THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Sparse Voxel DAGs for Shadows and for Geometry with Colors

DAN DOLONIUS

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2018

Sparse Voxel DAGs for Shadows and for Geometry with Colors
DAN DOLONIUS

© DAN DOLONIUS, 2018

Technical Report report no 177L
ISSN 1652-876X
Department of computer Science and Engineering
Research group: Computer Graphics

Department of computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Phone: +46(0)32 772 1000

Contact information:
Dan Dolonius
Department of computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Phone: +46(0)31 772 16 78
Email: dolonius@chalmers.se
URL: http://www.cse.chalmers.se/~dolonius/

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2018

Sparse Voxel DAGs for Shadows and for Geometry with Colors

Dan Dolonius
Department of Computer Science and Engineering
Chalmers University of Technology

Thesis for the degree of Licenciate of Engineering
a Swedish degree between M.Sc. and Ph.D.

Abstract

Triangles are probably the most common format for shapes in computer graphics. Nevertheless, when high detail is desired, *Sparse Voxel Octrees* (SVO) and *Sparse Voxel Directed Acyclic Graphs* (DAG) can be considerably more memory efficient. One of the first practical use cases for DAGs was to use the structure to represent precomputed shadows. However, previous methods were very time consuming in building the DAG and did not support any other attributes than discretized geometry. Furthermore, when used for scene object representation, the DAGs lacked proper support for properties such as object colors. The focus on this thesis is to speed up the build times of the DAG and to allow other, important, attributes such as colors to be encoded.

This thesis is a collection of three papers where we in **Paper I** solve the problem with slow construction times while also further compressing the DAG, allowing much faster feedback to an artist making changes to a scene and also opening up the possibility to recompute the DAG in run time for slowly moving shadows.

If a unique color per voxel is desired, which uncompressed would require 3 **bytes** per voxel, we realize that the benefit from compressing the geometry (down to or even below one **bit** per voxel) is rendered practically useless. We thus need to find a way to compress the colors as well. In **Paper IIA**, we solve this issue by mapping the voxel colors to a texture, allowing for the use of conventional compression algorithms, as well as a novel format designed for real-time performance. In **Paper IIB**, we further significantly improve the compression.

Keywords: voxel, geometry, octree, directed acyclic graph, compression, colors, shadows

Acknowledgements

I want to thank Ulf Assarsson and Erik Sintorn for their faith in me, accepting me as a Ph.D student. I again want to specially thank them for their invaluable support, patience, and guidance. Thanks to my parents and family for being supportive of my decisions in life and raising me to become the person I am. Finally, thanks to my colleagues and seniors at Chalmers, providing a enjoyable environment. So far it has been a blast working here, and I hope the remaining years will be the same.

List of Appended Papers

This thesis is a summary of three papers. The third paper (**IIB**) is an extension to the second paper (**IIA**), but is included since the contributions are arguably enough to constitute a paper on its own.
References to the papers will be made with roman numerals.

Paper I - Viktor Kämpe, Erik Sintorn, Dan Dolonius, Ulf Assarsson,
Fast, Memory-Efficient Construction of Voxelized Shadows,
IEEE Transactions on Visualization and Computer Graphics,
(Volume: 22, Issue: 10, Oct. 1 2016, Pages: 2239 - 2248).

Paper IIA - Dan Dolonius, Erik Sintorn, Viktor Kämpe, Ulf Assarsson,
Compressing Color Data for Voxelized Surface Geometry (original),
I3D '17 Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Article No. 13 (Best paper award).

Paper IIB - Dan Dolonius, Erik Sintorn, Viktor Kämpe, Ulf Assarsson,
Compressing Color Data for Voxelized Surface Geometry (extension),
IEEE Transactions on Visualization and Computer Graphics,
(Volume: PP, Issue: 99, Aug. 18 2017, Pages: 1 - 1).

Table of Contents

I Summary

1 Introduction **1**
- 1.1 Scene representation . . . 1
  - 1.1.1 Surface geometry . . . 2
  - 1.1.2 Surface properties . . . 4
  - 1.1.3 Light visibility . . . 6
- 1.2 Compressing voxelized geometry information . . . 7
- 1.3 Compressing voxelized light-visibility information . . . 8
- 1.4 Compressing voxel-property information . . . 9

2 Summary of Included Papers **12**
- 2.1 **Paper I** - Fast, Memory-Efficient Construction of Voxelized Shadows . . . 12
- 2.2 **Paper IIA** - Compressing Color Data for Voxelized Surface Geometry (Original) . . . 14
- 2.3 **Paper IIB** - Compressing Color Data for Voxelized Surface Geometry (Extension) . . . 17

3 Discussion and Future Work **19**

Bibliography **20**

II Appended Papers

Paper I - Fast, Memory-Efficient Construction of Voxelized Shadows

Paper IIA - Compressing Color Data for Voxelized Surface Geometry (original)

Paper IIB - Compressing Color Data for Voxelized Surface Geometry (extension)

Part I

Summary

1 Introduction

Computer graphics refers to when virtual scenes are visualized using computers. This is achieved by storing geometry with some properties, e.g. a green ball, and then simulating how the light interacts with the scene. To render realistic images, the objects in the scene need to be very detailed and complex light interactions have to be simulated. However, when the objects have more details, they will consume more memory, and when more complex simulations have to be performed, more processing power is needed. This quickly becomes an issue, as computers only have a limited amount of memory and processing power, and we thus need to find ways to efficiently represent these objects, as well as inexpensive methods of rendering them. In this thesis, we aim to improve and extend an existing representation of such scenes in order for them to consume as little memory as possible while still being inexpensive to visualize.

1.1 Scene representation

The original geometrical representation of a scene depends on how it was created. For example, a scanned object is represented as a point cloud, while an artist-generated object is usually built from parametric surfaces such as *Non-uniform rational B-splines* (NURBS) or recursively from *Subdivision surfaces* (SubD). These representations are, however, not suitable for rendering, as point clouds requires a lot of memory and parametric and subdivision surfaces are inefficient to render due to their high degree of freedom. Thus, in order to render these scenes, they are first converted to simpler data structures that are cheap to render. Fast render times are crucial in *real-time* applications, such as games, where each frame has to be rendered in less than 33 or 16 milliseconds. It is also important in *offline* rendering to have a data structure that can be efficiently ray traced, as iteration times decreases for artists since they have to wait less to see updated results from having changed the scene. Further, in final production, rendering times are usually long and consume a significant amount of electricity, which is yet another incentive for efficient offline rendering.

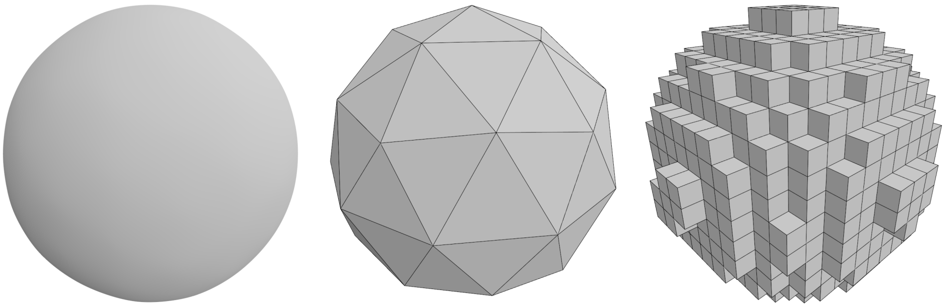


Figure 1: Left: Original. Middle: Triangulated. Right: Voxelized.

1.1.1 Surface geometry

In most real-time rendering algorithms, the scene representation is first decomposed into a list of *triangles* (see Figure 1). A collection of triangles connected by their common edges or corners is called a *triangle mesh*, and a collection of independent triangles is called a *triangle soup*. Triangles typically consumes 36 bytes when stored in floating-point precision and are one of the simplest possible representations of a surface element. Modern *Graphics Processing Units* (GPUs) are specifically built to be able to render huge amounts of triangles in parallel. In offline rendering algorithms, which are usually based on ray tracing, the scene is usually decomposed in the same way, and then an *acceleration structure* is built on top of the triangles to allow for efficient hierarchical intersection tests between a ray and the scene.

There are, however, two potential problems with representing large scenes using triangle meshes. To illustrate, consider a scene where the camera is inside a building looking out from a window. The scene is vast and contains a huge amount of triangles, and we do not want the entire scene in GPU memory when only a fraction is visible, as the entire mesh might be too large to fit in memory. The second issue is that the triangle resolution might not match the sampling frequency. To illustrate this, consider the single pixel highlighted in Figure 2. An entire house, represented by thousands of triangles, projects onto that pixel, and only one of those triangles will be sampled to represent the color for the pixel. To circumvent these problems, some sort of *Level Of Detail* (LOD) is required, i.e., objects far away from the camera are represented by fewer triangles than nearby objects. This adds a layer of complexity to the production of such assets, since the artist must create different versions of the same object with different triangle counts or use some kind of polygon-reduction algorithm which requires a lot of tweaking to get acceptable results.

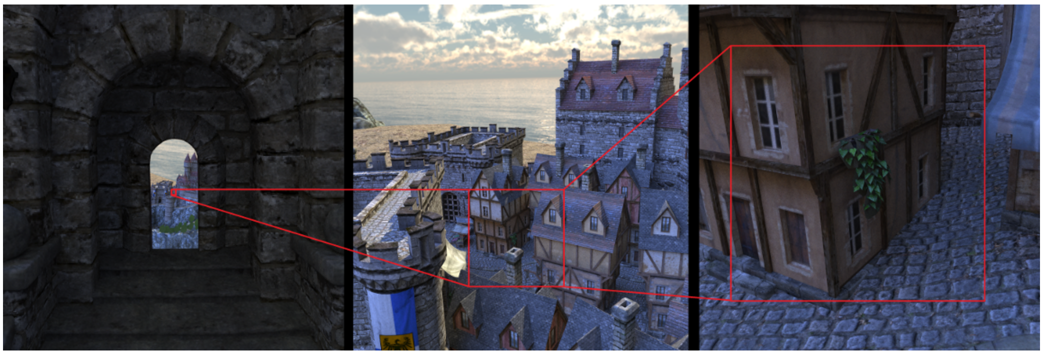


Figure 2: An example of how many small triangles can intersect one pixel.

An alternative to storing scenes as a triangle mesh is to store it as a collection of *voxels* (see Figure 1). Voxel is short for *volume element* and is the 3D analog to a *pixel* (picture element). To represent a surface using voxels, binary values can be stored in a 3D grid, indicating whether a voxel contains geometry or not. This representation, however, is costly and does not solve any of the problems with triangles. Fortunately, voxels are easy to store hierarchically by recursively splitting the voxels containing geometry along the three coordinate axes, to form eight smaller voxels. The benefit of storing voxels this way is that it consumes less memory and has an inherent LOD. It is therefore relatively simple to match the voxel resolution with the sample frequency and can thus be much more memory efficient than triangles for fine details, but can still be very expensive if implemented naively.

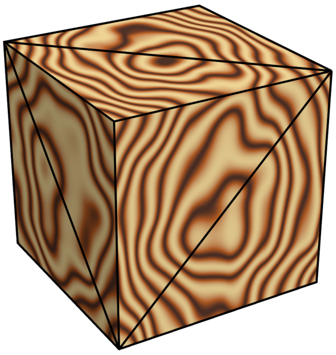


Figure 3: A textured cube composed of 12 triangles.

1.1.2 Surface properties

Just rendering geometry has a limited use unless the surface has some additional properties, as for example a material. A material specifies how light interacts with the surface, e.g., how it scatters (rough / shiny), if and how it passes through a material (translucent / opaque), and which wavelengths are reflected (color), etc. To store this information in high detail coupled with the geometry is not feasible as, e.g., a simple cube, which can be represented using only 12 triangles, would require several million triangles if we want to color it like a wooden block as in Figure 3. This is why conventional real-time and offline rendering algorithms decouple surface properties that need higher resolution than the geometry itself, and store them in *textures* instead. A texture is an N-dimensional lookup table, where each element is called a *texel*, or more intuitively, an image which can be wrapped around the object. An extra benefit is that LODs, just as for voxelized geometry, become straight forward as a hierarchy can be built by simply halving the resolution of the texture along its axes for each desired resolution. This is called a *mipmap hierarchy*.

The problem now lies with finding a mapping between the geometry and the texture. For triangles, this is generally achieved by projecting (or more intuitively, unfolding) the triangles onto the texture plane (see Figure 4) and for each vertex storing the coordinates of its projection. A simple analogue is a world map (texture) with latitude and longitude (coordinates). There are, however, some inherent problems with textures. As we can see in Figure 4, we have introduced discontinuities, in the texture space, along certain edges of the cube. This can introduce noticeable artifacts known as *seams* (where the texture has to be sewn together) if great care is not taken in where to place the seams in order to hide them. Further complicating the process is that the

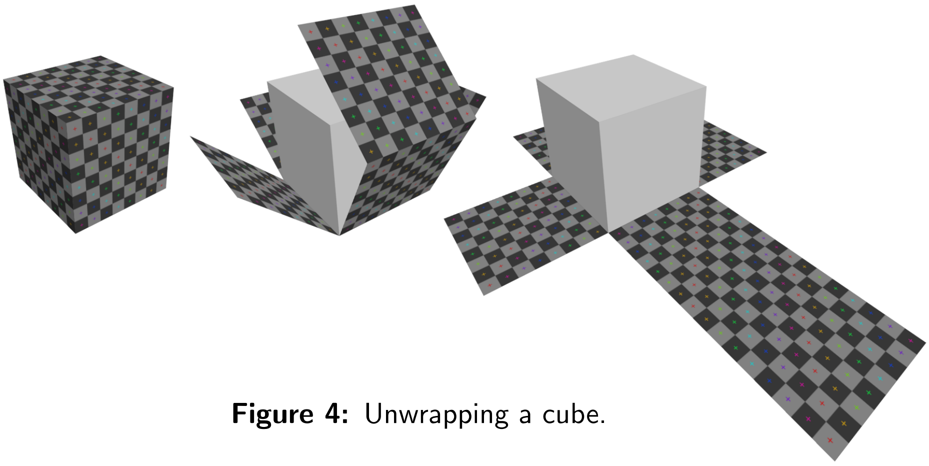


Figure 4: Unwrapping a cube.

texel density might vary over the surface depending on where the seams have been placed, similar to the distortions on world maps making some countries look larger than they actually are. While there exist algorithms to automate the unwrapping, as with LOD for geometry, a lot of manual tweaking has to be done, making this a time-consuming chore for many artists.

In computer graphics, *texture baking* is the procedure of transferring or precomputing properties from one model to another and storing them in a texture, which is possible by having decoupled the mesh from the surface properties. For example, normals from a high resolution mesh can be baked to a normal map to be used on a low resolution mesh or a light map can be baked by precomputing the light's effect on the surface. In the case of light maps, it is only possible to bake static and view-independent lighting, and it is therefore not possible to use tricks such as using the same texture on different parts of the geometry, as every point on the surface will be unique, which become very expensive at high resolutions.

When the scene is represented as voxels, each voxel can hold the corresponding surface properties. The downside is that it will be at least as expensive as a light map. Be that as it may, the benefits are that no unwrapping has to be done, and the sample density can be trivially homogeneous over the entire geometry.

1.1.3 Light visibility

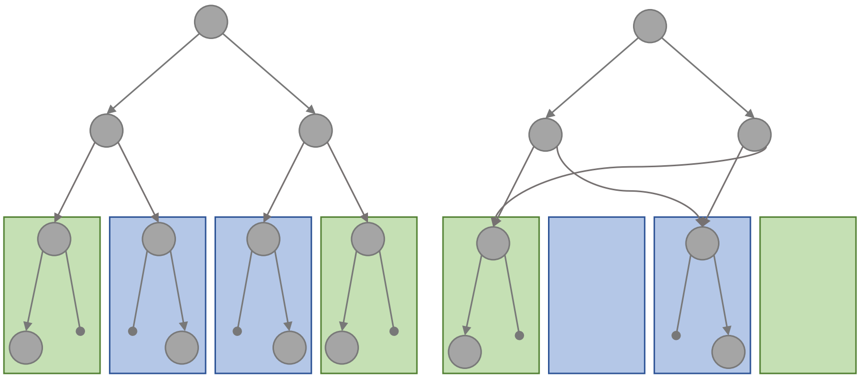
Another very important property of a point on a surface is whether it is visible to a light source, i.e. if it is in shadow, or not. This information is problematic to precompute for dynamic objects as they may enter or exit shadows, which is why this property, for such objects, generally is computed at run-time. On the other hand, for static lights and objects, it would be possible to precompute this information, although it might become too expensive at higher resolutions.

The most popular shadow algorithm in real-time applications is *shadow mapping*, where a 2D depth map, generated from the light source, is used to calculate the visibility for a certain point in space [19]. For dynamic shadow casters, a low-resolution map can be used that occupies little memory and can be calculated on the fly, with the obvious downside of having sub-optimal quality. For static shadow casters, a high-resolution map can be precomputed and compressed allowing higher quality at the expense of memory.

One of the drawbacks of standard shadow mapping is that the shadow map, unlike textures, can not be pre-filtered with, e.g., a mipmap. Instead, the shadow map has to be filtered in real-time (*Percentage Closer Filtering*) or an approximate pre-filtering method must be used (*Variance Shadow Maps*) [14, 4]. Another problem is that the shadow map will be sampled with different frequency for near-by and distant pixels, but the shadow map is of uniform resolution. One common solution to this problem is to use *Cascaded Shadow Maps* to handle shadows from far light sources on large open scenes by dividing the camera view frustum into several regions using a separate shadow map for each [5, 20]. Due to its simplicity and performance, it is one of the top contending algorithms for shadows in real time. It does, however, still suffer from undersampled geometric aliasing artifacts in distant regions.

Among other techniques there is also *shadow volumes*, where geometry is constructed for the regions occluded by a light source. The advantage with this method is that it generates pixel-perfect shadows, but the downside is that the geometry can become very complex and cause many triangles to be processed within the same pixel, as well as being more complicated for soft shadows. Recently, ray-tracing shadows in real-time has become a viable option on high-end hardware, but this is still very expensive when compared to shadow-mapping.

The final example is to store visibility information as voxels, which allows this information to be precomputed and easily queried. The advantage over light maps is that no *uv* coordinates are required, and hence no *uv* unwrapping is needed. A naive implementation would consume too much memory to be useful in practice, but Sintorn et al. managed to compress this information, making this an attractive alternative [16].



(a) Original SVO. **(b)** Transformed to DAG.

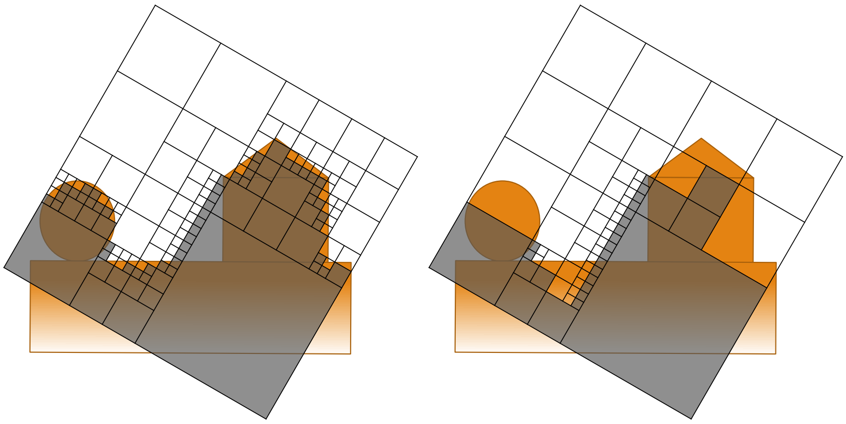
Figure 5: Illustration of SVO to dag transformation. Green and blue boxes represents identical subtrees. For brevity, the octree is represented as a binary tree.

1.2 Compressing voxelized geometry information

A voxel hierarchy constructed by splitting volumes containing geometry recursively in eight equally large subnodes, with pointers only to existing nodes, is commonly referred to as a *Sparse Voxel Octree* (SVO). It usually requires much less memory than a grid and provides an automatic LOD. While still potentially very expensive, *Efficient Sparse Voxel Octrees* (ESVO) by Laine et al., managed to significantly improve on the usability of SVOs [9]. The SVO data structure was later further optimized to the highly efficient *Sparse Voxel DAG* (DAG) by merging identical subtrees of an SVO [7] (see Figure 5). While providing extremely efficient compression (down to 0.08 bits per voxel), the voxelization is costly, and converting the voxels into a DAG also comes with a non-negligible cost.

In the following subsections, we will look closer into voxelized representations of shadows and color data.

1.3 Compressing voxelized light-visibility information



(a) Without relaxation.

(b) With relaxation.

Figure 6: Relaxation of boundaries, allowing larger homogeneous regions. © 2016 IEEE.

The drawbacks of shadow maps are high memory requirements for high resolutions and expensive filtering, and in order to alleviate these, Sintorn et al. suggested a drastically different approach, in which a static, high-resolution shadow map is encoded as a DAG [16]. They realized that if certain regions will never be queried, such as the inside of a closed object, one can consider the visibility of light in those regions to be undefined, and whether that region is lit or not can thus be chosen arbitrarily. When constructing the DAG, they resolve volumes containing undefined and lit regions to be fully lit and volumes containing undefined and shadowed regions to be fully shadowed, allowing formation of larger homogeneous regions and thus fewer nodes, as can be seen in Figure 6. In their paper, the authors manage to provide an extremely compact data structure for voxelized shadows.

A remaining problem is that the DAG is expensive to construct, as an SVO is first constructed and then reduced to a DAG by merging identical subtrees. In 2015, Kämpe et al. managed to accelerate construction of such DAGs by inserting identical nodes directly into the DAG, skipping a significant amount of work when merging the SVO [8]. Further, they used a different heuristic for undefined regions, allowing even better compression.

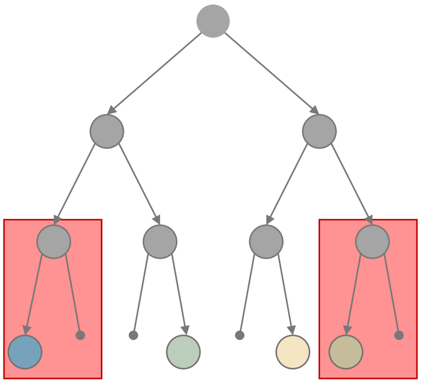


Figure 7: Red boxes represent previously identical subtrees which become different when color information is added to leaf nodes.

1.4 Compressing voxel-property information

One issue with the original DAG paper is that it does not support storing other attributes than the existence of geometry [7]. Coupling a property, such as color, with nodes in an SVO is trivial, as we in each node can simply attach a color index. Unfortunately, for a DAG such added information can change previously identical subgraphs to become unique, prohibiting the merging opportunities, see Figure 7(c). Williams et al., propose a coupling between a DAG and colors, but the suggested algorithm introduces an enormous amount of extra data, limiting the practical usefulness [18]. Another alternative is *Perfect Spatial Hashing*, which unfortunately, being hash based, will have a random lookup texture which significantly complicates compression [11]. In 2016, Dado et al. introduced a novel algorithm allowing light-weight decoupling between DAG and color data [3]. This is done by indexing each node in a depth-first order and then storing an index offset with each pointer in the DAG. Since the nodes remain identical, the DAG can be compressed as before and a color index can be retrieved by a running sum during traversal.

Compression of data can be divided in the two categories *lossless* and *lossy*, where the former is compression that does not alter the data in any way, and the later allows some deviation from the original data, generally in order to achieve better compression. There are however many cases where lossy compression is unacceptable. Common for both categories is that many compression formats exploit coherency in the data. For example, in an image, pixels are generally locally coherent, i.e., the neighboring pixels generally only slightly deviate from each other and can favourably be approximated by a function instead.

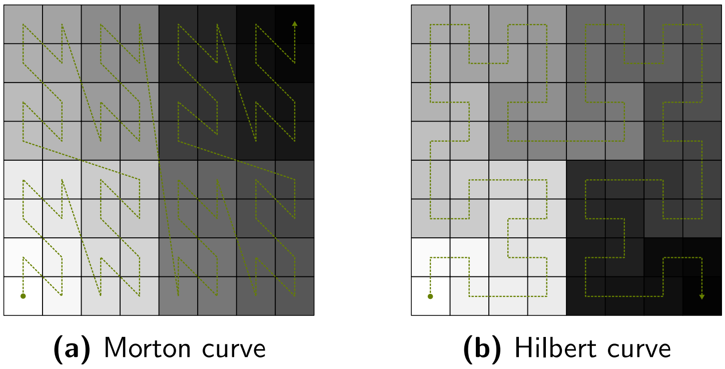

(a) Morton curve **(b)** Hilbert curve

Figure 8: Example of two common space-filling curves. The path is shown in green, and each cell is color coded to show how coherency is preserved.

While voxel colors certainly correspond to coherent 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. Just as with *uv* mapping, which maps the sparse surface to a dense, coherent texture, a mapping to a dense coherent domain is needed for voxels to have a shot at reasonable compression.

One possible option is to use a space-filling curve, which mathematically is defined as a curve whose range contains the entire N-dimensional hypercube [15]. The concept of space-filling curves was first proposed by Giuseppe Peano in 1890, which is also the reason that a space-filling curve on a 2D plane sometimes is called a peano curve. Two common curves, Morton (Z-order curve) and Hilbert, are visualized in Figure 8. The main advantage of these curves is that they preserve locality, i.e., two points which are close on the curve are also (to some extent) close in N dimensions, and vice versa. In computer graphics, these curves have many interesting applications, such as enabling fast parallel construction of *Bounding Volume Hierarchies* (BVH) or optimizing memory accesses on GPUs [10, 12].

In the paper by Dado et al. [3], they explain the coherency will be preserved when indexing the data in a depth-first order which, while not explicitly stated, will result in a mapping similar or equal to that of a Morton order. They then take advantage of this coherency and present a novel palette-based compression format achieving good compression rates.

Some issues still remain, however, as the extra index information nearly doubles the DAG in size and, while impressive, the size of the compressed color data can still be improved. Additionally, being palette based, this format requires a quantization of the colors, and thus is susceptible to banding artifacts as the palette size is reduced.

2 Summary of Included Papers

In this Section, we will describe the main contributions of each paper:

Paper I - An extension of the original paper [8], introducing a novel compression algorithm that improves compression speeds by almost 2.0× for large resolutions.

Paper IIA - A novel method for compressing surface properties stored with a voxelized scene representation, which outperforms previous work by up to 1.8× at similar quality.

Paper IIB - An extension to the previous method in which compression is further improved by up to 2.5×.

2.1 Paper I - Fast, Memory-Efficient Construction of Voxelized Shadows

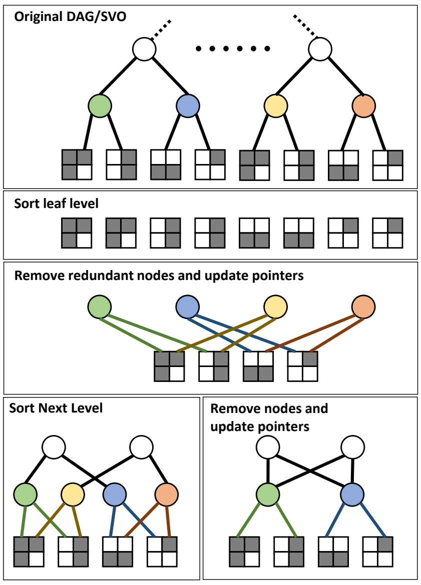

Original DAG/SVO
Sort leaf level
Remove redundant nodes and update pointers
Sort Next Level
Remove nodes and update pointers

Figure 9: The bottom-up compression algorithm used in previous work. © 2016 IEEE.

Problem: To compress an SVO into a DAG or to merge individual sub DAGs into one, Kämpe et al. suggest a straight forward bottom-up algorithm, as seen in Figure 9 [8]. The exact ordering of the nodes is not important when compressing the SVO, only that identical nodes can be found efficiently. To achieve this, they start by sorting all nodes at the leaf level according to a bit mask that represents a set of voxels. In this sorted list the identical nodes can be easily identified as they lie at consecutive indices. From a set of identical nodes only one is chosen to represent all the other nodes (which then can be removed) by redirecting the parent nodes to point to this remaining node. When the leaf level is processed, the next level is processed similarly, but this time using the child pointers instead. An optimal DAG is constructed when all levels have been processed. One problem with this algorithm is that the comparison operator for internal nodes is potentially very expensive. With eight child pointers, eight integer comparisons may have to be performed to find out whether nodes are equal or whether one node should precede another during sorting. Further, this bottom-up approach will process a lot of redundant nodes as compared to if they would have been processed top-down instead. Another problem is that the compression time will be proportional to the size of the leaf level of the SVO, which grows rapidly with higher resolutions.

Methodology: As in the original paper by Kämpe et al., we build the DAG by first constructing a set of sub DAGs, which are then merged into the final dag [8]. Instead of performing expensive pointer comparisons, we realized that if we instead generate hash values for the lowest internal node level, using the bit mask, we can immediately identify identical nodes in that level. We can then recursively generate hash values at the parent level, using the hash values from its children, and immediately identify the roots of identical subtrees without first compressing the lower levels. This is illustrated in Figure 10 for a binary tree. When all the hashes have been calculated, an optimal DAG can be constructed using a top-down algorithm. Starting at level below the root, all nodes are sorted using the hash values as key. In the same way as described earlier, redundant nodes are removed and the parents' pointers are updated. The next level is then generated by concatenating the child nodes of the remaining, now unique, nodes. This process is repeated recursively until the final level is processed. This top-down algorithm has the advantage that children of redundant subtrees can be discarded early, and thus do not need to be processed at all.

An additional benefit of having a sorting key is that a simple optimization can be applied to the final compression pass. Traditionally, merging sub DAGs is done by concatenating all levels of all sub DAGs, which are then processed bottom-up by sorting each layer, removing redundant nodes and updating the parents' pointers. However, since the sub DAGs are sorted on construction, there is no need to concatenate and sort each level of the two sub DAGs, and we can thus simply merge the layers of two sub DAGs in a single sweep over the nodes.

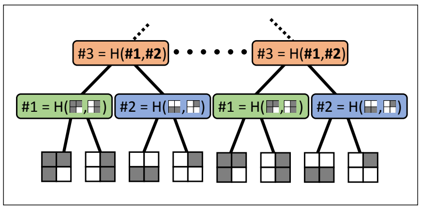


Figure 10: By calculating hashes, bottom up, from the contents of the child nodes, nodes that are roots of identical subtrees will have identical hashes. © 2016 IEEE.

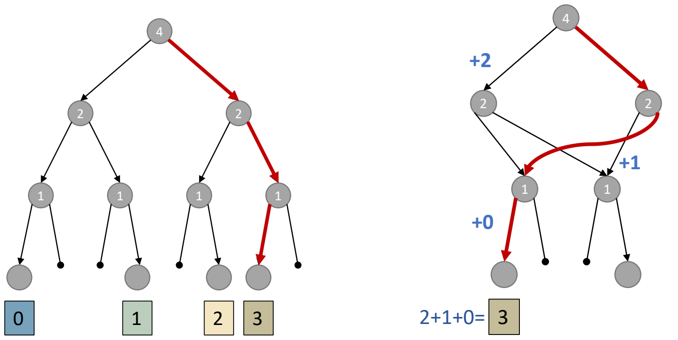
Contribution: In this paper, we managed to obtain a speed up when compressing sub DAGs for non-closed geometry by 1.3× to 2.5×. Since sub-DAG compression is also the most expensive part of the algorithm the total compression time is almost halved when rendering high-resolution shadow information. We also, in the paper, provide a less exact algorithm for identifying undefined regions for closed objects for scenes with non self-intersecting geometry.

2.2 Paper IIA - Compressing Color Data for Voxelized Surface Geometry (Original)

Problem: The memory issues with storing voxelized surface geometry at high resolutions is alleviated by storing the geometry as an SVO or a DAG[9, 7]. However, while there exist many good formats for compressing and querying conventional images, there are few algorithms for efficiently compressing voxelized surface properties. As opposed to a 2D texture mapping, which is dense and coherent and therefore suitable for compression, the DAG represents a surface in 3D with no mapping. We thus need to find a decoupling of the DAG and the colors with a coherent mapping as well as a way to compress the data, since there is no point in storing the geometry as a DAG if the data is going to consume orders of magnitude more memory.

Dado et al., managed to alleviate both these problems [3]. However, their data structure is fairly complex with a mapping that nearly doubles the size of the DAG, and finally, the quality and compression ratios could also be improved.

Methodology: Simply inserting color indices into the nodes will destroy the subtree-merging opportunities. However, if we instead in each node store the the number of voxels represented in the node's subgraph, previously identical nodes will remain identical, and the number of preceding voxels, i.e. the color index, can then be computed using a running sum of the voxel counts of preceding nodes during traversal, as illustrated in Figure 11. In our implementation, we use a 32-bit word to store the 8-bit child mask and the remaining 24-bits for the voxel count, which is typically sufficient for resolutions up to $1K^3$. For larger resolutions, a 24-bit counter is not enough, and we thus store the upper levels in a separate 32-bit word array. These nodes are, however, so few that the memory overhead is typically far less than 0.1%.


4
+2
2
2
+1
1
1
1
1
1
1
+0
0
1
2
3
2+1+0=
3

(a) SVO: Color indices can be trivially mapped to nodes.

(b) DAG: Storing total number of leaf nodes preserves identical nodes.

Figure 11: Example how a desired color index (red path) can be found by a running sum of preceding child nodes total number of leaf nodes.

The first step in encoding is to map the dataset to a 1D array by sorting the colors according to the voxels' positions along a space-filling curve. For the second step, we have investigated two methods of compression, namely:

- **Compression using a 2D mapping:** Here we map the 1D-array to a 2D-texture, again using a space-filling curve, and use conventional compression techniques.
- **Variable Bitrate Block Encoding:** With this novel method, we perform a compression on the 1D-array directly.

The latter method is inspired by the BC (S3TC) and ASTC families of compression, which are developed for real-time decompression with almost zero overhead [13, 6]. The main idea behind these formats is that the texture is divided in to a set of (typically 4 × 4 texels) blocks. Each block has a set of colors and, for each texel, a light-weight value used to reconstruct the original value using the blocks' colors. We partition the array in a set of consecutive blocks where we in each block store two colors and the starting voxel index. Further, we also store an array of weights (typically 3 bits) for each voxel, which we use to interpolate between the two

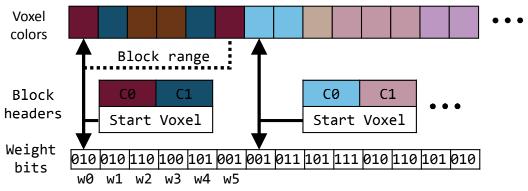

Voxel colors
Block range
Block headers
C0
C1
Start Voxel
C0
C1
Start Voxel
Weight bits
010 010 110 100 101 001 001 011 101 111 010 110 101 010
w0 w1 w2 w3 w4 w5

Figure 12: Visualization of data layout for fixed weight bit width.© 2017 IEEE.

colors in order to calculate the final color for that voxel. An illustration of this data structure can be seen in Figure 12.

Decoding the data is straight forward; when we have calculated the color index, we perform a binary search in the blocks to find the block defining that voxel, and after a direct lookup in the weight array (using the same color index), we can perform an interpolation between the block's two colors.

To encode the data to our new format, we start with each color being it's own block and in a sequential manner try to merge a block with its right **or** left neighbor. The criteria for merging is that we can only merge if the new *mean squared error* (MSE) is less than a specified threshold, and we merge with the neighbor that results in the lowest MSE. During an iteration, a block can only be merged once, so when the last block in the array is processed, the algorithm enters a new iteration, using the newly merged blocks instead. The algorithm finishes when an iteration has finished with no merges performed.

Contribution: We have provided a new real-time format to store and compress any interpolatable values for sparse 3D data and further provided a mechanism to connect this data to a DAG. It outperforms previous formats in both quality, compression and simplicity. Finally, we have shown that it is possible to map this sparse data to a dense texture which can be compressed using conventional algorithms, e.g., JPEG for small files offline or ASTC for fast hardware support and quality.

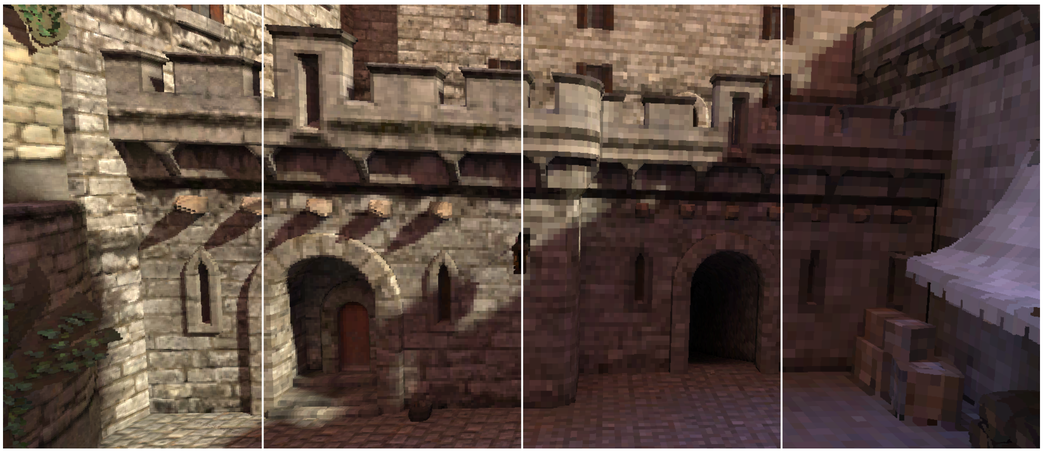


Figure 13: Different levels of mip mapping. © 2017 IEEE.

2.3 Paper IIB - Compressing Color Data for Voxelized Surface Geometry (Extension)

Problem: One issue with **Paper IIA** is that, given 3-bits per weight, there is a theoretical minimum compression size of 12% of the original data. It also lacks mipmapping, which means, e.g., that distant voxels colors will cause aliasing artifacts. In this extension, we address these problems by allowing a variable bitrate (0-4 bits/weight) per block, as well as implementing mipmapping. The main difficulty with allowing for a variable bitrate is that we can no longer directly access the weight required to decompress a specific color, and consequently we must also 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 the block headers and can completely overtake the reduced size of the weight array.

Methodology: 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 Figure 14). 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. In the paper, we used a macro-block size of $16K$ colors, which allows us to pack the data efficiently in the block headers, resulting in the only overhead being the two 64-bit words in the macro blocks. 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 original paper. We first do a direct lookup to find the macro block the voxel index belongs to, as well as the next macro block. With these, we know the range of blocks in which to do a binary search. When we have found the block the voxel belongs to, we use the weight information in the block and macro block to find the weight index. An added benefit of this is that we only need to do a binary search of a subset of all blocks, the blocks below the macro blocks, which improves performance.

To compress, we first build a tree of 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. The tree is built bottom up by the same algorithm as in the original paper, using zero bits per weight. To calculate the parents, we then use this set of compressed blocks and try to compress them further, using one more bit per weight. This process is repeated until we reach the maximum number of bits per weight. Finally, 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.

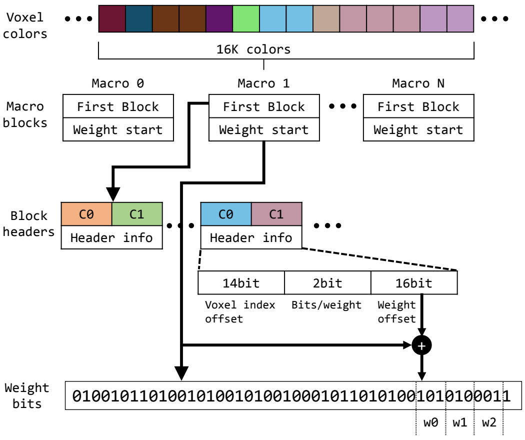

Voxel colors
16K colors
Macro 0
Macro 1
Macro N
Macro blocks
First Block
Weight start
First Block
Weight start
First Block
Weight start
Block headers
C0
C1
Header info
C0
C1
Header info
14bit
2bit
16bit
Voxel index offset
Bits/weight
Weight offset
Weight bits
0100101101001010010100100010110101001010100011
w0
w1
w2

Figure 14: Visualization of data layout for variable weight bit width. © 2017 IEEE.

Finally, mipmapping (Figure 14) is solved by storing the number of nodes in the subgraph, instead of the number of leaf voxels, per node, where the color of a node is the mean of the children's colors. How to encode the indices for mipmap colors was actually already noted in **Paper IIA**.

Contribution: With this paper, we have significantly improved the previous algorithm, allowing better quality at smaller sizes. There is also no longer a theoretical limit at 12%, as any string of colors can be compressed to a single block, with zero bits per weight, although that would generally lead to *very* poor quality. Although the variable bitwidth encoding introduces an extra level of indirection, the lookup speed is on par or faster than the original algorithm due to the reduced binary search range.

Introducing mipmapping unfortunately leads to that the colors of *internal* nodes at the same level will be scattered in memory. However, as colors of nearby voxels are still usually close to each other despite having interleaved the colors of internal nodes, compression rates are actually slightly better.

3 Discussion and Future Work

The focus in this thesis has mainly been on compressing simple properties, such as light visibility and colors. Compressing colors with a DAG allows simple visualization and storage of point clouds from scanned data. Backing this claim, we obtained a LIDAR scanned data set of the Chalmers campus, which we converted to a DAG along with the colors and included it in our paper. This also makes it potentially suitable for medical imaging, for example computer tomography or MRI, which produces large data sets. Furthermore it could also be used to implement cheap real-time reflections by baking the outgoing radiance into the DAG.

An alternative to texture mapping is the PTEX mapping system [2], developed by Walt Disney Animation Studios, which does not require any *uv* assignment, as it applies a separate texture to each face of a subdivision or polygon mesh. Coupling DAG properties to triangles could potentially be an alternative to PTEX and thereby further improve on the usability of the DAG-color format.

A limitation of the research regarding voxel properties so far is that only simple attributes such as colors or normals have been investigated. A tangent to our research would be to store and compress other properties such as incoming light for different points. The developers of the game **The Order: 1886** used *spherical gaussians* (SG) stored in textures to bake incoming lights [17, 1]. This property is possible to interpolate and should be propitiously compatible with some variant of how we compress colors.

Bibliography

[1] Ready at Dawn. https://mynameismjp.wordpress.com/2016/10/09/sg-series-part-1-a-brief-and-incomplete-history-of-baked-lighting-representations/.

[2] Brent Burley and Dylan Lacewell. Ptex: Per-face texture mapping for production rendering. In *Computer Graphics Forum*, volume 27, pages 1155–1164. Wiley Online Library, 2008.

[3] Bas Dado, Timothy R Kol, Pablo Bauszat, Jean-Marc Thiery, and Elmar Eisemann. Geometry and attribute compression for voxel scenes. In *Computer Graphics Forum*, volume 35, pages 397–407. Wiley Online Library, 2016.

[4] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165. ACM, 2006.

[5] Wolfgang Engel. Cascaded shadow maps. *ShaderX5: Advanced Rendering Techniques*, 1, 2006.

[6] Konstantine I Iourcha, Krishna S Nayak, and Zhou Hong. System and method for fixed-rate block-based image compression with inferred pixel values, September 21 1999. US Patent 5,956,431.

[7] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Trans. Graph.*, 32(4):101:1–101:13, July 2013.

[8] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. Fast, memory-efficient construction of voxelized shadows. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, pages 25–30. ACM, 2015.

[9] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[10] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[11] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 579–588. ACM, 2006.

[12] Anthony E Nocentino and Philip J Rhodes. Optimizing memory access on gpus using morton order indexing. In *Proceedings of the 48th Annual Southeast Regional Conference*, page 18. ACM, 2010.

[13] Jorn Nystad, Anders Lassen, Andy Pomianowski, Sean Ellis, and Tom Olson. Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 105–114. Eurographics Association, 2012.

[14] William T Reeves, David H Salesin, and Robert L Cook. Rendering antialiased shadows with depth maps. In *ACM Siggraph Computer Graphics*, volume 21, pages 283–291. ACM, 1987.

[15] George F Simmons. *Introduction to topology and modern analysis.* Tokyo, 1963.

[16] Erik Sintorn, Viktor Kämpe, Ola Olsson, and Ulf Assarsson. Compact precomputed voxelized shadows. *ACM Transactions on Graphics (TOG)*, 33(4):150, 2014.

[17] Jiaping Wang, Peiran Ren, Minmin Gong, John Snyder, and Baining Guo. All-frequency rendering of dynamic, spatially-varying reflectance. In *ACM Transactions on Graphics (TOG)*, volume 28, page 133. ACM, 2009.

[18] Brent Robert Williams. Moxel dags: Connecting material information to high resolution sparse voxel dags. *Master thesis*, 2015.

[19] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.

[20] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 311–318. ACM, 2006.