DOI: 10.1111/cgf.14757

EUROGRAPHICS 2023 / K. Myszkowski and M. Nießner
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 2

Editing Compressed High-resolution Voxel Scenes with Attributes

M. Molenaar[1] and E. Eisemann[1]

[1]TU Delft, The Netherlands

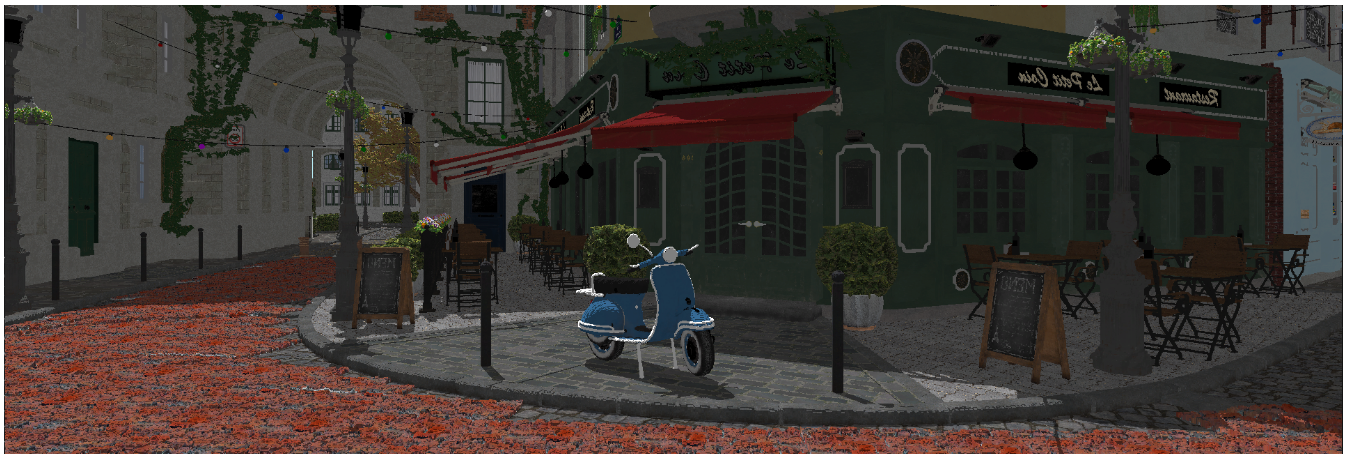

Figure 1: *Voxelization of the Lumberyard Bistro scene with a 3D brick texture painted on the ground.*

Abstract
Sparse Voxel Directed Acyclic Graphs (SVDAGs) are an efficient solution for storing high-resolution voxel geometry. Recently, algorithms for the interactive modification of SVDAGs have been proposed that maintain the compressed geometric representation. Nevertheless, voxel attributes, such as colours, require an uncompressed storage, which can result in high memory usage over the course of the application. The reason is the high cost of existing attribute-compression schemes which remain unfit for interactive applications. In this paper, we introduce two attribute compression methods (lossless and lossy), which enable the interactive editing of compressed high-resolution voxel scenes including attributes.

CCS Concepts
• ***Computing methodologies*** → *Volumetric models; Image compression;*

There is an increasing need for dynamic large-scale 3D voxel structures across various facets of the computer-graphics industry. Since voxels represent space, and not surfaces, they are a good fit for representing volumetric data, which is useful in many scenarios, such as 3D reconstruction [NZIS13] or indirect lighting in real-time rendering [CNS*11].

One downside of voxel representations are the typical high memory requirements. Given that video memory is very constrained on consumer hardware, often less than 8 gigabytes, compression schemes are employed (Sec. 1).Nevertheless, these often imply that access and manipulation of the data becomes slower or even impossible at interactive rates.

In this paper, we focus on voxel models of 3D boundary representations, which lead to sparse voxel occupancy. Various data structures have been developed to reduce storage requirements of such sparse voxel geometry by omitting empty voxels [LH06; MAB19] or exploiting redundancy using a DAG encoding [KSA13].

Along with voxel occupancy there is often a need to store additional information inside non-empty voxels, e.g., presence of colours for 3D painting. These attributes can be decoupled and compressed separately from the geometry which improves the compression ratio [DKB*16]. Various previous works propose specialised methods for compressing attribute data [DKB*16;

DSKA17]. However, none of these works focuses on compression time or enable real-time editing in this context. We present two novel compression algorithms with the aim of supporting real-time editing. Our lossless method achieves high performance making use of a suitable GPU mapping and competitive compression ratios. Our lossy method improves the compression ratio further, while maintaining a performance level that enables interactive large-scale edits in highly-detailed voxel scenes with attributes.

1. Related Work

A regular grid is the simplest method of storing voxel data and indexing is a constant time operation. Yet, the cubic memory costs limit the practical resolution. Especially, for sparse models, where comparatively few voxels are filled, much memory is actually wasted on empty space. In the following, we cover approaches addressing this challenge.

Spatial Hashing [GG98; ASA*09] stores filled voxels in a hash map, leading to a linear relation between memory use and non-empty voxels. Access becomes more computationally involved but can remain constant in *expected* time complexity for a suitable hash encoding. Nevertheless, *worst-case* complexity remains linear due to potential entry collisions, which is especially problematic for Single Instruction Multiple Data (SIMD) architectures, such as GPUs, where the throughput of a thread group is governed by the slowest thread. Perfect Spatial Hashing [LH06] avoids collisions with a *perfect* hash function. It enables a constant-time worst-case complexity. However, building a perfect hash is expensive, prohibiting its use for real-time modifications in large-scale voxel scenes.

Sparse Voxel Octree (SVO) recursively subdivide the volume into 2^3 same-sized regions, where empty space is not subdivided further. Hereby, octrees have the added benefit of acting as an acceleration structure for rendering. Still, while voxel updates in a single threaded program are conceptually straightforward, memory allocation and CPU/GPU transfer pose real-world challenges [KKK18]. Similarly, a full rebuild (e.g., [Kar12] for a GPU version) also is no alternative when opting for real-time updates in high-resolution volumes.

Sparse Voxel Directed Acyclic Graph (SVDAG) exploit repeating patterns, which manifest themselves as identical subtrees. Having parent nodes refer to a single remaining subtree saves memory. First expressed by [WD89] in two dimensions, it was later extended to 3D by [PU03] and popularised by [KSA13]. Extensions to this approach match identical subtrees under symmetry [VMG16; ČMBB19] or allow approximate merges [vdLSE20].

Storing attributes, such as colours, in an SVDAGs is challenging. SVOs can store such information in leaves. Yet, in an SVDAG, this would make merging subtrees more difficult, as geometry and attributes need to match [PU03]. Decoupling attributes from geometry [DKB*16], by collecting them along a space-filling curve (Morton) and storing them in a separate array, can be a remedy.

Attributes often occupy more memory than geometry. Therefore, various compression methods, lossless [DKB*16] and lossy [DSKA17], but none are suitable for real-time editing, with the exception of the HashDAG structure [CBE20], which demonstrated editing of existing SVDAGs at interactive frame rates. However, new and modified colours are not immediately compressed, leading to high memory usage when editing large regions. Our work addresses this issue and builds atop this solution. For additional information on SVDAGs and multi-resolution structures, the reader is referred to [ABD*18].

2. Our method

In the following, we describe our solution. We first provide a short overview of SVDAG structures, since they form the basis for efficient voxel editing (Sec. 2.1), in this context, we also discuss our proposed changes to enable our solution. Next, we introduce our two novel colour compression algorithms. The lossless method is GPU accelerated and re-compresses large amounts of colour in real-time (Sec. 2.2), followed by our lossy compression scheme (Sec. 2.3).

2.1. SVDAG Background

Constructing and updating a SVDAG: An SVDAG is typically built from an SVO, where the SVO construction is orthogonal to our topic and will not be covered here. Turning an SVO into a SVDAG requires searching for identical subtrees and eliminating all but one instance. This is a recursive process, i.e., a duplicate subtree might itself contain another subtree which appears somewhere else in the SVO.

Comparing two subtrees with M nodes is an $O(M)$ operation, which is computationally expensive. The bottom-up construction presented in [PU03; KSA13] significantly reduces computation time. Starting at leaf level, duplicates are removed and child pointers in the level above updated. The process iteratively works its way up the tree. Starting at the bottom guarantees that for subtrees with respect to the current processing level, two children are the same if and only if their pointers match. Thus finding duplicates simply entails comparing nodes and their child pointers.

For offline construction [KSA13], duplicates are removed by sorting the nodes at each level, which is impractical for real-time editing, as arrays would need to be sorted after insertions and removal of nodes. The HashDAG uses a hash map to guarantee fast insertions and stable pointers, resulting in a significant speedup, albeit at the cost of an increased memory usage [CBE20].

Decoupling attributes from geometry Decoupling attributes from the voxel geometry improves compression, as attributes do not have to match when merging subtrees. Attributes of (non-empty) voxels are collected along a 3D Morton curve, equivalent to a depth first traversal of the SVO/SVDAG (see Fig. 2 for a 2D example). The index of an attribute associated with a voxel is given by the number of non-empty voxels preceding the current voxel along the Morton curve. To accelerate the index computation, each node contains the number of non-empty voxels in its subtree, which remains compatible with the subtree merging procedure [DSKA17; CBE20]. During SVDAG traversal, the attribute index for a voxel can be accumulated efficiently.

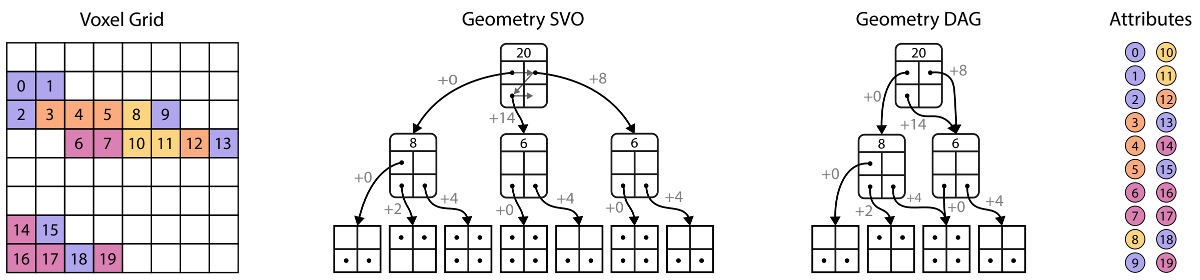

Figure 2: *The structure of the voxel grid (left) is captured in an SVO (centre/left) which is converted into a DAG (centre/right). Each node in the DAG stores the number of voxels in its subtree. The number of preceding voxels are accumulated during traversal (grey) to give an index into the attributes array (right). This two dimensional illustration trivially extends to three dimensions.*

For a static SVDAG all attributes can be exported to a single contiguous array. However, editing can fill previously-empty voxels and requires inserting attributes in this array. Therefore, both of our compression methods store (blocks of) attributes in a sorted array which results in $O(N)$ updates. To alleviate this issue we split the single large attribute array into smaller *chunks*, one for each node at a predetermined level of the tree. This defines an upper bound on the number of unedited attributes that need to be relocated in memory during editing. It also creates an opportunity for multi threading as chunks can be updated independently.

2.2. Lossless Attribute Compression

Real-time editing of SVDAGs with attributes requires fast compression and random access decompression. The latter excludes most popular sequential techniques, such as Lempel-Ziv (LZ) [ZL77] and Huffman coding [Huf52], which do not support efficient access to individual items. LZ en/decoding is therefore not suitable, even when using GPU versions [WS18; TRD*21; OS11; Ozs14; ZH14; LH19; SGDF19]. Our approach achieves constant time look-ups but still enables a strong compression. To simplify our explanations, we will explain our solution in the context of colour compression.

The lossy method by Dolonius [DSKA17] splits the colour array into large "macro" blocks which could be used to multi thread on the CPU. To achieve massive parallelism on the GPU we instead split the colour array into very small blocks of 128 consecutive entries, each of which is processed by different GPU work groups. Our underlying assumption is that the colour array exhibits limited local change, which we exploit in our compression scheme. We normalise the colours by computing a minimum m and maximum M per channel in each block and store the actual colour as an offset from the minimum. The number of bits used for the offset is linked to the value of $M - m$.

As a second step a Frame-Of-Reference compression [DUH*19] is applied. Instead of an offset to the minimum, we will store offsets to suitable reference colours. This leads to smaller values and thus less memory usage. The reference colours are computed by quantizing the normalised colours into eight equally sized bins (3 bits) per channel.

As many reference colours within a block are identical, we only store them once in a table and refer to them by index. Further, as

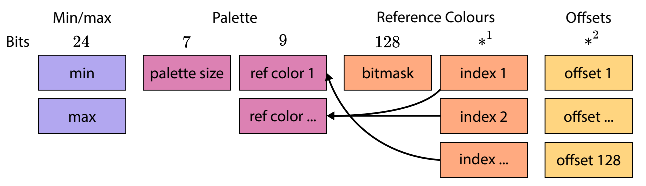

Figure 3: *Memory layout of a single block of 128 colours as compressed with our lossless compression method.*
$*^1 = \lceil \log_2(\textit{palette size}) \rceil$ $\quad *^2 = \sum_{c \in r,g,b} \lceil \log_2(\frac{c_{max} - c_{min}}{8}) \rceil$

especially consecutive entries are likely to share the same reference colour, we only store an index when it changes from one entry to the next. This information is stored in a 128 bit mask for the entire block. Accessing an index of an entry can be achieved by using a bitcount on the bit mask up to the entry in question. The resulting value indicates the offset into the index table.

All this information is stored in a compact memory format as shown in figure 3. Each block may occupy a variable amount of memory so we use a 64-bit look-up table to provide efficient access to individual blocks. Everything combined the fixed cost per block is $48 + 7 + 128 + 64 = 247$ bits resulting in a minimum theoretical compression ratio of $\frac{247}{128*24} \approx 8.04\%$.

Because blocks cover a fixed interval of colours, there is no option for fast insertions and removals. To limit compute time and avoid treating the entire scene, we introduced colour chunks (compare Sec. 2.1). The impact during an edit can then be limited; we only decompress the modified colour chunks into a temporary buffer (on the GPU), apply the modifications, and subsequently compress the colours again.

2.3. Lossy Attribute Compression

A small loss in precision opens the door to a much smaller memory footprint, as evidenced by many modern image file formats, such as JPEG [Wal91] with their impressive compression ratios. Dolonius [DSKA17] introduced a lossy scheme aimed specifically at compressing the 1D colour arrays associated with SVDAGs. Their algorithm is designed to achieve high quality but does not enable real-time modifications. We extend this work by presenting a novel fast algorithm, which is compatible with their file format. To ease understanding, we briefly recap their method before detailing our algorithm.

S3TC block image compression schemes divide an image into a regular grid of 4x4 pixel blocks. Colours inside each block are projected onto a line segment in RGB colour space and stored as an interpolation coefficient between the two endpoints. Dolonius [DSKA17] applies the same concept to the 1D colour array but utilises variable-sized blocks to better adapt to the input data.

The number of bits used to store the interpolation weights varies per block and are stored in a separate array. Like S3TC the line segment endpoints are encoded with 16 bits using the RGB565 format. Blocks can also encode a run of a single colour without loss of precision when desired. For additional information regarding the memory layout, we refer the reader to [DSKA17].

To compress a set of input colours, the Dolonius algorithm initially considers each colour as a separate block. It then sequentially merges neighbouring blocks using a least-squares line fit to the colours. This process is repeated until no two neighbouring blocks can be merged without exceeding a given error threshold. The number of bits used to store the interpolation bits impacts the final error, hence, all combinations are tested, tracked by creating a tree of all potential blocks. A tree cut then decides on the final blocks.

For real-time compression, this solution is not adequate and only used during preprocessing. When no more than two neighbouring colours fit on each line segment the algorithm finishes in $\Theta(N)$ time. However, in the worst case all colours map to a single line segment leading to a worst-case time complexity of $\mathcal{O}(N \log N)$. This is problematic as larger scenes generally contain more coherence due to oversampling of textures creating larger blocks.

2.3.1. Our line fitting

Our solution targets a compression based on a more local decision making. To facilitate explanations, we will cast the problem as a line fitting strategy to a stream of points. We aim at finding a low but unknown number of line segments which provide a good fit to a variable number of consecutive points from the stream. A good fit refers to an allowed margin, which controls the tradeoff of the number of line segments and the fitting quality.

We sequentially iterate over the input and decide for each colour whether it can fit onto a line segment with the previous colours or whether a new block must be created. This greedy algorithm ensures that each entry is touched only once. To incrementally construct line segments, we use a modified version of the Hough transform. We first focus on a 2D case before extending our solution to more dimensions.

Hough Transform [Hou62] is a well known operation in image processing for detecting lines in images. Lines are represented in a dual space as a point with coordinates θ and ρ; representing a line with angle θ at distance ρ from the origin (see Fig. 4).

For the classical line-detection use case, a texture representing the dual space is used. For a given point in primal space, the texels representing lines passing through the point are rasterized, which can be found with a simple relationship:

$$\begin{aligned} \vec{p} &= (x,y) \\ \rho(\theta) &= x\cos(\theta) + y\sin(\theta) \end{aligned} \tag{1}$$

By using additive blending [DH72], high values in the texture ultimately reveal those dual representations of lines that pass through many points in primal space. Relying on a texture and rasterization, quantizes the dual representation, which makes the line fitting approximate.

In our context, we want to incrementally map points to the dual space to verify if they all can still be well approximated by a line. This approach has a number of problems though. First, using quantization to approximate nearby values creates an unstable error margin. For example the values 9.51 and 10.49 will map to the same pixel while 10.49 and 10.51 would not, despite being much closer. A second problem is performance. For each new pixel many different θ values need to be evaluated. Furthermore it is not uncommon to fit millions of individual line segments when compressing a voxel scene. The rasterization method requires that the texture is cleared before fitting starts, which is an expensive operation.

To fix these issues we keep the presentation of ρ continuous. For each discrete step of θ, we store a minimum and maximum value of ρ, representing all possible lines with angle θ which pass by the previously added points with a distance less than the desired fitting margin. This margin creates a range of ρ values which are considered valid for a point (Fig. 4).

To further accelerate the process, we observe that only lines which pass close to *all* points are relevant. Hence there we skip θ values as soon as no value of ρ can define a valid line for the preceding points anymore. We accomplish this by keeping a list of θ steps for which a fitting line still exists (Fig. 4).

2.3.2. Extension to 3D

Many attributes can have more than two dimensions. For example, colours triplets in form of red, green, blue channels (RGB). The 3D equivalent of the Hough transform (for example, employed in [DDSD03]) requires two angles, which is prohibitively expensive due to the large number of values that would need to be cleared when starting the fitting.

Instead we perform a 2D Hough transform for the R/G, G/B and R/B planes. As soon as the fitting fails for one plane, we stop. Considering the resulting 2D lines as projections of the wanted 3D line, we can reconstruct the 3D line as a plane intersection involving two of these planes. For example, using lines in the *rb* and *gb* planes (defined as $l(t) = \vec{o} + t\vec{d}$) we construct a 3D line as follows:

$$\begin{aligned} \vec{o_{rgb}} &= (\vec{o_{rb}}^{\,r} - \vec{o_{rb}}^{\,b}\frac{\vec{d_{rb}}^{\,r}}{\vec{d_{rb}}^{\,b}}, \vec{o_{gb}}^{\,g} - \vec{o_{gb}}^{\,b}\frac{\vec{d_{gb}}^{\,g}}{\vec{d_{gb}}^{\,b}}, 0) \\ \vec{d_{rgb}} &= (\frac{\vec{d_{rb}}^{\,r}}{\vec{d_{rb}}^{\,b}}, \frac{\vec{d_{gb}}^{\,g}}{\vec{d_{gb}}^{\,b}}, 1) \end{aligned} \tag{2}$$

Tracking all three planes allows us to choose the combination that maximises the spread along a shared axis (b in Eq. 2), which reduces numerical issues due to division by values close to zero.

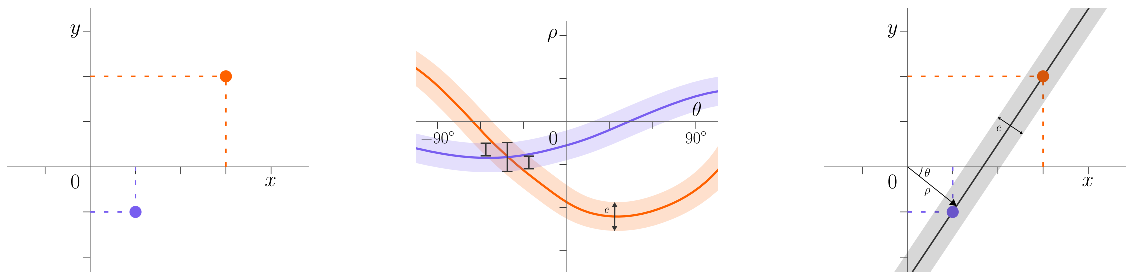

Figure 4: *Two points in (x,y) space (**left**) and their representation in (θ,ρ) space after Hough transform. (**center**). The intersection of the two lines in (θ,ρ) space defines a line in (x,y) which passes through both points (**right**). To allow for approximate fitting we define an error margin e in ρ which corresponds to an euclidean error in (x,y) space. To accelerate the search we discretize θ keeping only the overlapping values. The ρ bounds are stored in a continuous representation (**center**)*

3. Implementation

Here, we discuss implementation details and the integration in the available interactive-editing HashDAG framework [CBE20].

Editing the SVDAG We modified the original code to integrate our colour compression functions. Further, the original work was implemented on the CPU and we improved performance by having a separate copy of the DAG maintained in system memory, mirrored on the GPU. Editing tools, such as a sphere placement tool and paint brush are readily available. We also produced a new stamp tool that modifies the surface and colour simultaneously (see Fig. 1 and video). In all cases, we determine the region of their impact by subdividing the space into 2x2x2 regions recursively, traversing only those, where voxels are modified. For unmodified space, we copy the pointer to nodes in the original SVDAG.

The colour array consists of chunks, one for each node at level 8 in the tree when using lossy compression (same as HashDAG [CBE20]) and level 7 when using lossless compression (where level 0 is the root of the three). The reason for using different values is that the GPU (lossless method) requires larger pieces of work to make effective use of the hardware. We define the contents of new or modified colour chunks as an ordered stream of three basic operations: **copy** existing colours, **fill** a single colour and **write** new colours. Once all colour operations for a chunk have been recorded, the chunk can be compressed using either our lossless or lossy method on a worker thread.

Lossless colours Our lossless compression algorithm does not support incremental updates and requires a full decompression and re-compression of any modified colour chunk. Both compression and decompression are implemented in CUDA. For compression we spawn a work group for each 128 colour blocks with an equal amount of threads. Each work group compresses its block into a shared memory bit stream, which is atomically merged into a global bit stream when the work group finishes. We use the cooperative groups feature in CUDA to utilise the latest hardware intrinsics for reductions and prefix sums available in the GPU used for testing.

Lossy colours Because the lossy compression scheme uses variable sized blocks, partial updates are possible. Contiguous subsets of new colours are compressed into one or more blocks while unmodified blocks are copied.

To compress new colours, we apply the greedy line fitting algorithm (Sec. 2.3.2), where the angle θ is discretized into 96 steps, which was empirically chosen. Computation of ρ is performed in parallel using AVX2 instructions by grouping the θ steps into groups of size 8. The result of the Hough-transform fitting is a line rather than a line segment so a second loop is required to compute the endpoints. The number of bits used to store the interpolations weights is decided by exhaustively checking all options between 1 and 4 bits (using SSE intrinsics) and picking the lowest bit count which allows each point to be fit, while staying within the users specified error threshold.

Like Dolonius the threshold is defined as the maximum Euclidean distance between any input colour and its compressed representation in the output. We control the line fitting precision by adjusting the 2D precision parameter e (see figure 4) which we set to half of the desired 3D error.

In addition approximate line fitting, quantization of the interpolation weights, and line segment endpoints (RGB565 format) also increase the final colour error. As such, a situation may occur, where it is not possible to store the fitted colours into the same block. In such cases we encode these colours into individual colour blocks using the RGB101210 format (Sec. 2.3).

4. Results

To evaluate the performance of our algorithms and the HashDAG [CBE20], we will compare against the lossless compression scheme of Dado [DKB*16] and the compression algorithm of Dolonius [DSKA17]. In case of Dolonius, we use the original CUDA implementation and created our own version of the Dado algorithm based on their paper.

All tests were performed on a system running a 10th generation Intel I9 processor and a Nvidia RTX3070Ti on Linux. We found that CUDA performance on Windows is significantly degraded when using managed memory.

Colour Compression In order to accurately measure just the compression performance, we extract raw attribute arrays from various scenes and compress them in their entirety. The tested scenes are voxelized versions of the Epic Citadel, Lumberyard Bistro Exterior, and San Miguel scenes using diffuse textures as voxel colours.

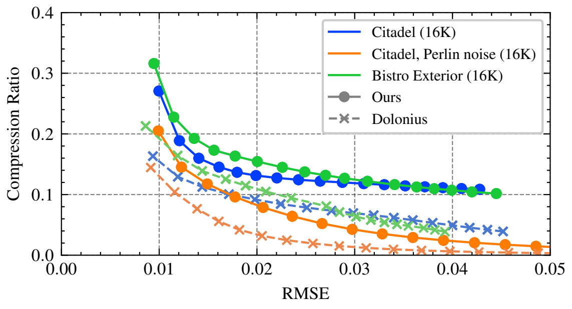

Figure 5: *A plot showing compression ratio as a function of Root Mean Square Error (RMSE). Comparing our lossy method to Dolonius [DSKA17].*

In addition we test the Citadel scene with coloured Perlin noise to emulate the low frequency signal of irradiance (Fig 6). For the lossy colour compression schemes, we use a colour error threshold of 0.06. As with actual voxel editing, we split the attribute array into smaller chunks. With the exception of our lossless method and Dolonius' CUDA version [DSKA17], the chunks are processed in parallel using multiple CPU threads.

The results are shown in Table 1. For our *lossless* method, we tested both the original RGB888 input, as well as RGB565. Storing RGB565 colours will lead to a small quality loss due to the conversion of the 24 bit input colours, but, being lossless, this error cannot grow over time when editing. We also include the method employed by HashDAG [CBE20] in the table which uses the memory layout of Dolonius. HashDAG only performs compression by detecting equivalent consecutive colours, a very simple form of run-length encoding.

Our lossless method is able to compress data at over 20GB/s which is two orders of magnitude faster than the method of Dado [DKB*16]. At lower resolutions our method is able to achieve compression ratios comparable to Dado. However our method does not scale as well to larger resolutions due to the fixed block size, and, thus, memory overhead per voxel. Increasing the block size would help, but grouping more colours also reduces their coherence in each block. In practice, we found that blocks of 128 provide a good balance for most scenes.

Note that the Citadel scene is a bit of an outlier with regards to memory scaling. The scene uses low-resolution textures and underneath the castle are large single-coloured triangles resulting in long homogeneous ranges in the colour array. The method of Dad o is able to efficiently store ranges of single colours, while our method is bound by the use of blocks. The opposite happens when storing Perlin noise rather than diffuse textures. Colours are not repeated locally, resulting in Dado using more memory than the uncompressed input. In comparison our method using local offsets handles this situation much better.

Our *lossy* compression algorithm was designed to achieve high run-time performance, while utilising the memory format of Dolonius [DSKA17]. At roughly 1GB/s on a 10 core CPU, our method is well suited to large scale interactive editing or real-time smaller edits. In terms of compression ratio our method typically requires between 30% and 80% more memory than Dolonius' offline method.

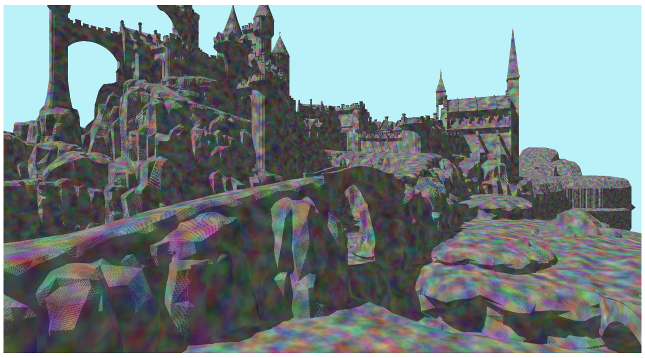

Figure 6: *The Epic Citadel scene using Perlin noise colours.*

Both lossy methods trade quality for compression ratio, which is illustrated in Fig. 5. The method of Dolonius outperforms our lossy algorithm, which we attribute to various factors. They perform an extensive search over the space of combination of potential line segments rather than our naive greedy method. The line fitting is performed using a least squares optimisation, which we expect to outperform our 3D mapping of the Hough transform in terms of accuracy. Their design decisions opt for better compression but it makes their method unsuitable for interactive editing. Given the gain in performance, the increased memory usage could be seen as modest.

Real-Time Editing To demonstrate the use of our methods for editing, we have recorded an editing session, where the Citadel scene is painted using a rainbow brush defined by a Perlin noise function for each of the three colour channels. Fig. 7 shows the memory usage during editing, the time it takes to perform the compression, update the colours, as well as the total frame time. The colour update time includes the CPU to GPU copy required for the colours to be displayed on screen plus the compression step.

We compare our compression methods to the work of HashDAG [CBE20] on which our code is based. HashDAG loads scenes using the same compression format as Dolonius and our lossy method. Colours that are modified during editing are not compressed and instead encoded into individual colour blocks leading to high memory usage over time (Fig. 7). In contrast the use of real-time compression using our methods ensures that memory usage remains stable.

Our lossy compression algorithm adds roughly 13 milliseconds (50%) to each frame. While this difference is not insignificant, it does show that our performance is suitable for interactive editing. Our lossless method, which is GPU accelerated, slightly outperforms the HashDAG method, which is in line with the compression performance on synthetic benchmarks (Table 1).

5. Conclusion

To our knowledge, this is the first demonstration of an interactive editing of a full SVDAG with attributes in the compressed domain. While previous work required decompressing attributes, leading to a large memory overhead, or relied on an offline processing, our method achieves fast execution times while keeping memory cost low.

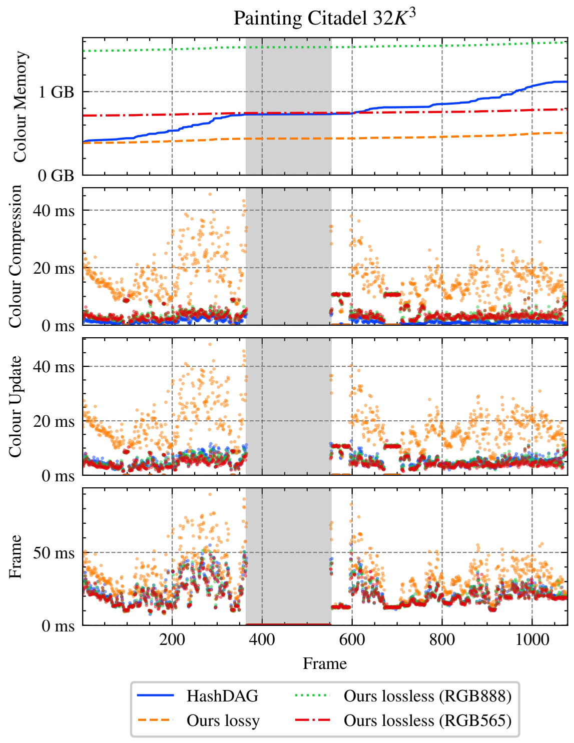

Figure 7: *From top to bottom: Colour memory usage, colour compression-, colour update- (including upload), and frame time, when painting the Epic Citadel* 32K^3 *with a rainbow colour brush. The grey area indicates frames in which no editing takes place.*

We have presented two methods for compressing attributes; a lossless and lossy solution. This makes our approach suitable for many application scenarios. The lossless method is constructed from well-known building blocks [DUH*19] and supports random-access decoding. It can compress large amounts of data in little time by leveraging the GPU. The lossy solution enables even lower memory usage although at the cost of larger colour errors and reduced performance. We believe our solutions increases the viability of using SVDAGs in interactive applications, as attributes such as colours, typically occupy more memory than the geometry itself.

While the results are positive, large edits still run at interactive- rather than real-time frame rates. This might be of relevance for some applications, such as VR painting. Although we do believe that real-time performance is possible if the GPU were to be utilised in the entire editing process. Further, the scenes themselves are currently static and how to integrate animation capabilities into SVDAGs is still an open question.

6. Acknowledgements

This work is supported in part by the VIDI Grant NextView, funded by the NWO Vernieuwingsimpuls.

References

[ABD*18] Assarsson, Ulf, Billeter, Markus, Dolonius, Dan, et al. "Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to Pre-computed Shadows". *EG 2018 - Tutorials*. DOI = 10.2312/egt.20181028 - ISSN = 1017-4656. The Eurographics Association, 2018. URL: `http://graphics.tudelft.nl/Publications-new/2018/ABDEJSS18` 2.

[ASA*09] Alcantara, Dan A, Sharf, Andrei, Abbasinejad, Fatemeh, et al. "Real-time parallel hashing on the GPU". *ACM SIGGRAPH Asia 2009 papers*. 2009, 1–9 2.

[CBE20] Careil, Victor, Billeter, Markus, and Eisemann, Elmar. "Interactively modifying compressed sparse voxel representations". *Computer Graphics Forum*. Vol. 39. 2. Wiley Online Library. 2020, 111–119 2, 5, 6.

[ČMBB19] Čerešník, Peter, Madoš, Branislav, Baláž, Anton, and Bilanová, Zuzana. "SSVDAG*: Efficient Volume Data Representation Using Enhanced Symmetry-Aware Sparse Voxel Directed Acyclic Graph". *2019 IEEE 15th International Scientific Conference on Informatics*. IEEE. 2019, 000201–000206 2.

[CNS*11] Crassin, Cyril, Neyret, Fabrice, Sainz, Miguel, et al. "Interactive indirect illumination using voxel cone tracing". *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, 1921–1930 1.

[DDSD03] Décoret, Xavier, Durand, Frédo, Sillion, François X, and Dorsey, Julie. "Billboard clouds for extreme model simplification". *ACM SIGGRAPH 2003 Papers*. 2003, 689–696 4.

[DH72] Duda, Richard O and Hart, Peter E. "Use of the Hough transformation to detect lines and curves in pictures". *Communications of the ACM* 15.1 (1972), 11–15 4.

[DKB*16] Dado, Bas, Kol, Timothy R, Bauszat, Pablo, et al. "Geometry and attribute compression for voxel scenes". *Computer Graphics Forum*. Vol. 35. 2. Wiley Online Library. 2016, 397–407 1, 2, 5, 6.

[DSKA17] Dolonius, Dan, Sintorn, Erik, Kämpe, Viktor, and Assarsson, Ulf. "Compressing color data for voxelized surface geometry". *IEEE transactions on visualization and computer graphics* 25.2 (2017), 1270–1282 2–6.

[DUH*19] Damme, Patrick, Ungethüm, Annett, Hildebrandt, Juliana, et al. "From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms". *ACM Transactions on Database Systems (TODS)* 44.3 (2019), 1–46 3, 7.

[GG98] Gaede, Volker and Günther, Oliver. "Multidimensional access methods". *ACM Computing Surveys (CSUR)* 30.2 (1998), 170–231 2.

[Hou62] Hough, Paul VC. *Method and means for recognizing complex patterns*. US Patent 3,069,654. 1962 4.

[Huf52] Huffman, David A. "A method for the construction of minimum-redundancy codes". *Proceedings of the IRE* 40.9 (1952), 1098–1101 3.

[Kar12] Karras, Tero. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees". *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. 2012, 33–37 2.

[KKK18] Kim, Yeojin, Kim, Byungmoon, and Kim, Young J. "Dynamic deep octree for high-resolution volumetric painting in virtual reality". *Computer Graphics Forum*. Vol. 37. 7. Wiley Online Library. 2018, 179–190 2.

[KSA13] Kämpe, Viktor, Sintorn, Erik, and Assarsson, Ulf. "High resolution sparse voxel dags". *ACM Transactions on Graphics (TOG)* 32.4 (2013), 1–13 1, 2.

[LH06] Lefebvre, Sylvain and Hoppe, Hugues. "Perfect spatial hashing". *ACM Transactions on Graphics (TOG)* 25.3 (2006), 579–588 1, 2.

[LH19] Lu, Li and Hua, Bei. "G-Match: a fast GPU-friendly data compression algorithm". *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2019, 788–795 3.

[MAB19] MUSETH, KEN, AVRAMOUSSIS, NICK, and BAILEY, DAN. "OpenVDB". *ACM SIGGRAPH 2019 Courses*. 2019, 1–56 1.

[NZIS13] NIESSNER, MATTHIAS, ZOLLHÖFER, MICHAEL, IZADI, SHAHRAM, and STAMMINGER, MARC. "Real-time 3D reconstruction at scale using voxel hashing". *ACM Transactions on Graphics (ToG)* 32.6 (2013), 1–11 1.

[OS11] OZSOY, ADNAN and SWANY, MARTIN. "CULZSS: LZSS lossless data compression on CUDA". *2011 IEEE International Conference on Cluster Computing*. IEEE. 2011, 403–411 3.

[Ozs14] OZSOY, ADNAN. "Culzss-bit: A bit-vector algorithm for lossless data compression on gpgpus". *2014 International Workshop on Data Intensive Scalable Computing Systems*. IEEE. 2014, 57–64 3.

[PU03] PARKER, ERIC and UDESHI, TUSHAR. "Exploiting self-similarity in geometry for voxel based solid modeling". *Proceedings of the eighth ACM symposium on Solid modeling and applications*. 2003, 157–166 2.

[SGDF19] STEIN, CHARLES MICHAEL, GRIEBLER, DALVAN, DANELUTTO, MARCO, and FERNANDES, LUIZ GUSTAVO. "Stream parallelism on the LZSS data compression application for multi-cores with GPUs". *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2019, 247–251 3.

[TRD*21] TIAN, JIANNAN, RIVERA, CODY, DI, SHENG, et al. "Revisiting huffman coding: Toward extreme performance on modern gpu architectures". *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2021, 881–891 3.

[vdLSE20] Van der LAAN, REMI, SCANDOLO, LEONARDO, and EISEMANN, ELMAR. "Lossy geometry compression for high resolution voxel scenes". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.1 (2020), 1–13 2.

[VMG16] VILLANUEVA, ALBERTO JASPE, MARTON, FABIO, and GOBBETTI, ENRICO. "SSVDAGs: Symmetry-aware sparse voxel DAGs". *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2016, 7–14 2.

[Wal91] WALLACE, GREGORY K. "The JPEG still picture compression standard". *Communications of the ACM* 34.4 (1991), 30–44 3.

[WD89] WEBBER, ROBERT E and DILLENCOURT, MICHAEL B. "Compressing quadtrees via common subtree merging". *Pattern recognition letters* 9.3 (1989), 193–200 2.

[WS18] WEISSENBERGER, ANDRÉ and SCHMIDT, BERTIL. "Massively parallel Huffman decoding on GPUs". *Proceedings of the 47th International Conference on Parallel Processing*. 2018, 1–10 3.

[ZH14] ZU, YUAN and HUA, BEI. "GLZSS: LZSS lossless data compression can be faster". *Proceedings of Workshop on General Purpose Processing Using GPUs*. 2014, 46–53 3.

[ZL77] ZIV, JACOB and LEMPEL, ABRAHAM. "A universal algorithm for sequential data compression". *IEEE Transactions on information theory* 23.3 (1977), 337–343 3.

Scene	Method	Compression Time	Compression Ratio (%)	RMSE
Bistro Exterior (8K)	Ours (lossy)	0.77s	18.44%	0.0158
649MB	Dolonius (lossy)	76.66s	**13.93%**	0.0170
	Ours (lossless RGB888)	**0.07s**	72.04%	0.0000
	Ours (lossless RGB565)	0.07s	35.91%	0.0140
	Dado (lossless)	4.15s	74.95%	0.0000
	HashDAG (lossless)	0.28s	249.21%	0.0000
Bistro Exterior (16K)	Ours (lossy)	3.25s	17.31%	0.0156
2643MB	Dolonius (lossy)	312.12s	**12.54%**	0.0168
	Ours (lossless RGB888)	**0.14s**	68.96%	0.0000
	Ours (lossless RGB565)	0.14s	33.62%	0.0139
	Dado (lossless)	20.10s	69.91%	0.0000
	HashDAG (lossless)	1.23s	245.24%	0.0000
Citadel (8K)	Ours (lossy)	0.22s	16.04%	0.0161
210MB	Dolonius (lossy)	22.97s	**12.36%**	0.0170
	Ours (lossless RGB888)	**0.03s**	65.84%	0.0000
	Ours (lossless RGB565)	0.03s	33.88%	0.0088
	Dado (lossless)	1.56s	63.72%	0.0000
	HashDAG (lossless)	0.08s	183.63%	0.0000
Citadel (16K)	Ours (lossy)	0.80s	14.52%	0.0161
844MB	Dolonius (lossy)	92.88s	**10.06%**	0.0171
	Ours (lossless RGB888)	**0.06s**	55.30%	0.0000
	Ours (lossless RGB565)	0.06s	26.53%	0.0088
	Dado (lossless)	10.62s	43.43%	0.0000
	HashDAG (lossless)	0.21s	138.15%	0.0000
Citadel (32K)	Ours (lossy)	3.19s	13.61%	0.0161
3412MB	Dolonius (lossy)	388.63s	**7.67%**	0.0163
	Ours (lossless RGB888)	**0.15s**	45.06%	0.0000
	Ours (lossless RGB565)	0.16s	20.20%	0.0089
	Dado (lossless)	64.00s	29.85%	0.0000
	HashDAG (lossless)	0.74s	97.02%	0.0000
Citadel, Perlin Noise (16K)	Ours (lossy)	0.61s	9.61%	0.0178
844MB	Dolonius (lossy)	37.50s	**5.58%**	0.0161
	Ours (lossless RGB888)	**0.06s**	68.59%	0.0000
	Ours (lossless RGB565)	0.06s	27.14%	0.0142
	Dado (lossless)	10.85s	172.75%	0.0000
	HashDAG (lossless)	0.40s	263.31%	0.0000
San Miguel (16K)	Ours (lossy)	2.41s	17.15%	0.0154
1974MB	Dolonius (lossy)	236.73s	**12.04%**	0.0156
	Ours (lossless RGB888)	**0.10s**	63.01%	0.0000
	Ours (lossless RGB565)	0.10s	32.93%	0.0149
	Dado (lossless)	17.47s	79.23%	0.0000
	HashDAG (lossless)	0.75s	200.46%	0.0000

Table 1: *Comparison between ours and existing colour compression methods for SVDAGs. The size of the uncompressed colour array (24 bit per colour) is given for each scene. The colour array is split into smaller chunks which allows the lossy and Dado implementations to be multi threaded. Our lossless method and Dolonius achieve parallelism through GPU processing and run in a single CPU thread. The lossy methods were configured to target a maximum error of* 0.06 *per colour channel.*