Creating a Voxel Rendering Engine

EGH400-2 Project Delivery - Semester 1, 2023

Supervised by Professor Clinton Fookes

Alex McDermott (N10494367)

The Queensland University of Technology
June 2, 2023

Abstract

This thesis project aimed to assess the feasibility of a real-time voxel rendering engine in the browser. This was done to potentially make this application available for medical use, and allow exploration of medical scans more intuitively from any internet-connected computer. Assessing the relevant literature and research already done in this space, this paper explores voxel data generation, storage, and rendering, touching on algorithms such as Digital Differential Analysis (DDA). Despite using a rudimentary fixed-step size ray marching algorithm for its implementation, the performance demonstrated here is still well above the 60 Frames per Second (FPS) real-time rendering target. Achieving a locked 120 FPS on the test hardware, this implementation is well-suited for the web and is a promising proof of concept for future work. This paper also explores the potential for future improvements and optimisations, such as DDA, WebGPU, and more advanced shading.

Contents

List of Figures

Acronyms

API Application Program Interface. 6, 8, 9

CPU Central Processing Unit. 5, 8

DDA Digital Differential Analysis. 1, 7, 8, 11, 13, 14

DRY Don't Repeat Yourself. 11, 14

ECS Entity Component System. 8

FOV Field of View. 5

FPS Frames per Second. 1, 13, 14

GPU Graphics Processing Unit. 4, 5, 6, 9, 10, 14

JIT Just in Time. 8

LOD Level of Detail. 1, 4, 5

OpenGL Open Graphics Library. 6

QUT Queensland University of Technoloogy. 14, 15

RAM Random Access Memory. 4, 13

SVO Sparse Voxel Octree. 3, 5

WebGL Web Graphics Library. 6, 9

1 Introduction

The primary goal of this research project was to create a voxel rendering engine; with a secondary objective of utilizing flexible technology to allow for the engine to be used across a variety of platforms, mainly the web. Firstly, what is a voxel, let alone a voxel rendering engine? A voxel is essentially a 3D pixel and is a common way of representing volumetric data such as clouds, medical scans, and destructible terrain as seen in games, TV shows, and movies. As opposed to traditional rendering, which utilizes triangles to approximate surface geometry, a voxel render uses a fixed grid of evenly spaced voxels where each voxel stores information about the object they are approximating at that particular point in space. Primarily, this is whether the cell is filled (transparent or opaque) or not. This allows for a more intuitive representation of reality, as these voxels mimic particles and allow for easy manipulation of objects as we've come to expect in everyday life. There are more advantages to voxels, as well as drawbacks, that will be discussed in this report, along with techniques that can be used to mitigate them. This thesis aims to assess the feasibility of this sort of application in the browser, with the potential for use in interactive medical visualizations. In summation, the core goals of this thesis are outlined below.

- Generate a voxel representation of an object.
- Implement an algorithm to directly render this data structure without the need for intermediary processing such as meshing into triangles.
- Assess the implementation and its feasibility for web platforms.
- Identify future improvements and optimisations given the performance budget.

2 Literature Review

Before delving into the existing research done in this field, I'd like to provide some context for this project. Due to unforeseen reasons in the first half of this project, my original thesis plans with Airbus fell through as they were unable to find a suitable supervisor to manage the project. Hence, in Week 10, I switched to this project, having little background knowledge as my Literature Review was done concerning the original topic Airbus proposed. As such, development in this second thesis unit has been slow as a significant amount of time was spent researching and understanding existing work which would ideally have been completed in the first half of the project. Because of this, the scope of the final implementation is quite limited but I believe the assessment was still highly beneficial, leaving room for many future improvements and expansions on current work.

2.1 Voxel Storage

2.1.1 Data Structure

The first and most prevalent technique is to utilize a data structure called an Sparse Voxel Octree (SVO). An SVO operates on the same principle as a quadtree, such as those used to minimize checks against distant objects in 2D collision detection. It is a 3D tree structure that iteratively subdivides itself as necessary down to a given depth depending on the resolution needed. This technique is documented well in literature but is particularly well explained in this paper on "Efficient Sparse Voxel Octrees" [1] where their explanation is accompanied by this visualization showing how tree depth level allows approximation of more precise geometry.

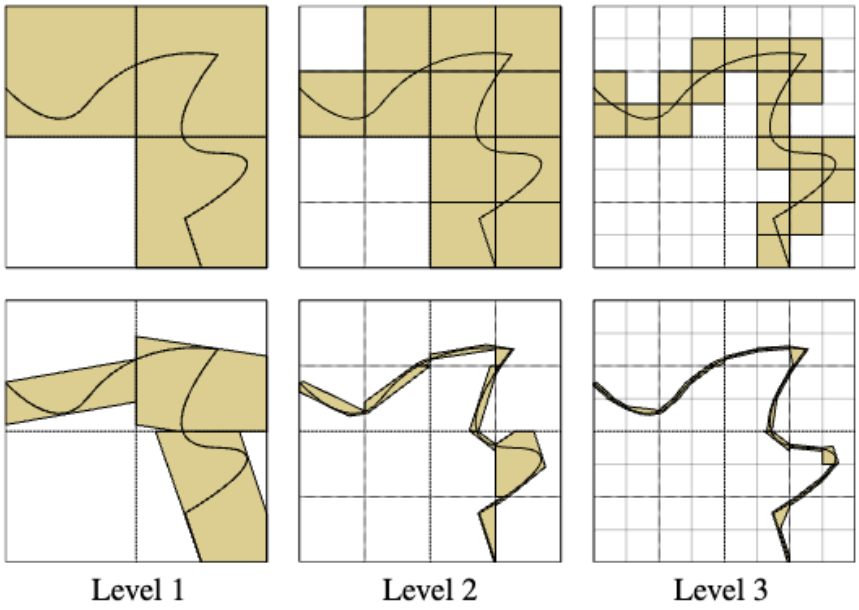

Figure 1: How octree depth affects volume LOD.

To provide an example of how this is useful, let's say we have a scene that has some ground, a house, some space above it and then some clouds in the sky; all represented in voxels. Representing this scene in an octree, we surround it with a single cube bounding box (proxy geometry). This cube is then split into 8 smaller cubes because there is a lot of geometry that needs to be represented; these smaller cubes are called octants. This idea of proxy geometry is inspired by the voxel game Teardown [2] where they use a similar technique to compose scenes of multiple separate voxel objects. The octants created by subdividing this proxy geometry are essentially smaller voxel volumes and they keep repeating this splitting process until a set tree depth is reached. The benefit of this is that, when an octant has no voxels in it, it has no reason to keep subdividing.

Considering our example scene, this allows the empty sky to essentially be represented by one large voxel that can be quickly jumped over when traversing this structure at render time. More importantly, if we assume our sky took up $\frac{1}{3}$ of our scene, this equates to around a 33% reduction in voxel memory footprint as it can now be stored as a single byte as all the voxels in that region are of a single empty type. This reduction by itself is already a significant saving, and when applied to a whole scene this effect is compounded over all the empty space such as inside the house, around the clouds, and possibly caves underground. These savings add up and drastically reduce Graphics Processing Unit (GPU) memory footprint and bandwidth usage.

2.1.2 Compression and Voxel Representation

As stated previously, voxel data is often stored in an evenly spaced 3D grid of voxels. In its simplest form, this can mean each voxel only needs to store a single bit, either 1 or 0 depending on whether the voxel is filled or not. Although this is very rarely the case in actual implementations, as things such as different materials, densities, normal information, colours, and more are likely also stored in the voxel. Let's instead assume that each voxel is represented by a byte or 8 bits. This gives us the flexibility to either store an index that could map to 255 different predefined materials or, if more control is needed, bit packing could be employed with a custom encoding format to store this information in the byte directly. To provide a basic example illustrating this, if we assume a hypothetical voxel volume size of 512^3, with each voxel being 1 byte, this already consumes around 135 MB of memory. This may not seem like a lot, seeing as it's common for modern computers to have upwards of 8 GB of memory, although since voxel rendering is primarily a GPU process, this memory will be stored in video Random Access Memory (RAM) which is much less prevalent and often tightly budgeted. This high memory footprint is one of the primary drawbacks of voxel usage. Fortunately, many techniques can be used to reduce the memory footprint of voxel data. The paper "Geometry and Attribute Compression for Voxel Scenes" [3] explores this palette-based compression scheme and aims to decouple voxel geometry from attribute data such as colour and normal information. In their highly detailed test scenes, they show reduced "memory usage from 4.49 times (for the citadel) up to 11.5 times (for San Miguel)" using this method.

Another common technique that can be used to reduce the memory footprint of storing voxel volumes is run-length encoding. Run-length encoding is a lossless form of data compression that simply counts how many times

the same value occurs in a sequence and stores that value along with the count. This is useful in the context of voxels, as it is common for many of the same types to appear next to each other, such as dirt on the ground or air blocks in the sky. Let's assume we are looking at a leaf octant of the SVO that is 16^3 voxels in size. If we assume that this octant is filled with air, we can store this entire octant in a single byte, as we know that all the voxels in this octant are of the same type. This is a very simple example, and in reality, the savings will be much less when applied over a whole scene, especially with lots of variation, but it still shows the potential for compression in voxel data.

Streaming

Streaming is another technique that can be used to reduce the memory footprint of voxel data. Streaming is the process of only loading the data that is needed at a given time. This is why you sometimes get a loading screen when going through doors in open-world games, it gives the game time to load all the models, textures and data related to that section of the game. In voxel applications, steaming is often implemented by "chunking" the environment into smaller, more manageable sections. Many have most likely seen this idea implemented in Minecraft [4] where sometimes artefacts can be seen where chunks are yet to be generated or loaded in, leaving cube-sized holes in the map. Chunking also works well with the octree solution we previously talked about, as only specific nodes of the octree need to be loaded into memory depending on where the observer is in the scene. Voxel steaming can be implemented to only load data into the GPU that is visible to the observer, allowing large savings in memory footprint as the need to store information on data outside the players Field of View (FOV) (such as behind them) is no longer needed. To provide an example, a typical FOV in games is around 60 degrees, let us assume the worst-case scenario, where want to load 180 degrees worth of voxels so the games doesn't lag while steaming in new data when rotating. Assuming the player is centred in the voxel space, this automatically discounts 50% of the scene that doesn't need to be loaded into GPU memory and thus a large saving in memory footprint. This idea can be seen documented in the paper "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering" [5] where they state: "for a given point of view, not the entire volume needs to be in memory. By organizing the data in a spatial sub-division structure, empty parts can be left un-subdivided and distant parts can be replaced by lower mipmap levels, leading to ... less GPU memory requirements". Here, the validity of only loading needed parts of the voxel volume is solidified. They also suggest another good optimisation; taking advantage of the iteratively increasing detail at deeper levels of the SVO structure, to reduce the number of ray traces, distance octree volumes can be limited in how far they traverse down the tree. This leaves more processing power for closer objects which take up a proportionally larger area on the screen. They illustrate this idea well in the following figure included in their paper.

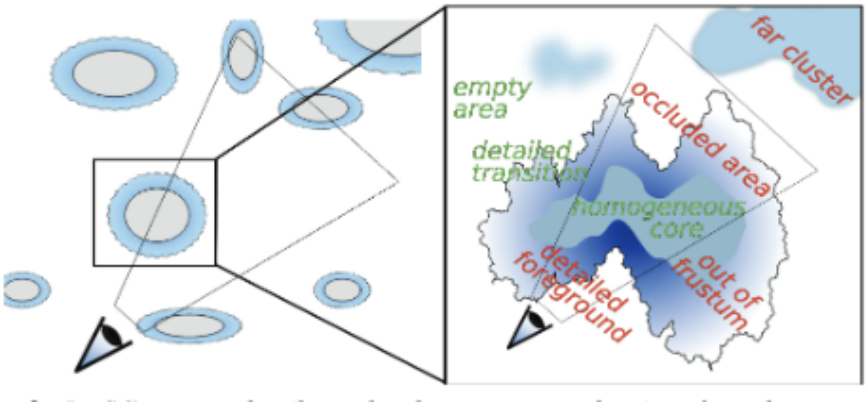

Figure 2: Visualisation of the camera frustum culling and dynamic LOD.

These are the three primary and most effective techniques used to reduce the memory footprint of voxel volumes. As stated, this is important as the less space a voxel takes up, the more of them we can store, opening up opportunities to represent more detailed and complex scenes. Having a small voxel footprint is also beneficial when it comes to rendering time as well, seeing as the voxel data has to be transferred between the Central Processing Unit (CPU) and GPU potentially every frame if updates have occurred. These transfers are expensive and a notorious bottleneck in graphics programming. This is another reason why it is important to keep the voxel footprint as small as possible, to reduce the amount of data that needs to be transferred between the CPU and GPU.

2.2 Voxel Rendering

2.2.1 Graphics APIs

Now that we have a basic understanding of how voxel data is stored and streamed efficiently, we can look further into how this data is displayed. As stated, voxel rendering is primarily a GPU process, and as such, we will require a means to communicate this data to the GPU device. This is often done using one of the four core main graphics Application Program Interfaces (APIs) being Metal, Vulkan, Open Graphics Library (OpenGL), or DirectX. These libraries are responsible for telling the GPU what to draw, where to draw it, and how to do it. As such they are notoriously low-level and require a large amount of boilerplate code to get set up. Fortunately in the browser, we have two alternatives Web Graphics Library (WebGL) and WebGPU, they are essentially wrappers around these low-level graphics APIs that are designed to be more accessible and provide the functionality needed in the browser. The first of which, WebGL, was first released in 2012 and based on OpenGL. WebGL is very mature, meaning there is plenty of documentation and examples available online. Although being more than a decade old, graphics hardware and rendering technology have come a long way in this time, and as such WebGL is not as efficient as modern graphics libraries such as Vulkan. WebGPU on the other hand is based on Vulkan and is designed to be an extensible replacement to WebGL that is more modern and efficient. Because of this, and due to the high-performance nature of the application being built as part of this thesis, WebGPU was initially chosen as the target for this project. Although after difficulties finding learning resources, limited documentation and changing specifications as the API are still being finalized, it was deemed to not be production ready yet. As such, the project was switched to use WebGL as it is more mature and has a larger community and support base. Although this switch was a setback in terms of the project timeline, it was greatly beneficial in terms of the utilisation of existing resources. In addition to this, instead of utilizing the WebGL API directly, a third-party library called ThreeJS was used which provides yet another abstraction on top of WebGL to make it even more accessible. This was done in the interest of further reducing boilerplate code, as well as looking to future-proof the project since the core developers of ThreeJS have already stated that they will be porting the library to WebGPU once it is production ready. This allows the program to utilize the stability and knowledge of the WebGL ecosystem while still being able to easily port to WebGPU and reap its performance benefits once it is production ready.

2.2.2 Rendering Approach

Now that a means of communication has been established with the GPU, a technique is needed to render voxels to the screen. As previously stated, the goal of this voxel renderer is to directly draw the voxel data as opposed to meshing the voxel geometry into triangles and rending that; which is a common approach used in games such as Minecraft. The reason this approach is so prevalent is because of how GPU architecture has evolved. As triangles are so versatile, and essentially the foundation for all 3D representations of objects seen in movies, tv shows and video games, GPU hardware has evolved to accommodate this. Leading to decades of GPU optimisations and design choices made with triangle rendering in mind as it is by far the dominant way to represent geometry. This makes meshing voxels into triangles using algorithms such as marching cubes, such a popular approach as it leverages these optimisations and additional hardware acceleration. This is not to say rendering voxels isn't possible or impractical, just that it is not as common and requires a different approach. This approach is called ray tracing, specifically, ray marching. It works by casting a ray from our camera, into the scene and "marching" along that ray, incrementally extending it and testing if a voxel has been hit. This is a vast simplification of a complex topic whose implementation will be explored later in this paper. A drawback of this approach is that ray casting can be quite expensive, especially when multiple rays are cast per pixel, per frame, and 60 times per second. At common resolutions such as 1080p that could easily require north of 100 million rays cast per second assuming two samples per pixel. This might sound like overkill or an unrealistic scenario but casting multiple rays per pixel is extremely common in ray tracing applications as it is often used to average sample values and decrease noise, or needed to calculate visual effects such as shadows, reflections, and refractions.

2.2.3 The DDA Algorithm

Having realized the performance pitfalls of our selected rendering approach, the algorithm we implement must be as efficient as possible if this approach is to be feasible within the browser. Knowing that we need to march a ray through our voxel space, a naive approach is to take steps of fixed size along the ray. This is problematic for two reasons. Firstly, when taking fixed step sizes over an axis-aligned volume of cubes at an arbitrary angle, it is possible to skip over voxel checks along the ray's path if a step happens to skip over a voxel corner (being less wide than a voxel's diagonal). This can lead to artifacts of voxel edges. Secondly, this approach is flawed from a performance standpoint as well. Taking fixed step sizes is inefficient when there is potential for optimizations to be made on account of having a fixed grid of voxels. Fortunately, there is a more performance-oriented approach

known as DDA. DDA takes advantage of this fixed-size grid by ensuring that each step along the ray equates to a movement of exactly one voxel in one and only one of the cardinal directions. This is done by calculating how a step size of one voxel unit along each cardinal direction will affect the length of the stepped ray; then choosing the axis that results in the smallest ray distance increase, to not miss voxel checks. This is a vast improvement over the naive approach as the voxel step size is maximised without skipping checks while the number of voxel checks is minimised as each cell is only checked once. A follow-up paper [6] by the authors of "Efficient sparse voxel octrees" cited previously visualises these unit size, axis aligned steps in the figure shown below. This figure shows how the DDA algorithm can take advantage of the fixed grid of voxels by always performing the minimum number of voxel checks at exactly one per ray step.

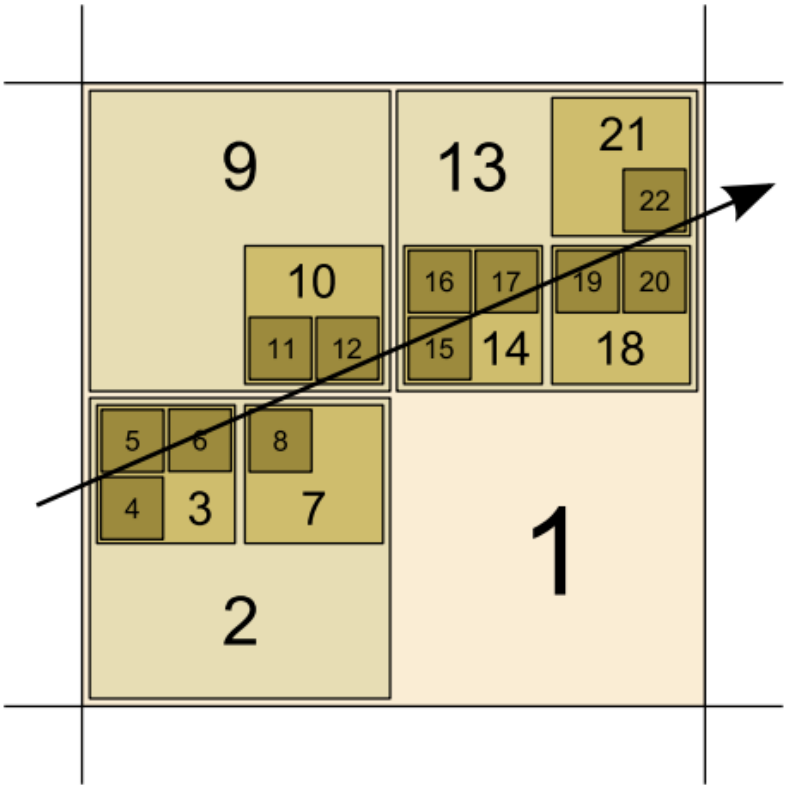

Figure 3: Visualisation of voxels checked when performing the DDA algorithm

2.2.4 Shading

Another advantage of the DDA algorithms stepping approach is the increased accuracy; since the hit position returned will also be the exact position on the boundary of a voxel. Whereas, with a fixed step size approach, the hit position could land anywhere within a voxel's bounds. The benefit of knowing the exact hit position is realized when it comes to handling lighting and shading calculations, as it aids in calculating the volume's normal at the hit position. In 3D graphics, the normal vector at a position is a vector that is perpendicular to the surface at that point. As seen in the paper "Fast phong shading" [7], the normal direction is foundational in calculating even the most simplistic of lighting models. It allows us to determine how much light is reflected off a surface and is needed for anything more than simple diffuse shading. In the figure seen below, the normal can be seen as the vector perpendicular to the surface at the hit position.

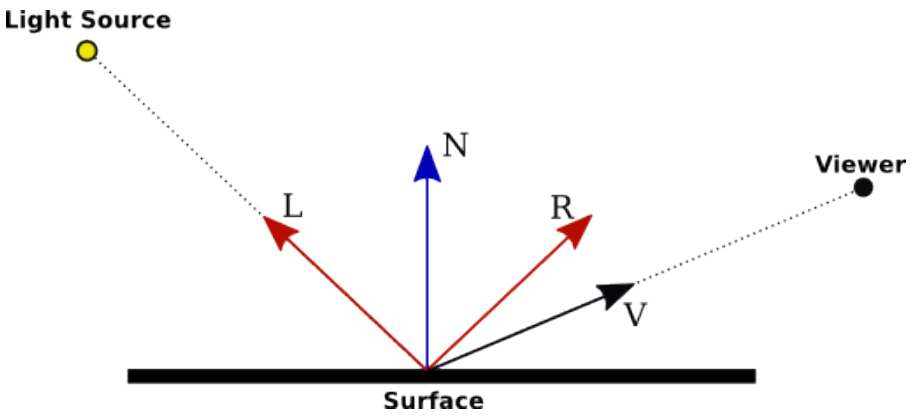

Figure 4: Important vectors involved in Phong shading

When shading voxel volumes, two approaches are most commonly taken, depending on the style of render being targeted. If trying to provide a smoother representation of a real-life object, such as a medical scan or cloud volume, normals might be stored directly in the voxel data. This way, no matter the "face" of a voxel a particular ray hits, the resulting hit position at each will share the same normal value. This makes the voxel nature of the render less prominent, as there are no shading differences between voxel "faces". Alternatively, if one wishes to embrace the voxel art style of the render or if modelling something cube-like already, such as a house, for this case, normals can be calculated in real-time. For this voxel rendering engine, the second approach will be targeted for two reasons. Firstly, to highlight the voxel style, and secondly, given the already high memory requirement for voxel data, it is ideal to keep the amount of data stored for each voxel to a minimum. As mentioned previously, the DDA algorithm is very beneficial here, as the required normal is essentially calculated as a by-product of the stepping process. Since the algorithm walks along the voxel boundaries, essentially in the same zigzag nature you see when calculating Manhattan distance, we know which side of the voxel was hit. Combining this with the sign of the ray's direction vector, the normal vector at the hit position can be derived.

3 Methodology

As stated at the beginning of the literature review, due to the organisational issues of Airbus and the falling through of our industry partnership, development progress on this thesis was delayed on account of having to redo the literature review. Despite this, a functioning voxel renderer was still created, although sadly many of the optimisations and more advanced techniques mentioned here were not able to be implemented as of yet within the given time frame. This section will outline the technologies that were used, as well as the implementation details of the voxel renderer in its current state.

3.1 Technology Stack

3.1.1 Rust and Bevy

Upon initial research into the topic of voxel rendering, it was promptly observed that there would be a serious need for highly performant code if this application was going to run in real-time in the browser. For this reason, as discussed above, this engine was initially written in Rust using the Bevy Entity Component System (ECS). This choice was made based on the high performance of Rust, its memory safety, and developer appeal as seen in the most recent StackOverflow developer survey where it ranked number one as the most loved language [8]. Bevy was chosen for similar reasons, it is written in Rust and would provide an abstraction on top of the very performant WebGPU graphics API. With both of these technologies being written in Rust, there was also the potential for compilation to Web Assembly which would allow the CPU side of the engine to run at native compiled speeds instead of being Just in Time (JIT) compiled by the JavaScript runtime. Despite the incredible performance potential of this technology stack, as mentioned above, due to severely lacking documentation, and the continually evolving specification of WebGPU while it is being finalised; development was extremely challenging and deemed non-production ready as a result of this. Observing the consequences of adopting bleeding-edge technology, a step back was taken and the timeline of this project was reassessed to reconsider how this implementation could be approached.

3.1.2 JavaScript, ThreeJS and React Three Fibre

Given the already short project timeline and after encountering the issues described above; it was decided to switch language to a more stable development platform. The language that was chosen was JavaScript, this was the choice for two primary reasons. Firstly, JavaScript (specifically TypeScript) is the language I am most familiar with myself, which would be beneficial when accounting for the shortened project timeline as minimal time would be spent learning language-specific features. Secondly, JavaScript is the native language of the web, making it a natural choice for a browser-based application. With JavaScript replacing the language, a replacement would also be needed for the Bevy framework that was previously being used. The replacement that was settled on was ThreeJS, an extremely popular JavaScript library that wraps the WebGL API the same as Bevy wrapped the WebGPU API. In the modern web landscape, it is almost always commonplace to use some sort of frontend library when building any sort of web app. For this project, I chose NextJS because of its popularity, my familiarity with it, as well as its ability to generate static sites which would be beneficial for hosting the final application. In addition to this, I also utilized React Three Fibre as an additional wrapper around ThreeJS allowing me to specify a scene using React components within my NextJS application. This switch to an interpreted language will likely have some kind of performance impact, but given the time saved by not having to learn a new language, and the increased stability of the development environment, it was deemed a worthwhile trade-off. In addition to this, being a predominantly GPU bound application, the performance impact of the language choice is likely to be minimal.

3.2 Voxel Data Generation

Delving into the working implementation that has been created as part of this project. Before anything can be rendered to the screen, a voxel volume must be created. Numerous formats have been created by the community to store voxel data, such as that used by the MagicaVoxel modelling program [9]. Although being in the proof of concept stage, it is currently not necessary to implement this functionality and this has been designated as a future improvement if the project is deemed to be feasible for web platforms. Instead, a custom minimalistic approach was taken, as when implementing a new system, less complexity leaves less room for error and allows for easier debugging during the development process. As eluded to above, the format used in this paper simply stores a single number per voxel, whether it is filled or not. The code snippet shown below generates a test scene in the form of a sphere centred around the voxel volume's origin.

```
const uniforms = useMemo(() => {
  const targetResolution = 21;
  const resolution = targetResolution | 1;
  const maxSteps = 100;
  const stepSize = 0.01;
  const voxelSize = 1 / resolution;
  const data = new Uint8Array(resolution * resolution * resolution);

  let i = 0;
  for (let z = 0; z < resolution; z++) {
    for (let y = 0; y < resolution; y++) {
      for (let x = 0; x < resolution; x++) {
        const radius = (resolution - 1) / 2;
        const dx = x - radius;
        const dy = y - radius;
        const dz = z - radius;
        const isFilled = dx * dx + dy * dy + dz * dz <= radius * radius;
        data[i++] = isFilled ? 1 : 0;
      }
    }
  }

  const texture = new Data3DTexture(data, resolution, resolution, resolution);
  texture.format = RedFormat;
  texture.needsUpdate = true;

  const uniforms: Uniforms = {
    resolution: { value: resolution },
    maxSteps: { value: maxSteps },
    stepSize: { value: stepSize },
```

```
      voxelSize: { value: voxelSize },
      voxels: { value: texture },
    };

    return uniforms;
  }, []);
```

Observing the code snippet above, several notable implementation details should be discussed. Firstly, is the use of the React useMemo hook; this ensures that this volume generation logic is only run once on the component's initial render. This is controlled by the final line of the snippet where the dependency array has zero elements. This essentially caches the volume and prevents React component updates and hot reloads from regenerating the volume would be inefficient. Secondly is the "targetResolution" variable; this is the result of a stylistic decision that I made. I would like the voxels volumes rendered by this engine, to be centred around the middle point of a single voxel to aid in intuitive design, positioning and interaction as opposed to having objects offset to one side. Given an integer value for the targetResolution, performing a bit-wise OR with a value of 1 will always produce an odd-numbered resolution which is needed to achieve this effect. Related to this, when calculating the spheres radius, a value of resolution minus one is used. This calculates a sphere slightly smaller than the volume as this extra headroom is needed to later centre the voxels by subtracting half their size in the shader. The third and final notable implementation detail is the use of a Uint8Array for storing the voxel data. This is a typed array that stores 8-bit unsigned integers, in JavaScript this is the most efficient form of array available. Since the number and size of elements are both known, this allows the interpreter to allocate enough memory for the array in advance, as opposed to a traditional array which is dynamically allocated. The use of this array ensures that the voxel data is stored in a contiguous block of memory, this is beneficial as it allows for faster operations by utilising the cache more efficiently. Having now declared all the data needed to render this voxel sphere, this information is packed into the uniform object and transferred to the GPU where this information will be rendered.

3.3 Ray Tracing and Step Size

Within the shader, we must first receive the uniform values we have passed in as well as the camera position and worldPosition. The worldPosition of our pixel is calculated in the vertex shader and passed in into our fragment shader from there. As seen in the code snippet below along with our received uniforms, the worldPosition is the result of matrix multiplication with the pixels model space coordinate, multiplied by the model transform.

```
  worldPostion = (modelMatrix * vec4(position, 1.0)).xyz;

uniform vec3 cameraPosition;

uniform int resolution;
uniform int maxSteps;
uniform float stepSize;
uniform float voxelSize;
uniform sampler3D voxels;

in vec3 worldPostion;
```

An interesting implementation detail here is that our Uint8Array passed from JavaScript is interpreted by the shader as a sampler3D object. This is due to the fact this data has been parsed as a 3D texture by the GPU, allowing us to index into it with normalised 0 to 1 coordinates to sample the voxel state at a given point. Having now loaded all the required information into GPU memory and obtaining references to them in our shader, we can progress to the main logic of the voxel rendered. The algorithm starts as seen below, where a ray is created with its origin at the camera and pointing to the worldPosition of our current pixel (somewhere on the surface of our proxy geometry).

```
void main() {
  Ray ray = Ray(cameraPosition, normalize(worldPostion - cameraPosition));
  RaycastResult result = rayCast(ray);
  if (all(isinf(result.hitPosition))) {
    fragColor = vec4(0.0);
    return;
  }
```

```
  // fragColor = vec4(vec3(normaliseDepth(result)), 1.0);
  fragColor = vec4(findNearestVoxelIndex(result.hitPosition), 1.0);
}
```

This small function encapsulates the whole renderer, we can say that if the result of the ray cast is infinity, the pixel will return an all-black value represented by the vec4(0.0) where the 4th element indicates an alpha channel of 0 meaning the pixel is transparent. The visualisations produced in the case of a voxel hit will be discussed later in this report during the results section. Flowing on from this, the ray cast function can be seen implemented below. Sadly, as discussed previously in this report, time constraints as well as development difficulties did not allow for implementation of the more efficient DDA algorithm; hence, this less accurate simpler method uses a fixed step size. Despite this, the method can produce a visually pleasing result and is a good proof of concept for the project as it represents a worst-case scenario. A DDA implementation would only improve on and deliver more promising performance for a browser implementation and is the number one priority in terms of future implementation. The step size used here is a constant value of 0.01, this was chosen as it is small enough to produce a smooth result but large enough to not be too computationally expensive. This value could be made smaller to increase accuracy at the cost of performance, or larger to increase performance at the cost of accuracy. Although this value cannot be increased in isolation as it has a high interaction with the maxSteps variable. Having a small step size and a low max steps will yield poor results as the ray will not be able to sufficiently explore the voxel space before running out of iterations which can lead to only the surface or highly extruded pieces of the volume being rendered.

```
RaycastResult rayCast(Ray ray) {
  for (int i = 0; i < maxSteps; i++) {
    vec3 testPosition = worldPostion + ray.direction * stepSize * float(i);
    float voxel = lookupVoxel(testPosition);
    if (!isinf(voxel) && bool(voxel)) {
      return RaycastResult(testPosition, length(testPosition - ray.origin));
    }
  }

  return RaycastResult(vec3(INF), INF);
}
```

Having strictly followed the Don't Repeat Yourself (DRY) programming principle and implementing single-use composable functions as is best practice; each function is mainly responsible for a single core operation. The rayCast function shown above is responsible for the extension of the ray and gradual exploration of the voxel space. Hence we will now look at the lookupVoxel function, which handles bounds checking and extraction of the voxel data from the 3D texture. As seen below, we ensure the index position is in the range zero to one as if we sample outside of this range, the default behaviour of the sampler is to repeat our texture, meaning we will still receive data but it will be incorrect.

```
float lookupVoxel(vec3 position) {
  vec3 voxelIndex = findNearestVoxelIndex(position);
  if (!isInRange(voxelIndex, 0.0, 1.0)) { return INF; }
  float voxel = texture(voxels, voxelIndex).r;
  return voxel;
}
```

Progressing deeper to the actual index calculation seen in the snippet below, this function covers the conversion of the 3D coordinates of a sample position, to the zero-to-one range required by the voxel data texture lookup. As our proxy geometry is a unit cube centred at the origin, we add 0.5 to all components of the position vector to convert them to the range zero to one. Division via the voxel size (in world space units) then flooring the result converts this float value to zero index integer value. We then add half the voxel size to ensure we sample the centre of each voxel to avoid confusion if sampling on a voxel boundary. Finally, we divide by the voxel count to convert the value back to the zero-to-one range required by the sampler.

```
vec3 findNearestVoxelIndex(vec3 position) {
  vec3 normalise = position + vec3(0.5);
  vec3 quantize = floor(normalise / vec3(voxelSize));
  vec3 offset = quantize + vec3(0.5 * voxelSize);
  vec3 voxelIndex = offset / vec3(resolution);
  return voxelIndex;
}
```

4 Results

4.1 Visualisation

Having now explored the logic and implementation details behind this basic but highly extensible voxel render we are now able to view the results it produces. In the case of a voxel being hit, there are currently two implemented visualisations; the normalised depth, and the position of the voxel. Firstly the normalised depth can be seen below.

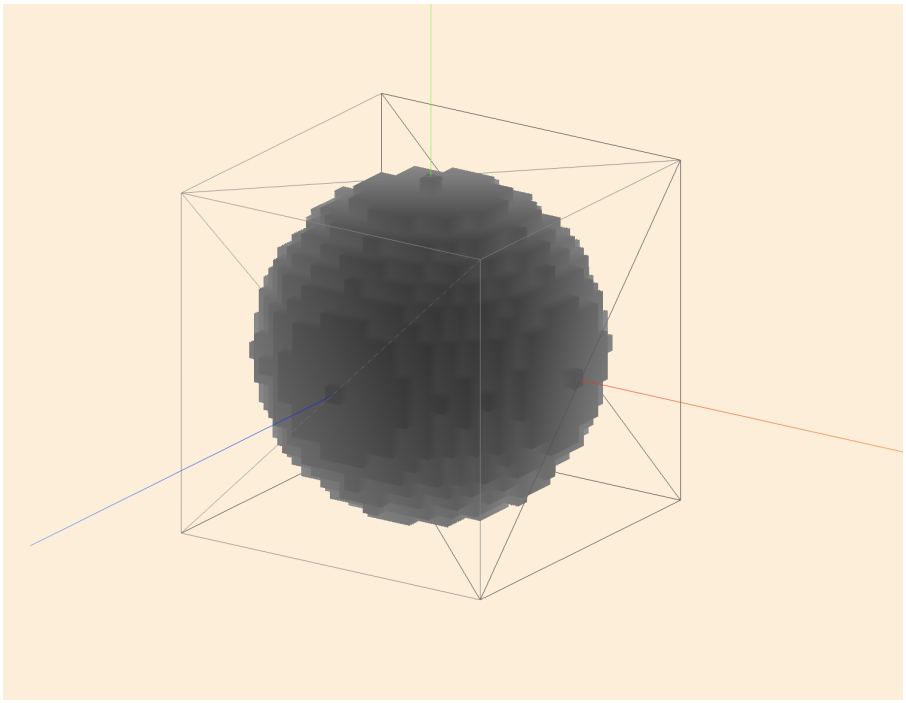

Figure 5: Visualisation of the normalised pixel depth.

This depth is said to be normalised as in theory, the ray could potentially progress infinity far into the scene but we are only able to represent colours in the range zero to one. To combat this, before visualisation, the depth value of the hit ray has the distance to the camera subtracted, then divided by the diagonal width of the voxel volume. This essentially maps the depth value to the range of zero to one, with zero being close to the camera (front of the voxel volume) and one being far away (back of the voxel volume). This logic can be seen implemented in the code snippet below.

```
float normaliseDepth(RaycastResult result) {
  float halfDiagonal = length(vec3(0.5));
  float cameraDistance = length(cameraPosition);
  float minDepth = cameraDistance - halfDiagonal;
  float maxDepth = cameraDistance + halfDiagonal;
  float normalisedDepth = (result.depth - minDepth) / (maxDepth - minDepth);
  return normalisedDepth;
}
```

The second visualisation that is currently implemented is the position of the voxel. This render simply colours each pixel by using the position of the voxel as its colour. This is possible as the proxy geometry is a unit cube centred around the original so all positions are already in the unit range of -0.5 to 0.5 and a simple addition of 0.5 produces the zero-to-one range of values required for non-clamped colouring. An interesting shading effect that arises due to this is that the volume appears slightly shaded as pixels closer to the origin (those making up inset corners) are shaded darker than those closer to the edge of the volume (those on outset corners).

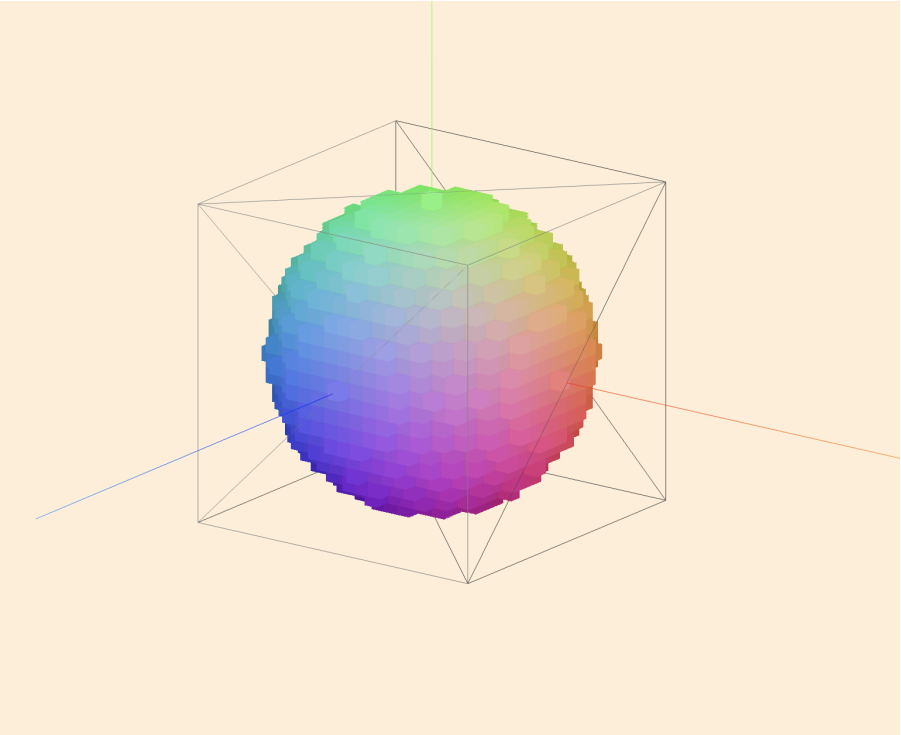

Figure 6: Visualisation rending the voxel volume colouring each voxel by its centre position.

A significant amount of time was invested in calculating per "face" normals so that proper shading could be achieved. The technique used for this was inspired by a forum post [10] suggesting taking the maximum component of the direction from the voxel centre to the hit position. Although, despite this insight and having tried an implementation of this, the result was not successful. This was likely attributed to the inaccuracies of the fixed step size ray marching approach being taken here. An attempt was also made at implementing the DDA so that correct and accurate hit positions could be obtained, although this implementation was not completed within the project timeline. This can be attributed to difficulties extrapolating existing documented 2D solutions into the 3D case, as well as converting calculations to use vector operations as opposed to calculating axis step components individually. The two primary resources followed when attempting to implement this algorithm were "Lode's Computer Graphics Tutorial - Raycasting" [11] and "An Overview of the Fast Voxel Traversal Algorithm" [12]; both of which are exceptionally well-written resources.

4.2 Performance

In terms of the measured performance of the solution, the renders seen above were created using the following key settings, maxSteps: 100, stepSize: 0.01, resolution (voxel): 21, resolution (display): 1080p. These settings are the key factors that change the render's performance. Given my systems configuration with an M1 Pro processor and 32GB of RAM, I was able to render these visualisations interactively in the browser at a locked 120 FPS (the maximum refresh rate supported by the laptop's in-built display).

4.3 Risk Management

Thankfully, being a software application, there is a negligible chance of physical risks associated with this project. Despite this, there are still some minimal data-related risks present which have been accounted for as follows. Data stored in a single place can be susceptible to lose from hardware failure or misplacing the physical device. To combat this, Git version control has been used to both keep track of incremental program changes on a line-by-line basis, as well as keep a cloud-based backup through the GitHub service [13]. Another crucial aspect that must be considered when working on a programming project is the data involved and the security and privacy expected when handling it. In the case of this voxel engine, there is no sensitive information currently being handled.

4.4 Ethical Considerations

Ethics are an important consideration when completing any project. For Engineers, we must look to the Engineers Australia Guidelines [14] to aid in our professional conduct. Firstly, regarding codes 1.1, 1.3, 3.1,

and 3.3, in summary, one must act in good conscience, respect dignity, uphold trustworthiness, and honestly communicate. This has been upheld throughout this project via diligent citation and referencing of a vast number of helpful resources, giving credit where credit is due. The codes described competent practising under section 2 which has also been upheld by my continued effort to stay up to date with the constantly evolving field of software and technology. I do this out of passion but also in the interest of always striving to produce a clean, maintainable software solution that follows best practices such as DRY mentioned previously.

4.5 Sustainable Development Principles

Sustainability is another key aspect that must be taken into account when undertaking any engineering project. Being a small software-based project it is fortunate that this project has a very limited impact in terms of environmental sustainability. The one main impact it does have however is the power consumption of the hardware it is run on. In an attempt to minimise this, developers such as myself must strive to write efficient code that effectively uses compute cycles. So far, this has been implemented in this project by utilising the GPU for per-pixel parallel computing.

Another crucial factor relating to sustainability when undertaking this project was the maintainability of the code. As any developer would know, when coming back to one of your projects, or picking up another developer's project, it can take a significant amount of time to become acquainted with the code base and understand what is going on. This is doubly true if the code was poorly written and documented. Taking note of this, to support sustainable development, it was a goal to write clean and sustainable code from the beginning. This means comments within the source code, meaningful commit messages in version control, progress documentation such as this, as well as clean code structure with sensible variable names.

4.6 Key Stakeholders

After accounting for the issues we had with our industry partners, there are now fewer stakeholders in this project. Aside from myself and the academic experience this project will impart to me, there are no other third parties that are significantly invested in the outcome of this thesis. My supervisor, Professor Clinton Fookes, would hold a limited amount of stake in this project as its success or failure could potentially reflect on his supervisory presence. Although his support so far through this project's late changes has been warmly welcomed. Since it has ultimately changed to be a student-led project, I suspect his and the investment of Queensland University of Technoloogy (QUT) will ultimately be limited.

5 Conclusion

Having only a single computer, I cannot attest to performance on other devices, although given that this render performs at double the useful interactive target FPS of 60, it would not be unreasonable to say that a computer with half the processing power would still be able to render this scene at an interactive rate around the 60 FPS mark. Especially once taking into consideration the performance improvements that could be achieved by the implementation of the more efficient DDA algorithm. This, coupled with the coming finalisation of the more efficient WebGPU specification and support of this new standard by ThreeJS, paves the way for many future performance improvements. To conclude, this thesis project aimed to assess the feasibility of creating a real-time voxel rendering engine capable of running in the browser. Having achieved twice the real-time FPS target despite using an inefficient algorithm, It can be observed via these results that this is indeed a feasible goal.

5.1 Future Work

As mentioned numerous times throughout this literary piece, several future improvements can be made to this work. Firstly and undoubtedly most beneficially would be the implementation of the DDA algorithm to allow for faster and more accurate ray casting when compared to the fixed step size approach currently implemented. Following on from this, as a by-product of the DDA algorithm, per "face" normal vectors will be known for each voxel allowing for stylised voxel shading as opposed to the position or depth-based shading present currently. Lastly, with the potential to turn this project into a completed solution, a more robust voxel data format would need to be implemented. Thus would likely still be a custom format internally but would include the implementation of file loaders for common formats such as that used for MagicaVoxel [9]. This would allow for the import of existing voxel models into the engine such as game assets, medical scans, or any other content in the supported format.

A Project Timeline

The table included below outlines all deliverables that were completed as part of this project as well as the presentation remaining. It also specifies their dependencies along with the expected completion date. A number of these items are outlined by QUT as part of completing an honours thesis in engineering, helping provide good structure to the research. With regards to the interim milestones, these have been goals that I have set myself with the help of communication with the project coordinator, Professor Clinton Fookes and have helped keep the project on track; guiding the production of a working implementation upon completion of this unit.

Number	Name	Due	Dependency	Type	Status
1	Project Proposal	W07 Sem 2, 2022	N/A	Assessed	Completed
2	Get shader working	W10 Sem 2, 2022	1	Interim	Completed
3	Implement ray marching	W11 Sem 2, 2022	2	Interim	Completed
4	Render volume to cube	W12 Sem 2, 2022	3	Interim	Completed
5	Progress Report	W13 Sem 2, 2022	4	Assessed	Completed
6	Oral Presentation	W03 Sem 1, 2022	5	Assessed	Completed
7	Generate voxel sphere	W08 Sem 1, 2023	6	Interim	Completed
8	Implement ray marching	W10 Sem 1, 2023	7	Interim	Completed
9	Produce shaded render	W12 Sem 1, 2023	8	Interim	Completed
10	Final Project Report	W13 Sem 1, 2023	9	Assessed	Completed
11	Final Oral Presentation	End Sem 1, 2023	10	Assessed	Incomplete

References

[1] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010, pp. 55–63.

[2] candr, "Tuxedo labs teardown technical teardown - twitch vod 26/11," Dec 2020. [Online]. Available: https://www.youtube.com/watch?v=0VzE8ROwC58

[3] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann, "Geometry and attribute compression for voxel scenes," *Computer Graphics Forum*, vol. 35, no. 2, pp. 397–407, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12841

[4] Microsoft, Apr 2023. [Online]. Available: https://www.minecraft.net/en-us

[5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 2009, pp. 15–22.

[6] S. Laine and T. Karras, "Efficient sparse voxel octrees–analysis, extensions, and implementation," *NVIDIA Corporation*, vol. 2, no. 6, 2010.

[7] G. Bishop and D. M. Weimer, "Fast phong shading," *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 103–106, 1986.

[8] StackOverflow, "Stackoverflow developer survey," 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021

[9] ephtracy, "voxel-model/magicavoxel-file-format-vox.txt at master · ephtracy/voxel-model," 2015. [Online]. Available: https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox.txt

[10] zCybeRz, "How do i calculate the surface normal of a voxel?" 2022. [Online]. Available: https://www.reddit.com/r/GraphicsProgramming/comments/tne9y7/comment/i21g1b4/

[11] L. Vandevenne, "Lode's computer graphics tutorial - raycasting," 2020. [Online]. Available: https://lodev.org/cgtutor/raycasting.html

[12] cgyurgyik, "fast-voxel-traversal-algorithm/fastvoxeltraversaloverview.md at master · cgyurgyik/fast-voxel-traversal-algorithm," 2020. [Online]. Available: https://github.com/cgyurgyik/fast-voxel-traversal-algorithm/blob/master/overview/FastVoxelTraversalOverview.md

[13] GitHub, "Github," 2022. [Online]. Available: https://github.com

[14] E. Australia, "Code of ethics," 2022. [Online]. Available: https://www.engineersaustralia.org.au/publications/code-ethics