Utrecht University

Real-time Ray tracing and Editing of Large Voxel Scenes

Thijs van Wingerden (4123565)
Supervisor: dr. ing. Jacco Bikker

ICA-4123565

June 25, 2015

Abstract

A novel approach is presented to render large voxel scenes in real-time. The approach differs from existing solutions in that a large emphasis is put on allowing the user to edit and stream large datasets. Previous solutions often use compression schemes involving hierarchical data layouts such as sparse voxel octrees that require some form of preprocessing, which prevents efficient editing. By keeping data in raw format we avoid having to preprocess the data making it directly editable. We allow for efficient storage of large empty spaces using a system of layered grids . Our results show our solution has competitive rendering performance and memory usage and allows fast and easy access to the data. Along with the raytracing algorithm and datastructure an occlusion based streaming system is presented. Our system also includes an easy interface for editing voxels. Furthermore, we support streaming from disk and basic level of detail. Combining these features we are able to render and edit potentially infinite landscapes while using a finite amount of memory.

Contents

1 Introduction

1.1 Triangles versus Voxels

Historically voxels have been a less popular alternative to triangles. Modern rasterization hardware does not directly support voxel rendering, and unlike triangles, voxels do not represent continous data. On top of this, substantially less research has gone into animation of voxel data as triangle mesh animation. However, advances in hardware and ever increasing geometry detail justify a reconsideration of voxels as the fundamental primitive for rendering and editing in interactive applications such as games.

The main reason triangles have been the prefered solution is their compact storage. Given the large amounts of memory available on modern systems voxel datasets can now be stored at sufficient resolutions to represent even the largest of environments. Since storage has become less of an issue voxels have recently become an attractive option to represent highly detailed geometry. Fine detail is stored directly, rather than in auxiliary data structures such as textures, normal maps and displacement maps. Triangles can represent similarly detailed data as voxels but become less efficient as the detail increases. The benefit of compact storage of triangles is lost when we no longer describe large surfaces using only a few triangles. This is most apparent once the number of triangles per pixel is less than one. Voxels on the other hand store high resolution data efficiently because of their fixed resolution nature. This data can easily be augmented with lower resolution versions of the same data, allowing for efficient level of detail operations and reduced memory requirements for distant geometry. Finally, the regular grid layout of voxel data allows for efficient streaming algorithms.

1.2 Voxel Applications

The primary benefit of voxels is their capability of representing complex volumetric data. Voxels are currently the best solution for representing subsurface data or editable geometry, which is useful in applications such as 3d sculpting tools or procedural world generators. Prodecural world generators using voxels do not suffer from the same limitations imposed by alternatives such as 2D heightmaps, it is possible to generate true 3D structures such as overhangs, bridges and caves.

Even when the geometry does not contain subsurface data voxels provide notable benefits. Ray tracing of voxels is efficient and can be done with little or no preprocessing. The voxel data layout is intuitive, and suitable for high detail as well as raw data from e.g. medical scans.

An important application of voxels is editable geometry. If subsurface data is stored editing becomes natural and easy. Removing a voxel will reveal the underlying geometry and not create artifacts such as holes. Normals and colors are stored directly with the voxels, rather than separately. For editing only two operations are needed; changing the color of a voxel and turning voxels on or off. Editing can be used to simulate large

scale animations, e.g. erosion or fluid propagation if done efficiently.

1.3 Context

Our system is designed with video games in mind. This means the system must run in real-time an not exceed the storage limits of modern consumer hardware. Realistic physically based rendering is not our goal, but we do not want to exclude the possibility. Our system must potentially support advanced features such as shadows, reflections or ambient occlusion. What we do focus on is allowing real-time interaction on a large scale with the environment. We want to be able to perform real-time modifications on the entire environment combined with advanced feature such as streaming or procedural content generation. Furthermore we want to be capable of handling worlds of virtually infinite size. The voxel resolution must be high enough to represent sufficient detail and the draw distance must be far enough to render large landscapes.

1.4 State of the Art

Datastructures have been developed which are capable of rendering and storing large voxel datasets. Yet most of them come with significant drawbacks. They are geared towards static scenery and do not allow editing and streaming, nor do they support additional shading parameters. Techniques such as sparse voxel octrees (SVOs) provide efficient storage and fast raytracing. Most voxel raytracing techniques today use some form of sparse voxel octrees. The most significant drawback of SVOs is their inherent static nature. If a voxel is changed the octree must be reconstructed which induces significant latency. Even if little preprocessing is necessary editing is still inefficient, especially when neighbouring data lookups are required. Efficient editing systems could be developed that minimize the number of reconstructions done in the octree but this would limit the user in freedom and ease of use. Modifying the datastructure to add, for example, more shading parameters is a complex task.

1.5 Discussion

As mentioned in the previous paragraph, most solutions store voxels in a way that hinders editing. These solutions were developed with converted triangle data and high quality lighting in mind. There seems to be a tradeoff between these advanced features and editability of the data.

A simple solution that allows for efficient editing is raw storage of voxel data, as used in the earliest voxel rendering algorithms. While editing and attribute lookups are fast the problems become apparent at high data resolutions. Ray tracing an enormous voxel grid will be too slow and take up large amounts of memory. Without a hierarchy large open areas can not be stored or ray traced efficiently. Obviously this is not the right solution for high resolution data sets used today.

1.6 Research Question

The goal of this project is to efficiently store and render large voxel datasets with potentially infinite view distance. Our primary aim is to render large landscapes, as these in particular benefit from the ability to represent complex shapes and large data sets. While, as mentioned before, there already exist some compact datastructures which are capable of rendering large voxel datasets. They all come at a cost, often related to to flexibility in editing, streaming and storing additional shading parameters; features which are important for games. The aim is thus to develop a scheme for efficiently storing, editing and rendering large voxel environments. Our research questions in more detail are:

- What is the maximum detail level that can be used within these constraints on consumer graphics hardware.
- Are there compression schemes for color and geometry that do not limit editing and rendering efficiency, and how do these compare to hierarchical voxel representation schemes.
- Can streaming be used the reduce the amount of in-core data.
- Compared to existing schemes, what are the consequences of the additional flexibility, both in terms of storage and performance.

1.7 Contribution

The algorithm presented in this thesis is suitable for many applications and allows editing and streaming while maintaining competitive performance. The algorithm does not use a sparse octree like most current similar solutions; instead it stores geometry in small brickmaps which contain raw data. An extensible framework is provided which supports advanced features such as streaming and editing. Compression, occlusion based streaming and level of detail are shown to signficantly reduce memory usage. A stackless ray tracing algorithm is presented which offers competitive performance. Memory usage is shown to be slightly higher than other solutions but remains competitive.

In summary the work presented in this paper shows that editability and flexibilty can be maintained at little cost in speed and memory usage. It shows it is possible to render, stream and edit large datasets in real-time. Advanced features such as occlusion based streaming and world scale simulations can be done while maintaining full flexibilty for the user.

2 Related Work

Voxels received considerable interest in recent years, both in academia and the industry. For this research, we focus on existing work that could store large datasets and render them in real-time. Some of these focus on compression or efficient rendering. Very little has been done on the subject of editing and little on streaming. We also discuss work on compression, not necessarily applied to voxels, where this is relevant to our research.

2.1 Historical Overview

The first research done on ray tracing voxels was in 1987 by John Amanatides and Andrew Woo [1]. They provide a simple algorithm to traverse a ray through a voxel grid. They store the voxels in an array and use no hierarchical structure. The voxels however do not describe the actual geometry as they store triangles within the voxels. In their benchmarks ray tracing a scene with a 512x512 resolution took between 20 minutes and an hour to render, therefore at that time being far from real-time. In subsequent years little research on the subject of voxels has been done, until 2005 when there was another rise in interest in voxels. Since then more efficient ways of storing and ray tracing voxel grids have been presented.

The Far Voxels [11] framework developed in 2005 extended the idea of the previous paper by developing more sophisticated methods of dividing triangles in voxels while also keeping a fixed number of triangles. Their algorithm runs in real-time from 10 to 73 fps and can render large datasets with over 50M primitives. They also provide a method for streaming which progressively refines the detail of a dataset.

Another paper by Afra [10] proposes a technique to render extremely large models by using a voxel hierarchy framework. The data is preprocessed and a kd-tree datastructure is produced which stores the primitives and LOD voxels. The datastructure is ray traced on the CPU and provides one bounce global illumination. Another key feature is the asynchronous streaming of the datastructure from disk, which means even with low memory large models can still be rendered. They can render 337 million triangles at 37 fps using only 800MB memory. With global illumination they can render the same dataset at 7 fps using 1000MB memory.

An alternative approach is proposed in a paper by Forstmann [9] which RLE encodes voxels in an array on the up(Y) axis. The scene is raycasted in planes perpendicular to the x-z plane and passing through the viewpoint. All RLE elements intersecting a plane are then rasterized. The RLE elements are rendered at a level of detail based on distance to the viewpoint. Finally smoothing and anti-aliasing of the voxels is performed.

Several algorithms have also been proposed for ray tracing raw voxels not containing any other primitives. A good example is the paper on Efficient Sparse Voxel Octrees by Laine and Karras [13]. They provide an efficient algorithm for ray tracing a sparse voxel octree on the

GPU.

A paper by Crassin et al. [7] presents an algorithm that calculates indirect illumination in real-time with up to two bounces. The algorithm uses a sparse voxel octree generated in real-time from an input mesh which can support animations. Visibility and incoming light is calculated by approximate cone tracing. It supports ambient occlusion, diffuse and glossy reflections. The algorithm does not require heavy precomputations and gives real-time performance.

The work of the previously mentioned paper was based on the PhD thesis by Crassin [6], in which he developed a framework that can render several billion voxels on the GPU. The scheme makes use of paging and caching to dynamically load data onto the GPU, based on visibility information gathered during rendering in a feedback loop. This is similar to the streaming approach used in our system. The voxels are pre-filtered by integrating several shading parameters over the area of a voxel which has the size of a pixel. Additionally by using cone tracing, similar to the followup paper by Crassin et al. [7], they achieve real-time indirect lighting. The datastructure used is a combination of a sparse voxel octrees and brickmaps.

An alternative to regular sized grids is proposed by Costa and Pereira [5]. They also store primitives in a voxel grid, in their case from scanned models. However they propose a rectilinear grid instead which does not have a fixed size in all dimensions. They measured it can be up to 79% more efficient at ray tracing irregular scenes than a regular grid.

2.2 Smoothed Voxels

Both rasterization and ray tracing techniques exist to render voxels. Depending on their application, voxels can contain other geometry such as triangles, be smoothed or triangulated. When converted from triangle data the blocky nature of voxels is often obscured by a smoothing technique. If the data exists in raw voxel format smoothing is optional and could be done as a post processing step. The blocky nature of voxels becomes less of an issue at high resolutions, becoming unnoticeable at one pixel level resolutions. An example of a smoothing and triangulization technique is the marching cubes algorithm [4]. A comparison between raw data and the results provided by this algorithm can be seen in figure 1 and 2.

2.3 Sparse Voxel Octrees

The paper on sparse voxel octree presents a related concept to our system. A basic octree is a hierarchy that divides every node into eight child nodes. A node represents a cube which is then divided into eight smaller cubes. The sparse voxel octree only divides nodes if the child nodes contain geometry. This is an efficient way of storing empty space. Ray tracing works by recursion, going one level deeper on solid nodes and terminating on non-solid nodes. This makes traversing empty space efficient when ray tracing.

This does mean nodes must contain

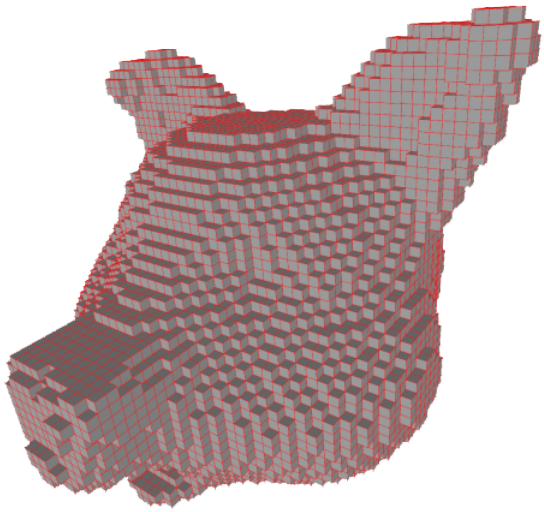

Figure 1: Raw voxel data.

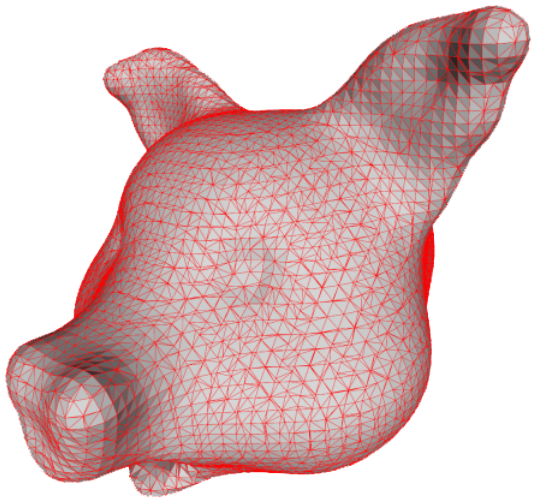

Figure 2: Smoothed triangle mesh using marching cubes.

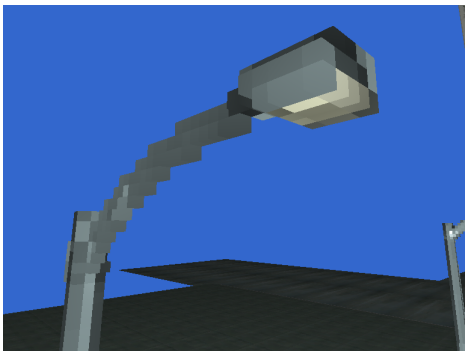

Figure 3: Cubical voxels.

Figure 4: Voxels with contours.

pointers to their child nodes. By storing children close to their parents nodes only need 16 bit pointers. In case this is not possible the pointer points to another 32 bit pointer. Color and normal attributes are stored separately from the solid nodes and are retrieved using implicit pointers based on the node location. While this datastructure has shown to be efficient in terms of storage due to the structuring it becomes clearly inefficient to edit.

They did not use cubical voxels in their representation of the data. Instead they propose a technique called *contours*, which smooths the voxels by intersecting them with two parallel planes. This reduces the number of voxels needed and can represent more complex geometry. A comparison between raw voxels and contours can be seen in figure 3 and 4. They also provide some streaming based on distance to camera. Using their approach they need up to 3 gigabytes of

GPU memory for high resolution datasets and achieve, depending on the scene, between 30 and 100 million rays per second.

2.4 Voxel Compression

Sparse voxel octrees is an obvious first step when compressing voxel data. This approach has been significantly improved by reusing data patterns. In a paper by Kämpe et al. [12] a directed acyclic graph datastructure is presented which can render voxels faster than a sparse voxel octree and store it more efficiently. Compared to an SVO the number of nodes needed is decreased for 26 up to 576 times based on the regularity of the geometry. In another paper [16] they propose a more efficient way to use DAGs to store shadows by using undefined voxels to require less DAG nodes. While this allows for very compact storage, run-time modifications of the data are not possible. The storage of shading parameters in an additional SVO further restricts flexibility.

In a paper by Smith [17] two algorithms are described that enhance the details of voxels while they can still be edited at original resolutions. One technique subdivides the voxels and another uses pixel art scaling algorithms to create more detail. This paper is meant to increase the detail in games as Minecraft or Voxelstein 3D while not increasing the difficulty of editing the world.

2.5 Voxel for Games

Since Minecraft [15] made voxels popular there has been a rise in commercial applications that use voxels. A lot of similar games exist which allow some form of world generation or editing. Voxels in these games are commonly quite large due to technical limitations. Our platform could benefit future generations of similar games as it supports high resolution datasets and has been designed with editing and world generation in mind. Several commercial engines exist that facilitate the use of voxels in games. Examples of these engines are Atomontage [2], Voxel Farm [18] and the C4 Engine [3]. Most of these allow editing of the environment.

2.6 Summary

So far the work done on voxels has either been on high image quality in most academic research or on high interactivity in most commercial games. Current research on voxels focuses on high quality physically based rendering. For games, voxels are attractive because of their versatility and editability. The combination of high resolution datasets and editability presents new possibilities in both worlds; this allows games to support smaller voxels or an increased draw distance and gives academic research the option to efficiently edit large datasets.

Figure 5: Example of large voxels and low draw distance in the game Minecraft [15].

Figure 6: Example of large voxels and low draw distance in the game CubeWorld [8].

3 Algorithm

For our algorithm the focus has been on keeping GPU memory requirements low and performance high. The CPU is in charge of data and memory management and the GPU is in charge of rendering. The system allows for a lot of freedom on the CPU to implement voxel loading, editing and generation.

In order to render large worlds with finite memory we store different versions of the data on disk, in RAM and on the GPU. The entire world, including subsurface data, is stored on disk in a compressed format. The largest amount of memory is available here and by compression we ensure large worlds can be stored. The CPU then stores a part of the entire world based on what is required for rendering and editing. Here we keep voxels in raw data format to allow fast access and modification. Using a streaming system we then upload the data required for visualization to the GPU. We ensure low memory usage by removing hidden voxels and compressing color.

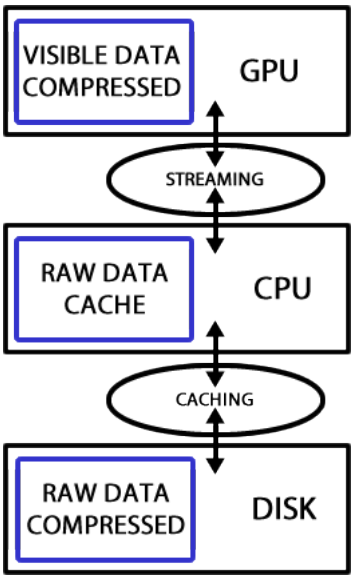

Figure 7: Data representation on different storage devices.

For real-time performance we propose an efficient rendering and streaming system. Ray tracing is done on the GPU for high performance. Streaming is done in a feedback loop which only uploads non-occluded data to the GPU. It retrieves a buffer from the GPU which tells us based on the last render pass what data needs to be uploaded. If the data is not yet available in RAM we load the required part from disk. Once the data is acquired we upload it to the GPU. An overview of the system can be seen in figure 8. We also use the occlusion based streaming system for editing. Instead of directly uploading the modified data we make it behave as if it has not yet been uploaded to the GPU. If the streaming system then requests it we simply upload the new version.

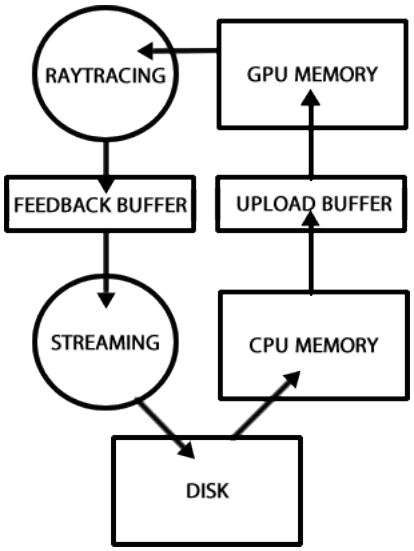

Figure 8: System overview.

3.1 Voxel Data Representation

Our datastructure consists of a hierarchy of uniform grids. At the lowest layer the grids contain raw voxel data, called brickmaps. Brickmaps encode solid data in a bitmask and store color seperately, they are small and can be stored in high numbers.

Brickmaps are stored in the brickgrid, which is a uniform grid spanning the entire draw distance. This grid contains pointers to brickmaps. Figure 9 shows a brickmap, as a blue grid, contained in the brickgrid, as a red grid. This datastructure is already capable of efficiently representing a large voxel world due to the sparse storage of brickmaps.

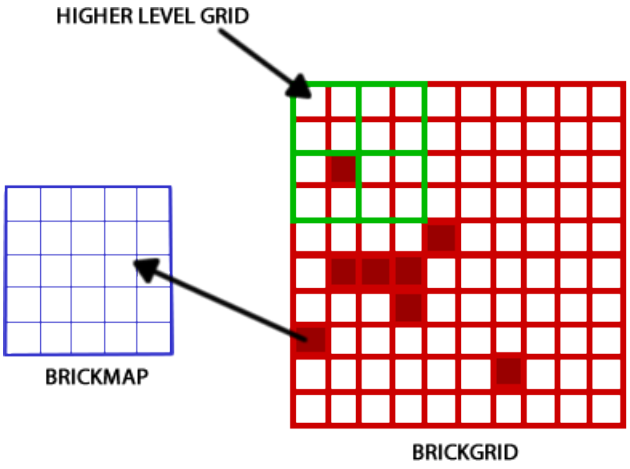

Figure 9: Datastructure Overview. Solid nodes in the brickgrid represent a pointer to a brickmap. Non-solid nodes represent an empty pointer.

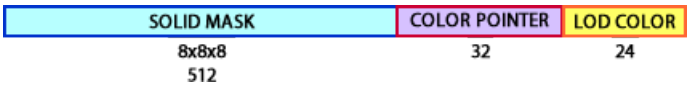

Figure 10: Brickmap datastructure.

3.1.1 Brickmaps

Brickmaps are the core of our datastructure and together they form the actual representation of the data. We empirically determined a good size for the brickmap to be $D^B = (8, 8, 8)$. Since brickmaps represent fine detail, which can be random in nature, the size of the brickmap is equal in all dimensions. Given this size 512 bits are required to represent the solid data of the brickmap. All together the datastructure consists of exactly 71 bytes, not counting shading attributes. The exact layout can be seen in figure 10.

We store shading attributes such as color separately because a relatively high number of voxels are empty and have no

shading attributes. Instead the brickmap contains a pointer to an array where these attributes are stored.

Finally the brickmap datastucture contains a level of detail(LOD) color which is used in case the brickmap is out of LOD range.

3.1.2 Brickgrid

Every cell of the brickgrid contains a pointer. A pointer either points to a loaded brickmap, an unloaded brickmap or is an empty pointer. If the pointer is empty the corresponding brickmap is also empty and no additional data is required. For streaming and editing purposes the grid cell can contain a brickmap which is not yet loaded into GPU memory. In this case the first three bytes of the pointer contain a LOD color and the last byte contains flags to determine it is unloaded.

The dimensions of the brickgrid can be set freely, and will generally follow the relative dimensions of the bounding box of the geometry. It should be noted that the size of the brickgrid also determines the draw distance as no brickmaps can be loaded outside of the grid. Given brickmap size $D^B \in \mathbf{R}^3$ and brickgrid size $D^G \in \mathbf{R}^3$ the draw distance D^V is given by

$$D^V = (D_0^B D_0^G, D_1^B D_1^G, D_2^B D_2^G)$$

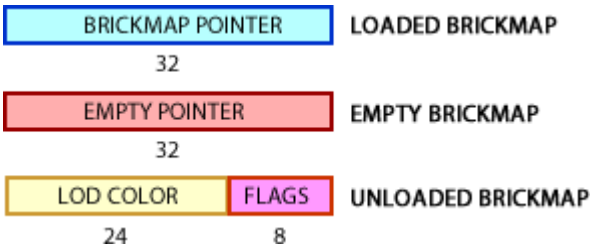

Figure 11: Brickgrid pointers.

3.1.3 Shading Attributes

Shading attributes are stored separately from the brickmaps. In our implementation we only store color. The voxels are rendered as cubes so normals are derived implicitly. The color data is stored in a compressed format based on S3 color map compression [14], specifically DTX1. DTX1 stores 16 pixels in 64 bits of data, achieving a compression ratio of 1:6. Applied to voxels this averages out to 4 bits per voxel. DTX1 also allows for an alpha mask to be applied, for our purposes we ignore this capability.

Color compression is applied right before uploading the brickmap to the GPU. This means colors can be stored in raw format in RAM which makes editing of the voxel data significantly easier. For static geometry it might be preferable to compress the color data in a preprocessing step. This will improve streaming performance without affecting any other parts of the system.

Compression combined with compact storage provides a flexible low memory solution to storing and retrieving shading attributes. Since compression is done in blocks of 16 colors there is a small overhead to be considered, in the worst

case space for an additional 15 colors must be allocated. Given the good compression ratio this is not much of an issue. Another advantage of this approach is that it can easily be extended to include additional shading parameters. Examples of these are normals or additional non cubical geometry, all of which can optionally be compressed.

3.1.4 Memory Management

Since brickmaps and color data are stored in dynamic quantities memory management is required to control memory usage. Built-in GPU memory allocation could be used but the nature of the data can be exploited resulting in faster and more efficient allocation.

Brickmaps are of constant size and stored in a simple object pool. The CPU keeps track of this pool and any allocations are sent to the GPU as an index in the object pool buffer. Allocation and deallocation are then done in $O(1)$ time without any memory overhead. This simple yet elegant solution is trivial to implement.

For color data a more complex system is required as color data is of dynamic size depending on the number of visible voxels in a brickmap. We use a memory allocation scheme that stores memory nodes at different levels. Every level has a different node size and on allocation we remove a node from the lowest level with a sufficiently high node size. This system allows for fast $O(1)$ allocation and in the worst case $O(F \log(F))$ deallocation. Where F stands for the number of free nodes at the corresponding level.

3.2 Streaming

Support for streaming is a natural extension to our system. Since little reconstruction of the datastructure is required to modify data streaming can be done fast and efficiently. Unlike voxel octrees there is not much room for LOD based streaming since our LOD level count is lower. While the system could be extended to allow for more LOD levels we have chosen for occlusion based streaming. This means the voxel data is gradually uploaded to the GPU based on feedback from the ray tracing algorithm. Therefore in practice only visible areas are ever uploaded to the GPU saving a lot on memory usage and allowing for large worlds to be rendered.

3.2.1 Occlusion Based Streaming

The process works by a feedback loop between GPU and CPU. The brickgrid is loaded into GPU memory in its entirety. Pointers to brickmaps are replaced by special pointers where the first three bytes determine the LOD color and the last byte has the 'unloaded flag' set.

During rendering if the algorithm encounters an unloaded brickmap it is added to the feedback list and the LOD color is used to render it. Once the rendering algorithm has completed the feedback buffer is retrieved from the GPU. This buffer determines what brickmaps need to be uploaded. For every brickmap a visibility test is performed. This test

removes invisible voxels, along with its corresponding shading attributes. During rendering this information is entirely unnecessary. Once the tests are completed we send the brickmaps in a batch to the GPU.

While memory management is done on the CPU this information is included in the unpack buffer. This way the GPU knows where to place the brickmap, the color data and which brickgrid pointer to replace. In our system we have set the unpack and feedback buffer size to 256 brickmaps per frame. In practice this works quite well and the entire scene can be updated within seconds while never heavily impacting the frame rate.

3.2.2 Editing

Our streaming system can easily be used to support real-time editing of the voxels. Editing is done on the CPU which then only has to send information to the GPU telling it which brickmaps are out of date. The brickgrid pointers are then set to unloaded and the streaming system will take care of the rest.

Since color compression and the visibility test are done right before uploading the CPU only needs to worry about memory management. When updating a brickmap the brickmap object pool pointer and color data must be freed. Since these operations are fast editing is easy and allows the user a lot of freedom. Editing can easily be executed in parallel, notifying the main thread once a brickmap has been edited. The data in fact only needs to be ready when it is requested to be uploaded by the GPU.

Since the editing process is so simple it can be done at almost the same speed as our streaming system. This allows entire landscapes to be updated and global effects such as erosion, snow, rain and fluid propagation can efficiently be simulated in real-time.

3.2.3 Streaming From Disk

Because of occlusion based streaming it is possible to render large geometry while only holding a part of it in GPU memory. Streaming from the CPU is fast but unfortunately RAM is usually a limited resource. By streaming from disk even larger geometry can be streamed while almost never running out of resources.

In this case data is stored in a cache between the GPU and disk. First the brickmaps are grouped together in a brickchunk, containing a group of brickmaps. Each time a brickmap is accessed we first determine what chunk it is in. We then look up the brickmap in the chunk and its corresponding color data.

If the GPU requests a brickchunk which is not yet in the cache it will be loaded from disk. We only allow a limited number of brickchunks to be stored in RAM so we never run out of memory. In our case we set the limit to one thousand based on the memory available. To enforce our artificial limit we store the brickchunks in a list. Each time a brickchunk is accessed it is moved to the front of the list. If the chunk is not yet in memory it is loaded

from disk and placed at the front of the list. If the list is full the last element of the list is freed and removed, keeping the number of brickchunks below the limit.

Additionaly the data on the disk is encrypted using LZ4 compression. Each chunk is compressed separately to allow for independent loading. This compression algorithm is fast at decompressing while yielding still good compression ratios. Especially for raw data, as is the case in our system, the algorithm is efficient. In our experiments the typical compression ratio for landscapes is 1:10. The compression combined with the cache means potentially infinite landscapes can be rendered while never running out of resources. The data could be streamed over a network or simply loaded from disk.

3.3 Ray tracing

Ray tracing is performed using a 3DDDA grid traversal on each level of the hierarchy and terminates when a solid cell is encountered or the ray leaves the grid.

Ray tracing a brickgrid works by first calculating the initial cell hit, which is determined by discretizing the intersection of the ray and the bounding box of a grid. After finding this cell we progress through the grid by comparing the intersections between the ray and every face of the current cell. We then change the current cell depending on the face we intersect first, until a cell is either solid or out of bounds. Since faces are axis aligned it is straightforward to calculate the intersection point and the cell index corresponding to that axis is modified.

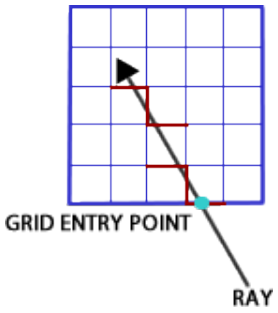

Figure 12: Ray tracing a grid. Red bars represent faces passed through. The cyan dot represent the initial entry point of the ray.

3.4 Additional Optimizations

For the purpose of optimization we add a third layer on top of the brickgrid. This layer is another uniform grid containing one bit per cell, spanning the entire draw distance. The bit determines whether the cell contains any geometry. This can be seen visually in figure 9 where the extra layer is the green grid on top of the brickgrid. When ray tracing we first ray trace this grid before descending into the brickgrid. Since the cells of this grid are generally very large it can traverse large empty areas very efficiently. This results in faster rendering of scenes with large open areas.

4 Implementation

In our implementation we focused on creating a working proof of concept and real-time ray tracing performance. Using toolkits such as OpenCL and CUDA we are allowed to deviate from the standard graphics pipeline and do real-time ray tracing on the GPU. While CPU ray tracers exist they are almost always outperformed by their GPU equivalents. With the support on most modern GPUs our implementation can run on a great variety of hardware.

During development CPU performance has not been our focus, it has not proven to be a bottleneck since most of the implementation on the CPU is straightforward. The pipeline is not complicated either since there is no preprocessing and the whole system is based on one feedback loop which takes care of all communication between GPU and CPU.

4.1 Ray tracing

Initially the viewmatrix is sent from the CPU to the GPU. Then, after initializing the feedback buffer, the ray tracing algorithm is executed. The core of our ray tracing algorithm can be seen in appendix A, it calculates the intersection between a ray and a cube.

The ray tracing algorithm begins by calculating the view ray and then initializing the ray tracing procedure for the highest layer. On hit the algorithm ray traces the brickgrid corresponding to the current cell on the highest layer. In our implementation one cell in the higher layer contains $(4, 4, 4)$ brickgrid cells.

On brickgrid hit the algorithm performs differently then before. In this case we hit a brickmap. First a check is done whether we are out of LOD range. If so the LOD color is retrieved. Based on whether the brickmap is loaded the color is either retrieved from the brickmap or from the first three bytes of the brickmap pointer.

In case we are in LOD range we check the pointer flag to determine whether the corresponding brickmap has been uploaded yet to the GPU. If this is the case we ray trace the brickmap. Otherwise we check for another flag to determine whether this brickmap has already been requested to be uploaded. If this is not the case we add the brickmap index to the feedback array. We also set the mentioned flag for this pointer, so it is not added to the array again. However if the feedback array is full we abort the function and we do not set the requested flag. The user is free to set the size of the feedback array which can affect the streaming speed.

If the brickmap is loaded we continue to ray trace the corresponding grid. In case we hit a voxel we calculate the color index. This is done by counting all set bits at every index lower then the current bit index. For example if 13 visible voxels have a lower index than our current voxel the color attachment index is 13.

Bitcounting can be done fast and efficiently giving the worst performance for the voxel with the highest index. By

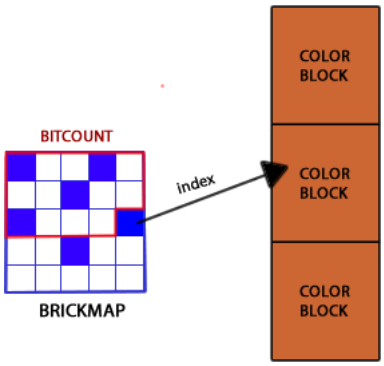

Figure 13: Color Attachment Lookup.

storing additional index offsets per a number of voxels would increase speed at a memory cost. Since color attachment lookups are done only once per pixel it is not a priority for optimization. Memory costs become even lower since we reduce the number of visible voxels by the visibility test.

Once we have found the color index we look it up in the color array and decompress it. By keeping track of the last face we passed through during the tracing of the grid we can determine the normal. In case we immediately hit a solid voxel, and did not pass through any faces, we find the normal by normalizing and rounding the difference between the hit point and center of the brickmap. Upon completion the color and normal are returned and shading is applied.

4.2 Streaming

Based on the last render pass the feedback array is retrieved from the GPU. The elements in this array denote missing bricks in GPU memory. This information is used by the CPU to retrieve data from disk, or to generate the data (in case of procedural content).

Retrieved brickmaps are processed before uploading. Voxels completely surrounded by opaque voxels are removed. The visibility test can be extended to test neighbouring brickmaps for occluded voxels. Doing this comes at an extra performance cost but can save a lot of memory around the borders. In case streaming from disk is enabled the border test should only be performed if the neighbouring brickmap is already loaded into RAM. Loading an additional chunk from disk may not be worth the memory reduction it gives. Turning visibility testing on has no effect on the ray tracing results. This would only be the case if it were possible to look inside voxels, which we assume is not the case.

After invisible voxels have been removed we compress the colors of the remaining voxels by DTX1 compression. Each DTX1 block stores two 16-bit R5G6B5 color values c_0 and c_1. Each color in the block then has two corresponding bits. If the no bits are set the color is equal to the c_0, if all bits are set the color is equal to c_1. If only the first bit is set the color is equal to $\frac{2}{3}c_0 + \frac{1}{3}c_1$. If only the second bit is set the color is equal to $\frac{1}{3}c_0 + \frac{2}{3}c_1$.

After brickmaps are processed they are put into an array. This array contains unpack entries which contain the raw brickmap data and offer enough data to store any number of colors attachments. Colors are stored in blocks of 8 bytes containing 16 colors. Since a brickmap contains at most 512 voxels the maximum data used by color attachments is, with

4 bits per voxel, 256 bytes. It also stores the brickmap pool pointer, color pointer and color size. Using these pointers the brickmap and color data can be moved to the correct place in GPU memory. The datastructure is memory-wise not efficient since every entry takes up the maximum color data possible for a brickmap. Since the size of this array is equal to the size of the feedback array, which is 256, this is not an issue.

Streaming can be done efficiently since brickmap data is only actually required once the GPU has requested it. It is even possible to generate brickmaps on the fly while the ray tracer is running. The only thing that is required beforehand is that it is known which brickmaps will contain voxel data. By putting a limit on the number of updates per frame we ensure the frame rate on both GPU and CPU is not too heavily impacted by streaming. If the frame rate is high enough this will not heavily impact the streaming rate either.

4.3 Memory Management

The color memory management system works by keeping a list of free nodes at a number of levels. The lowest level is 16 bytes and the highest level 256 bytes. Initially all nodes are stored at the 256 byte level. In case a node is required at a level that has no nodes a node from a higher level is taken and split into two nodes of the current level. If one level contains too many free nodes all nodes are sorted and adjacent nodes merged and moved up to a higher level.

4.4 Editing

The streaming system can easily be used in conjunction with editing. It runs on a different thread and notifies the main thread once a brickmap has been modified. The main thread only needs to know which brickmap index has been updated, so we store the indices in a buffer.

At the end of the render pass we check whether new indices have been added to the buffer and once we are done we clear it. Below is a list of steps that describe the editing process.

- We look up the brickmaps by index and read the LOD color, which might have changed.
- The brickpool and color pointers are freed since they will be reallocated during uploading.
- We then send the index and LOD color to the GPU in an array.
- The GPU processes this array and for each index the corresponding pointer in the brickgrid is set to unloaded
- The new LOD color is stored in the pointer so the level of detail is already updated before streaming.

In our system editing can be done at little performance loss. This is due to the asynchronous in which editing can be done. The rendering thread operates completely separately and must only be notified when something has changed.

4.5 Streaming from Disk

When loading or saving to disk the brickmaps are grouped in chunks. In our implementation chunks contain $(16, 16, 16)$ brickmaps. In our format we first stored the brickgrid, with a solid bit and a LOD color. This buffer is then compressed using LZ4 compression. When streaming from disk this buffer is loaded so it is known which brickmaps are empty and which are not.

Then an array of offsets into the file are stored. Each offset represents a chunk. This is where the initial loading ends.

When a chunk is required, which has not yet been loaded into memory, the chunk offset is looked up in the offset buffer. Then using the offset the chunk is read from the file and decompressed. All brickmaps contained in the chunk are then loaded in one by one. The chunk also keeps a local brickgrid indexing the brickmaps. When a brickmap is requested it can simply be looked up in the local brickgrid.

While streaming from disk does slow the process down it allows memory usage on CPU to be constant. It also allows the cache size to be dependent on available RAM. Storage on disk is not an issue either since LZ4 compression gives a signficant size reduction when storing raw voxel data while giving fast decompression speeds.

5 Results

To get a clear answer to our research questions we setup experiments to test the performance of our system. Initially we set out to create a system that can render large voxel datasets in real-time with support for advanced features such as real-time streaming and global editing. In order to verify whether our system is capable of these conditions a few criteria must be met. Most importantly the memory usage on GPU must remain under the limit of current hardware capabilities. Low memory usage means more detail can be put into the system providing higher quality. Since our datasets are so large this is the primary condition for the system to be feasible.

Secondly the system must run in real-time. In order to compete with rasterization-based techniques ray tracing performance must be similar. If our approach adds a lot of extra latency it might not be worth the advantages it provides over other methods. Support for real-time editing becomes useless if slow ray tracing performance makes it feel slow and unresponsive.

Thirdly the performance impact of streaming and editing must be reasonable. The advantage of having a simple and dynamic datastructure is that modification should be fast. If this is not the case more complex datastructures might be equally suited for editing and streaming. While our focus has been partially on user freedom over performance we still expect much better editing performance than other techniques.

5.1 Measurements

For every test scene several statistics have been gathered. To measure memory usage we first calculate the static memory usage which depends on the scene parameters. Given a brickgrid size $D^G \in \mathbf{R}^3$ and extra layer node size $D^L \in \mathbf{R}^3$ we can calculate the the brickgrid memory usage M_G and the extra layer memory usage M_L.

$$M_G = 4(D_0^G D_1^G D_2^G)$$

$$M_L = \frac{D_0^G/D_0^L D_1^G/D_1^L D_2^G/D_2^L}{8}$$

In our case on cell in the extra layer contains 64 brickgrid pointers due to corresponding node size $D^L = (4, 4, 4)$. For this node size the ratio of memory usage between the brickgrid and extra layer is 2048. That is the brickgrid uses 2048 times as much memory. We keep track of dynamic memory usage by counting the allocations and deallocations in our memory managers. We do not count the static buffers used for streaming and editing since they are small and the user can set the size of these buffers.

Rendering performance is measured in millions of rays per second. This is done by calculating the time per frame in nanoseconds and taking into account the screen resolution. Rays per second (RPS), given frametime dt and screen resolution $w \times h$, is then:

$$RPS = \frac{10^9}{dt/(w * h)}$$

To get a representative result we measure 100 frames and take the average.

To measure streaming and editing

performance we store along with the frametime the number of brickmaps uploaded to the GPU. Then by comparing the frametime against the number of brickmaps uploaded we find the streaming performance. Editing can be measured in a similar manner. When editing the number of brickmaps uploaded will be higher than one as well, therefore it can again be used to compare results. The editing simulation itself happens on another thread and should not influence ray tracing performance.

5.2 Experiments

We used three test scenes and measured our data from three different perspectives. We use three perspectives since our occlusion based streaming system only loads in visible brickmaps resulting in different memory usage. We attempted to use three perspectives in all our test scenes. The first perspective uses a distant camera to test memory usage and performance when viewing the entire environment. This perspective is also best suited for comparison with other approaches that do not use occlusion based streaming. The second perspective is at medium distance and at a different angle. The third perspective is only a part of the environment from up close.

The first test scene we used is a landscape generated by perlin noise (Figure 14, 15, 16). We generated this scene on a resolution of 8096x8096x256. It is designed to alternate between small mountains and flat terrain. The algorithm was designed with large generated editable terrains in mind so it makes a natural testing scene. Since the scene is customly generated we can not compare the results with other techniques. Fortunately it does allow for great testing of the streaming and editing functionalities.

The second test scene is the Conference Room scene (Figure 17, 18, 19). We used this in order to compare memory usage between our algorithm and that of Efficient Sparse Voxel Octrees by Laine and Karras [13]. Out of their test scenes it most resembles a large environment for which our system was designed. While also single models can be used all of our testing has been done in landscape-like environments. We tested the scene on two resolutions, 1024x1024x1024 and 2048x2048x2048. These are equivalent to octrees of respectively depth 10 and 11.

The third scene we used is the Hairball scene (Figure 20, 21, 22). A very irregular scene also used in Sparse Voxel Octrees to test worst case performance. This scene would make an interesting addition since our algorithm might be effective at rendering and storing highly random geometry. Constructing a sparse voxel octree on such geometry is memory consuming. Unlike brickmaps which do not increase in memory usage for more random data. Also occlusion based streaming could provide significant benefits for such dense geometry.

The system used for the experiments consists of an Intel i7-3770 CPU @ 3.40 GHz, 10GB RAM and an ATI Radeon HD 7970 video card.

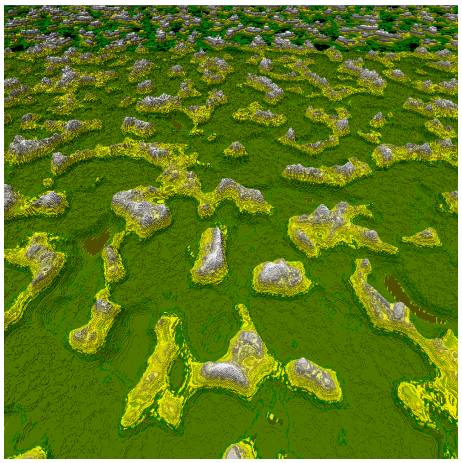

Figure 14: Landscape 8096x8096x256 Viewpoint 1.

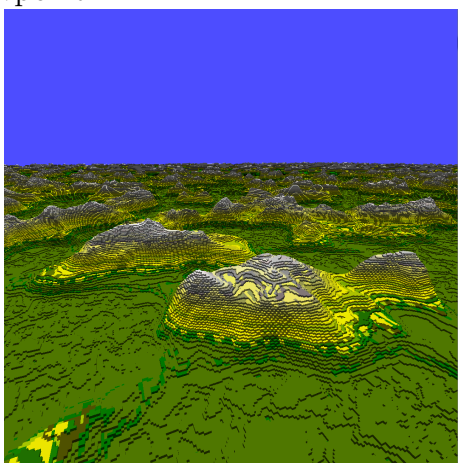

Figure 15: Landscape 8096x8096x256 Viewpoint 2.

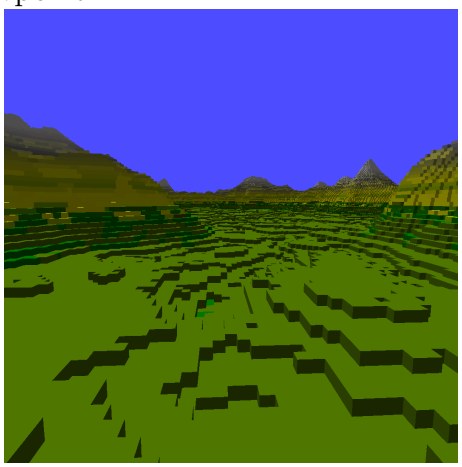

Figure 16: Landscape 8096x8096x256 Viewpoint 3.

Figure 17: Conference 2048x2048x2048 Viewpoint 1.

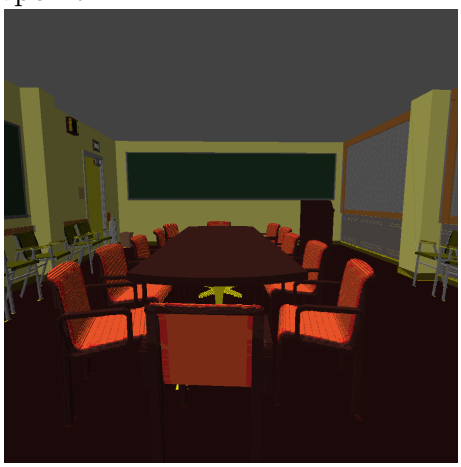

Figure 18: Conference 2048x2048x2048 Viewpoint 2.

Figure 19: Conference 2048x2048x2048Viewpoint 3.

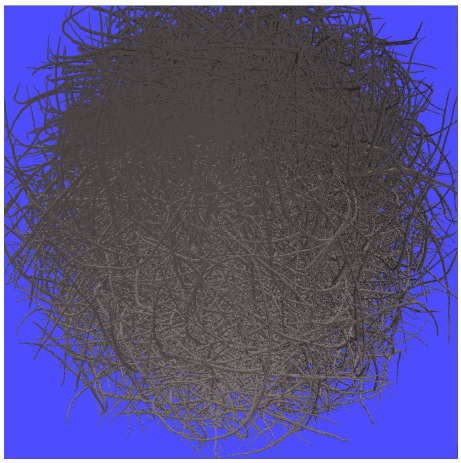

Figure 20: Hairball 2048x2048x2048 Viewpoint 1.

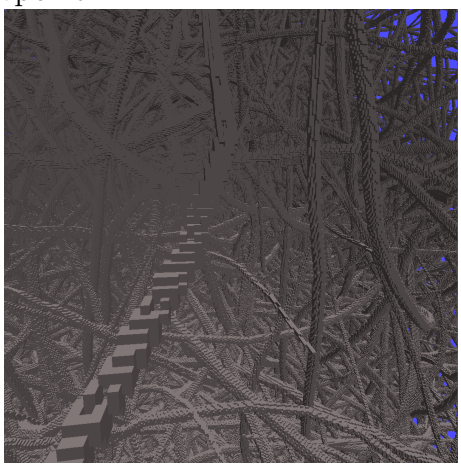

Figure 21: Hairball 2048x2048x2048 Viewpoint 2.

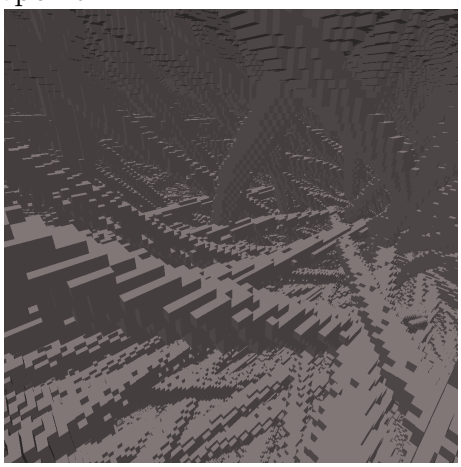

Figure 22: Hairball 2048x2048x2048 Viewpoint 3.

5.3 Memory Usage

The measurement results can be viewed in the table below. The memory usage consists of static memory required for the grid and extra layer and dynamic memory usage required by brickmaps and color data. Voxel resolutions used for conference and hairball are 2048x2048x2048, for landscape we used 8096x8096x256. For all test scenes static memory usage is the same. The brickgrid uses 64MB and the extra layer uses 0.03125MB.

Scene	Viewpoint	Memory Usage(MB)
Landscape	1	90.265
	2	71.806
	3	64.539
Conference	1	99.710
	2	82.935
	3	71.492
Hairball	1	112.146
	2	77.267
	3	64.948

Table 1: Memory usage on GPU in MB per scene. Scenes and viewpoints can be seen in figures above.

First we observe that static memory makes up the largest part of memory usage for all test scenes. For some viewpoints that occlude most of the environment almost all data is taken up by static memory. Since we are storing small brickmaps in a large grid the grid takes up most memory by far. Due to occlusion based streaming the memory used by brickmaps and color is fairly low. Visibility testing on voxels and color compression make sure the memory usage is even lower. However if the entire scene would be loaded onto the GPU dynamic memory would probably be larger than static memory.

Comparing the viewpoints we see that for more up close viewpoints the memory usage goes down. A higher viewpoint id means the camera is more zoomed in. From this we can clearly notice the effect of occlusion based streaming. Viewpoint 1 in all scenes almost loads in the entire scene. Viewpoint 3 only loads in a small part. Taking into account the 64MB static memory the difference is very large. Even though the memory savings are significant one may chose to not use streaming anyways since it is noticeable while moving through the world. Storing level of detail in the pointers does a good job of hiding the streaming process but LOD popping is still visible.

Looking at the difference between scenes we see hairball takes significantly more memory than conference or landscape. This is to be expected since it is the most random and complex geometry out of the three. Landscape uses the least memory which is also to be expected since the system was designed around landscapes. Since the landscape is so much larger on the width and length it is easier for LOD to pop in and reduce memory usage. Conference is in equal size across all dimensions and therefore LOD does not come much into effect.

5.3.1 Comparison with SVOs

If our algorithm is to be truly feasible it must compete with other current solutions. The most similar and prominent

one being sparse voxel octrees. We have taken measurements from the paper Sparse Voxel Octrees paper by Laine and Karras [13] and compared them to our own. We used their cubical voxel memory usage for the conference and hairball scene, not taking into account their smoothing technique called contours. Our measurements are taken from viewpoint 1 of the previous section. The results can be seen below.

Scene	Brickmap	SVO
Conference 1024	23	21
Conference 2048	100	89
Hairball 1024	25	262
Hairball 2048	112	1157

Table 2: Memory usage comparison in MB between our brickmap approach and Sparse Voxel Octrees by Laine and Karras [13].

The results show that for the conference scene their memory usage is slightly lower than ours. For the Hairball scene however our memory usage is much lower than theirs. As expected sparse voxel octrees (SVOs) struggle handling random organic data than our brickmap approach. SVOs rely on the fact that large cubical regions are either empty or solid. The hairball is most likely a worst case scenario for SVOs since it has few solid or empty cubical areas. Our approach can deal with this more efficiently since detailed geometry is stored the same way in a brickmap no matter its shape. We only rely on small cubical areas the size of a brickmap being empty which is much more likely to happen.

For the Conference Room our memory usage is slightly worse. In fact SVOs are probably more efficient at storing this kind of geometry than the results show. While we used a global viewpoint and tried to store as much geometry in one view occlusion based streaming still saved us a signficant amount of memory. SVOs only use LOD based streaming, if they used occlusion based streaming their memory usage would drop significantly. Even though we are less efficient at storing Conference Room the difference is not very large. Our memory usage seems to grow at the same rate as well for both scenes. This suggests our memory usage will never be much higher than SVOs for any resolution.

As can be observed from our measurements, memory usage of our system is competitive with SVOs. Our system does benefit a lot from occlusion based streaming which could be implemented for SVOs as well. Yet our algorithm allows for natural and simple implementation of streaming where it would be more involved for SVOs.

5.4 Ray tracing Performance

The number of rays we can trace per second is another important property of the system. A higher number of rays allows for more advanced rendering techniques such as shadows, ambient occlusion, anti-aliasing or indirect lighting. Table 3 shows the measurements for the three test scenes.

Scene	Viewpoint	MRays/s
Landscape	1	104.252
	2	164.275
	3	117.941
Conference	1	75.819
	2	86.349
	3	111.623
Hairball	1	40.015
	2	68.448
	3	84.243

Table 3: Rendering performance in million rays per second. Screen resolution: 1068 x 986. Scenes and viewpoints can be seen in figures above.

As can be seen we achieve real-time performance in every test scene ranging from 40 to 164 MRays/s. If we look at the difference between viewpoints we see that, as one would expect, zooming out has a negative impact on the frame rate. For Hairball the performance doubles when viewing from up close compared to viewing the entire mesh. One notable thing is that for the landscape scene we achieve better performance for the second viewpoint. This is because of the angle taken which shows a large portion of open air. This means less rays hit a brickmap and exit the grid faster resulting in better performance.

Ray tracing performance does seem quite viewpoint or scene dependent. Which is a good since it implies environment based optimization is going on. It is also bad since a non steady frame rate can ruin immersion and make the user perceive lag. Optimization is not perfect however since there is a noticeable drop in frame rate for larger resolution brickgrids meaning more empty grid cells to trace through. This can be fixed by increasing the node size of the extra layer making it skip more brickgrid cells. This is however inefficient for detailed geometry for which the node size is too large giving bad performance in that case. Having multiple extra layers above the brickgrid can fix this problem.

5.4.1 Comparison with SVOs

Comparing ray tracing performance with Sparse Voxel Octrees is a bit more difficult since they are unclear on what dataset size they used in their performance measurements. They claim they used maximum size they can fit in 4 GB. For hairball this is most like the 2048x2048x2048 version since they did not provide a higher resolution in their results.

Scene	MRays/s
Conference	43.8
Hairball	22.5

Table 4: Rendering performance results in million rays per second for Sparse Voxel Octrees by Laine and Karras [13].

It seems that we achieve better performance looking at our results. For Hair-

ball this is to be expected since our algorithm is much more efficient at storing the dataset. For Conference Room it is hard to say if our results are better depending on the dataset resolution they used. Still it would appear our performance is on a competitive level with SVOs. In our system there is no need for a stack which is a bottleneck for SVOs on the GPU as mentioned in Efficient Sparse Voxel Octrees by Laine and Karras [13]. Laine and Karras also provide additional measurements with beam improvements which traces a lot of rays at once providing better ray starting points. A similar algorithm can be developed for our solution potentially improving rendering performance.

5.5 Streaming performance

We measured streaming performance for the generated landscape test scene. In figure 23 we plotted the ray tracing performance and number of brickmaps uploaded against the frame index. The frame index increases over time as frames are processed and the world is streamed to the GPU. The results show a drop in rays per second as more data is loaded in. Right before the final batch of brickmaps is uploaded we measure a rendering performance of 75.8 MRays/second against 104.3 MRays/second in the first frame no brickmaps are uploaded. This suggest a frame drop of about 27% while streaming. While noticeable our system is still realtime during all stages of the streaming process. The performance is at its worst right before all brickmaps are uploaded. We implemented a snow simulation to test our editing system. We chose the snow simulation since it is a global simulation that modifies the entire landscape. Only our system is capable of editing at this scale. It gives the same performance as streaming since actual editing happens on a different thread and streaming loads in modified or new brickmaps.

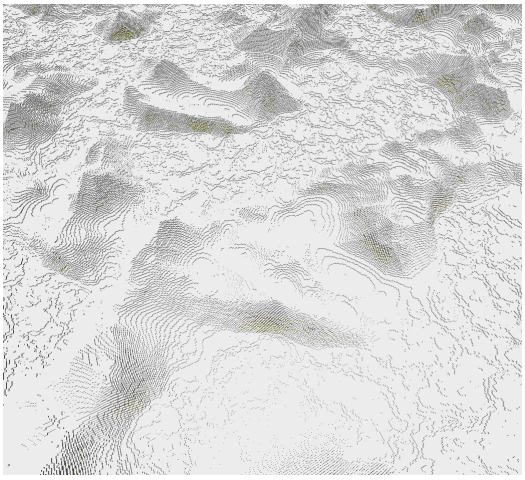

Figure 24: Landscape after snow simulation.

5.6 Summary

Our implementation has shown the datastructure can support a high number of features and is easily extensible. Certainly it is also possible for a large number of performance improvements to be implemented. But we have shown the algorithm to be robust and highly flexible which was our main focus. Our implementation could serve as a basic framework for either lage landscape simulations or games/applications with an editable world. There is still a lot of room for improvement, especially for level of detail and optionally smoothing voxels. All in all

with our implementation we have shown what we wanted to and hope it can be built upon to provide more reliable and applicable solutions.

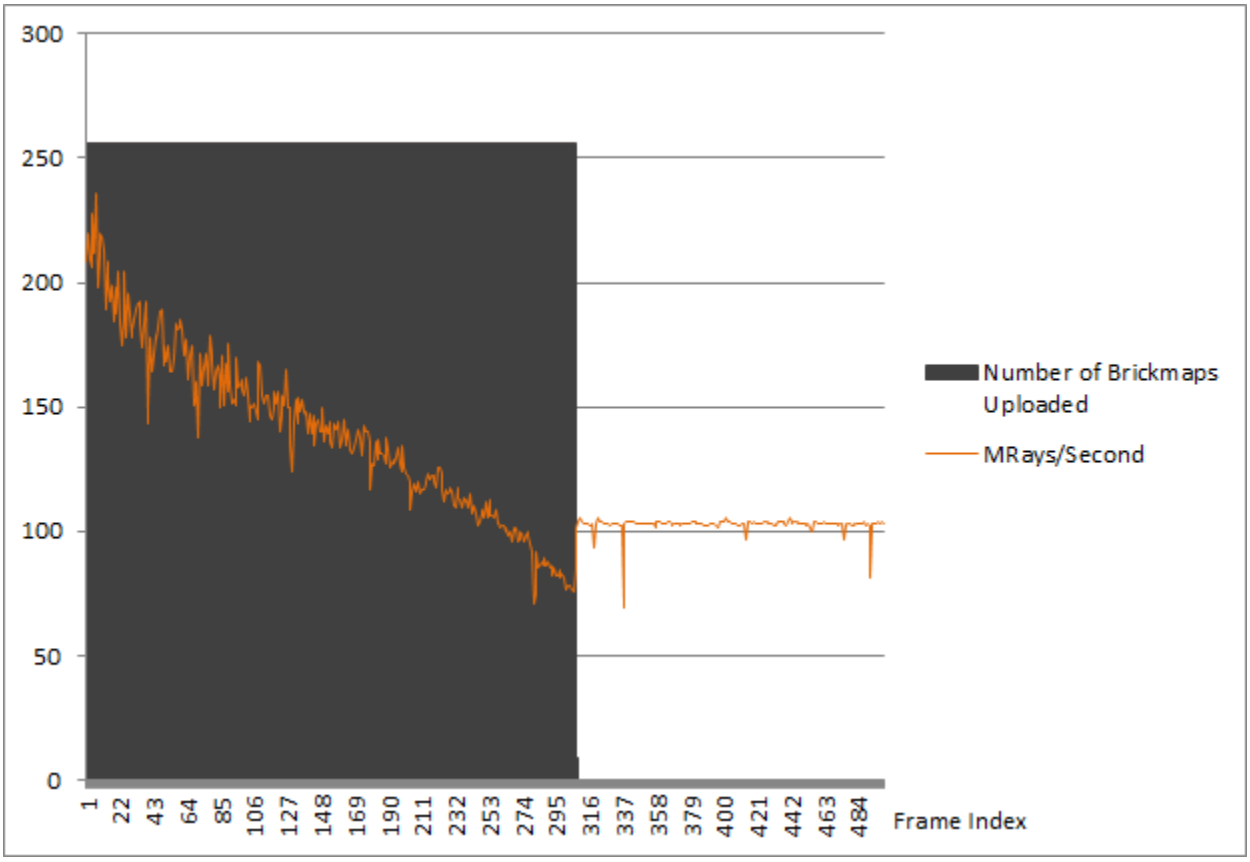

Figure 23: Ray tracing performance against number with number brickmaps streamed per second over time.

6 Conclusion

We have presented a system that provides a real-time solution to ray tracing of large voxel datasets. It allows interactive editing at large scales and supports streaming to allow for potentially infinite landscapes. There is support for color compression and level of detail. Yet one of the strongest points of the system is that the dataformat is kept in raw format. Accessing the datastructure becomes fast and easy allowing the system to be extended upon and easily changed to suit the user's needs. We believe the results are promising. It already shows good results and great promise for the future by allowing a lot of extensions and further improvements to be done. Finally it asks the question whether constructing complex datastructure such as sparse voxel octrees is worth the constraints it puts on editing and flexibility.

6.1 Future Work

One thing there could be improved on is level of detail. While there is already some minor support there is still a lot of work to be done. Extra levels of detail in the brickgrid can make ray tracing a lot faster for high resolution datasets. Also empty areas could be traversed a lot faster. Our results have already shown adding extra layers above the brickgrid does not significantly increase memory costs. On brickmap level LODs might improve visual quality and allow for better compression and faster retrieval of shading parameters.

References

[1] John Amanatides and Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *In Eurographics 87*. 1987, pp. 3–10.

[2] *Atomontage Engine*. http://www.atomontage.com/.

[3] *C4 Engine*. http://www.terathon.com/.

[4] Evgeni V. Chernyaev. *Marching Cubes 33: Construction of Topologically Correct Isosurfaces*. Tech. rep. 1995.

[5] Vasco Costa and João M Pereira. "Compact Rectilinear Grids for the Ray Tracing of Irregular Scenes". In: *WSCG 2011*. 2011.

[6] Cyril Crassin. "GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes". English and web-optimized version. PhD thesis. UNIVERSITE DE GRENOBLE, 2011. URL: http://maverick.inria.fr/Publications/2011/Cra11.

[7] Cyril Crassin et al. "Interactive Indirect Illumination Using Voxel Cone Tracing". In: *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)* 30.7 (2011). URL: http://maverick.inria.fr/Publications/2011/CNSGE11b.

[8] *Cube World*. https://picroma.com/cubeworld.

[9] Sven Forstmann and Jun Ohya. "Efficient, High-Quality, GPU-Based Visualization of Voxelized Surface Data with Fine and Complicated Structures." In: *IEICE Transactions* 93-D.11 (2010), pp. 3088–3099. URL: http://dblp.uni-trier.de/db/journals/ieicet/ieicet93d.html#Forstmann010.

[10] Attila T. fra. "Interactive Ray Tracing of Large Models Using Voxel Hierarchies". In: *Computer Graphics Forum* 31.1 (2012), pp. 75–88. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.02085.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.02085.x.

[11] Enrico Gobbetti and Fabio Marton. "Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms". In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 878–885. ISSN: 0730-0301. DOI: 10.1145/1073204.1073277. URL: http://doi.acm.org/10.1145/1073204.1073277.

[12] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. "High Resolution Sparse Voxel DAGs". In: *ACM Trans. Graph.* 32.4 (July 2013), 101:1–101:13. ISSN: 0730-0301. DOI: 10.1145/2461912.2462024. URL: http://doi.acm.org/10.1145/2461912.2462024.

[13] Samuli Laine and Tero Karras. *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. NVIDIA Technical Report NVR-2010-001. NVIDIA Corporation, Feb. 2010.

[14] D.-M. Liou, Y. Huang, and N. Reynolds. "A new microcomputer based imaging system with C3 technique". In: *Computer and Communication Systems, 1990. IEEE TEN-*

CON'90., 1990 IEEE Region 10 Conference on. 1990, 555–559 vol.2. DOI: 10.1109/TENCON.1990.152671.

[15] *Minecraft*. https://minecraft.net/.

[16] Erik Sintorn et al. "Compact Precomputed Voxelized Shadows". In: *ACM Trans. Graph.* 33.4 (July 2014), 150:1–150:8. ISSN: 0730-0301. DOI: 10.1145/2601097.2601221. URL: http://doi.acm.org/10.1145/2601097.2601221.

[17] Adam M. Smith. "Two Methods for Voxel Detail Enhancement". In: *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: ACM, 2011, 6:1–6:4. ISBN: 978-1-4503-0872-4. DOI: 10.1145/2000919.2000925. URL: http://doi.acm.org/10.1145/2000919.2000925.

[18] *Voxel Farm*. http://voxelfarm.com/.

Appendix

A
Ray-Box intersection algorithm

Below is the algorithm pseudocode used to determine the intersection between a cube with extremes p_0 and p_1 and a ray. The ray is given by $l = \text{ray.start} + t(\text{ray.delta})$.

Data: p0 : Point; p1 : Point; ray : Ray;
Result: Hit : Boolean
Vec3 t1 = (p0 - ray.start)/ray.delta ;
Vec3 t2 = (p1 - ray.start)/ray.delta ;
Vec3 vmax = max(t1,t2) ;
Vec3 vmin = min(t1,t2) ;
Float tmax = min(vmax.x, vmax.y, vmax.z);
Float tmin = max(vmin.x, vmin.y, vmin.z);
return (tmin $\leq$ tmax && tmax $\geq$ 0) ;

Algorithm 1: Ray-Box intersection

The algorithm works by first determining the t-values for both extreme points on the cube. The t-values are then split between two vectors one containing the minimum t for each axis and one containing the maximum. By taking the maximum of the minimum vector we find the t-value t_{min} at which the ray intersects the box. This holds since for every axis the t-value required to pass the corresponding edge is either lower or equal. Similarly by taking the minimum of the maximum vector we get the t-value t_{max} at which the ray exists the box. If $t_{max} \geq t_{min}$ the ray does not intersect the box since the exit point can not be before the entry point. If $t_{max} \leq 0$ the box is exited before the ray starts, therefore the box is not intersected either.