UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

Sabre

A reusable GPU Voxel Renderer

AUTHOR

Liam Power · 16148983

SUPERVISOR

Dr. Chris Exton

Final Year Project Report — B.Sc. Computer Games Development

Computer Science and Information Systems

University of Limerick

Abstract

With the rise in power of consumer graphics hardware, it becomes possible to simulate and render increasingly large and complex virtual environments in games. Voxel-based technology — first introduced as a means to visualise medical and scientific datasets — is an innovative alternative to contemporary polygon-based techniques for representing massive game worlds that can be modified in real-time.

A significant problem faced by game programmers when integrating voxels into their product is the overhead of developing a robust voxel storage and rendering scheme, as naive storage of voxels can consume huge amounts of memory. In place of developing such a solution in-house, developers may instead choose to utilise a library to provide the voxel storage and rendering technology.

This project aims to deliver such a library. Through the use of specialised data structures and the exploitation of the inherently parallel nature of voxel data, this project provides drop-in support for massive voxel scenes that can be viewed at a very high level of detail using modern OpenGL graphics programming techniques. In addition to this core library, this project also provides an example viewer application demonstrating the capabilities of the library.

Since this project is designed for easy integration, the core voxel library does not rely upon any external dependencies, other than the OpenGL graphics API for interfacing with the graphics hardware.

Declaration

This work is submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Games Development. I declare that, except where otherwise indicated through appropriate reference and quotation, this report is entirely my own work and has not been submitted to any other academic institution, or for any other academic award at this University.

Liam Power
May 5th, 2020.

Acknowledgements

I would first like to thank my supervisor, Dr Chris Exton, for his patience and support throughout this project. I would also like to thank the faculty at the Computer Science and Information Systems department, in particular Dr Patrick Healy, without whom I would most certainly be still trying to draw my first triangle.

Thanks are very much due to Dr. Samuli Laine and Dr. Tero Karras; without their pioneering research in sparse voxel octrees, this project would never have been possible.

I would also like to thank my friends Jack, Eric and Jonathan, and my brother Cathal for their assistance in testing the software portion of this project.

Finally, I would like to thank my parents; my mother for her unfailing kindness and encouragement, and my father for passing on his knowledge of programming. Without their influence, this project would have never even been conceived.

Contents

Chapter 1: Introduction

1.1 Context

1.1.1 Polygon Rendering

Since the 1990s, polygons (more specifically, triangles) have been the dominant method of representing geometry in games. A flat surface can be very efficiently encoded by a polygon as the rendering engine only needs to store the vertices at the corners of the triangles making up the surface. The surface is rendered by filling in the pixels between vertices.

While polygons are an excellent representation of perfectly flat surfaces, their efficiency decreases when surface details such as bumps, cracks and ridges are added. This is because, for each new surface feature added, more polygons are required to represent that feature accurately. Although techniques such as normal mapping can help to provide the illusion of details on a flat surface, changing the surface geometry may incur expensive re-calculation, making real-time editing of extremely large meshes very difficult.

1.1.2 Voxels

The word *voxel* is derived from the term *Volume Pixel*. Just as a pixel represents a single colour value in a regular two-dimensional bitmap grid, a voxel represents a single point in three-dimensional space with some associated data values, such as colour, light or temperature. Like pixels, voxels do not store their positions. Instead, the position of any particular voxel can be derived implicitly from its grid location.

Voxels may be assembled into scenes, much like Lego® pieces. By using a very large number of very small voxels, creators can produce extremely detailed scenes (see figure 1.2). These voxel scenes may then be arbitrarily modified by both developers and end-users. This concept is demonstrated in the 2011

video game *Minecraft*, where the entire game world is a continuous voxel scene. Players may build any structure they like out of voxels, and the game world may be edited at any point by players.

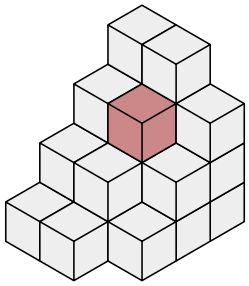

Figure 1.1: Voxels assembled into a staircase shape, with a single voxel highlighted in red [28]

NovaLogic Inc., developers of the *Comanche* games, were one of the earliest adopters of voxel technology. NovaLogic's voxel engine functioned by raycasting a voxel grid to display realistic terrain. In 2011, voxels gained global recognition with the release and explosive success of *Minecraft*. Since then, voxels have been applied to more ambitious game projects, such as Hello Games' *No Man's Sky* [21] and Media Molecule's *Dreams*, where players can shape completely new games entirely out of voxels using the *Dreams* engine [8].

1.1.3 Code Structure

The project's codebase is split into three modules. The `sabre` module is an executable program which functions as a test-bed and debug viewer for voxel objects. The `sabre_svo` module contains all of the code necessary to generate and manipulate voxel objects. The `sabre_render` module handles data synchronisation between the CPU and GPU using the OpenGL graphics API. These latter two modules comprise the *Sabre* "library" — the main artifact that will be distributed to client developers. The viewer module is intended only as a demonstration of the core library's feature set. An optional module, `sabre_import`, allows client developers to convert polygon mesh assets into fully voxelised scenes.

This project is written in C++, but makes minimal use of the C++ STL and object-oriented programming. To allow developers to use this project from other programming languages it is necessary to use C linkage, using the `extern "C"` construct.

The primary distribution mechanism intended for this project is as source

Figure 1.2: Voxels can do more than just low-resolution blocky assets; this Indonesian statuette, composed of roughly 150,000 voxels, was rendered with this project. The 3D model used to produce this scene was sourced from the Morgan McGuire Computer Graphics Archive [20]

code. This allows clients to simply compile the library into their own projects, eliminating complex and error-prone library and environment setup. This distribution model is made possible by removing dependencies wherever possible, and bundling required dependencies, such as the `cgltf` library.

1.2 Motivation

The ability to easily store and render massive scenes and game worlds which players can edit at will allows for the evolution of many interesting and novel game dynamics. For example, in a real-time strategy game, players might design and build fortified bases or might dig through the terrain to launch a surprise attack on their opponent. A pre-packaged software library which allows developers to quickly add such a feature to their game could save game studios considerable amounts of time and money.

When compared to polygon meshes, voxels have some interesting advantages for game developers wishing to build large, detailed environments which players can edit at will. Firstly, it is quite straightforward to edit an object composed of voxels; one simply edits the voxel data in the same fashion as pixels are edited within an image.

It is also very simple to procedurally generate voxels since they are just points

in 3D space. For example, a Perlin Noise function may be used to generate a field of voxels resembling realistic terrain features (this is the approach used by *Minecraft* to generate its game levels [9]). This property of voxels saves huge amounts of time compared to hand-sculpting landscapes using polygon meshes.

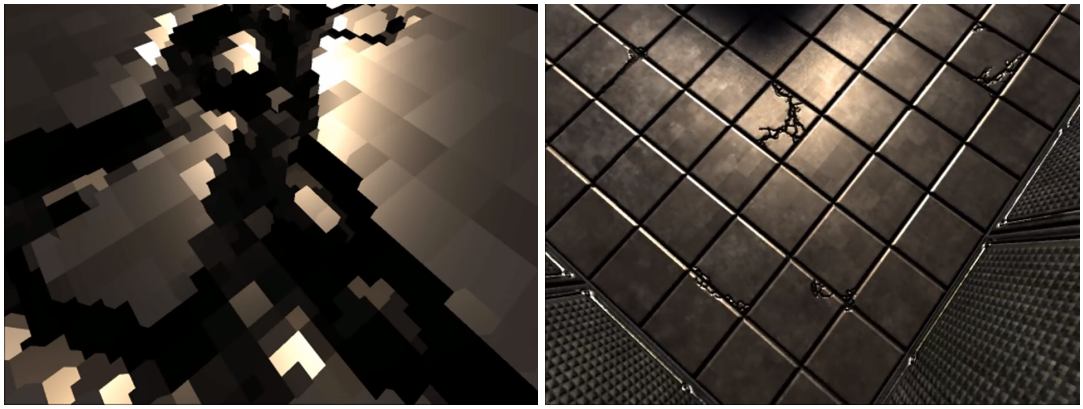

Figure 1.3: Two screenshots of a voxel engine by NewSoundGames[25]. The image on the left shows a zoomed-in view of the floor crack shown in the rightmost image

By storing voxels in a space-efficient data structure, such as a Sparse Voxel Octree, it is possible to create highly detailed scenes. Figure 1.3 shows a voxel engine capable of representing surface detail at sub-millimetre precision. This paves the way for innovative software solutions beyond games; 3D modellers might use this technique to build very high resolution models, or medical imaging devices might use this technique to produce extremely detailed images.

Voxel ray-casting also comes with some desirable advantages. Since a ray is fired for every pixel on the screen, geometry which does not appear within the camera's field of view is never processed. This means that frustum culling is effectively free with ray casting. Ray-casting also belongs to a class of problems termed *embarrassingly parallel*, meaning that it is a problem extremely well suited to parallelisation. This means that ray-casting is an excellent candidate for general-purpose GPU (GPGPU) programming. In this project, I utilise an OpenGL compute shader to perform the voxel rendering.

Finally, this project serves as an excellent vehicle to learn about modern high-performance rendering techniques and GPU programming; two areas of computer science that have always interested me.

Figure 1.4: Bust of Serapis rendered with ray-casting. The 3D model used to produce this scene was sourced from the Morgan McGuide Computer Graphics Archive [20]

1.3 Objectives

The overarching goal of this project is to produce a lightweight SDK that will allow client developers to support very large, highly detailed virtual environments that can be edited in real-time. To achieve this, there are four main features that must be implemented:

1. Support for extremely detailed scenes. I intend to support this feature through the use of specialised data structures for voxel storage (see Chapter 3 for a discussion on voxel storage schemes). In concrete terms, the final artifact should be capable of rendering scenes to a voxel resolution of 1 millimetre.

2. Support for very large scenes. Again, this feature is largely dependent on the choice of data structure for voxel storage.

3. High performance. Fast rendering and responding quickly to user input is crucial in video games to maintain the illusion of fluid motion. Excluding client errors or exceptional circumstances (for example, the host machine does not support hardware accelerated rendering), *Sabre* should complete all rendering operations within 33 milliseconds. This corresponds roughly to a frame-rate of thirty frames-per-second. Milliseconds-per-frame can be measured using Nvidia's Nsight graphics debugger (see section 2.1.3).

4. Allow arbitrary modifications to the scene at run-time.

Chapter 2: Research

Voxel technology is currently an area of active research within the computer graphics community — a Google Scholar search for the term "voxel rendering" turns up nearly 80,000 results[1] — so there is no shortage of literature to draw from.

2.1 Technologies

2.1.1 Programming Language

The choice of programming language and environment has an enormous impact on the project's life cycle. Since this project is designed for use in interactive video game engines, a language supporting compilation to native code was mandatory. Additionally, as this project will need to carefully orchestrate data flow between the CPU and GPU, support for control over low-level implementation details such as data alignment and memory layout was also required.

With these concerns in mind, I evaluated three programming languages: C, C++ and Rust. Though Rust boasts some very attractive features such as guaranteed memory safety and a powerful type system, it is still a very new programming language, with a very small userbase in the commercial game programming community. C++ is a strong candidate, however it is generally more difficult to integrate a C++ class library with other languages than a pure C function library because many languages (Lua, for example) expect foreign code to conform to a C API. C has the advantage of simplicity, but I would prefer to avail of the better type-checking built into most C++ compilers for this project.

After evaluating each of the above programming languages, I have chosen to implement this project in C++, but exporting a C API. This means that

[1] *As of January 6th, 2020*

the library code should be usable from other programming languages which expect a C API, while still allowing me to take advantage of C++'s more ergonomic features, such as raw string literals and operator overloading.

2.1.2 Graphics API

An essential component of any high-performance graphics application is the API used to issue GPU instructions. This is an important decision to make for the project as the code required to perform some operation may look considerably different between separate graphics APIs, so the choice of API has a large impact on the final codebase.

Excluding manufacturer-specific options, there are four main graphics APIs used to write high-performance GPU code today:

- OpenGL, a cross-platform and community-driven API originally developed by Silicon Graphics Inc. and now managed by the Khronos Group. Modern versions of the OpenGL API support advanced low-level techniques such as compute shaders and shader storage buffers required to render large amounts of voxels on the GPU. In OpenGL the graphics card may be programmed using GLSL, a high level programming language with special support for computer graphics operations.
- Direct3D (D3D), a proprietary graphics API developed by Microsoft. Part of the DirectX family of SDKs, D3D offers a high-level API with features roughly comparable to OpenGL along with its own GPU programming language, HLSL. As it is controlled entirely by Microsoft, Direct3D is limited to the Windows and XBox platforms.
- Vulkan, a cross-platform API recently introduced by the Khronos group in 2016 as an alternative to OpenGL. The primary motivation for introducing a new API was to provide application programmers more fine-grained control over the GPU's actions, and to take better advantage of the massively parallel nature of GPU systems . Vulkan allows for extremely granular control over an application's resource usage, though this requires significantly more management code to be written. Vulkan shaders are compiled offline into an assembly-like intermediate representation called SPIR-V. As such, Vulkan shaders may be written in either GLSL or HLSL but an appropriate compiler is needed to transform the shader source into SPIR-V assembly.
- Metal, a proprietary API developed by Apple Inc. Metal was introduced

in 2014 for much the same reasons as Vulkan, and is quite similar in design. Metal uses its own shading language called "the Metal shading language".

In the above options, there is a clear distinction between OpenGL and Direct3D, and Vulkan and Metal. These latter two APIs were designed much later than OpenGL and Direct3D, and more closely resemble how modern GPUs function. As such, they can potentially provide much greater performance and control over GPU resources, but at the cost of code complexity [29].

When selecting a graphics API for use in this project, I based my choice primarily on how familiar I was already with each API. Since I was already familiar with modern OpenGL programming, and OpenGL code can run on more platforms than Direct3D code, I chose to use OpenGL as the graphics API for this project. Although I would have liked to explore the newer Vulkan API, I do not have easy access to Vulkan-compatible hardware.

2.1.3 Tools

Since I was developing primarily on the Windows operating system, I chose to use Microsoft WinDBG as my debugger. WinDBG is a source-level debugger which supports custom scripting, disassembly and time-travel debugging.

Traditional debuggers can only examine code running on the CPU. As this project relies heavily on GPU code to render the voxel scene, a tool capable of debugging GPU code was required. Graphics debuggers are more closely tied to the underlying hardware than CPU debuggers; NVidia and ATI both offer products which only work on the respective manufacturer's hardware. Since I own a NVidia graphics card, I chose NVidia's Nsight Graphics as my graphics debugger.

To streamline development across multiple machines, and to have safe, reliable code storage I chose Git as my version control system, with the project being hosted on GitLab.com.

2.2 Mathematics

Since one of my primary goals for this project is to become more proficient with 3D game engine programming, I have decided to write the entirety of the mathematics code myself. Having some previous experience in this task already, I began by porting several maths utilities I had previously written

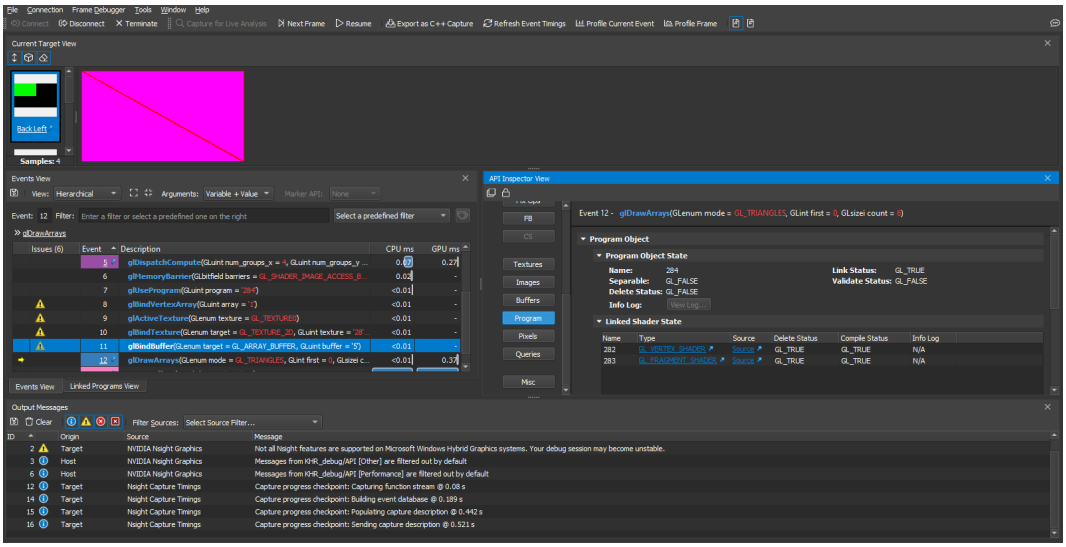

Figure 2.1: A screenshot of the viewer application running under Nsight Graphics

in C++ to this project. This ad-hoc math library contains some very simple vector operations (dot product, cross product, normalisation formula), some matrix operations (vector-matrix product, matrix-matrix product) and some miscellaneous functions specifically related to graphics programming such as the perspective projection formula.

My primary reference for mathematics was Lengyel's *Foundations of Game Engine Development, Volume 1: Mathematics* [16]. I also relied heavily on my Computer Graphics II course notes for developing the perspective projection code [10].

I also utilised quaternions for handling camera rotations in the prototype renderer, and I plan to do the same in the final viewer application. A quaternion can encode a rotation more efficiently than a matrix because it only requires four floating-point numbers to store the rotation as opposed to the sixteen required by a matrix. I used Van Verth's excellent overview to gain an understanding of the theory behind quaternion rotation, in particular the quaternion multiplication formula and how to rotate a vector by a quaternion [27].

2.3 Data Storage

When starting this project, I knew I would require a data structure capable of storing massive amounts of voxel data in an efficient form. Initially, I considered classical pointer-based Octrees. Octrees are a type of hierarchical data structure used to efficiently represent 3D space by discarding large empty regions. They were first introduced by Meagher in 1980 [22]. In their most

basic form, octrees use eight child-pointers to link parent and child nodes together. Laine and Karras improved on this design by encoding child data inside parent nodes; their design uses eight bit masks to determine whether or not a particular node has children, and if its children are leaf nodes. This removes the need to store leaf nodes at all.

2.4 Rendering Methods

Techniques available for rendering voxel data can be split into two broad categories: polygon-based methods and volumetric methods. Polygon-based methods represent the scene to be rendered as a collection of voxels in memory, but convert it to a polygon mesh before rendering. Once the scene has been converted into a polygon mesh, it can then be rendered in the same way as any other polygon. There are many algorithms available for transforming a field of voxels into a polygon mesh, one of which I initially planned to use in this project: Marching Cubes [17].

Though I considered using polygon rendering for this project, I decided not to proceed with it because the rendered polygon meshes will inherit all of the limitations of existing polygon geometry while simultaneously incurring the memory overhead of voxel storage. Using polygon rendering would require re-creating the polygon mesh of a voxel object every time it was changed. Instead I chose to use *ray-casting* for this project — a volumetric rendering approach which operates by shooting a ray into the voxel scene and examining all voxels intersected by the ray. The output image is produced by casting a ray for each pixel on the screen.

Laine and Karras introduce a voxel ray-casting algorithm in their *Efficient sparse voxel octrees* paper [15]. I have based my GLSL code on their approach.

2.4.1 OpenGL Compute Shaders

Since ray-casting is a highly parallelisable problem, it is well-suited to GPUs. I chose to use OpenGL compute shaders to perform the ray-casting due to the fact that I was already familiar with the GLSL programming language. To learn the specifics of compute shader programming, I began by reading an excellent overview by Mike Bailey [4]. I also used the official OpenGL reference pages to look up specific API details while programming [13].

To set up a proper rendering pipeline, I used the approach detailed by Burjack in the online tutorial article *Ray tracing with OpenGL Compute Shaders*

(Part I) [6]. This involved first creating an OpenGL image object to represent the output image. For each square "tile" of some small number of pixels, a compute shader is executed to fire a ray into the scene and perform the actual ray-casting computations. If the ray intersects voxel geometry, the tile is filled with colour. The final image is produced by executing the same compute shader many times for all of the output image's constituent tiles. Finally, this image is rendered onto a quadrilateral polygon mesh to produce the scene render.

2.4.2 Ray-Box Intersection

A crucial component of any ray-casting engine is how to determine when a ray intersects a voxel. I began by implementing Tavian Barnes' intersection algorithm [5] in GLSL. Some time later, I came across another version of the same algorithm by Majercik et al. [19] which uses GLSL's built-in vector math functions to more concisely express the same operation.

2.5 Polygon Renderer Prototype

The earliest development work on this project began with a prototype voxel renderer which transformed voxel data into polygon meshes using Lorensen and Cline's Marching Cubes algorithm. This prototype allowed me to get familiar with low-level OpenGL programming.

This prototype showcases one of the core strengths of voxel-based technology: the ability to automatically generate geometry. Figure 2.2 shows a sphere built entirely out of voxels. This sphere was generated by filling all voxels which satisfied the following implicit surface function:

$$x^2 + y^2 + z^2 - r^2 = 0$$

The remaining voxels are considered "air" or empty.

This prototype stored all voxel data in a simple array, with individual voxels addressed by their 3D coordinates. Albeit simple, this approach was, fundamentally, a poor choice for the final implementation because of its poor scalability. Storing all of the voxel data in a single flat array places a hard limit on the number of voxels available to represent the scene.

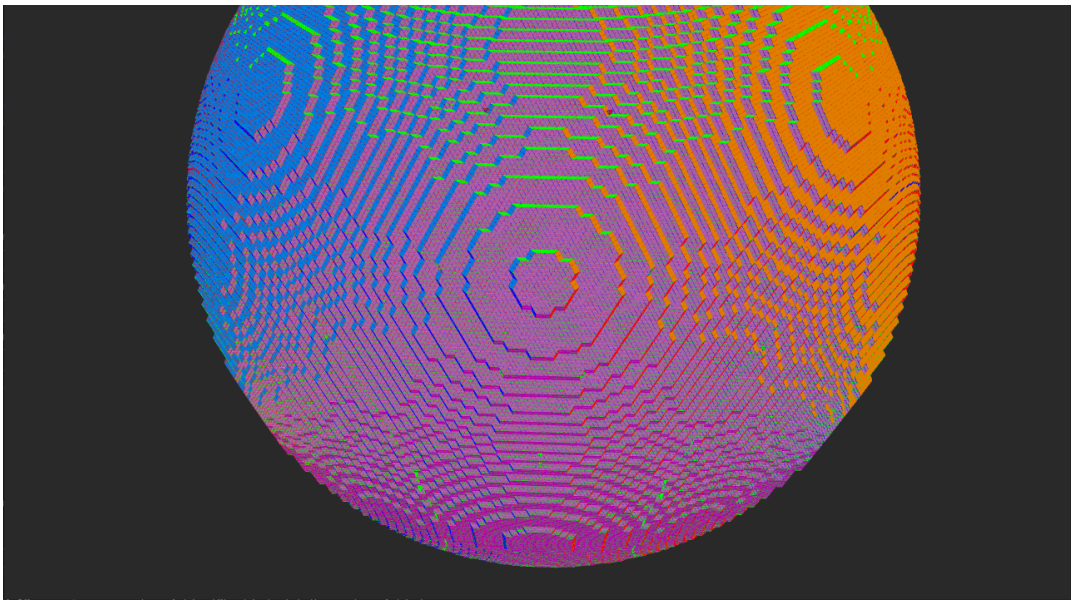

Figure 2.2: A screenshot from the prototype polygon renderer showing a sphere built from voxels

Chapter 3: Voxel Data Storage

All voxel engines must address the question of how to store and process voxel data. For any reasonably large scene with a high level of detail, the number of voxels required to represent such a scene can grow very large. Implementations must also devise a strategy for handling large, empty regions of space; an ideal solution would not store voxels that do not contain interesting geometry.

The most basic voxel storage strategy is to store each individual voxel in a flat array. This approach was used by *Minecraft* [23] [1]. Although this scheme has the advantage of being very simple to implement, and provides $O(1)$ voxel access and update times, the amount of memory required to store the voxel world is linear in the number of voxels. This restricts games using this technique to a low resolution "blocky" style. To represent more detailed worlds, some way of compressing voxel data is required.

3.1 Octrees

The octree, first developed in 1980 by Meagher [22], is a hierarchical data structure used to efficiently partition three-dimensional space. An octree begins as a cube encompassing all of the available world space with dimension d. This cube is then recursively partitioned into eight identical child cubes of dimension $\frac{d}{2}$ (see figure 3.1 for an example). Children are only themselves partitioned if geometry is contained within their bounds. By only storing voxels which contain real geometry, octrees allow large swathes of empty voxels to be cut away, considerably reducing the amount of memory required to store a voxel scene.

In classical octrees, each octree node contains eight pointers to it's children, with leaf nodes having eight null child pointers. While this structure is more

[1] Technically, Minecraft uses a paged array of flat arrays termed "chunks". Chunks are paged in and out depending on where the player travels in the game world.

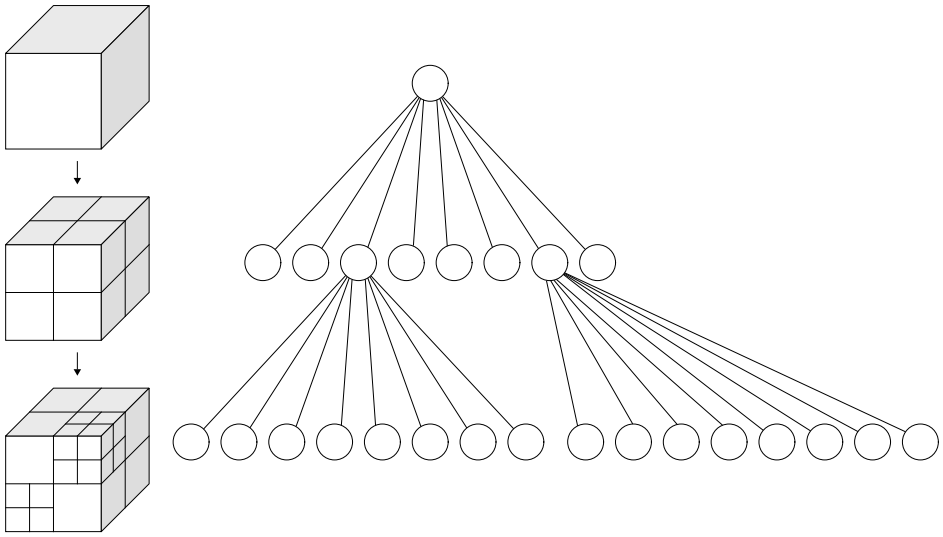

Figure 3.1: Example of an octree. Sourced from Wikipedia under the CC-BY-SA license

space-efficient than flat arrays, there is still room for improvement. On a 64-bit architecture with 8 byte pointers each octree node would consume 64 bytes of storage just for the child pointers. Laine and Karras introduced the *Sparse Voxel Octree* in 2010 [15] as an improvement over traditional octrees.

3.2 The Sparse Voxel Octree

This project stores scene voxel data in a Sparse Voxel Octree (SVO) data structure. Unlike traditional octrees, SVOs do not store eight child pointers for each node. Instead, two eight-bit masks are stored in each node; one to indicate that there is geometry resident in each child node corresponding to each bit position, and another to indicate whether or not a particular child is a leaf. This means that leaf nodes need not be stored at all, reducing the storage space required for a voxel scene even further than regular octrees.

The implementation of the sparse voxel octree in this project closely follows the ideas laid down by Laine and Karras. Throughout the project, the following convention is used for numbering cube octants: 0 to 7 increasing in X, then Y, then Z. Figure 3.2 shows an example.

At the highest level, the SVO tree data structure used to store the entire scene in *Sabre* consists of a list of fixed-size blocks of octree nodes. Only non-leaf and non-empty nodes are actually stored; Laine and Karras' methodology stores all of the information required to construct a leaf node in its parent.

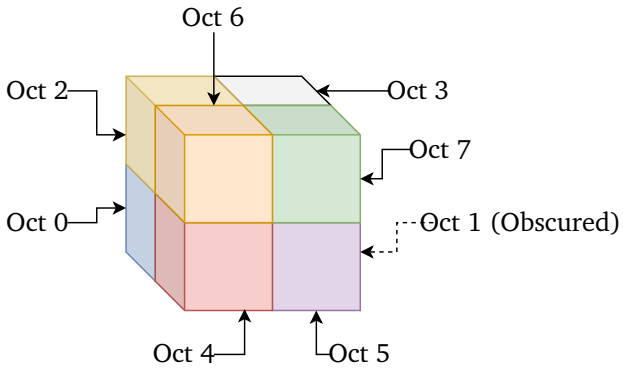

Figure 3.2: Convention used when numbering octants. Octant 1 is to the right of octant 0

3.2.1 SVO Blocks

Since *Sabre* is designed to handle very large voxel scenes, it would be infeasible to attempt to store all of the SVO nodes required for a scene contiguously in memory. Instead, *Sabre* uses a singly-linked list of `svo_block` structures (see listing 1). Blocks are fixed-size chunks of memory that store a predetermined number of nodes, controlled by the `SVO_ENTRIES_PER_BLOCK` constant.

```
struct svo_block
{
    usize       NextFreeSlot;
    svo_block*  Prev;
    svo_far_ptr FarPtrs[SVO_FAR_PTRS_PER_BLOCK];
    svo_node    Entries[SVO_ENTRIES_PER_BLOCK];
};
```

Listing 1: SVO block structure. `SVO_ENTRIES_PER_BLOCK` is a constant that determines how many entries are stored within each block

Blocks function similarly to stacks. The `NextFreeSlot` field is the stack pointer, denoting an offset to the next available memory address that can hold a new `svo_node` structure. In the case that all slots are used up, a new block is transparently allocated and the two are linked together via the `Prev` pointer.

This stack-based approach is used for performance reasons; allocating each node entry on the heap would introduce the considerable overhead of a call to `malloc` or similar every time a new node was inserted (these allocations would also have to be freed one at a time). With millions of nodes, this strategy would be unacceptably slow. With the block-based approach, a call to the system allocator is only required each time a new block is required, and

individual node allocations are very fast as they only require (in most cases) incrementing `NextFreeSlot`.

SVO blocks also store a fixed number of `far_ptr` instances (see listing 3). These are used in case child references for an individual node overrun block boundaries.

3.2.2 SVO Nodes

All voxel geometry in *Sabre* is fundamentally stored in a `svo_node` structure (see listing 2 and figure 3.3). The `OccupiedMask` field is used to test if this particular node contains any geometry at all, where each bit at position i determines if there is geometry resident in the i^{th} child octant of this node. Each bit in the `LeafMask` field indicates whether or not the i^{th} child octant is a leaf node — this means that storage of leaf nodes becomes unnecessary as leaf nodes can be inferred by looking at the leaf mask of their parent.

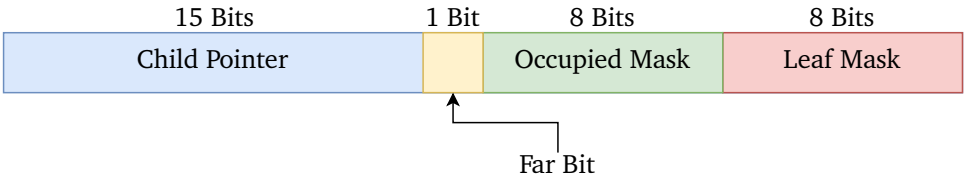

Figure 3.3: Data layout inside an individual 32-bit SVO entry

```
union alignas(4) svo_node
{
    struct
    {
        uint8_t  LeafMask;
        uint8_t  OccupiedMask;
        uint16_t ChildPtr;
    };

    uint32_t Packed;
};
```

Listing 2: Source of the `svo_node` structure

Leaf Mask

The `LeafMask` field is an 8-bit unsigned integer used to determine which of a node's sub-octants are to be considered leaf voxels (i.e. that there are no

further child nodes in that octant). Each bit position i in the mask corresponds to the i^{th} child octant.

Occupied Mask

The `OccupiedMask` field is an 8-bit unsigned integer used to determine which of a node's sub-octants have geometry contained within the bounds of their enclosing cube. An occupied octant may or may not need to be subdivided further; the `LeafMask` field encodes this information. Each bit position i in the mask corresponds to the i^{th} child octant.

Child Pointer

The `ChildPtr` field is a 15-bit unsigned integer representing the offset of a node's first non-leaf child in it's SVO block (see section 3.2.1). In certain cases, a node's first child may end up in a different block, in which case this field instead stores a far pointer offset. This far pointer offset points to the far pointer, in the *parent* node's block, which contains the location of the block containing the child node and the offset of the child node within that block.

```
struct sbr_far_ptr
{
    uint32_t BlkIndex;
    uint32_t NodeOffset;
};
```

Listing 3: Source of the `far_ptr` structure. This is used to support node references that span blocks; the `Block` pointer points to the child node's containing block and `BlockIndex` is the offset of the child node inside the containing block

3.2.3 Far Pointers

Given a large enough octree, it is likely that at some point a node's child will end up allocated in a different block. If this happens, the node's child pointer must be converted into a far pointer (see listing 3). Far pointers store a block index, denoting the block a node's child resides in, and a block offset, denoting where in that block the child is stored.

Far pointers are stored and allocated in almost exactly the same way as SVO nodes; the `FarPtrs` member of the block struct stores a stack of far pointers, with the head of the stack represented by `NextFarPtrSlot`. New far pointers are allocated by simply incrementing `NextFarPtrSlot`.

If it has been determined that a node is to be allocated outside its parent's block, then that node must have its far bit flag set (see figure 3.3). A new far pointer is then allocated from the parent node's block and the parent node's child pointer bits are used to store an index into the far pointers array.

Note that far pointers are only required for the first of a node's children. This is because children are always stored sequentially; if a node's children were to straddle a block boundary, there would be no need for a far pointer because the block containing the other children is guaranteed to be the parent block's `Next` pointer.

3.3 SVO Construction

Clients can create SVO scenes by calling the `SBR_CreateScene` function, the prototype of which is described in listing 4. The library relies on client-supplied *sampler* functions to determine the shape and data attributes of the scene. Clients can also customise the level of detail (corresponding to the octree's depth) and the scene scale.

```
extern sbr_svo*
SBR_CreateScene(uint32_t ScaleExponent,
                uint32_t MaxDepth,
                shape_sampler* ShapeSampler,
                data_sampler* NormalSampler,
                data_sampler* ColourSampler);
```

Listing 4: Prototype for the SVO scene construction function. The sampler parameters provide occupancy, normal and colour data respectively

3.3.1 Sampler Functions

The library builds SVO scenes by recursively calling a client-supplied function pointer which returns a boolean indicating whether or not there is geometry resident between two opposing points. Voxel attribute data, such as colours and normals, are created in a similar manner, with the user-defined functions returning the attribute value for a particular leaf voxel. Allowing the client to specify their own sampler functions affords a significant degree of flexibility and control to the user — in the distribution package, this method is used to construct both generated scenes (the "Sphere" scene) and imported mesh scenes.

The client provides these sampler functions by creating instances of `_sampler` structures; `shape_sampler` structs provide occupancy data, while `data_sampler` structs provide colour and normal data. These structures are detailed in listing 5

```
typedef sbrv3 (*data_fn)(sbrv3, const sbr_svo* const, const void* const);
typedef bool (*shape_fn)(sbrv3, sbrv3, const sbr_svo* const, const void* const);

struct shape_sampler
{
    shape_fn SamplerFn;
    const void* const UserData;
};

struct data_sampler
{
    data_fn SamplerFn;
    const void* const UserData;
};
```

Listing 5: The `sampler` types. The `UserData` pointer is supplied by the clients at sampler initialisation and is read-only.

Along with providing their own function pointers for samplers, client developers can also store a pointer to arbitrary data in the sampler struct, which is then made available to the sampler functions at SVO creation time. This exists to allow clients to interact with external data sources from within the sampler functions; for example, the polygon mesh importer stores the mesh triangle index in the `UserData` pointer of its shape sampler. The user data pointers are read-only, helping to enforce the concept of the sampler functions as "pure" functions that do not affect the external environment.

There is some contention surrounding the use of void pointers in the C++ community, and not without good reason — void pointers throw away valuable type information, and rely instead on unsafe casting to transform void pointers into usable types. I considered several safer alternatives for use in this project, namely providing the user data as a template parameter or using functor style classes as the samplers. Ultimately, I decided that the inclusion of these practices would break C compatibility of the API, a trait I was not willing to sacrifice.

3.3.2 Determining the Scene Scale

The client-supplied parameters `ScaleExponent` and `MaxDepth` together control the size of the world region represented by the SVO, and the size of the individual voxels.

The scale of the world (i.e. the dimensions of the root octree node) are calculated by computing $2^{\texttt{ScaleExponent}}$. This method is used in place of a more straightforward approach where the client supplies the root node dimensions directly for two reasons: first, it ensures that all dimensions throughout the octree will be powers of two, meaning that efficient bit-shift operations can be used in tree construction and rendering. Second, it allows for much larger and more detailed scenes to be created.

The `MaxDepth` parameter controls the maximum depth of the octree; larger values will produce more detailed scenes with smaller leaf voxels, but at the cost of increased construction time.

If `ScaleExponent` is less than `MaxDepth`, voxel dimensions will start to become fractional values. This is a problem, since the rendering and construction code requires all voxel dimensions be integers to make use of fast integer math operations. We constrain all voxel dimensions to integers by introducing a scale bias when `ScaleExponent` is less than `MaxDepth`. Coordinates in (possibly fractional) world space must first be transformed by this scale bias before they can be applied to the octree scene. Likewise, point coordinate obtained from traversing the octree must have the inverse of this scale bias applied before they can be useful in world space.

The scale bias is calculated by subtracting `ScaleExponent` from `MaxDepth`. For example consider an octree with `ScaleExponent` 5 (meaning the maximum dimensions of the root node are $2^5 = 32$) and a `MaxDepth` of 7. Without the scale bias, this octree would produce the following voxel dimensions, in order: $16, 8, 4, 2, 1, 0.5, 0.25$. To eliminate the fractional value, we must introduce a multiplication by $2^{7-5} = 2^2 = 4$.

3.3.3 Building the SVO

Once the scene scale has been determined, the library can begin constructing the octree. Construction begins at depth 1, where the library determines which of the root's child octants need to be further subdivided by calling the `shape_sampler` function pointer provided by the client. This process continues recursively until the sampler function no longer evaluates to `true` when

passed a child octant's corner coordinates, or `MaxDepth` is reached.

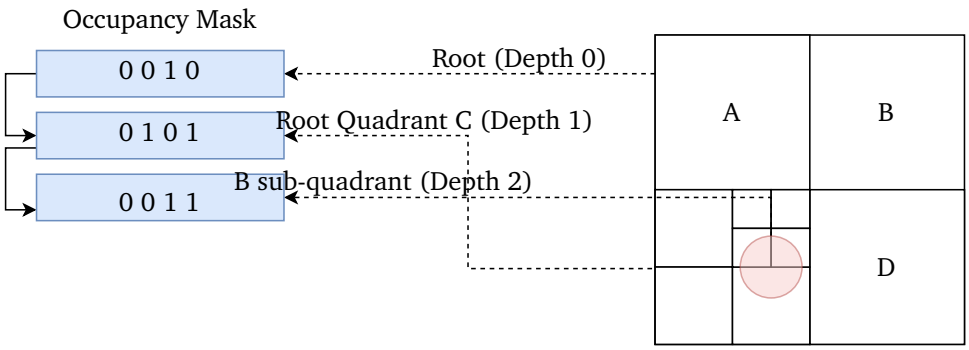

Figure 3.4: Layout of nodes inside a block. In memory, nodes are organised in a depth-first fashion. In this example, A, B and C are leaf nodes.

Chapter 4: Dynamic Edits

An important feature of voxel-based engines is their ability to make arbitrary modifications to the scene at runtime. *Minecraft* is well-known for this feature, allowing players to insert and delete bricks at any point to build up their own creations, like a virtual LEGO set.

Since this project uses an SVO data structure to store voxel data, insert and delete operations become more complex, requiring re-building of tree levels in certain cases.

4.1 Client API

The core library supports scene modifications through the functions `SBR_InsertVoxel` and `SBR_DeleteVoxel`. The prototypes for these functions are shown in listing 6. Both functions take a pointer to the scene to be modified and a vector specifying the position at which the modification should happen as parameters. These API functions are not thread safe and may allocate memory (though only in the comparatively rare case that there is no space inside the last block of the entire tree data structure).

```
// Delete voxel at position `VoxelP`
void SBR_DeleteVoxel(sbr_svo* Tree, sbrv3 VoxelP);

// Insert voxel at position `VoxelP`
void SBR_InsertVoxel(sbr_svo* Tree, sbrv3 VoxelP);
```

Listing 6: Prototypes for the insert/delete functions exposed to clients

In addition to these functions, the helper functions `SBR_GetNearestFreeSlot` and `SBR_GetNearestLeafSlot` can be used to compute the `VoxelP` parameters more intuitively. These utility functions calculate the closet valid insert/delete position from a point along a given direction. They operate by firing a ray, constructed from the given origin and direction parameters, into the scene and

returning the first voxel this ray hits (or, in the case of `GetNearestFreeSlot`, the last empty octant the ray passed through before it hit a voxel).

4.2 Inserting Voxels

To insert a leaf voxel at some position P, we must first locate the deepest node containing P. This is accomplished by recursively selecting the child octant containing P and then jumping to that corresponding child, starting at the root node. The deepest node is reached when there is no child node corresponding to the octant containing P. If the deepest node happens to be at the maximum tree depth, then we can simply mark the octant containing P as a leaf. The more common case is that the deepest node is not at the correct depth, so a new branch must be created in the tree.

4.2.1 Determining the Child Octant

This project uses a consistent scheme for assigning 3-bit codes to octants, described by figure 3.2; these codes range from 0 to 7. The octant code for a position P with respect to a given origin can be determined by first partitioning space into eight regions along the X, Y and Z Cartesian planes. We observe that, when formatted as zyx, the bits in an octant code corresponding to a particular axis are set when a point lies on the positive side of that axis plane.

As an example, consider the point $(1, 0, 0)$ which lies on the positive side of the X plane with respect to the origin point $(0, 0, 0)$. Since this point lies on the positive side of the X plane, the x bit should be set in its octant code. This process is repeated for the Y and Z axes to build up the complete octant code. Listing 7 shows how this procedure is implemented in C code.

```
static inline svo_oct
GetOctantForPosition(sbrv3 P, sbrv3 ParentCentre)
{
    sbrv3u G = sbrv3u(GreaterThan(P, ParentCentreP));
    return (svo_oct) (G.X + G.Y*2 + G.Z*4);
}
```

Listing 7: Computing the child octant number for a position `P` relative to the parent octant centre. The `GreaterThan` function produces a vector of either 1 or 0 values (corresponding to the boolean results of a component-wise comparison). These integer values can then be combined into an octant number ranging from 0 to 7

4.2.2 Creating a New Branch

If the deepest node we can reach is not at the maximum tree depth, then a new branch must be created, reaching from the deepest node we were able to reach to the maximum depth. To do this, a new parent node must first be inserted to function as the root of this branch — see section 4.4 for details on this process. Next, we must continuously allocate new nodes, setting the octant containing P as occupied at each one, until the maximum depth is reached, at which point a leaf node is inserted. Figure 4.1 illustrates an example of this scenario, with the blue level indicating the reallocated sibling node, and the green levels indicating the newly-created nodes.

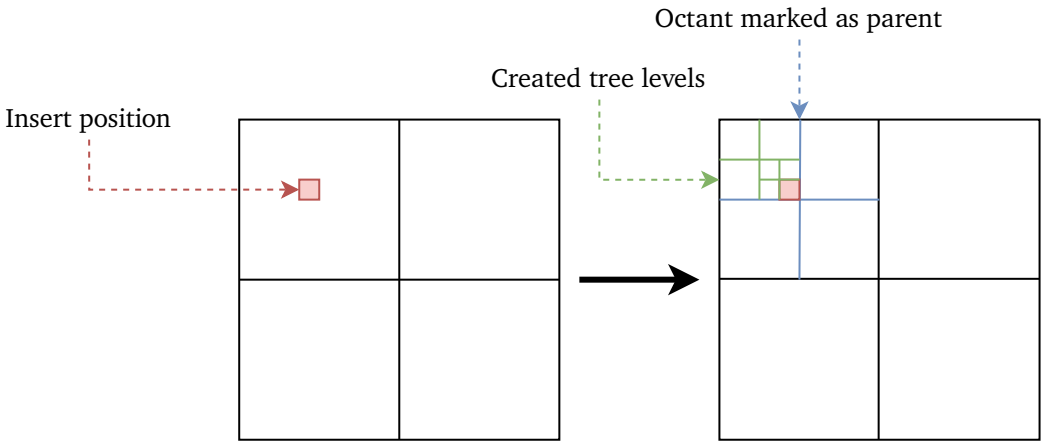

Figure 4.1: State of the SVO before (left) and after (right) a voxel has been inserted. Note the first inserted level (blue) and the levels following (green) which must be created and added into the tree

4.3 Deleting Voxels

The process for deleting a voxel is similar to the insertion process. The tree is descended to the deepest level or until a leaf voxel is encountered. If the traversal reaches the deepest level and the supplied position corresponds to a leaf voxel, the bits in the parent voxel's `LeafMask` and `OccupiedMask` corresponding to that voxel's octant are cleared.

If the traversal encounters a leaf voxel that is not at the deepest level, this leaf voxel must be recursively split into a parent voxel and seven leaf voxels one level down. Figure 4.2 shows an example of this situation. This situation also calls for reallocation of this newly-created branch's ancestor, in a similar manner to voxel insertions.

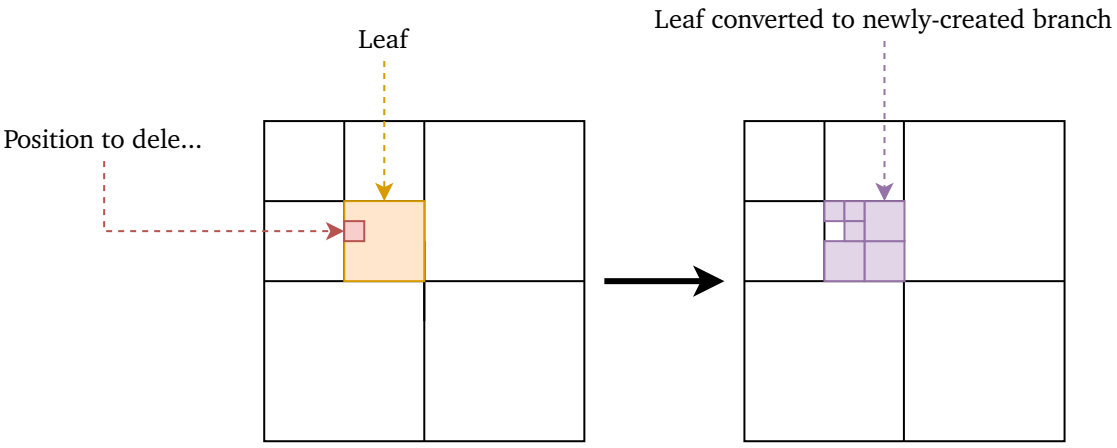

Figure 4.2: Example of a leaf voxel being recursively split up when it contains a to-be-deleted voxel.

4.4 Node Reallocation

Certain mutating operations require creating new branches through the tree (for example, inserting a very small voxel into a large empty node). Since sibling nodes are stored sequentially, inserting or deleting a branch requires inserting or deleting a sibling node at some level. To avoid a costly reorganisation of the entire block (or even worse, the entire tree), sibling modifications are implemented by copying the modified sibling array to the end of the block and updating the parent node's child pointer to this new location.

If there is no more space left at the end of the block, a new block must be allocated from system memory and linked to the tree. If this happens, care must be taken to ensure that any sibling node copied into the new block has their child pointer converted to a far pointer in the old block pointing into the new one.

This process can only occur once, since even though their locations have changed the siblings' child pointers remain valid, keeping the structure of the tree intact.

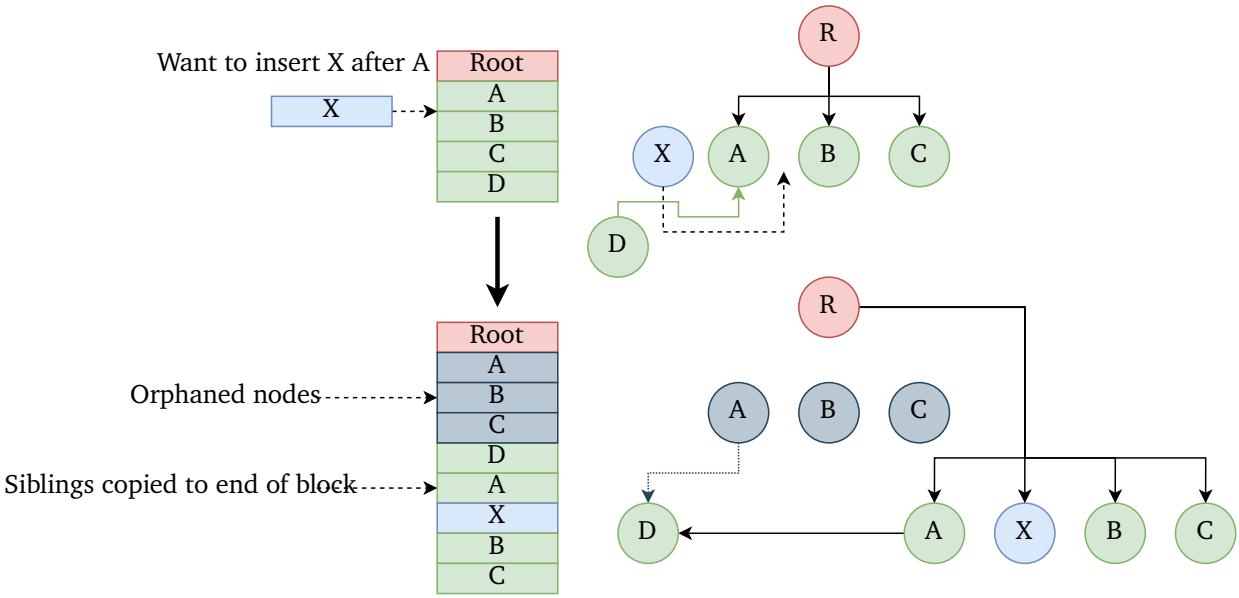

Figure 4.3: Illustration of the sibling reallocation process. Node `X` is to be inserted as a sibling to `A`. `A` and its current siblings are copied to the end of the SVO data block with `X` inserted at the correct place. Finally, the child pointer of `R` is updated to reflect the new location, leaving the previous copies of `A`, `B` and `C` orphaned. Note that node `D` remains unmoved since the child pointer from the copied `A` remains valid

Chapter 5: Importing Meshes

Although advancing all the time, tools for authoring voxel-based content are still in their infancy when compared to polygon mesh authoring tools. Additionally, no contemporary voxel content creation tools can export to the SVO format required by this project. Since it is likely that any client of this library could already have a significant number of existing polygon mesh assets, it would be helpful to have some way of automatically converting these assets into SVO scenes for use with this library.

Included with the product distribution is an optional module called `sabre_import` which provides support for converting polygon mesh files in the GLTF format into fully modifiable SVO scenes. There are several example meshes included with the distribution, sourced from the Morgan McGuire Computer Graphics Archive [20]. Figure 5.1 shows an example of a mesh voxelised at a high level of detail using this importer.

The polygon mesh loading API is exposed to clients through the `SBR_ImportGLBFile` function, which takes a file name and a max depth level as parameters.

5.1 The GLTF File Format

GLTF[1] is a relatively new asset storage file format developed by the Khronos group in 2015 [12]. GLTF files store a description of the data they contain (for example, how many vertices there are and what format these vertices are encoded in) in a JSON string. The actual binary data, containing the mesh vertices, colours, normals, etc. is stored directly after this JSON string (in the case of `.glb` files) or in a separate file (in the case of `.gltf` files).

I initially evaluated several 3D model file formats, such as the Stanford PLY format or the Wavefront OBJ format, before settling on GLTF for use in this project. The driving reason for picking GLTF was the fact that GLTF files store

[1] *GL Transmission Format*

Figure 5.1: The famous Stanford bunny model, voxelised to a 1024^3 resolution grid

the mesh data in a tightly packed binary format, drastrically reducing the file size compared to This project uses the `cgltf` C library [14] to load `.glb` files into a list of triangles which are then voxelised into the final SVO scene. I initially evaluated several other polygon file formats but settled on GLTF.

5.2 Importing Polygon Meshes

Once the client has called `SBR_ImportGLBFile`, the import process begins by loading the GLTF file with CGLTF. This results in a `cgltf_mesh*` handle, which can be used to load the data associated with the mesh file, such as the vertex positions. Vertices, colours and normals are then loaded into a large buffer of `tri_data` structs (described in listing 8). An index is then built mapping positions to colours, normals and boolean occupancy values. The purpose of this index is to provide a fast way for sampler functions to look up the occupancy or attribute values at a particular point, to facilitate efficient tree building.

```
struct tri3
{
    sbrv3 V0;
    sbrv3 V1;
    sbrv3 V2;
};

struct tri_data
{
    tri3 T;
    sbrv3 Normal;
    sbrv3 Colour;
};
```

Listing 8: The datatypes used to store mesh triangle data in this project. Each triangle has one associated colour and normal value.

5.2.1 Mesh Loading

Once the basic mesh has been loaded by CGLTF, the program then begins parsing the triangle data into a list of `tri_data` structs (listing 8). GLTF meshes can be composed of a variety of primitives; the importer in this library is only concerned with the `triangle` primitive type. For each primitive, there will usually also be a set of indices, possibly compressed, which describe the ordering of primitives. The first step in the actual mesh loading process is to decode these indices into a temporary buffer, as shown in listing 9.

```
cgltf_size IndexCount = Prim->indices->count;
uint32_t* IndexBuffer = (uint32_t*) malloc(IndexCount * sizeof(uint32_t));

// Copy the indices into the index buffer.
for (cgltf_size Elem = 0; Elem < Prim->indices->count; ++Elem)
{
    IndexBuffer[Elem] = (uint32_t) cgltf_accessor_read_index(Prim->indices, Elem);
}
```

Listing 9: Decoding the primitive indices into a temporary buffer. This buffer is freed after the positions for this particular primitive block have been loaded

Once the indices have been loaded, the mesh's `position` attributes must be loaded (note that here, the term "attribute" refers to a different CGLTF-specific concept than the term "attribute" elsewhere in this document). Position at-

tributes are first loaded into a temporary buffer, in the same manner as indices. The final step is to loop through every triplet of indices and construct a `tri3` instance using the corresponding position values. Triangle normals are computed here, since not all meshes may store the pre-computed normals and the computation has negligible effect on the overall import time. The end result of this process is a `tri_buffer` containing all the mesh triangles and their normals.

5.2.2 Building the Triangle Index

As described in section 3.3, SVOs are constructed using client-provided pure sampler functions, which provide the occupancy and attribute data for particular world points. In the mesh importer, it would be too expensive to have these sampler functions iterate through the entire triangle list for each invocation, so we first construct an index mapping colour, normal and occupancy values to points. The samplers, then, are reduced to simple, constant-time lookups.

In the interests of time-saving, these index data structures are implemented using the STL containers `std::unordered_map` (for the colour and normal indexes) and `std::unordered_set` (for the occupancy index). To build the index, each triangle in the triangle list is recursively checked against octree node bounds using a SIMD triangle intersector based on the Akenine-Möller triangle intersection formula [1]. Octree nodes that do intersect a triangle are added to the occupancy index. If this process reaches the octree leaf depth, the node centres are also added to the colour and normal indices.

5.2.3 Building the SVO

Finally, once the respective indices have been built, the SVO construction process is kicked off using the `SBR_CreateScene` function. The scene scale is determined by selecting the closest integer $\log 2$ of the maximum mesh dimension (conveniently stored by GLTF files).

The samplers supplied to the octree builder are, thanks to the index building process, extremely simple. Samplers need only check if the supplied position values have entries in the indexes, an operation defined to have amortized $O(1)$ complexity.

Chapter 6: Rendering

Contemporary polygon-based games rely almost exclusively on the specialised triangle rasterisation hardware found in modern GPUs to transform a list of triangles into an image. Unfortunately, since individual voxels do not carry any information about the scene's overall shape it is not possible to use this approach with voxels in their native format and, as previously discussed, it would be too memory-consuming to convert the voxel scene into a polygon representation prior to rendering.

Ray-casting is an alternative method of rendering well-suited to voxel data. Ray-casting operates by firing a ray into the voxel scene for every pixel of the screen. If the ray hits a leaf voxel, the pixel is coloured. Since this process is very computationally expensive, it is necessary to harness the processing power of modern graphics hardware to perform many ray-casts in parallel. In *Efficient Sparse Voxel Octrees*, Laine and Karras introduce an efficient algorithm for rendering sparse voxel octrees; the renderer written for this project is a GLSL implementation of this algorithm.

6.1 Renderer Pipeline

Figure 6.1 illustrates the high-level rendering architecture of this project. Voxel data is generated on the CPU and then transferred to the GPU, where it is rendered using a compute shader raycaster.

6.1.1 Compute Shader Background

The term *shader* refers to an executable program that runs on the GPU rather than the CPU. Originally applied to very simple programs that performed highly specific tasks such as calculating pixel colours, the term has since come to mean any program that executes on the GPU even if that program has nothing to do with computer graphics.

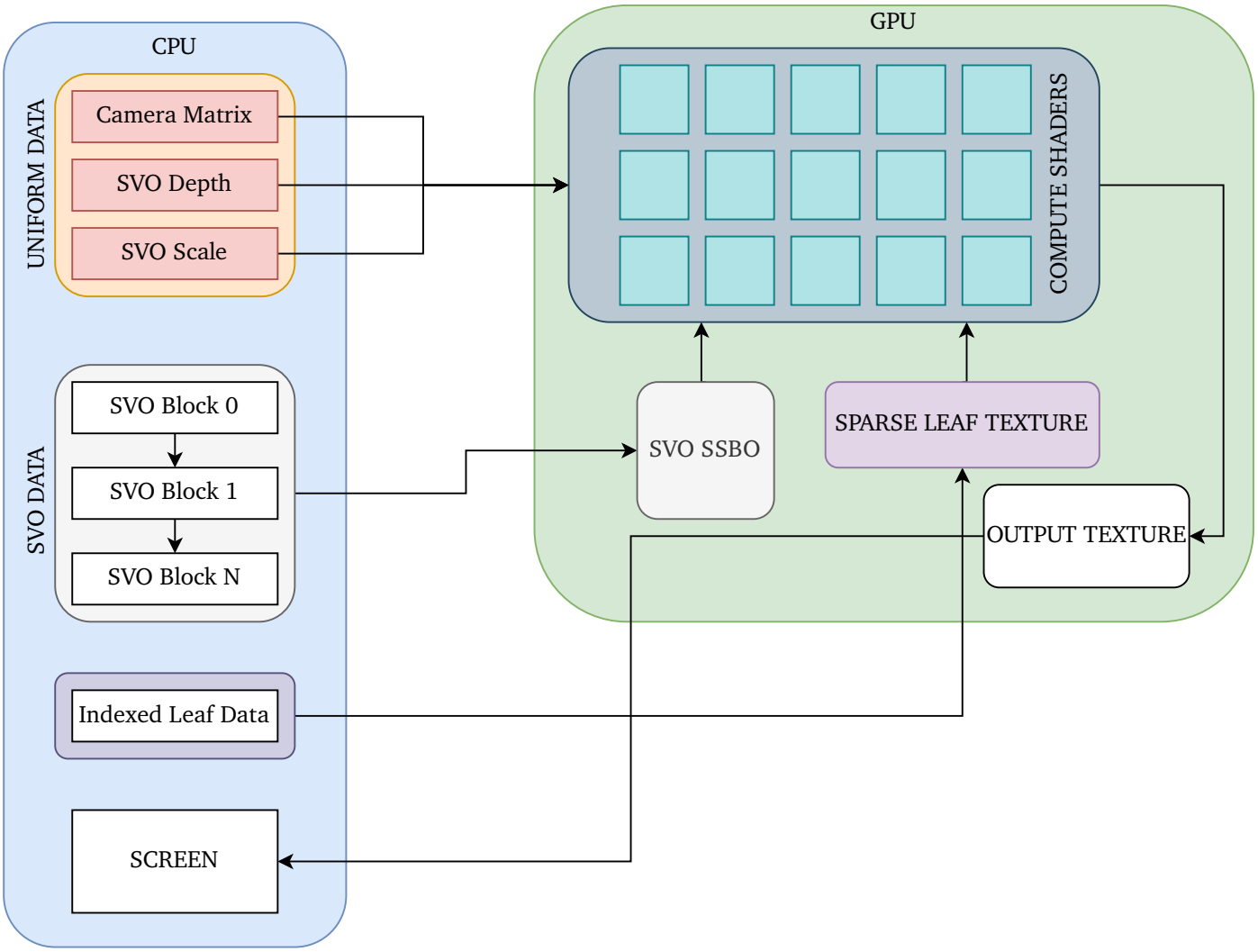

Figure 6.1: The rendering architecture

In OpenGL, compute shaders operate slightly differently to the graphics-specific vertex and fragment shaders. Shader invocations are organised into *work groups* — multi-dimensional "blocks" of shader execution units. For example, all the elements of a two-dimensional array may be processed in parallel by invoking a compute shader with a work group of the same dimension as the array. The GPU will execute the same compute shader for every work group item.

By representing the screen viewport as a two-dimensional array of texels, a compute shader may be used to compute all of the texel colours in parallel. This serves as the input of the raycasting algorithm.

In this project, the screen viewport is represented by an OpenGL texture object. This texture serves as the output of the renderer compute shader kernel.

6.1.2 Transferring Occupancy Data

Voxel occupancy data defines the layout of the sparse voxel octree, indicating whether or not a particular point in space contains a voxel. Voxel occupancy data begins life on the CPU, where it is stored in large blocks linked together by pointers (see section 3.2.1 for a treatment of CPU-side voxel data storage). In order to render this voxel data, these SVO blocks must be transferred into GPU accessible memory as fast as possible. In this project, SVO occupancy data is stored on the graphics card as large memory buffers called "Shader Storage Buffer Objects" (SSBOs) which are directly accessible from the ray-casting compute shader kernel. Two SSBOs are used to store the voxel octree data: one for the occupancy information, and one for the far pointers (see section 3.2.3).

The process of transferring the occupancy data is entirely opaque to clients of the library. Clients simply call the `SBR_CreateRenderData` function to obtain a `sbr_render_data` handle that can be used by the library to access these SSBOs at a later time. This is by design, since the less clients rely on OpenGL-specific implementation details the easier it becomes to port the software to other graphics APIs, Direct3D for example.

6.1.3 Pre-render setup

Before the system can begin actually rendering the SVO data, some preliminary setup work needs to take place each frame. This essentially consists of computing the view transform matrix and transferring this to the graphics card. The auxiliary struct `sbr_view_data` stores the camera orientation matrix and position vector and is later passed into the `SBR_DrawRenderData` function, which kicks off the render to the GPU.

6.1.4 Voxel Rendering

Once the pre-render setup has been completed, the compute shader raycasting kernel is executed for each pixel of the render output image. In this kernel, the ray's origin and direction are determined by first computing the world-space coordinates of the kernel's pixel on the near projection plane; figure 6.2 shows how these coordinates are calculated. The ray's direction is obtained by computing the normalised vector going from the eye through this projected pixel.

Section 6.3 gives comprehensive overview of the raycasting algorithm used

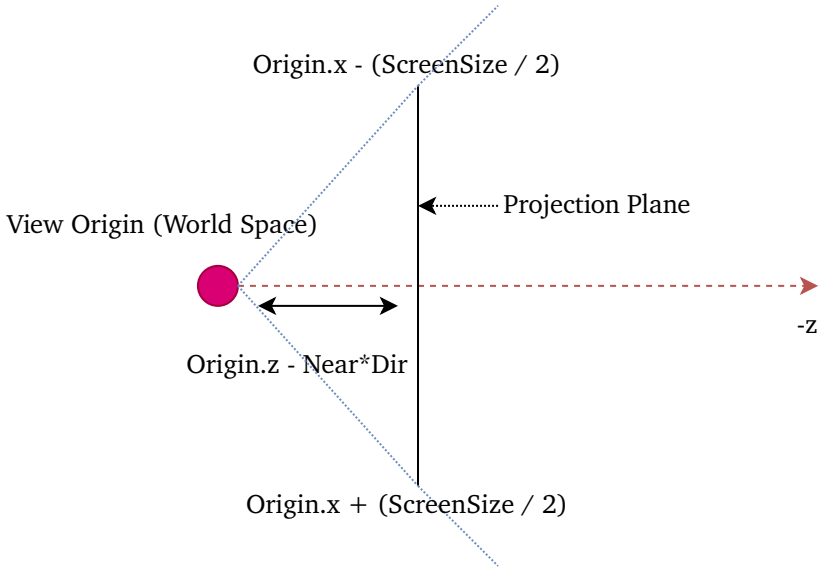

Figure 6.2: Illustration of how the screen projection plane is computed given the eye coordinates and view direction in world space

in this project, which is a GLSL implementation of the process described by Laine and Karras. During the main rendering loop, the raycasting compute shader is invoked. The GPU will execute one compute shader kernel for each pixel of the output image. Compute shaders are invoked with the OpenGL call `glDispatchCompute`.

6.1.5 Displaying the final image

Once the GPU has finished executing the compute shaders, the traditional polygon rendering pipeline is used to present the finished image to the screen. This involves covering the entire viewing window with a flat polygon and then attaching the rendered texture to this polygon. Extremely simple "no-op" fragment and vertex shaders are then used to present this image to the screen.

6.2 Ray-Box Intersection Algorithm

The SVO data structure organises space into a hierarchical cube grid. In order to traverse this grid and render leaf voxels it is necessary to determine where a particular camera ray intersects a cube in space.

Any point along a ray R can be expressed with the following equation:

$$R_p + tR_d$$

Where R_p is the ray origin, and R_d is the ray's direction. The scalar value t

indicates how far along the ray the point in question lies. When a ray intersects a three-dimensional cube, the t-values correspond to the intersection of the ray with the six planes that make up the cube.

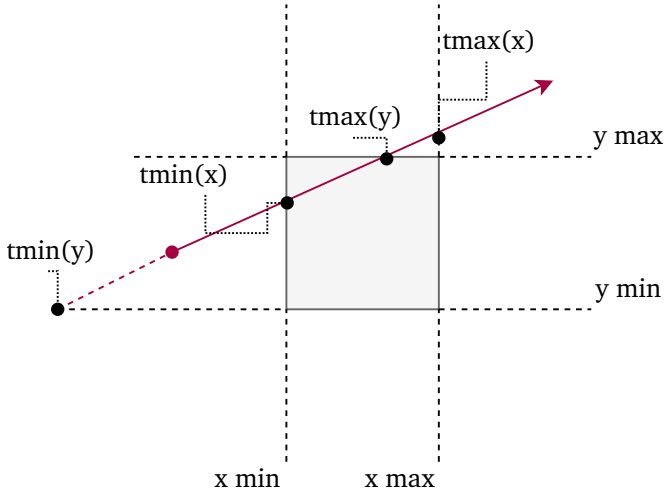

Figure 6.3: Illustration of the ray-cube intersection points on the six cube planes

This project uses Majercik et al's GLSL ray-box intersection code to very efficiently compute the t_{min} and t_{max} values for a given cube.

6.3 Raycasting Algorithm

Laine and Karras outlined the basic algorithm for SVO raycasting in their original paper. The rendering code for this project is based upon their conceptual algorithm.

The regularity of the SVO data structure allows very efficient raycasting of large voxel data sets because large regions of empty space can be quickly traversed. As defined by Laine and Karras, the voxel raycaster uses three primary operations: *Push* (traverse deeper into a parent voxel), *Advance* (process the next voxel of the same depth) and *Pop* (exit a parent voxel). These operations are shown in figure 6.4

6.3.1 Push - Entering a child voxel

When the raycaster encounters a non-empty voxel, it will first check if the voxel is a leaf. Leaf voxels do not require any further traversal and can be coloured immediately, terminating the ray for this pixel. If the voxel is occupied but not a leaf, we must descend into the SVO hierarchy and examine next node the ray enters — this is the *push* operation.

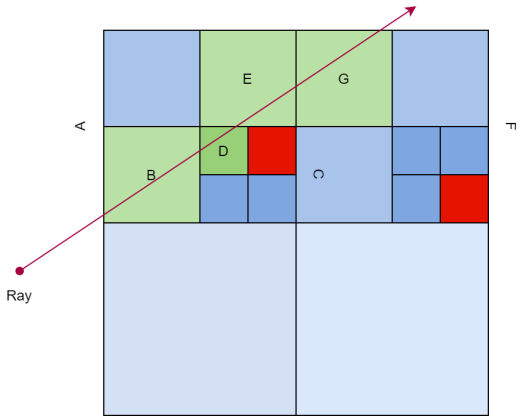

Figure 6.4: Illustration of a ray travelling through the octree. The traversal path is A, B, C, D, E, F, G

Since we may need to return to a voxel from deeper down in the tree (this is the *pop* operation, see section 6.3.2) we maintain a stack of parent voxels that can be restored if needed.

Determining the Child Octant

Determining which octant the ray is currently intersecting is done similarly to the CPU-side code in listing 7. Listing 10 shows the GLSL translation of this function, which makes use of the built-in efficient boolean vector operations in GLSL.

```
uint GetOctant(in vec3 P, in vec3 ParentCentreP)
{
    vec3 G = vec3(greaterThan(P, ParentCentreP));
    return G.x + G.y*2 + G.z*4;
}
```

Listing 10: GLSL source of the `GetOctant` function

Determining the Child Node

All SVO nodes contain a child pointer field that indicates the location of the first non-leaf child voxel. To obtain a given node's child, we must first check if the first child is located within the same block as the parent (this does not necessarily imply that the particular child we are looking for is in the same block as its parent). To do this we can inspect the *far-flag* bit of the current node's child pointer bitfield. If this flag is not set, we can interpret the child pointer field as a direct offset into the parent's block. If the far flag is set, we

must look up the block index of the first child and skip to that block.

Listing 12 shows the complete source code for the child node lookup procedure. The block index of the parent node `Blk` is passed as an `inout` parameter because it must be both read from and written to by `GetNodeChild`. This is so that the caller can receive an updated current block index in case the child is located in a block different to that of the parent. Note that this can occur even if the parent's first child is located inside the same block as the parent, as illustrated by figure 6.5.

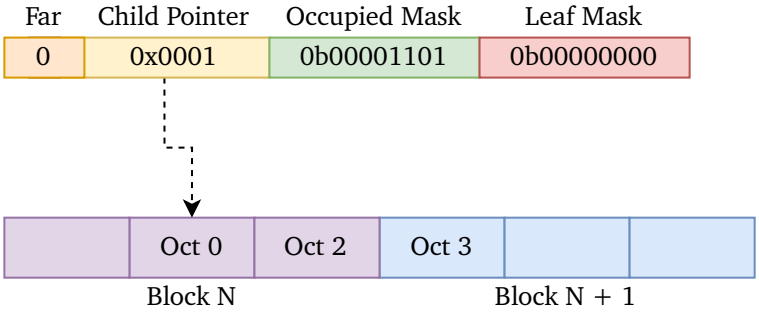

Figure 6.5: Child pointers always point to the first non-leaf child (siblings are stored sequentially). In this example, the parent node has three occupied child octants (0, 2 and 3). The first child resides in the same block as its parent, so the child pointer value can be interpreted as an offset into block N.

Since siblings always reside side-by-side in memory, it is relatively simple to compute the index of a particular child when given the index of the first child. To find the index of a child node relative to the first child, we first compute a mask of non-leaf octants. We then count the number of set bits lower than the octant we are trying to locate. This produces an offset from the first child.

```
uint ChildPtr = (Parent & CHILD_PTR_MASK) >> 16;
uint OccBits = (Parent & OCCUPIED_MASK) >> 8;
uint LeafBits = (Parent & LEAF_MASK);
uint NonLeafMsk = OccBits & (~LeafBits);
uint SibMsk = (1 << Oct) - 1;

uint ChildOffset = bitCount(NonLeafMsk & SibMsk);
```

Listing 11: Calculating the offset of a child octant `Oct`

Storing the Child Context

The raycaster maintains a stack of parent voxels which track the ray's progress through the SVO structure. The voxel stack is implemented as a GLSL array of the `st_frame` structure outlined in listing 13

```
uint GetNodeChild(in uint Parent, in uint Oct, inout uint Blk)
{
  uint ChildPtr = (Parent & CHILD_PTR_MASK) >> 16;
  uint OccBits = (Parent & OCCUPIED_MASK) >> 8;
  uint LeafBits = (Parent & LEAF_MASK);
  uint NonLeafMsk = OccBits & (~LeafBits);
  uint SibMsk = (1 << Oct) - 1;

  uint ChildOffset = bitCount(NonLeafMsk & SibMsk);

  // Check if we need to look up a far pointer for this child
  if ((Parent & FAR_PTR_BIT_MASK) == 0)
  {
    uint Child = Nodes[Blk*EntriesPerBlk + ChildPtr + ChildOffset];

    Blk += (ChildPtr + ChildOffset) / (EntriesPerBlk);

    return Child;
  }
  else
  {
    // Find the far ptr associated with this node. To do this, we need to compute
    // the byte offset for this block, then index into that block's far ptr
    // list for this node.
    uint FarPtrIndex = (Parent & CHILD_PTR_MASK) >> 16;
    uint FarPtrBlkStart = Blk*FarPtrsPerBlk;
    far_ptr FarPtr = FarPtrBuffer[FarPtrBlkStart + FarPtrIndex];

    // Skip to the block containing the first child
    Blk = FarPtr.BlkIndex;
    uint BlkStart = Blk*EntriesPerBlk;

    // Skip any blocks required to get to the actual child node
    Blk += (FarPtr.NodeOffset + ChildOffset) / EntriesPerBlk;

    uint Child = NodeBuffer[BlkStart + FarPtr.NodeOffset + ChildOffset];

    return Child;
  }
}
```

Listing 12: Complete GLSL source code for the `GetNodeChild` function

For each push operation at some depth d, a stack entry is created at index $d - 1$. This can then be restored by a later *pop* operation.

```
struct st_frame
{
  uint Node;
  int  Scale;
  vec3 ParentCentre;
  uint BlkIndex;
};
```

Listing 13: GLSL source of the `st_frame` structure stored in the voxel rendering stack.

6.3.2 Advance - Examining a Sibling Voxel

If the current octant is not occupied, we must advance the current octant to whichever octant of the same scale the ray enters next. To do this, the current octant must be advanced along whichever axes the ray exits the current octant through, i.e. the t_{max} values.

If the current octant is represented with an integer ranging from 0 – 7, with bit indices corresponding to the x, y and z axes respectively, we can determine the next octant's integer representation by flipping the bits for which the t-value for that axis matches the t_{max} value. Figure 6.6 shows a two-dimensional example: the current octant is set to 10 and the t_{max} value corresponds to the x-axis, thus we should flip the x bit in this example. By the `equals()` GLSL intrinsic, we can exploit the fast parallel hardware of the GPU to perform the comparison for all three axes at once. The result of this comparison is a mask that can be XORed with the current octant to produce a candidate next octant (we must still check if the next octant is valid given the direction of the ray; if this is not the case we must execute the *Pop* operation).

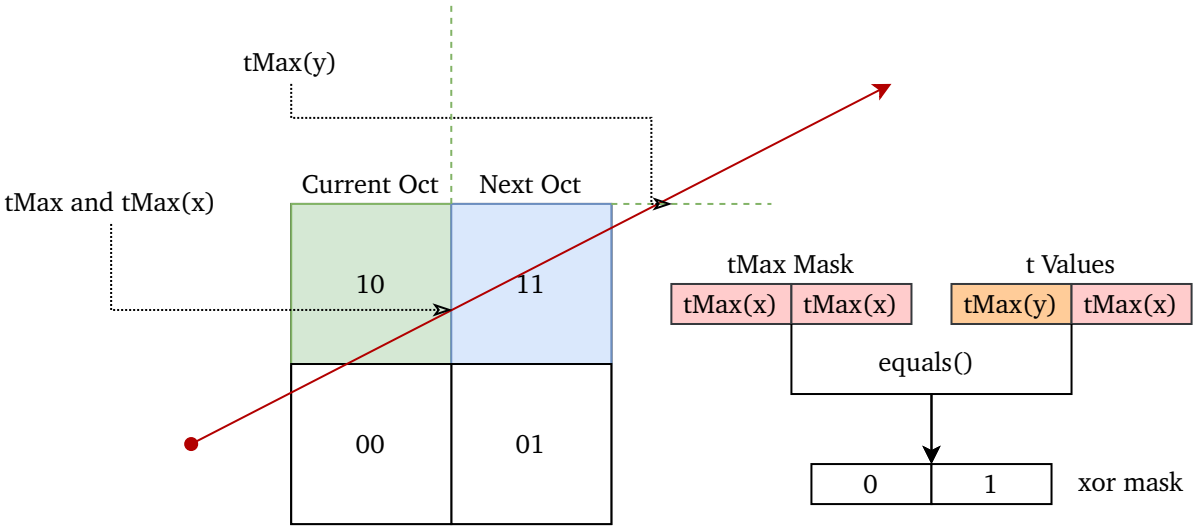

Figure 6.6: Creating a xor-mask for computing a candidate for the next octant

Listing 14 shows the GLSL source code for the `GetNextOctant` function. The code makes use of some GLSL-specific features to avoid costly branching. First, the `equals` GLSL intrinsic is used to compute a boolean vector representing which components of `tMaxes` are equal to the `tMax` value. This is then multiplied by the `OCT_BITS` constant vector $(1, 2, 4)$ which is used to shift the values of `XorMsk3` into the correct bit positions. Finally, the values of `XorMsk3` are added together to produce the scalar XOR mask that will perform the correct bit-flips on `CurrentOct`

```
uint GetNextOctant(float tMax, vec3 tMaxes, uint CurrentOct)
{
    // if tMax == tValues.x then NextOct ^= 1;
    // if tMax == tValues.y then NextOct ^= 2;
    // if tMax == tValues.z then NextOct ^= 4;
    uvec3 XorMsk3 = uvec3(equals(vec3(tMax), tMaxes));
    XorMsk3 *= OCT_BITS;
    uint XorMsk = XorMsk3.x + XorMsk3.y + XorMsk3.z;
    return CurrentOct ^ XorMsk;
}
```

Listing 14: GLSL source of the `GetNextOctant` function

6.3.3 Pop - Exiting a Voxel

Whenever the raycaster traverses a parent voxel, but does not actually intersect leaf geometry, the raycaster must exit that parent voxel and proceed to the next voxel of that parent's scale. This is the *pop* operation.

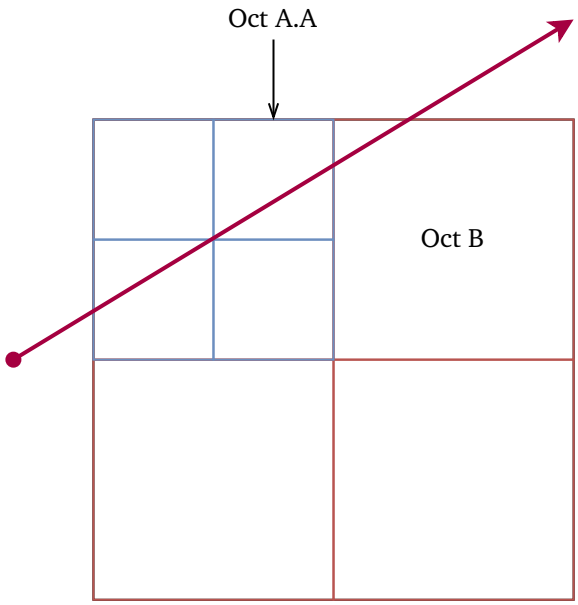

Figure 6.7: Ray exiting octant *A.A* and proceeding to octant *B*. Note the change of scale.

To determine what scale voxel the ray enters next, we must examine the path taken from the ray's starting position through the octree up to this point. By restricting the octree to only positive space, we can determine a path taken through the octree by looking at the binary representation of each coordinate of the ray's current position. For example, in figure 6.7, the current node centre position might be $(8, 8)$, while the ray's current position might be $(16, 27)$. Consider the binary representations of these positions in figure 6.8

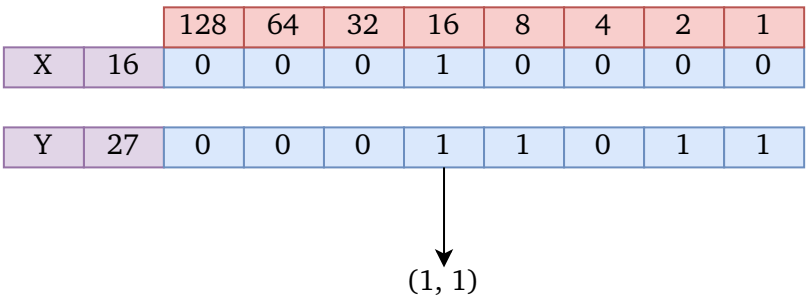

Figure 6.8: Deriving an octree path from the bit indices of the ray's current position. In this two dimensional example, it can be seen that the ray is in quadrant 3 (top-right) at scale 16.

By associating each bit position *i* in these binary numbers with an octree scale value, we can determine which octant the ray has passed through at which a particular scale. We can repeat this process for the centre position of the ray's current octant, giving a path through the octree to the current octant. By comparing these bit vectors, we can determine the scale of the next octant to process. In the example above, the highest differing bit is bit 16, indicating that we exit the current voxel at scale 16. Thus, we can determine the next SVO node to process by restoring parent node to the stack entry at scale 16 and continuing the raycast.

This process hinges on efficiently determining the highest-differing-bit (HDB) between two binary numbers. Fortunately, the GLSL language has a built-in functionality for determining the highest set bit in a given binary number — the `findMSB` function. We can combine this function with the standard XOR operator to very quickly determine the highest differing bits in two given uint-vectors.

```
uvec3 HighestDiffBits(uvec3 V0, uvec3 V1)
{
  // First, determine which bits differ between V0
  // and V1 using the XOR operator. Then, determine
  // the highest set bit in the result.
  return findMSB(V0 ^ V1);
}
```

Listing 15: GLSL source of the `HighestDiffBits` function

6.4 Rendering Leaf Data

Although ray-casting can be used to efficiently render the sparse voxel octree data structure, it only deals with occupancy information; that is, whether a voxel is present at some location or not. To render rich voxel attribute data, such as colours and normals, the renderer must somehow determine the attribute data for a particular voxel given its world coordinates. One approach would be to place all of the leaf data inside a very large three-dimensional texture that stores each voxel at the smallest possible level of detail in a single pixel. While this technique would support very fast data retrieval (simply lookup the texel at a leaf voxel's world coordinates), it quickly becomes infeasible for any reasonably large and/or detailed scene, requiring massive amounts of texture memory. In *Efficient Sparse Voxel Octrees*, Laine and Karras

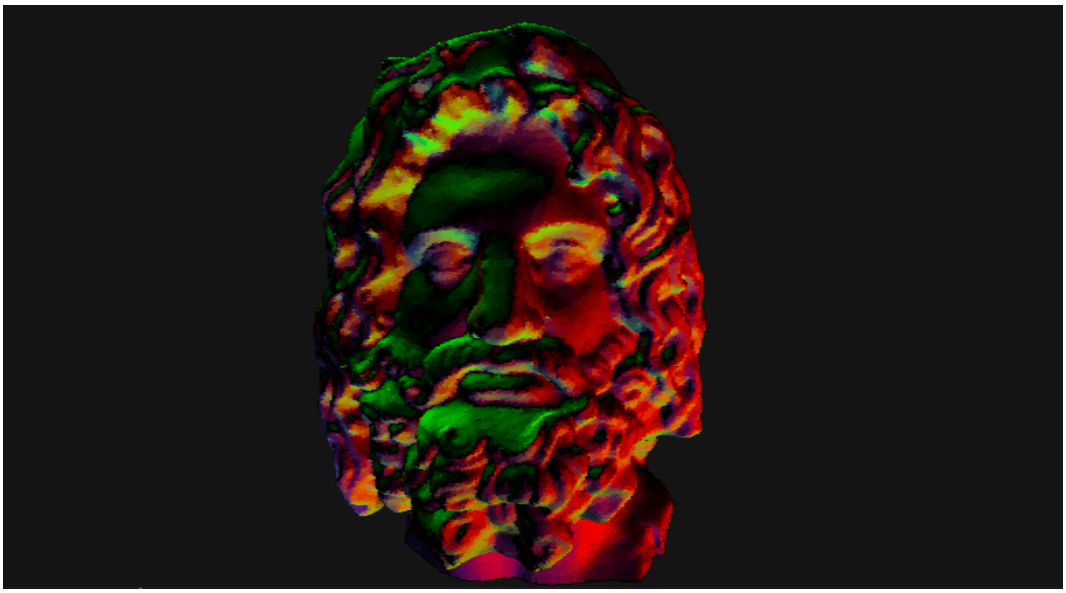

Figure 6.9: Voxelised head of Serapis, with full per-voxel colour and normal data stored in a sparse texture

address the problem of voxel attributes by holding the leaf data in a secondary buffer that mirrors the layout of the sparse voxel octree. While this does make retrieval simple, since when the raycaster hits a leaf voxel it will know the index of that voxel's parent entry in the SVO buffer, this method also leaves large gaps in this secondary buffer where non-leaf voxels would be. In this project, the OpenGL sparse textures extension is used to support constant voxel lookup times without having to store large amounts of empty space or non-leaf voxels.

6.4.1 Sparse Textures

Sparse Texturing, also called Virtual Texturing, is a graphics programming technique that allocates a very large (normally multi-gigabyte) virtual texture, but only stores smaller regions of that texture in physical memory. Sparse textures mirror the CPU-side concept of virtual memory, where "pages" — blocks of data — are moved in and out of physical RAM as they are needed.

A major advantage with sparse textures is the fact that, provided the host machine supports the `GL_ARB_sparse_textures` extension, sparse texture lookups are essentially identical to regular texture lookups from a shader point of view, since the texture hardware automatically fetches the correct page based on the lookup coordinates.

To create this sparse texture, the tree leaf data is first indexed by the coordinates of the enclosing voxel's centre. This indexed data is then partitioned

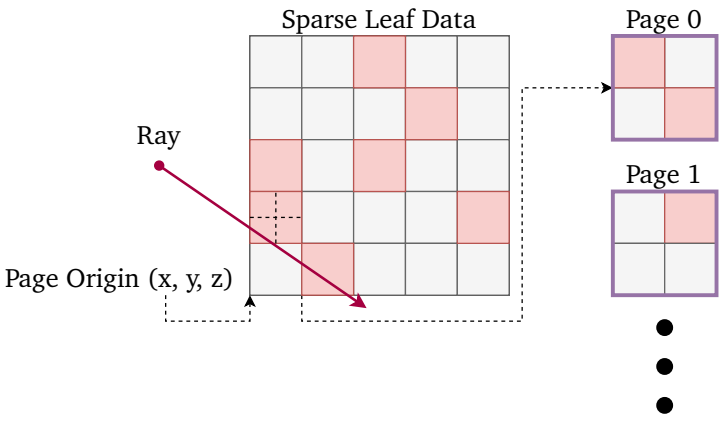

Figure 6.10: Sparse textures only map regions of the world that contain leaf data into physical memory pages, eliminating large areas of unused data

into buckets, which can be thought of as three-dimensional coarse grids, with either a leaf data point or null at every point. The page coordinates for each bucket are calculated by dividing each coordinate in the bucket origin by the respective hardware page dimensions, obtained at runtime.

6.4.2 Alternative Approaches

Unfortunately it was found that, at least for the shipped test models, the sparse texture pages actually have very poor occupancy, resulting in a lot of wasted space even when large empty world regions are not stored. An ideal approach would support constant-time leaf lookups without requiring so much extra space.

Alcantara et al's *Real Time Parallel Hashing on the GPU* [2] might provide such an approach. Alcantara et al. showed how large sparse volume data sets could be stored on the GPU using parallel cuckoo hashing. For N voxels, their approach consumed roughly $1.42N$ space, an acceptable tradeoff for constant voxel lookup times.

```
const vec3 PageSize = Get3DSparseTexturePageSize();

for (Leaf in Leaves)
{
    vec3 LeafCoords = Leaf.Position % PageSize;
    vec3 BucketCoords = Leaf.Position / PageSize;

    if (BucketDoesNotExist(BucketCoords))
    {
        Bucket = CreateBucket(BucketCoords);
    }
    else
    {
        Bucket = GetBucket(BucketCoords);
    }

    Bucket[LeafCoords] = Leaf.Data;
}

for (Bucket in Buckets)
{
    CommitTexturePage(Bucket.Coords, Bucket.Data);
}
```

Listing 16: Pseudocode for building the leaf data sparse texture

Chapter 7: Viewer Application

Along with the core voxel library, this project also contains a small application which serves as an example of how client developers could use the library to render voxel scenes. The viewer application, contained within the `sabre.cc` code module, provides a simple first-person scene viewer which allows users to explore and modify generated or imported voxel scenes.

Since the viewer application is not bound by the same restrictions as the core voxel library with regards to third-party code, the viewer utilises several libraries to speed up development. The GLFW [18] library is used to create the window and handle input, the Dear ImGUI library [7] is used to create a quick-and-dirty main menu and graphical controls and the GLAD library [11] is used to load a modern OpenGL context.

7.1 Controls

The viewer application supports the following controls:

Input	Action
W	Move forward
A	Move left
S	Move right
D	Move back
Left Shift	Move down
Space	Move up
Right Mouse	Place voxel
Left Mouse	Destroy voxel

7.2 Features

The viewer application allows users to select from a selection of pre-defined scenes intended to serve as showcases of the project's capabilities. Three

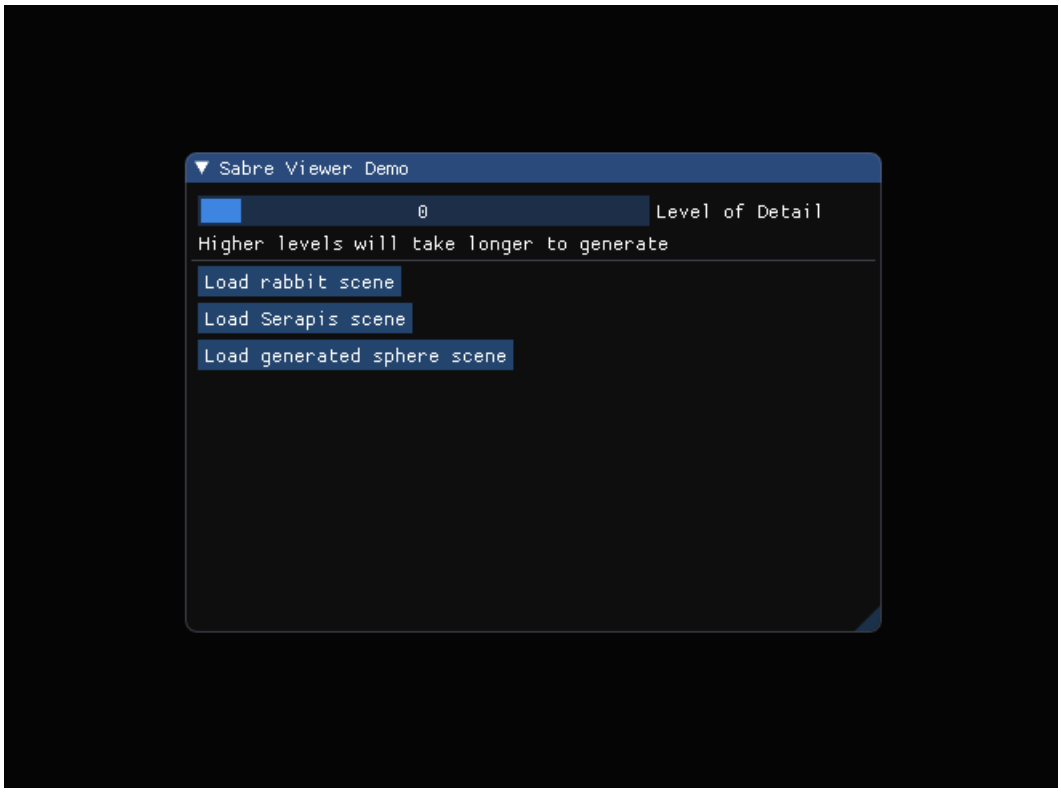

Figure 7.1: The main menu screen, built with Dear ImGUI

scenes are shipped with the project:

- Rabbit The famous Stanford Bunny, loaded from an external data file.
- Serapis Bust of Serapis, loaded from an exernal data file.
- Sphere A solid sphere, generated at runtime.

Users can also select a detail level for these scenes by dragging the "detail level" slider, as shown in figure 7.1

Chapter 8: Evaluation

This project has been, by a very large margin, the most complex piece of software I have ever worked on at the University. Over the course of development I met with several unexpected challenges, but ultimately I believe this project does fulfil all of the original objectives I set out to achieve except for one: infinite scene support.

8.1 Achieved Objectives

8.1.1 Performance

One of the core objectives of this project was to deliver a library that could compute and render an entire video frame in no more than 33 milliseconds, corresponding to a frame-rate of 30 frames per second. The finished product maintains a steady 60 frames per second (corresponding to 16 milliseconds per frame) even when dealing with very detailed scenes. These measurements were obtained on a NVidia GTX 940MX graphics card using the `glfwGetTime` APIs. Encouragingly, it was observed that on a GeForce GTX 1080 system, the software was able to render all of the shipped scenes in 8 milliseconds.

While this project may have met its performance objectives, there is always room for improvement. In particular, the polygon mesh importing code could benefit greatly from some optimisation; currently the huge numbers of `std::set` and `std::unordered_map` inserts are the main bottleneck [1]. I chose to use the hashtable-based triangle indexing approach so that the mesh import logic could be cleanly encapsulated into a pure sampler function, but it may be worth exploring alternative mesh voxelisation techniques that do not have such high overhead. Baert et al. describe a different approach to sparse voxel octree construction in *Out-of-core construction of sparse voxel octrees* [3], where the mesh to be voxelised is first partitioned into several high-resolution voxel

[1] *Measured with Intel VTune Amplifier*

grids, ordered by Morton codes[2], which may be intermediately saved to disk while other partitions are being processed. The SVO is then constructed out of these partitions without needing to load the entire high-resolution grid into memory.

8.1.2 Scene Detail

Another important goal for this project was the ability to render very small voxels, allowing highly-detailed scenes to be created. At the highest detail level of '10', the shipped product can simulate voxels at a scale of $\frac{1}{1024}$. To apply some real-world context to this, the Stanford Bunny is 7.5 inches high [26]; an individual voxel in this model at real-world scale would be roughly 0.18 millimeters in dimension.

8.1.3 Arbitrary Scene Modifications

The shipped product supports deleting and inserting arbitrary voxels at interactive frame rates. However, dynamic uploading of voxel leaf data remains to be implemented, though it would be relatively simple to add. Since the library already holds all of the leaf data on the CPU side, to insert e.g. a new colour one would simply need to take the leaf voxel centre position (known at insert-time) along with the leaf data and add this to the CPU-side list of leaves. The renderer can then determine the correct texture page to modify or create by calculating the texture page coordinates using the algorithm outlined in listing 16.

Another caveat associated with voxel edits is the fragmentation of blocks due to nodes being reallocated. Even though the holes that appear in blocks due to this phenomenon are very small, they would still accumulate over thousands of edits. A simple solution to this problem would be to mark fragmented blocks and then periodically re-construct all marked blocks, re-packing the nodes. The potential performance impact of this could be mitigated by copying the block to be reconstructed into a separate thread, perform the reconstruction and then atomically swap the old block for the newly defragmented one.

[2] *Morton order, also called z-order, is an locality-preserving method of encoding 3D positions into integers by interleaving the bits of the x, y and z coordinates. [24]*

8.1.4 Educational Value

This project has been of enormous educational value; since completing the project I have had my views on topics such as memory management and API design challenged by various hurdles encountered throughout development. Previously, I had held the opinion that dynamic memory management should be avoided whenever possible, and that systems should be built to run within tightly defined bounds. Having had to deal with much larger datasets in this project than I had in the past, I now see that this is not always a practical design decision, and that carefully managed dynamic memory allocation can still be performant and simple to understand.

I have also had the opportunity to learn about a great many new technologies and tools. GPGPU computing is becoming of huge importance, and I was pleased to get the opportunity explore this area using compute shaders. I also had the chance to explore advanced computer graphics techniques, such as sparse textures, and cutting edge tools like NVidia's NSight Graphics.

8.2 Potential Improvements

8.2.1 SVO Block Streaming

The ability to stream octree blocks in and out of memory would be a substantial improvement to this project. Such a feature would allow *Sabre* to support infinite world scenes by continuously streaming in generated SVO regions as the player moves around, analogous to *Minecraft*'s world generation.

SVO streaming would also open up opportunities for more performance optimisations; octree regions that are not visible to the camera could be purged from memory, for example.

8.2.2 Generic Voxel Attributes

Currently, the library supports two voxel attributes: colours and normals. A way for clients to specify their own shading attributes would bring this library into line with industry-standard tools, since many clients may want to precisely define their own material properties such as colour or glossiness.

Adding user-defined materials to this software would certainly be possible, although GPU resources would need to be managed carefully since, with the current design, each separate attribute requires its own OpenGL texture unit.

8.2.3 Support for Dynamic Linking

Currently, the library is distributed as a collection of source and header files that are compiled directly into client applications. This approach is well-suited to C-based application development, but is less convenient for languages that do not follow a traditional C-like "compile and link" development process, like Python or Lua. In those languages, it is easier to interface with C libraries by dynamically loading the library at runtime. This technique requires that the library in question be compiled into either a *Dynamic Link Library* (DLL) on Windows or a *Shared Object* (SO) on Unix-like systems.

8.2.4 Texture Sampling

An interesting improvement to this library would be to have the mesh importer retrieve associated colour values for voxels by sampling textures attached to the 3D model. This would pave the way for a fully-integrated polygon to voxel workflow, allowing clients to keep their existing infrastructure while still leveraging the advantages of voxels.

8.3 Conclusion

Overall, I am very satisfied with this project. There are, of course, many more improvements to be made in order to bring this project in line with industry-standard tools, but I feel that *Sabre* shows the viability of voxel rendering as a competitor to polygon-based workflows.

Over the course my final year at University, I have learned more than I ever thought I would have to know about memory management, GPU programming, API design, performance engineering and time management, to name but a few. I have come away from this project with my views on many software engineering practices radically changed for the better, becoming a more well-rounded and strategically-minded programmer.

Code References

Certain code snippets throughout the project are based on external sources. These sources are listed below along with the file name and line number(s) at which they occur.

- `sabre_math.h` Over several years, I had accumulated math utilities inside this file. It is cited here in the interests of avoiding self-plagiarism. Except when indicated otherwise, all code within this file is my own. Power, 2019.
- `sabre_data.cc:419-425` Ray projection plane computation method. Partly based on code sourced from https://github.com/johnBuffer/VoxelCaster/blob/6477c4e2dc2f90a578d02d3ad7d3c40ab7443a14/src/gpu_raycaster.comp. Jean "JohnBuffer" Tampon, 2019.
- `sabre_import.cc:35-51` Morton code encoding for 3-vectors. Sourced from https://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/. Fabien "ryg" Giesen, 2009.
- `sabre.cc:80-99` Cube-sphere intersection formula. Sourced from https://stackoverflow.com/a/4579069/3121161. Ben Voigt, 2011.
- `sabre_math.h:1172:1191` Fast rotation of vector by quaternion. Sourced from https://fgiesen.wordpress.com/2019/02/09/rotating-a-single-vector-using-a-quaternion/. Fabien "ryg" Giesen, 2019.
- `sabre_data.cc:165-178`, `sabre_svo.cc:848-862` Ray-box intersection. Further discussed in the report with full citation to paper. Majercik et al., 2018. "A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering".
- `sabre_import.cc:111` SSE component-wise sign negation. https://stackoverflow.com/a/20084034/3121161. Paul T. Russel, 2013.
- `sabre_import.cc:154-158` SSE dot product. https://stackoverflow.

com/a/35270026/3121161. Peter Cordes, 2016.

- `sabre_import.cc:433-525` GLTF mesh loading partially based on *MeshOptimizer*. https://github.com/zeux/meshoptimizer/blob/acd6d73a8535cee3b35f7542c gltf/parsegltf.cpp. Arseny "zeux" Kapoulkine, 2020.

References

[1] Tomas Akenine-Möllser. "Fast 3D triangle-box overlap testing". In: *Journal of graphics tools* 6.1 (2001), pp. 29–33.

[2] Dan A Alcantara et al. "Real-time parallel hashing on the GPU". In: *ACM Transactions on Graphics (TOG)* 28.5 (2009), pp. 1–9.

[3] Jeroen Baert, Ares Lagae, and Philip Dutré. "Out-of-core construction of sparse voxel octrees". In: *Proceedings of the 5th high-performance graphics conference*. 2013, pp. 27–32.

[4] Mike Bailey. *How to Use and Teach OpenGL Compute Shaders*. 2014. URL: https://www.khronos.org/assets/uploads/developers/library/2014-siggraph-bof/KITE-BOF_Aug14.pdf (visited on 01/03/2020).

[5] Tavian Barnes. *Fast, branchless ray/bounding box intersections*. 2011. URL: https://tavianator.com/fast-branchless-raybounding-box-intersections/ (visited on 01/03/2020).

[6] Kai Burjack. *Ray tracing with OpenGL Compute Shaders (Part I)*. 2017. URL: https://github.com/LWJGL/lwjgl3-wiki/wiki/2.6.1.-Ray-tracing-with-OpenGL-Compute-Shaders-%5C%28Part-I%5C%29 (visited on 01/04/2020).

[7] Omar Cornut. *Dear ImGUI*. https://github.com/ocornut/imgui. [Software Library]. 2019.

[8] Alex Evans. *Learning From Failure*. 2015. URL: https://www.youtube.com/watch?v=u9KNtnCZDMI (visited on 01/05/2020).

[9] Jon Fingas. *Here's how 'Minecraft' creates its gigantic worlds*. 2015. URL: https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/ (visited on 01/04/2020).

[10] Patrick Healy. *Computer Graphics II: Transformations*. 2019. URL: http://garryowen.csisdmz.ul.ie/~cs4085/resources/lect04.pdf (visited on 01/04/2020).

[11] David Herberth. *GLAD*. https://github.com/Dav1dde/glad. [Software Library]. 2019.

[12] The Khronos Group Inc. *glTF Overview - The Khronos Group Inc.* 2015. URL: https://www.khronos.org/gltf/ (visited on 05/03/2020).

[13] Khronos Group. *OpenGL 4 Reference Pages*. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/ (visited on 01/03/2020).

[14] Johannes Kuhlmann. *CGLTF*. 2018. URL: https://github.com/jkuhlmann/cgltf (visited on 05/03/2020).

[15] Samuli Laine and Tero Karras. "Efficient sparse voxel octrees". In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2010), pp. 1048–1059.

[16] Eric Lengyel. *Foundations of Game Engine Development, Volume 1: Mathematics*. Terathon Software LLC, 2016. ISBN: 978-0985811747.

[17] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 163–169. ISBN: 0897912276. DOI: 10.1145/37401.37422. URL: https://doi.org/10.1145/37401.37422.

[18] Camilla Löwy. *GLFW 3*. https://glfw.org/. [Software Library]. 2019.

[19] Alexander Majercik et al. "A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering". In: *Journal of Computer Graphics Techniques Vol* 7.3 (2018).

[20] Morgan McGuire. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data.

[21] Innes McKendrick. "Continuous World Generation in No Man's Sky". In: *Game Developers Conference*. 2017.

[22] Donald JR Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic ..., 1980.

[23] Minecraft Wiki Contributors. *Chunk format*. 2019. URL: https://minecraft.gamepedia.com/index.php?title=Chunk_format&oldid=1478710 (visited on 01/04/2020).

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

[25] New Sound Games. *Sparse Voxel Octree demo*. 2012. URL: https://youtu.be/km0DpZUgvbg (visited on 01/03/2020).

[26] Greg Turk. *The Stanford Bunny*. 2000. URL: https://www.cc.gatech.edu/~turk/bunny/bunny.html (visited on 05/04/2020).

[27] Jim Van Verth. "Math for Game Developers: Understanding quaternions". In: *Game Developers Conference*. 2013.

[28] Wikimedia Commons. *File:Voxels.svg — Wikimedia Commons, the free media repository*. [Online; accessed 2-January-2020]. 2018. URL: https://commons.wikimedia.org/w/index.php?title=File:Voxels.svg&oldid=314773564.

[29] Chris Wyman et al. "An Overview of Next-Generation Graphics APIs". In: *ACM SIGGRAPH 2015 Courses*. SIGGRAPH '15. Los Angeles, California: Association for Computing Machinery, 2015. ISBN: 9781450336345. DOI: 10.1145/2776880.2787704. URL: https://doi.org/10.1145/2776880.2787704.