Rendering Large-Scale Environments Efficiently

Molenaar, M.L.

DOI
10.4233/uuid:89e15bc5-780e-4d2a-8bf6-ab074650de05

Publication date
2025

Document Version
Final published version

Citation (APA)
Molenaar, M. L. (2025). *Rendering Large-Scale Environments Efficiently*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:89e15bc5-780e-4d2a-8bf6-ab074650de05

Rendering Large-Scale Environments Efficiently

Mathijs Molenaar

Rendering Large-Scale Environments Efficiently

Mathijs Lucas Molenaar

Rendering Large-Scale Environments Efficiently

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof. dr. ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates
to be defended publicly on
Thursday 18, December, 2025 at 12:30 o'clock

by

Mathijs Lucas MOLENAAR

Master of Science in Computer Science, Utrecht University, the Netherlands
born in Amsterdam, the Netherlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof. dr. E. Eisemann,	Delft University of Technology, *promotor*
Dr. R. Marroquim,	Delft University of Technology, *copromotor*

Independent members:

Prof. dr. P. César Garcia	Delft University of Technology
Prof. dr. U. Assarsson	Chalmers University of Technology
Dr. M. Billeter	University of Leeds
Dr. J. Bikker	Breda University of Applied Sciences
Prof. dr. R. Hanssen	Delft University of Technology

Keywords: Voxel, Sparse Voxel Directed Acyclic Graph, Rendering, Data Compression
Printed by: ProefschriftMaken
Cover by: Mathijs Molenaar

An electronic copy of this dissertation is available at
https://repository.tudelft.nl/.

Contents

Summary

Despite the exponential growth in computational power, displaying highly detailed scenes remains challenging due to memory and performance limitations. One method of representing scenes is using voxels; the tiny cubic cells of a regular grid. In this dissertation, we are concerned with the Sparse Voxel Directed Acyclic Graph (SVDAG) data structure and how it can be used to improve large-scene rendering. The SVDAG is a hierarchical data structure that aims to reduce memory usage by identifying geometric redundancy. We address several challenges regarding this type of voxel compression.

Chapter 3 introduces a general framework for identifying geometric redundancies under a specific set of geometric transformations such as mirror symmetries and 90-degree rotations along the primary axis. In addition, it presents a novel algorithm for finding redundancy after applying translations. These transformations are stored compactly using a novel pointer encoding, resulting in additional compression over a regular SVDAG.

In Chapter 4 we address the challenge of real-time voxel editing by developing a GPU-accelerated editing framework for SVDAGs. This work relies on a GPU hash table, and we present and evaluate various implementations. Our results demonstrate that compact voxel structures, previously limited to static scenes, can now be efficiently modified and updated in real time.

Chapter 5 focuses on voxel scenes where each voxel carries associated attributes (such as color). To maintain high performance during editing, we propose two attribute compression algorithms, one lossless and one lossy, that are optimized for runtime speed rather than just memory savings. The approach separates geometry from attributes, allowing both to be compressed and edited independently, and integrates elegantly into existing SVDAG editing pipelines.

Finally, Chapter 6 proposes a technique to accelerate out-of-core path tracing through conservative ray batching using geometry proxies derived from SVDAGs, without any loss in image quality. The memory compact voxel geometry provided by SVDAG compression helps to significantly reduce the amount of geometry data that must be loaded from disk, accelerating the rendering of large scenes that do not fit entirely in system memory.

Samenvatting

Ondanks de exponentiële groei in computer rekenkracht blijft het weergeven van zeer gedetailleerde scènes een uitdaging vanwege zowel geheugen- als prestatiebeperkingen. Een mogelijke methode om scènes weer te geven is met behulp van voxels: de kleine kubusvormige cellen van een driedimensionaal raster. In dit proefschrift focussen wij ons op de Sparse Voxel Directed Acyclic Graph (SVDAG) datastructuur, en hoe deze gebruikt kan worden om de weergave van grote en gedetailleerde scènes te verbeteren. De SVDAG is een hiërarchische datastructuur die geheugengebruik vermindert door herhalende patronen te identificeren. Wij behandelen verschillende uitdagingen met betrekking tot dit type voxelcompressie.

Hoofdstuk 3 introduceert een algoritme voor het identificeren van geometrische herhaling na toepassing van een specifieke set van geometrische transformaties zoals spiegelsymmetrieën en rotaties van 90 graden langs de primaire assen. Daarnaast wordt een nieuw algoritme gepresenteerd voor het vinden van redundantie in de vorm van verschoven voxels. Deze transformaties worden compact opgeslagen met behulp van een nieuwe pointer codering, wat resulteert in extra compressie ten opzichte van een gewone SVDAG.

In Hoofdstuk 4 behandelen we de uitdaging van het modificeren van voxel modellen in real-time door middel van een methode die de videokaart gebruikt. Dit werk is gebaseerd op een GPU hash tabel; wij presenteren en evalueren verschillende implementaties hiervan. Onze resultaten tonen aan dat compacte voxelstructuren, die voorheen beperkt waren tot statische scènes, nu efficiënt kunnen worden bewerkt.

Hoofdstuk 5 richt zich op voxel scènes waar elke voxel geassocieerde attributen heeft zoals kleur. Om hoge prestaties te behouden tijdens het bewerken, stellen we twee attribuut compressie algoritmen voor: één lossless en één lossy, die geoptimaliseerd zijn voor runtime snelheid in plaats van alleen geheugengebruik. De aanpak scheidt geometrie van attributen, waardoor beide onafhankelijk gecomprimeerd en bewerkt kunnen worden, en integreert elegant in bestaande SVDAG bewerking programma's.

Ten slotte wordt in Hoofdstuk 6 een techniek voorgesteld om out-of-core path tracing te versnellen door middel van conservatieve ray batching met behulp van geometrische proxies afgeleid van SVDAGs, zonder dat dit leidt

tot kwaliteitverlies. De compacte voxel geometrie die wordt opgeslagen middels SVDAG compressie helpt om de hoeveelheid data die van de harde schijf geladen moet worden te reduceren. Daarmee wordt het renderen van grote scènes die niet volledig in het systeem geheugen passen versneld.

1

Introduction

The first full-length movie to be entirely generated by a computer was Toy Story, released in 1995 by Pixar Animation Studios. Although multiple studios had previously released computer animated shorts, this was the first to span a runtime of over one hour. It was a marvel of engineering, since none of the (software) tools that we would use today were available. Making this movie required developing and researching into many different software tools that we now take for granted. By today's standards, the movie does look dated. The lighting is flat, the 3D models simplistic, and the animation wooden. However, this was all a limitation of the available compute power at the time.

After multiple decades of exponential improvements in hardware capabilities, our computers have become multiple orders of magnitude faster than those used to develop early 3D movies. Indeed, even a simple smartphone contains more computational power than the supercomputers used back in the day. However, the availability of such enormous amounts of computational power has not (yet) solved all fundamental problems in computer graphics. With every increase in the number of computations we can perform, we raise the bar in terms of the visual fidelity that we expect the computer to generate. Not only do moviegoers now expect realistic lighting (i.e. path tracing), but the virtual worlds in which we immerse ourselves have become larger and more detailed with every new release. As a result, movie studios actively employ their own supercomputers to store and render their highly detailed scenes. In that sense, nothing has changed, and rendering the pretty images we see on the silver screen may still take weeks of number crunching.

A very similar development has taken place in the real-time entertainment domain (gaming), although spread over an even larger timescale. In 1958 William Higinbotham released the first computer game called Pong. A simple two-dimensional tennis game in two dimensions, where the ball

consists of a single square and the tennis rackets of a rectangle. By the 2000's animation studios like Pixar were producing engaging stories realized in a fully computer-generated world, and similar developments also took place in the gaming industry. With computers becoming more affordable, people were now able to run three-dimensional video games at home using affordable gaming consoles. However, the quality of the virtual worlds was clearly lacking behind their movie counterpart as it is hard to feel an emotional connection to Hagrid displayed by the PlayStation 1. This is obviously to be expected, as the images in the virtual worlds must be produced in real-time on hardware that a regular consumer can afford. By now, almost 25 years later, the hardware running these games has accelerated by orders of magnitude, and as a result, modern video games look more lifelike than ever. But, just as in movies, we are still limited by both computational power and storage requirements. While developers want to immerse their players in a highly detailed and large virtual world, the reality is that storage space is limited, and this problem does not seem to be going away.

Virtual worlds can be represented digitally in various ways. The most common solution is to model only an object outside, doing so with an infinitely thin surface. These surfaces are most often stored explicitly, using basic primitives such as triangles or quads. However, they can also be stored implicitly, for example, as a signed distance field. Alternatively, a volumetric approach encodes a thin outer shell of the surface and *optionally* the inner structure as well. Volume data structures can be classified as either unstructured or structured. An example of an unstructured data structure is a complex of tetrahedrons. It adapts well to local details, but can be difficult to construct and operate on. In contrast, a structured grid is easy to process and, as we will see, can be highly compressed, allowing for very high voxel grid resolutions.

In this dissertation, we aim to address the storage concerns of representing and displaying very large and detailed virtual worlds with the use of Sparse Voxel Directed Acyclic Graphs, or SVDAGs for short. The SVDAG data structure allows for very compact storage of (binary) voxel scenes. The word "voxel" is a three-dimensional extension of the concept of a pixel (or texel); it represents a single cube-shaped cell on a regular three-dimensional grid. Conceptually, one might think of it as the cells of a checkerboard but extended into the third dimension. We will propose several improvements to existing SVDAG research in order to achieve better compression (Chapters 3 and 5), real-time modifications (Chapter 4), and show a novel use case (Chapter 6).

2

Background

The concept of voxels is certainly not new. These nicely aligned cubes have been used for many different use cases, such as physics simulations [1, 2], medical imaging [3], real-time rendering [4–6], offline rendering [7], 3D reconstruction [8, 9], 3D painting[1, 10–12], and 3D printing [13]. We can categorize these into two different types: *dense* and *sparse* voxel grids. In this dissertation, we solely focus on the latter *sparse* voxel grids, how to store them effectively, and how to apply them to accelerate rendering.

In a *dense* voxel grid, each voxel is "occupied" and assigned some value. An example of this is some forms of medical scans, where each voxel stores the density of the biological material. In contrast, a *sparse* voxel grid assumes that most voxels are empty, they do not store any value. This type of data is often found in real-time rendering [10], where the voxels are used to represent the *surfaces* of objects, but not their interiors. As mentioned above, in this dissertation, we focus exclusively on *sparse* voxel grids.

From a high-level conceptual overview, computer memory is presented as a one-dimensional array of numbers. Thus, storing any two- or higher-dimensional data set requires a mapping to the one-dimensional domain (e.g. $f : \mathbb{N}^N \to \mathbb{N}$). Due to its simplicity and computational efficiency, the most common solution is to collect all rows of our grid and concatenate them into a single array. The downside of this technique is that we assign a memory address also to the empty voxels, wasting a lot of memory. This generates the need for more complex data structures that account for the sparsity of a data set.

2.1. Related Work

Various methods have been proposed for compactly storing *sparse* voxel grids. For example, the spatial hash [14, 15] stores all nonempty voxels

in a hash table. Spatial hashing has recently been used to accelerate real-time light transport [16]. A special hash function is used to map the three-dimensional positions of the voxel to a single number ($\mathbb{N}^3 \to \mathbb{N}$). Testing whether a voxel is empty requires a hash table look-up which has $\mathcal{O}(1)$ amortized time-complexity; however, the time-complexity in the worst case is $\mathcal{O}(N)$. *Perfect* spatial hashing [17] aims to solve this by guaranteeing constant-time accesses using a *perfect* hash function. However, a downside of this approach is that it takes a considerable amount of computation to find such a *perfect* hash function.

N-wide trees provide another effective storage solution as was shown by Crassin *et al.* [5], who use it to build an out-of-core GPU renderer with multiple levels of detail. OpenVDB[2] combines spatial hashing and wide trees into a single structure, providing improved data locality when iterating through spatially close voxels. OpenVDB was originally developed for the CPU, but was later ported to the graphics card[18, 19]. Recently, Kim, Lee, and Museth [20] achieved significant memory reductions over OpenVDB by replacing the lower parts of the tree with a neural representation.

2.2. SPARSE VOXEL OCTREE (SVO)

A commonly used specialization of these trees is the Sparse Voxel Octree. Starting with the entire N^3 voxel grid, the domain is subdivided into eight equal-sized octants of $\left(\frac{N}{2}\right)^3$. Each nonempty octant is then subdivided again to create $\left(\frac{N}{4}\right)^3$ octants. This process is repeated until a predetermined leaf size is reached (typically 2^3 or 4^3), which are stored as dense grids.

During this process, we can construct a *tree* structure to represent the subdivisions. Initially, the entire N^3 voxel grid is represented by the *root node*, storing *pointers* to the eight $\left(\frac{N}{2}\right)^3$ child octants. If a child octant only contains empty voxels, then that child is empty, and the parent node stores an empty pointer. Since this is a common occurrence, programmers typically choose to encode empty octants using a bitmask, rather than storing explicit *null* pointers. Figure 2.1 provides an illustration of the two-dimensional equivalent of an octree. The exact same concepts apply to higher dimensions (e.g., 3D), but the two-dimensional example is easier to visualize.

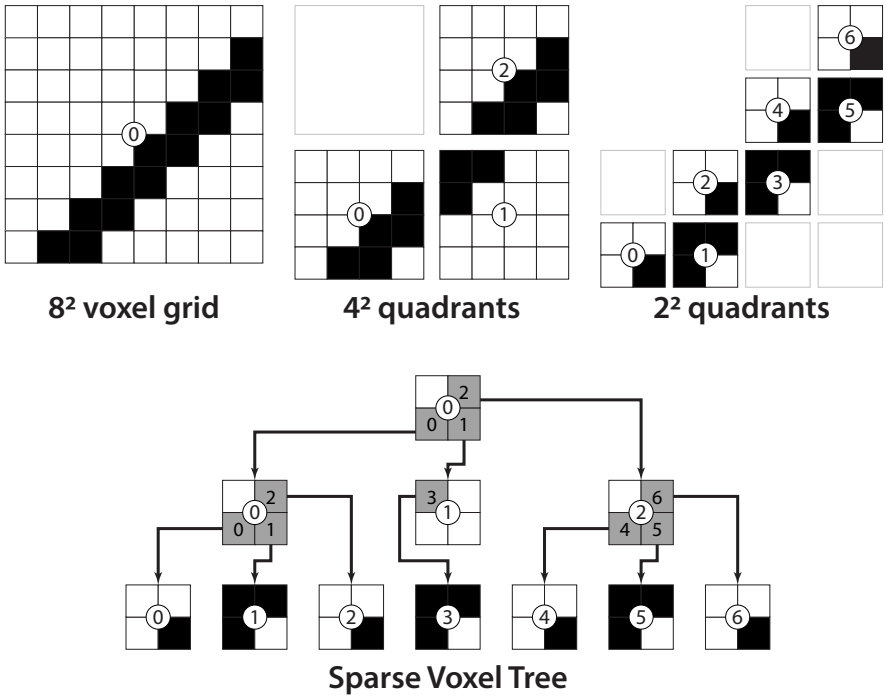


Figure 2.1.: Two-dimensional example of a sparse grid and its sparse voxel (quad)tree equivalent. Empty voxels are white, filled voxels are black.

2.3. Sparse Voxel Directed Acyclic Graph (SVDAG)

While a Sparse Voxel Octree (SVO) may reduce memory requirements by multiple orders of magnitude compared to a regular grid (for sparse scenes), it is not yet the most compact solution. Instead, the Sparse Voxel Directed Acyclic Graph (SVDAG) provides a generalization of the SVO, which is typically more compact than its counterpart. The concept was initially presented by Webber and Dillencourt [21] in two dimensions and later applied in three dimensions by Parker and Udeshi [22]. Research interest grew when Kämpe, Sintorn, and Assarsson used SVDAGs to compactly store voxelized triangle models[23], after which they were used for precomputed shadows [6, 24] and segmentation volumes in medical imaging [3].

The core concept behind SVDAGs is to detect reoccurring patterns

and to subsequently eliminate this redundant information. Consider the two-dimensional example shown in Figure 2.1. It contains repeating geometry at multiple levels in the tree. At the 2^2 level, the quadrants $\{0, 2, 4, 6\}$ and $\{1, 3, 5\}$ store an equivalent pattern. The same holds for 4^2 quadrants 0 & 2. We update the tree by removing, for each group of matching quadrants, all but one instance; the other removed quadrants are replaced by a *pointer* to this remaining instance. This process is illustrated in Figure 2.2.

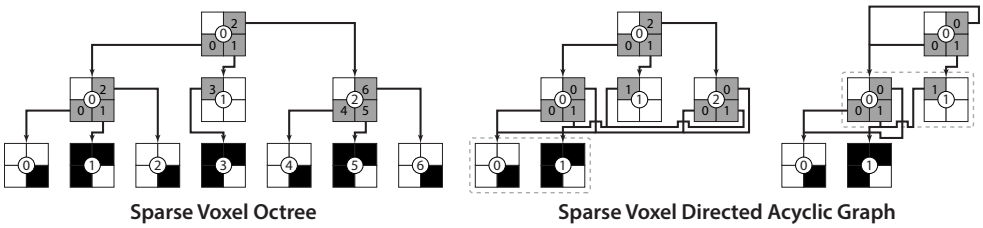


Figure 2.2.: Two-dimensional example of converting a Sparse Voxel Octree (SVO) to a Sparse Voxel Directed Acyclic Graph (SVDAG).

2.3.1. Efficient Construction

SVDAG compression can be applied to an existing Sparse Voxel Octree with low computational overhead. At a high level, the algorithm consists of iterating over all octants in a level and eliminating duplicates. The trick is to thus accelerate the process of searching for matching octants. This poses two sub-questions: How to quickly compare two subtrees? And how can we subsequently find matching subtrees without resorting to a (slow) linear search?

The answer to the first question is to perform the conversion process in a breadth-first bottom-up fashion. This means that we consider all equal-sized octants (e.g. N^3) at once, and we do so by starting at the leaves and working our way up the tree. Testing whether two leaves are identical is a trivial operation; we flatten the dense grids into an integer number. The resulting comparison requires only a single computer instruction.

By processing our SVO in a bottom-up fashion, we can now efficiently test whether the subtrees rooted at the parent nodes represent the same geometry. After processing the previous level, we know that if two child subtrees store the same geometry, then their pointers must also match. Thus, comparing a pair of inner nodes only requires a comparison between the eight child pointers.

The second challenge posed by SVDAG construction is how to find duplicates without iterating over all other subtrees. This is a field that has been well-researched in computer science. Sorting the leaves and nodes (per level) ensures that equivalent items are positioned consecutively in memory. Alternatively, one iterates over the octants and builds a dictionary of previously visited octants. This latter approach requires more (temporary) memory, but allows for faster updates, as we will see in Chapter 4.

2.3.2. Our Contributions

Sparse Voxel *Directed Acyclic Graphs* provide great memory savings over Sparse Voxel *Octrees* with few downsides. Their effectiveness is tied to how much geometric reuse can be discovered. As such, there is an incentive to not only consider exact matches, but to also search for octants that are geometrically similar [25] or equivalent under some transformation [26, 27]. In Chapter 3, we advance the field of SVDAG compression by proposing novel algorithms for efficiently finding matches under various geometric transformations. Combined with an efficient encoding scheme, we are able to beat the memory usage of the state-of-the-art.

While SVDAGs are often used for static scenes, some use cases require either modification or incremental construction of voxel scenes. As shown by Careil, Billeter, and Eisemann [28], SVDAGs can also be used in this context due to their fast construction process. Their framework implements SVDAG editing on the CPU which limits performance scaling. We address this issue in Chapter 4 by proposing a GPU-accelerated editing framework based on a novel GPU hash table design.

It is not uncommon for voxel grids to have some data associated with each nonempty voxel. For example, we might want to associate a material or a color with each voxel. When directly incorporated into the SVDAG, this means that two octants only match if both their geometry and associated voxel values are equivalent. Previous works [29–31] suggest a decoupling scheme that separates geometry from voxel attributes. The geometry is then stored in an SVDAG, while the attributes are compressed separately [30, 31]. Existing work on attribute compression focused solely on memory usage, resulting in algorithms that are too slow to be integrated into an SVDAG editing framework. In Chapter 5 we propose two compression algorithms with a focus on run-time performance and integrate them into an editing framework [28].

As mentioned above, voxel data have many uses beyond the direct

display of small cubes. The compactness of SVDAGs, and their ability to act as a ray tracing acceleration structure, make them a great tool for improving other rendering methods [3, 6, 24]. In Chapter 6, we propose a novel use of SVDAGs to accelerate out-of-core rendering. This application is very well suited to the SVDAG, as it requires a data structure that is both very compact and fast to intersect.

3

Transform-Aware Sparse Voxel Directed Acyclic Graphs

Mathijs Molenaar, Elmar Eisemann

Sparse Voxel Directed Acyclic Graphs (SVDAGs) have proven to be an efficient data structure for storing sparse binary voxel scenes. The SVDAG exploits repeating geometric patterns; which can be improved when considering mirror symmetries. We extend the previous work by providing a generalized framework to efficiently involve additional types of transformations and propose a novel translation matching for even more geometry reuse. Our new data structure is stored using a novel pointer encoding scheme to achieve a practical reduction in memory usage.

Parts of this chapter have been published in the Proceedings of the ACM on Computer Graphics and Interactive Techniques Volume 8, Issue 1 (2025)[32].

3.1. INTRODUCTION

In this chapter, we focus on improving a specific class of 3D voxel structures, namely *static* Sparse Voxel Directed Acyclic Graphs (SVDAGs). This representation, first suggested in a 2D image context [22] and later ported to 3D [23], extends the concept of Sparse Voxel Octrees by removing duplicate subtrees. The initial work on SVDAGs only considers two subtrees equivalent if they represent the exact same geometry. Later work extends this concept to equivalence under mirror symmetry [26], and approximate similarity [25]. We introduce a more general framework for efficiently finding any transformation that can be described by reordering child nodes; e.g. 90 degree rotations. Additionally, we show how similarity under geometric translation further decreases the size of the structure. Finally, we propose a new compact pointer scheme that aids in reducing memory usage.

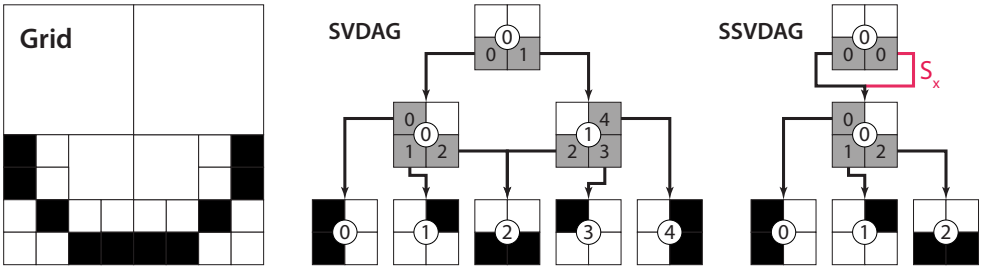


Figure 3.1.: Example of a grid with its SVDAG and Symmetry-Aware SVDAG (**S**SVDAG) representations.

3.2. BACKGROUND

We will now provide the necessary background for readers unfamiliar with Symmetry-Aware Sparse Voxel Directed Acyclic Graphs (**S**SVDAGs), and we present an improvement to the **S**SVDAG construction algorithm. For an introduction to regular Sparse Voxel Directed Acyclic Graphs (SVDAGs), we refer the reader to Chapter 2.

3.2.1. SYMMETRY-AWARE SVDAG (SSVDAG)

Symmetry-Aware SVDAGs [26] (**S**SVDAGs) have been introduced as the first and, so far, only method to support transformations inside an SVDAG. The

core idea is that two subtrees can be duplicates if their geometry matches after mirroring around the primary axes (Figure 3.4). In this scheme, the child pointers encode both the memory addresses and the mirror axes. The latter are encoded using 3 bits, one bit per axis (x, y, z).

LEAVES

Construction of an **S**SVDAG follows a structure similar to that of a regular SVDAG. The subtrees at the level $L = 1$ are flattened into 2^3 voxel grids. Finding duplicates with respect to symmetry transformations is more complicated than a regular SVDAG, as there may be multiple *different* grids g_i, which become equivalent after a symmetry transformation $S_{x,y,z}(g)$ is applied. We consider two grids equivalent when they represent the exact same geometry (Figure 3.2). We denote that using $\equiv$. This raises the question of which grid g_i should be chosen as the representative. The suggested solution[26] is to define, for each 2^3 grid g, the *canonical representation* $g^\star = \text{argmax}_{S_{x,y,z}} B(S_{x,y,z}(g))$, where $B(g)$ is the binary number when concatenating all voxels of g. This *canonical representation*, and the accompanying transformation, are precomputed for each possible 2^3 voxel grid and stored in a look-up table. Duplicates can then be easily detected by comparing the *canonical representation* $g^\star$ of each grid g.

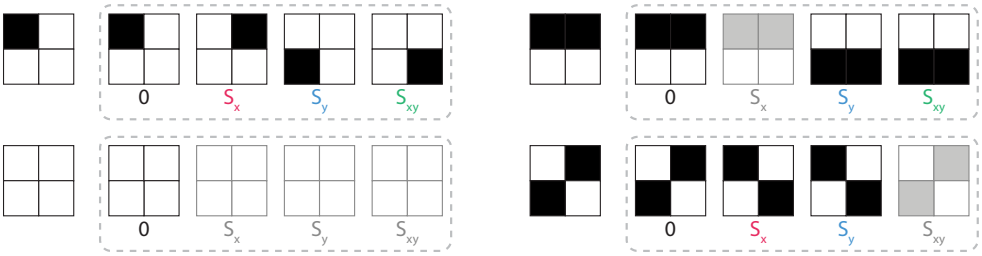


Figure 3.2.: Different 2^2 leaves (left) that can be transformed into the same *canonical representation* (right) in **S**SVDAG. Grids displayed in gray indicate invariance to that transformation.

INNER NODES

To search for duplicate inner nodes ($L > 2$), [26] recognize that mirror symmetry can be applied recursively. To apply a symmetry transformation to an inner node, one swaps its children along the respective symmetry axis and then applies the same transformation to those subtrees (Figure 3.3). By processing the tree from the bottom up, the child subtrees c_i, c_j that

are equivalent under *one* symmetry transformation (i.e. $c_i \equiv S_{x,y,z}(c_j)$), are guaranteed to have the same child pointer addresses. To determine whether two subtrees are equivalent under a *specific* transformation $S_{x,y,z}$, each subtree maintains a (temporary) 3-bit mask, storing its invariance to x, y, or z symmetry ($g \equiv S_{x,y,z}(g)$). To search for duplicate nodes, the search space is first reduced by clustering nodes with the same child pointers. An exhaustive search for the eight possible transformations $S_{x,y,z}$ is used to find all representative inner nodes and compute their respective mirror invariance.

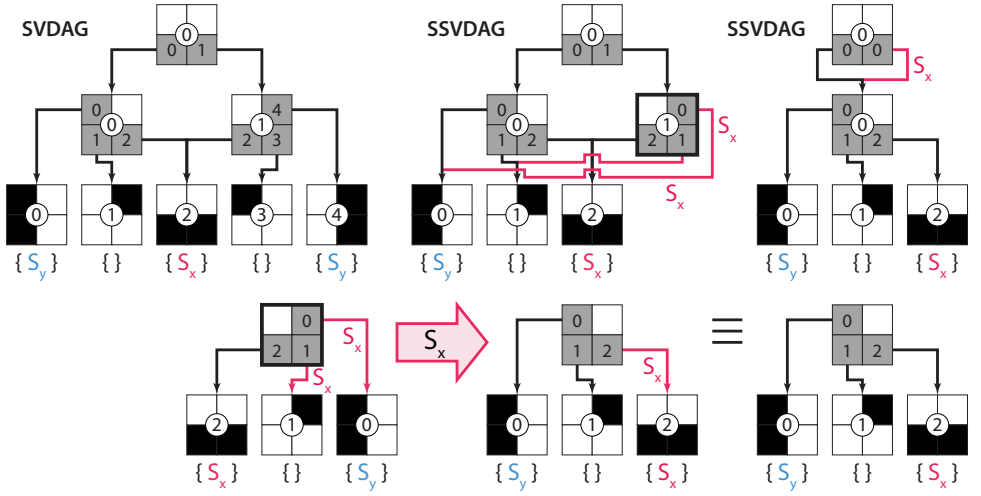


Figure 3.3.: Top row: converting an SVDAG to a Symmetry-Aware SVDAG (**S**SVDAG). Underneath each leaf we denote the symmetries to which it is invariant. Bottom row: after a mirror transformation, the two inner nodes are equivalent. One child received a different x transformation (pink) but is invariant to it.

3.2.2. IMPROVING SYMMETRY-AWARE SVDAG

In the original work [26], the authors assume that the invariance to symmetry along the x, y, or z axes is independent, and thus a 3-bit mask $M = \{m_x, m_y, m_z\}$ is sufficient to encode all invariances. However, while an invariance along, for example, the x and y axes implies xy invariance, the opposite does not always hold. The right/bottom example given in Figure 3.2 illustrates a grid invariant under xy, but not under x or y

symmetry. This type of invariance is currently missed by the original **S**SVDAG method [26], which leads to an increase in the number of inner nodes. This issue can be easily remedied using a 7-bit mask to store all invariance (x, y, z, xy, yz, xz, xyz). We found that this has an impact on the compression ratio of **S**SVDAG (Table 3.2).

3.3. Our Method

We extend the concept of **S**SVDAGs (Section 3.2.1) to support all transformations that can be described by reordering the children of the nodes to remove geometric redundancy. We then select a suitable subset of transformations to achieve practical compression. In addition, we introduce a new matching via geometric translations. In Section 3.4, we will describe a method for efficient encoding.

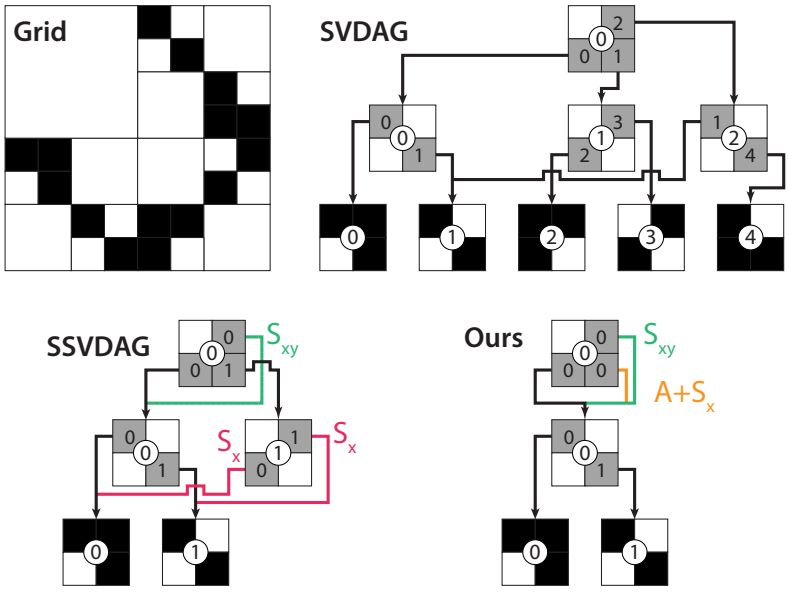


Figure 3.4.: Example of a 2D grid represented by different data structures. The SVDAG contains 9 items. This is reduced to 5 items by the (improved) **S**SVDAG [26]. Our method represents the same grid with just 4 items. S_i indicates mirror symmetry along axis i; A indicates that the x and y axes are swapped.

3.3.1. ARBITRARY TRANSFORMATIONS

We define a node N as a tuple of eight child pointers, one for each octant: $N := (c_0, \dots, c_7)$. A child pointer consists of a memory address and a transformation that is applied to that child; empty octants are represented by a null pointer ($c_i = \varnothing$). We will first consider all $8!$ permutation transformations that can be applied to the set of child octants. We define a permutation as a bijective function $T : \{0, \dots, 7\} \to \{0, \dots, 7\}$. To obtain a transformed child pointer $T(c_i)$, we concatenate T with the transform stored in c_i. Like Symmetry-Aware SVDAGs (**S**SVDAGs), these permutations are applied recursively to all child levels. A transformed node $T(N)$ consists of the shuffled children with the transform T applied to each child pointer. In theory, it would be possible to select a variable subset of children to which this transformation is applied, but for the transformation subset that we chose to use in practice, it proved beneficial to apply it to all children. However, more exploration would be possible.

LEAVES

Like in previous work, we flatten subtrees at level $L = 1$ into 2^3 voxel grids g_i. Rather than searching only for exact geometric matches, we need to consider that two *different* voxel grids can be deduced from the *same* representative grid using different transformations. As such, we need a deterministic method to pick a single representative. Let $\hat{G}$ be the set of grids that can be transformed into g: $\hat{G} = \{\hat{g} \mid T(\hat{g}) \equiv g\}$. We define the *canonical representation* $g^\star$ of a grid g as the "highest-value" grid in $\hat{G}$: $g^\star = \operatorname{argmax}_{\hat{g} \in \hat{G}} B(\hat{g})$, where $B(g)$ is the binary number obtained by concatenating all voxels of a voxel grid. Note that this definition differs slightly from that of **S**SVDAG (Section 3.2.1), which assumed that $T^{-1} = T$, which is not generally the case.

Using a precomputed look-up table, we replace each leaf by a *canonical-transform pointer* (containing the canonical grid $g^\star$ and the corresponding transformation) and a (temporary) bitmask that stores the invariance of $g^\star$ with respect to all possible transformations ($g^\star \equiv T(g^\star)$). This mask requires $8! = 40,320$ bits, which limits the practical scene size. We will address this later by reducing the number of permutations that are considered.

INNER NODES

When processing level L, we know that two subtrees at level $L - 1$ are equivalent under *some* transformation if and only if they share the

same *canonical representation* (canonical-transform pointer address). To reduce the search space, we first create clusters of nodes that have the same child pointer addresses. Note that the order in which the children appear or the transformations that are applied to them may be different. We then iterate through the nodes in each cluster, testing for each node N_i whether it can be deduced from any of the previous nodes: $\exists j < i, T : N_i \equiv T(N_j)$. We perform this test using an exhaustive search for all possible transformations T. If no match is found, then we consider the node a *canonical representation*, computing its invariance and outputting the node.

To test whether $N_i \equiv T(N_j)$, we first apply the permutation T to node N_j, which reorders the children and updates the transformations stored in their pointers. Two transformed child subtrees are equivalent $T_i(c_k) \equiv T_j(c_k)$ if they have the same *canonical representation* c_k (canonical-transform pointer address) and their transformations result in the same geometry. The latter can be tested by checking the invariance bit mask of the *canonical representation* for transformation: $c_k \equiv T_i^{-1} \circ T_j(c_k)$.

Practical Subset

Using *all* possible transformations reduces the number of tree nodes the most, but is impractical when it comes to encoding these transformations. We aim to find a small subset of transformations that still cover most reuse. Any valid subset must adhere to two requirements. First, the combination of two transformations in the subset must also be in the subset: $\forall T_i, T_j \in \Omega : T_i \circ T_j \in \Omega$. Second, for any transformation in the subset, the subset must also contain the inverse: $\forall T_i \in \Omega : T_i^{-1} \in \Omega$.

In practice, we found that the combination of two types of geometric transformations together accounts for almost all reuse (Table 3.1). Symmetries flip children along the primary axis; this is the same as **S**SVDAG [26] (with the improvement discussed in Section 3.2.2). Axis permutations swap the primary axis (e.g. $(x, y, z) \rightarrow (z, x, y)$); in total, there are 6 different axis permutations. When combined, these two types of transformation can represent any 90-degree rotation along one of the primary axes, which was not previously possible [26]. This is the symmetry group of the cube and the octahedron.

Table 3.1.: The number of nodes & 4^3 leaves when considering only the mirror symmetry (S) and axis permutation (A) subset $S + A$, rather than all possible permutations. All scenes use a resolution of $16k^3$.

Scene	SVDAG	S	A	S+A	All
Bistro	6.74M (+41.3%)	5.59M (+17.1%)	6.08M (+27.5%)	4.94M (+3.6%)	4.77M (100%)
Bunny	9.75M (+103.1%)	6.44M (+34.2%)	6.89M (+43.6%)	4.81M (+0.2%)	4.80M (100%)
Crown	23.01M (+65.5%)	16.88M (+21.4%)	18.03M (+29.6%)	14.00M (+0.6%)	13.91M (100%)
Citadel	9.41M (+67.9%)	6.89M (+23.0%)	7.56M (+35.0%)	5.64M (+0.7%)	5.60M (100%)
Lucy	5.68M (+45.5%)	4.71M (+20.6%)	4.92M (+26.0%)	3.92M (+0.6%)	3.90M (100%)
Dragon	5.97M (+45.3%)	4.96M (+20.8%)	5.12M (+24.8%)	4.13M (+0.6%)	4.11M (100%)

3.3.2. TRANSLATIONS

A general limitation of SVDAGs is that repeating geometry presents itself only as duplicate subtrees if they align with the SVDAG node boundaries. This can be partially remedied by including geometric translation. More specifically, we consider what we name *destructive* translations. This means that, when translating the voxels of a subtree $(2^L)^3$, any voxels that are translated "out" of the subtree will disappear (Figure 3.5).

The search space of all possible translations becomes very large for deep subtrees. We therefore only consider translating one subtree to obtain another, instead of applying destructive translations on both, and limit ourselves to searching only for translation matches in subtrees of up to 16^3 voxels ($L = 4$). In practice, higher levels have diminishing returns anyway, as the increased translation range $[-2^L + 1, +2^L - 1]^3$ takes more bits to encode.

Because translations operate across the boundaries between child subtrees, it is difficult to describe them in terms of SVDAG node/leaf operations. The subtrees are therefore flattened into $(2^L)^3$ voxel grids. We initially experimented with a naive search, iterating over the N voxel grids, applying each possible translation, and subsequently searching for a matching grid (using a hash table). This naive search has a time complexity

of $\mathcal{O}(NV^2)$, where $V = (2^L)^3$ is the number of voxels. The quadratic term originates from the fact that there are $\mathcal{O}(V)$ different translations, which take $\mathcal{O}(V)$ to apply. Although we consider SVDAG compression to be a preprocess; $\mathcal{O}(NV^2)$ is impractical (weeks of computation for $64K^3$ scenes on a modern consumer computer).

FAST TRANSLATION SEARCH

We propose a faster matching solution to reduce the complexity to $\mathcal{O}(NV)$. We solve the problem for groups of translations at a time, distinguished by the direction of the translations along each axis. For example, in the following, we will consider translations of the form $(-x, -y, -z)$ with $x, y, z \geq 0$. The process for all seven other direction groups is similar.

For each voxel, we compute a hash value that summarizes the voxel grid in the $(+x, +y, +z)$ direction; assuming that all voxels beyond the subtree boundary are empty. We then find possible matches via translation in the $(-x, -y, -z)$ direction. If the hash of a voxel at the most extreme subtree location in direction $(-x, -y, -z)$ matches any of the other voxel hashes in another grid, we have found a potential match (e.g., if (i_x, i_y, i_z) is a matching voxel, its translated subtree by $(-i_x, -i_y, -i_z)$ is a potential match). Since two different grids may produce the same hash code (hash collision), we confirm potential matches by translating the grids and testing for equivalence.

The hash computation over a running window (a *rolling hash*) can be accelerated [33, 34]. However, rolling hashes are typically not suitable for input with a small alphabet size (just 0 and 1 in our case), resulting in a high number of hash collisions. Instead, we reformulate our problem so that a regular hash function can be used.

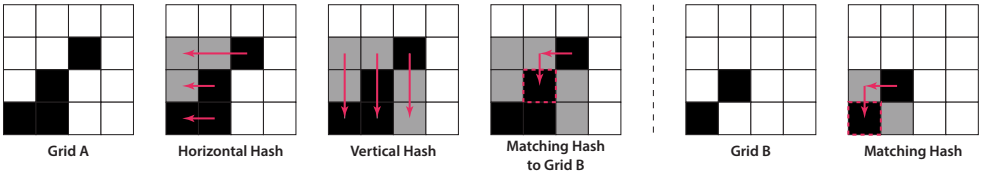


Figure 3.5.: We compute for each voxel a hash describing the area to the top/right. The hash is first computed horizontally, and then vertically. The hash at the lower/left voxel of grid B matches a voxel in grid A, indicating that B could be constructable from A by a translation of $(-1, -1)$.

Consider a one-dimensional example: rather than computing a hash

value based on the full region to the right of each voxel, we compute a hash that describes the region to the rightmost *occupied* voxel. The hash values can now be computed with a linear scan, starting with the rightmost occupied voxel: $H(v_i) = \text{hash\_combine}(H(v_{i+1}), v_i)$; we use an implementation of the Boost C++ library [35]. This method can be easily extended to our 3D grids. Similarly to separable image filters, we apply the hash function along each of the primary axes, one at a time (Figure 3.5). With this solution, finding all translations for the $L = 4$ (16^3) nodes takes a couple of hours for a $64K^3$ scene on the CPU. Note that this is significantly slower than the hierarchical search for mirror symmetry and axis permutations, which can typically be performed in less than a minute.

We can test the full set of our proposed transformations in a unified way. While testing for translations, we can check matching between different directions to additionally search for symmetry as well. For example, a voxel in direction $(+x, +y, -z)$ that matches a corner voxel (of a different grid) in direction $(+x, +y, +z)$, indicates that one can be transformed into the other by a combination of translation and z-axis symmetry. We use a similar process to search for permutations of axes.

DEPENDENCY GRAPH

A consequence of allowing *destructive* translations is that the dependency graph between transformed subtrees becomes complex. To reduce memory usage during compression, we only store a single "parent" per subtree, rather than the entire dependency graph. A parent is another subtree that can be transformed into the current subtree. If there are multiple parents, then we store the one with the highest memory address. This encourages clusters of similar subtrees to pick the same parent. We sort the subtrees in advance based on their number of occupied voxels to favor the most destructive translations. The resulting reduced dependency graph may contain multiple levels of indirection (Figure 3.6). That is, a subtree that can be constructed from another subtree, which itself can be constructed from yet another subtree. Due to limitations in the SVDAG encoding, we are forced to duplicate subtrees in these cases (**C** in Figure 3.6).

3.4. IMPLEMENTATION

Achieving a high compression ratio requires not only a reduction in the number of nodes/leaves, but also a compact encoding scheme. Such a

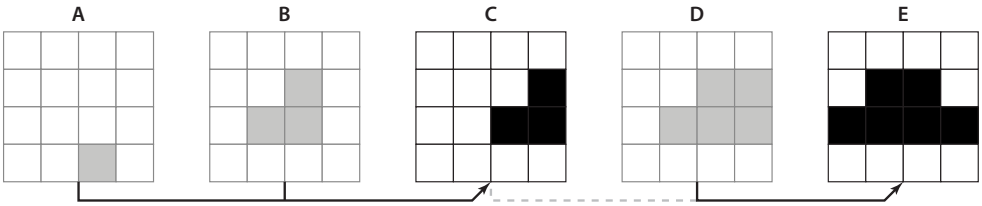


Figure 3.6.: Five grids and their simplified dependency graph, storing only a single match per grid. **E** is stored as a unique subtree because it cannot be constructed from any of the other grids. **C** must be stored because **A** and **B** depend on it. The remaining grids (gray) can be constructed from **C** and **E**.

scheme should be compact, but still enable direct traversal of the SVDAG. This precludes the use of pointerless encoding, such as those presented in [36, 37]. Instead, we use a more traditional encoding where a node is represented by a bitmask to indicate which children are nonempty, followed by child pointers to these children.

3.4.1. Transform ID

Our **T**SVDAG complicates pointer encoding, as they do not only contain a memory address, but potentially also a child-reorder transformation (symmetry & axis permutation) and a translation. We encode all transformations in the least-significant bits, followed by the corresponding memory address.

Each child reorder transformation in our subset (Section 3.3.1) is assigned a unique number. For translations, we find that the most common translation distance along any axis is 0, with other distances distributed somewhat uniformly. Thus, we treat a translation of 0 as a special case. We combine the child-reorder transformation identifier with a 3-bit mask indicating a 0 translation along each of the primary axes. This combined identifier, which we call the *Transform ID*, is stored using Huffman coding to ensure that the most frequently occurring transformations use the least number of bits. We limit the length of Huffman codes [38] to prevent very rare transforms from creating extremely long pointers. The *Transform ID* is stored in the least significant bits of the pointer, followed by a translation distance for each axis with a non-zero translation, and finally the memory address to which it is pointed (Figure 3.7).

3.4.2. NODE ENCODING

Similar to [26, 39], we opt for variable-length pointer encoding, with either 16-, 32-, or 48-bit pointers. The latter is necessary, as encoding transformations create significantly longer pointers. Like [26], we sort our nodes such that the most referenced ones appear first. This ensures that the most frequently used pointers are the shortest.

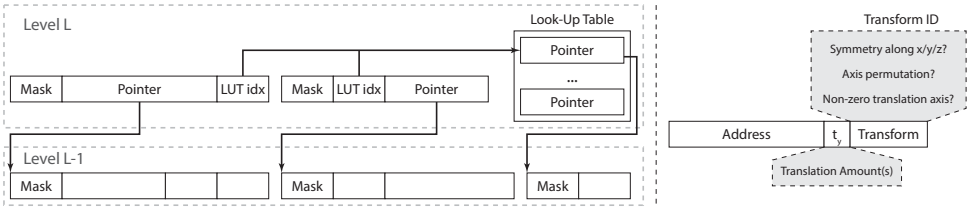


Figure 3.7.: Left: illustration of two nodes at level L, each with two children. The children are referenced either directly or through a look-up table. Right: example of a pointer with a translation along the y axis. The *Transform ID* encodes symmetry, axis permutation and non-zero translation axis. The translation amounts (t_y) are stored in the more significant bits.

POINTER TABLES

Because of variable-length node encoding, our address space is indexed at 16 bit intervals rather than at node intervals. As a result, many memory indices are never referenced because they do not point to the start of a node. This significantly reduces the number of child pointers that can utilize the compact 16-bit and 32-bit encoding. Second, transformation encoding significantly bloats the pointer sizes. This incurs a heavy penalty if the same pointer occurs many times.

To remedy both issues, we selectively store pointers in a look-up table using fixed size entries of 64 bits (Figure 3.7). This is somewhat similar to the far pointers of [39]. 16-bit pointers now always encode an index into the look-up table, whereas 32-bit pointers reserve 1 bit to indicate whether they contain a table index or a direct pointer; 48-bit pointers are always direct pointers. One look-up table is constructed per SVDAG level using a greedy algorithm. First, the pointers are sorted according to their frequency (how often they occur in the parent level). We then iterate over the sorted pointers, comparing the cost of storing them directly, versus in the look-up table, and subsequently insert them into the look-up table if it is the cheaper option. The cost of a direct pointer is simply its length

in bits, rounded up to either 32-bits or 48-bits, times the number of times it occurs. A table pointer incurs a fixed cost of 64 bits per entry, but the resulting 16- or 32-bit pointer will result in lower overall memory usage if it is referenced frequently.

Node Header

The lengths of the child pointers are encoded using a 16-bit mask at the start of the node. Conceptually, this represents eight 2-bit integers storing the length of the child pointers, where a length of 0 represents an empty child. Some tasks, such as rendering, require accessing the n^{th} child pointer. This can be efficiently computed using the *__popc* intrinsic:

Algorithm 1 Efficient extraction of the n'th child pointer.

$prefix \leftarrow nodes[nodeStart]\ \&\ (0xFFFF >> (16 - 2 * childIndex))$
$sum \leftarrow \_\_popc(prefix) + 2 * \_\_popc(prefix\ \&\ 0b1010101010101010)$
$pointer = nodes[nodestart + 1 + sum]$

3.4.3. Rendering

To visualize our **T**SVDAGs, we implement a CUDA-based ray tracer that can display various SVDAG formats. Path tracing uses a megakernel approach, although a wavefront implementation is also available [40]. We also experimented with replacing the top levels of the SVDAG, typically showing very little reuse, by a hardware-accelerated BVH using Optix. We found that this reduces performance, despite promising findings by [41].

Ray Traversal

Ray intersection is implemented as a stack-based traversal. The intersection points between the ray and the three primary planes passing through the center of a node are computed, from which a bitmask is created indicating which children are intersected. The traversal order is determined by the sign bits of the ray direction following the source code of [28].

Child Reordering

Our renderer supports a subset of transformations, namely mirror symmetry and axis permutations (Section 3.3.1). The transformation state

is stored on the traversal stack, enabling efficient backtracking. To reduce the size of the stack, we store the transformations by assigning them a unique identifier; this is similar to the *transform ID* (Section 3.4) but without the translation part. A look-up table in GPU constant memory is used to apply the respective permutations. A similar table stores the result of applying two transformations ($T = T_i \circ T_j$) as defined in these compact identifiers.

TRANSLATION

Our definition of *destructive* translation requires careful adjustments to the traversal algorithm. When encountering a translating pointer, the contents of the subtree should be translated; removing any voxels that are moved outside of the bounds of the (non-translated) subtree. This is achieved using a second stack only for those levels of the **T**SVDAG which may contain translations (up to 16^3). Each entry in this stack stores the accumulated amount of translation and the distance at which the ray entered & exited the subtree (tmin & tmax). These values are used to translate the child node centers and to limit their intersection bounds.

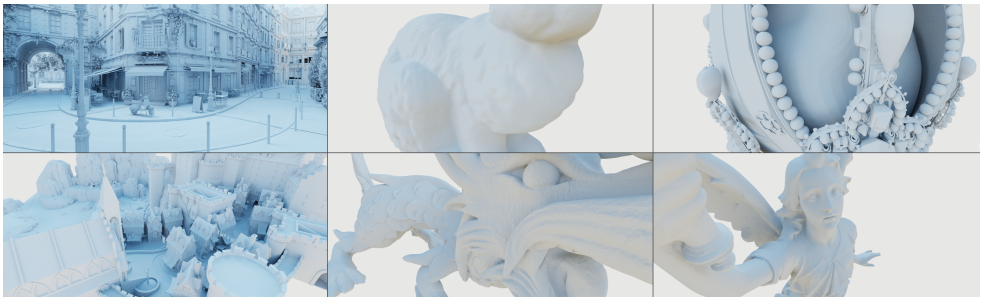


Figure 3.8.: The scenes used for our evaluation at a voxel resolution of $64k^3$.

3.5. RESULTS & DISCUSSION

We evaluated our **T**SVDAG method on a variety of different inputs (Figure 3.8). We chose a combination of architectural scenes (Citadel, Bistro), artist-made models (Crown), and 3D scans of real-world objects from the Stanford 3D Scanning Repository (Bunny, Lucy, Dragon). These were turned into voxel models using the voxelizer in [26]. For the Citadel scene, we removed most of the surrounding landscape, making the scene

more cube-shaped, leading to a higher voxel occupancy. To add context to our results, we include comparisons against a Sparse Voxel Octree, SVDAG, and **S**SVDAG. All structures encode 4^3 leaves using 64 bits.

Table 3.2.: Number of nodes and 4^3 leaves for a Sparse Voxel Octree (SVO), Sparse Voxel Directed Acyclic Graph (SVDAG), Symmetry-Aware Sparse Voxel Directed Acyclic Graph (**S**SVDAG), and our method(s). We present our method with symmetry S, axis permutation A, and translation T.

Scene	SVO	SVDAG	**S**SVDAG	S	A	S + A	S + A + T
Bistro ($64k^3$)	1692.1M	69.6M	56.6M	55.2M	60.0M	47.6M	**41.1M**
21.4B voxels	(+2889%)	(+23%)	(100%)	(-2%)	(+6%)	(-16%)	**(-27%)**
Bunny ($64k^3$)	1267.2M	75.0M	58.0M	55.5M	59.0M	45.0M	**29.2M**
15.2B voxels	(+2085%)	(+29%)	(100%)	(-4%)	(+2%)	(-22%)	**(-50%)**
Crown ($64k^3$)	3205.8M	178.3M	143.4M	137.7M	150.3M	114.5M	**83.5M**
38.6B voxels	(+2136%)	(+24%)	(100%)	(-4%)	(+5%)	(-20%)	**(-42%)**
Citadel ($64k^3$)	1428.8M	65.3M	52.8M	48.7M	54.8M	41.6M	**28.2M**
17.2B voxels	(+2607%)	(+24%)	(100%)	(-8%)	(+4%)	(-21%)	**(-47%)**
Lucy ($64k^3$)	526.6M	52.9M	40.3M	39.1M	41.4M	32.4M	**26.3M**
6.3B voxels	(+1207%)	(+31%)	(100%)	(-3%)	(+3%)	(-20%)	**(-35%)**
Dragon ($64k^3$)	523.9M	53.7M	40.6M	39.5M	41.1M	32.7M	**26.2M**
6.3B voxels	(+1191%)	(+32%)	(100%)	(-3%)	(+1%)	(-19%)	**(-35%)**

3.5.1. Structure Size

We will first consider the effectiveness of our methods in reducing the size of the structure, measured in the number of nodes & leaves. Table 3.2 shows the total number of elements (nodes + leaves) for various scenes and methods. The odd numbered rows count the number of elements as a percentage relative to **S**SVDAG [26]. We abbreviate symmetry, axis permutation, and translation by S, A, T, respectively. Translations are computed for nodes/leaves at the levels representing grids of 16^3, 8^3 and 4^3 resolution.

When comparing **S**SVDAG to our improved method of finding mirror symmetry (Section 3.2.2), we see that our fix indeed improves the compression ratio. The number of elements marked as duplicates increases by a couple percent depending on the scene. Only permuting the primary axes, without symmetries, leads to a notable reduction in the number of nodes and leaves compared to a regular SVDAG. When combining both, we see a significant improvement over previous work, which only considers axis mirroring. Translating subtrees of resolution 4^3, 8^3, and 16^3 further reduces the number of nodes and leaves. The latter

suggests that a higher compression ratio may be achieved when one is not limited to only transformations that can be described as a recursive reordering of child pointers.

Table 3.3.: The impact of the different features of our pointer encoding on the overall size of the Transform SVDAG. N/A indicates that the scene could not be encoded due to running out of pointer bits.

Scene	Base	Pointer LUT	Pointer LUT + Huffman
Bistro ($64k^3$) 21.4B occupied voxels	876MB (+16%)	753MB (100%)	**721MB** **(-4%)**
Bunny ($64k^3$) 15.2B occupied voxels	N/A	718MB (100%)	**663MB** **(-8%)**
Crown ($64k^3$) 38.6B occupied voxels	N/A	1965MB (100%)	**1851MB** **(-6%)**
Citadel ($64k^3$) 17.2B occupied voxels	715MB (+14%)	625MB (100%)	**597MB** **(-4%)**
Lucy ($64k^3$) 6.3B occupied voxels	683MB (+20%)	567MB (100%)	**535MB** **(-6%)**
Dragon ($64k^3$) 6.3B occupied voxels	681MB (+19%)	571MB (100%)	**540MB** **(-5%)**

3.5.2. POINTER COMPRESSION

Reducing the number of elements is only half of the story to achieve a good compression ratio. In theory, with fully arbitrary transformations, a single canonical representation would suffice to represent all subtrees. However, in practice, the cost of encoding these transformations would outweigh the cost of simply storing duplicates. Thus, we evaluate how the presented encoding scheme (Section 3.4) allows us to compactly store our **T**SVDAGs with various types of transformations.

Table 3.3 evaluates the impact of our pointer encoding scheme for our full **T**SVDAG (symmetry, axis permutation, and translation). Without pointer look-up tables ("Base" in Table 3.3), we find that some scenes fail to encode due to running out of the 48-bit pointer space. This typically happens for pointers at level $L = 4$ with translations along multiple axes, as storing the translation amounts requires 5 bits per axis (in addition to the *Transform ID*). Introducing a look-up table for large- or frequently used pointers solves this issue by providing storage for 64-bit pointers. As expected, the additional indirection helps to reduce memory usage, as

frequently used pointers can now be stored as a small index into the table. Finally, applying Huffman coding to the *transform ID* further reduces memory usage by 4% to 8% depending on the scene.

Table 3.4.: Memory usage of an SVO, SVDAG, **S**SVDAG, and our method. We present our method with symmetry S, axis permutation A, and translation T. "$S + A(B)$" uses the "base" pointer encoding (no look-up tables or Huffman coding), the other columns use our full encoding scheme. An equivalent $64k^3$ dense voxel grid would require 35TB.

Scene	SVO	SVDAG	**S**SVDAG	S	A	S + A (B)	S + A	S + A + T
Bistro	13537MB (+1456%)	1309MB (+50%)	870MB (100%)	838MB (-4%)	890MB (+2%)	803MB (-8%)	764MB (-12%)	**721MB (-17%)**
Bunny	10137MB (+838%)	1736MB (+61%)	1080MB (100%)	1014MB (-6%)	1067MB (-1%)	900MB (-17%)	864MB (-20%)	**663MB (-39%)**
Crown	25647MB (+866%)	4146MB (+56%)	2654MB (100%)	2500MB (-6%)	2708MB (+2%)	2286MB (-14%)	2182MB (-18%)	**1851MB (-30%)**
Citadel	11430MB (+1126%)	1454MB (+56%)	932MB (100%)	840MB (-10%)	935MB (+0%)	787MB (-16%)	749MB (-20%)	**597MB (-36%)**
Lucy	4213MB (+526%)	1264MB (+88%)	673MB (100%)	645MB (-4%)	674MB (+0%)	597MB (-11%)	561MB (-17%)	**535MB (-21%)**
Dragon	4191MB (+516%)	1280MB (+88%)	681MB (100%)	655MB (-4%)	675MB (-1%)	605MB (-11%)	571MB (-16%)	**540MB (-21%)**

3.5.3. Overall Compression Ratio

Table 3.4 compares our memory usage with a Sparse Voxel Octree (SVO), Sparse Voxel Directed Acyclic Graph (SVDAG) and a Symmetry-Aware Sparse Voxel Directed Acyclic Graph (**S**SVDAG). The Sparse Voxel Octree is encoded by storing the children of a node in consecutive memory. It requires that a node only stores a pointer to the first child; the remaining children can be accessed with a simple offset. As commonly used in the literature, we store these nodes using 64 bits: a 32-bit pointer plus a child mask. For SVDAG [23] and **S**SVDAG [26], we use the encodings described in their respective papers.

Despite the efficient SVO encoding that reduces the number of bits per node, the SVO still requires significantly more memory than an SVDAG. In comparison, the additional transformations supported by our method, combined with our encoding scheme, provide significant memory savings over a traditional SVDAG [23]. Compared with the previous best method, **S**SVDAG, we see a typical improvement between 20% and 30% depending on the scene.

3.5.4. RENDER PERFORMANCE

We measure rendering performance by capturing the frame time of a 1080p path-traced image with up to four random bounces. The scenes are lit by a (constant) environment light without next-event estimation. Figure 3.9 shows the results as measured on an NVIDIA RTX4070.

As expected, an SVDAG with fixed 32-bit pointer encoding outperforms variable pointer length methods in render time; typically by about 10%. With only child-reordering transformations (no translations), our method performs slightly worse than **S**SVDAG when we disable the pointer encoding features that require additional memory accesses (pointer look-up tables and Huffman). Enabling these encoding features increases the frame time by about 12%. Translations add an additional cost of about 30%. We theorize that most of this cost is caused by having to maintain an additional, but short, traversal stack (Section 3.4.3).

The increased rendering cost of **S**SVDAG and our method stem from the mechanisms to save memory and are linked to two main aspects. First, the GPU memory system was not designed to efficiently read variable-length pointers. Second, the larger traversal stack leads to either additional registry pressure or registry spilling. Given these observations, we suspect that the relative rendering cost of **S**SVDAG and our method will be lower on a CPU renderer.

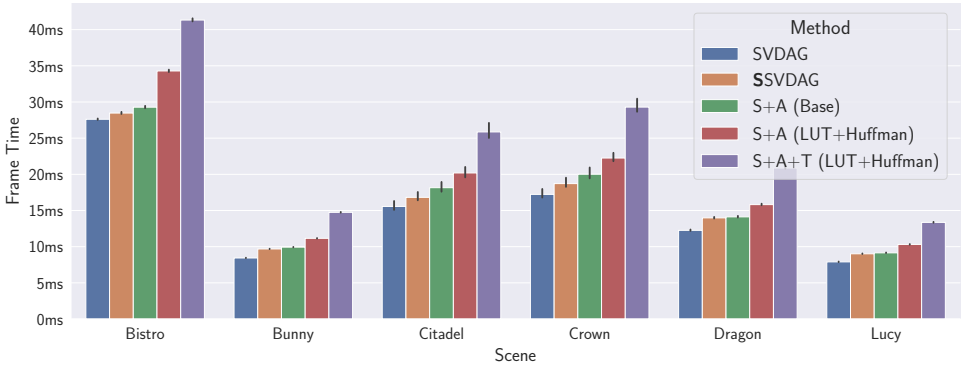


Figure 3.9.: Path-traced rendering performance comparison between a traditional SVDAG with 32-bit encoding, **S**SVDAG, and various configurations of our method.

3.6. Conclusion

Sparse Voxel Directed Acyclic Graphs (SVDAGs) have proven to be a very efficient method for storing sparse binary voxel data, while still enabling efficient random-access and ray intersections. The SVDAG is constructed from a Sparse Voxel Octree (SVO) by fusing similar subtrees. Previous work shows how additional compression can be achieved by finding similarities via mirror transformations. However, the previous solution only considered a subset, and we suggested a simple extension (Section 3.2.2).

We then generalized the framework for efficiently finding subtrees under any transformation, which can be described as a (recursive) reorder of child pointers. We have shown how a limited subset of these transformations can be used to achieve practical compression. However, not all transformations can be described as permutations of a tree structure, and we explored translations as an example. Currently, finding these translations is still expensive due to the very large search space. However, we believe that this is an interesting area for future research. When combined with our novel pointer compression scheme, we achieve a typical improvement of 20% to 35% over the state-of-the-art [26].

4

Editing Compact Voxel Representations on the GPU

Mathijs Molenaar, Elmar Eisemann

A Sparse Voxel Directed Acyclic Graph (SVDAG) is an efficient representation to display and store a highly detailed voxel representation in a very compact data structure. Yet, editing such a high-resolution scene in real-time is challenging. Existing solutions are hybrid, involving the CPU, and are restricted to small local modifications. In this work, we address this bottleneck and propose a solution to perform edits fully on the graphics card, enabled by dynamic GPU hash tables. Our framework makes large editing operations possible, such as 3D painting, at real-time frame rates.

Parts of this chapter have been published in the Eurographics Digital Library (2024) [42].

4.1. INTRODUCTION

A voxel representation that has received a lot of attention in recent years is Sparse Voxel Directed Acyclic Graphs (SVDAGs). This representation involves a sparse compressed voxel encoding, where repeated information is only stored once. Thus, the typically high memory consumption of voxel representations is significantly reduced. One major downside of this representation is that being compressed means that real-time modifications are not straightforward, precluding the use of this representation in many voxel contexts.

In this chapter, we present a method for storing, editing, and rendering high-resolution voxel models using SVDAGs. Our work is closely related to (and builds on top of) the HashDAG [28]. Yet, while HashDAG implements editing on the CPU and achieves only interactive, not real-time frame rates, we present a fully GPU-based method to make full use of the highly parallel hardware. Our contributions include a novel GPU hash table structure to encode the DAG, an efficient implementation for graphics hardware, and extensive evaluation of various implementation variants.

4.2. BACKGROUND & RELATED WORK

Editing SVDAGs is more difficult than modifying octrees. Adding a new subtree requires eliminating redundancy in its interior and the existing DAG structure. Similarly to the construction algorithm in [23], this operation can be efficiently implemented using a bottom-up approach, which only requires node-node comparisons, rather than tree-tree comparisons. A more detailed explanation is provided in Chapter 1.

4.2.1. HASHDAG

Careil, Billeter, and Eisemann [28] propose the HashDAG to display and edit SVDAGs interactively. At the core of this framework lies a hash table, which stores all SVDAG nodes and leaves. This hash table, which is created at start-up, is sized to be large enough to contain any nodes/leaves that are created during editing. This conservative memory estimate is likely to exceed the physical memory constraints, which is addressed using virtual memory.

The hash table is replicated in CPU and GPU memory (for editing and rendering, respectively). Editing is performed on the CPU by traversing the SVDAG and deciding at each node whether to modify it or keep it as-is. The recursive algorithm descends into the to-be-modified children until

the leaf level is reached. At this point, the updated leaves are constructed and added to the SVDAG hash table. This returns a pointer to the new leaf, which is used to construct an updated node at the parent level. It is effectively a recursive implementation of the bottom-up algorithm (Chapter 2). Multi-threading is achieved by spawning tasks for the first levels of recursion. To prevent race conditions, each bucket of the hash table is protected by a mutex. After editing, the changes are then uploaded to the GPU so that they can be displayed. This can lead to large and very scattered memory transfers which become a significant performance bottleneck. Our goal is to move the entire editing operation to the GPU to achieve better performance; allowing large edits in real-time.

4.2.2. GPU Hash Tables

Like HashDAG[28], our SVDAG editing framework is based on hash tables. Thus, we require a suitable high-performance hash table that runs on the graphics card. Early work on GPU hash tables often focused on static construction [15, 43, 44]. Although useful for many computer graphics applications [17], they are not suitable for our use case.

Various dynamic GPU hash tables have been developed recently, which differ in their collision handling (two keys mapping to the same bucket). In *open addressing* schemes, collisions are handled by moving a key/value (KV) pair to a different bucket. Searching then requires iterating over multiple buckets until the item is found, or we can guarantee that the key is not in the hash table.

Stadium Hashing [45] and WarpDrive [46] use probing schemes. If the bucket pointed to by the hash function is full, they iterate over neighboring buckets in a deterministic pattern until an empty slot is found.

Cuckoo hashing [47], as applied in [48, 49], assigns each key/value pair to two buckets using two different hash functions. When encountering a full bucket (a.k.a. a collision), the new item is inserted by removing an existing item from the bucket. The removed item is subsequently inserted into the hash table again using the secondary hash function, following the same process. Since items may only reside in either of the two buckets, lookups have a constant worst-case time complexity. Insertions can be computationally expensive due to the potential for unbounded recursion.

In *closed addressing* hashing schemes, such as SlabHash [50], collisions are handled by creating a linked list of key/value pairs in each bucket. Compared to open-addressing this can make a table grow indefinitely, although the lookup performance tends towards a linear search as the

number of items grows to infinity. The same concept also applies to HashDAG [28], where virtual memory is used to create large buckets.

Developing hash tables for the GPU introduces additional challenges, as a lot of performance can be gained by designing memory layouts that better match the GPU hardware capabilities. A common approach in most existing work [46, 48–50] is to assign insertion or search operations to warps (currently a group of 32 threads) instead of individual threads. Each bucket of the table stores multiple slots, typically a multiple of 32. The threads inside a warp cooperate to check consecutive slots to either find a search key or an empty slot in the case of insertion. This results in fast coalesced memory accesses, which maximizes memory bandwidth and, thus, performance.

4.2.3. SLAB HASH

For reasons that will be discussed in Section 4.3.2, SlabHash[50] forms the basis for our hash table implementation for SVDAG editing on the GPU. SlabHash consists of an array of N buckets. The search keys are transformed into an arbitrary integer H by a hash function and inserted into the bucket $B = H \bmod N$. By deterministically assigning items to buckets, the search space is reduced from the entire data set to a single bucket. Each bucket is represented by a linked list of slabs, allowing it to grow dynamically (Figure 4.1). The slabs consist of 31 slots to store key/value pairs, plus a pointer to the next slab. This memory layout ensures that slabs are aligned to GPU cache lines (128 bytes).

Searching the hash table for a specific item is performed using a warp of 32 threads. After computing the hash, a warp iterates over the linked

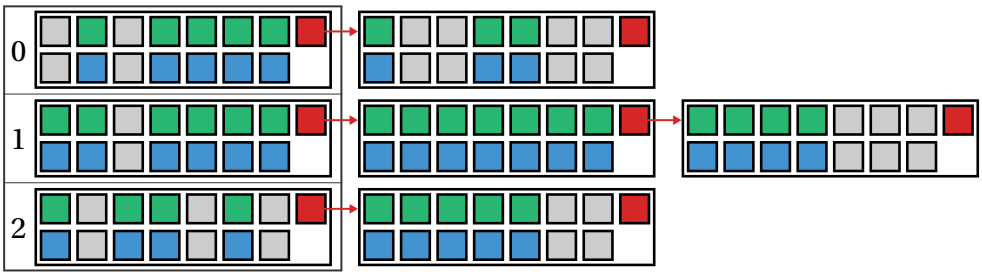


Figure 4.1.: Illustration of SlabHash [50] with four buckets. For illustrative purposes each slab contains 10 slots, consisting of a key (green) and a value (blue). Gray indicates an empty slot. The last slot (red) is used to store a pointer to the next slab.

list of slabs in the selected bucket. For each slab, thread i loads the i'th key from the current slab and compares it with the key that is being searched for. If a match is detected, all threads in the warp collectively return the value in that slot. Empty slots are indicated by a special reserved key (typically 0), which makes the insertion process similar to searching for that value and atomically swapping it with the desired key.

4.3. Our Method

In this chapter, we provide an overview of our method. We start with a general overview of the Sparse Voxel Directed Acyclic Graph (SVDAG), followed by our unique GPU hash table design. Finally, we describe how editing is implemented in practice using various different compute kernels.

4.3.1. SVDAG Representation

Leaves in our SVDAG encode regions of 4^3 voxels, using a 64-bit mask followed by 4-bit material indices for each of the occupied voxels. The leaves are padded to align to 32 bits, which is the fundamental *WORD* size of our SVDAG. The inner nodes of the SVDAG are stored in a similar way, starting with a bitmask indicating which of the 2^3 child regions are occupied, followed by the 32-bit child pointers to each nonempty child. We additionally reserve some bits to store whether a region is homogeneous (fully filled with a single material), and if so, with what material. This information is used to accelerate editing tools that benefit from knowing whether a region is fully filled.

Materials

There are various real-world scenarios in which a user might want to store not only occupancy but also a material (or some other numeric value) in each voxel. This is supported in our GPU editing framework using 4-bit material identifiers for each occupied voxel, allowing for 16 unique materials. The number of bits can be easily adjusted if more precision is required. Furthermore, one may use these values to encode an index in a position-dependent palette to allow more variety

4.3.2. CUSTOM GPU HASH TABLE

We rely on a hash table to store the SVDAG nodes and leaves, which allows us to quickly search for duplicates. This introduces the following requirements to the GPU hash table design:

Pointers must be stable: Inserting new items should not change the pointers of items that already reside in the hash table. If a pointer were to change, all SVDAG nodes pointing to that item would have to be updated. This would change their respective hash keys, moving them to a different location in the hash table, and thus invalidating their pointers as well. Due to this recursion, moving a single item would require a complete scan over the entire SVDAG.

Large items: Our nodes and leaves are stored using variable length encoding. For simplicity, we create separate hash tables for each potential size (measured in *WORDs*). Most items exceed the size of the basic data type of a GPU (8 bytes), eliminating the possibility of atomically swapping items as in some related work [43, 44, 46, 48–50].

Optimized for SVDAG traversal SVDAG traversal is performed during both editing and rendering. We optimize for this operation, even if it comes at a (slight) cost to insertion or search performance. In practical terms, this means that we try to avoid splitting items into different memory locations.

Based on these requirements, we conclude that open addressing schemes are unsuitable for our use case. Probing insertion requires low load factors (thus high memory usage) and replacement strategies, such as Robin Hood and Cuckoo hashing, do not provide pointer stability. Furthermore, open addressing requires re-hashing when the table becomes full. A closed addressing scheme, such as SlabHash [50], is more suitable, even though it may perform slightly worse than open addressing [49].

SlabHash [50] stores a linked list of slabs. A slab is an array of 31 item slots, plus a pointer to the next slab. Empty slots are indicated by a reserved value (typically 0), which is atomically swapped with the desired key when inserting. Combining multiple items in a single slab reduces stress on the memory allocator and allows efficient processing on the GPU.

Extending SlabHash to support larger keys is challenging, as GPU hardware does not provide atomic operations for values larger than 8 bytes. The size of our SVDAG nodes and leaves ranges from 2 to 10 32-bit *WORDs*, making atomic swapping of entire items impossible. Thus, we must make sure not to perform simultaneous insertions and lookups.

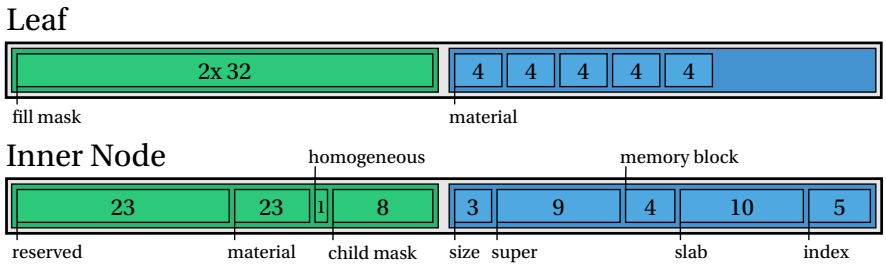


Figure 4.2.: Memory layout of SVDAG nodes and leaves. A leaf starts with a 64 bit mask (single green block for brevity) followed by up to 64 material IDs (blue). A node starts with a 32-bit header (green) followed by up to 8 child pointers (blue).

We reserve one special value for the first eight bytes of an item (see Figure 4.2) to indicate empty slots. Because an inner node must have at least one nonempty child, the child bitmask (stored in the fourth byte) will never be zero. Similarly, a leaf node must contain at least one filled voxel, and thus its 8-byte bitmask may not be zero either. Therefore, we can use an 8-byte value of 0 to indicate empty slots.

Hash Table Variants

We will now discuss four (novel) variations of SlabHash that we have developed with the goal of supporting larger items. We will refer to these as: *Atomic64, Ticket Board* and two *Acceleration Hash* variants.

Atomic 64 is a straightforward extension of SlabHash that stores the first eight bytes (64 bits) of all items in a contiguous array at the beginning of the slab (Figure 4.3). In case the items are larger than eight bytes, the slots are followed by the remaining bytes of the items. The remainders are stored in an array-of-structures layout such that accessing an item typically requires loading two cache lines (the first eight bytes plus the remainder). Like SlabHash, each slab consists of 31 slots and is padded to achieve cache line alignment (128 bytes on our hardware).

In line with previous work, search and insertion operations are performed by warps consisting of 32 threads (Algorithms 2 and 3 in Appendix A). Each thread reads the first eight bytes of its corresponding slot and compares them with the search item. The subsequent comparison (or insertion) of the remainder of the item is also performed by the entire warp.

This memory layout is compact, but it splits nodes and leaves into multiple cache lines. This causes the most common operation, traversing the SVDAG, to be more expensive. Although we found this to have little impact on primary visibility rays, where traversal is coherent, it does reduce path-tracing performance. Placing the camera inside a voxelization of the watertight Stanford bunny, we found that the rendering performance decreased by 9% at 1 indirect bounce and by more than 16% at 6 indirect bounces.

Ticket Board inspired by Stadium hashing [45], stores items contiguously in memory to make the SVDAG traversal more efficient. A bitmask at the start of the slab indicates whether each of the 32 slots is occupied. A downside of this design is that search becomes less efficient. Searching requires the entire slab to be loaded from memory, irrespective of whether it contains the item being searched for. As such, there is also no reason to strive for cache line alignment, and we forgo padding.

Acceleration Hash is a new variant that aims to address the limited search performance of *ticket board*. Instead of a single bit, we store an array with a second, *different*, hash of the item stored in each corresponding slot. The hash-value range starts at 1 such that a value of 0 indicates an empty slot. When searching, each thread in a warp loads its corresponding hash value (stored contiguously in memory) and compares it to the secondary hash of the item being searched for. With this solution, warps only load a single cache line in order to decide whether the slab might contain the item being searched for.

We consider two variants of the acceleration hash. The first optimizes for performance by using 32-bit hash values. We again only use 31 slots to ensure cache line alignment. The second version uses 8-bit hashes, reducing memory overhead from 4 bytes to just a single byte per slot. For compactness, we do not perform any memory padding and use 32 slots per slab (Figure 4.3).

MEMORY ALLOCATION

When inserting an item into the hash table, a warp may encounter the situation where all the slabs in the bucket are occupied. In this case, the warp must allocate a new slab and add it to the linked list of the bucket. We use SlabHash's *SlabAlloc* [50] to allocate these (fixed-size) slabs. We will provide a brief overview of the allocator. For more details, we refer the reader to the original work of Ashkiani, Farach-Colton, and Owens [50].

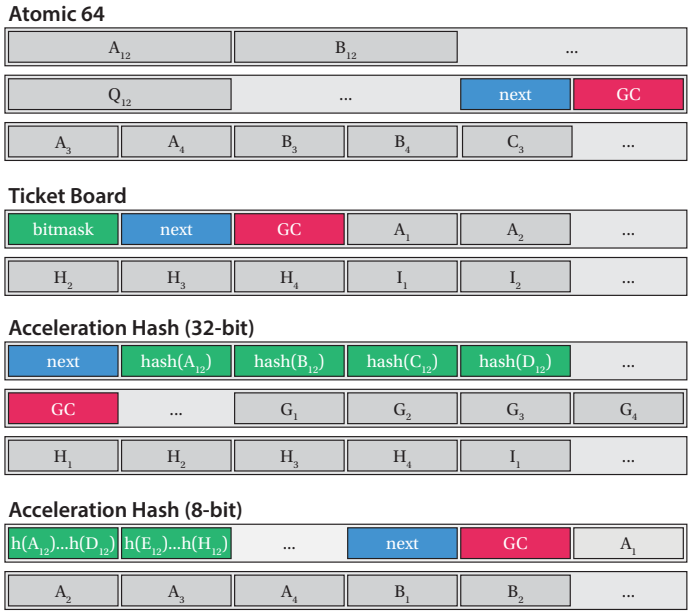


Figure 4.3.: The proposed slab memory layouts displayed as 128-byte cache lines, storing items consisting of four WORDs (16 bytes) each. X_i indicates the i'th WORD of item X. Repeating patterns are indicated by "...". Each method contains a pointer to the next slab (blue), and a garbage collection bitmask (red). The bottom methods are tightly packed and may not always start at the beginning of a cache line.

The allocator maintains *memory blocks*, which contiguously store 1024 slabs, along with a bitmask to indicate which of those slabs has already been used. The allocation of a slab from a given *memory block* is carried out by a warp of 32 threads. The 1024-bit mask is loaded in a single memory transaction, with each thread searching a 4-byte subregion for zero bits (indicating an unused slab).

To be able to allocate more than 1024 slabs, we must maintain multiple *memory blocks*. These are grouped into what are called *super blocks* of (in our case) sixteen *memory blocks* each. During each editing frame, the CPU makes a conservative estimate of how many hash table insertions will potentially take place. This is added to the number of currently allocated *memory blocks* (tracked with a GPU atomic counter) to decide whether the CPU must allocate additional *super blocks* in that frame.

To now perform parallel allocations on the GPU, a warp selects a *super*

block based on its warp index. To reduce thread contention, each warp selects one *memory block* at random within the chosen *super block*. If the selected *memory block* does not contain an empty slab, then the warp moves onto the next *super block*.

4.3.3. EDITING

In this section, we discuss how we implemented SVDAG editing using the hash tables described above. Our implementation provides a limited number of editing tools, such as 3D painting (placing new voxels), recoloring (updating the voxel attributes), and a tool that copies a small region of the scene to a new location. We chose these as they can be considered representative of a variety of operations - additional tools could easily be added.

SVO CONSTRUCTION

The first step of editing consists of constructing a Sparse Voxel *Octree* (SVO) of the scene as it is after the editing operation, which is technically a graph and not a tree, as the unmodified regions of the scene still refer to the SVDAG (Figure 4.4). We reserve a couple of bits of the node header to indicate for each child pointer whether it points to the SVO or the main SVDAG (*reserved* bits in Figure 4.2).

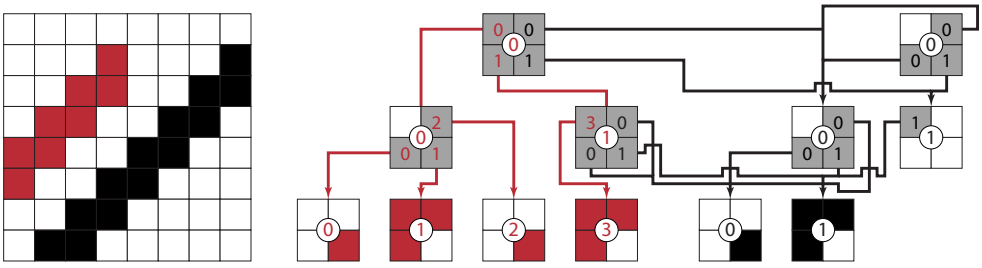


Figure 4.4.: The existing SVDAG is traversed by the editing tool (red) to create a Sparse Voxel *Octree* (SVO) of the scene after editing. This is technically not an octree because it contains pointers to the original SVDAG.

The SVO is constructed following a top-down breadth-first traversal of the SVDAG. At each level, we collect the following information about

the current nodes: their three-dimensional position, size, and whether they are homogeneous (completely filled with a single material). This information is sent to the editing tool, which decides whether to either subdivide a region, fill it, or keep it as-is.

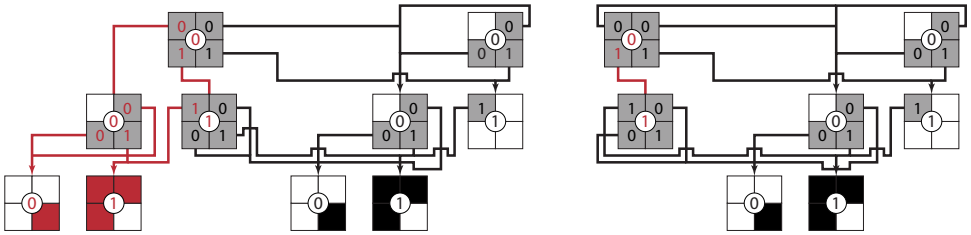
Simple editing tools like the 3D paint brush do not interact with the existing scene; they simply overwrite it. However, the copy tool must traverse both the source and target regions. To do so efficiently, an editing tool can maintain a state during traversal; something that is not supported by the HashDAG framework [28]. This allows the copy tool to perform a dual traversal of the source- and target-copy regions.

The traversal process is implemented using two GPU kernels: one for inner nodes and one for the leaves. Inner nodes are processed by groups of eight threads, each determining whether one of the eight child regions must be visited. These are appended to a work-queue using a combination of warp intrinsics and an atomic counter. After each level, the counter is copied to the CPU in order to determine how many items the next level may at most produce. Since leaves represent regions of $4^3 = 64$ voxels, we spawn 64 thread workgroups so that each thread processes a single voxel. The leaves are then constructed in shared memory before being copied to global memory.

Merging into the existing SVDAG

After the intermediate SVO is constructed (Figure 4.4), it must be merged into the existing SVDAG. This process is performed level by level, starting from the bottom, which allows efficient detection of duplicate subtrees (Chapter 2). We divide the duplicate elimination process into two parts: removing duplicates within the SVO, and removing duplicates between the SVO and the existing SVDAG (Figure 4.5).

In a single-threaded application, one could simply iterate over the SVO nodes/leaves, trying to find them in the SVDAG, and inserting them if they are not present yet. Running the same algorithm in a massively parallel environment requires that search and subsequent insertion are a single atomic operation. This requires locking, which limits scalability, especially with many duplicate items (which map to the same hash table buckets). We work around this problem by eliminating duplicates before we search and insert them into the SVDAG. We have experimented with duplicate detection through sorting, but the performance cost dominated the editing time. Instead, we construct a second hash table into which all nodes/leaves at the current level are inserted, along with a pointer to their location in the SVO. Note that we intentionally allow spurious duplicate



SVDAG after editing [1] SVDAG before editing SVDAG after editing [2]

Figure 4.5.: The Sparse Voxel *Octree* representation of the scene after editing is turned into a SVDAG by merging duplicates within itself (left). It is then merged into the original SVDAG, resulting in an SVDAG with two root nodes representing the scene at different points in time (right).

insertions. A subsequent search operation of each item selects a single representative.

After performing the previous operations, we are left with two SVDAGs (Figure 4.5). To merge them, we check whether each node/leaf in the new SVDAG was already present in the original SVDAG. If this was not the case, then they are inserted into the original SVDAG. Finally, child pointers of the parent SVO nodes must be updated to the new memory addresses. During this step, we also check whether the updated subtrees are fully empty or fully filled with a homogeneous material. The latter information can be used to accelerate some editing tools and is stored in the node header (Figure 4.2). The entire process is repeated for the parent levels until we reach the root node.

4.4. Implementation Details

In this section, we will discuss some of the implementation details with regard to the hash tables and the SVDAG.

4.4.1. Hash Tables

The implementation of our SVDAG consists of multiple hash tables, each storing items of one specific size. Nodes and leaves share the same hash table, which means that a single piece of memory may store both a node and a leaf at the same time when their bit pattern matches. As mentioned in Section 4.3.2, both the insertion and search operations are performed

by 32 threads (one warp) per item. The pseudocode for both operations can be found in the Appendix A.

4.4.2. Finding Duplicates in the SVO

As discussed in Section 4.3.3, we separate the process of eliminating duplicates within the intermediate SVO from the elimination of duplicates between it and the SVDAG. All nodes and leaves in the SVO are padded such that they are the same size in memory and stored in a single contiguous array per level. To eliminate duplicates within these arrays, we initially experimented with sorting but found that using a specialized hash table provides better performance.

The requirements for this duplicate detection hash table are more relaxed compared to the SVDAG hash tables (Section 4.3.2). While this does open the door for open addressing schemes, we stick with the slab approach for simplicity. We use a very similar memory layout to the 64-bit atomic scheme (Figure 4.3) with some notable changes. Since the hash table needs to store both keys and values (SVO pointers), we store an additional 32-bit value for each slot at the end of the slab.

With many duplicates, we run the risk of creating long linked lists of slabs for some buckets. To reduce this issue, we check, when an item is inserted, whether it is already present in the first slab. If the first slab already contains the key, then the insertion is canceled. This may not catch all duplicate insertions, but it provides a good trade-off between insertion and search performance in practice.

In a second pass, we iterate over all items that were inserted into the hash table and search for them again. The search function is configured such that, in the case of duplicate insertions, it always returns the first key/value pair. We select one unique representative for each group of matching items by comparing the inserted items' memory index with that returned by the hash table.

4.4.3. Garbage Collection

Since nodes and leaves may have multiple parents, releasing them from memory requires reference counting or garbage collection. Reference counting adds an additional memory overhead, so we opt for garbage collection. Each slab of the hash table contains a 32-bit mask that indicates for each slot whether it is still being referenced. When the garbage collector is invoked, these masks are initialized to zero. We then traverse the SVDAG from top to bottom, using one kernel invocation per level, to

activate the respective bits. Finally, we iterate over the slabs in the hash table and clear the inactive slots. If, as a result, the slab does not contain any occupied slots, then it is removed from the bucket and returned to the memory allocator.

Our garbage collection is a proof-of-concept implementation and must be triggered manually. Because the implementation has not been optimized, it causes a momentary pause when garbage collection is invoked. This could theoretically be alleviated by running garbage collection asynchronously. Both SVDAG traversal and hash table iteration can be split into smaller steps, which can be interleaved with rendering. For example, one may traverse only a part of the SVDAG each frame or iterate over a subset of the hash table buckets. Alternatively, reference counting may deliver more consistent performance at the cost of increased memory usage.

4.5. RESULTS

To evaluate our solution, we first test the proposed hash table schemes in a separate benchmark, comparing them to existing work on GPU hash tables. Finally, we consider the entire SVDAG editing program as a whole. All evaluation was performed on an RTX4080 equipped system running PopOS 22.04 LTS.

4.5.1. HASH TABLES

We evaluate four different hash table schemes: *SlabHash* extended using *64-bit atomics*, *ticket board*, and *acceleration hash* using 32 or 8 bits (Section 4.3.2). Our first test consists of a simple hash table benchmark that aims to replicate the SVDAG editing behavior. We initialize each hash table with $2^{25} \approx 33M$ items and then perform a search operation and an insertion operation for $2^{23} \approx 8M$ million items.

Figure 4.6 shows the results of our benchmark at different load factors for an item size of 6 *WORDs* (24 bytes). Looking at insertion performance, we find that high load factors (lower number of buckets) result in improved performance despite an increase in thread contention. This is explained by an increase in the L1 and L2 cache hit rate. The opposite is true for search performance, which decreases because it needs to iterate over a longer list of buckets. Note that this is not the case for insertions, as there we typically find an empty slot in the first slab.

As expected, we observe that both insertion and search performance are

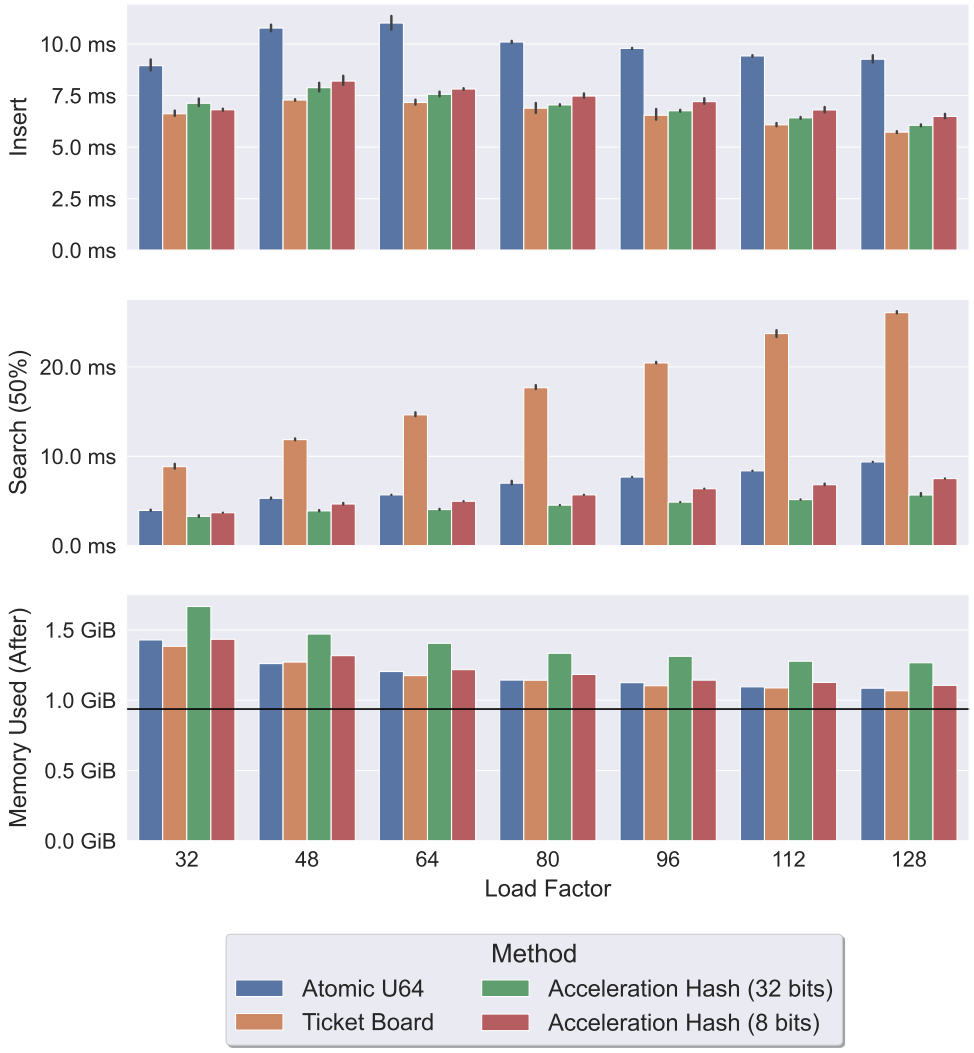


Figure 4.6.: Search and insertion performance as well as memory usage for different load factors. We search and insert 8M items into a table initially filled with 33M items of 6 *WORDs* (24 bytes) each. Memory usage is measured in slabs and thus includes unused slots. The black line indicates the size of the input data.

correlated with the number of bytes that need to be loaded from memory. This explains why the 64-bit method trails in search performance, as it checks more bytes before deciding whether an item might match.

The 32-bit acceleration hash leads to the best search performance, as it initially loads only half the amount of memory. Reducing the size of the acceleration hash did not improve the run-time performance, which could be attributed to various factors. For example, the slabs are not aligned anymore, resulting in reduced memory bandwidth (due to memory coalescing). Additionally, by allowing only 255 unique hash codes, we increase the probability of hash collisions occurring. We found that in practice, an additional 0.21 slots are checked per query. An interesting observation is that search performance may increase as the hit rate (percentage of successful searches) decreases. When an item is not present, the search operation needs to visit all slabs and cannot stop early (when an item is found). However, comparing an item after a potential match is expensive due to non-coalesced memory accesses and branching. Especially for larger items, traversing all slabs during an unsuccessful search can be faster than having to compare an item.

Considering memory usage, we see that the additional 32 bits of acceleration hash cause significant overhead compared to the other three methods. The 64-bit atomic method uses additional memory for padding, whereas the other methods (additionally) spend some memory on acceleration hashes/ticket boards. Note that the results in Figure 4.6 include the memory used by empty slots, that is, the hash tables could potentially fit more items without growing in size.

Table 4.2 shows the results of the same benchmark for different item sizes. We include both SlabHash [50] and DyCuckoo [49] for comparison. As expected, insertion and search performance both scale with item size. Interestingly, we found that item sizes consisting of an even number of *WORDs* typically perform slightly better than item sizes with an odd number. We suspect that this is due to those items straddling cache lines more frequently. Comparing our methods with SlabHash, on which they are based, we see that insertion performance is comparable, while search performance is reduced. One of the reasons for this decline is that, in order to support dynamic memory growth, we require an extra level of pointer indirection to access a slab. Although this is mentioned in the SlabHash paper, it is not implemented in the published code. Our methods also require more memory bandwidth (64-bit atomics) or additional computation (acceleration hash), which introduces some overhead.

4.5.2. SVDAG EDITING

In this second part of the evaluation, we will look at the run-time performance of our SVDAG editing system compared to the CPU-based

Scene	Method	Memory
San Miguel 64K 4-bit Materials	Nodes/Leaves Only	2929 MiB
	Atomic U64	3509 MiB
	Ticket Board	**3461 MiB**
	Acceleration Hash (32 bits)	4269 MiB
	Acceleration Hash (8 bits)	3622 MiB
Citadel 128K No Materials	Nodes/Leaves Only	980 MiB
	Atomic U64	1199 MiB
	Ticket Board	**1155 MiB**
	Acceleration Hash (32 bits)	1400 MiB
	Acceleration Hash (8 bits)	1203 MiB
Citadel 128K 4-bit Materials	Nodes/Leaves Only	5997 MiB
	Atomic U64	7164 MiB
	Ticket Board	**7082 MiB**
	Acceleration Hash (32 bits)	8685 MiB
	Acceleration Hash (8 bits)	7404 MiB

Table 4.1.: Memory usage of the tested scenes both with (4-bit) and without (N/A) materials. This includes memory that is allocated but not currently used (partially filled slabs).

HashDAG framework [28]. For both methods, we target a hash table load factor (average number of items per bucket) of 96 directly after the scene has been loaded. We used the same test scenes as in [28] but converted the colors into a palette of 16 unique materials with similarly colored textures, mimicking popular voxel-based games. The test scenes used are Citadel and San Miguel (Figure 4.7), which are voxelizations of textured triangle meshes. Citadel is stored with a voxel resolution of $128K^3$, whereas San Miguel uses a resolution of $64K^3$. The reason for this difference is the elongated shape of the Citadel scene, which results in a higher level of sparsity and thus a smaller SVDAG size.

Table 4.1 shows the memory usage for both scenes. Including materials in the SVDAG significantly increases memory usage because of a combination of larger leaves and a reduction in repetition (since subtrees only match if shape and material are identical). Materials are assigned on the basis of the diffuse textures of the original triangle mesh. This creates a high level of local material detail that may not be present in other data sets. Despite this, storing the same scenes in a Sparse Voxel Octree (SVO) would have required roughly four times more memory ($22GiB$ Citadel,

$14GiB$ San Miguel) compared to our SVDAG. When only binary voxel occupancy is stored, without materials, the SVDAG uses only 6.5% of the memory of an SVO.

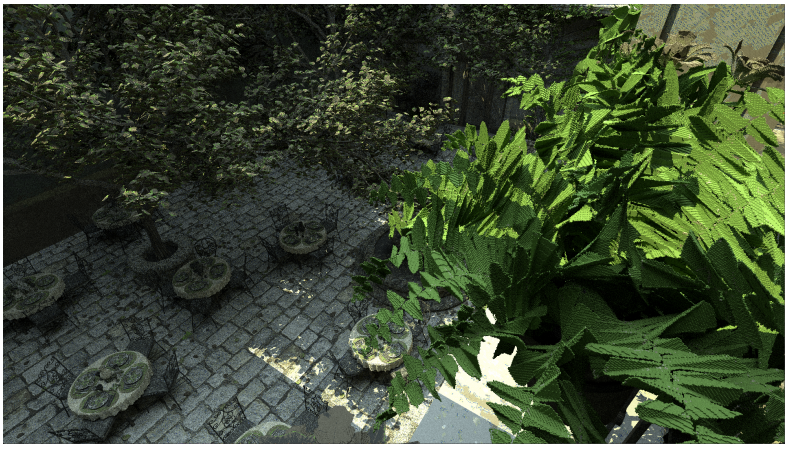


Figure 4.7.: Vegetation in the San Miguel scene.

We test our method using two different tools. We start with a *sphere tool*, which acts as a three-dimensional paint brush, placing voxels within the radius of a spherical brush. This creates a stress test for duplicate detection, as the surface of the sphere has many repeating patterns. Note that filling large homogeneous regions of space such as the interiors of the spheres is handled as a special case in our implementation. For the second test, we applied the copy tool to gradually make a copy of some of the vegetation in the San Miguel scene. This tool copies a cubic region around the mouse cursor. For larger objects, the user has to drag the cursor along the surface to progressively copy it.

The results of both tests are shown in Figure 4.8. The placement of large spheres in the Citadel scene takes between $15ms$ and $18ms$, depending on whether the materials are enabled. The radii of the spheres vary between 1140 and 1820 voxels per frame, which is reflected in the frame times. The largest occupy roughly 25 billion voxels, creating an intermediate Sparse Voxel *Octree* of 114MiB when materials are enabled. Note that this octree does not subdivide the interiors of the spheres. SVDAG compression reduces this to 23MiB for the first sphere and to merely 15MiB for subsequent spheres as geometric repetition increases. The copy operation in the San Miguel scene is distributed over many frames, taking only a couple milliseconds each frame. This is well within the realm of real-time frame rates.

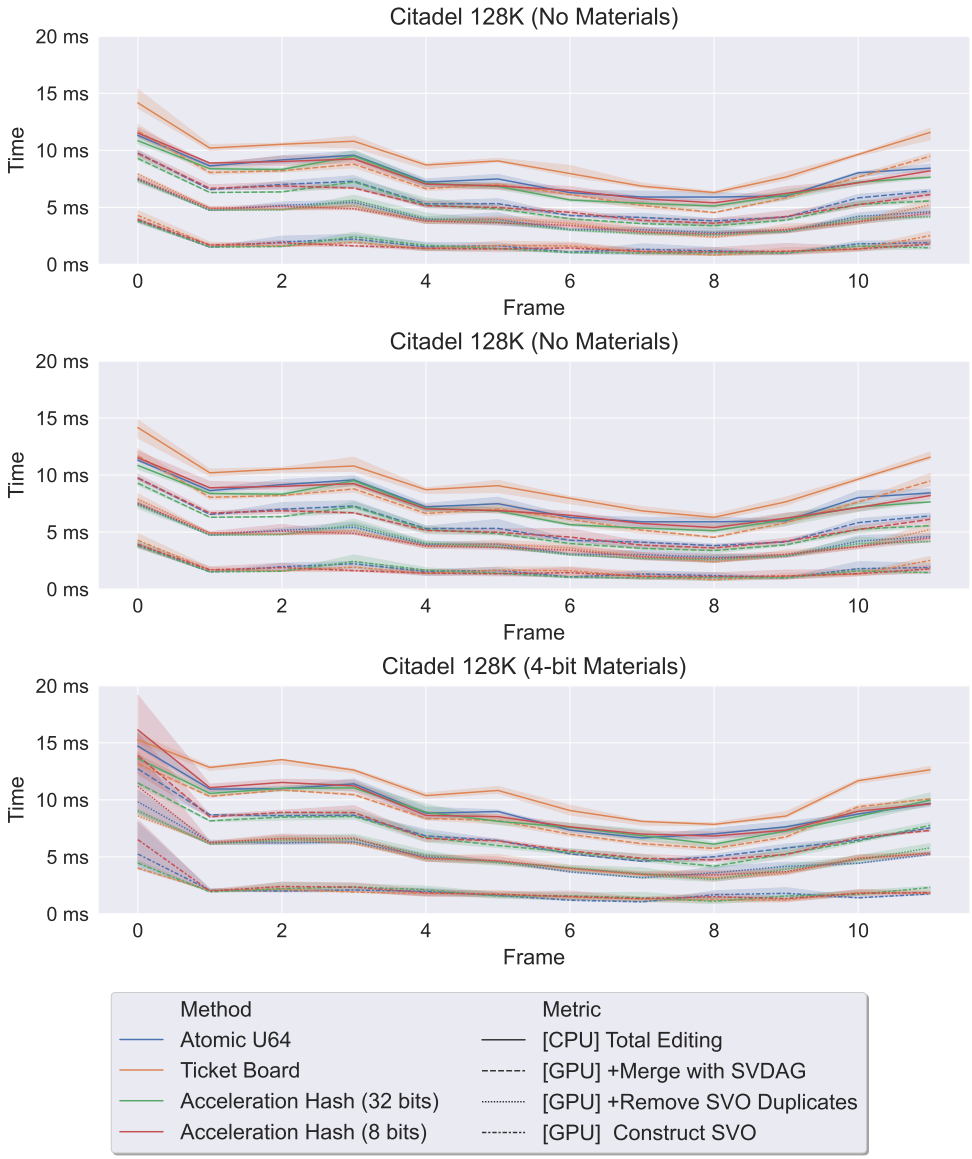


Figure 4.8.: Breaking down editing performance in San Miguel $64K^3$ and Epic Citadel $128K^3$, as measured on the CPU, into the most costly GPU kernels. The remaining time is spent on smaller GPU kernels, memory allocation, synchronisation and other CUDA related overhead.

Figure 4.8 breaks down the total editing time (CPU wall clock) into the

most costly GPU kernels (GPU time). Considering the Citadel scene without materials; traversing the editing region and constructing the temporary Sparse Voxel *Octree* takes $1.7ms$ on average. Removing duplicate nodes and leaves within this octree (Section 4.4.2) is a relatively costly operation at just over $2.5ms$. Finally, finding duplicates between the SVO and SVDAG and inserting the unique items takes between $1.3ms$ and $3.2ms$ depending on the hash table. These timings increase by roughly 30% when enabling materials due to an increase in SVDAG leaf sizes. The remaining time is spent on various other GPU kernels (updating parent pointers, detecting homogeneous regions, etc.) and some CPU/GPU synchronization. We found that this synchronization is negatively influenced by memory allocation calls, such as $cudaMalloc$ or $cudaMallocAsync$, which is why we replace all memory allocations with the Vulkan Memory Allocator [51].

From the four proposed hash table implementations, the two acceleration hash methods perform best, followed by the 64-bit atomics method. This is exclusively due to a difference in search performance with $0.94ms$ for the *atomics* method versus $0.87ms$ and $0.70ms$ for the *8-bit and 32-bit acceleration hash* methods, respectively. The *ticket board* method performs significantly worse with $2.51ms$. The cost of inserting new nodes and leaves into the hash tables is comparable for all four methods at roughly $0.6ms$. Based on both run-time performance and memory usage, we conclude that the *atomics* and the *8-bit acceleration hash* methods are most suitable for our use case.

To compare against existing work, we perform the same editing operations using HashDAG [28]. Some of the improvements we made, such as faster memory allocation, were also ported to HashDAG since our method was implemented in the same codebase. To ensure a fair comparison, we disable the voxel colors, as they are not directly comparable to our material system. It takes HashDAG $52.6ms$ to place the spheres in the Citadel scene using 32 CPU threads, which is roughly 5 times slower than our GPU-based method. Additionally, HashDAG requires $192.8ms$ to copy these changes to the GPU, which is necessary to render the updated scene.

4.6. CONCLUSION

Sparse Voxel Directed Acyclic Graphs (SVDAG) have proven to be a memory-efficient data structure for storing dynamic sparse voxel data. In this chapter, we have shown that it is possible to edit SVDAGs entirely on the graphics card, vastly improving editing performance compared to the previous CPU method and circumventing the expensive CPU/GPU

memory copies. The main building block is a GPU-based dynamic hash table, which can store large items and provides pointer stability. We propose four different implementations inspired by SlabHash [50]. Despite our additional requirements, stemming from the SVDAG support, the performance of our solution remains close to the state-of-the-art.

Our GPU-driven SVDAG editing pipeline consists of almost a dozen compute kernels and an additional hash table optimized to deal with duplicate removal. We enable large SVDAG modifications in real-time, while reducing memory usage by several factors compared to an octree.

Our application supports simple voxel attributes stored within the SVDAG. Although this works well for small and coherent attributes, additional memory savings may be achieved by separating geometry from attributes and compressing them separately. However, current methods [30, 31, 52] are not optimized for many fragmented modifications. We believe this to be an interesting avenue for future work.

Size	Method	Insert	Search (25%)	Search (50%)	Search (75%)	Memory
1	SlabHash	5.8 ms	**3.1 ms**	**2.8 ms**	**2.4 ms**	-
	DyCuckoo	**4.7 ms**	3.1 ms	2.8 ms	2.4 ms	-
2	DyCuckoo	5.5 ms	**3.5 ms**	**3.1 ms**	**2.7 ms**	-
	Atomic U64	6.9 ms	7.6 ms	6.9 ms	6.2 ms	**318.7 MiB**
	Ticket Board	**5.4 ms**	11.0 ms	9.9 ms	8.9 ms	328.4 MiB
	Accel Hash (32)	5.8 ms	4.0 ms	4.1 ms	4.3 ms	478.1 MiB
	Accel Hash (8)	5.9 ms	5.6 ms	5.6 ms	5.4 ms	362.7 MiB
3	Atomic U64	9.6 ms	7.9 ms	7.6 ms	7.3 ms	**478.1 MiB**
	Ticket Board	**5.7 ms**	13.3 ms	13.4 ms	11.0 ms	485.1 MiB
	Accel Hash (32)	6.2 ms	**4.1 ms**	**4.4 ms**	**4.8 ms**	637.3 MiB
	Accel Hash (8)	6.4 ms	5.9 ms	5.9 ms	5.9 ms	519.4 MiB
4	Atomic U64	9.5 ms	7.9 ms	7.5 ms	7.1 ms	**637.4 MiB**
	Ticket Board	**6.0 ms**	15.8 ms	14.5 ms	13.2 ms	642.0 MiB
	Accel Hash (32)	6.2 ms	**4.0 ms**	**4.3 ms**	**4.5 ms**	796.7 MiB
	Accel Hash (8)	6.6 ms	5.9 ms	6.0 ms	6.0 ms	676.3 MiB
5	Atomic U64	10.0 ms	8.2 ms	7.9 ms	7.7 ms	**796.2 MiB**
	Ticket Board	**6.2 ms**	18.8 ms	17.3 ms	15.8 ms	798.6 MiB
	Accel Hash (32)	6.6 ms	**4.3 ms**	**4.8 ms**	**5.2 ms**	955.6 MiB
	Accel Hash (8)	6.9 ms	6.1 ms	6.3 ms	6.4 ms	832.9 MiB
6	Atomic U64	9.8 ms	8.1 ms	7.7 ms	7.3 ms	956.1 MiB
	Ticket Board	**6.4 ms**	22.7 ms	20.4 ms	19.2 ms	**955.7 MiB**
	Accel Hash (32)	6.8 ms	**4.4 ms**	**4.9 ms**	**5.4 ms**	1115.4 MiB
	Accel Hash (8)	7.1 ms	6.2 ms	6.4 ms	6.6 ms	990.1 MiB
7	Atomic U64	10.4 ms	8.4 ms	8.3 ms	8.2 ms	1115.1 MiB
	Ticket Board	**6.6 ms**	25.9 ms	23.7 ms	21.5 ms	**1112.4 MiB**
	Accel Hash (32)	7.0 ms	**4.6 ms**	**5.2 ms**	**5.7 ms**	1274.6 MiB
	Accel Hash (8)	7.3 ms	6.4 ms	6.7 ms	7.0 ms	1146.7 MiB
8	Atomic U64	10.3 ms	8.3 ms	8.1 ms	8.0 ms	1274.3 MiB
	Ticket Board	6.8 ms	29.2 ms	26.8 ms	24.7 ms	**1269.2 MiB**
	Accel Hash (32)	**6.7 ms**	**4.3 ms**	**4.5 ms**	**4.8 ms**	1433.9 MiB
	Accel Hash (8)	7.5 ms	6.4 ms	6.7 ms	7.0 ms	1303.8 MiB
9	Atomic U64	10.7 ms	8.6 ms	8.6 ms	8.7 ms	1433.5 MiB
	Ticket Board	**7.0 ms**	29.3 ms	27.3 ms	24.9 ms	**1425.8 MiB**
	Accel Hash (32)	7.5 ms	**4.8 ms**	**5.5 ms**	**6.1 ms**	1592.6 MiB
	Accel Hash (8)	7.7 ms	6.6 ms	7.0 ms	7.4 ms	1459.9 MiB
10	Atomic U64	10.3 ms	8.4 ms	8.0 ms	7.6 ms	1593.2 MiB
	Ticket Board	**7.2 ms**	30.3 ms	27.4 ms	25.6 ms	**1582.6 MiB**
	Accel Hash (32)	7.5 ms	**4.9 ms**	**5.5 ms**	**6.2 ms**	1752.2 MiB
	Accel Hash (8)	7.8 ms	6.7 ms	7.1 ms	7.5 ms	1616.6 MiB

Table 4.2.: Searching 8M items and subsequently inserting 8M items into a hash table. The hash tables initially store 33M items targeting a load factor (after insertion) of 96. Search performance is evaluated for various hit rates (number of search operations that succeed). Item size is measured in *WORDs* (4 bytes). Memory usage is reported *before* insertions.

5

Editing Compressed High-resolution Voxel Scenes with Attributes

Mathijs Molenaar, Elmar Eisemann

Sparse Voxel Directed Acyclic Graphs (SVDAGs) are an efficient solution for storing high resolution voxel geometry. Algorithms for the interactive modification of SVDAGs that maintain compressed geometric representation have been proposed. Although compression techniques for voxel attributes, such as colors, exist; they do not operate in real time, leading to high run-time memory usage. In this chapter, we introduce two attribute compression methods (lossless and lossy), which enable the interactive editing of compressed high-resolution voxel scenes including attributes.

Parts of this chapter have been published in Computer Graphics Forum Volume 42, Issue 2 (2023) [52].

A downside of voxel representations is the typical high memory requirements. Because graphics card memory is very limited on consumer hardware, often just a couple of gigabytes, compression schemes are required (Sect. 5.1). Nevertheless, these often imply that access and manipulation of the data becomes slower or even impossible at interactive rates.

In this work, we focus on voxel models of 3D boundary representations, which lead to sparse voxel occupancy. Various data structures have been developed to reduce the storage requirements of this sparse voxel geometry by omitting empty voxels [17, 53] or exploiting redundancy using a DAG encoding [23].

Along with voxel occupancy, there is often a need to store additional information inside non-empty voxels, e.g., presence of colors for 3D painting. These attributes can be decoupled and compressed separately from the geometry which improves the compression ratio [30]. Various previous works propose specialized methods for compressing attribute data [30, 31]. However, none of these works focuses on compression time or enables 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 by 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.

5.1. BACKGROUND & RELATED WORK

Sparse Voxel Directed Acyclic Graph (SVDAG) exploits repeating patterns in a SVO, which manifest themselves as identical subtrees. Having parent nodes refer to a single remaining subtree saves memory. First expressed by [21] in two dimensions, it was later extended to 3D by [22] and popularized by [23]. Extensions to this approach match identical subtrees under symmetry [26, 27] or allow approximate merges [25]. For more details, we refer the reader to Section 2.

Storing attributes, such as colors, in an SVDAG is a challenge. SVOs can store such information in leaves. However, doing so in an SVDAG reduces the number of subtrees that can be merged, since both geometry and attributes must match to be considered a duplicate. Williams [29] proposed to decouple the voxel attributes by collecting them along a space-filling curve (Figure 5.1). The Morton space-filling curve [54] was chosen because

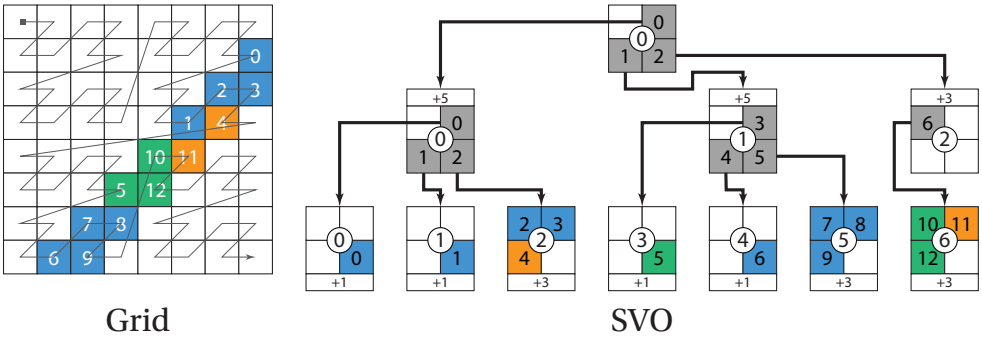
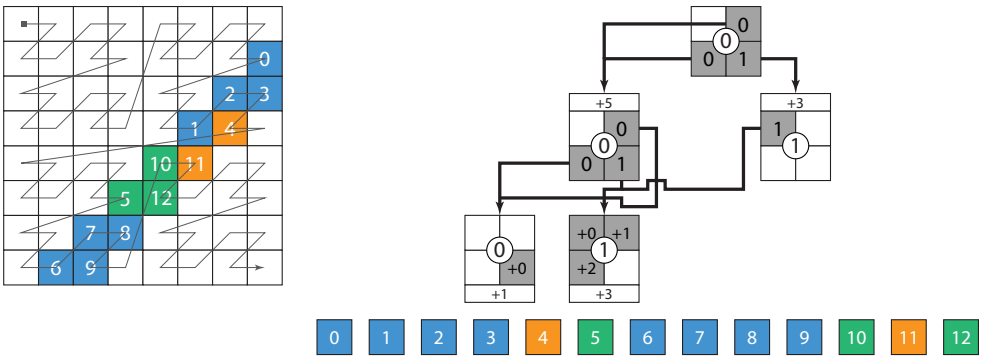


Figure 5.1.: The structure of the voxel grid (left) is captured in an SVO (right). Each SVO node stores the number of voxels in its subtree. The Morton order indices can be computed from the SVO by summing the number of preceding voxels.

it is equivalent to a depth-first depth traversal of an SVO or SVDAG. The attributes of non-empty voxels are stored in a one-dimensional array. The index of each voxel is determined by the number of preceding non-empty voxels along the Morton curve. To accelerate the computation of these indices, the SVDAG stores, for each subtree, the total number of non-empty voxels in that subtree (Figure 5.2). Williams [29] store these numbers in the parent nodes, while later works reduce the memory overhead by storing this information either in the child pointers [30] or the child nodes [31].

The method described by Williams [29] decouples geometry from attributes. This ensures optimal geometric reuse in the SVDAG but leaves the attributes uncompressed. Dado *et al.* [30] propose a lossless compression scheme specifically designed to compress the (one-dimensional) array of colors. While the compression stage is performed in an offline preprocess, the decompression of individual colors is possible in real-time. This allows the colors to remain in compressed format when rendering. In the same vein, Dolonius *et al.* [31] propose a lossy compression scheme for the SVDAG attributes (colors). Lossy compression means that the decoded colors might change slightly from the original colors in an attempt to save more memory.

The latter technique[31] was used in the HashDAG framework [28] to interactively modify the SVDAG structures. The program loads an existing SVDAG scene with colors that are compressed with the Dolonius *et al.* [31]



5

Figure 5.2.: The structure of the voxel grid (left) is captured in an SVDAG (right). Each SVDAG node stores the number of voxels in its subtree. The Morton order indices can be computed from the SVO by summing the number of preceding voxels.

method. During the editing process, the user may fill in new voxels or change the colors of existing voxels. This is implemented in HashDAG [28] by embedding uncompressed colors in a memory format compatible with Dolonius *et al.* [31]. Thus, prolonged editing sessions will lead to a stark increase in color memory as more of the scene is replaced by uncompressed colors. The creation of a new voxel effectively shifts the attribute indices of all subsequent voxels in the attribute array, which is an $\mathcal{O}(N)$ operation. To alleviate this issue, HashDAG splits the color array into smaller *chunks*, one for each inner node at a predetermined level of the SVDAG. This defines an upper bound on the number of attributes that must be relocated during editing. It also creates an opportunity for multi-threading as chunks can be updated independently.

In this chapter, we will present a real-time compression algorithm that is compatible with the memory layout of Dolonius *et al.* [31], and is integrated in HashDAG [28]. This allows for real-time editing of scenes without inflating their colors. We also present an alternative *lossless* compression scheme that is implemented on the GPU, resulting in even higher run-time performance.

5.2. Our method

We will now describe our two distinct SVDAG color compression algorithms that provide real-time performance. The first is a novel lossless compression scheme in which compression is implemented entirely on the GPU. The second is an algorithm for lossy compression of colors in a memory format that is compatible with the Dolonius *et al.* [31] method.

5.2.1. Lossless Attribute Compression

Real-time editing of SVDAGs with attributes requires not only fast compression, but also efficient decompression of individual attributes. The latter excludes most popular sequential techniques, such as Lempel-Ziv (LZ) [55–62] and Huffman coding [63], which do not support random access decoding. Our approach must achieve constant-time access to individual attributes while achieving a high compression ratio. Although our method could be adjusted to store arbitrary values; we simplify our explanations by staying in the context of color compression.

The lossy method of Dolonius *et al.* [31] already splits the color array into large *macro* blocks, which can be used to parallelize CPU processing. To achieve massive parallelism on the GPU, we split the color array into much smaller blocks of 128 consecutive entries, each of which is processed by a group of GPU threads. Our underlying assumption is that the color array exhibits limited local change, which we exploit in our compression scheme. We normalize the colors by computing a minimum C^i_{min} and a maximum C^i_{max} per color channel i in each block. The colors inside a block are interpreted as an offset to C^i_{min}. The offsets are encoded using $B^i = \text{bitwidth}(C^i_{max} - C^i_{min})$ bits per color channel i.

As a second step, a *Frame-Of-Reference* compression [64] is applied. Instead of directly storing an offset to C^i_{min}, we store an offset to a suitable reference color. This leads to smaller offsets, which can be stored with fewer bits. The reference colors are determined by quantizing the normalized colors into eight equally sized bins (3 bits) per channel.

We expect the same reference color to occur multiple times within a block; thus, we store them in a per-block look-up table and refer to them by index. Furthermore, consecutive colors are likely to share the same reference color. So, rather than storing a reference color (index) for each color, we instead store the reference index only if it changes from one color to the next. This information is encoded with a 128-bit mask for the entire color block (1 bit per color). To decode a color, the corresponding reference color is computed using a *popcount* instruction.

All of this information is stored in a compact memory format as shown in figure 5.3. Each block may occupy a variable amount of memory, so the 64-bit start location of each block is stored in a secondary array. Together, the fixed cost per block is $48 + 7 + 128 + 64 = 247$ bits, resulting in a minimum theoretical compression ratio of $\frac{247}{128*24} \approx 8.04\%$ for 8-bit RGB colors.

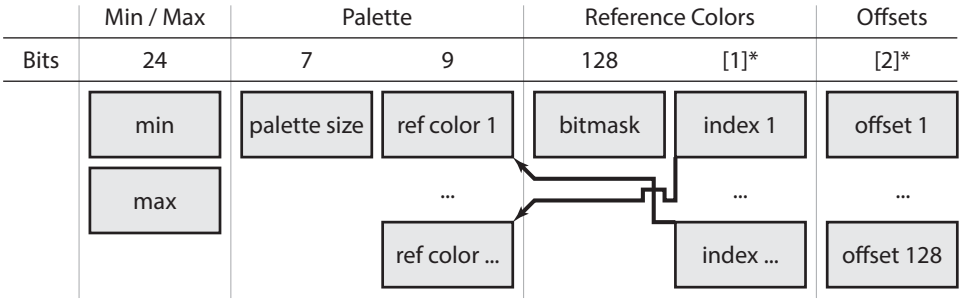


Figure 5.3.: Memory layout of a single block of 128 colors as compressed with our lossless compression method.
$[1]^* = \lceil \log_2(\text{palette size}) \rceil \qquad [2]^* = \sum_{c \in r,g,b} \lceil \log_2(\frac{c_{max} - c_{min}}{8}) \rceil$

Because blocks cover a fixed interval of colors, there is no option for fast insertions and removals. Instead, we rely on the HashDAG framework to split the color array into smaller *chunks* to limit the computation time (Section 5.1). When editing, the affected color chunks are decompressed into a temporary buffer (by the GPU), modified, and subsequently compressed again.

5.2.2. 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 [65] with their impressive compression ratios. Dolonius *et al.* [31] introduced a lossy scheme aimed specifically at compressing the one-dimensional color array associated with an SVDAG. Their specific compression algorithm is designed to achieve high quality but is too slow for real-time updates. We present a fast, novel algorithm for color compression that is compatible with their file format. We will now briefly explain their method before explaining our algorithm in more detail.

S3TC block image compression schemes divide an image into a regular grid of 4×4 pixel blocks. The colors inside each block are projected onto a

line segment in RGB color space and stored as an interpolation coefficient between the two endpoints. Dolonius *et al.* [31] applies the same concept to the one-dimensional color array but using variable length 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 using 16 bits in a RGB565 format. Blocks can also encode a run of a single color without loss of precision when desired. For additional information on memory layout, we refer the reader to [31].

To compress a set of input colors, the algorithm of Dolonius *et al.* initially considers each color as a separate block. They then sequentially merge neighboring blocks by applying a least-squares line fit to the colors. This process is repeated until no two neighboring blocks can be merged without exceeding a given error threshold. The number of bits used to store the interpolation bits impacts the final error; therefore, they test all possible interpolation bits, creating a tree of potential color blocks. The final selection of blocks is decided by a tree cut.

While providing high-quality results, the method of Dolonius *et al.* is too slow for real-time use. If not more than two neighboring colors fit in each line segment without exceeding the error threshold, then the algorithm finishes in $\Theta(N)$ time. However, if all colors can be assigned to a single line segment, then we encounter the worst-case time complexity of $\mathcal{O}(N \log N)$. This is problematic as larger scenes generally contain more coherence as a result of oversampling of textures, creating larger color blocks.

Our line fitting

Our solution increasingly relies on localized decision-making. In the following explanations, we will generalize the problem of color compression to that of fitting a line through a stream of points. The goal is to find a collection of consecutive line segments that approach the input points within a user-provided threshold. Changing this threshold allows for control over the number of line segments (memory usage) and the quality of the fit (image quality).

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

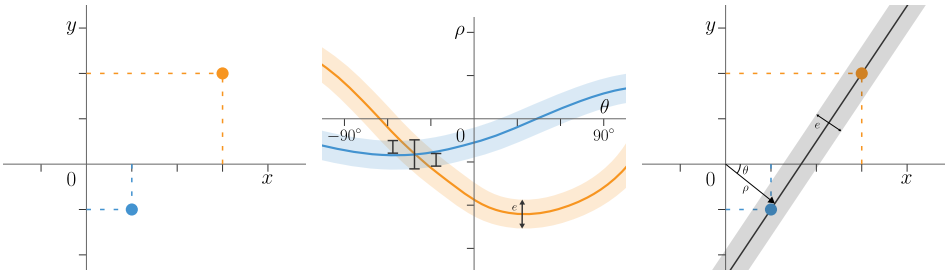


Figure 5.4.: Two points in (x, y) space (**left**) and their representation in (θ, ρ) space after Hough transform. (**center**). The intersection of the two lines in (θ, ρ) space defines a line in (x, y) which passes through both points (**right**). To allow for approximate fitting we define an error 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**)

Hough Transform [66] is a well-known operation in image processing to detect 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 Figure 5.4).

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

$$\begin{aligned} \vec{p} &= (x, y) \\ \rho(\theta) &= x\cos(\theta) + y\sin(\theta) \end{aligned} \tag{5.1}$$

Using additive blending [67], the texels with high values represent lines that pass through many points in the primal space. Using textures to represent the dual space is a form of quantization which makes the line fitting approximate.

Our goal is to incrementally map points to the dual space and check whether there is still a line that passes through all preceding points. However, using the typical texture approach introduces some problems. First of all, the quantization into texels leads to an unstable error margin. For example, the values 9.51 and 10.49 will map to the same texel, while 10.49 and 10.51 would not, despite being much closer in $\mathbb{R}$. The second problem is performance. For each new pixel, many different values θ

must be evaluated. Finally, the rasterization method requires the texture to be cleared before fitting begins, 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 that pass by the previously added points within the user threshold. This threshold creates a range of values for ρ that are considered valid for all preceding points (Figure 5.4).

To further accelerate the process, we observe that only lines that pass close to *all* points are relevant. Hence, we can skip the (discrete) steps θ that have no ρ that fits all the preceding points. We accomplish this by keeping a list of θ steps for which a line fit still exists (Figure 5.4).

EXTENSION TO 3D

Attributes usually consist of more than two dimensions. For example, colors are typically encoded as a triplet of red, green, and blue channels (RGB). The three-dimensional equivalent of the Hough transform (for example, employed in [68]) requires two angles, which is prohibitively expensive for our real-time compression 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 (simultaneously). As soon as the line fit fails for one plane, we stop. By considering the resulting two-dimensional lines as projections of the desired 3D line, we can reconstruct the 3D line as a plane intersection involving any two of the three planes. For example, using lines in the R&B and G&B planes (defined as $l(t) = \vec{o} + t\vec{d}$), we construct a 3D line as follows:

$$\begin{aligned} \vec{o_{rgb}} &= (\vec{o_{rb}}^{\,r} - \vec{o_{rb}}^{\,b}\frac{\vec{d_{rb}}^{\,r}}{\vec{d_{rb}}^{\,b}}, \vec{o_{gb}}^{\,g} - \vec{o_{gb}}^{\,b}\frac{\vec{d_{gb}}^{\,g}}{\vec{d_{gb}}^{\,b}}, 0) \\ \vec{d_{rgb}} &= (\frac{\vec{d_{rb}}^{\,r}}{\vec{d_{rb}}^{\,b}}, \frac{\vec{d_{gb}}^{\,g}}{\vec{d_{gb}}^{\,b}}, 1) \end{aligned} \tag{5.2}$$

Tracking all three planes allows us to choose the combination that maximizes the spread along a shared axis (b in Eq. 5.2), which reduces numerical problems due to division by values close to zero.

5.3. IMPLEMENTATION

Here, we discuss the details of the implementation and integration in the HashDAG interactive SVDAG editing framework [28].

Editing the SVDAG The HashDAG framework maintains two copies of the scene, one in CPU memory and one in GPU memory. Editing is performed by the CPU, and the changes made to its instance of the scene are copied to the GPU at the end of each frame. Editing tools, such as a sphere placement tool and paint brush, are provided by the HashDAG framework. We also provide a new *stamp* tool that applies a colored height field to a surface.

The array of voxel colors is split into chunks, one for each node at level 8 in the tree when using lossy compression (the same as HashDAG [28]) and level 7 when using lossless compression (where level 0 is the root node). 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 color chunks as an ordered stream of three basic operations: **copy** existing colors, **fill** a single color and **write** new colors. Once all color operations for a chunk have been recorded, the chunk can be compressed using either our lossless or lossy method on a worker thread.

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

Lossy colors Because the lossy compression scheme uses variable-sized blocks, it is possible to perform partial updates to a color chunk. If none of the colors in a block has changed, then the block is simply copied over. Color compression thus only needs to be applied to the array of new colors. We implement our greedy line fitting algorithm (Section 5.2.2) on the CPU, where the angle θ is discretized in 96 steps, which was empirically chosen. Calculation 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 is a line rather than a line segment, so a second loop is required to compute the endpoints of each segment. The number of bits used to store the interpolation 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 *et al.* [31], the threshold is defined as the maximum Euclidean distance between any input color and its compressed representation in the output. We control the line fit precision by adjusting the 2D threshold parameter e (Figure 5.4) which we set to half the desired 3D error.

In addition, approximate line fitting, quantization of interpolation weights, and quantization of the line segment endpoints (RGB565 format) will further increase the final color error. As such, a situation may occur where it is not possible to store the fitted colors in the same block. In such cases, we encode these colors into individual color blocks using the RGB101210 format (Section 5.2.2).

5.4. RESULTS

To evaluate the performance of our algorithms and HashDAG [28], we will compare against the lossless compression scheme of Dado *et al.* [30] and the compression algorithm of Dolonius *et al.* [31]. In case of Dolonius *et al.*, 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.

Color Compression In order to accurately measure just the compression performance, we extract raw attribute arrays from various scenes and compress them in whole. The scenes tested are voxelized versions of the Epic Citadel, Lumberyard Bistro Exterior, and San Miguel scenes using diffuse textures as voxel colors. In addition, we test the Citadel scene with Perlin noise as colors to emulate the low frequency signal of irradiance (Figure 5.5). For lossy color compression schemes, we use a color 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's CUDA version [31], the chunks are processed in parallel using

multiple CPU threads.

The results are shown in Table 5.1. For our *lossless* method, we tested both the original RGB888 input and RGB565. Storing RGB565 colors will lead to a small quality loss due to the conversion of the 24 bit input colors, but, being lossless, this error cannot grow over time when editing. We also include the method employed by HashDAG [28] in the table which uses the Dolonius *et al.* memory layout. The only for of compression applied by HashDAG is detecting consecutive equivalent colors, a simple form of run-length encoding.

Our lossless method is capable of compressing the data at more than 20GB/s, which is two orders of magnitude faster than the Dado *et al.* method [30]. At lower resolutions, our method achieves compression ratios comparable to Dado *et al.* However, our method does not scale as well to larger resolutions because of the fixed block size, and thus memory overhead per voxel. Increasing the block size would help, but grouping more colors also reduces their coherence in each block. In practice, we found that blocks of 128 colors provide a good balance for most scenes.

Note that the Citadel scene is a bit of an outlier with respect to memory scaling. The scene uses low-resolution textures and underneath the castle are large single-colored triangles resulting in long homogeneous ranges in the color array. The method of Dado *et al.* is capable of efficiently storing ranges of single colors, while our lossless method is bound by the use of blocks. The opposite happens when storing Perlin noise rather than diffuse textures. Colors are not repeated locally, resulting in Dado using more memory than the uncompressed input. In comparison, our method

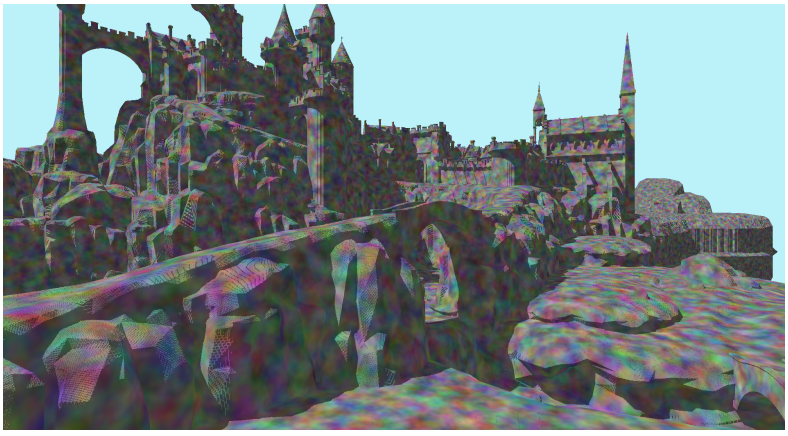


Figure 5.5.: The Epic Citadel scene using Perlin noise colors.

using local offsets handles this situation much better.

Our *lossy* compression algorithm was designed to achieve high run-time performance, while using the Dolonius *et al.* memory format [31]. At roughly 1GB/s on a 10 core CPU, our method is well suited to large-scale interactive editing or small edits in real time. In terms of compression ratio, our method typically requires between 30% and 80% more memory than the offline method by Dolonius *et al.*

Lossy methods trade quality for compression ratio, which is illustrated in Figure 5.6. The method of Dolonius *et al.* 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. Line fitting is performed using a least-squares optimization, 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 increase in 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 in which the Citadel scene is painted using a rainbow brush defined by a Perlin noise function for each of the three color channels. Figure 5.7 shows the usage of memory during editing, the time it takes to perform compression, update the colors, and the total frame time. The color update time includes the CPU to GPU copy

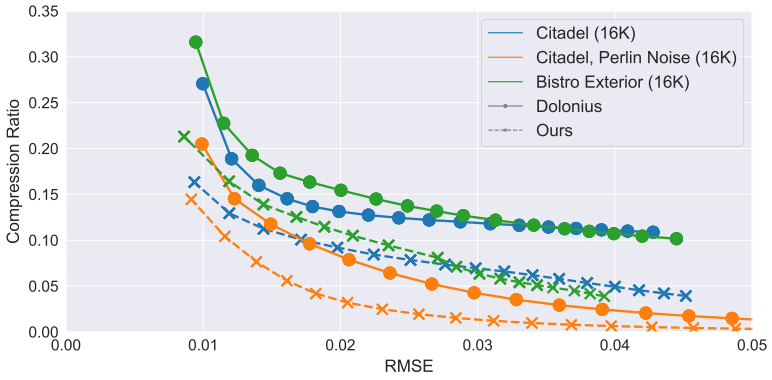


Figure 5.6.: A plot showing compression ratio as a function of Root Mean Square Error (RMSE), comparing our lossy method to Dolonius *et al.* [31].

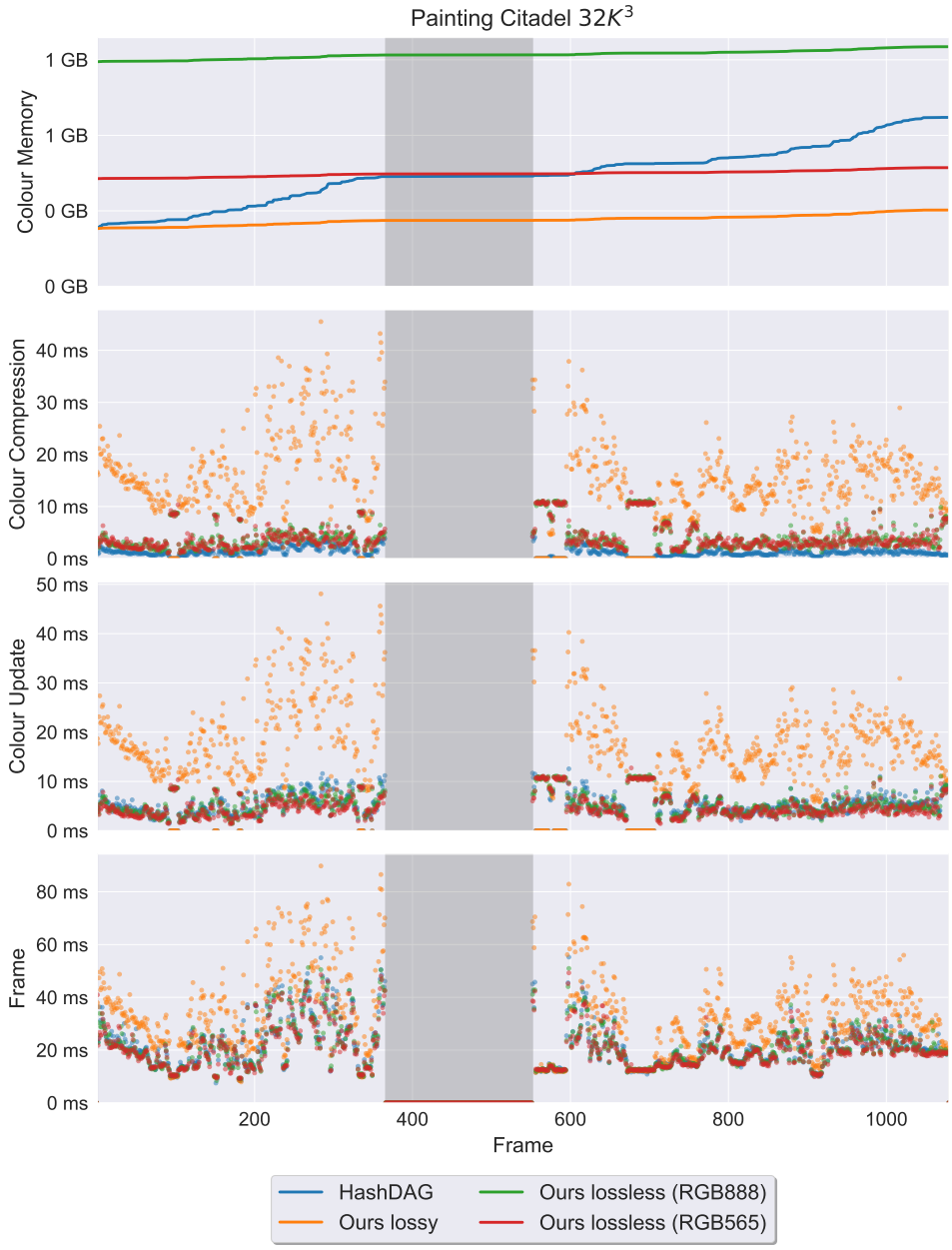


Figure 5.7.: From top to bottom: color memory usage, color compression-, color update- (including upload), and frame time, when painting the Epic Citadel $32K^3$ with a rainbow color brush. The gray area indicates frames in which no editing takes place.

required for the colors to be displayed on screen plus the compression step.

We compare our compression methods with the work of HashDAG [28] on which our code is based. HashDAG loads scenes using the same compression format as Dolonius *et al.* and our lossy method. Colors that are modified during editing are not compressed and instead encoded in individual color blocks, leading to high memory usage over time (Figure 5.7). 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 (5̃0%) to each frame. Although 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 5.1).

5.5. 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.

We have presented two methods for compressing attributes: a lossless and a lossy solution. This makes our approach suitable for many application scenarios. The lossless method is constructed from well-known building blocks [64] 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 color errors and reduced performance. We believe that our solutions increase the viability of using SVDAGs in interactive applications, as attributes such as colors 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 used 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.

Scene	Method	Time	Compression	RMSE
Bistro (8K) 649MB	Ours (lossy)	0.77s	18.44%	0.0158
	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 (16K) 2643MB	Ours (lossy)	3.25s	17.31%	0.0156
	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) 210MB	Ours (lossy)	0.22s	16.04%	0.0161
	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) 844MB	Ours (lossy)	0.80s	14.52%	0.0161
	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) 3412MB	Ours (lossy)	3.19s	13.61%	0.0161
	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 (16K) Perlin Noise 844MB	Ours (lossy)	0.61s	9.61%	0.0178
	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) 1974MB	Ours (lossy)	2.41s	17.15%	0.0154
	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 5.1.: The lossy methods target an error of 0.06 per color channel.

6

Conservative Ray Batching using Geometry Proxies

Mathijs Molenaar, Elmar Eisemann

We present a method for improving batched ray traversal as was presented by Pharr et al. [70]. We propose to use conservative proxy geometry to more accurately determine whether a ray has a possibility of hitting any geometry that is stored on disk. This prevents unnecessary disk loads and thus reduces the disk bandwidth.

Parts of this chapter have been published in the Eurographics Digital Library (2020) [69].

6.1. INTRODUCTION

Displaying large triangle scenes is an ongoing challenge in computer graphics. Generating realistic looking images requires accurate simulation of light, which is typically solved using Monte Carlo light simulation (path tracing). This process requires tracing millions of light rays through the scene and computing where they intersect the geometry. Acceleration structures such as a Bounding Volume Hierarchy (BVH) allow us to quickly compute a small selection of primitives which might intersect a light ray; allowing us to skip most computations. However, for very large scenes, both the geometry and the acceleration structure may not fit into system memory.

A major challenge in this field, which is called out-of-core rendering, is that some of the data must be offloaded to a larger storage device; typically Solid State Storage (SSD). The bandwidth and access times of even the fastest SSDs are significantly worse than those of system memory, resulting in a performance bottleneck. The goal of a good out-of-core renderer is to minimize the amount of memory that is loaded from disk and the time spent waiting for these loads.

In this paper, we discuss and improve batched ray traversal, which was first introduced in [70]. Batched ray traversal is a technique that improves the performance of in-core, out-of-core, and distributed rendering by making memory access patterns more coherent. We propose using Sparse Voxel Directed Acyclic Graphs (SVDAGs) as a compact proxy geometry. By intersecting with this proxy first, we reduce the amount of memory that needs to be loaded from disk, reducing the time it takes to render an image. Although we implemented this technique with batched ray traversal [70], it could also be applied to other out-of-core or distributed rendering frameworks.

6.2. RELATED WORK

The research field of large scene visualization can be divided into two categories. The first aims to minimize the memory requirements by reducing geometric complexity. This comes at the cost of a (slight) degradation in image quality. Alternatively, carefully designed renderers can visualize scenes at full geometric complexity by caching data to disk. The challenge in this second category is to reduce disk traffic and hide the latency associated with disk access.

Walt et al. [71] have shown a distributed ray-tracing system that is able to produce images at interactive rates by replacing geometric objects that

are not resident in memory with a low-resolution proxy. A similar concept for GPU out-of-core rendering can be used for volumetric data sets [5, 72]. Offline rendering solutions have also explored replacing geometry with proxies until a scene fits into memory [73]. Similarly, Yoon et al. [74] replace geometry with oriented planes based on a screen-space error function.

Most out-of-core rendering frameworks rely on a paging system to move data between disk and system memory [71, 75]. Alternatively, application-controlled data movement may provide more room for domain-specific optimizations; Christensen *et al.*[76] use application-controlled caching of surface tessellation to aid out-of-core traversal performance, while Wald, Slusallek, and Benthin[77] use a two-level acceleration structure hierarchy to manage data movement in a distributed system.

A common limitation of these works is that incoherent rays are difficult to handle efficiently, due to their incoherent memory access patterns, which are particularly pronounced in Monte-Carlo global-illumination solutions. To combat ray divergence, breadth-first traversal, ray stream traversal, and ray reordering schemes have been proposed. The breadth-first traversal of an acceleration structure by a collection of rays ensures that each node is touched at most once [78–81], but an "early-out" is impossible, as no front-to-back traversal can be ensured. Ray stream traversal techniques [82, 83] solve this, though at the cost of potentially visiting nodes multiple times. Finally, global ray reordering schemes [84, 85] sort rays before acceleration structure traversal in an effort to improve the resulting memory access patterns.

Batched ray traversal[70] aims to improve performance by increasing the coherence of memory access patterns. The scene is stored in a two-level acceleration structure. Rays are batched (enqueued) at the transition between the top and bottom levels of the hierarchy. When enough rays have been batched, the corresponding bottom-level acceleration structures are loaded into memory and ray intersection is performed. Batched ray traversal was designed for out-of-core rendering, but has also proven effective in-core rendering by reducing CPU cache misses [86–88].

6.3. Algorithm Overview

Here, we give a more detailed overview of our approach. We first discuss batched ray traversal and our specific implementation. Then we present the memory-efficient geometric proxy, which allows us to quickly and conservatively restrict the number of rays that require access to the lower-level acceleration structure. In an out-of-core system, this solution

reduces the dependence of disk accesses.

6.3.1. BATCHED RAY TRAVERSAL

Batched ray traversal is designed to improve memory coherency by grouping rays passing through the same region of space (see Figure 6.1). The objects in the scene are grouped into clusters, each of which forms a *batching point* for rays to be queued. A ray is queued at a batching point if it intersects the axis-aligned box that tightly bounds the batching point geometry. Each batching point contains an acceleration structure such that rays can be quickly intersected against the contained geometry. In the case of out-of-core rendering, such as ours, geometry is stored on disk and is loaded when needed.

The rendering process starts by spawning an initial set of camera rays and inserting them into the first batching point at which they intersect. The scheduler is then responsible for continuously selecting a batching point to load and traverse. We use a simple scheduler that always selects the batching point with the most rays. This ensures that the disk bandwidth is amortized over many rays. When a light path ends, a new camera ray is automatically spawned such that the number of rays in the system remains constant. If a ray needs to continue its traversal of the scene after visiting a batching point (e.g., it missed the geometry inside), then it is batched at the next batching point along its path.

The top-level acceleration structure needs to be able to efficiently restart traversal while storing as little state per ray as possible. We choose a 4-wide BVH following Gasparian [88]. Geometry intersections at batching points are implemented using Embree ray-tracing kernels. The

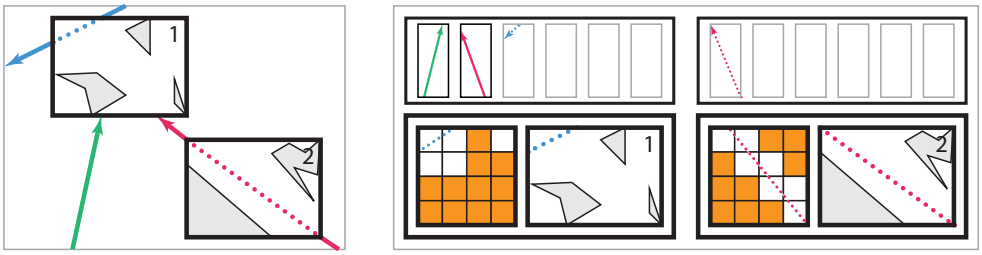


Figure 6.1.: The scene is spatially divided into batching points (left). Rays that hit the bounding box of a batching point are queued (right/top). Our proposal is to store SVDAGs which provide more accurate intersection information (orange), reducing the number of ray batch events (dotted rays).

acceleration structures are (re)built when needed and are stored in a Least Recently Used (LRU) cache.

6.3.2. Proxy Geometry

Our contribution is to store, for each batching point, proxy geometry that is both memory-efficient and conservative. When traversal of the top-level structure reaches a batching point, the ray is first intersected against this approximation. A ray is only batched if it intersects; otherwise, traversal of the top-level acceleration structure continues.

We approximate the geometry as a binary volume, as there are very efficient storage solutions that have been used in several real-time applications [23, 30, 36, 89]. A proxy volume supports both closed and open geometry without issues. However, care must be taken to ensure that the volume is conservative with respect to the geometry it represents.

In a preprocess, each batching point conservatively voxelizes [90] its corresponding geometry into a voxel grid. The binary grids are then converted to Sparse Voxel octrees and subsequently compressed to Sparse Voxel Directed Acyclic Graphs (Chapter 2). We generate multiple SVDAGs, one per batching point, but allow them to share nodes and leaves between each other. This can also be interpreted as a single SVDAG with multiple root nodes (one per batching point). Intersecting rays with such an SVDAG can be performed by most existing SVO traversal algorithms. We use the traversal algorithm presented by Laine and Karras[39].

6.4. Results & discussion

To evaluate our method, we implement a out-of-core unidirectional path tracer with the batching system described above. The scenes used are the crown model, the landscape scene, and the Moana Island scene by Walt Disney Animation Studios. The crown model was artificially subdivided 5 times to increase its primitive count. For the Moana Island we only render the triangle/quad control meshes, ignoring other geometric primitives such as curves which are not supported in our renderer. All objects use the same flat Lambert material in order to minimize shading cost.

Selecting the appropriate batching point size is important and nontrivial. More (smaller) batching points increase the memory overhead of the batching system and require more memory to store SVDAGs. The minimum number of triangles per batching point was empirically chosen for each scene, see Table 6.1. This results in 132, 335, and 1984 batching

Table 6.1.: Triangle counts of the tested scenes.

	Crown	Landscape	Island
Unique	860,272,002	25,947,395	142,771,030
Instanced	860,272,002	4,330,336,849	31,443,289,446
Per batching point	10,000,000	20,000,000	25,000,000

points for the crown, landscape, and island scenes, respectively. Our code is available at `https://github.com/mathijs727/pandora`.

6.4.1. VOXEL GRID RESOLUTION

The resolution of SVDAG proxy geometry affects memory usage, computational overhead, and culling effectiveness; see Figure 6.2. As expected, the memory usage of the SVDAGs scales cubic with respect to the voxel grid resolution. The large discrepancies in memory usage between the scenes can be attributed to the different numbers of batching points (and thus SVDAGs) per scene.

Increasing the resolution quickly has a diminishing effect on the number of times that rays are bached. Scenes with larger batching points, such as the island scene, seem to benefit more from high voxel resolutions. This is probably because the island scene has the most intricate geometric details.

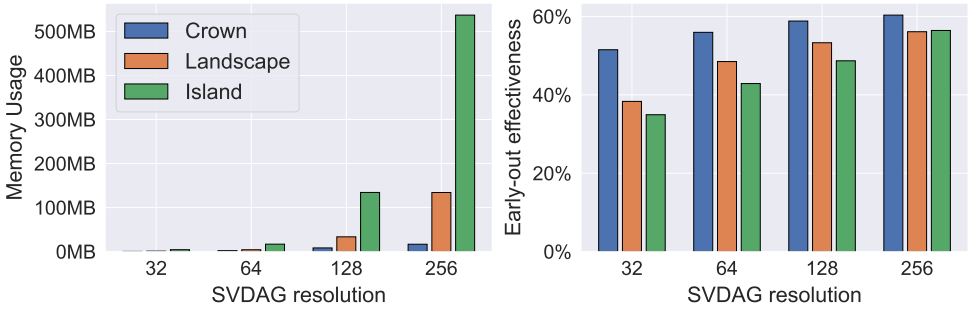


Figure 6.2.: Total SVDAG memory usage (left) and the effectiveness of our system (right) at different voxel grid resolutions.

6.4.2. Proxy Geometry

To evaluate the impact of the SVDAG proxy geometry on our system as a whole, we rendered all scenes with 128 samples per pixel with different memory budgets with a voxel grid resolution of 128^3. Figure 6.3 shows the reduction in the total disk bandwidth when our solution is enabled. We see a large difference between our method and the reference in the island and landscape scenes. The impact in the crown scene is more subtle. We suspect that this is caused by its relatively simple shape despite its high primitive count(due to our artificial subdivision).

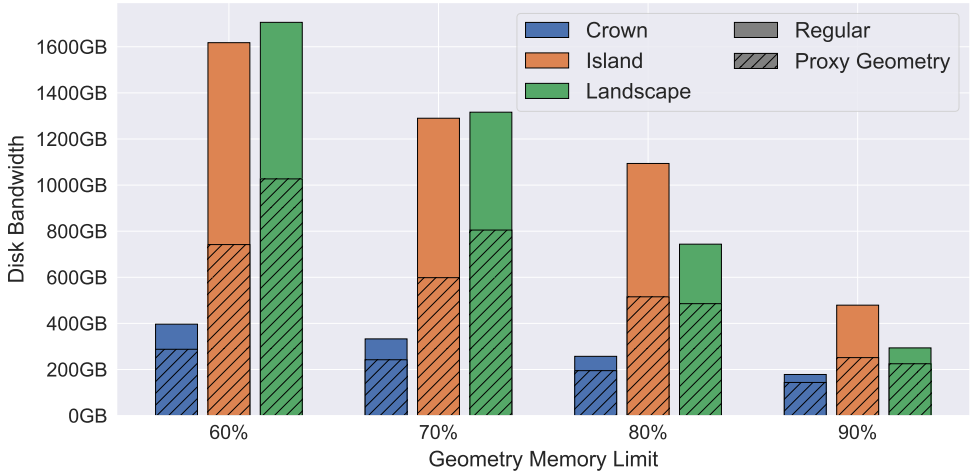


Figure 6.3.: The total disk bandwidth at different geometry memory limits with and without our technique. When proxy geometry intersection testing was enabled, the memory limit was adjusted to compensate for the memory used by the SVDAGs.

6.4.3. Discussion

Our addition of proxy geometry using SVDAGs to accelerate batched ray traversal works well for all the scenes and memory limits that were tested. Every time a ray is batched, it delays its traversal, requiring a disk read to continue. By reducing the number of times that rays are batched, we improve the flow of rays through the system. An issue with batched traversal is that rays can get stuck at infrequently visited batching points. Only when near the end of a render will these batching points be loaded and traversed. Path tracing may lead to many bounces, causing long delays when finishing a render. In our limited tests, discarding these

straggler rays did not reduce the effectiveness of our method.

Our solution works well for batched ray traversal[70], but could also be applied to other traversal schemes. For out-of-core traversal, where the discrepancy between processing power and disk bandwidth is high, the overhead of SVDAG traversal is negligible. We are unsure whether this approach could also work for in-core traversal, and we leave this up to future work.

The hit points found by intersecting against the proxy geometry could also be used to guide the bottom-level acceleration structure traversal. For example, rays could be sorted according to their expected hit points as suggested by Moon *et al.* [84].

Additionally, our method could work well for shadow rays. Here, we would voxelize geometry conservatively in two manners; one as described above, the other as an inner voxelization that conservatively predicts a valid intersection. Testing against this inner voxelization would reveal whether a shadow ray is occluded. If not, then we would test the ray against the outer voxelization. Similarly to our current system, the ray only needs to be batched if it hits the outer voxel structure. Otherwise, it can continue its traversal and select the next batching point along its path.

6.5. Conclusion

The contribution presented in this paper is to batch rays only when an intersection with a (conservative) proxy geometry occurred in order to reduce disk bandwidth. To proof our concept, we build an out-of-core unidirectional path tracer based on batched ray traversal[70] with SVDAGs as proxy geometry.

Our results show that this improves the performance of batched ray traversal. We also believe the same concept can be applied more broadly for other traversal schemes but we leave this for future work.

7

Conclusion

There has always been a desire to store, modify, and display increasingly larger and more geometrically detailed scenes. These can be represented in various ways, such as geometric surfaces, continuous surfaces, or volumes. In practice, there is no one-size-fits-all solution, and most applications combine multiple different solutions for various parts of the graphics pipeline. In this dissertation, we focus on sparse voxel volumes and their uses in the graphics pipeline, both directly (Chapters 5, 4, 3) and indirectly (Chapter 6). These volumes were represented using the Sparse Voxel Directed Acyclic Graph (SVDAG) [23] which forms the backbone of our work. This data structure is both very compact and has desirable properties such as fast access to individual voxels, and the possibility to act as an acceleration structure for ray tracing.

The goal of any compression algorithm is to detect patterns in a data set. The SVDAG applies the same methodology to voxel scenes. Assuming that there is some structure to the input data, the goal is to find repeating patterns and encoding those efficiently. A standard SVDAG only considers exact geometric matches that align with the voxel tree structure (SVO). As noted in previous work [26, 27], removing this limitation has a positive effect on the maximum achievable compression ratio. In Chapter 3 we address this problem by considering matching under various types of transformations. First, we extended previous work [26] by considering all transformations that can be described by a *recursive* permutation of SVDAG node children. We showed that almost all matches can be described with only a tiny subset of all reordering transformations. Although we achieved significant improvements over previous work, we remain limited in defining operations on the underlying tree structure. As a result, two subtrees that represent the same voxel patterns cannot be matched if those patterns are not perfectly aligned. We subsequently addressed this by providing an algorithm to efficiently match voxel patterns after a translation. Finally, we described a novel pointer com-

pression scheme to store the final Transform-Aware SVDAG. The result is a SVDAG structure that is more compact than in the previous work.

While Sparse Voxel Directed Acyclic Graphs are often used as an immutable data structure, they do not have to be static. In Chapter 4 we presented an SVDAG editing framework inspired by HashDAG[28]. By performing all major computations on the graphics card, we were able to unlock a new level of performance. Compared to previous work on SVO editing, our SVDAG structure is significantly more compact; allowing editing of higher resolution voxel scenes.

In Chapter 4 we only considered scenes where voxels have either no attributes or where the attributes consist of a small set of materials. However, this is not a limitation of the SVDAG itself. In Chapter 5 we addressed interactively editing scenes with more complex voxel attributes, such as colors. We propose both a novel GPU-accelerated color encoding scheme and a highly efficient algorithm compatible with the encoding proposed in previous work [31].

Although voxels can be displayed directly, they can also act as a building block to other computer graphics algorithms. In Chapter 6, we utilized voxels to improve the run-time performance of an out-of-core path tracer that displays surface meshes (e.g. triangles). Again, an SVDAG was used to reduce memory usage and to accelerate ray-intersection queries.

7

7.1. CHALLENGES AND FUTURE WORK

As discussed in Chapter 3, the SVDAG structure can be further compressed by matching under transformations. Although we considered all possible *recursive* permutations inside the SVDAG structure, there are many more transformations that cannot be expressed this way. By considering translation invariance, we demonstrate that removing this limitation extends the potential for SVDAG compression. We believe that this was only the tip of the iceberg and that many more interesting savings could be achieved by considering other transformations. With the advent of machine learning, it would be interesting, for example, to learn a set of transformations that most optimally apply to a specific scene or data set.

In Chapter 4 we have demonstrated a real-time editing framework for SVDAGs. However, it did not support detailed attributes. Chapter 5 only partially addressed this issue by presenting faster compression algorithms. However, only the lossless color compression scheme is GPU accelerated; thus, there is currently no suitable solution for GPU editing of very large scenes where some compression artifacts are acceptable in order to achieve much lower

memory usage. Additionally, we did not combine both methods into a single GPU editing framework. We believe that there are still significant engineering challenges to be solved to create a simple, reliable, and efficient editing framework that combines both SVDAG compression and color compression.

Finally, in this work, we tend to display voxels as cubes. While this may be suitable for some applications, where they are part of an art style, this level of discretization is not always desirable. Some work has been done on displaying voxels not just as small cubes, but as more complex shapes [2, 39]. However, these require additional information about the surface encoded by the voxels (e.g. surface normals or signed distance) and have not been applied to Sparse Voxel Directed Acyclic Graphs. We believe that there is a need for a rendering algorithm that is able to display an SVDAG as a somewhat smooth surface, without the need to encode parameters into the voxels.

7.2. Closing Words

In this dissertation we have proposed various solutions to advance the field of Sparse Voxel Directed Acyclic Graph (SVDAG) research, making them more compact, quicker to modify, and showing how they can be used indirectly in other disciplines within the field of computer graphics. Although voxels and the SVDAG are obviously not always the best tool for the job, we do believe that their usage in both research and industry is currently underutilized. We hope that the work presented in this dissertation raises awareness and inspires future researchers and artists to create, modify, and display virtual environments that are larger and more detailed than was otherwise possible.

A

Appendix: Editing Compact Voxel Representations on the GPU

Algorithm 2 Warp-centric search operation

```
1: threadMask ← 0x7FFFFFFF
2: hash ← HashFunction(needle)
3: search64 ← U64(needle)
4:
5: slab ← table[hash % numBuckets]
6: while slab ≠ end do
7:     comp64 ← U64(slab[2 * threadIdx])
8:     match ← comp64 = search64
9:     if warp.ballot(match) & threadMask ≠ 0 then
10:        if match then
11:            for i = 0 to itemSize − 2 do
12:                comp32 ← slab[64 + threadIdx × (itemSize − 2) + i]
13:                if needle[2 + i] ≠ comp32 then
14:                    match ← 0
15:                    break
16:                end if
17:            end for
18:        end if
19:
20:        activeMask ← warp.ballot(match) & threadMask
21:        if activeMask ≠ 0 then
22:            outThreadIdx ← bitscan(activeMask)
23:            return encodePointer(slab, outThreadIdx)
24:        end if
25:    end if
26:    slab = slab[62]
27: end while
```

Algorithm 3 Pseudocode to insert an item into the atomic64 hash table. Both the item to be inserted (*needle*) and the slabs are pointers to 32-bit *WORDs*. We have omitted memory fences for brievity.

```
1: threadMask ← 0x7FFFFFFF
2: hash ← HashFunction(needle)
3: insert64 ← U64(needle)
4:
5: slab ← table[hash % numBuckets]
6: loop
7:     slab = warp.shfl(slab)
8:     if slab = end then                                   ▷ Add new slab to bucket
9:         newSlab ← allocateAsWarp()
10:        newSlab[threadIdx] ← 0
11:        newSlab[32 + threadIdx] ← 0
12:        if threadIdx = 30 then
13:            newSlab[62] ← table[bucket]
14:            h ← atomicCAS(table[bucket], slab[62], newSlab)
15:            if h = slab[62] then
16:                slab ← newSlab
17:            else
18:                free(newSlab)
19:                slab ← h
20:            end if
21:        end if
22:    end if
23:                                                  ▷ Attempt inserting first 64 bits
24:    comp64 ← slab[threadIdx]
25:    activeMask ← warp.ballot(comp64 = 0) & threadMask
26:    if activeMask ≠ 0 then
27:        outThreadIdx ← bitscan(activeMask)
28:        if threadIdx = outThreadIdx then
29:            prev ← atomicCAS(slab[2 * threadIdx], 0, insert64)
30:            if prev = 0 then
31:                inserted ← 1
32:            end if
33:        end if
34:                                                  ▷ Insert remaining bytes
35:        if warp.shfl(inserted, outThreadIdx) then
36:            if threadIdx ≤ itemSize − 2 then
37:                slab[64 + outThreadIdx * (itemSize − 2) + threadIdx] ← needle[2 +
   threadIdx]
38:            end if
39:            return encodePointer(slab, outThreadIdx)
40:        end if
41:    else
42:        slab ← slab[62]
43:    end if
44: end loop
```

Acknowledgements

During my Masters at the University of Utrecht, I was convinced that I wanted to try to pursue a future in computer graphics. This ended up being easier said than done. While I already had some ideas for a Master thesis project, the graphics teacher declined due to already supervising various other students. He recommended that I contact professor Elmar Eisemann at the Delft University of Technology. Despite working at a different university, Elmar offered to supervise me together with the - then post-doc - Markus Billeter. I am eternally grateful that they offered their time and expertise to an outsider.

I recall Elmar floating the idea around of me becoming a PhD student. At the time, I did not call him up on that offer. And I cannot remember why, to be honest. I initially applied to graphics programming jobs at game companies, but realized that moving to a different country was a step too far for me. Based on advice from a fellow student, I ended up at a high-frequency trading company. Within days after starting there, I realized that I had made the wrong decision; within a week I was looking at other options.

I messaged Elmar to ask him whether he was still open to the idea of me doing a PhD. I was embarrassed to ask him after initially picking a trading company over a PhD. To my relief, Elmar was not (too) upset with me and still wanted to offer me a position. He was very close to receiving the funding for a new position. However, due to politics, this project ended did not end up happening.

While I was waiting for another PhD position to open, Elmar helped me get a teaching assistant job developing the homework assignments for a new course. After a couple of months, he suddenly mentioned that there was a new PhD position coming in. This was a special position with additional time allotted to Bachelor education. To compensate, the PhD would take an additional two years; 6 in total. After working from home as a teaching assistant for months, I immediately jumped on this opportunity. And here we are now, 6 years later...

Elmar is the best "boss" that I could have ever imagined. A full professor is an incredibly taxing job. But despite his incredibly busy schedule, Elmar always tries to take the time to talk to everyone. Our meetings frequently go

over time (if his schedule allows) so we can talk about super-duper important stuff such as soccer, movies, and video games. By the way: congratulations on FC Köln promoting to the Bundesliga!

But on a serious note: Elmar has always been extremely helpful and supportive with everything I have done. He nudged me in the right directions, motivated me when I lost hope, and always gave me the feeling that he trusted me. I spend a good chunk of my time helping out with Bachelor courses: designing & grading assignments, designing timetables, hiring teaching assistants, and giving a few supporting lectures. Elmar would regularly ask me for my advice on courses, and I really felt part of the teaching staff. I would like to express my gratitude for entrusting me with this despite me only being a PhD *student*; it really made me feel appreciated.

Elmar is the most glass-half-full person that I have ever known. He will always stay positive, trying to find the most productive way of improving a bad situation, rather than complaining about it. He will always try to help other people; a character trait that I feel I have personally benefited from. Elmar is also part of many different committees, such as the board of examiners, conference committees, management committees, and the financial oversight committee. With how little free time Elmar has, I asked him why he takes on all these additional obligations. His response was that he just wants to improve the world, in whatever small way possible. I think that is incredibly noble and selfless.

Ricardo

I would like to explicitly thank my co-promotor, Ricardo Marroquim, for taking over in Elmar's absence. With his help, the process of going through all the required paperwork, which is required for the PhD defense, has been quite smooth. In hindsight, I am also more appreciative of the way that he ran the computer graphics Bachelor course in the last couple of years. Ricardo has been the most stable force in the course that many in our group consider cursed.

Colleagues

I would also like to extend my gratitude to all my colleagues, both PhD and staff. I cannot cover all of them individually. Instead, I would like to briefly recall some of my favorite memories.

For my first *physical* conference, Eurographics 2023, I was sharing a room with Amir, who is a bit of a video game addict. He was still young, and in

his naivety, he played video games on his laptop while talking to his friend through his microphone; not something you want when you have to present the next morning. Amir and I still play League of Legends together for some time. Let's just say that I now know why Amir rarely shows up to the office before noon... But in all seriousness: Amir has become one of the most active CGV members and he deserves a shout-out for organizing the Discrete Differential Geometry course and the group outing to Kitchen Rani.

On that same trip, I offered Amir and Petr a ride home. The deal was that we would add some extra stops along the way. First, at the former Ironworks at Völklingen (thanks Lucas for the recommendation!), and then to Phantasialand the day after. We had a great time and I love how Petr was up for going onto any coaster. Petr was especially enthusiastic about doing the parkour/climbing route that was clearly designed for kids. We were crawling through an obstacle course like a bunch of small kids.

I have played video games ever since I was a little kid; but I was never much into board games. However, after playing Baldur's Gate 3, I decided to give the Dungeons and Dragons role-playing game a chance. I have joined multiple evening sessions at Mark's house; who is the game master. I don't think he realizes how skilled and artistic he is, quickly improvising stories and acting out voice lines.

While I haven't known her for that long, I am proud to be able to call Priyanka my *friend*. I like discussing general life-stuff with her. Our cultural differences give us different insights into various situations. Priyanka always talks about her role models and the people she looks up to. I am sure that she will become what she wants to be, and be a role model for many other people.

Finally, I would like to thank all of my other colleagues that I have not previously mentioned. In no particular order: Soumya, Ali, Baran, Benno, Peter, Nicolas, Lukas, Jackson, Chen-Chi, Annemieke, Yang, Alex, Mika, Mrinal, Fengshi, and Celine. I hold fond memories of all of you, and I hope that we will meet again someday.

Family

Last, but certainly not least, I want to share my appreciation to my parents. Unlike most PhD students, I was able to keep living with my parents. I got really lucky because I am not sure how I would have managed the COVID lockdown on my own in a small apartment. It has also provided me with a lot of stability, knowing that there is always someone at home supporting me. Doing a PhD is no easy feat. Having only published a short paper (based on

my Master's thesis) three years into my PhD definitely did not help my self-esteem. But my parents and my brother were always there for me, talking me up when I felt down. You always supported me when I needed it, and distracted me when I needed stress relief. I will always remember the fun times we've had, such as the movies and (motor)sports we watched together, and our yearly vacations. While this is the end of my PhD journey, it will not be the end of us making new memories together.

Curriculum Vitæ

Mathijs Lucas Molenaar

27-07-1995	Born in Nieuw Sloten, The Netherlands.

Education

2007–2013	Keizer Karel College *VWO Atheneum*
2013-2016	Universiteit van Amsterdam *BSc, Computer Science*
2016-2019	Universiteit Utrecht *MSc, Game & Media Technology*

Awards

2025	Best Paper Award at I3D 2025

98

List of Publications

4. M. Molenaar and E. Eisemann. “Transform-Aware Sparse Voxel Directed Acyclic Graphs”. In: *Proc. ACM Comput. Graph. Interact. Tech.* 8.1 (May 2025). DOI: 10.1145/3728301. URL: https://doi.org/10.1145/3728301

3. M. Molenaar and E. Eisemann. “Editing Compact Voxel Representations on the GPU”. in: *Pacific Graphics Conference Papers and Posters*. Ed. by R. Chen, T. Ritschel, and E. Whiting. The Eurographics Association, 2024. ISBN: 978-3-03868-250-9. DOI: 10.2312/pg.20241310

2. M. Molenaar and E. Eisemann. “Editing Compressed High-resolution Voxel Scenes with Attributes”. In: *Computer Graphics Forum* 42.2 (2023), pp. 235–243. DOI: https://doi.org/10.1111/cgf.14757. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14757. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14757

1. M. Molenaar and E. Eisemann. “Conservative Ray Batching using Geometry Proxies”. In: *Eurographics 2020 - Short Papers*. Ed. by A. Wilkie and F. Banterle. The Eurographics Association, 2020. ISBN: 978-3-03868-101-4. DOI: 10.2312/egs.20201006

Bibliography

[1] S. Lefebvre, S. Hornus, F. Neyret, *et al.* "Octree textures on the GPU". In: *GPU gems* 2 (2005), pp. 595–613.

[2] K. Museth. "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM transactions on graphics (TOG)* 32.3 (2013), pp. 1–22.

[3] M. Werner, M. Piochowiak, and C. Dachsbacher. "SVDAG Compression for Segmentation Volume Path Tracing". In: *Vision, Modeling, and Visualization*. Ed. by L. Linsen and J. Thies. The Eurographics Association, 2024. ISBN: 978-3-03868-247-9. DOI: 10.2312/vmv.20241196.

[4] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. "Interactive indirect illumination using voxel cone tracing: a preview". In: *Symposium on Interactive 3D Graphics and Games*. I3D '11. San Francisco, California: Association for Computing Machinery, 2011, p. 207. ISBN: 9781450305655. DOI: 10.1145/1944745.1944787. URL: https://doi.org/10.1145/1944745.1944787.

[5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. "GigaVoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering". In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. Boston, Massachusetts: ACM, 2009, pp. 15–22. ISBN: 978-1-60558-429-4. DOI: 10.1145/1507149.1507152.

[6] V. Kampe, E. Sintorn, D. Dolonius, and U. Assarsson. "Fast, Memory-Efficient Construction of Voxelized Shadows". In: *IEEE Transactions on Visualization and Computer Graphics* 22.10 (Oct. 2016), pp. 2239–2248. ISSN: 10772626. DOI: 10.1109/TVCG.2016.2539955.

[7] T. Müller, M. Gross, and J. Novák. "Practical path guiding for efficient light-transport simulation". In: *Computer Graphics Forum*. Vol. 36. 4. Wiley Online Library. 2017, pp. 91–100.

[8] J. Chen, D. Bautembach, and S. Izadi. "Scalable real-time volumetric surface reconstruction." In: *ACM Trans. Graph.* 32.4 (2013), pp. 113–1.

[9] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. "Real-time 3D reconstruction at scale using voxel hashing". In: *ACM Transactions on Graphics (ToG)* 32.6 (2013), pp. 1–11.

[10] D. Benson and J. Davis. “Octree textures”. In: *ACM Transactions on Graphics (TOG)* 21.3 (2002), pp. 785–790.

[11] D. DeBry, J. Gibbs, D. D. Petty, and N. Robins. “Painting and rendering textures on unparameterized models”. In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, pp. 763–768.

[12] Y. Kim, B. Kim, and Y. J. Kim. “Dynamic Deep Octree for High-resolution Volumetric Painting in Virtual Reality”. In: *Computer Graphics Forum* 37.7 (2018), pp. 179–190. DOI: https://doi.org/10.1111/cgf.13558.

[13] V. Kužel. “Real-time voxel visualization and editing for 3D printing”. MA thesis. Univerzita Karlova, Matematicko-fyzikální fakulta, 2021.

[14] V. Gaede and O. Günther. “Multidimensional access methods”. In: *ACM Computing Surveys (CSUR)* 30.2 (1998), pp. 170–231.

[15] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. “Real-time parallel hashing on the GPU”. In: *ACM SIGGRAPH Asia 2009 Papers*. SIGGRAPH Asia ’09. Yokohama, Japan: Association for Computing Machinery, 2009. ISBN: 9781605588582. DOI: 10.1145/1661412.1618500.

[16] P. Gautron. “Advances in Spatial Hashing: A Pragmatic Approach towards Robust, Real-time Light Transport Simulation”. In: *ACM SIGGRAPH 2022 Talks*. SIGGRAPH ’22. Vancouver, BC, Canada: Association for Computing Machinery, 2022. ISBN: 9781450393713. DOI: 10.1145/3532836.3536239. URL: https://doi.org/10.1145/3532836.3536239.

[17] S. Lefebvre and H. Hoppe. “Perfect spatial hashing”. In: *ACM Transactions on Graphics (TOG)* 25.3 (2006), pp. 579–588.

[18] K. Museth. “NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation”. In: *ACM SIGGRAPH 2021 Talks*. 2021, pp. 1–2.

[19] R. K. Hoetzlein. “GVDB: Raytracing Sparse Voxel Database Structures on the GPU”. In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by U. Assarsson and W. Hunt. The Eurographics Association, 2016. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161197.

[20] D. Kim, M. Lee, and K. Museth. “NeuralVDB: High-resolution sparse volume representation using hierarchical neural networks”. In: *ACM Transactions on Graphics* 43.2 (2024), pp. 1–21.

[21] R. E. Webber and M. B. Dillencourt. "Compressing quadtrees via common subtree merging". In: *Pattern Recognition Letters* 9.3 (1989), pp. 193–200. ISSN: 0167-8655. DOI: https://doi.org/10.1016/0167-8655(89)90054-8.

[22] E. Parker and T. Udeshi. "Exploiting Self-similarity in Geometry for Voxel Based Solid Modeling". In: *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*. SM '03. Seattle, Washington, USA: ACM, 2003, pp. 157–166. ISBN: 1-58113-706-0. DOI: 10.1145/781606.781631.

[23] V. Kämpe, E. Sintorn, and U. Assarsson. "High Resolution Sparse Voxel DAGs". In: *ACM Trans. Graph.* 32.4 (July 2013), 101:1–101:13. ISSN: 0730-0301. DOI: 10.1145/2461912.2462024.

[24] L. Scandolo, P. Bauszat, and E. Eisemann. "Compressed Multiresolution Hierarchies for High-Quality Precomputed Shadows". In: *Computer Graphics Forum*. Vol. 35. 2. Wiley Online Library. 2016, pp. 331–340.

[25] R. van der Laan, L. Scandolo, and E. Eisemann. "Lossy geometry compression for high resolution voxel scenes". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.1 (2020), pp. 1–13.

[26] A. J. Villanueva, F. Marton, and E. Gobbetti. "SSVDAGs: Symmetry-aware sparse voxel DAGs". In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2016, pp. 7–14.

[27] P. Čerešník, B. Madoš, A. Baláž, and Z. Bilanová. "SSVDAG*: Efficient Volume Data Representation Using Enhanced Symmetry-Aware Sparse Voxel Directed Acyclic Graph". In: *2019 IEEE 15th International Scientific Conference on Informatics*. IEEE. 2019, pp. 000201–000206.

[28] V. Careil, M. Billeter, and E. Eisemann. "Interactively modifying compressed sparse voxel representations". In: *Computer Graphics Forum*. Vol. 39. 2. Wiley Online Library. 2020, pp. 111–119.

[29] B. Williams. *Moxel DAGs: Connecting material information to high resolution sparse voxel DAGs*. California Polytechnic State University, 2015.

[30] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann. "Geometry and Attribute Compression for Voxel Scenes". In: *Computer Graphics Forum* 35.2 (2016), pp. 397–407. DOI: https://doi.org/10.1111/cgf.12841.

[31] D. Dolonius, E. Sintorn, V. Kämpe, and U. Assarsson. "Compressing color data for voxelized surface geometry". In: *IEEE transactions on visualization and computer graphics* 25.2 (2017), pp. 1270–1282.

[32] M. Molenaar and E. Eisemann. “Transform-Aware Sparse Voxel Directed Acyclic Graphs”. In: *Proc. ACM Comput. Graph. Interact. Tech.* 8.1 (May 2025). DOI: 10.1145/3728301. URL: https://doi.org/10.1145/3728301.

[33] M. O. Rabin. “Fingerprinting by random polynomials”. In: *Technical report* (1981).

[34] R. M. Karp and M. O. Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM journal of research and development* 31.2 (1987), pp. 249–260.

[35] Boost. *Boost C++ Libraries*. http://www.boost.org/. Last accessed 2024-01-10. 2024.

[36] V. Kämpe, S. Rasmuson, M. Billeter, E. Sintorn, and U. Assarsson. “Exploiting coherence in time-varying voxel data”. In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM. 2016, pp. 15–21.

[37] B. Madoš and N. Ádám. “Transforming hierarchical data structures–a psvdag–svdag conversion algorithm”. In: *Acta Polytechnica Hungarica* 18.8 (2021), pp. 47–66.

[38] A. Moffat. “Huffman Coding”. In: *ACM Comput. Surv.* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: 10.1145/3342555. URL: https://doi.org/10.1145/3342555.

[39] S. Laine and T. Karras. “Efficient sparse voxel octrees”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2011), pp. 1048–1059.

[40] S. Laine, T. Karras, and T. Aila. “Megakernels considered harmful: Wavefront path tracing on GPUs”. In: *Proceedings of the 5th High-Performance Graphics Conference*. 2013, pp. 137–143.

[41] H. H. Söderlund, A. Evans, and T. Akenine-Möller. “Ray Tracing of Signed Distance Function Grids”. In: *Journal of Computer Graphics Techniques Vol* 11.3 (2022).

[42] M. Molenaar and E. Eisemann. “Editing Compact Voxel Representations on the GPU”. In: *Pacific Graphics Conference Papers and Posters*. Ed. by R. Chen, T. Ritschel, and E. Whiting. The Eurographics Association, 2024. ISBN: 978-3-03868-250-9. DOI: 10.2312/pg.20241310.

[43] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. “Building an efficient hash table on the GPU”. In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 39–53.

[44] I. García, S. Lefebvre, S. Hornus, and A. Lasram. “Coherent parallel hashing”. In: *ACM Transactions on Graphics (TOG)* 30.6 (2011), pp. 1–8.

[45] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan. "Stadium hashing: Scalable and flexible hashing on gpus". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 63–74.

[46] D. Jünger, C. Hundt, and B. Schmidt. "WarpDrive: Massively parallel hashing on multi-GPU nodes". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 441–450.

[47] R. Pagh and F. F. Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004), pp. 122–144.

[48] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores". In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1226–1237.

[49] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun. "Dycuckoo: dynamic hash tables on gpus". In: *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE. 2021, pp. 744–755.

[50] S. Ashkiani, M. Farach-Colton, and J. D. Owens. "A dynamic hash table for the GPU". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 419–429.

[51] AMD GPUOpen. *Vulkan Memory Allocator*. https://gpuopen.com/vulkan-memory-allocator/. 2023.

[52] M. Molenaar and E. Eisemann. "Editing Compressed High-resolution Voxel Scenes with Attributes". In: *Computer Graphics Forum* 42.2 (2023), pp. 235–243. DOI: https://doi.org/10.1111/cgf.14757. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14757. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14757.

[53] K. Museth, N. Avramoussis, and D. Bailey. "OpenVDB". In: *ACM SIGGRAPH 2019 Courses*. 2019, pp. 1–56.

[54] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[55] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

[56] A. Weißenberger and B. Schmidt. "Massively parallel Huffman decoding on GPUs". In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[57] J. Tian, C. Rivera, S. Di, J. Chen, X. Liang, D. Tao, and F. Cappello. "Revisiting huffman coding: Toward extreme performance on modern gpu architectures". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2021, pp. 881–891.

[58] A. Ozsoy and M. Swany. "CULZSS: LZSS lossless data compression on CUDA". In: *2011 IEEE International Conference on Cluster Computing*. IEEE. 2011, pp. 403–411.

[59] A. Ozsoy. "Culzss-bit: A bit-vector algorithm for lossless data compression on gpgpus". In: *2014 International Workshop on Data Intensive Scalable Computing Systems*. IEEE. 2014, pp. 57–64.

[60] Y. Zu and B. Hua. "GLZSS: LZSS lossless data compression can be faster". In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. 2014, pp. 46–53.

[61] L. Lu and B. Hua. "G-Match: a fast GPU-friendly data compression algorithm". In: *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, pp. 788–795.

[62] C. M. Stein, D. Griebler, M. Danelutto, and L. G. Fernandes. "Stream parallelism on the LZSS data compression application for multi-cores with GPUs". In: *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2019, pp. 247–251.

[63] D. A. Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[64] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner. "From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms". In: *ACM Transactions on Database Systems (TODS)* 44.3 (2019), pp. 1–46.

[65] G. K. Wallace. "The JPEG still picture compression standard". In: *Communications of the ACM* 34.4 (1991), pp. 30–44.

[66] P. V. Hough. *Method and means for recognizing complex patterns*. US Patent 3,069,654. 1962.

[67] R. O. Duda and P. E. Hart. "Use of the Hough transformation to detect lines and curves in pictures". In: *Communications of the ACM* 15.1 (1972), pp. 11–15.

[68] X. Décoret, F. Durand, F. X. Sillion, and J. Dorsey. "Billboard clouds for extreme model simplification". In: *ACM SIGGRAPH 2003 Papers*. 2003, pp. 689–696.

[69] M. Molenaar and E. Eisemann. “Conservative Ray Batching using Geometry Proxies”. In: *Eurographics 2020 - Short Papers*. Ed. by A. Wilkie and F. Banterle. The Eurographics Association, 2020. ISBN: 978-3-03868-101-4. DOI: 10.2312/egs.20201006.

[70] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. “Rendering Complex Scenes with Memory-coherent Ray Tracing”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 101–108. ISBN: 0-89791-896-7. DOI: 10.1145/258734.258791.

[71] I. Wald, A. Dietrich, and P. Slusallek. “An Interactive Out-of-core Rendering Framework for Visualizing Massively Complex Models”. In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH ’05. Los Angeles, California: ACM, 2005. DOI: 10.1145/1198555.1198756.

[72] E. Gobbetti, F. Marton, and J. A. Iglesias Guitián. “A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets”. In: *The Visual Computer* 24.7 (July 2008), pp. 797–806. ISSN: 1432-2315. DOI: 10.1007/s00371-008-0261-9.

[73] J. Pantaleoni, L. Fascione, M. Hill, and T. Aila. “PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes”. In: *ACM Trans. Graph.* 29.4 (July 2010), 37:1–37:10. ISSN: 0730-0301. DOI: 10.1145/1778765.1778774.

[74] S.-E. Yoon, C. Lauterbach, and D. Manocha. “R-LODs: fast LOD-based ray tracing of massive models”. In: *The Visual Computer* 22.9-11 (2006), pp. 772–784.

[75] M. Cox and D. Ellsworth. “Application-controlled demand paging for out-of-core visualization”. In: *Visualization’97., Proceedings*. IEEE. 1997, pp. 235–244.

[76] P. H. Christensen, D. M. Laur, J. Fong, W. L. Wooten, and D. Batali. “Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes”. In: *Computer Graphics Forum* 22.3 (2003), pp. 543–552. DOI: 10.1111/1467-8659.t01-1-00702.

[77] I. Wald, P. Slusallek, and C. Benthin. “Interactive Distributed Ray Tracing of Highly Complex Models”. In: *Rendering Techniques 2001*. Ed. by S. J. Gortler and K. Myszkowski. Vienna: Springer Vienna, 2001, pp. 277–288. ISBN: 978-3-7091-6242-2.

[78] I. Wald, C. P. Gribble, S. Boulos, and A. Kensler. "SIMD Ray Stream Tracing-SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering". In: *Informe Técnico, SCI Institute* (2007).

[79] C. P. Gribble and K. Ramani. "Coherent ray tracing via stream filtering". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 59–66.

[80] J. A. Tsakok. "Faster Incoherent Rays: Multi-BVH Ray Stream Tracing". In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: ACM, 2009, pp. 151–158. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572793.

[81] K. Ramani, C. P. Gribble, and A. Davis. "Streamray: a stream filtering architecture for coherent ray tracing". In: *ACM Sigplan Notices*. Vol. 44. 3. ACM. 2009, pp. 325–336.

[82] R. Barringer and T. Akenine-Möller. "Dynamic Ray Stream Traversal". In: *ACM Trans. Graph.* 33.4 (July 2014), 151:1–151:9. ISSN: 0730-0301. DOI: 10.1145/2601097.2601222.

[83] V. Fuetterling, C. Lojewski, F.-J. Pfreundt, and A. Ebert. "Efficient Ray Tracing Kernels for Modern CPU Architectures". In: *Journal of Computer Graphics Techniques (JCGT)* 4.4 (2015).

[84] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon. "Cache-oblivious Ray Reordering". In: *ACM Trans. Graph.* 29.3 (July 2010), 28:1–28:10. ISSN: 0730-0301. DOI: 10.1145/1805964.1805972.

[85] C. Eisenacher, G. Nichols, A. Selle, and B. Burley. "Sorted deferred shading for production path tracing". In: *Computer Graphics Forum*. Vol. 32. 4. Wiley Online Library. 2013, pp. 125–132.

[86] P. A. Navratil, D. S. Fussell, C. Lin, and W. R. Mark. "Dynamic ray scheduling to improve ray coherence and bandwidth utilization". In: *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*. IEEE. 2007, pp. 95–104.

[87] J. Bikker. "Improving Data Locality for Efficient In-Core Path Tracing". In: *Computer Graphics Forum*. Vol. 31. Wiley Online Library. 2012, pp. 1936–1947.

[88] T. Gasparian. "Fast Divergent Ray Traversal by Batching Rays in a BVH". MA thesis. Utrecht University, 2016.

[89] E. Sintorn, V. Kämpe, O. Olsson, and U. Assarsson. "Compact Precomputed Voxelized Shadows". In: *ACM Trans. Graph.* 33.4 (July 2014), 150:1–150:8. ISSN: 0730-0301. DOI: 10.1145/2601097.2601221.

[90] M. Schwarz and H.-P. Seidel. “Fast Parallel Surface and Solid Voxelization on GPUs”. In: *ACM Trans. Graph.* 29.6 (Dec. 2010), 179:1–179:10. ISSN: 0730-0301. DOI: 10.1145/1882261.1866201.