Interactively Modifying Compressed Sparse Voxel Representations

V. Careil[1,2] , M. Billeter[1] , E. Eisemann[1] [†]

[1]Delft University of Technology, The Netherlands
[2]Université de Rennes, France

Figure 1: *To test and demonstrate our method for editing large sparse voxel geometries, we have implemented an interactive prototype application with support for interactive editing operations. The left image shows a copy operation in the* Epic Citadel *scene, voxelized at a resolution of* $(128k)^3$. *The statue inside this building contains about* 170k *voxels. The middle image shows larger scale edits, copying an entire building (order of* 80M *voxels). The right images illustrate tools to add and delete voxels, as well as paint voxel color attributes. The bottom right image uses the* San Miguel *model, voxelized at* $(64k)^3$, *where we first solidified, then carved a hole in a column.*

Abstract
Voxels are a popular choice to encode complex geometry. Their regularity makes updates easy and enables random retrieval of values. The main limitation lies in the poor scaling with respect to resolution. Sparse voxel DAGs (Directed Acyclic Graphs) overcome this hurdle and offer high-resolution representations for real-time rendering but only handle static data. We introduce a novel data structure to enable interactive modifications of such compressed voxel geometry without requiring de- and recompression. Besides binary data to encode geometry, it also supports compressed attributes (e.g., color). We illustrate the usefulness of our representation via an interactive large-scale voxel editor (supporting carving, filling, copying, and painting).

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

1. Introduction

Compressed sparse voxel structures have gained popularity as an alternative representation for highly-detailed geometry. Voxel-based approaches encode the scene as a high-resolution grid, where cells ("voxels") hold information to define the scene. While naively storing such grids is infeasible for large resolutions, hierarchical representations can exploit sparsity [Mea82] and similarity [KSA13, JMG16, KRB*16] to achieve significant compression rates. Kämpe et al. [KSA13] demonstrate $(128k)^3$ resolutions at less than 1GB, while still enabling real-time rendering of this compressed form. Originally, such *DAG* (directed acyclic graph)-based structures only encoded solid geometry; however, later works [DKB*16, DSKA18] extend the methods to include compressed per-voxel attributes, such as color.

Until now, sparse voxel DAG structures are pre-built in a separate, often off-line, construction step, which limits their use to static elements. We introduce a solution to dynamically modify sparse voxel

† victor.careil@ens-rennes.fr; {m.j.billeter;e.eisemann}@tudelft.nl

DAGs in the compressed domain. It enables interactive geometry edits, and paves the way to many applications requiring dynamic changes to a compressed DAG. Furthermore, our data structure is compatible to existing attribute-compression schemes [DSKA18]. We demonstrate interactive color manipulations as an example. Specifically, we make the following contributions:

- A method for modifying sparse voxel DAGs in their compressed form, i.e., without decompression/recompression;
- A novel data structure to enable interactivity and ensure little impact on compression/rendering/traversal performance;
- A natural recording of changes to enable undo/redo operations and associated garbage collection to free memory, when needed.

2. Related Work

Voxels are used in many applications to represent complex geometry. The regularity of voxel structures makes them appealing if opting for dynamic modifications, as in editing tools [3DC, SVR], as well as games [Mc, Ast]. Nevertheless, uncompressed voxel structures grow rapidly in size with increasing resolution.

Compressing voxel volumes is an option for reducing the memory costs. While several rendering algorithms have been proposed [BRGIG*14], solutions for fast editing of large volumes are lacking. Our work focuses on the modification of sparse structures that can, e.g., arise when converting solid geometries into very high-resolution voxel data.

Sparse voxel octrees [Mea82] (SVOs) encode sparse volumes efficiently by employing a hierarchy. Meagher [Mea82] encodes geometry in an octree, where leafs are either filled or empty, while inner nodes are either empty or contain at least one pointer to a non-empty child node on the next level. By only storing non-empty parts of the structure, SVOs deal very well with sparse data (as their name indicates). SVOs can be directly traversed without decompression and efficiently rendered via ray tracing [LK10], which made them suitable for many applications, including indirect illumination [CNS*11], multi-scale editing [SVR] and out-of-core rendering [CNLE09, CNSE10, GMI08].

Kämpe et al. [KSA13] show that the compression rates of SVOs can be increased significantly by also exploiting similarities in the actual geometric data. Their method identifies identical subtrees in an SVO and removes duplicates by changing all references to point to a single instance of the subtree. This transforms the octree into a directed acyclic graph (DAG). Using this method, Kämpe et al. encode volumes with resolutions of up to $(128k)^3$, while remaining compact (e.g., the structures fit fully into GPU memory) and maintaining real-time rendering performance. Later work increases compression rates by exploiting additional similarities [JMG16, JMG17], but we base our method on the simpler (but still very compact) original sparse voxel DAGs. Sparse voxel DAGs have found various applications, including shadows [SKOA14], many-view rendering [KBLE19], and time-varying geometry [KRB*16]. In the latter, Kämpe et al. use DAGs with multiple roots. Following a different root results in a different geometric representations, while identical data is still shared. The total compression increases significantly compared to storing separate DAGs. We use a similar insight to store multiple states (before and after edits) efficiently.

The original sparse voxel DAGs only encode geometry. Dado et al. [DKB*16] and Dolonius et al. [DSKA18] show how additional information, such as colors and other attributes (e.g., normals), can be compressed alongside a sparse voxel DAG. Our method is compatible with such approaches, and we demonstrate the use of colors, employing the compression method of Dolonius et al.

A sparse voxel DAG can be built efficiently from a sparse voxel octree [ABD*18, SKOA14]. Many methods for building sparse voxel octrees exist, including out-of-core approaches [BLD13]. Nevertheless, while these aim at efficient ground-up builds, we focus on efficient online modifications of a sparse voxel DAG structure.
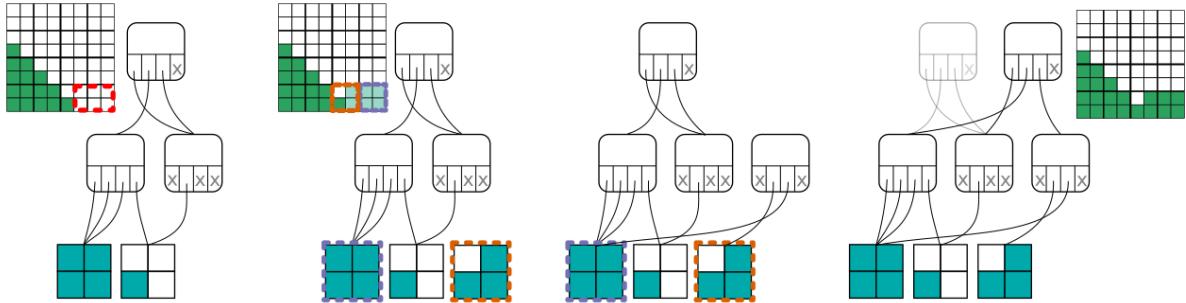
3. Method

Modifying uncompressed voxel structures is straight-forward (and indeed one of its strengths). However, compressed structures, such as Voxel DAGs, are largely considered static. The reason is that any node in a DAG may be referenced by multiple parents. Thus, naively changing an existing node potentially affects many parts of the scene. To avoid this problem, our method keeps the original structure intact but attaches additional information to indicate modifications. These additions are light-weight as they take place directly in the compressed domain, while maintaining an efficient traversal (and, thus, rendering) performance. Further, this encoding keeps track of all modifications, which can then be naturally un- and redone.

To explain our method, we first describe the principle of modifying the geometry in a DAG (Section 3.1) without discussing implementation details. We then introduce our data structure to achieve interactive performance (Sec.3.2). Next, we cover several extensions, including the use of attributes, demonstrated with color manipulations, e.g., for painting applications (Sec.3.3). At all time, geometry and color data remain compressed but the DAG is augmented with information in each modification step. A garbage collection is used to trim this information, when memory is required (Sec.3.5).

3.1. Modifying DAG Geometry

We assume that modifications take place within a bounding volume (e.g., brush). Independent of the applied operation, we, thus, first identify the voxel locations that require changes. To do so, our method traverses the DAG structure in a depth-first manner. If a node is not affected by the change, we stop the traversal, otherwise, we descend into the children. During this traversal, it is possible to descend into nodes that did not exist in the original DAG structure and will be added and compressed on the fly. An example that requires a full descent would be a single voxel that is added at the highest resolution level. If a larger region is filled, an earlier stop might be possible when an entire subtree is filled.

Once the necessary depth and location for the manipulation is reached, we create a node representing the newly modified geometry (following Kämpe et al. [KSA13], DAG leafs are $4^3 = 64$ subvolumes encoded in bits of a 64-bit integer). At this point, we need to integrate the new node into the DAG. There are two cases; there already exists an identical leaf-subvolume node somewhere in the DAG, which means that we can use the existing version instead, or we need to append a new node to the DAG (without invalidating

Figure 2: *Overview of the editing process. (Left) Original voxel data and the corresponding DAG. We wish to fill the region highlighted in red. (Children are shown in counter-clockwise order, starting top-left.) (Center-left) We traverse the DAG structure and identify the leaf nodes affected by the modification (highlighted in orange and purple in the voxel data). The purple leaf already exists in the DAG, so we find it and return a pointer to the existing node. The orange leaf is new, so we append it to the leaf level. (Center-right) We propagate the pointers to the parent node, and construct it. The DAG does not contain such a node yet, so we create a new one. (Right) We propagate the new pointer to its parent (the root). Three of the parent's children are unmodified, and we copy the corresponding values from the old instance. The fourth child is our modified node. The new node does not exist either, and becomes a new root node that represents the modified volume. The old root is kept (unless trimmed later) and traversing the old root returns the original voxel data.*

existing nodes). In both cases, we obtain a pointer to a node within the DAG that contains the modified geometry. The pointer is then passed to the parent. In the special case, where a node ends up empty, we propagate a special (reserved) pointer value to the parent node. This instructs the parent to omit the node in its child mask and to indicate its absence (no pointer needs to be stored). For each parent, once its children have been processed, we again create a new node that reflects the changes. Such interior nodes consist of a bit-mask identifying non-empty children (eight bits) and a set of pointers (one per non-empty child). As in the previous case, the same two options exist: the same node already exists, and we use it instead, or we add the new node to the DAG. Again, either choice results in a pointer to a node in the DAG with the correctly-modified geometry, which is then passed again to the parent.

The procedure is repeated until reaching the root of the DAG, where we create a new root node. It represents the modified DAG, including the geometric changes. Please note that the trees attached to the previous root node reflect the previous version of the DAG, which enables un- and redo operations by storing such previous roots and starting traversals from them. Section 3.5 will show how to remove this unnecessary data, if needed. We illustrate the entire editing process in Figure 2. Listing 1 includes pseudo code for the recursive editing function. The efficiency of our method hinges on two operations: finding a node in the DAG and inserting new nodes. The next section describes how to achieve these tasks efficiently.

3.2. The HashDAG structure

Our data structure, the *HashDAG*, uses two main components: hashing for efficient finding of nodes and virtual memory to manage the overall memory usage. The DAG data structure contains many pointers, which would be impractical to update. We therefore design the HashDAG such that its data does not need to be moved. We achieve this through a setup that uses fixed, predetermined memory addresses. The address space is managed with a standard virtual memory setup, i.e., where fixed-size pages are mapped to com-

```
nodeptr edit( editOp, lvl, center, node )
{
  if( !editOp.affects_volume_at( center, lvl ) )
  │ return NodeUnchanged;

  if( lvl == DAGDepth-2 )
  │ return edit_leaf(editOp, lvl, center, node);

  childMask = 0; childPtrs[8] = { EmptyNode, ... };
  if( node != EmptyNode )
  │ childMask = get_child_mask( node );
  │ childPtrs = unpack_child_pointers( node );

  for( i = 0; i < 8; ++i )
  │ newChild = edit( editOp, lvl+1,
  │   position_of_child( center, lvl, i ),
  │   childPtrs[i] );
  │
  │ if( newChild != NodeUnchanged )
  │ │ childPtrs[i] = newChild;
  │ │ if( newChild != EmptyNode )
  │ │ │ childMask |= (1<<i); // set bit i to one
  │ │ else
  │ │ │ childMask &= ~(1<<i); // clear bit i

  if( childMask == 0 )
  │ return EmptyNode;

  childPtrs = pack_child_pointers( childPtrs );
  p = find_node( lvl, childMask, childPtrs );
  if( p == NodeNotFound )
  │ p = append_node( lvl, childMask, childPtrs );
  return p;
}
```

Listing 1: *Recursive editing function. The* `editOp` *defines the editing operation (i.e., encodes what changes should be made). Its method,* `affects_volume_at()`, *determines if the edit affects a certain volume, defined by node size at a specific level and position. Leaf levels are handled separately (*`edit_leaf()`, *not shown, computes a 4^3 leaf volume and finds/inserts it in the DAG). The key operations* `find_node()` *and* `append_node()` *are detailed further in Section 3.2.*

Figure 3: *One level of the HashDAG. In the figure, the level's buckets correspond to rows in the table. The index inside the bucket is given by the column. The level contains four nodes (colored differently). Each node starts with the child mask, immediately followed by the child pointers. The number of child pointers is equal to the number of set bits in the child mask. In this illustration, we use a 16-bit address space. Each bucket spans 256 consecutive 16-bit words. Combining the 8-bit address of a bucket (leftmost column) with the index (upper and lower bytes, respectively) in the bucket (top row) creates a virtual address. Nodes are identified by the address of their child mask. For example, the first node $(0101, p_0, p_1)$ has address 0x3200, and the third $(0100, r_0)$ receives 0x3303.*

pact ranges of addresses via an address translation table. Pages are only allocated when the memory at the corresponding addresses is needed. The HashDAG has no special requirements regarding the page size, making it compatible with, e.g., hardware supported virtual memory/"sparse resources".

The HashDAG allocates a fixed address space for the whole DAG. The size of the address space is linked to the size of the pointers in the DAG structure. In our case, we rely on 32-bit pointers, and consequently use a 32-bit address space. Note that addressing is performed on a 32-bit word granularity, meaning that the address space spans 16GB of memory. Each node occupies a number of consecutive 32-bit words: a node starts with a word containing the child mask at the first address (the 8-bit child mask is padded to 32-bits), and is followed by k words containing the child pointers. Here, k is equal to the number of non-empty children of that node, reflected by the number of set bits in the child mask.

Each level in the DAG receives a predetermined portion of the address space. Varying the size per level allows us to allocate less of the address space close to the root (which contain fewer nodes) and use more of it for the levels that typically contain a large number of nodes. By doing so, we maintain a more even distribution of nodes per amount of address space overall. In our implementation, we decided to keep these memory regions to sizes of a power of two. We denote the size of the memory region at level ℓ as 2^{N_ℓ}.

We further subdivide the regions of each level into fixed-size buckets of size 2^M. Hence, a level ℓ contains 2^P buckets, where $P = N_\ell - M$. We track the number of used 32-bit bit words for each bucket in a counter. Nodes are always placed in the bucket at the index given by a P-bit hash of the node's contents. The hash is computed using a standard hash function [App16], taking the child mask and associated child pointers (each of which is just a 32-bit value) as input. Figure 3 illustrates the address layout of a level.

When searching for a node during editing (to check for its existence or to find its address), we compute the hash of the node, and perform a linear search within the bucket identified by the hash. A

node can be accessed via its virtual address v, computed by combining the hash, h, and the offset in the bucket i (see Figure 4):

$$v = \text{levelBaseAddr} + h \times 2^P + i.$$

The base address of the level, levelBaseAddr, is precomputed based on the selected distribution of the memory regions for the different levels in address space (i.e, from the N_ℓ). If the node is not found, it is appended to the corresponding bucket. Normally, the node is simply placed at the word following the final element of the last node in the bucket (or at index zero if no nodes are present yet). However, we ensure that nodes do not span across multiple pages: if a node does not fully fit into the remaining space of the last page of the bucket, we allocate a new page, and move the node to the beginning of that page. The remaining words in the old page are set to zero. The counter of the bucket is incremented by the amount of potentially skipped words plus the size of the node.
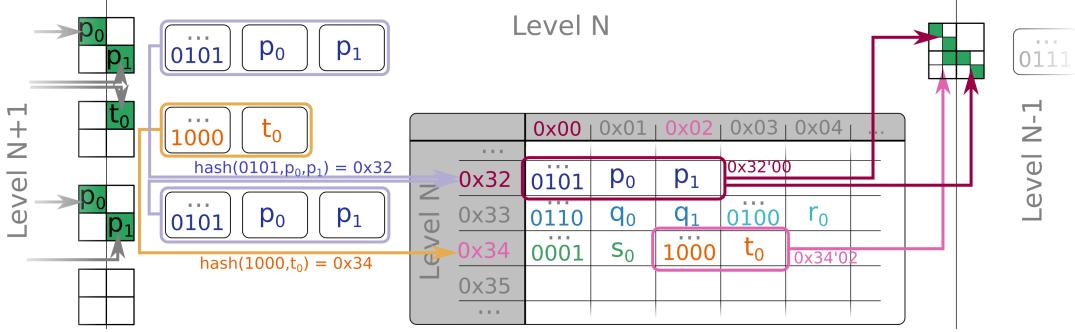
Since nodes cannot span multiple pages, we only need to perform one address translation per node during traversal. Similarly, during editing, the translated address is needed as part of the get_child_mask() and unpack_child_pointers() operations and could be performed just once for both accesses.

A key property of the HashDAG is that the hashing is only performed when editing the structure. During traversal, however, no hashes need to be computed, limiting the overhead of the HashDAG to that arising from the extra indirection due to the virtual memory.
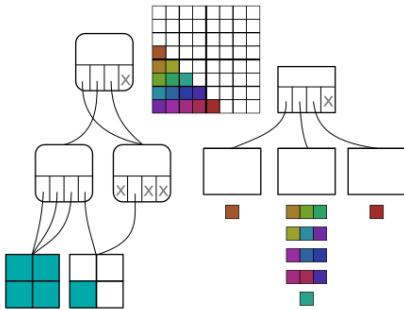
3.3. Attribute encoding

Our method maintains the structure and topology of the DAG, which keeps it compatible to previous methods that integrate attributes, such as colors. Attributes are typically stored as a supplementary array that is compressed independently and each node in the DAG makes use of its index (which is computed during a depth first traversal) to look up the corresponding attribute. Several of the proposed attribute compression schemes are relatively efficient when involving few attributes but can become costly for larger sizes. To avoid these costs, we split our scene in chunks of a fixed size (corresponding to subtrees of the DAG at a fixed depth, e.g., 10). The attribute compression can then be applied to each of these subtrees independently. Consequently, a local change only requires us to locally update the corresponding subtree (see Figure 5). The calculation of node indices is compatible with our geometry-modification method, as we also rely on a depth-first traversal.

To exemplify the possibility of manipulating attributes, we employed the color compression method by Dolonius et al. [DSKA18]. The compressed color data is stored for each subtree of fixed depth. When traversing the DAG, we first find the corresponding color chunk and then decode the color as in [DSKA18]. When editing, the affected color chunks are rebuilt entirely: color data from unchanged voxels in the same subtree is transferred to the new chunk in compressed form (and voxel indices are updated as needed), as well as colors from new voxels that are added into the color chunk. For details on the actual compression technique, which relates in parts to block compression methods such as S3TC/DXT/BCn [BC18], the interested reader is referred to their original article.

Figure 4: *Editing with the HashDAG structure. The figure illustrates the process at Level N; this is the same level as shown in Figure 3. The edit updates the structure shown there. Level N receives the child pointers $p_{0,1}$ and t_0 from Level $N+1$ (c.f. recursive calls in Listing 1). We construct the child masks by checking the returned pointer values. For each of the three non-empty nodes, we hash the child mask and the pointers. This identifies the buckets in which the nodes will reside. The first and third nodes (blue) are identical and a linear search will find them at index* `0x00` *of the hash table. Combining the bucket's address (*`0x32`*) with the index gives the virtual address of the node,* `0x3200`*. The second node (orange) is not found in bucket* `0x34`*, so we append it to the first possible location, index* `0x02` *here. The final virtual address becomes* `0x3402`*. The virtual addresses are returned to Level $N-1$, where the process repeats.*

Figure 5: *We subdivide the scene into chunks that correspond to subtrees of the geometry DAG at a fixed depth (=1 in this figure). Each chunk applies the color compression of Dolonius et al. [DSKA18] independently. The compressed color chunks are stored in a separate SVO. The figure shows the color data uncompressed for simplicity; we refer to the original publication for details on the compression.*

3.4. Optimizations and implementation details

We noticed that many manipulations lead to locally constant voxel volumes. A special handling of these cases leads to a significant increase in performance when adding or clearing large solid volumes. Specifically, during initial loading, we ensure that a node representing a completely full subtree exists at all levels (this adds at most one node to each level of the DAG structure), and keep track of its location. During editing, we perform an additional test that checks whether or not the modification would completely fill or empty the node in question. If it is completely filled, we simply return the pointer of the well-known full-subtree node of the corresponding level. If it is empty, we already handle the situation efficiently, as the empty volume is simply recorded by indicating an absent child node in the parent's child mask. A similar optimization also works for solid colors, but varying colors require the generation of the corresponding compressed color data.

Our implementation supports rudimentary multithreading. We perform a single-threaded traversal until the color chunk depth. Any continued traversals become independent jobs that are handed over to a thread pool of fixed size. For the HashDAG structure, we use a mutex per bucket. Color updates do not need synchronization, since each color chunk is being rebuilt by a single job.
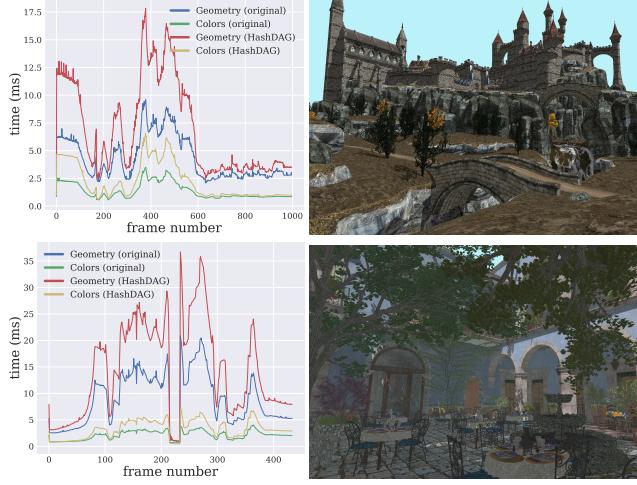
3.5. Garbage collection

As indicated in earlier sections, each modification adds new data to the DAG structure, while the old versions of the DAG structure remain valid. While this may be useful in some applications, the recovery memory by removing the old versions may be desirable in some cases.

We apply a garbage collection method to trim the information. We start with a list of root nodes that we wish to keep (for example, the root node representing the state after the latest modification). We iterate over the nodes and extract a list of unique child pointers to the next level. We repeat this procedure with the new list, until we reach the leaf level. By creating the unique list for each level, the method scales with the number of DAG nodes that we want to keep, and thus remains relatively efficient.

Next, we iterate over the DAG levels, starting at the leaf level. We compact each level to only contain the nodes identified in the previous step, while maintaining a mapping from the original index to the new index in the compacted list. The mapping is used at the next higher level to update the child pointers to their new values.

4. Results and Discussion

To illustrate our approach, we produced a prototype, in which users can load and interactively edit an existing voxel DAG model. Our implementation performs modifications on the CPU and renders the DAG structures on the GPU, using CUDA. We ran measurements on an Intel Core i7-8700K CPU (6 cores with $2\times$ hyperthreading) with 16GB of DDR4 memory and a NVIDIA GeForce RTX 2080 GPU under Linux.
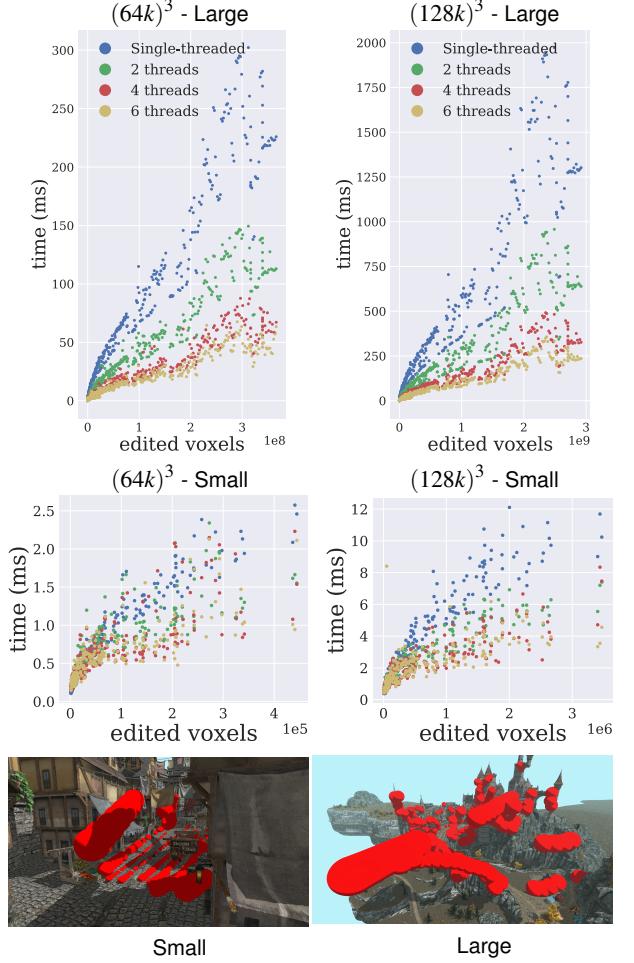
Figure 6: *Overheads from the virtual memory and the color SVO. We show the rendering performance in a short predefined fly-through through each of the scenes, for our custom DAG structure as well as a standard DAG without virtual memory. The top row shows results for the* Epic Citadel *scene, at a* $(128k)^3$ *resolution. The bottom row shows the* San Miguel *scene, at* $(64k)^3$. *Rendering performance is on average about* $1.5\times$ *slower due to the virtual memory, with a worst case of about* $2\times$ *slowdown. Resolving colors adds overheads of similar magnitude. Tracing geometry and resolving colors are separate passes, the full rendering time is the sum of both. Note that the irregular tree model in San Miguel is quite expensive to render, making it more expensive despite the lower resolution.*

Figure 6 compares the traversal performance of our HashDAG structure with the original DAG implementation. Our virtual memory scheme introduces an overhead of $1.5\times$ to $2\times$ for the raytracing of the geometry (although the performance already varies greatly for different scenes). Resolving attributes requires an additional traversal of an SVO to locate the chunk associated with the intersected node. In case of our implemented solution for colors, it introduces an overhead of similar magnitude. We believe the raytracing overhead could be mitigated by using sparse resources [Vk19] - hardware support for user-controlled virtual memory mappings. However, at the time of writing, sparse resources are unsupported in CUDA and such an implementation remains future work.

Our prototype includes several editing tools; copying parts of the existing voxel structure, adding and removing (carving) voxels in regions of different shapes, painting colors and a non-optimized tool for filling hollow spaces (see Figure 1). We added the latter only for visualization purposes because voxel DAG structures are often created from traditional meshes resulting in thin shells, as only the surfaces are voxelized. Filling such shells before carving into them ended up being useful for visual purposes (Figure 1, bottom right). To solidify such objects, we perform a flood fill operation that marches over the voxel grid from the user selected position.
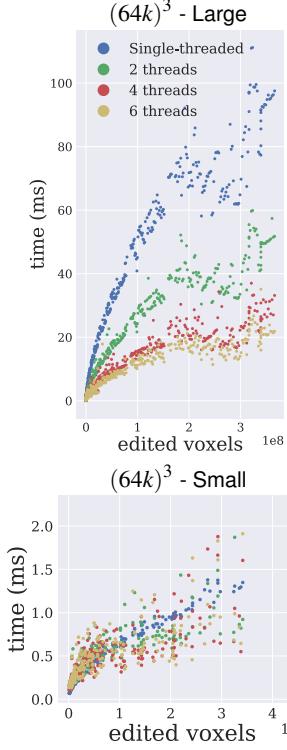
To evaluate the performance of our method, we focus on simple editing operations and group the edits by the number of voxels they change. More complex operations (such as copies) carry overheads unrelated to the presented method (e.g., traversing the copy's

Figure 7: *Performance of geometry and color modifications w.r.t. size of the modification in voxels, using varying numbers of threads. We present data from two separate editing sequences, both in the* Epic Citadel *scene at* $(64k)^3$ *and* $(128k)^3$ *resolutions. The first sequence includes smaller modifications* ($< 350k$ *voxels in* $(64k)^3$ *and* $< 2.5M$ *in* $(128k)^3$). *The second sequence performs larger modifications affecting up to* $350M$ *voxels in* $(64k)^3$ *and* $3G$ *in* $(128k)^3$, *respectively. The images at the bottom show the scenes after the modifications from the sequences (small edits to the left, large to the right).*

source). Figure 7 shows the performance of modifications of varying sizes. We include results from two test sequences at two resolutions, $(64k)^3$ and $(128k)^3$. The two test sequences are repeated for both resolutions such that sizes of the modifications in world space are maintained (i.e., the same operation will affect many more voxels in the higher resolution scene). The first test focuses on small modifications (affecting up to $350k$ and $2.5M$ voxels, respectively), the second on large edits, affecting up to $350M$ and $3G$ voxels.

Figure 8 repeats the same tests, albeit for geometry only (i.e., the color updates are omitted). Here, the effect of the full-subtree optimization (Section 3.4) becomes obvious, as the time required for modifications of increasing sizes scales sublinearly. For large
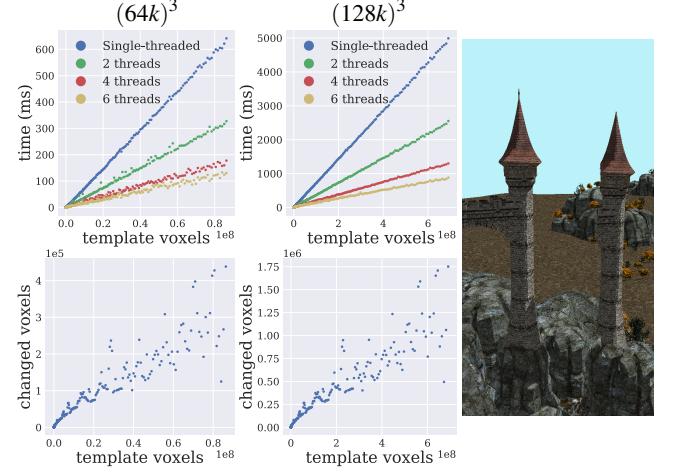
Figure 8: *Performance of geometry-only modifications w.r.t. size of the modifications in voxels, using varying numbers of threads. The edit-sequences are identical to those in Figure 7.*

Figure 9: *Performance of copy operations with different source volume ("template") sizes (top row). In contrast to the modifications from Figure 7 that add solid volumes, the copy operations operate on very sparse (but irregular) templates (bottom row). Thus, very few voxels actually change, and most of the time is spent on evaluating the template. The copies become entirely CPU bound. (Right) Together the operations construct the tower to the right.*

edits, our method scales well with additional threads up to a certain limit (four on our machine), at which point adding additional threads yields only minor increases in performance. We suspect that the method becomes memory bound at that point. For small edits, multithreading helps little, as the edits are smaller than the subtrees we parallelize over (and any small gains are likely offset by additional overheads related to the multithreading).

The combination of geometry and color modification is more costly than geometry-only edits. For color edits, we also observe a much larger variation in performance. This is a consequence of varying workload. When adding a new voxel in empty space with no pre-existing surrounding geometry, the new color chunk only contains the color of the added voxel. However, if the voxel is added in the vicinity of other geometry, building the color chunk involves transferring existing color data from the surrounding geometry and adding the new colors as well.

Figure 9 displays results from a sequence of copy-operations. The goal of this tool is to emulate sparse and highly irregular edit operations (in itself it is not intended to be a very efficient implementation of a copy tool). The copy operations are implemented in two phases. First, we create a template by decompressing the source volume. Second, we apply our method, using the template as a guide with which voxels are modified or left as-is (empty voxels in the source are left as is). We only measure performance for the second phase. For instance, the full- and empty-subtree optimizations never apply, as we do not store hierarchical information in the template.
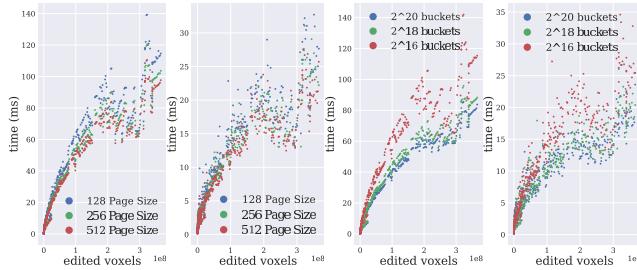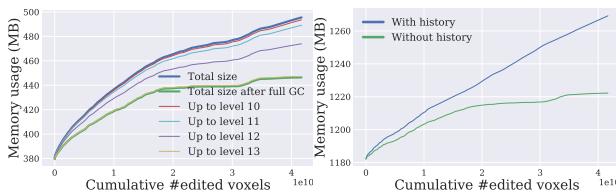
Note that the source volume's colors are decompressed in phase one as well. We do not optimize the copies by transferring already compressed color data, again with the goal of demonstrating the efficient manipulation, not an efficient tool. For variants on the editing operations and additional tests, we refer the interested reader to the supplementary material.

Both page size and the number of buckets per HashDAG level affect performance. By default, we use a page size of 512 together with 2^{16} buckets on lower levels (≥ 10) of the DAG. Levels close to the root need fewer buckets (we use 1024), which allows us to reduce memory demands. In Figure 10, we show performance with respect to page size and bucket count. In general, increasing either page size or bucket count improves performance, but comes at the cost of a higher memory overhead. The memory overhead stems from allocated but only partially filled pages. A larger page size increases the unused space for each partially filled page, and a larger bucket count increases the number of partially filled pages. The exact memory overhead varies over time. However, immediately after loading the HashDAG structure (*Epic Citadel* at $(64k)^3$, weighing in at 380MB of data), we observed memory overheads of approx. 105MB (128 word page size), 260MB (256 words) and 550MB (512), using the default bucket count of 2^{16}. Tests with increasing bucket counts use a page size of 128, as to conserve memory and enable large bucket counts. Overheads are approx. 105MB (2^{16} buckets), 480MB (2^{18}) and 1165MB (2^{20}). Additional memory overheads from page padding to avoid placing nodes in multiple pages are negligible in comparison ($\lesssim$ 20MB for $(128k)^3$ with 2^{16} buckets and page size 128, which is the worst case for the page padding in our set of tests).

The quality of hash functions is also critical to the performance of the HashDAG. Uneven distribution of nodes across the buckets would quickly lead to problems, first negatively impacting performance, and ultimately filling up some buckets completely, prevent-

Figure 10: *Performance for different page sizes and bucket counts (geometry only, for the large edits sequence from Figure 7). Page size and bucket count are two of the main variables that control the performance of the HashDAG. We show both unthreaded (left figure of each pair) and threaded (right figures) performance. Performance increases with larger page sizes, but this also results in a larger unused memory overhead from partially filled pages. Increasing the number of buckets has a similar impact (the test was performed with page size equal to 128).*

Figure 11: *Memory use over the large edits sequence (Figure 7) for the* Epic Citadel *scene at* $(64k)^3$*. (Left) Memory usage by the geometry. The graph includes total memory usage with and without garbage collection (GC), as well as performing partial garbage collection of levels zero to the specified number. (Right) Memory usage for color data in the same sequence. We show usage both with and without keeping historical data for undo/redo support.*

ing further nodes from being added there. For interior nodes, we rely on the MurmurHash3 [App16] function. For leaf nodes, we simply use the bit mixing step of the MurmurHash3. For high levels, e.g. ≤ 5, many buckets are empty (and non-empty buckets contain on average very few nodes, typically in single digits). For lower levels of the HashDAG, we have always observed a normal distribution of the node counts per bucket. For example, at $(64k)^3$ with 2^{16} buckets, we have observed the following: buckets in level 12 (the level with the most nodes in this instance) contain on average approx. 600 32-bit words of data ($\sim$120 nodes), with a standard deviation of 60 words. The smallest bucket has 327 words, and the largest 903 - that is less than half of the maximum bucket size that we have reserved in this configuration (2048 words per bucket).

The reserved bucket size is a hard upper limit: we cannot change the size of the buckets' reserved address space at runtime. It is therefore necessary to ensure that the reserved space is large enough to enable future edits (up until a garbage collection step clears unused nodes). For very large data sets ($\gg (128k)^3$) or for very large amounts of changes, it is possible to use a full 32-bit address space per level, or even switch to a larger address space (and thus larger pointers), in order to ensure that there is sufficient space available in each bucket.

As indicated in the description of our method, old states of the structure remain valid and accessible via alternate root nodes (we optionally keep historical color data as well). In Figure 11, we show the memory usage over time, with an increasing number of changes (represented by the cumulative number of voxels changed so far). Additionally, we show the minimum size of the data, as if the garbage collection method were run after each modification. Memory usage still grows, which is to be expected, as new geometry is added with each modification. Historical color data, needed to support undo/redo operations, is relatively expensive. A limitation of our current implementation prevents us from freeing up the initial color data ($\approx$1.2GB), compressed using the method of Dolonius et al. [DSKA18], even as parts of it are replaced with new color chunks generated on the fly.

The garbage collection implementation is currently unoptimized, and doing a garbage collection in the *Epic Citadel* scene at $(64k)^3$ takes 25 seconds, of which 18 seconds are spent on the lower levels (≥ 12). The performance can certainly be improved, and we have identified significant overheads related to our usage patterns of standard containers (mainly `std::unordered_{set,map}`).

Our implementation suffers from other overheads that we have not presently addressed, and that are unrelated to our method. For example, transferring modified data from the CPU to the GPU for rendering is currently very expensive (order of tens of milliseconds), as it is implemented with a large number of small `cudaMemcpy` operations. A better approach would be to collect all small changes resulting from a single modification into a single buffer, transfer it, and use a CUDA kernel to distribute the small modifications to the GPU's copy of the data. However, this remains future work.

It is noteworthy that our solution requires only compute time in the order of milliseconds for scenes with 10^{15} voxels and edits of the size of several million voxels, including updates to an additional attribute in the form of color. We invite the reader to watch the accompanying video to experience an interactive editing session.

5. Conclusion and Future Work

Our solution enables interactive editing of an unprecedented scale. We efficiently modify very high resolution compressed sparse voxel structures, while avoiding de- and recompression cycles. This possibility is enabled via our HashDAG, a compressed structure that can be modified on the fly and is relatively easy to implement. Our prototype editor allows users to perform modifications to existing voxel structures at many different scales interactively on commodity hardware, while even keeping track of changes at a low additional memory cost. It maintains the traversal mechanism and access possibilities of existing hierarchical voxel grids, which makes it easy to integrate the solution in other applications, while offering additional compression and modification capabilities. One example could be a large-scale procedural-world generation, inspired by the work of Peytavie et al. [PGMG09], that allows for directly influencing the high-resolution terrains. The possibility of testing visibility efficiently in a voxel grid could also make it interesting to use the editor for performance-sensitive scene design [ED07].

Acknowledgements

References

[3DC] 3DCoat. https://3dcoat.com/, accessed 2020-02-26. 2

[ABD*18] ASSARSSON U., BILLETER M., DOLONIUS D., EISEMANN E., JASPE A., SCANDOLO L., SINTORN E.: Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to Pre-computed Shadows. In *EG 2018 - Tutorials* (2018), The Eurographics Association. doi:10.2312/egt.20181028. 2

[App16] APPLEBY A.: MurmurHash3, 2016. https://github.com/aappleby/smhasher/wiki/MurmurHash3, accessed 2020-02-26. 4, 8

[Ast] Astroneer - A Game of Aerospace Industry and Interplanetary Exploration. https://astroneer.space/, accessed 2020-02-26. 2

[BC18] Block compression (Direct3D 10), 2018. https://docs.microsoft.com/en-us/windows/win32/direct3d10/d3d10-graphics-programming-guide-resources-block-compression, accessed 2020-02-26. 4

[BLD13] BAERT J., LAGAE A., DUTRÉ PH.: Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference* (2013). doi:10.1145/2492045.2492048. 2

[BRGIG*14] BALSA RODRÍGUEZ M., GOBBETTI E., IGLESIAS GUITIÁN J., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: State-of-the-art in compressed GPU-based direct volume rendering. *Computer Graphics Forum 33*, 6 (2014). doi:10.1111/cgf.12280. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2009), I3D '09. doi:10.1145/1507149.1507152. 2

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing: A preview. In *Proceedings of the 2011 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2011), I3D '11. doi:10.1145/1944745.1944787. 2

[CNSE10] CRASSIN C., NEYRET F., SAINZ M., EISEMANN E.: Efficient rendering of highly detailed volumetric scenes with gigavoxels. In *GPU Pro*, Engel W., (Ed.). A K Peters, 2010, ch. X.3. http://maverick.inria.fr/Publications/2010/CNSE10. 2

[DKB*16] DADO B., KOL T. R., BAUSZAT P., THIERY J.-M., EISEMANN E.: Geometry and attribute compression for voxel scenes. *Computer Graphics Forum (Proc. of Eurographics) 35*, 2 (2016). doi:10.1111/cgf.12841. 1, 2

[DSKA18] DOLONIUS D., SINTORN E., KÄMPE V., ASSARSSON U.: Compressing color data for voxelized surface geometry. *IEEE Transactions on Visualization and Computer Graphics 25*, 2 (2018). doi:10.1109/TVCG.2017.2741480. 1, 2, 4, 5, 8

[ED07] EISEMANN E., DECORET X.: Visibility sampling on GPU and applications. *Computer Graphics Forum (Proc. of Eurographics) 26*, 3 (2007). doi:10.1111/j.1467-8659.2007.01076.x. 8

[GMI08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9 (2008). doi:10.1007/s00371-008-0261-9. 2

[JMG16] JASPE A., MARTON F., GOBBETTI E.: SSVDAGs: Symmetry-aware sparse voxel DAGs. In *Proceedings of the 2016 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2016), I3D '16. doi:10.1145/2856400.2856420. 1, 2

[JMG17] JASPE A., MARTON F., GOBETTI E.: Symmetry-aware sparse voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes. *Journal of Computer Graphics Techniques (JCGT) 6*, 2 (2017). http://jcgt.org/published/0006/02/01/. 2

[KBLE19] KOL T. R., BAUSZAT P., LEE S., EISEMANN E.: MegaViews: Scalable many-view rendering with concurrent scene-view hierarchy traversal. *Computer Graphics Forum 38*, 1 (2019), 235–247. doi:10.1111/cgf.13527. 2

[KRB*16] KÄMPE V., RASMUSON S., BILLETER M., SINTORN E., ASSARSSON U.: Exploiting coherence in time-varying voxel data. In *Proceedings of the 2016 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2016), I3D '16. doi:10.1145/2856400.2856413. 1, 2

[KSA13] KÄMPE V., SINTORN E., ASSARSSON U.: High resolution sparse voxel DAGs. *ACM Transactions on Graphics 32*, 4 (2013). doi:10.1145/2461912.2462024. 1, 2

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), I3D '10. doi:10.1145/1730804.1730814. 2

[Mc] Minecraft. https://www.minecraft.net/en-us/, accessed 2020-02-26. 2

[McG17] MCGUIRE M.: Computer graphics archive, 2017. https://casual-effects.com/data. 9

[Mea82] MEAGHER D.: Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 1 (1982), 85. doi:10.1016/0146-664X(82)90128-9. 1, 2

[PGMG09] PEYTAVIE A., GALIN E., MERILLOU S., GROSJEAN J.: Arches: a Framework for Modeling Complex Terrains. *Computer Graphics Forum (Proc. of Eurographics) 28*, 2 (2009). doi:10.1111/j.1467-8659.2009.01385.x. 8

[SKOA14] SINTORN E., KÄMPE V., OLSSON O., ASSARSSON U.: Compact precomputed voxelized shadows. *ACM Transactions on Graphics 33*, 4 (2014). doi:10.1145/2601097.2601221. 2

[SVR] SculptVR - Build your world with friends in ScultVR. http://www.sculptrvr.com/, accessed 2020-02-26. 2

[Vk19] Vulkan 1.1.123 - A Specification - Chapter 31 - Sparse Resources, 2019. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html, accessed 2020-02-26. 6