in that offset list.

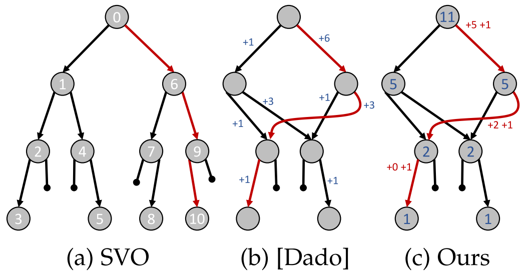
complete overview is outside our scope. However, Balsa Rodriguez et al. [23] provide a detailed state-of-the-art report on real-time GPU-based compressed-volume rendering. What differs these methods from our approach is that they compress three-dimensional voxel structures (e.g., using cosine transforms 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 [3] 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, ray-tracing 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. [4] 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 Fig. 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.



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

Fig. 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. [4]. 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.

In this section, we will describe a method where the additional information can mostly be stored in the 24 bits per node that are otherwise used as padding to achieve aligned memory accesses, and therefore has a negligible memory overhead. Even compared to a DAG without padding, our added information only adds approximately 15 percent overhead. Dado et al. further suggest a method for compressing their offsets, achieving a memory overhead of approximately 35 percent [4]. While this could orthogonally be added to our method as well, reducing our overhead to approximately 5 percent, it complicates addressing logic and Dado et al. [4] report approximately halved performance in rendering when using compressed offsets.

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, **p**, to the next node, **n**, along a path from the root to a leaf, the voxel counts of all **n**'s preceding siblings plus one (for the color occupied by **p**) are added to the index. Consequently, the index will continuously represent the voxel index for **p**. Fig. 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 percent. 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 offset compression [4], [20], although that could be added orthogonally. Storing the number of contained sub-nodes in each node, rather than a color offset per pointer, does reduce the performance of the color lookup (see Section 5.4). Thus, the better choice depends on whether a roughly halved geometry DAG size is more important than optimal performance in the color lookup pass.

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 **n**'s siblings are added to the index as we traverse the tree. Note that, whether our method for decoupling geometry and colors or those of previous work are used, the colors of internal nodes will be interleaved with the colors of nodes at lower levels. This has the unfortunate effect that, unlike traditional 2D mip-map hierarchies, the colors of *internal* nodes at the same level will be scattered in memory which may lead to poor memory access patterns. This problem is not further explored in this paper, but we consider it an interesting topic for future work.

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 conventional image-compression algorithms to it. Specifically, we either generate the image using a Morton or a Hilbert mapping. The algorithms presented in this section enable, for the first time, the compression of voxel attributes using conventional hardware accelerated texture compression, and off-line image compression algorithms.

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.

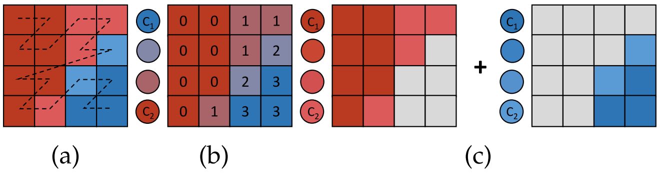


Fig. 4. a) The original block of colors. b) With BC1, sharp contrasts are blurred. Potentially bleeding across separate surfaces. c) Formats with partitioning can have separate endpoints for separate surfaces.

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 Fig. 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.

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 Fig. 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.

4.2 Variable Bitrate Block Encoding - Fixed Weight Size

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. [4] 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 formats that will be described in this and the next section similarly lie in this middle ground, but achieve much better compression at similar qualities, are simpler to implement and scale to extremely high compression rates.

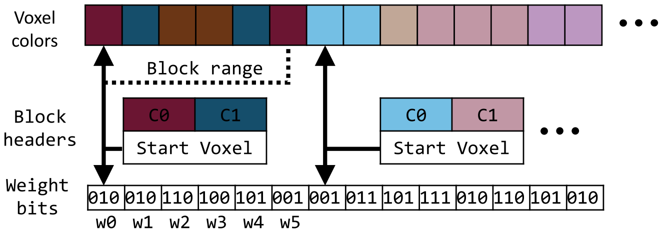


Fig. 5. In our suggested format, with a fixed bitwidth per weight, the array of voxel colors are divided into blocks of varying length that can be described with two endpoint colors, c_0 and c_1 and a weight per color that interpolates between these.

4.2.1 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. In the first version of our method, the weights are all of constant bitwidth, W, (usually 2-4 bits) and so are directly indexable. Our data structure is illustrated in Fig. 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

$$c_j = (w_j/W)c_1 + ((W - w_j)/W)c_0. \tag{1}$$

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 reciprocal of 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.

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)
  $(c_0, c_1, w)$ ←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← second element in blocks;
    while (not at end of list) do
      leftmse ←Eval(block.left.start, block.end);
      rightmse ←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←block.right.right;
  while (any block was merged);
```

4.3 Variable Bitrate Block Encoding - Variable Weight Size

In the algorithm described in the previous section, the number of bits per interpolation weight is fixed. This means that several blocks may be described using more bits per weight than is actually required to stay within the requested error threshold. In this section, we show how even better results can be obtained by carefully choosing the number of bits per weight on a per-block basis.

4.3.1 Data Structure and Decompression

When each block can have a different number of bits per weight, the weight required to decompress a specific color can no longer be obtained by directly indexing a global array of weights. Consequently, we must store a pointer into the weight array, as well as the number of bits per weight, in each block header. This additional information increases the size of block headers, and can completely overtake the reduced size of the weight array. We approach this problem by first dividing all colors into large *macro blocks* of constant size. These macro blocks are compressed independently, and the array of macro-block headers can be directly indexed by the color index. Each macro-block header contains a pointer to the first weight index and an index to the first block header for this macro block (see Fig. 6). This reduces the number of bits required for the weight and voxel indices in the block headers, since we can now simply store smaller offsets relative to the macro block.

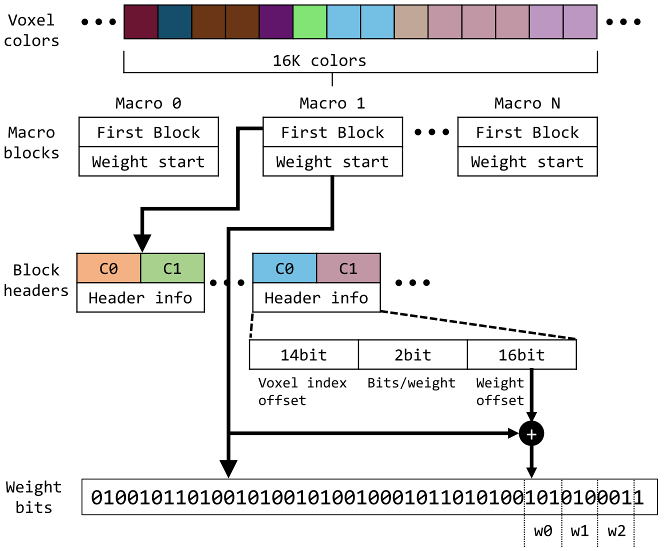


Fig. 6. With variable weight sizes, each block specifies the number of (0-4) bits per weight. For each sequence of $16K$ voxel colors, a macro block points to the first block and weight bit. Each block then provides the offsets to block and weights from its macro block.

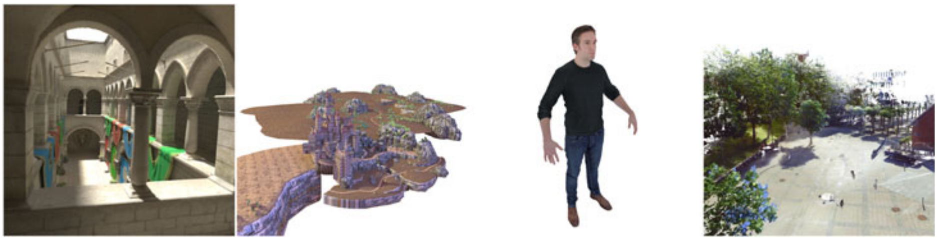
In this paper, we let a macro block be $16K$ colors and restrict the maximum bitwidth to four bits per weight. If we also store the two endpoint colors as RGB565, we are able to store the block headers using only 64 bits, i.e., at no extra cost compared to the algorithm in the previous section.

This is achieved in the following way: The two endpoint colors require 16 bits each. The voxel-index offset, bits/weight, and weight-index offset need to fit in the remaining 32 bits. Since every header now only needs to address up to $16K$ colors we can store the voxel index offset in 14 bits. Also, by restricting the maximum number of bits per weight to four, we only need to be able to address up to $64K$ individual weight bits, i.e., we need 16 bits for the weight-index offset. The remaining 2 bits will indicate whether the block has 1,2,3 or 4 bits per weight. For the zero bits per weight case, we realize that we will never utilize the maximum value of the 16 bit weight-index offset, since we have reserved precision for the case where every weight has four bits and, thus, the maximum weight-index offset will never exceed $64K - 3$. Accordingly, we can use one of those values as a sentinel since for the zero bit case we do not need to address any weights.

The only overhead now is the macro blocks, which require two 64 bit words each. This results in 1/128 extra bits per color and can, in all realistic cases, be considered negligible.

The decompression is similar to that of the algorithm in the previous section. We first do a direct lookup to find which macro block the voxel index belong to. We extract the first block header index for that and the next macro block, giving us a range of blocks in which to search for the color. We locate the block, by a binary search, and simply use the weight index offset in the macro block and the weight-index offset in the header to calculate the final weight index. An added benefit of this is that, since we first do the direct lookup of the macro blocks, we have effectively reduced the number of steps in the binary search, which improves lookup performance.

TABLE 1
The Scenes Used in the Evaluation of Our Algorithm



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

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.

4.3.2 Compression

We first build a tree of potential compressed blocks using different bitwidths for weights, and then find a cut through this tree containing the set of blocks that minimizes the required size. We build the tree bottom up by first compressing the colors, as in the algorithm in the previous section, using zero bits per weight. This set of compressed blocks make up the leaf level of our tree. To calculate the parents we then use the compressed set of blocks, and try to compress them further, this time using one more bit per weight. We repeat this process until we have reached the maximum number of bits per weight. This way, the color range of a parent will equal the combined ranges of its children.

Algorithm 2. Compress Blocks Using Variable Weight Bits per Block

```
input: $e_t$ - the error threshold
       oldblocks a linked list of blocks, initially one per color
#Compute the block tree
for (min bitrate to max bitrate) do
  blocks ← Compress ($e_t$, oldblocks, bitrate);
  AddToTree (blocks);
  oldblocks ← blocks;
# Do a tree cut to find least cost nodes
for (each level above leaves, bottom up) do
  forall nodes in level do
    if (my_cost $<$ $\sum$ children_cost) then
      RemoveChildren (node);
    else
      if (root_node) then
        MakeChildrenRoots (node);
      else
        ConnectChildrenToParent (node);
      Remove (node);
#Push to final solution
forall (remaining nodes) do
  PushToSolution (node);
```

To perform the cut, we process the nodes one level at a time in a bottom up order. For a given node, if the combined memory cost of its children is less than that of the node itself, we remove that node and connect the children to the parent of that node. Otherwise we remove the children. When we have processed all nodes we end up with a set of blocks which is our final solution. The procedure for the compression is detailed in Algorithm 2.

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 8 GB of data (500M points) and has been voxelized without any surface reconstruction.

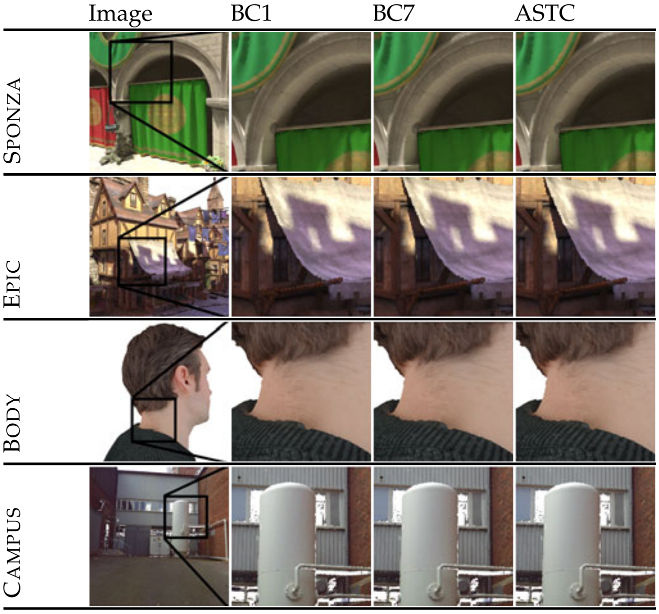
All scenes have been converted to a geometry DAG as described by Kämpe et al. [3] with voxel-color connections inserted as described in Section 3. Once the one-dimensional array of colors has been obtained, we compress that using a number of different algorithms listed below. We evaluate the quality of the compressed data first globally by calculating the *Global Root Mean Square Error* (GRMSE) 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 GRMSE provides some notion of the quality of the compressed data as a whole, but, being the average of the error of *all* voxel colors, might hide local artifacts that affect only a few voxels.

To mitigate this problem, we also chose a number of viewpoints in each scene and render an image for each. For these images, we calculate the *Root Mean Squared Error* (RMSE) 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, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2017.2741480, or on the authors homepage.[1]

[1] http://www.cse.chalmers.se/~dolonius/dolonius2017tvcg/supplementary

TABLE 2
Comparison of Hardware Texture Formats

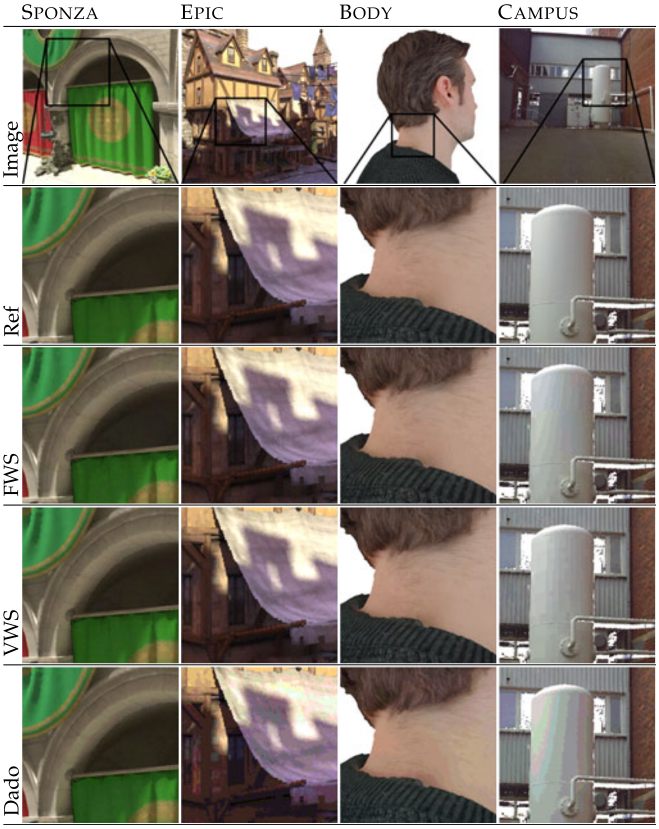
		SPONZA	EPIC	BODY	CAMPUS
BC1	*Compression*	16.7%	16.7%	16.7%	16.7%
	Glob. RMSE	2.15	2.48	2.4	4.29
	RMSE	3.27	4.73	1.35	2.03
	MS-SSIM	0.953	0.974	0.996	0.942
BC7	*Compression*	33.3%	33.3%	33.3%	33.3%
	Glob. RMSE	0.715	0.863	0.705	1.81
	RMSE	1.06	1.32	0.437	0.806
	MS-SSIM	0.992	0.997	1.0	0.983
ASTC	*Compression*	33.3%	33.3%	33.3%	33.3%
	Glob. RMSE	0.795	0.915	0.69	2.0
	RMSE	1.08	1.31	0.412	0.842
	MS-SSIM	0.992	0.997	1.0	0.983



To inspect the scene, we have implemented a real-time raytracer in CUDA that 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, 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.

TABLE 3
Quality Comparison for Methods Selected in Table 4 at Approximately Equal Compression Ratio



- *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. [4].
- *JPG, JPG2000* Image Magick.
- *PNG* pngout and pngquant [24], [25].

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

TABLE 4
Comparison of Variable Bitrate Block-Encoding Formats

		Sponza				Epic				Body				Campus			
		Comp.	GRMSE	RMSE	MSSIM	Comp.	GRMSE	RMSE	MSSIM	Comp.	GRMSE	RMSE	MSSIM	Comp.	GRMSE	RMSE	MSSIM
FWS	e_t :6.38	18.8%	1.12	1.42	0.976	20.2%	1.81	1.34	0.994	17.2%	1.99	1.18	0.995	44.2%	1.94	1.29	0.953
	e_t :9.56	15.5%	1.63	2.12	0.951	15.3%	2.58	1.95	0.988	13.3%	2.87	1.8	0.99	29.0%	2.62	1.91	0.915
	e_t :12.8	14.2%	2.1	2.9	0.914	13.9%	3.3	2.56	0.982	12.7%	3.7	2.35	0.985	23.2%	3.27	2.65	0.827
	e_t :25.5	13.1%	3.35	5.46	0.82	12.7%	5.93	5.23	0.943	12.5%	5.85	4.09	0.951	15.4%	5.69	5.04	0.633
VWS	e_t :6.38	11.0%	1.26	1.59	0.966	15.9%	1.84	1.42	0.993	14.2%	1.93	1.21	0.993	37.1%	2.07	1.46	0.925
	e_t :9.56	7.87%	1.83	2.35	0.926	11.1%	2.62	2.09	0.984	11.0%	2.84	1.87	0.983	23.5%	2.85	2.23	0.811
	e_t :12.8	6.15%	2.24	3.09	0.872	9.03%	3.35	2.82	0.972	9.54%	3.67	2.49	0.971	17.9%	3.56	2.83	0.711
	e_t :25.5	3.78%	3.64	5.52	0.747	5.17%	5.82	6.01	0.907	6.15%	6.78	4.52	0.926	9.35%	6.13	4.95	0.519
Dado	colors: all	37.2%	0.0	0.0	1.0	89.9%	0.0	0.0	1.0	65.2%	0.0	0.0	1.0	103.0%	0.0	0.0	1.0
	16K	21.2%	0.977	1.1	0.98	31.9%	1.69	1.17	0.994	35.0%	0.856	0.527	0.998	39.3%	2.06	1.4	0.939
	4K	17.1%	1.75	1.81	0.959	27.9%	2.9	1.95	0.986	31.7%	1.5	0.797	0.997	31.7%	3.52	2.35	0.86
	2K	15.2%	2.23	2.33	0.939	25.6%	3.82	2.46	0.98	30.8%	1.91	0.96	0.995	29.1%	4.54	3.08	0.82
	256	11.2%	5.0	5.62	0.799	18.8%	8.24	5.48	0.923	24.1%	4.25	2.2	0.977	21.4%	11.1	8.01	0.606

Our and Dado et al.'s formats are compared at varying quality settings. The highlighted settings are chosen as being of approximately equal compression ratio as the FWS result with error-threshold 9.56.

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 percent. In Table 9, 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

In Table 4, we compare our novel compression schemes, described in Sections 4.2 and 4.3, with our own implementation of the algorithm suggested by Dado et al. [4]. For our FWS format, the results are similar between the first three scenes. The compressed data is 17-20 percent of the original with virtually no perceptible error, 13-15 percent with very high quality, and we can push it down towards 12 percent (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.

The quality results for the variable-weight compression scheme (described in Section 4.3) are similar to those of the fixed weight compression scheme for all scenes. Not unexpectedly we generally have a slightly higher GRMSE for variable bit weights, since we can make a more aggressive approximation by reducing the number of interpolation points while respecting the error threshold, ignoring the degradation in average quality. Conversely, the FWS algorithm can be seen as having a higher quality than requested in these cases. Meanwhile, with the VWS algorithm, we have further reduced the compressed size by around 16 percent for the high quality encodings and up to 71 percent for lower qualities. Unlike the fixed-weight format, which has a memory cost per voxel limited by its weights, the variable-weight format has virtually no lower bound on the degree of compression it can achieve. In Table 7, we can see that at the lowest quality for the Sponza scene, we end up with 0.9 bits per voxel for the colors.

In the Sponza scene the variable-weight format and Dado et al.'s format perform much better than in any of the other scenes. We have investigated this further to demonstrate some interesting properties of these formats.

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 percent 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, the fixed weight size version of 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. Similarly, the variable-weight version of our algorithm

TABLE 5
A Detailed Evaluation of the SPONZA Scene

		ORIGINAL		MODIFIED	
		Comp.	GRMSE	Comp.	GRMSE
FWS	e_t :6.38	18.8%	1.12	20.3%	1.58
	e_t :9.56	15.5%	1.63	16.0%	2.3
	e_t :12.8	14.2%	2.1	14.7%	2.91
	e_t :25.5	13.1%	3.35	13.1%	4.56
VWS	e_t :6.38	11.0%	1.26	15.8%	1.7
	e_t :9.56	7.87%	1.83	11.7%	2.43
	e_t :12.8	6.15%	2.24	9.26%	2.98
	e_t :25.5	3.78%	3.64	5.73%	5.47
Dado	colors: all	37.2%	0.0	65.3%	0.0
	16K	21.2%	0.977	31.7%	1.29
	4K	17.1%	1.75	26.2%	2.35
	2K	15.2%	2.23	23.7%	2.99
	256	11.2%	5.0	17.0%	7.16

ORIGINAL is the original scene and in MODIFIED we have removed the redundant geometry.

can store the entire block with a single, zero bits per weight, block header. 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. As expected, the results for the fixed-weight format remain relatively unchanged, while the variable-weight and Dado et al.'s format have degraded. On all scenes, our variable-weight format is clearly the better choice, offering very high quality at compression ratios that cannot be reached with any of the other algorithms.

Finally, in Table 6 we compare the compression ratio and quality obtained when compressing only the leaf node colors (as in Table 4) to compressing the colors of all nodes. The colors of internal nodes are obtained by averaging the colors of their immediate sub-nodes, which means that many of the new colors will be very similar to the subsequent colors in the array and end up in the same compressed block when using our FWS or VWS methods. Consequently, for our methods, the compression ratio is generally better when considering all colors and the quality is not affected. With the algorithm of Dado et al. [4], adding the colors of all internal nodes means that more disparate colors must be quantized to a global palette of fixed size, and thus, while the compression ratio is slightly reduced, the quality also degrades slightly.

TABLE 6
Results of Compressing the Colors of all Nodes to Only Leaf Nodes for Error Thresholds Selected in Table 4

			Colors	Comp.	GRMSE	RMSE	MSSSIM
SPONZA	FWS	e_t: 9.56	Leaf	15.5%	1.63	2.12	0.951
			All	15.0%	1.62	2.15	0.950
	VWS	e_t: 6.38	Leaf	11.0%	1.26	1.59	0.966
			All	10.1%	1.26	1.60	0.964
	Dado	colors: 2K	Leaf	15.2%	2.23	2.33	0.939
			All	14.9%	2.34	2.44	0.932
EPIC	FWS	e_t: 9.56	Leaf	15.3%	2.58	1.95	0.988
			All	14.7%	2.57	1.96	0.988
	VWS	e_t: 6.38	Leaf	15.9%	1.84	1.42	0.993
			All	14.3%	1.82	1.42	0.993
	Dado	colors: 256	Leaf	18.8%	8.24	5.48	0.923
			All	18.3%	8.48	5.53	0.924
BODY	FWS	e_t: 9.56	Leaf	13.3%	2.87	1.8	0.99
			All	13.1%	2.86	1.8	0.99
	VWS	e_t: 6.38	Leaf	14.2%	1.93	1.21	0.993
			All	13.4%	1.93	1.22	0.993
	Dado	colors: 256	Leaf	24.1%	4.25	2.2	0.977
			All	24.1%	4.39	2.1	0.976
CAMPUS	FWS	e_t: 9.56	Leaf	29.0%	2.62	1.91	0.915
			All	24.4%	2.58	1.87	0.920
	VWS	e_t: 9.56	Leaf	23.5%	2.85	2.23	0.811
			All	19.0%	2.79	2.25	0.812
	Dado	colors: 2K	Leaf	29.1%	4.54	3.08	0.82
			All	28.2%	4.69	3.03	0.821

The image shows the four highest levels of detail for a view of the EPIC scene.

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 8. Table 9 shows what type of artifacts are introduced and many more examples are available in the supplementary material, available online. JPG and JPG2K can both produce fairly high quality results at compressed sizes from 15-20 percent. 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 GRMSE numbers in Table 8, but in the second row of PNG images in Table 9, 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

TABLE 7
Bits per Voxel Color for Each Scene and Quality Setting

Bits per voxel		SPONZA	EPIC	BODY	CAMPUS
FWS	e_t :6.38	4.5	4.85	4.13	10.6
	e_t :9.56	3.72	3.66	3.18	6.97
	e_t :12.8	3.42	3.33	3.04	5.56
	e_t :25.5	3.13	3.04	3.0	3.7
VWS	e_t :6.38	2.64	3.82	3.4	8.91
	e_t :9.56	1.89	2.65	2.65	5.64
	e_t :12.8	1.48	2.17	2.29	4.29
	e_t :25.5	0.906	1.24	1.48	2.24
Dado	colors: all	8.94	21.6	15.6	24.7
	16K	5.08	7.66	8.4	9.44
	4K	4.09	6.69	7.61	7.61
	2K	3.66	6.14	7.38	6.98
	256	2.69	4.52	5.77	5.14

Cost of geometry not included.

TABLE 8
Comparing the Quality of Voxel Data Compressed Using Conventional Off-Line Image-Compression Formats

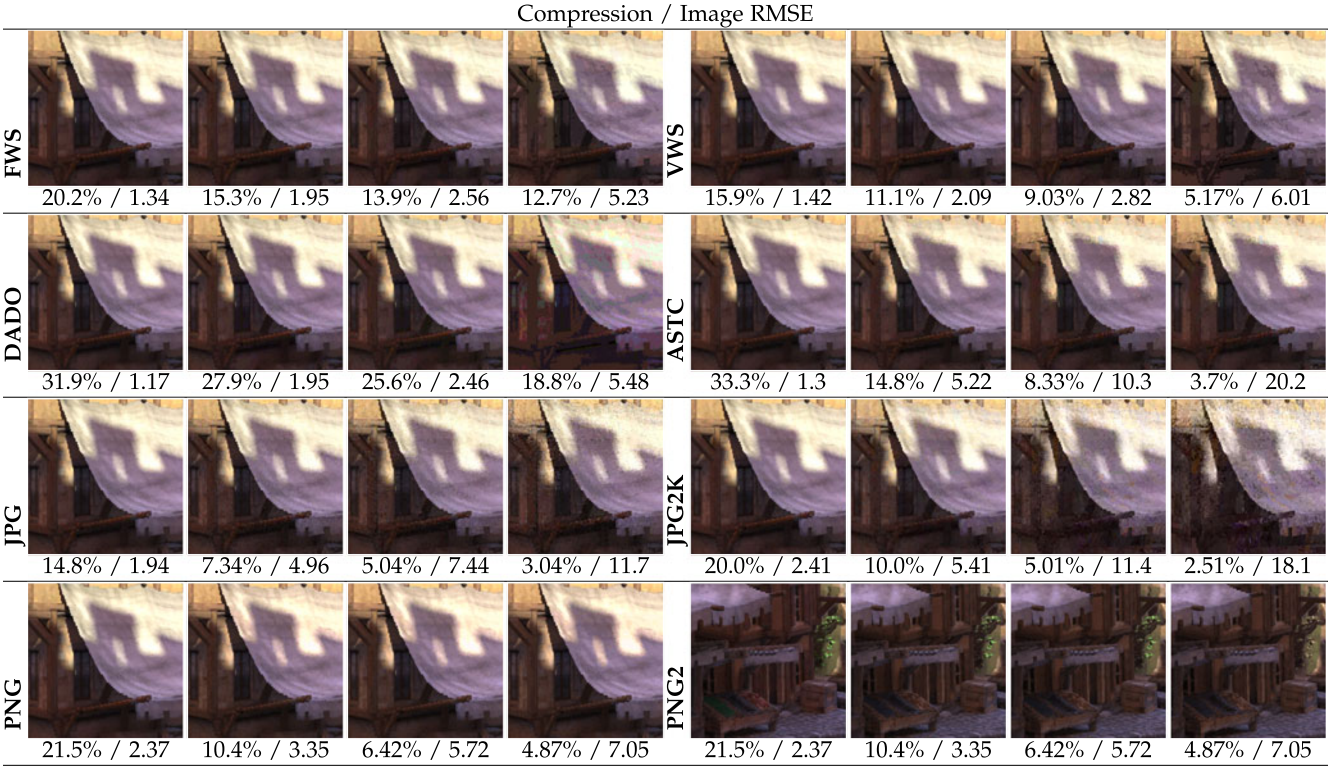
		SPONZA				EPIC				BODY				CAMPUS			
quality setting		Comp.	GRMSE	RMSE	MSSSIM	Comp.	GRMSE	RMSE	MSSSIM	Comp.	GRMSE	RMSE	MSSSIM	Comp.	GRMSE	RMSE	MSSSIM
JPG	95	11.7%	1.75	1.95	0.956	14.8%	2.13	1.94	0.987	14.9%	2.02	1.35	0.996	21.0%	2.91	1.74	0.924
	85	6.42%	3.22	4.6	0.869	7.34%	3.76	4.96	0.945	7.49%	4.04	2.73	0.988	10.4%	5.59	3.37	0.847
	75	4.71%	4.34	6.52	0.809	5.04%	4.73	7.44	0.91	5.09%	5.07	3.33	0.983	7.07%	7.3	4.4	0.801
	50	3.08%	6.44	9.47	0.727	3.04%	6.1	11.7	0.853	3.05%	6.14	3.95	0.975	4.08%	9.92	5.7	0.718
JPG2K	x5	20.1%	0.929	1.12	0.981	20.0%	1.71	2.41	0.982	20.1%	1.33	0.708	0.997	20.4%	2.88	1.95	0.893
	x10	10.1%	2.06	2.44	0.92	10.0%	3.19	5.41	0.938	10.0%	2.77	1.44	0.991	10.3%	5.07	3.43	0.75
	x20	5.09%	4.25	5.67	0.777	5.01%	5.31	11.4	0.823	5.01%	4.86	2.69	0.976	5.17%	7.85	5.22	0.607
	x40	2.56%	8.27	11.1	0.593	2.51%	7.75	18.1	0.692	2.51%	6.96	4.01	0.956	2.58%	11.0	7.55	0.491
PNG	lossless	29.8%	0.0	0.0	1.0	52.7%	0.0	0.0	1.0	50.2%	0.0	0.0	1.0	89.0%	0.0	0.0	1.0
	100	12.4%	0.956	2.14	0.967	21.5%	1.9	2.37	0.985	25.6%	0.985	0.73	0.998	21.7%	3.26	1.77	0.925
	70	4.9%	2.95	5.69	0.788	10.4%	3.86	3.35	0.97	7.57%	4.2	2.35	0.978	18.6%	3.78	2.11	0.892
	30	3.08%	5.14	8.47	0.641	6.42%	6.34	5.72	0.93	4.82%	6.66	4.06	0.932	11.6%	6.26	3.72	0.736
	10	2.27%	6.68	11.3	0.515	4.87%	8.19	7.05	0.897	3.56%	8.22	5.24	0.889	8.8%	8.1	4.59	0.639

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 percent 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.

TABLE 9
Change in Image Quality (RMSE) with Compression Rate per Method



Lookup. In the table below, we present the time taken to render the images shown in Table 3 at 1024 x 1024 resolution, on a GTX 1080 graphics card, using our methods and that of Dado et al., both when using per-pointer and per-node offsets to calculate the voxel-color index.

		SPONZA	EPIC	BODY	CAMPUS
Raytracing		2.1 ms	3.4 ms	4.8 ms	2.7 ms
Color lookup (only leaf nodes)					
FWS	per-pointer	0.4 ms	0.4 ms	0.5 ms	0.5 ms
	per-node	0.6 ms	0.7 ms	0.7 ms	0.9 ms
VWS	per-pointer	0.3 ms	0.4 ms	0.5 ms	0.4 ms
	per-node	0.5 ms	0.6 ms	0.7 ms	0.7 ms
Dado	per-pointer	0.4 ms	0.4 ms	0.6 ms	0.5 ms
	per-node	0.6 ms	0.7 ms	0.7 ms	0.8 ms
Color lookup (all nodes)					
FWS	per-pointer	0.4 ms	0.4 ms	0.5 ms	0.5 ms
	per-node	0.7 ms	0.8 ms	0.7 ms	0.9 ms
VWS	per-pointer	0.3 ms	0.4 ms	0.5 ms	0.4 ms
	per-node	0.6 ms	0.7 ms	0.7 ms	0.8 ms
Dado	per-pointer	0.4 ms	0.4 ms	0.6 ms	0.5 ms
	per-node	0.7 ms	0.8 ms	0.8 ms	0.9 ms
Geometry DAG size					
	per-pointer	15 MB	336 MB	168 MB	184 MB
	per-node	9 MB	189 MB	96 MB	118 MB

As expected, the lookup performance is somewhat faster when using per-pointer offsets (1.2x - 1.92x), but the resulting DAG-sizes are also significantly larger (1.5x - 1.8x). Although using variable bit-widths for weights requires one more level of indirection in decompressing a color, the introduction of macro blocks reduces the range of blocks in which a binary search is required, so VWS performance is on par with, or better than, FWS. We also present the times taken to look up the leaf colors when the colors of all nodes are stored. There is only a very slight increase in these times, since leaf colors for nearby voxels are still usually close to each other despite having interleaved the colors of internal nodes.

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 usually achieves better than 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.

Acknowledgments

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 [26].

References

[1] D. Dolonius, E. Sintorn, V. Kämpe, and U. Assarsson, “Compressing color data for voxelized surface geometry,” in *Proc. 21st ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, 2017, Art. no. 13. [Online]. Available: http://www.cse.chalmers.se/dolonius/dolonius2017i3d.pdf

[2] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, “Interactive indirect illumination using voxel cone tracing,” *Comput. Graph. Forum*, vol. 30, no. 7, pp. 1921–1930, Sep. 2011.

[3] V. Kämpe, E. Sintorn, and U. Assarsson, “High resolution sparse voxel DAGs,” *ACM Trans. Graph.*, vol. 32, no. 4, 2013, Art. no. 101.

[4] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann, “Geometry and attribute compression for voxel scenes,” *Comput. Graph. Forum*, vol. 35, no. 2, pp. 397–407, May 2016.

[5] B. R. Williams, “Moxel DAGs: Connecting Material Information to High Resolution Sparse Voxel DAGs,” Master’s thesis, Comput. Sci. Dept., California Polytechnic State Univ., San Luis Obispo, CA, USA, 2015.

[6] S. M. Rubin and T. Whitted, “A 3-dimensional representation for fast rendering of complex scenes,” *SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 110–116, Jul. 1980.

[7] C. L. Jackins and S. L. Tanimoto, “Oct-trees and their use in representing three-dimensional objects,” *Comput. Graph. Image Process.*, vol. 14, no. 3, pp. 249–270, 1980.

[8] D. Meagher, “Geometric modeling using octree encoding,” *Comput. Graph. Image Process.*, vol. 19, no. 2, pp. 129–147, 1982.

[9] K. Museth, “VDB: High-resolution sparse volumes with dynamic topology,” *ACM Trans. Graph.*, vol. 32, no. 3, pp. 27:1–27:22, Jul. 2013.

[10] J. Elseberg, D. Borrmann, and A. Nüchter, “One billion points in the cloud–an octree for efficient processing of 3D laser scans,” *J. Photogrammetry Remote Sens.*, vol. 76, pp. 76–88, 2013.

[11] E. Gobbetti and F. Marton, “Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms,” *ACM Trans. Graph.*, vol. 24, no. 3, pp. 878–885, Jul. 2005.

[12] E. Gobbetti, F. Marton, and A. J. Iglesias Guitián, “A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets,” *Visual Comput.*, vol. 24, no. 7, pp. 797–806, 2008.

[13] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, “GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering,” in *Proc. ACM Symp. Interactive 3D Graph. Games*, Feb. 2009, pp. 15–22.

[14] S. Laine and T. Karras, “Efficient sparse voxel octrees,” *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 8, pp. 1048–1059, Aug. 2011.

[15] R. E. Webber and M. B. Dillencourt, “Compressing quadtrees via common subtree merging,” *Pattern Recognit. Lett.*, vol. 9, no. 3, pp. 193–200, 1989.

[16] M. S. Parsons, “Generating lines using quadgraph patterns,” *Comput. Graph. Forum*, vol. 5, no. 1, pp. 33–39, Mar. 1986.

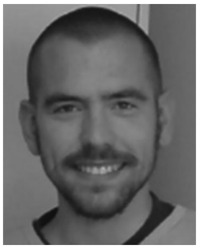
[17] E. Parker and T. Udeshi, “Exploiting self-similarity in geometry for voxel based solid modeling,” in *Proc. 8th ACM Symp. Solid Model. Appl.*, 2003, pp. 157–166.

[18] E. Sintorn, V. Kämpe, O. Olsson, and U. Assarsson, “Compact precomputed voxelized shadows,” *ACM Trans. Graph.*, vol. 33, no. 4, 2014, Art. no. 150.

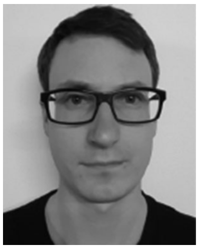
[19] V. Kämpe, E. Sintorn, and U. Assarsson, “Fast, memory-efficient construction of voxelized shadows,” in *Proc. ACM 19th Symp. Interactive 3D Graph. Games*, 2015, pp. 25–30.

[20] A. Jaspe Villanueva, F. Marton, and E. Gobbetti, “SSVDAGs: Symmetry-aware sparse voxel DAGs,” in *Proc. ACM 20th ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, Feb. 2016, pp. 7–14.
[21] S. Lefebvre and H. Hoppe, “Perfect spatial hashing,” *ACM Trans. Graph.*, vol. 25, no. 3, pp. 579–588, Jul. 2006.
[22] S. Guthe, M. Wand, J. Gonser, and W. Straßer, “Interactive rendering of large volume data sets,” in *Proc. IEEE Vis.*, 2002, pp. 53–60.
[23] M. Balsa Rodriguez, et al., “State-of-the-art in compressed GPU-based direct volume rendering,” *Comput. Graph. Forum*, vol. 33, no. 6, pp. 77–100, Sep. 2014.
[24] K. Silverman, Ken silverman’s utility page, 2016. [Online]. Available: http://advsys.net/ken/utils.htm
[25] K. Lesiński, pngquant, 2016. [Online]. Available: https://pngquant.org/
[26] E. Lindskog, J. Berglund, J. Vallhagen, and B. Johansson, “Visualization support for virtual redesign of manufacturing systems,” *Procedia CIRP*, vol. 7, pp. 419–424, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212827113002783

Dan Dolonius received the MSc degree in applied mathematics. He is working toward the PhD degree in computer graphics at Chalmers University of Technology. He has worked at Autodesk with their render engines and applications. His research interests include real-time rendering, compression, and GPU algorithms.



Erik Sintorn received the PhD degree from Chalmers University of Technology, in 2013, where he now is an assistant professor in the Computer Graphics Research Group, Department of Computer Science and Engineering. His research is focused on real-time shadows, transparency, and global illumination.



Viktor Kämpe recieved the MS degree in engineering physics and the PhD degree in computer science and engineering from Chalmers University of Technology, Sweden, in 2011 and 2016, respectively. His research interests include highly detailed voxel geometry and visibility, in both static and temporal varying scenes, and free viewpoint video.

Ulf Assarsson is a professor in computer graphics in the Department of Computer Science and Engineering, Chalmers University of Technology. His main research interests include real-time rendering, global illumination, many lights, GPU-Ray Tracing, and hard and soft shadows. He is co-author of the book *Real-Time Shadows*.