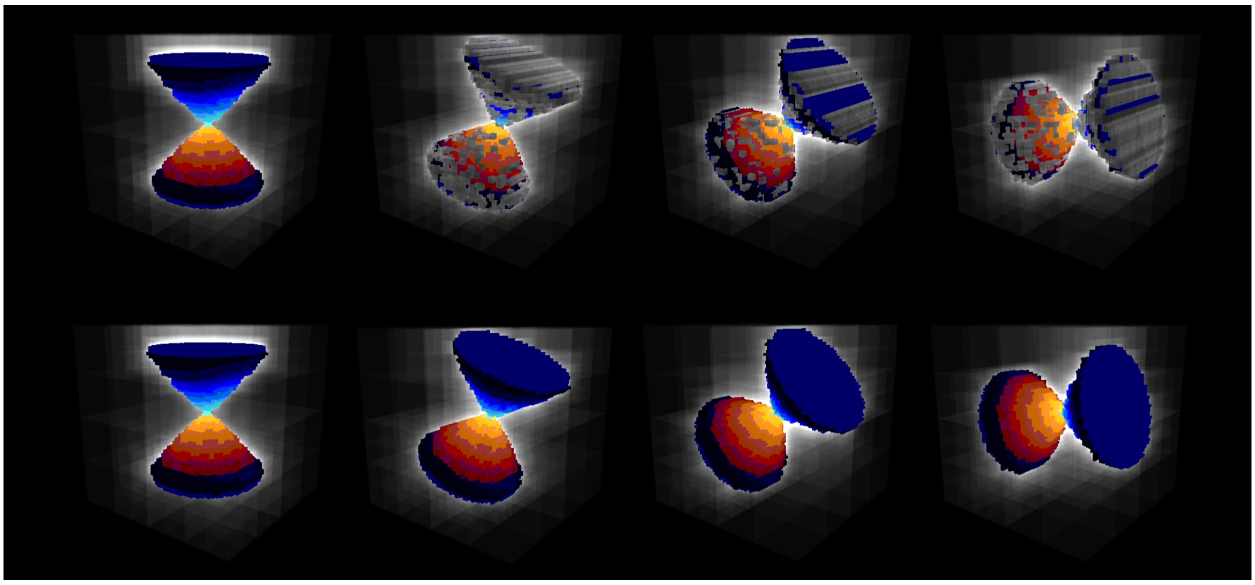


Compression of Sequential Voxel Data for Sequence-based Animation

Master's thesis in Computer science and engineering

FILIP ANTONIJEVIC

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Compression of Sequential Voxel Data for Sequence-based Animation

FILIP ANTONIJEVIC

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

FILIP ANTONIJEVIC

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visual example of spatiotemporal coherency in a voxel animation sequence. This demonstrates how data from a previous time step is retrieved from a previous time step to be re-used in each consecutive time step. The transparent voxels represent the spatiotemporally coherent voxels in the animation sequence.

Typeset in LaTeX
Gothenburg, Sweden 2023

FILIP ANTONIJEVIC
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents a method for lossy compression of voxel data in an animation sequence. Uncompressed voxel animation can become very memory intensive and therefore need a more efficient method to store and render such data. To achieve this, a combination of methods were implemented in order to create a compact data structure, given the name *Hyper Octree*. This method first involves the use of a modified sparse voxel octree that is able to exploit spatiotemporal coherency between each consecutive frame in a sequence. The second method is a compression of colours using the DXT1 format in order to further compress the data. This format causes a loss in quality, so a novel approach of using HSV values to sort or reorder the colours in order to be in close proximity to each other in memory before compression. This format also causes redundant data to be generated, so the data is reduced as well to further compress the data. The result shows that the Hyper Octree was able to greatly reduce the memory consumption and the HSV sorting was able to preserve the quality of the colours moderately. Sorting colours with HSV appeared to be most effective depending on which component in the HSV colourspace was the most frequent in the voxel data's set of colours.

Keywords: voxel, octree, grid, compression, animation, spatiotemporal, sorting, re-ordering, DXT, HSV

Acknowledgements

I would like to give my gratitude to my thesis supervisor Erik Sintorn who used much of his precious time to discuss, assist and explain the many fundamental aspects of this projects. This project would not have been possible without him and I thank him for everything.

I would also like show my gratitude to thesis examiner Ulf Assarsson, who also gave important feedback that not only put the project in better focus, but even helped increase its overall quality in the end. His input been invaluable and I thank him as well.

Filip Antonijevic, Gothenburg, 2023-06-20

Contents

1

Introduction

Voxels are commonly used in video games for real-time voxel visualisation for rendering objects and scenes, and can be considered as an alternative form of graphical rendering in comparison to the triangle-based pipeline used in all modern GPUs [7]. However, voxel visualisation is not natively supported on any general GPU nor officially supported through hardware-accelerated APIs, compared to polygon-based solutions. Despite this, in recent years there have been more and more ray-based solutions applying raytracing, cone tracing and ray-marching for rendering, alongside octrees used for efficient storage and faster rasterization [24, 2, 7, 17, 16, 18]. Despite this, many modern commercial games that utilize voxel visualisation still prefer to use a polygon-based or hybrid approach, mixing rendering and storage technologies in order to take advantage of the current GPU pipeline and hardware acceleration [24]. This approach is used for both static and animated scenes and objects. For animated scenes and objects, games such as *Trove* [37], *Stonehearth* [38] and *Cube World* [39] use a pure polygon-based approach for utilizing a voxel-like form of voxel visualisation. The animation is based on a conventional skeletal animation or rigging method to deform the voxel models that have been converted into mesh data beforehand. This might be done in order to achieve the visual characteristics of voxels as a type of graphical representation rather than taking advantages of the inherent qualities of voxels, which is detailed structures that are built up using atomic units [24].

However, using a polygon-based approach cannot be considered as voxel visualisation, since it no longer uses voxels in any capacity for neither storage nor graphical rendering. Using this approach only leads to a conventional, GPU rasterized polygon model that only in appearance looks cubic, mimicking the use of voxel data. Other than an artistic choice of the developers, the reason why this method might be more common for animated voxel data in video games might be because of the memory consumption of such data. Mileff et al. [24] suggest that voxel models have relatively large set of data and these have been computationally expensive to store and render because of the lack of support in both hardware and software, up until very recent innovations. Since hardware only supports a triangle-based pipeline it would not be very efficient to use a true voxel format. Such format would consume a lot memory, especially in huge quantities [24, 2]. This would also hold true if such format was used for animation. This does suggest that there is a need for a more memory efficient method for using voxel data in an animation sequence.

It is important to clarify that animation with voxel data does not refer to deforming mesh data as previously described. Rather, it refers to a volumetric data structure with an extra dimension where each element represents a set of sequential voxel data, i.e. a 4D container where voxels are a discretization of 3D volumes over time [8]. Games that use voxel visualisation for this purpose are e.g. *Voxatron* [35] and *3D-Dot Game Heroes* [36]. These use voxel animation for both environments and characters but are limited to low voxel resolutions. If the animations have many time steps or use a high voxel resolution, or possibly a a combination of both, it will inevitably lead to high memory consumption and impact the performance. Without any type of compression performed on animated voxel data the memory usage will become too large to be realistically used in a video game. Therefore, memory efficiency is important for animated voxel data as it improves both storage and reduces memory for rendering [2]. If a method could be conceived to efficiently store animated voxel data, it could encourage more application of true voxel visualisation in video games, as well as more utilization of sequence-based voxel animation over polygon-based approaches.

1.1 Aim

The goal of this project is to explore how sequence-based animation with voxel data can be made memory efficient through compression and decompression using sparseness, colour compression and redundancy. Redundancy means to exploit spatiotemporal coherence [3], i.e. to re-use redundant data that exists in the same place at a different time step. In this case, the goal is to find a lossy solution to achieve efficient memory usage for voxel animation. This entails not only compression of geometry through temporal coherency but compression of colour attributes as well. To achieve this goal it will be necessary to find a method appropriate for compression of voxel sequences. The method will be limited to its application for voxel animation in video games.

This project will not explore nor consider an implementation for lossless or near-lossless compression methods. This is because lossless methods are used to ensure full recovery of the data when it is decompressed. Usually these sorts of methods are important if the data must be recovered with guaranteed zero errors, which is common and necessary in medical imaging applications [11]. In the context of video games, it is only important that the data in its compressed format appears similar visually to the original data in its uncompressed format. This can be accomplished using a lossy compression method, achieving a greater compression ratio compared to a lossless or near-lossless method [11].

Furthermore, this project will not attempt any other graphical implementations other than what is relevant for the completion of this project. What is relevant is a way to render voxels (which will not include any advanced shadows or lighting techniques), efficient storage for voxels and colours, a mechanism to play an animation sequence and a way to find spatiotemporal coherency in the data for efficient memory storage and usage.

1.1.1 Motivation

There are plethora of voxel formats. However currently, there seems to be only one format that supports animation using voxel data. The VOX format is a voxel data format with animation capabilities and is primarily used in the voxel modelling software MagicaVoxel [32]. The VOX format can easily be integrated into game development systems due to its simple, RIFF-style format [41]. The base format does not support animation but an extension version of it has support for sequence-based animation. Although this format allows animation, it is an uncompressed format and in addition contains data specifically for use in the software rather than being optimized for more general purpose use [41, 42]. Furthermore, there is a resolution limitation of the 3D volume of maximum 256^3 voxels in total. This limitation is because of a hard limitation in the modelling software rather than being a limitation of the format itself. Other voxel modelling software such as Qubicle [33] does currently not support animation in the manner of MagicaVoxel, allowing instead animation through bone animation techniques. Bone rigged animation, although a very common type of animation technique, cannot be considered as voxel animation since voxel animation can only strictly be done using a form of visualisation, where each stage in the animation is a separate volumetric data. Another voxel modelling software is SLAB6 which uses KV6 and KVX formats. Although these formats are supposedly highly optimized, these are also limited to 256x256x255 [34] and can only store 256 colours for the whole voxel model. These formats were originally designed for Build, an old computer game engine, where the limit size was 128x128x200 [34]. These formats do not seem to support any animation capabilities. Because of this, there does not seem to be a standard, efficient format for compressing a voxel data structure with a time factor for the purpose of animation. If such a format could be conceived, it would encourage more application of voxel visualisation in video games, both for static and animated environments and objects.

1.2 Problem Statement

To give a definite form to the aim of this project, the following research questions have been formulated:

- What is the best method for lossy compression of animated voxel data for sequence-based animation that keeps similar fidelity and quality?

This problem requires to consider two aspects that need to be solved: compression of voxel geometry and voxel colour. The state-of-the-art shows that using a form of SVO (sparse voxel octree) or SVDAGs (sparse voxel directed acyclic graph) have shown to be efficient for storage and rendering of voxel data [2, 7, 4, 17, 18]. However, these techniques are mostly optimized for inherently reconstructing static surfaces, i.e. surfaces that are not reconstructed often. None of these techniques have inherently been designed for sequence-based voxel animation, although Kämpe et al. [3]'s approach using the Temporal DAGs is a recent state-of-the-art method within the area of compressing time-varying voxels. This is similar in nature to the aim this project. Ma and Shen [12] approach of exploiting temporal coherency between frames by merging similar subtrees of volumetric data is of particular interest as a very relevant approach to how animated voxel data may be compressed by exploiting spatiotemporal coherency.

For colour compression, the state-of-the-art suggests that the best approach for the goal of this project would be to apply a colour quantization technique to compress colour attributes. Laine and Karras apply a simplified DXT1 to compress the colour data in the SVO [7], while Dolonius et al. [5] and Dado et al. [6] explore to decouple the colours from the SVDAG to only needing to count from the root down to every branch of the tree in order to get the right index for the colour data. However, they differ in their approaches by how the colour compression is performed. Dado et al. [6] uses a palette compression scheme by quantizing all colours, then implicitly retrieving them through material indices stored in blocks, while Dolonius et al. [5] uses a space-filling curve to map colours and then perform various existing image and texture compression algorithms on those colours.

1.2.1 Proposal

For compression of voxel geometry the author proposes an approach of implementing an voxel octree data structure that is able to exploit spatiotemporal coherency. More specifically, the proposal is to implement an SVO with capabilities to become a sequence-based animation, while also implementing a mechanism to exploit temporal coherency in a similar way how SVDAGs are constructed to exploit spatial coherency [4] and how Temporal DAG exploits temporal coherency [3] respectively; and as well as implementing a method similar to how Ma and Shen's technique [12] of merging subtrees that are in consecutive time steps if they are similar. Of course, the difference is that this would be done as an offline approach and use a lossy encoding unlike Kämpe et al that only attempted a lossless solution [3]. The reasoning for an offline approach to encode the data is because this project is not concerned to achieve real-time encoding of arbitrary data for a limited bit rate. Instead the purpose is to encode the voxel data to be memory efficient for storage, create a format that will later be used in an application in real-time, i.e. specifically in video games for animation. The assumption is that the data will not be entirely arbitrary but instead specified by the developer.

For colour compression, the state-of-the-art attempts to compress both voxel and attributes separately by decoupling the voxel data. The author proposes a similar approach to both Dado et al. [6] and Dolonius et al. [5] of decoupling the colour data from from the voxel geometry and compress the data separately. On the colour data the author proposes to perform a DXT1 compression, while the voxel geometry will be compressed as described previously. The motivation for a lossy compression is to reduce the memory consumption of colours. Although DXT1 has a compression ratio 6:1 or 4:1 [46], it is likely to lead to some quality degradation due to how colours are decompressed if the colours in a block are not close in the colour space, which will lead to undesirable results. Because of this, a novel attempt will be made to mitigate this quality degradation using a colour packing scheme by sorting the colours using the inherent properties of the uncompressed colours. Specifically, the attempt will be made to sort or order colours using HSV values for a similar purpose like Dolonius et al. [5] that used a space-filling curve to map the colour data to a 2D image for their compression procedure. The motivation for using HSV values has to do with their inherent properties that will allow colours to be sorted in close proximity to each other in that colour space [29]. The attempt will be done in order to keep a good compression ratio while also preserving the quality of the colours when decompressed as much as possible. This does however, have the implication that the compression ratio might not be as high as it possibly could be. Nonetheless, it is therefore even more important that the spatiotemporal compression has a high enough compression ratio to complement for any loss of possible compression gains when attempting to mitigate the quality degradation.

Since the use of sparse voxel octrees, exploiting spatiotemporal coherency and decoupling of attributes was shown to be effective for static and time-varying voxel data, the hypothesis for this project is that these will also be effective methods for animated voxel data. How effective these methods are for the purpose of this project will be evaluated in order to give a conclusive answer to the research question.

2 Previous Work

In general, voxel data compression and rendering is a well-researched area. However, the main area of research is mostly compression and optimization of static voxel data structures by exploiting sparseness. When these are used the surfaces are not reconstructed often. Previous research that focuses on sequential voxel data for animation appears to be scarce, although time-varying voxel data [3] and interactive modifying sparse voxels [10] has been explored before by Kämpe et al. and Careil et al. respectively. There has been some research done on efficient animation with SVOs, e.g. Espe et al. [9] present a novel method for animation of voxel models using the SVO format. However in this case animation refers to model transformation such as rotation, translation and anisotropic scaling and the challenge comes from preserving the hierarchical structure for efficient traversal. Thus, their research is out-of-scope for this project.

2.1 Time-varying Voxel Compression

Kämpe et al. [3] encode voxel data with time-varying factor for the purpose of streaming 3D video and efficient memory storage, named the Temporal DAG. Each time step has identical grid resolution. This variation exploits coherence using a sequence of SVOs by encoding them into a single DAG where each root represents a time step. Coherency is searched for in each SVO which has its nodes arranged in a list per level in order to find all subtrees that are identical. Identical nodes are found by sorting the list of nodes and once sorted all identical nodes are adjacent in the list. The nodes on parental level are updated to point to the corresponding nodes. When the top level is reached, all coherence has been encoded within each frame and between frames. When all coherence is found, the DAG's topology is fixed and all allocated pointers for non-existing children are removed in order to make it compact. Further compression is applied by minimizing the bits per pointer and encode the final result into a bitstream. Results show that temporal DAGs require a large number of pointers but that many of these can be stored implicitly. Also, memory performance is superior to other voxel representations that encode less coherence. However for longer sequences of time steps, a single DAG requires too much working memory since it accumulates many nodes which increases the memory consumption for each time step. This method only considered lossless encoding in order to guarantee limited bit rate for arbitrary data. A lossy compression would lessen the effectiveness of the compression for real-time encoding of arbitrary data.

Ma and Shen [12] describe a method to use coherency between each time steps by the use of octree encoding for spatial and difference encoding for temporal compression by fusing together octrees that have similar values and also merging subtrees if they are identical. Their method involves to first quantize the time-varying data at the voxel level, then the data is encoded and organized hierarchically into an octree. The octree is used to control rendering, image quality and compression. This approach is able to fuse voxels with identical values and continues to do so until there are no more voxels to merge. Finally, they implement a difference encoding in order to predict each sample based on the past. This encoding allows for each octree to be partially merged with a previous octree in the timeline by finding identical subtrees and only store a pointer to that subtree. The result of this approach shows great improvement using the compressed data for both storage and rendering while visualization results stay visual indistinguishable compared to uncompressed.

Yu et al. [30] proposed a GPU-based method to efficiently visualize large set of 3D data that evolves over time (4D data) from virtual heart simulations, specifically to study the evolution of cardiac excitation waves in normal and pathological conditions. Their proposed approach is to compress the data in two parts: the first part is to use a standard hierarchical vector quantization method (SHVQ) with N-nearest neighbour search on the GPU. They used an optimized variety of SHVQ, utilizing indexing and codebooks. The second part is to decompress the compressed data on the GPU and using a ray casting method for rendering. The result shows, using large test data of time-varying cardiac electrophysiological simulation data, that the approach can achieve rendering of the data above 35 frames per second, achieving real-time rendering speeds with high quality that are accurate and decreases the time in the compression part.

2.2 Voxel Colour Compression

Dolonius et al. [5] explored decoupling and compressing color information from voxelized surfaces for real-time use. Decoupling method involves storing voxel counts, that counts from root to a node every preceding siblings plus one in order to index the colour data to a DAG node. Using the original SVO an array of colors is generated by depth first traversal of the tree. This array is later transformed into a 2D image using a 2D space-filling curve (Hilbert or Morton), then several various compression methods are used. In particular hardware accelerated texture compression was used such as BC7 and ASTC. Off-line image compression algorithms PNG, JPG and JPEG2000 were also used. The result is a method which allows for decoupling geometry and color, with color data ordered to a 3D-space filling curve with much colour coherency. It also showed that BC7 and ASTC provide with 3x compression with little loss in quality and with great speed and efficiency. They also considered the application of off-line image compression to be better adapted for their novel method.

Dado et al. [6] present a method compress arbitrary data by decoupling geometry and voxel data using a novel mapping scheme using palette compression in order to perform specialized compression for attributes and geometry separately. Their approach, based on an implementation of SVDAG, is to first assign indices to all nodes in the initial tree in depth-first order, to then calculate and offset for the difference between the index of child node and the parent node. The original index to the colour data will be produced when all offsets are summarized along a path from the root, i.e. the index is reconstructed via the offsets. The colours are stored in an attribute array in a depth-first order and using this mapping scheme the voxel attribute can be efficiently retrieved from the array. The voxel geometry is efficiently represented when all the geometry is stored in SVDAG after the decoupling. The attribute array still uses a large amount of memory, which is subsequently compressed. Their approach uses a variable-length, palette compression. The attribute array A = { a_0,..., $a_{\Lambda - 1}$ }, where Λ is the total number of leaf nodes in the SVDAG. A material array M = { a_0,..., $a_{\lambda - 1}$ }, where λ is the number of unique attributes in the scene, is used to store all λ unique attributes. Another array is used for storing indices into M. Since all attributes in A are stored depth-first, the spatial coherency of the scene is retained [6]. The attributes are partitioned into multiple blocks of consecutive entries, where each contains a number of different indices. There is an associated *palette* for each block containing necessary indices into the material array to retrieve all attributes in the block. Furthermore, a compression of the offset values, for mapping attributes, is performed by using fewer bits to represent them by analysing and finding the minimum number of bits required to encode offsets at each level of the SVDAG. This is possible because each node has an offset to its child that is at least + 1 offset, meaning these can be implicitly stored. Finally, the same compression technique is applied to child pointers. However, it is not as effective since most pointers generally require full four bytes per pointer. In short, the result from this approach outperforms existing state-of-the-art techniques, and also concluding that it is a well-suited method for GPU architectures [6].

2.3 Animation Compression

There has been some research previously done on compression of animated data, although these are mostly concerned with mesh-based geometry, which are types of compression that are not directly compatible or applicable for this project. Despite this, they are important to mention as they have indeed given insight and understanding to further motivate and influence the development of compression schemes used for this project.

Sattler et al. [13] used trajectories of vertices, clustered using Lloyd's algorithm in combination with principal component analysis (PCA) to segment each mesh parts that move almost independently. The parts can be compressed better than using standard PCA on the complete animation. The eigenvectors and weights in the clustered PCA (CPCA) are compressed in the time domain, and further compressed using quantization, as well as for connectivity data. Han et al. [14] compressed time-varying mesh sequences (frames) with the use of an extended Block Matching Algorithm to reduce temporal redundancy of the geometry data in time-varying mesh. The algorithm divides the frames into blocks which are compared with reference frames' blocks. An estimation is calculated to find blocks that match, which generates motion-compensated block subtracted from the current block to produce residuals. These are transformed with DCT and quantized. Finally, the DCT coefficients and motion vectors are entropy encoded. Zhang et al. [15] used an octree encoding for motion vectors of animated mesh geometry that capture the coherence of motion in spatial localities. A hierarchical octree is generated for each frame. For each frame motion vectors are generated by looking at the motion between frames. The vectors represent the motion between the previous frame to the current frame which are used to prediction of the next frame.

3

Theory

This Section provides with the the necessary context, theory and explanations in order to understand to further understand the aim of this project, the difficulties that may arise and how these may be mitigated. It will also give context to the implementation, as well give a detailed explanation of the theory behind each part of the implementation.

3.1 Definition of Voxel

A voxel is defined as closed, axis-aligned unit cube representing an attribute or element within a three-dimensional integer lattice, or more commonly a grid. The grid is defined as $\mathbb{Z}^3$, as a subset of a 3D Euclidean space $\mathbb{R}^3$ and whose coordinates are integers [22]. All voxels within the grid can be categorized either to represent objects or represent the transparent background depending on how each attribute is defined within the grid. In other words, the attributes only need to be considered to either mean occupied or empty within the grid [22]. A voxel can also be seen as an atomic unit of 3D model or a three-dimensional pixel [24]. Incidentally, The term *voxel* originates from a combination of the terms *pixel* and *volume*, meaning "volumetric pixel" [21].

3.2 Voxel Data Structures

Voxels can be stored in many different kind of voxel data structures. The purpose and application of each data structure depends on the storage and how it is used. Most types of data structures assumes that the voxel data will be static and the surfaces will not be reconstructed often or at all.

3.2.1 Voxel Grid

Example of a common data structure is a voxel grid. A voxel grid is a three-dimensional grid packed with data. Each cell in a grid must contain some data of the same type, e.g. color attributes. Dense voxel grids are considered a good representation for reconstructing surfaces, however these will eventually have very high memory footprint. Denseness entails more data to be packed into the grid, meaning more memory usage which becomes apparent with higher resolutions, making them

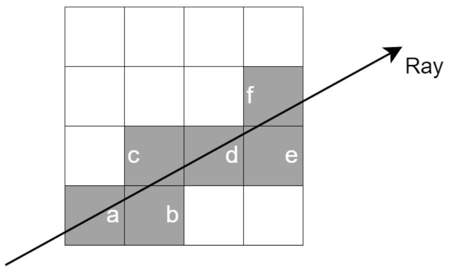


Figure 3.1: An example of how voxel grid traversal is performed in a two dimensional space. The principle is the same if extended to a three dimensional space [1]. Here, the ray traverses from its ray origin in a given direction and visits the voxels a, b, c, d, e and f in that order. This example and the figure are based on material written and created by Amanatides and Woo [1].

not optimal for efficient storage nor for interactive software [3, 2]. This is because ray tracing of voxel grids mostly processes empty volumetric data: more specifically, empty cells within a voxel grid will still contain data that is defined as an empty attribute because each cell of a grid must contain something. Therefore, it is widely considered more efficient to have a sparse representation for the data [7, 2]. This is especially true for higher resolutions.

3.2.1.1 Voxel Grid Traversal

Retrieval of data from voxel grids can be done through ray tracing. Amanatides and Woo [1] presented an incremental grid traversal algorithm for 3D space partitions. The algorithm traverses the grid through casting rays, defined as $\vec{u} + t\vec{v}$ for $t \geq 0$. where $\vec{u}$ is ray origin, $\vec{v}$ is ray direction and t is the span of the ray within a voxel. Each iteration, a ray is traced from $\vec{u}$ with direction $\vec{v}$, then traced along each axis (x, y, z) and computes the t value. Finally, it chooses the smallest one by comparison of each axis in order to determine the ray direction v. This procedure is continued along each cell of the grid until a voxel is found [1]. See Figure 3.1 for a visual example of the voxel grid traversal on a 2D grid.

3.2.2 Octree

Sparseness is a type of spatial coherence that efficiently represents only regions of the data that is non-empty [3, 2]. This can be represented with the use of an octree data structure. An octree is a spatial partitioning tree structure that has been constructed by recursively subdividing a finite scene into octants. The hierarchy of the tree represents the spatial partitioning of the 3D volume: each node represents an octant of the subdivided scene. The root node has eight successor nodes, all of which represent eight octants. Subsequently, each of these successor node are subdivided further into eight octants. Recursively this subdivision is performed until a certain threshold is met, such as until the subdivision produces octants becoming the smallest size defined by the threshold. This threshold could be, and usually so, the size of the voxels [20]. Incidentally, each node contains a pointer to each child node. This means for each node, there is a total of eight pointers to each child node.

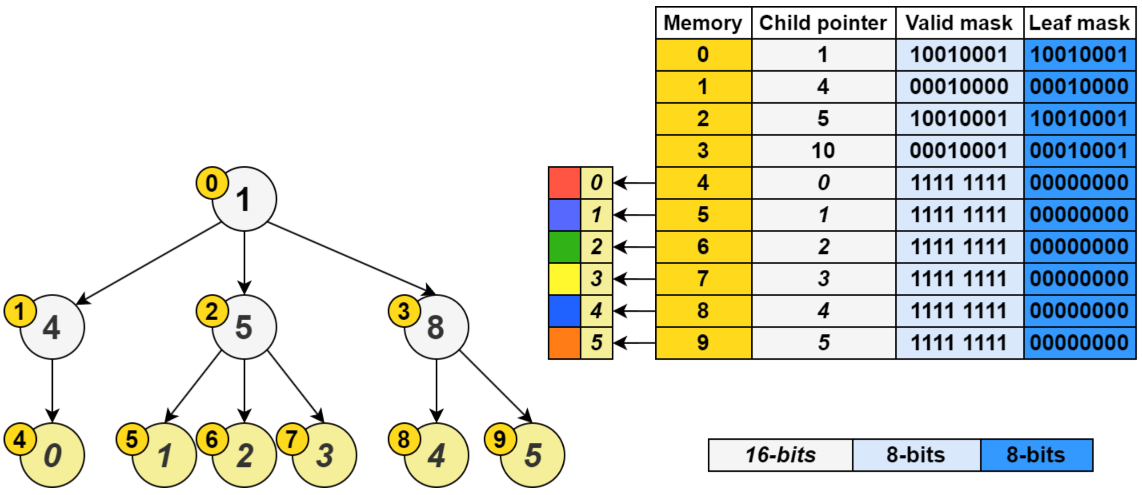
3.2.3 Sparse Voxel Octree

The storage of eight pointers to each successor node in the octree will become a very expensive proposition concerning memory usage. Efficient Sparse Voxel Octree (SVO) is a more compact octree data structure originally introduced by Laine and Karras [7]. The tree is constructed by recursive subdivision into octants, not unlike the standard representation of octrees. However, the difference is that the data structure is designed to minimize the memory footprint by a scheme that reduces most of the data to be stored in conjunction with its parent, i.e. divided into blocks that are stored contiguous in memory. This means that it removes the need to store the leaf nodes individually [7]. Within these blocks all references are relative in memory. These blocks are an array of child descriptors, a 64-bit descriptor that corresponds to a single non-leaf voxel. Leaf nodes are described by the parent nodes and therefore do no require their own descriptor. The descriptor is composed of two 32-bit parts, the first part describes the set of child voxels while the second part is concerned with contour data. The second part concerning contour data will not be described as it is unrelated to this project. More importantly, the first 32-bit part consists of 15-bit + 1 bit pointer for every non-leaf node and two bit masks, *valid mask* and *leaf mask*. The first 16-bits contains an index to the first child node of the current node, while the bit masks describe if there are any child nodes and if they are leaf nodes respectively. The valid mask is also used to locate where the child nodes are located in the memory. Since all child nodes are stored contiguous in memory, it is a simple matter of counting the bits, then fetching the corresponding child node using the bit counting as an offset. See Figure 3.2 for the tree layout and its corresponding array, as well has how nodes are traversed using the child indices stored at the corresponding node.

There are several variations of this data structure. One such is the sparse voxel directed acyclic graph (SVDAG or DAGs) first introduced by Kämpe et al. [4]. This is a data structure that reduces the SVO down to a sparse voxel directed acyclic graph (SVDAG or DAG) using a bottom up approach. The approach is to merge identical leaves, updating the child-pointers to the new leaves. The purpose is to find nodes with identical pointers and child masks that then are merged by only considering the root nodes until it is not possible merge any more. From this the smallest DAG is extracted [4].

It is important to note that the limitation of SVDAGs is that these only consider coherency of voxel geometry but are unable to store explicitly any other information concerning attributes, such as colour first and foremost [4]. Despite this, there are methods to implicitly encode and retrieve colour data from an SVDAG. Dado et al. [6] and Dolonius et al. [5] describe various methods to map colours and retrieve colour data from SVDAGs, described in detail in Section 2.

Other variations of this data structure exists. Here are a but a few examples: Symmetry-aware SVDAG [17] is a more efficient compression of DAGs through a similarity transform and storage of pointers using variable bit-rate encoding. Clustered SVO [16] is a new hierarchical data structure based on SVO that uses wide range of pointer lengths for the child nodes to achieve more compact encoding. Pointerless



Memory	Child pointer	Valid mask	Leaf mask
0	1	10010001	10010001
1	4	00010000	00010000
2	5	10010001	10010001
3	10	00010001	00010001
4	*0*	1111 1111	00000000
5	*1*	1111 1111	00000000
6	*2*	1111 1111	00000000
7	*3*	1111 1111	00000000
8	*4*	1111 1111	00000000
9	*5*	1111 1111	00000000

16-bits	8-bits	8-bits

Figure 3.2: An example of a Sparse Voxel Octree and the underlying array of child descriptors. As can be seen, the nodes in the tree store an index to the nodes first child, while also keeping track of the number of children nodes shown in the 2 8-bit masks valid mask and leaf mask. Note that the leaf nodes at the bottom of the tree point to a colour array rather than storing a colour directly. The layout at the bottom right of the image corresponds to the explanation given in Section 3.2.3. Figure was inspired by the work of Laine and Karras [7]. This example uses a quadtree for simplicity sake.

SVODAG [18] uses a pointerless approach to the construction of SVDAGs, i.e. no pointers are stored in each child node and only use a purely binary representation for the octree.

3.2.3.1 SVO Traversal

Laine and Karras [7] describe their rendering algorithm for traversing each voxel by casting rays against the octree and tracing them independently. Defining a ray as $p_t(t) = \mathbf{p} + t\mathbf{d}$, with t along any axis n (x, y and z axes) for an axis-aligned plane of a cube, $t_n(n) = (1/d_n)n + (\text{-}p_n/d_n)$. The axis-aligned cube is represented as pair of corners $c0 = (x_0, y_0, z_0)$ and $c1 = (x_1, y_1, z_1)$ where $t_n(c0_n) \leq t_n(c1_n)$. The span of t-values are defined as $tc_{min} = \max(t_x(x_0), t_y(y_0), t_z(z_0))$ and $tc_{max} = \max(t_x(x_1), t_y(y_1), t_z(z_1))$. The octree is mirrored, redefining the coordinates so each component of $\mathbf{d}$ becomes negative. More specifically, this is done by determining an octant bitmask based on the direction of the ray at the beginning of traversal. A logical XOR (exclusive or) is performed on the octant mask when permuting the child slot indices when accessing them from the field of child descriptors. This mirroring guarantees that $x_0 > x_1$, $y_0 > y_1$ and $z_0 > z_1$ and makes the functions t_x, t_y and t_z descending. This is done to simplify the computation of tc_{min} and tc_{max} which depend much on the sign of $\mathbf{d}$ [7].

Voxels are stored in by their parents, therefore are all expressed by using the data associated with the parent. That is, the current voxel is expressed using its parent voxel *parent*, holding the current descriptor and child slot index *idx* ranging from

0 to 7. Since there is no way to store spatial location of voxel there is a need to maintain a cube that corresponds to the current voxel. The cube is defined as using a non-negative integer *scale* that defines the extent of the cube with exp_2(scale - s_{max}), where s_{max} is the max scale of the cube. It is also defined with a vector *pos* = (x_1, y_1, z_1) that has range between 0 to 1 in all dimensions. Positioned at the origin, the entire octree is defined and contained by this cube of size s_{max}. It is then possible to determine the next voxel of the same scale along the ray by comparing $t_x(x_0)$, $t_y(y_0)$ and $t_z(z_0)$ against tc_{max} then advancing the cube along the axis. If a voxel has the same parent, a new child slot index is obtained by flipping the bits of *idx* corresponding to the same axes. Also, *idx* and *pos* need to be unmirrored after traversing the octree since they are mirrored according to the octant mask [7].

Incidentally, it is not enough to use an incremental traversal but must instead be a hierarchy traversal. It is necessary since SVOs express sparseness in that they can only store voxel data represents non-empty voxels [7]. Liane and Karras's [7] algorithm intersects the voxels using a ray that traverses the octree in a depth-first order. The algorithm incorporates the use of a stack (of max depth s_{max}) that stores the parent voxels associated with the parents of the current voxel. The entries of the stack are addressed using the cube scale values. Each iteration, there are three cases or stages for fetching a voxel:

- *Push*, means to proceed to the child voxel if the ray has entered it. That is, this is executed when descending the hierarchy, potentially storing previous parents into the stack at position of the *scale* based on on a conservative check, i.e. if $h < tc_{max}$, where h is the current t_{max} [7].

- *Advance* means to select the next neighbouring voxel. In order to know if the ray position stays with the same parent voxel, the first step is to assume that it does stay and then compute candidate positions for *idx** and *pos**. Finally, if the resulting *idx* is valid compared with the ray direction by observing if the flipped bits are not conflicting and if this is true, then it does advance to the next voxel using the candidate positions of pos* and *idx** as their new values respectively. the candidate *idx** is obtained by finding if there are any conflicting bits that have been flipped in the current *idx* [7].

- *Pop*, means to proceed to the next neighbouring voxel of the highest parent that the ray exits. This is executed when ascending the hierarchy while, and uses the current *pos* to find new values for *pos'*, *scale'* and *idx'*. That is, *scale'* by finding the hiughest bit between *pos'* and *pos*. From this, the child slot index *idx'* from *pos** from the corresponding *scale'*. *pos'* is obtained by from *pos** but clear the bits that corresponds to *scale'*, which will derive a cube witht the correct scale factor that contains *pos**. Finally, by popping from the stack, the parent voxel is restored [7].

3.3 Lossy Compression

Lossy compression is a type of compression that incorporates methods to represent or to look the same as the original data by reducing, discard and and approximating the

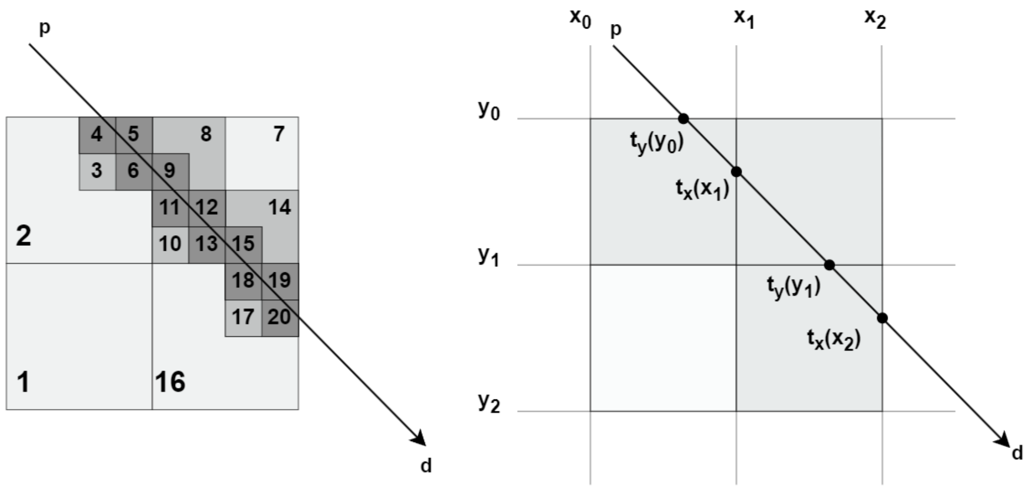


Figure 3.3: The example to the left demonstrates how the ray traverses the octree in a hierarchical order. The example to the right demonstrates how the ray traverses each voxel at equal scale. The figure was inspired by the work of Laine and Karras [7]. This example uses a quadtree for simplicity sake.

original data. This is done in order to reduce memory storage or for more efficient transmission of data [11]. For colours, there are many different type of methods, most resorting to discard and reduction of the data in some way. The most common type of lossy colour compression is using vector quantization, or specifically colour quantization. This is a type of technique used to reduce the number of colours in the original data to the point of keeping the original data having mostly similar fidelity and quality although with reduced memory storage [11].

The simplest of colour quantization is using indexed colour, where colours in images are instead represented by a number that refers to a distinct colour in a colour palette. The palette only contains distinct colours instead of storing each individual colour which multiple ones can be exactly the same. However this type of compression can only be gained from this method if the number of colours are small enough, otherwise the compression ratio will be negligible. For example, if a voxel grid contains elements of size 8-bit, this means only 256 unique colours can be referred to when rendering. If more than 256 colours are needed, e.g. using size of 32-bit, indexing will not gain any true compression as this is the equivalent of storing 24-bit RGB colours within the grid. In this case, a more useful application of indexed colours could be used for decoupling of colours from geometry [6, 5] while choosing other types of lossy compression techniques.

Another type of colour quantization is using octrees to store to store each sample in RGB space [23]. This is done by branching out each node depending on the most significant bit of each colour sample, and each successor most significant bit is used for each lower depths of the octree. Each colour is represented by a leaf of depth eight, while all intermediate nodes are the subcubes of the RGB space [23]. The whole tree uses a binary representation for each colour. Although the compression ratio and memory efficiency is high using this method, it is not suitable for neither voxel grid nor voxel octrees. This is because voxel grids need to store data directly within the grid, while for voxel octree it will be somewhat of a redundant solution. For both voxel grid and SVOs, the author believes a more suitable solution is to use a form of texture compression and efficient storage of the voxel colours rather than color quantization using octrees. A more efficient compression method using colour quantization is S3TC or DXT texture compression, explained in detail below [26].

3.3.1 S3TC Texture Compression

S3TC, or also known as DXT [25, 46] is a lossy texture compression format where each pixel in an image is divided into sets of 4x4 blocks (which are non-overlapping) and these pixels within the blocks are quantized to a set of values. These values are approximated with points on a line in the color space in order represent colours. A minimum and maximum, that have been quantized into into 16-bit, 565 RGB colour format, define the span of this line, and each colour in the block is stored as interpolation values points along this line. This interpolation value will construct a bitmap colour based on where it is located along the line [25, 46]. This compression format has five formats designed for specific image data types, however all have a procedure similar to the one described previously. The difference between each format is how information concerning alpha is stored, while DXT1 does not support alpha transparency [46]. Each format also stores the blocks differently, varying in both size and type of data stored, i.e. related to alpha transparency data. in Section 3.3.1.1 there is an detailed explanation on the DXT1 format. The other formats will not be discussed further as these are out of scope for this project.

3.3.1.1 DXT1

DXT1 is the smallest variation of S3TC, constructed exactly as described above. The blocks are encoded as 64-bit, each with 2 16-bit for the minimum and maximum 16-bit 565 RGB colours called c_0 and c_1 and 32-bit for 16 codewords that are 2-bits each [26]. The 2-bit codewords define how to combine c_0 and c_1 to produce the right corresponding colour to the original image data, although in a compressed format. If c_0 is larger than c_1, then calculate two color components $\mathrm{c}_2 = \frac{2}{3}\mathrm{c}_0 + \frac{1}{3}\mathrm{c}_1$ and $\mathrm{c}_3 = \frac{1}{3}\mathrm{c}_0 + \frac{2}{3}\mathrm{c}_1$. Otherwise, if c_0 is lesser or equal than c_1, then $\mathrm{c}_2 = \frac{1}{2}\mathrm{c}_0 + \frac{1}{2}\mathrm{c}_1$ and c_3 becomes a black color representing transparency as a pre-multiplied alpha format. These calculations are performed by first finding checking the block of codewords that determine the value for each pixel [46]. Because transparency is considered a black colour in this format, this will cause artifacts in the output, such as a black border surrounding transparent areas in a texture when linear texture filtering is applied. This is because the interpolation will be done between a black (transparent) texel and an opaque texel that are neighbours in the texture image [46].

Although this format has a fixed compression ratio of 6:1 for 24-bit colours (alpha is 255) and 4:1 for 32-bit colours [46], due to how all colours in a block are interpolated between two minimum and maximum colour values, this will result in quality degradation. For example, if a block is occupied by colours that are not on a linear line in the colour space, this results in an interpolation which will not match the

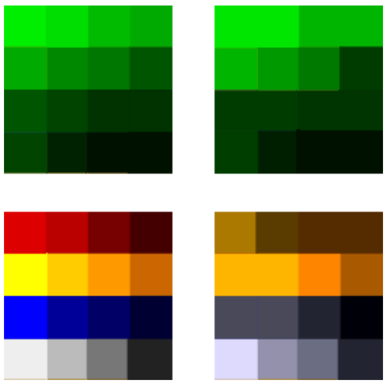


Figure 3.4: An example of how the DXT1 compression format affects quality of the colours. Although the compressed set of green colours were able to preserve the quality relatively well, the bottom set of colours have severe loss in quality. This happens due to it being many different colours across the RGB colourspace, therefore the DXT1 block is unable to encode and interpret the colours correctly. Inspired by a figure created by FSDeveloper [48].

original data at all [48]. See Figure 3.4 to see how the quality can be affected by this compression format.

3.4 Definition of Sequence-based Animation

It is important to give a clear definition of the term *sequence-based animation*. The definition is grounded in the meaning of traditional animation, i.e. a *sequence* of drawings on a transparent background that represents the illusion of movement between several frames when played. A "frame" means the drawing itself representing the state of motion in time. This collection of frames together form the illusion of motion [27]. For this project the definition remains similar, although each frame in an animation sequence represents a corresponding voxel model that creates the illusion of motion. The term "frame-based animation" is interchangeable with "sequence-based animation". Formal definitions for how this type of animation can be represented in voxel data structures is given in Section 3.4.1 and 3.4.2 for voxel grid and octrees respectively.

3.4.1 Voxel Grid

For a set of voxel grids G, a sequence is defined as set of continuous voxel grids of size n^3, where n is a value that is power of 2, $G_i = \{n_0^3, n_1^3, \ldots, n_m^3\}$ where i is the current time step and m is the total number of voxel grids in the sequence where $i \leq m - 1$, $i \in \mathbb{N}_0, m \in \mathbb{N}_{\neq 0}$. Thus, the full size of a sequence-based animation voxel grid becomes $n^3 \times m$.

3.4.2 Octree

For a set of octrees T, a sequence is defined as a set of continuous set of subtrees s, where $T_i = \{s_0, s_1, \ldots, s_m\}$, where i is the current time step and m is the total number of octrees in the sequence where $i \leq m - 1$, $i \in \mathbb{N}_0, m \in \mathbb{N}_{\neq 0}$. Each s_i consists of a total of nodes $N_i = C_i + L_i$, where C_i is the total non-leaf nodes and L_i is the total leaf nodes.

4

Methods

This section explains how all necessary methods were implemented and what was taken into consideration during their implementation. The implementation of the methods are strongly influenced by the descriptions and implementations of the original authors (which are mentioned in each following sections). However, there were some necessary modifications that had to be made in order to adapt them for the purpose of this project.

Although construction and compression is done entirely on the CPU, in an offline approach, rendering and decompression is done in real-time on the GPU through the use of vertex and fragment shaders. All implementations were done using OpenGL and C++.

4.1 Rendering and Data Structures

All rendering is performed on the GPU using vertex and fragment shaders. The setup for the rendering is a blank canvas constructed from two triangles. On the fragment shader both rendering and decompressing of colour data is performed. Since block descriptors are stored at the leaf level of the hyper octree and the compressed grid, it is merely a case of finding the corresponding block and decompress it. Incidentally, the vertex shader is merely used for passing the two triangles to the GPU and nothing more.

Since rendering voxel grids and sparse voxel octree differ in many ways, the implementation of their respective rendering algorithm and how the voxel data is interpreted by the respective algorithm is also quite different. See Section 4.1.1 and Section 4.1.3 for more information of how this was implemented.

4.1.1 Voxel Grid

The voxel grid was implemented as a volume texture, or 3D texture, then ray traced using the technique introduced by Amanatides and Woo [1]. A 3D texture is a container with dimensions of width, height and depth. The 3D texture can be populated with any data type of choosing, but for its uncompressed format it is populated with 24-bit RGB colours: both for empty and non-empty voxels, where a colour value of 0 interpreted as an empty voxel. The 24-bit colours always assume that the alpha channel is 255, while for empty voxels it is 0. The data can be sampled

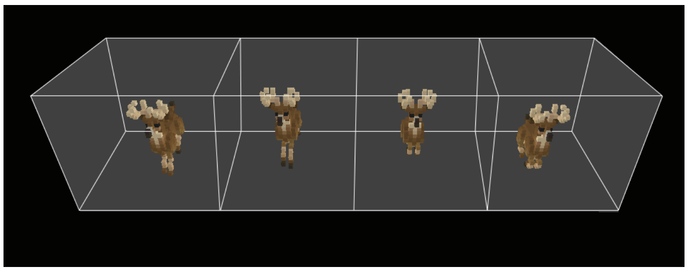


Figure 4.1: Illustrative figure of how each voxel data in an animation sequence is stored using a voxel grid. Each voxel data is stored within a voxel grid the size of N^3 and represents the animation sequence at a certain time step. Each time step has an offset the size of N. Original model was created by ephtracy [43].

by accessing through a shader using coordinates with these three dimensions, all ranging between 0.0 and 1.0 [47]. In order to access the data, two important steps must be done during the traversal. The first step is to check if the ray is intersecting with the voxel grid. This is a ray-box intersection implemented as modified 3D SDF (signed distance field) using Inigo Quilez's implementation of a box SDF primitive [49], where it is only necessary to check if the ray is within the primitive using the current position and the half size of the current voxel data. If this is true, then it is possible to test if the voxel fetched at the ray's current position is non-empty or not. If it is non-empty, then it is possible to sample the 3D texture by taking the current position and then normalizing it using the full size of the voxel grid. Let $\vec{p}$ be the current position of the ray and let $\vec{s}$ be the size of the grid, then define the normalized coordinate $\vec{c}$, where $\vec{c} = \frac{\vec{p}}{\vec{s}}$. using $\vec{c}$, the corresponding voxel in the voxel grid may be fetched from the 3D texture.

The normalized coordinate $\vec{c}$ must also be a given an offset that will position the ray at the right voxel at the right time step. Note that the animated voxel grid will not be uniformly sized, please refer to Section 3.4.2 for further information.

4.1.2 Compressed Voxel Grid

A compressed voxel grid is implemented similar to a regular voxel grid, except that that each 3D texture is populated with 16-bit indexing values. These indexing values are actually block descriptors, which are explained in detail in Section 4.1.7.2. The index 0 is a reserved value used to represent an empty voxel, just the same as it is used in a regular voxel grid. The term "compressed grid" will be used hereafter.

4.1.2.1 Animation sequence

Constructing an animation sequence is trivial. Define a 3D texture that is of size $N^3 \times M$, then populate the texture at each coordinate corresponding to each voxel in the voxel scene per time step. See Figure 4.1 for how each voxel data per time step is stored in the 3D texture.

4.1.3 Sparse Voxel Octree

Sparse voxel octree data structure was implemented as described by Laine and Karras [7] but with some modifications. First, this implementation uses an index-based approach instead of C-style pointers. This has the benefit of being easier to convert to an array readable by a fragment shader, but introduces some complications for removing and reassigning the pointers correctly. See Section 4.1.4.2 for how the reassigning of indices is done.

Second, the format of the child descriptors has been changed substantially compared to Laine and Karras's original format, at least for the implementation of the Hyper Octree, see Section 4.1.4. For regular SVO, the format is only slightly altered for simplicity sake, using 16-bits for child indices and ignoring any regard for far pointer semantics. Otherwise the format is remains unchanged.

For the Hyper Octree the format is quite different. Instead of having the format being 15-bit for child indices + 1 bit far pointer and 2 8-bit masks for valid and leaf bits respectively [7], the format is now 23-bit for child indices, 1-bit flag that is set if the node is redundant (i.e. spatiotemporal coherency) and 1 8-bit leaf mask. The 23-bit child indices will allow for larger resolutions of voxel data because more nodes may be stored, as much as up to 2^{23} of total nodes compared to only 2^{16} of total nodes. The 1-bit flag allows for explicit checking if a node is marked as redundant or not, replacing the original semantics regarding far pointers. Also, semantics regarding contour data section of the child descriptor will not be added to this format as it is out-of-scope for this project. This results in a reduced child descriptor from 64-bit to 32-bit. Despite now only utilizing 1 8-bit leaf mask, the usage of 2 8-bit masks as Laine and Karras describe [7] can still be done implicitly, since the value of both bits are always the same except at the leaf level. At the leaf level, the valid mask will have all bits set (0xFF) and the leaf mask will have all bits cleared (0x00). Therefore, their values can always be stored implicit using only 1 8-bit mask. Otherwise, the functionality of the bit mask remains unchanged to the functionality of the original 2 bit masks.

The rendering of the Sparse voxel octree is based on the ray cast algorithm source code described and presented by Laine and Karras [7]. Although no substantial modifications were made to the implementation of their ray cast algorithm, a few but important additions were added for exploiting spatiotemporal coherence in the voxel data. Every time a new node is fetched, an iterative search may be performed if the node is marked as redundant in order to find the corresponding subtree in a previous time step. Also, the original offset value for the time step is saved on the stack in order for the ray casting algorithm to render the voxels properly, regardless if the current node is marked as redundant or not. See Section 4.1.4 for a more detailed explanation how the spatiotemporal compression is performed.

In contrast to Liane and Karras that perform voxelization of non-voxel data [7], this implementation does not perform any voxelization but instead builds an SVO out of a set of pre-defined voxel data. The SVO is constructed in a depth-first order, with a max depth of the octree defined by the resolution of the voxel data N, i.e. $depth_{max} = \log_2 N$, where N must be a power of 2.

The colour data and voxel geometry is decoupled, which means that at the leaf level of the octree there is only an index that points to the position in a colour array where the colour data in actuality is stored. This was done in order to compress both the voxel geometry and the colour attributes separately, similar to previous work as mentioned previously.

4.1.3.1 Animation Sequence of SVO

Constructing an animation sequence for octrees involves merging several octrees into and stored a contiguously in memory. Each size of the octree (i.e. the number of total nodes) becomes the offset value between each frame. Since the colour data is decoupled, this means that a unique offset value for colour data also needs to be calculated. These offset values are stored in an array at indices representing the corresponding time step.

4.1.4 Hyper Octree - Spatiotemporal Compression

In order to exploit spatiotemporal coherency in order to compress the animated sparse voxel octree, the author's method of choice is influenced through a combination of the methods described by Kämpe et al. [4, 3] for compressing redundant nodes in an SVO and Ma and Shen's method [12] for encoding an octree to merge similar subtrees in consecutive time steps. The method used to find similar subtrees in a tree structure was created by the author independently, although was loosely inspired by the method described by Grossi [28] for finding common subtrees between ordered trees. This SVO format has been given the name Hyper Octree by the author in order to differentiate this the regular format that is uncompressed.

4.1.4.1 Subtree Similarity Search

The first step of the method is to create a record of all subtrees in each octree. A subtree must have at least one child node, therefore individual nodes are not considered subtrees. Each record contains a codeword as a text string, index and a colour array: a text string S that encodes all possible edges for each subtree, depth value D, node index I and an array A of all colours stored at the leaf level of the subtree. The records are stored in the chronological order of the animation. See Figure 4.3 for an example of how subtrees are recorded.

Once all records have been generated, it is then possible to begin a comparison of all subtrees and find subtrees that are the same between two consecutive octrees. Given two consecutive octrees O_j and O_i in a Hyper Octree H, where i is the next consecutive time step after j. Using each octree's corresponding subtree record R_j and R_i, the comparison is to check if any text string in both R_j and R_i, and each corresponding colour array, are equal. If this is true, the subtree is marked and the whole subtree is removed from O_i, except for the root node of that subtree. In addition, a new index $O_{i_{new\_idx}}$ is fetched from R_i, and stored in memory $O_{j_{new\_idx}}$ to be later used for reassigning the node's child index (See Section 4.1.4.2). This node is given a special marker by setting the 1-bit flag to one, ensuring that during

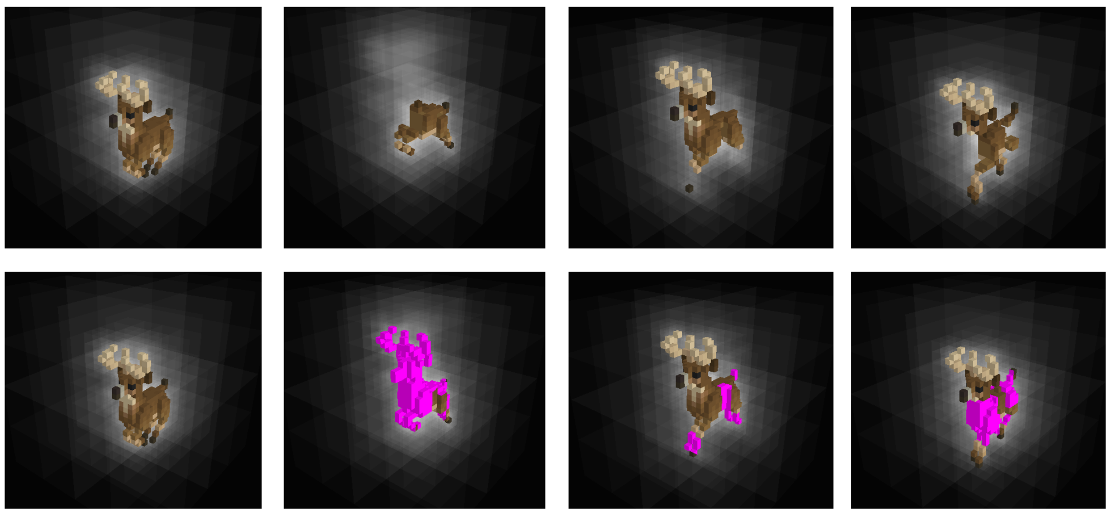
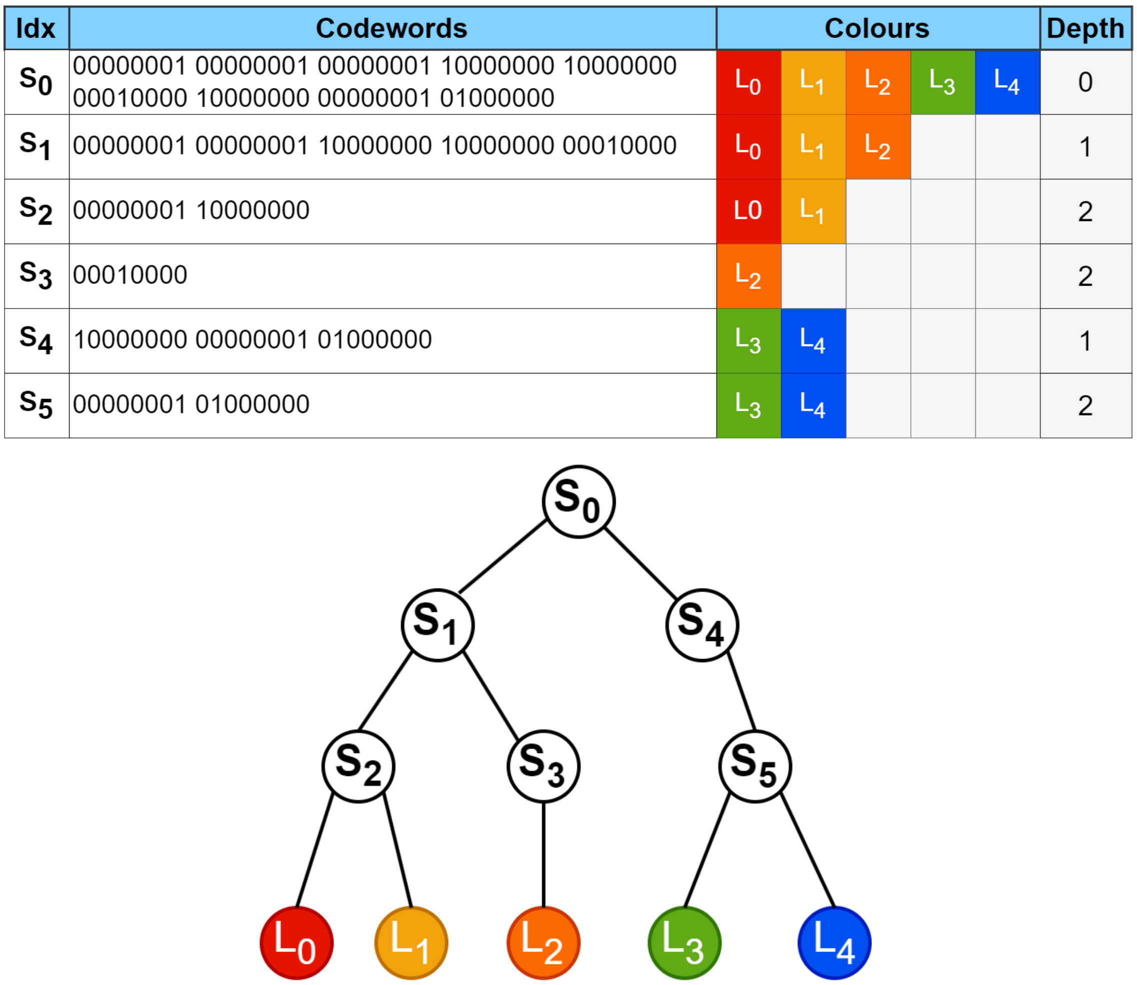


Figure 4.2: Example of how exploiting spatiotemporal coherency allows for reconstruction of each octree at a given time step by pointing to identical subtrees in previous time steps. In the above series of figures it can be observed what happens after similar subtrees have been found and subsequently removed from the octree. The bottom series of figures demonstrates how geometrically the model appears when it is reconstructed using subtrees that exists in an previous time step. The the magenta-coloured voxels are only illustrative of this result. Original model was made by ephtracy [43].

traversal that this node is marked as redundant. This works because under normal circumstances this bit is set to zero. The child index is then updated to point to the relevant the subtree in previous time step in octree O_j.

4.1.4.2 Reassign Indices

After all redundant subtrees are removed, it is important reassign indices to point to the right location in memory. A simple method is used to achieve this. Before the nodes are removed, record the current location index I that every node is stored at in memory. Then, for every octree O and for each node that has been marked as redundant, remove it from the node array including the leaf nodes. Then once all nodes have been removed, reassign every node by for every node N_i, for for a O_j in the same array where it's child index C_{O_i} equals the previous location index I_{O_j}. If these are equal, assign index j to be the new child index of node N_i. Also, for every node that has the special marker, assign the stored new index $O_{i_{new\_idx}}$ to the child index of $O_{i_{child\_idx}}$.



Idx	Codewords	Colours					Depth
S_0	00000001 00000001 00000001 10000000 10000000 00010000 10000000 00000001 01000000	L_0	L_1	L_2	L_3	L_4	0
S_1	00000001 00000001 10000000 10000000 00010000	L_0	L_1	L_2			1
S_2	00000001 10000000	L0	L_1				2
S_3	00010000	L_2					2
S_4	10000000 00000001 01000000	L_3	L_4				1
S_5	00000001 01000000	L_3	L_4				2

Figure 4.3: Example of how subtrees are recorded in a given octree. The index value is the index where each corresponding node is stored in memory, not to be confused as the node's child index. Note that only a node with at least one child node is considered as a subtree, while leaf nodes alone are not considered subtrees. This example uses a quadtree for simplicity sake.

4.1.5 Exploiting Spatiotemporal Coherency during Traversal

During traversal, it will now be possible to find a redundant subtree because of the special marker described previously. For every iteration in the traversal, a node is fetched from the octree using an appropriate offset value from the corresponding time step stored in an auxiliary array. If the currently fetched node has a 1-bit flag that is non-zero, this means that it is redundant and therefore it becomes possible to retrieve a subtree from the previous time step. This exploits the spatiotemporal coherency in the voxel data between each consecutive time steps. If the 1-bit flag is zero, this means that this is a unique subtree and therefore no subtree from the previous time step will be retrieved. This process guarantees that spatiotemporal coherency will be achieved during the traversal of the octree. Once the process is complete, the traversal continues as normal. See the pseudo-code below for this method.

```
function FindCoherentNode(nodes, timeStep, descriptor)
  if descriptor is redundant
      index := fetch index from descriptor
      descriptor := nodes[index + offsets[timeStep - 1]]
  return descriptor
```

4.1.6 Colour Compression

Compression of colour data is done separately from the spatiotemporal compression. Therefore, colours are stored in an colour array, similar to Dado et al. and Dolonius et al. [6, 5]. The colour compression of choice is S3 Texture Compression, specifically the DXT1 format. This format was chosen because of its high compression ratio of 6:1 for 24-bit RGB colours [25, 46]. See Section 3.3.1 for more information about how this compression format works.

Indexed colours are used in both voxel grid and sparse voxel octree implementations, although voxel grid only uses this for its compressed grid format, while octrees use it for compressed and uncompressed formats. Assignment of indices is done at the construction stage for both voxel grids and octrees which is trivial. Reassigning indices after compression procedures is less trivial and is explained in Section 4.1.7.2.

DXT1 compression while octree uses this for both uncompressed and compressed formats. Assignment of indices is done at the construction stage for both voxel grids and octrees which is trivial.

4.1.7 DXT1

The Compression and decompression procedure using DXT1 is done as previously described in detail in Section 3.3.1. The compression procedure is implemented using an *stb* library originally created by Fabian "ryg" Giesen [44]. Once the voxel colours are compressed, the 64-bit blocks are stored in an array of 2D unsigned integer vectors (i.e. uvec2) and each component is of size 32-bits. This is because of an inherent limitation where is no other way to pass a 64-bit long unsigned integer by itself. Once all the data is stored, the array is passed into the fragment shader, where only a single block needs to be fetched decompressed. The decompression is done as needed and since the compression is done in a depth-first order, the original order is retained after decompressing and therefore colours can be fetched without complications. Therefore, no other auxiliary data structure is utilized, achieving a 6:1 compression ratio [46]. Finally, the original voxel colour array is removed from memory after this process has been finalized. The implementation for the decompressor was inspired by the source code written by Benjamin Dobell [45].

4.1.7.1 Block Descriptors

Accessing blocks is done on the leaf level of the octree. Since the original voxel colour arrays are no longer needed, these are replaced with DXT1 block descriptors. Block descriptor is a either 2 bytes of 4 bytes descriptor that packs information concerning accessing the right DXT1 block. The information is stored within a leaf in the octree and describes which block the current compressed colour is stored at and which of the pixels within that block needs to be fetched. The format for the block descriptors are very similar between voxel grids and octrees and only differ whether to use 2 bytes or 4 bytes for the entire block descriptor.

For octrees, these descriptors are of size 23-bits, consisting of 4-bits that is used to identify which codeword within the block needs to be decompressed and 19-bits for identifying which block needs to be fetched. More specifically, the first 4-bit is a mask which are used to represent a value between 0 and 15. This value is the identifier to find which 2-bit codeword needs to be used to fetch the right pixel in the DXT1 block. For voxel grids, the descriptors are very identical with the exception that the descriptors are packed into mere 16-bits. These descriptors consists of the same 4-bit mask as previously described and use 12-bits to store the index to the block to needs to be decompressed.

The block descriptors are also correctly mapped in the octrees and voxel grids using a simple iterative process. This is done by examining two colour values, one stored at the original location in the voxel colour array before sorting and one stored in a new location after sorting. If these two colours are the same, then the index of the original location is used to store the block descriptor in the leaf node. This achieves a similar result to using a Morton ordering like Dolonius et al.[5] and Lee et al. [19] used for mapping voxel colours to 2D images. The reason for the approach used in this project was because of its simplicity and because of time constraints. In the end, the author believes that the result should achieve the same result. See the pseudo-code below for how this procedure is implemented.

```
function MapColours(octree, oLeafs, sLeafs, blocks)
   markedLeafs is an array of bools, size of leaf amount
   i: = 0
   while i < size of oLeafs
      j: = 0
      while j < size of sLeafs
          if marked[j] is true, j := j + 1 then continue

          if oLeafs[i] equals sLeafs[j]
             store blocks[j] at octree[i]
             set markedLeafs[j] to true
             then break
          else j := j + 1
      i := i + 1
    return octree
```

4.1.7.2 Redundant Blocks

Due to the nature of the DXT1 format to encode the pixels of a texture into 4x4 blocks [46], this means if multiples of the same colour are encoded into several blocks, the final result will produce a huge amount of redundant set of blocks. This is especially true in this case where across multiple time steps many redundant colours will be stored in several blocks. This becomes very wasteful of memory usage and will exceed the limitations of the number of bits that are used for fetching the block index. Voxel grids are limited to only 2^{12} number of indices, while octrees may use up to 2^{23} which is considerable more in comparison. However even with octrees even these would not be enough as the number of blocks would exceed beyond this amount at higher voxel resolutions. Besides, the author speculates that this may also be the case if the colour array contains many redundant colours no matter how which voxel resolution is used in the end. This is definitely the case if many frames are used in an animation sequence.

Because of this, a simple procedure was created by the author in order to reduce the blocks down to only the most unique block of colours, removing all redundant blocks. The procedure is as follows: after storing all block descriptors as described in Section 4.1.7.2, it is then possible to remove redundant blocks from memory by sorting all blocks, then finding all unique elements in memory and subsequently discard all redundant ones. This is implemented by first sorting all blocks using std::sort, then erasing all redundant elements using std::unique and a data structure erase operator [50]. This will result in an array of unique DXT1 block colours.

Afterwards, it is important to once again remap the indices for each block in the octree at the leaf level. This part is in a similar way to the pseudo-code described in Section 4.1.7.2, however instead of comparing leaf nodes, the comparison becomes between each DXT1 block associated with each leaf node to the new set of DXT1 block, performing a search to find each corresponding block at a new index value.

When a corresponding index is found, the codewords from the old block descriptor is retrieved then inserted into the new block descriptor that now contains the correct new index value.

4.1.8 HSV Model

HSV is an alternative model of the RGB monitor gamut based on perceptual variables, which are hue (H), saturation (S) and value (V) (or brightness). The model is described using a layer of disks where each of the values has a point, all of which describe different values based on human perpetual colourspace. Hue (H) describes the position of a colour in on the disk, ranging between 0 to 360 degrees. Saturation (S) describes the gray value of the hue. Value (V) is the intensity of the saturation [29]. The author hypothesizes that these characteristics could be used to sort colours in such a way that would put colours that are perceived to be close to each other in the colour spectrum. This may be possible because of the aforementioned attributes of using such model. A method for how this is achieved is described in Section 4.1.8.1.

4.1.8.1 Voxel Colour Sorting with HSV

When compressing colours using DXT1, there will be some quality degradation because of the limitations of the format as described in Section 3.3.1. to alleviate the quality degradation or possibly mitigate it, a novel approach has been used to sort the colours using HSV values. This is done before colour compression, then after compression the leaf pointers must be updated to point to the new sorted order. This is a novel approach based entirely on the characteristics of HSV values [29] and also inspired by Lee et al.'s approach of sorting colours that have been packed using Morton order [19]. Although this attempt does not use any space-filling curve to pack the colours, the same result is achieved using a simple procedure explained in Section 4.1.7.2. The reason for this was because of time constraints.

As stated previously, this approach is done before any type of compression is performed. The approach is quite simple: In order to sort by HSV, a list is created that contains HSV values alongside corresponding 24-bit RGB values. The HSV values are converted from the RGB values using the algorithm described by Smith [29]. Then, using std::sort of the standard C++ Library [51], it becomes possible to sort this list based using Hue (H), Saturation (S) and Value (V) in any order of six possible permutations. Finally, the sorted list's RGB value is transferred to colour array of the leaf nodes. See Figure 4.4 to observe an example of how the difference in possible quality using this method using static voxel data compared to unsorted colour compression.

4.2 Animation Support

In order for the voxel scene to become animated the voxel scenes need to be encoded to interpret several concatenated voxel scenes as different time steps in an animation. For both voxel grids and octrees this is quite trivial: calculate an offset for each

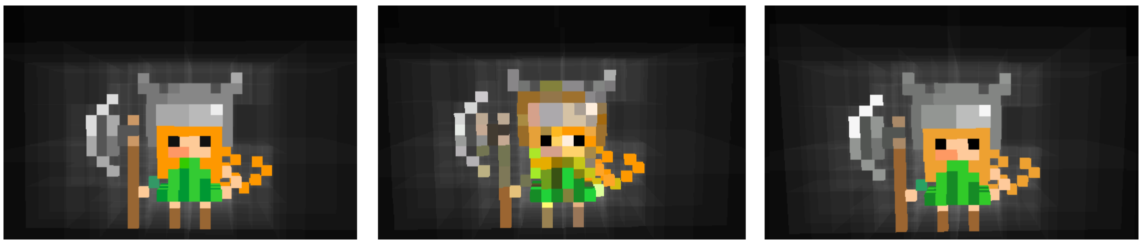


Figure 4.4: An example of how HSV can order colours to be closer to each other for the DXT1 compression results in better preserved quality when compared to an unsorted attempt. The image to the left is uncompressed, the middle is DXT1 compressed unsorted and the right image is DXT1 sorted using the permutation VSH, one of six possible permutation of HSV colour sorting. The model was derived from MagicaVoxel [32].

time step, then pass a shader uniform parameter that controls which offset to use in order to render out the corresponding time step in the animation. The type of offset value differs between voxel grids and octrees. For voxel grids, this is offset value represents an offset of the current ray position, i.e. since the animation is encoded as a collection of several voxel grids it is only necessary to offset the position of the ray to a voxel grid beyond what is rendered on the screen. For octrees, this is an offset that points to the root node of each octree. This offset value is based on the the size of each octree, with every offset per octree accumulating the previous offset values, since the full octree is stored linearly in an array.

4.3 Evaluation

The evaluation of the methods described above will be done in order to give a conclusive answer to the problem statement. Each part of the evaluation will be used to test and compare the results with various test data (See Section 4.3.1). The evaluation will compare primarily three types of data structures: voxel grids, SVO and Hyper Octree. It will also encompass a comparison between all six possible permutations of the HSV sorting scheme, in order to evaluate how successful each permutation is to minimize quality degradation when the DXT1 colour compression is used.

4.3.1 Test Data

Although there is plentiful of test data for static voxel data, there is lack of voxel data for sequence-based animation. Therefore, most of the test data used for evaluating this approach had to be created by the author. Those that were found for the evaluation were created by ephtracy [43] and can be seen in Figure 4.13.

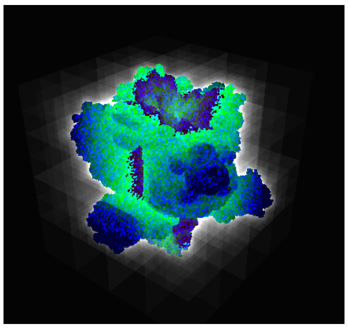
The test data was created using the voxel modelling software MagicaVoxel [32] alongside a few extra utilities and extensions for generating geometry and brush functionality created by Lachlan McDonald [52]. It is the opinion of the author of this thesis that the test data provided for the evaluation is the most likely type of data that will take most effective advantage of both the colour compression and spatiotemporal coherency. The author also believes that the test data represents a closer approximation to the type of data that could be applied in a voxel video game as well, i.e. animation that is not entirely arbitrary nor random and has a continuity between each frame [35, 36, 38, 39]. For the compression schemes used, exploiting spatiotemporal coherency in completely random and non-continuous test data would be very difficult to achieve effectively, while the colour compression most likely would result in poor quality. Therefore such data is considered by the author not to be suitable in order to provide a conclusive answer to the research question. Nonetheless, the test data provided for the evaluation was chosen and created in order to evaluate the effectiveness of the compression schemes in various situations and cases, such as density, varying types of movement and even colour changes. See Figure 4.13 for all the test data used for the evaluation.

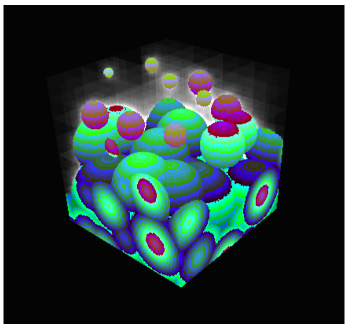
The animations of each test data is diverse in order to test various cases of how animation may be applied to animated voxel data. The colour also varies greatly: while it may appear some only have a few colours, any test data that appears to have some kind of shading is actually just colour attribute. The following is a brief description of each test data:

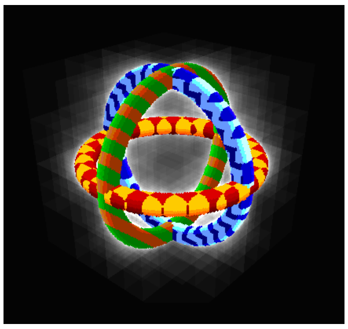
- BULB is a mandelbulb that each time step becomes larger and morphs into a more spherical-like shape, while all of it's colours remain consistent between each frame.
- ORBS is a collection of spherical objects that move and merge into each other and slowly over time change colour.
- BALL is a red checkered ball that is made to appear bouncing and rotating. This one is inspired by the "Boing" Ball demo originally created by R.J. Mical and Dale Luck [40].
- RINGS are three intersecting rings with different patterns that rotate in various angles.
- CUBE is a greeble-shaped checkered pattern cube with that contracts and then expands, appearing to mimic a heart pulse.
- CLOCK is an hourglass that has sand pouring down when turned.
- Finally, HORSE and DEER appear as running animals as the names suggests. As previously mentioned, these were created by ephtracy [43].

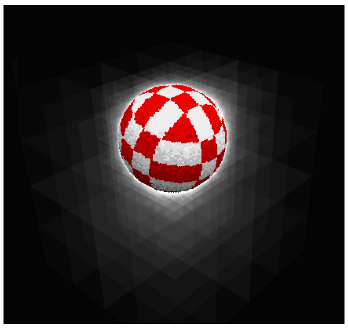
4.3.1.1 Quality Comparison

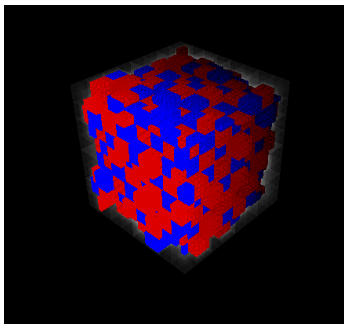
The quality of the compressed data is important to be at an acceptable level where the data appears as similar as possible to the original data, while requiring less memory usage. Therefore, in order to see which combination of methods produces the most memory effective result with the highest quality possible there will be a need to do a quality comparison. The quality comparison will be done by comparing Peak signal-to-noise ratio values (PSNR), a common approach to evaluate and measure the quality when reconstructing lossy compressed image data [31]. PSNR value represents how much noise has affected the quality and fidelity of the original data. Values between 30 and 50 dB (decibel) are typical values for PSNR in lossy image data. Higher value means it is better and values that go over 40 dB are considered to be very good. Values that are 20 dB or below are within the unacceptable quality range [31]. By calculating a PSNR value for each test data using all six permutations of the HSV colour sorting scheme, it becomes possible to compare which permutation is the most successful in improving the quality of compressed colours.

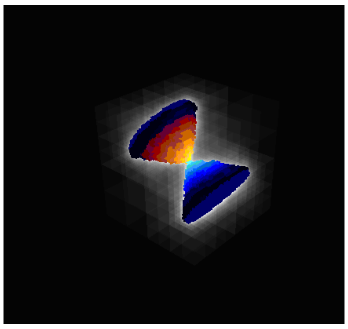

BULB: 128^3,11 frames.

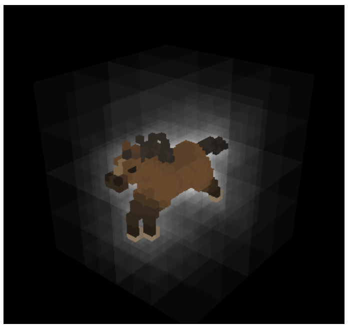

ORBS: 64^3, 21 frames.


RINGS: 128^3,18 frames.


BALL: 128^3, 19 frames.


CUBE: 64^3,15 frames.


CLOCK: 64^3,13 frames.


HORSE: 32^3, 4 frames.

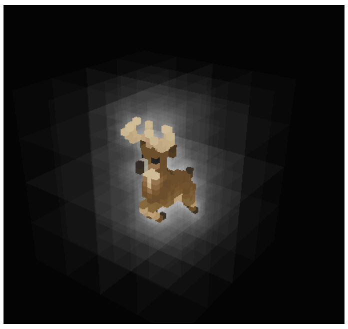

DEER: 32^3, 4 frames.

Figure 4.13: The test data used for evaluation. Each data is rendered using the modified SVO data structure Hyper Octree. Except for HORSE and DEER that was originally created by ephtracy [43], all other data was created by the author with the voxel modelling software MagicaVoxel [32].

4.3.1.2 Memory Consumption Analysis

A memory usage analysis will be performed of how much memory is used from the final output after all compression procedure have been applied to the animated voxel data. In this case, the memory usage refers to how much memory is used to store geometry, colours and other necessary auxiliary data structures, such as the aforementioned array for time step offsets. Measurement will be done during the construction and compression process. This will be done for both regular voxel grids, compressed grids, sparse voxel octree and Hyper Octree.

5

Results and Discussion

This section presents the result and discussion concerning the evaluation performed as described in Section 4.3 on the test data that can be seen Figure 4.13. In Tables 5.1-5.4, there are red and green highlights in certain cells: these indicate the best and worst case for each test data depending on the context. The evaluation and the execution of the methods were performed on a NVIDIA GeForce RTX 3060 GPU and AMD Ryzen 7 5800H CPU.

5.1 Quality

Table 5.1: Quality Measurement (in dB)

Data	**HSV**	**VHS**	**SVH**	**VSH**	**SHV**	**HVS**
BULB	18.1822	20.7589	23.0819	18.095	20.0038	19.7372
ORBS	18.1169	22.0709	26.2958	21.9395	19.8364	17.8267
BALL	18.243	33.9142	18.243	33.9142	18.243	33.9142
RINGS	18.0617	26.0889	29.7724	33.5734	19.4854	19.8031
CUBE	20.3879	37.2228	37.2228	37.2228	20.3879	20.3879
CLOCK	19.8702	30.9893	36.2239	36.7487	19.6886	21.7318
DEER	38.8142	38.509	36.8928	35.7898	38.8322	38.5063
HORSE	39.6377	39.5039	39.5071	39.5149	39.6177	39.5039
Average	23.9143	31.1322	30.9045	32.0998	24.5112	26.4264

The result from the quality evaluation can be seen in Table 5.1. See Figure A.9 for how each test data appears when DXT1 is applied with no sorting. Also see Figure A.10 for a visual comparison of each permutation. The quality measurement was performed independently of the data structures used to store the voxel data. This was possible since the colour data was decoupled from the geometry, which allows the colours be stored in a separate colour array that is then retrieved from through colour indexing. Since indexing is used for both grids and octrees to retrieve the colour data in their uncompressed formats, it then becomes possible to do the same with DXT1 blocks in the compressed formats.

The result shows that the permutations VSH and VHS provide on average the highest PSNR values, while HSV and SHV provides the lowest PSNR values on average. Incidentally, all of these permutations are in fact the complete opposite of each other, completely mirrored in the order of the sorting. BULB and ORBS only produced quality outcomes that were within the unacceptable range with all permutations. The rest: RINGS, CUBE, BALL and CLOCK produced good quality outcome within permutations that used the brightness (V) component first, while all other ones produced outcomes within the unacceptable range.

This suggests that on average for any given voxel data, if sorting by HSV the permutations using the component value or brightness (V) first will lead to higher quality outcome of the compressed voxel colour data than with any other permutation. This may indicate that depending on which component is used first for sorting will have an impact on the quality outcome. The quality may also be affected by which proceeding components are used for the sorting, where it is very clear that a combination of saturation (S) and hue (H) give the highest result in both VSH and VHS. This does not necessary correspond with the lowest PSNR values on average however, where HSV and SHV have very different components for the first sorting, but indeed have the same last component of brightness (V). This suggest that sorting by saturation and hue and then brightness will give the lowest quality outcome on average.

In Figure A.10, this can be seen clearly how indeed permutations VSH gives the closest approximation to the quality of the uncompressed format. This is not necessarily true for VHS however, where there actually is a bit more loss in quality despite the higher PSNR value with some of the test data. A possible explanation why the order of the components in the HSV sorting causes the reordering of the colours to mostly improve the quality outcome is because of the amount of similar colours and what type of colours are used in the test data. What is meant with 'type of colours' is the characteristics of a color, i.e. where does the colour lie on the colourspace and how intensive and what brightness does the colour have. Which of these characteristics is the most pronounced in a set of colours is the deciding factor to how the colours will be reordered and how this reordering was done affects the final quality output. For example, HORSE produces consistently very high values across all permutations, and DEER also produces relatively high values as well that only varies slightly with a few of the permutations and slight loss in quality such as with SVH and VSH. Even here the slight difference in the values of DEER compared to HORSE is because of the same reason. On the one hand, HORSE uses only a set of colours are very closely related or stored in the RGB colour space: a set of varying brownish and dark colours. On the other hand, DEER uses a set of lighter brown, yellowish colours along with notable light set of colours for its antlers and other parts. Since the DXT1 format stores colours in blocks and then interpolates the value in these blocks as the corresponding colours [25, 46], this means that the loss of quality will very much depend on which colours are stored in the block. In the case of HORSE these correspond to colours that all may be interpolated no matter which order they are stored. This means that the loss quality is practically non-existent no matter which permutation is used. This is also partially true for DEER, although the slight loss of quality with some permutations is because of the brighter colours which may not be as easily interpolated if stored together with more darker colours.

This is in contrast to the rest of the test data, which have many different varying colours in both brightness (V), saturation (S) and hue (H). The quality seems to be improved depending on which of the components of HSV colourspace is most present in the test data. In cases such as BULB and ORBS seems to improve the quality the most using saturation (S) first and foremost, even if it is only a small improvement. In contrast the rest of the test data appears to be best suited for sorting after brightness (V) first suggesting that brightness is the most common decisive component on average. Therefore, the conclusion can be drawn that the brightness component gives the best quality outcome using HSV sorting for DXT1 compression format because it is the most common, unifying component.

5.2 Memory Consumption

Table 5.2: Total Memory Consumption (in MB)

Data	Voxel Grid	Compressed Grid	SVO	Hyper Octree
BULB	92.273	46.162	23.591	12.056
ORBS	22.020	11.021	25.392	12.887
BALL	159.384	79.694	23.4	8.934
RINGS	150.995	75.50	21.051	7.522
CUBE	15.729	7.865	23.330	6.964
CLOCK	13.632	6.816	3.149	1.093
DEER	0.524	0.262	0.014	0.006
HORSE	0.524	0.262	0.030	0.014

The result from the memory usage analysis can be seen in Table 5.2. The difference between the voxel grid and the compressed grid is that the Compressed Grid is a voxel grid that utilizes DXT1 compression. This compression is the same for both grids and octrees. Also, SVO in this case means a regular sparse voxel octree that does not exploit spatiotemporal coherency nor compresses colours using DXT1, instead only utilizes a colour array that are indexed to from the leaf nodes.

Overall, the Hyper Octree was able to achieve the greatest compression ratio compared to Voxel Grid, Compressed Grid and SVO. This is attributed to the utilization and amalgamation of sparseness, spatiotemporal coherency and DXT1 colour compression in the Hyper Octree resulting in considerable reduction in memory consumption on average when compared to the rest. Also, reducing the DXT1 down to only unique blocks made it possible to reduce the memory consumption even further, resulting in orders of magnitude less memory consumption for DXT1 blocks. This can be seen in Table 5.3, where it can be seen that the memory consumption of the original array of DXT1 blocks compared to the reduced DXT1 blocks is considerable improved. This also improved the memory consumption for the Compressed Grid that uses the same set of DXT1 blocks.

It may therefore be argued that the reduction of the DXT1 blocks was also done out of necessity: neither the Hyper Octree nor the Compressed Grid would have had sufficient amount space within 19-bits and 12-bits respectively to index such huge amount of blocks. In any case, such huge amount of memory consumption with the original DXT1 blocks would have not gained any value from the compression itself it the blocks where not reduced down to only the most unique ones.

Table 5.3: DXT1 Blocks Memory Consumption (in MB)

Data	**Original DXT1**	**Reduced DXT1**
BULB	8.389	0.0244
ORBS	8.389	0.01072
BALL	4.194	0.002016
RINGS	4.194	0.00264
CUBE	4.194	0.000264
CLOCK	0.524	0.0004
DEER	0.002048	0.000392
HORSE	0.008192	0.00024

However, the octree-based data structures can only achieve greater compression ratio when sparseness is utilized in the voxel data. This can be clearly observed when comparing dense voxel data: the regular Voxel Grid that was able to achieve a more efficient memory consumption compared to the SVO in the case of CUBE and ORBS. This is also true with the Compressed Grid, which achieved slightly better compression compared to SVO with CUBE and even the Hyper Octree in the case of ORBS. But it must be stated that this is a quite expected outcome because of the nature of dense voxel data and voxel grids. Voxel grids are inherently dense and have a memory footprint that is directly proportional to its volume. In contrast, compact data structures using an octree only a fraction of the entire volume stored in memory, representing only the occupied space of the data within this entire volume [7, 3, 2]. From this it can be understood and concluded that any animated sequence that contains dense voxel data will be more memory efficient if stored in a dense data structure such as a voxel grid rather than octree. If dense data is stored in a sparse and compact data structure it will only increase the memory consumption and eventually its size will exceed the memory of dense voxel grid if the same data was stored in both.

This can be observed in the Table 5.2, where once again this holds true for both ORBS and CUBE when compared to SVO. The Hyper Octree is overall able to reduce much of the memory consumption because of the removal of redundant nodes and the use of DXT1 blocks. Even so, the Hyper Octree could not reduce down below the memory usage of the Compressed Grid in the case of ORBS, which is a dense type of data. Therefore the previously stated conclusion holds.

Table 5.4: Frame Node Count

	First Frame		Average Consecutive		Total	
	SVO	HO	SVO	HO	SVO	HO
BULB	398529	215500	499934	253853	5897812	3007888
ORBS	290256	155538	288463	145879	6347993	3219010
BALL	307099	164675	291729	108851	5849962	2232857
RINGS	290841	157401	276214	95690	5262706	1879821
CUBE	535969	286709	353104	96947	5832544	1740921
CLOCK	60433	32748	55898	18497	787123	273209
DEER	884	529	657	275	3513	1632
HORSE	1872	1064	1423	619	7567	3543

5.2.1 Node count

The result from the node count for SVOs and Hyper Octrees can be seen in Table 5.4. This evaluation was influenced by Kämpe et al.'s approach of a similar nature [3]. This comparison looks at how many nodes were able to be compressed for the first frame, the average of the consecutive frames and the total amount of nodes. This comparison does not include regular Voxel Grids nor Compressed Grids as these data structure cannot exploit spatiotemporal coherence and would therefore have no purpose in this comparison, other than to become a somewhat contrived comparison. See Figures A.1 - A.8 for a visual comparison of how much each test data is able to exploit spatiotemporal coherency.

The result shows that the Hyper Octree was able to reduce the total amount of nodes up to a factor of 2 for the majority of the test data. It was even capable of reducing the total amount of nodes with a factor of 3 in the case of CUBE. Out of all the test data BULB reduced by the lowest amount of total nodes, just below a factor of 2. For average consecutive frames, the majority of the test data were able to reduce consecutive frames to have less than the amount of the first frame, also by a factor of 2. Some of the test data were able to reduce a bit more, in the case of CUBE and CLOCK reducing by a factor of 3, while RINGS and BALL reduced by a factor of just below 3. BULB reduced by the least amount of nodes, by a factor of just below 2. Finally, for the first frame none of the test data were able to reduce the amount of nodes higher than by a factor just below 2, with DEER reducing the least amount and CUBE reducing the highest amount. Overall, the Hyper Octree was able to reduce the total amount of nodes considerably when compared to the regular SVO.

The reason for the factor of reduction in nodes depends on two aspects: the amount of nodes necessary for storing colours and how much spatiotemporal coherency can be exploited between each frame. For the first frame, the reduction factor depends much on how and how many colours are stored in memory. Since the regular SVO only uses 16-bits for storing indices, it would not be possible to directly store each colours at the leaf node. Instead, only an index is stored which points to the true location of the colour data. Although this implementation used a colour array, these colours might as well have been stored in the octree directly. This would have meant there would have been double the amount of leaf nodes: one for storing an implicit index and the colour data itself. Even in the cause of a separate colour array, it still means there is double the amount of leaf data. This is in contrast to the Hyper Octree: First, the Hyper Octree stores only a block descriptor in order to fetch the corresponding DXT1 block. This means that the colour array is completely discarded. Also, the DXT1 blocks are also reduced by orders of magnitudes reducing the amount of leaf nodes required. We can consider the the DXT1 blocks as the leaf nodes. In the end, this reduces the number of required nodes in the first frame significantly.

This difference in by how high the factor of reducing the nodes in each frame most likely depends on how much spatiotemporal coherency exists in the data. This depends on how likely it is to find similarities between each frame. In the case of CUBE, we observe that it was able to gain the most out of all the test data using spatiotemporal coherence. This is because the locality of the voxels is very consistent and same between each frame, since the data is dense and much of it remains mostly the same. In contrast BULB reduced the least in consecutive frames, mostly likely because the data changes drastically between each frame and the locality of the colours. The diversity of the colours may also be deciding factor.

For the total amount of frames and the consecutive frames, the factor depends on the amount of spatiotemporal coherency that exists between each frame. In Table 5.2.1, this can be observed in the average node count in consecutive where in some of the test data such as CUBE, RINGS, BALL, DEER and CLOCK there is more spatiotemporal coherency than there is in BULB, ORBS, and HORSE. An example could be BULB, which is the only test data that required more nodes in the consecutive frames compared to the first frame when observing the rest of the test data. This implies that BULB has the least amount of spatiotemporal coherency. Another example could be ORBS, that required almost the same amount of nodes for the first frame and the consecutive frames, suggesting a similar implication. When observing the coherency as shown visually in Figures A.1 - A.8, it becomes a bit more obvious why this is so, where BULB and ORBS indeed show to have less spatiotemporal coherency between each frame when compared to the rest of the test data. This may also explain why the memory consumption is still relatively high for both BULB and ORBS when using the Hyper Octree as seen in Table 5.2 despite having reduced memory consumption considerably when compared to the other data structures. So, the total amount of nodes in the Hyper Octree is dependent on how much spatiotemporal coherency can be achieved in the consecutive frames, which in of itself depends much on the data in question.

5.3 Limitations

The current approach used in order to exploit spatiotemporal coherency in animated voxel data is limited to only exploiting coherency between each frame. While this approach was first suggested as a compression scheme by Ma and Shen [12], as well as a similar method used by Kämpe et al. [3], it is only able to achieve spatiotemporal coherency if there is similarities between each consecutive time step. This limitation also applies to their approaches [12, 3] as this approach was based upon their suggestions. A possible improvement in order to achieve a possibly greater compression ratio would be to find and exploit spatiotemporal coherency across several consecutive frames. That is, instead of only comparing two consecutive frames, it could be possible to check for coherency in every frame for each frame in order to find similarities in the subtrees with each iteration. This way, it would be possible to retrieve voxel data from several time steps or possibly keep a spatiotemporal continuity in the data much longer and as a result could achieve a much greater compression ratio. Naturally, this would create some computational overhead in the creation process, especially at larger voxel resolutions, because such comparisons would be very detailed and precise but also time consuming and memory intensive. Nonetheless, this may prove to be useful to explore further.

The HSV sorting was able to improve the quality of the DXT compression format and especially so using permutations where the first component in the sorting was brightness (V). The permutation VSH proved to be the best permutation on average for reordering colours for ordering similar colours to be in close proximity to each other. Lee et al. [19] concludes that sorting colours, or reordering of colours, to maximize the locality of similar colours appears to improve the quality of the data. Their conclusion appears to be in conjunction with the conclusion of this project, even if this project did not use any Morton ordering and instead simply remapped indices to the corresponding colour. Despite this, using permutations where (V) is the first component is not always guaranteed to reorder the colours effectively in order to improve the quality. This seems to be true for test data with much variety in saturation and hue. As seen in Table 5.4 and in Figure A.10, Although the majority of the test did improve the quality using the permutation VSH, BULB and ORBS where not able to improve significantly while the rest where visually satisfactory. This may be because of the huge amount of colours used in both BULB and ORBS vary in mostly hue and saturation and not so much in brightness. The limitation of this approach is that there is no method for deciding which permutation should be used and therefore will lead to inadvertent results. A possible and simple solution would be to use the best corresponding permutation for each test data. Otherwise, it may be best to default using the VSH as it was proven to give the best result on average.

6 Conclusion

6.1 Conclusion

The aim of this project was to compress animated voxel data using a combination of different approaches that could exploit the characteristics of frame-based animation sequences. These where sparseness with sparse voxel octree, spatiotemporal coherency by finding similar subtrees between frames and colour compression using DXT1 format. A sorting scheme using the colourspace HSV to reorder colours before the compression was also implemented in order to improve the quality of the compressed colours. Redundant DXT1 blocks are also reduced down to only the unique blocks to further compress the data. The amalgamation of these methods was given the name *Hyper Octree*, a sparse voxel octree-based data structure that is able to exploit spatiotemporal coherency and utilize compressed colours.

The result shows that this method is able to reduce memory consumption by at least a factor of 2 and even up to a factor of 3 in some special cases when there were a high degree of spatiotemporal coherency in the data. Quality of compressed colours were able to be improved by sorting colours using HSV values. In particular, the permutation VSH proved to give the best average result. However, this was not always the case since some test data would improve much better using other permutations, depending if saturation or hue was the dominant characteristics in set of colours. In any case, the sorting scheme proved a possibility to improve the quality of lossy compressed colours in voxel data.

The original research question as it was written in Section 1.2, was as follows:

- *What is the best method for lossy compression of animated voxel data for sequence-based animation that keeps similar fidelity and quality?*

In conclusion, the most efficient method is to utilize sparseness, exploit spatiotemporal coherency, colour compression and colour sorting. More specifically, one possible and effective implementation is to use a sparse voxel octree capable spatiotemporal coherency (re-use data from a previous time step) and compressing colours using the DXT1 compression format. Also, sorting the colours using HSV values before compression will reorder all colours to be in close proximity to each other in the HSV colourspace. This may lead to improved quality of the compressed colours as long as the correct permutation is used, since the output will depend much on the type of colours used in the voxel data. Overall, the permutation VSH proved to be give the most improved quality out of all six permutation on average and could serve as a good default. In the end, this gives a lossy compression for animated voxel data that is able to mostly keep a high degree of quality while still reaching a high degree of compression as well.

6.2 Future Work

The author considers that the methods implemented were neglected to be optimized to be fast and efficient concerning time. This was neglected as it this was not the aim of the project itself, but unfortunately this lead to considerable amount of time needed in order to create and generate the animation sequences in the Hyper Octree format. One reason for this comparing all possible subtree records between two consecutive octrees. Although the method was functionally correct and produced a correct output, it caused an considerable overhead that would lead the procedure to take up to several hours if the voxel data was of a higher voxel resolution. For much larger resolution this method would not be suitable and would benefit from finding a more optimized way in order to compare large amount of subtrees without causing an overhead if possible.

This project was limited to only considering using the DXT1 compression format for colour compression. This was mostly because of time constraints, and while the DXT1 format together with the HSV sorting proved to be moderately successful in retaining good quality and relatively high compression rate, other types of compression methods should be considered. These may possible be better for retaining high quality while also being able to achieve high compression ratio. It would also be interesting to attempt another way of sorting colours together with DXT1 or another compression technique, to see how different types of components may affect the quality.

6.3 Ethical Considerations

The ethical considerations for this project were practically non-existent. The outcome from this thesis, the test data produced for the evaluation and the methods created for the purpose of answering the research question are unlikely to have any greater impact on any ethical aspects or concerns. However, for the project itself, the author decided against using or creating test data in the likeness of any artifact that have any religious or cultural significance. Misrepresentation of such artifacts is something to be avoided as it may be considered offensive and sensitive. Therefore it was decided that such test data would not be used for the evaluation.

Bibliography

[1] Amanatides, J. and Woo, A. (1987). "A Fast Voxel Traversal Algorithm for Ray Tracing," Proc. Eurographics 87 Conf., pp. 3-10, 1987.

[2] Aleksandrov, M., Zlatanova, S. and Heslop, D. J. (2021). "Voxelisation Algorithms and Data Structures: A Review". Sensors (Basel, Switzerland), 21(24), 8241. https://doi.org/10.3390/s21248241.

[3] Kämpe, V., Rasmuson, S., Billeter, M., Sintorn, E. and Assarsson, U. (2016). "Exploiting coherence in time-varying voxel data". I3D '16: Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. pp. 1521. https://doi.org/10.1145/2856400.2856413.

[4] Kämpe, V., Sintorn, E. and Assarsson, U. (2013). "High Resolution Sparse Voxel DAGs". ACM Transactions on Graphics Volume 32, Issue 4, July 2013. Article No.: 101. pp 1-13. https://doi.org/10.1145/2856400.2856413.

[5] Dolonius, D., Sintorn, E., Kämpe, V., and Assarsson, U. (2019). "Compressing Color Data for Voxelized Surface Geometry". in IEEE Transactions on Visualization and Computer Graphics, vol. 25, no. 2, pp. 1270-1282, 1 Feb. 2019, doi: 10.1109/TVCG.2017.2741480.

[6] Dado, B., Kol, T.R., Bauszat, P., Thiery, J.M. and Eisemann, E. (2016). "Geometry and attribute compression for voxel scenes". In Computer Graphics Forum (Vol. 35, No. 2, pp. 397-407).

[7] Laine, S. and Karras, T. (2011). "Efficient Sparse Voxel Octrees". in IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 8, pp. 1048-1059, doi: 10.1109/TVCG.2010.240.

[8] Hickson, S., Birchfield, S., Essa, I. and Christensen, H. (2014). "Efficient Hierarchical Graph-Based Segmentation of RGBD Videos". 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 344-351, doi: 10.1109/CVPR.2014.51.

[9] Espe, A. E., Gjermundnes, O. and Hendseth, S. (2019). "A Method for Rigid-Body Animation of Sparse Voxel Octrees for Use in Ray Tracing". Journal CoRR abs/1911.06001, November 2019. https://doi.org/10.48550/arXiv.1911.06001.

[10] Careil, V., Billeter, M. and Eisemann, E. (2020). "Interactively Modifying Compressed Sparse Voxel Representations". Computer Graphics Forum (Proc. Eurographics), 2020. http://graphics.tudelft.nl/Publications-new/2020/CBE2.

[11] Rodríguez, M.B., Gobbetti, E., Guitián, J.A.I., Makhinya, M., Marton, F., Pajarola, R. and Suter, K.S. (2014). S"tate-of-the-art in compressed GPU-based direct volume rendering". Comput. Graph. Forum 2014, 33, 77100.

[12] Ma, K.-L. and Shen, H.-W. (2000). "Compression and Accelerated Rendering of Time-Varying Volume Data". Proc. 2000 Int'l Computer Symp. - Workshop on Computer Graphics and Virtual Reality, pp. 82-89, 2000.

[13] Sattler, M., Sarlette, R., and Klein, R. (2005). "Simple and efficient compression of animation sequences". In Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation (pp. 209-217).

[14] Han, S.-R., Yamasaki, T. and Aizawa, K. (2007). "Time-varying mesh compression using an extended block matching algorithm". IEEE Transactions on Circuits and Systems for Video Technology 17, 11, 1506-1518.

[15] Zhang, J., and Owen, C.B. (2007). "Octree-based animated geometry compression". Computers and Graphics 31, 3, 463–479.

[16] Mado, B., Chovancová, E., Chovanec, M. and Ádám, N. CSVO: "Clustered Sparse Voxel OctreesA Hierarchical Data Structure for Geometry Representation of Voxelized 3D Scenes". Symmetry 2022, 14, 2114. https://doi.org/10.3390/sym14102114

[17] Villanueva, A.J., Marton, F. and Gobbetti, E. (2017). "Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes". Journal of Computer Graphics Techniques Vol, 6(2).

[18] Vokorokos, L., Mado, B. and Bilanová, Z. (2020). "PSVDAG: Compact Voxelized Representation of 3D Scenes Using Pointerless Sparse Voxel Directed Acyclic Graphs". Computing and Informatics, 39(3).

[19] Lee, K., Yi, J., Lee, Y., Choi, S. and Kim, Y.M. (2020), September. "GROOT: a real-time streaming system of high-fidelity volumetric videos". In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (pp. 1-14).

[20] I. Gargantini. "An Effective Way to Represent Octrees". (1982). Communications of the ACM, Volume 25 Issue 12, Dec. 1982, Pages 905-910

[21] Foley, James D., Andries van Dam, John F. Hughes and Steven K. Feiner (1990). "Spatial-partitioning representations, Surface detail". Computer Graphics: Principles and Practice. The Systems Programming Series. Addison-Wesley. ISBN 978-0-201-12110-0.

[22] Cohen-Or, D. and Kaufman, A. "Fundamentals of surface voxelization". Graph. Model. Image Process. 1995, 57, 453461.

[23] Gervautz, M. and Purgathofer, W. (1988). "A simple method for color quantization: Octree quantization". In New Trends in Computer Graphics: Proceedings of CG International88 (pp. 219-231). Springer Berlin Heidelberg.

[24] Mileff, P. and Dudra, J. (2019). "Simplified voxel based visualization". Production Systems and Information Engineering, 8, pp.5-18.

[25] Van Waveren, J.M.P. (2006). "Real-time DXT compression". May 20th, 2006.

[26] Iourcha, K.I., Nayak, K.S. and Hong, Z. System and Method for Fixed-rate Block-based Image Compression with Inferred Pixel Values, U.S. Patent 5956431A, Abbrev. October 2nd, 1997.

[27] Laybourne, K. (1998). "The Animation Book: A Complete Guide to Animated Filmmaking From Flip-Books to Sound Cartoons to 3-D Animation". New York: Three Rivers Press. ISBN 051-788602-2.

[28] Grossi, R. (1993). "On finding common subtrees". Theoretical Computer Science, 108(2), pp.345-356.

[29] Smith, A.R. (1978). "Color gamut transform pairs". ACM Siggraph Computer Graphics, 12(3), pp.12-19.

[30] Yu, S., Zhang, S., Wang, K., Xia, Y. and Zhang, H. (2017). "An efficient and fast GPU-based algorithm for visualizing large volume of 4D data from virtual heart simulations". Biomedical Signal Processing and Control, 35, pp.8-18.

[31] Bull, D.R. (2014). "Digital picture formats and representations". Communicating pictures, pp.99-132.

[32] Ephtracy. (2021). MagicaVoxel.

[33] Minddesk Software GmbH. (2016). Qubicle Voxel Editor.

[34] Ken Silverman. (2011). Slab6.

[35] Joseph, W. (2011). Voxatron. PC. Japan, Tokyo.

[36] Silicon Studio Co., Ltd. (2009). 3D Dot Game Heroes. Playstation 3. Japan, Tokyo.

[37] Trion Worlds. (2015). Trove. Windows, PlayStation 4, Xbox One, Nintendo Switch. U.S., Redwood City, California.

[38] Radiant Entertainment. (2018). Stonehearth. Windows, macOS. U.S., Los Altos, California.

[39] Picroma. (2019). Cube World. Windows. Germany.

[40] Swiss Cracking Association. "Amiga Workbench Demos". Swiss Cracking Association. https://www.sca.ch/amiga/guests/boing.html [Accessed 2023-05-10].

[41] Ephtracy. "MagickaVoxel-file-format-vox.txt". Github. https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox.txt [Accessed 2022-11-08].

[42] Ephtracy. "MagickaVoxel-file-format-vox-extension.txt". Github. https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox-extension.txt [Accessed 2022-11-08].

[43] Ephtracy. "MagicaVoxel". Github. http://ephtracy.github.io/ [Accessed 2023-05-27].

[44] Sean Barret. "stb_dxt.h". Github. https://github.com/nothings/stb/blob/master/stb_dxt.h [Accessed 2023-04-14].

[45] Benjamin Dobell. "S3TC DXT Decompression". Github. https://github.com/Benjamin-Dobell/s3tc-dxt-decompression [Accessed 2023-04-14].

[46] Khronos. "S3 Texture Compression". Khronos. https://www.khronos.org/opengl/wiki/S3_Texture_Compression [Accessed 2023-05-16].

[47] Khronos. "Texture". Khronos. https://www.khronos.org/opengl/wiki/Texture [Accessed 2023-05-16].

[48] FSDeveloper. "DXT compression explained". FSDeveloper. https://www.fsdeveloper.com/wiki/index.php/DXT_compression_explained [Accessed 2023-05-16].

[49] Inigo Quilez. "Distance functions". Inigo Quilez. https://iquilezles.org/articles/distfunctions/ [Accessed 2023-02-14].

[50] C++ Reference. "unique". C++ Reference. https://en.cppreference.com/w/cpp/algorithm/unique [Accessed 2023-05-19].

[51] C++ Reference. "sort". C++ Reference. https://en.cppreference.com/w/cpp /algorithm/sort [Accessed 2023-04-19].

[52] Lachlan McDonald. "magicavoxel-shaders". Github. https://github.com/lachlanmcdonald/magicavoxel-shaders [Accessed 2023-05-09].

A

Appendix 1

The Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7 and A.8 are a visual representations of exploiting spatiotemporal coherency. Each Figure represents the test data used in the evaluation and each one show how effectively spatiotemporal coherency is achieved. The magenta-coloured voxels indicate which voxels are redundant and are reconstructed by retrieving data from the previous time step. The sequence of each animation is laid out in chronological order.

Figure A.10 is a full compilation of all the quality outcomes from performing DXT1 colour compression and sorting the colours using HSV permutations. Observing the various test data in each corresponding column, it becomes clear that the quality outcome varies depending on which type of colours and how much variety there is with each colour, as this affects the reordering and subsequent compression output.

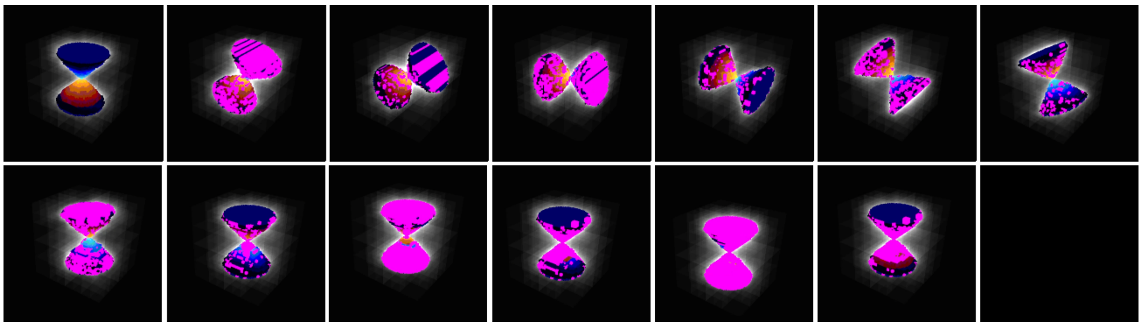


Figure A.1: Spatiotemporal coherency in CLOCK.

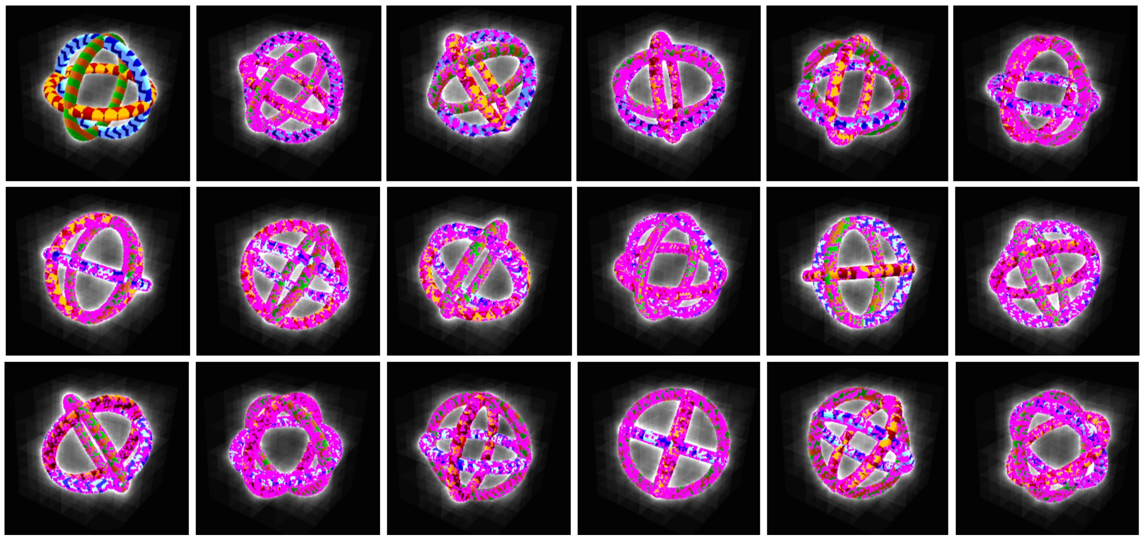


Figure A.2: Spatiotemporal coherency in RINGS.

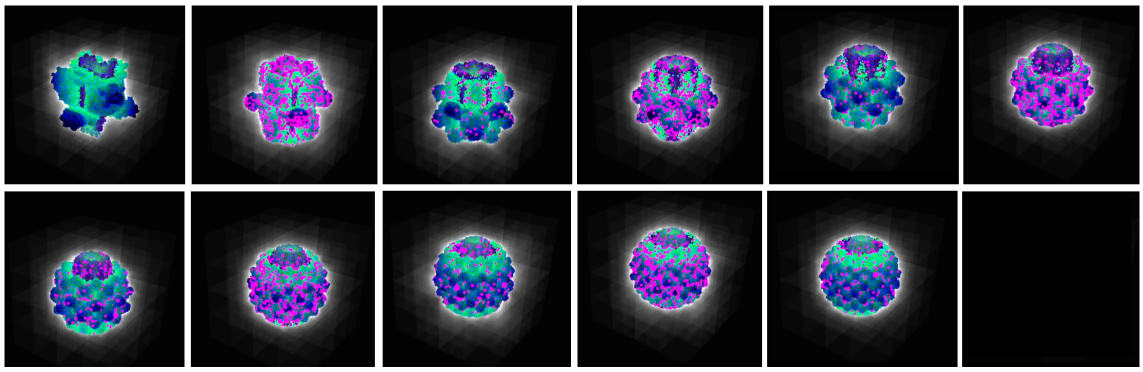


Figure A.3: Spatiotemporal coherency in BULB.

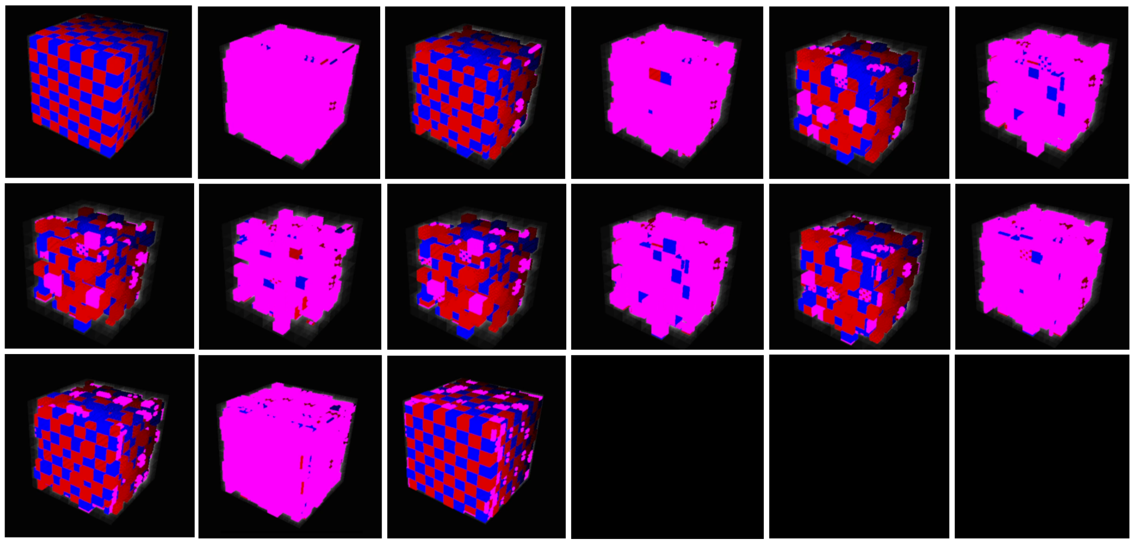


Figure A.4: Spatiotemporal coherency in CUBE.

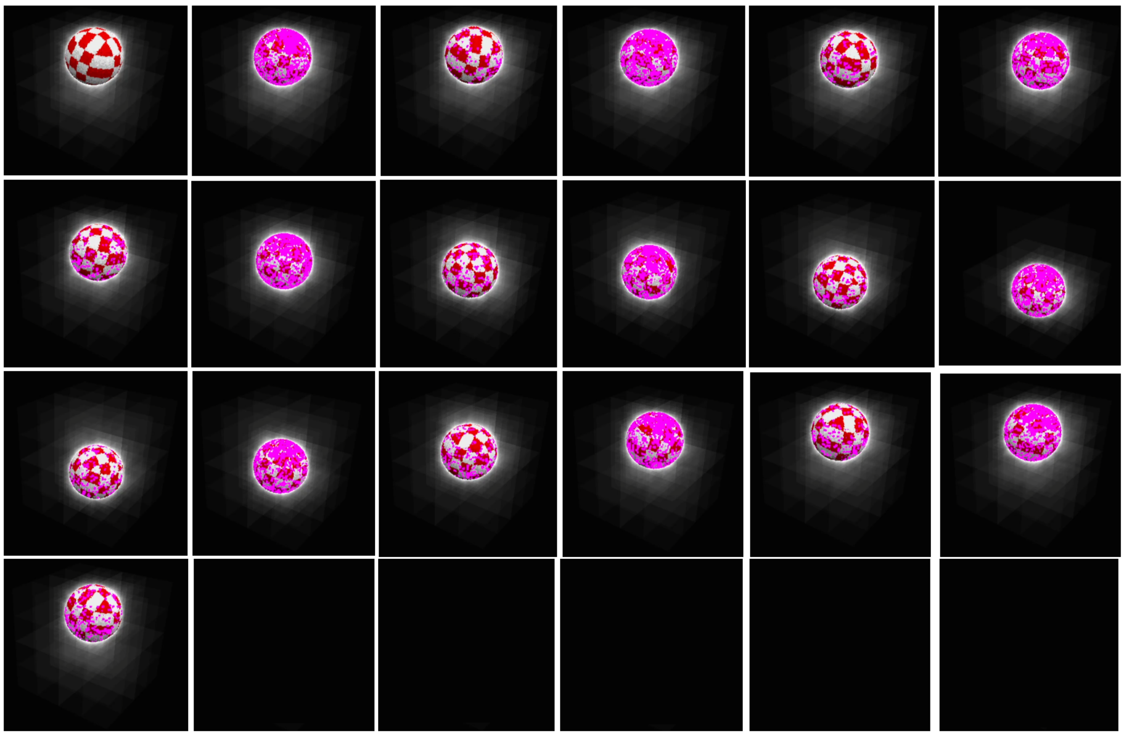


Figure A.5: Spatiotemporal coherency in BALL.

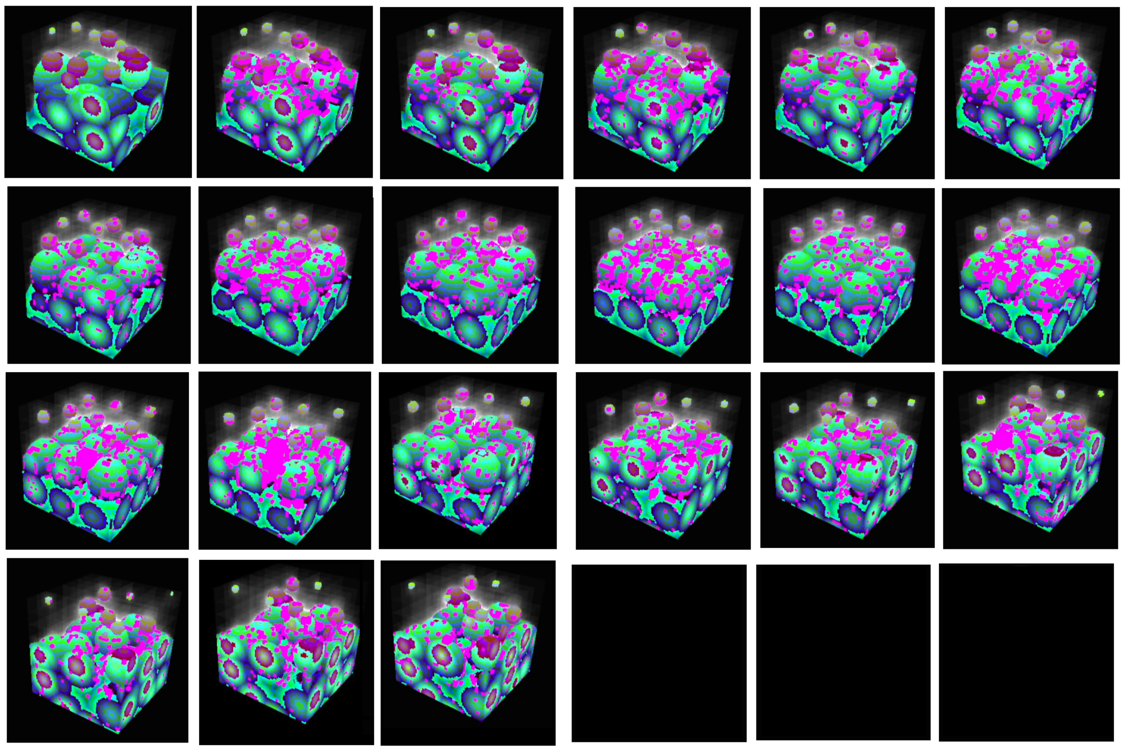


Figure A.6: Spatiotemporal coherency in ORBS.

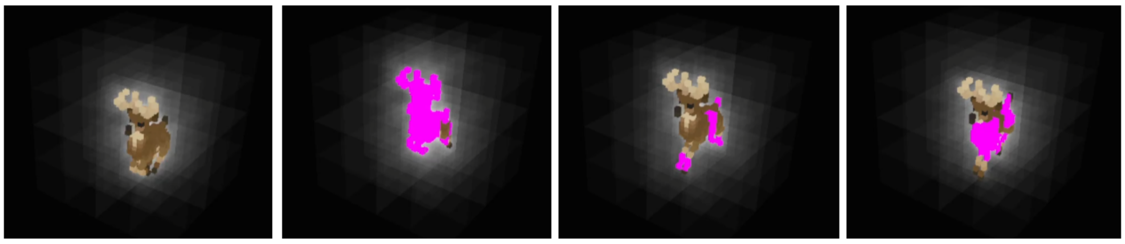


Figure A.7: Spatiotemporal coherency in DEER.

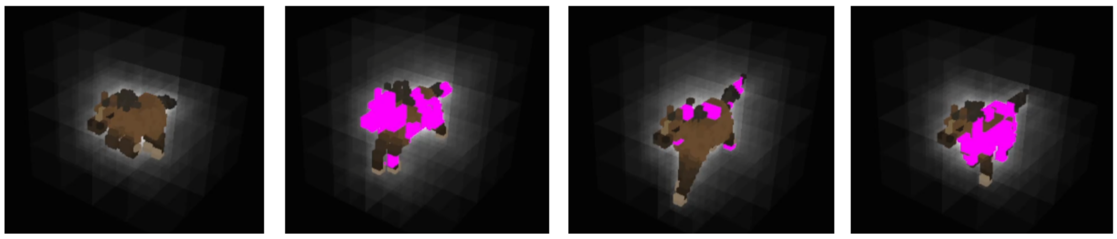


Figure A.8: Spatiotemporal coherency in HORSE.

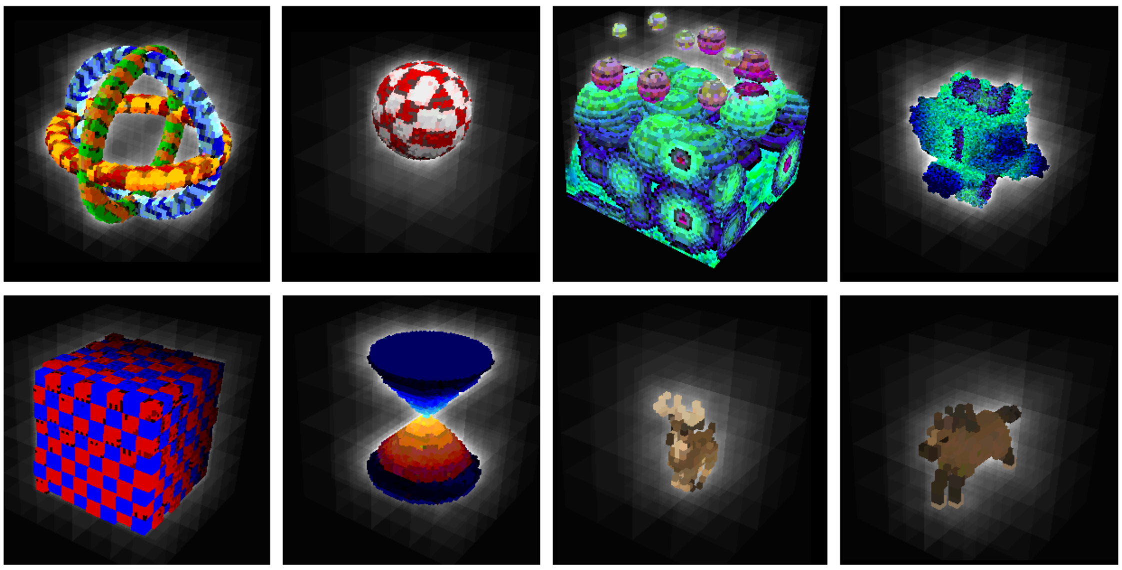


Figure A.9: How each test data appears when colours are compressed using DXT1 with no HSV sorting.

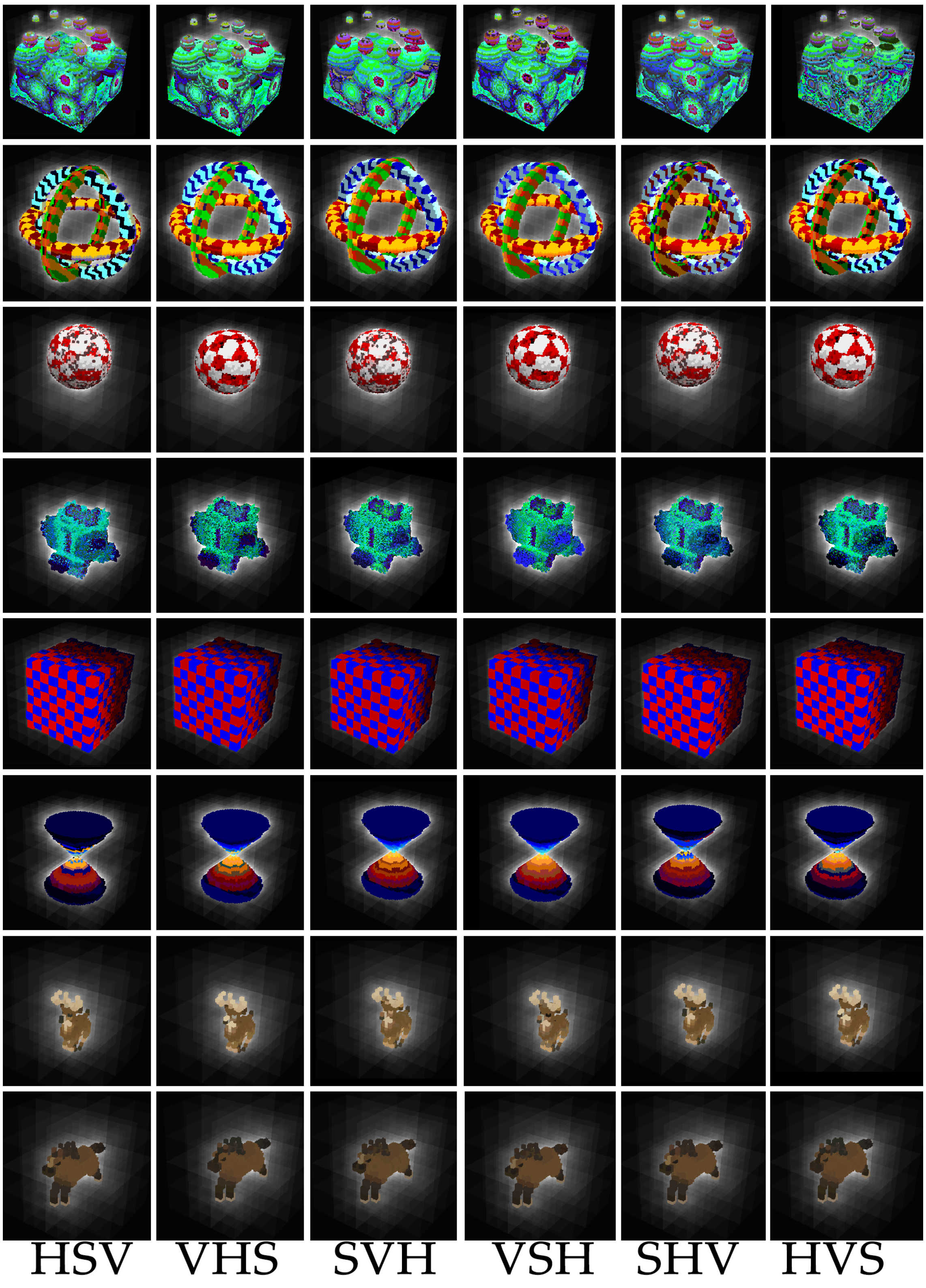


Figure A.10