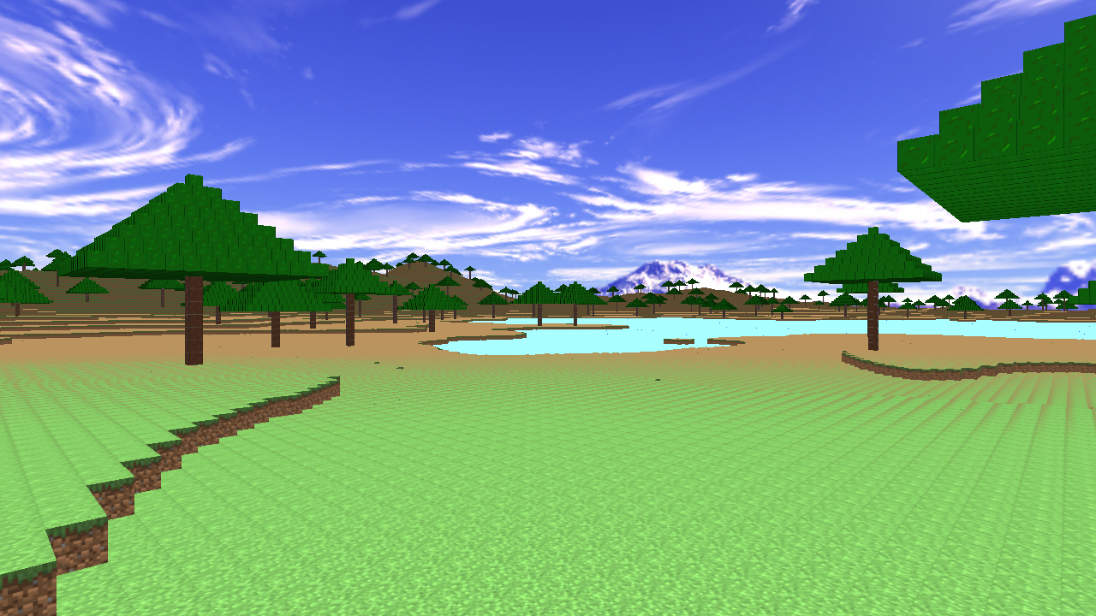


Computer Games Development

Project Report

Year IV

“An Investigation into the Implementation of a Voxel-Based Map Generator”



Alan Bolger

Student No: C00232036

Date of Submission: 04/05/2021

# Contents

[Contents 1](#_Toc71097604)

[Project Abstract 2](#_Toc71097605)

[Literature Review 2](#_Toc71097606)

[Evaluation and Discussion 7](#_Toc71097607)

[Project Milestones 12](#_Toc71097608)

[Major Technical Achievements 12](#_Toc71097609)

[Project Review 12](#_Toc71097610)

[Conclusions 13](#_Toc71097611)

[Future Work 13](#_Toc71097612)

[References 13](#_Toc71097613)

[Appendices 13](#_Toc71097614)

# Project Abstract

*“An Investigation into the Implementation of a Voxel-Based Map Generator”*

*Alan Bolger*

This project aims to find out what the best practices are when building a voxel-based map generator, and what programming techniques can be used to improve the storage efficiency and overall rendering performance. Research will have to be carried out on several different subjects to aid in this. These include:

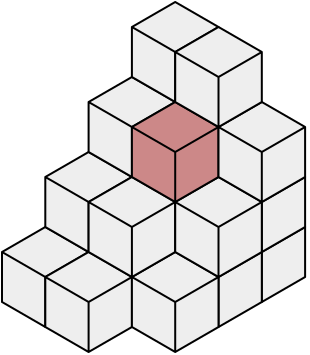
* Voxel Data Storage
* OpenGL (Graphics API)
* Ray casting
* Ray tracing (Experimental)
* Procedural Generation of Maps (Noise Maps)
* UI Implementation (WIP)

# Literature Review

The following are literature subjects that were investigated during this project:

* **Voxels**

A voxel is a data point that is stored on a uniformly spaced 3D grid. The name came from a combination of the words ‘volume’ and ‘pixel’. A voxel is similar to a pixel in the sense that they do not usually have their position defined in their dataset. When rendering voxels, the position of a voxel is known by its position in relation to its surrounding voxels.

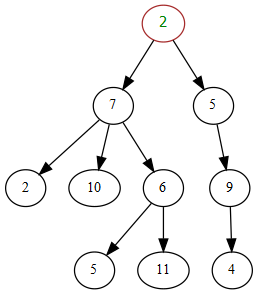


*A series of voxels in a stack, with a single voxel shaded.*

By Vossman; M. W. Toews - Own work; originally created in w:Adobe Illustrator, and later in a text editor, CC BY-SA 2.5, <https://commons.wikimedia.org/w/index.php?curid=1313585>

Voxels have been used in many games over the past several years and are normally associated with blocky type games (such as Minecraft), but voxels don’t have to be cubes. Voxels are really only a point, and this point is only relative to the other voxels around it, which means it is up to the game renderer to decide the dimensions of the voxel and the space between each voxel.

* **Voxel Data Storage (Octrees)**



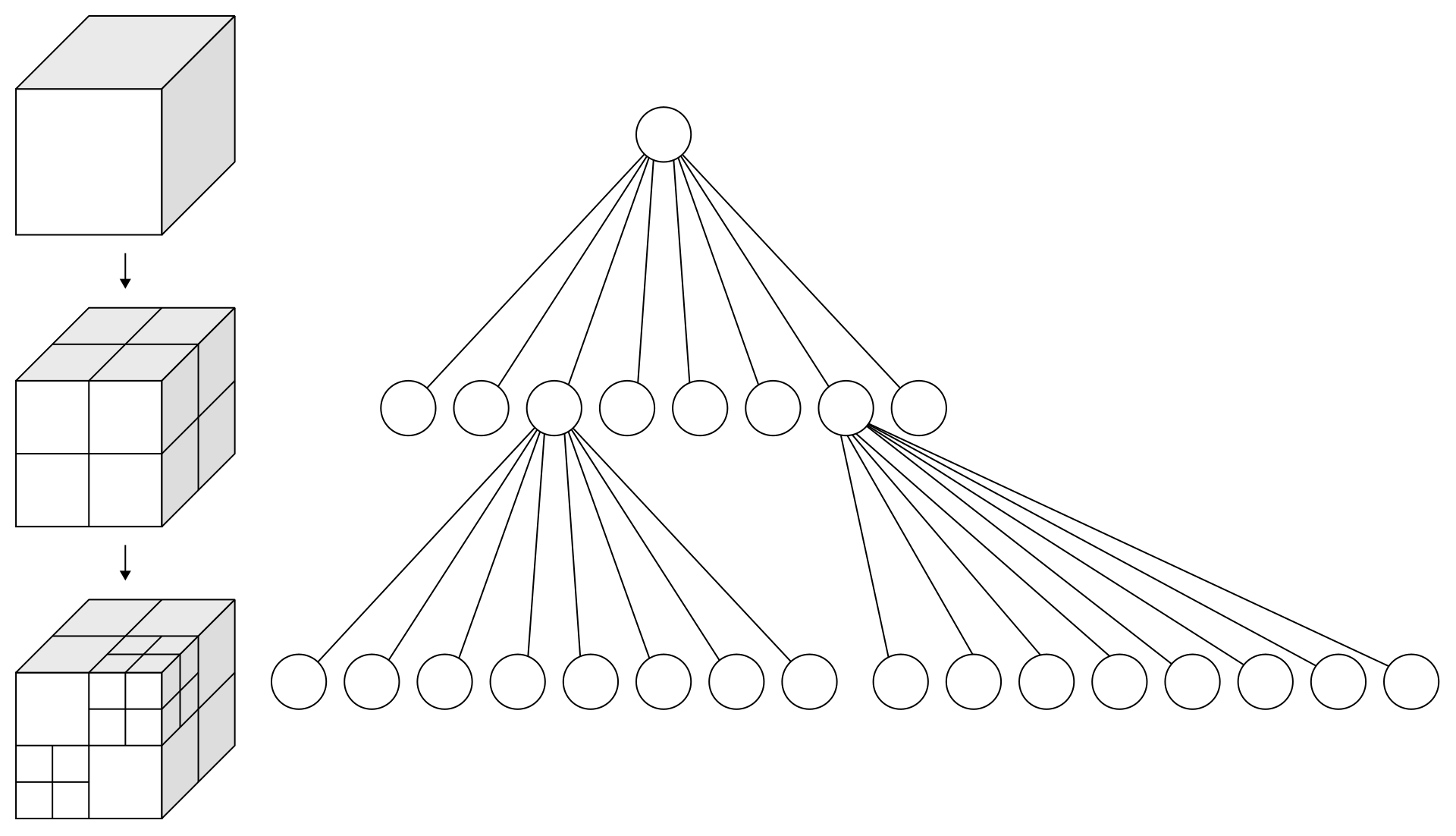
*A diagram of a tree data structure.*

By Paddy3118 - Own work, CC BY-SA 4.0,

<https://commons.wikimedia.org/w/index.php?curid=83223854>

Octrees are a type of tree data structure. A tree is a data structure that mimics a hierarchical tree structure (see diagram above) and has parent nodes with subtrees of children and a root value.

In an octree data structure, each internal node has eight children. Octrees are widely used as a way to divide 3D space into smaller sections by subdividing nodes recursively into eight octants.



*A diagram of the subdivision of an octree’s nodes.*

By WhiteTimberwolf, PNG version: Nü - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=9851485>

Point region octrees store a 3D point, or position, in each node. This point is the centre point for the subdivision of that node. Each of the eight children of a node represent a corner point, and these corner points make up the node’s bounding box. In matrix-based octrees, it is the centre point of the area that the node represents which is used as the subdivision point.

At their lowest level, an Octree’s structure matches the uniformly sized cells that are needed to store a voxel world. They can also provide tight compression of sparsely populated areas. These particular octrees are called Sparse Voxel Octrees (or SVOs).

However, the time complexity of searching and inserting to an octree is O(log(N)) where N is the number of octree nodes.

* **OpenGL (Graphics API)**



By OpenGL - https://www.khronos.org/legal/trademarks/, Public Domain,

<https://commons.wikimedia.org/w/index.php?curid=98607430>

To render the generated map, the OpenGL API is used. OpenGL is a cross platform and cross language API (application programming interface). It is used to render both 2D and/or 3D vector-based graphics. Its main use is to communicate with a GPU (graphics card). OpenGL was released by Silicon Graphics in 1992. Since 2006, Khronos Groups manages OpenGL.

This project uses voxels to store the terrain data. When rendered, the voxels are represented by cubes. These cubes are polygonal models with textures wrapped around them.

As there are a large number of voxels to be rendered, drawing each cube individually would create thousands, if not millions of draw calls. Draw calls are an expensive operation because every time you draw, the CPU has to communicate with the GPU to let it know which buffer to use for data, where vertex data is stored, etc. This is actually quite slow to achieve. To overcome this, a rendering technique called **instancing** is used instead. When using instancing, an array of all the voxel’s world positions is sent to the GPU shader. The shader is notified of how many cubes are being instanced and then uses the array of voxel world positions to render the cubes. This is done using just one draw call which makes it far more efficient than using separate draw calls. Instancing only works if each model being rendered is the same. As we have 4 different types of voxel in this project, we have to use 4 different instancing calls as the texture is changing on each voxel.

**Voxel Data Storage (Octree Alternative)**

An alternative to using an octree would be to use an array-based storage system. An array is a data structure that contains elements, with each element storing some data. In additional to being single dimensioned, arrays can also be stored and accessed as 2-dimensional arrays (X and X index positions) or 3-dimensional arrays (X, Y and Z index positions).

If a 3d array was used to store voxels, then the X, Y and Z index position of any element storing a voxel in the array would be the voxel’s world position. Also, arrays are extremely fast when accessing any given element, and the time complexity of accessing an element is O(1), which means no matter how big the array is in size, it will always take the same amount of time to access any element in that array.

Arrays tend to take up more memory than octrees, but this problem can be overcome with some good memory management.

***This is the method I chose for storing voxels as it is quicker to access voxels and can be customised to take up minimal memory.***

* **Ray Casting**

Ray casting is normally described as a computer graphics rendering technique that uses the same basic ray projection algorithm as ray tracing (see below). In this project, ray casting is not used for rendering, but is instead used for user interaction with the voxel world (using the mouse).

Ray casting works by projecting a ray from the observer’s eye (which in this case is actually the mouse coordinates) and finding the closest geometric object that the ray intersects with.

Ray casting in this project works by casting a ray whenever the user clicks the left or right mouse button. As we are using ray casting for mouse picking and not rendering, only a single ray needs to be cast (instead of a ray for each pixel on the screen which would be the case if rendering using ray casting). The origin (or starting point of the ray) is the current mouse coordinates when the user clicked the mouse button. The ray is then projected through the scene and an intersection point is returned if the ray is blocked by any objects in the scene. The intersection point can then be used to identify which object is being hit with the ray. The object can then be manipulated in whatever way is necessary.

* **Ray Tracing**

Ray tracing is a 3D computer graphics rendering technique that is very similar to ray casting because they essentially use the same algorithm for casting/tracing rays. Ray tracing can provide extremely detailed rendering when compared to typical rendering techniques. The main difference between ray casting and ray tracing is that ray casting (when rendering graphics) does not use recursion and ray tracing does.

Ray tracing works by projecting a ray from the observer’s eye for each pixel on the screen and finding the closest geometric object that the ray intersects with. As the ray is traced and travels through the scene, it simulates any interactions that the ray has with any other objects. When the ray eventually intersects with an object, it can then be reflected off the intersected object and will continue tracing through the scene in the new direction. It will continue to simulate any interactions that the ray has with other objects until it intersects with another object and the process is repeated. The number of times a ray can reflect off another object is called the recursion level. The greater the amount of recursion, the more detailed the rendering will be. However, greater recursion levels result in slower rendering as it requires more computations.

In this project, ray tracing is experimental and is only used as a possible additional way of rendering the scene in 3D.

* **Procedural Generation of Terrain Maps**

Procedural generation, in computer science terms, is the process of generating data by utilising algorithms. By procedurally generating maps, it means each time a new map is generated, it will be different from the previous map that was generated, providing a different seed is used each time. The seed value is a numerical value that can be entered manually or randomly generated. This means that using the same seed will generate the same map every time.

For this project, procedural generation is used to generate terrain maps. A terrain map is a 2D array of height values. The world is made up of a uniform 3D grid containing voxels that each have X, Y and Z positions. Each height map array element holds a value of between 0 and 1. To represent these heights in a 3D world, the X and Z position of any voxel in the world map is equal to the X and Y index value of the array. The Y position of the voxel is then represented by the value stored in that particular array index multiplied by a modifier value to scale it suitably to a world position.

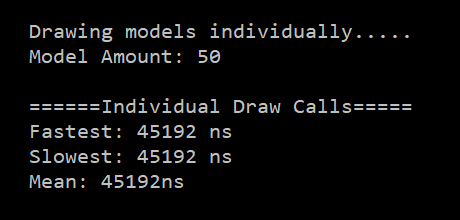
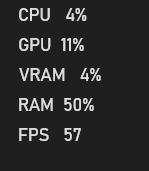
The height/terrain map values are created by using simplex noise, which is a type of gradient noise that is similar to Perlin noise. This is implemented through a noise function that takes the X and Y values of a map as function inputs, and based on these inputs, produces a value between 0 and 1. This means that the noise function will produce the same value for the same inputs every time.

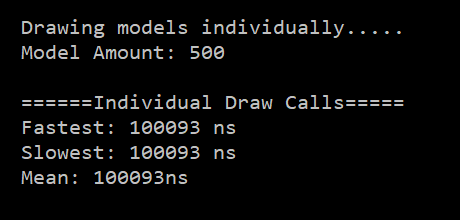
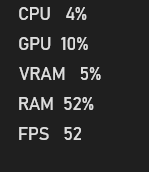
Procedurally generating maps has some advantages over manually creating maps. By using procedural generation, map data does not have to be stored with the application executable as it is generated after the application is run. This reduces the amount of data storage that the application needs. Procedural maps are also easy to save, as technically there is no actual saving – all you need to know is the seed that was used to generate the map.

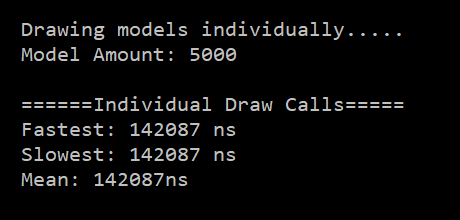
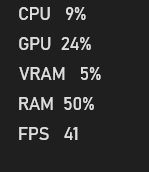
# Evaluation and Discussion

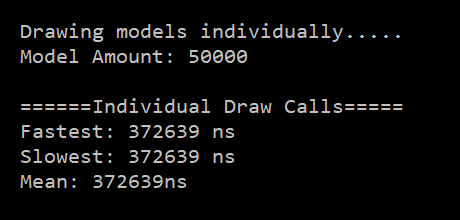
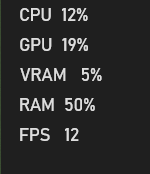
* **OpenGL Rendering of Models**

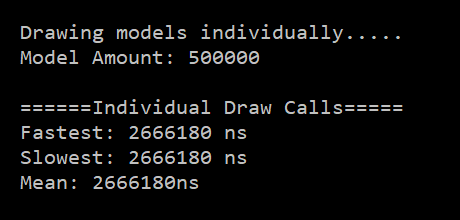
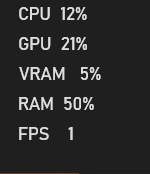
Two different methods were tested to examine GPU performance. The first set of tests examined using **individual draw calls** to render the cube models. On the left is how long the draw calls took, and on the right is the performance impact.

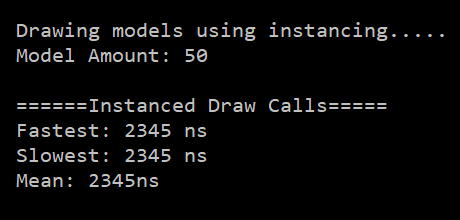
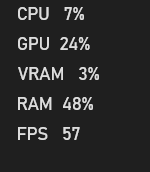
 

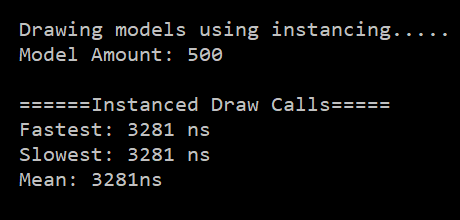
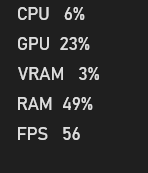
 

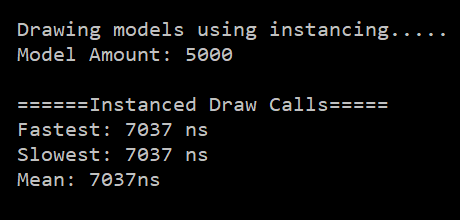
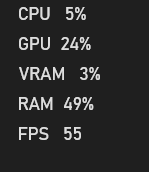
 

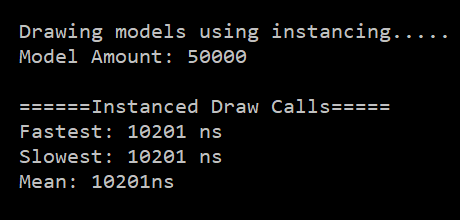
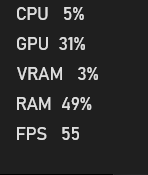
You can see that as the amount of individual draw calls increases, the performance drops significantly. By the time we reach 500,000 draw calls, the frame rate has dropped to 1 FPS and the time taken to complete the draw calls has reached 2666180 ns

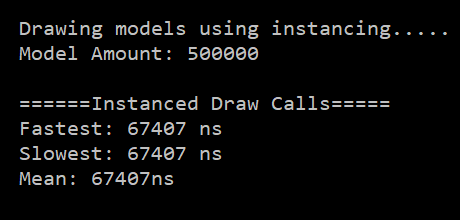
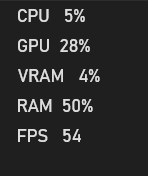
The second set of tests examined using instanced draw calls to render the cube models. On the left is how long the draw calls took, and on the right is the performance impact.

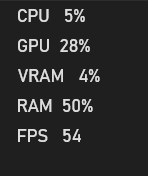
 

Looking at these results shows that using instanced draw calls instead of individual draw calls is far more efficient. Even when drawing 500,000 models, the performance impact is minimal.

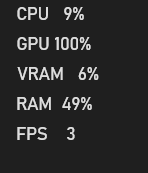
The bar chart above shows the time taken in nanoseconds to draw several different amounts of models.

* **Standard Rendering vs Ray Tracing**

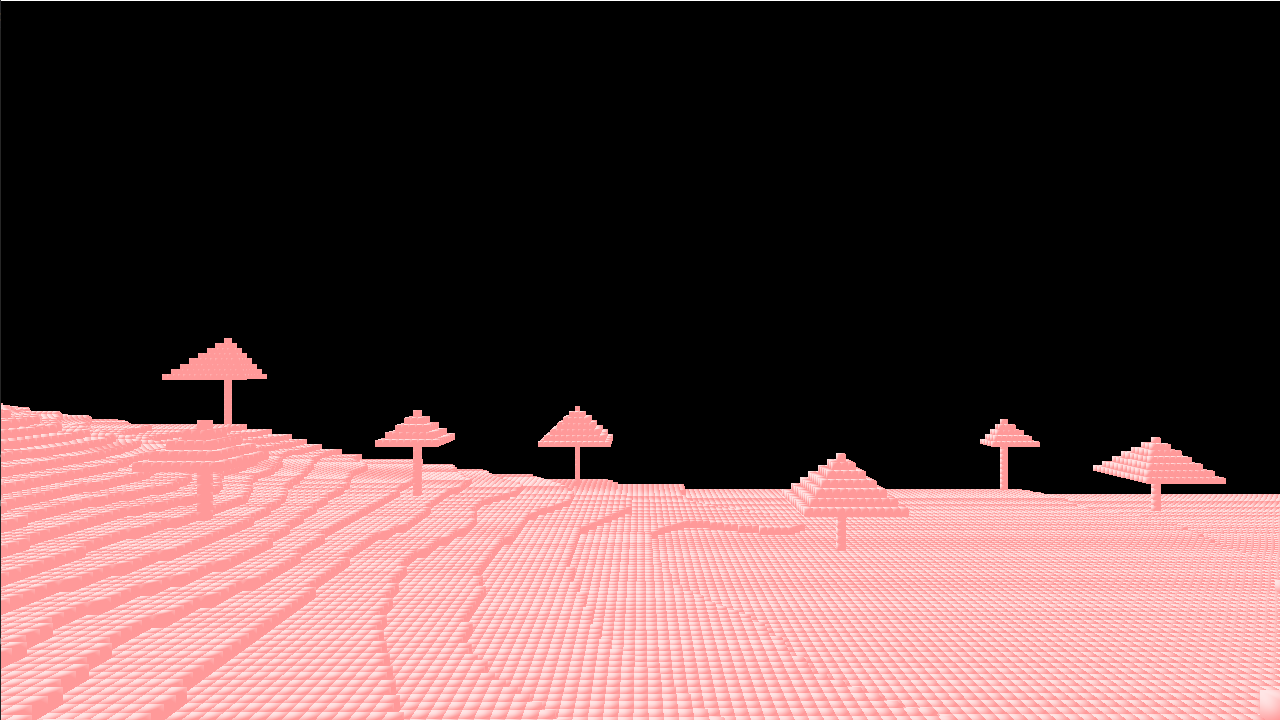
Standard rendering using **instancing** gave the following results:



However, when attempting to ray trace even a small part of the map, the performance results were extremely poor as shown below. The reason the GPU is at 100% is because all of the ray tracing was executed on the GPU’s compute shader. If this had have been executed on the CPU instead, the application would have locked up and become unresponsive.



So even when using the compute shader, which uses parallel processing to carry out the ray tracing, the frame rate is unacceptable. Therefore, it is not viable to use ray tracing for a secondary form of rendering for this project.



*Shown above is the scene rendered using ray tracing.*

# Project Milestones

**Milestone 1**

Adding OpenGL functionality to efficiently draw large amounts of models without performance impact.

Target time for completion: December 2020 (Completed)

**Milestone 2**

Creating a terrain generator that can produce height maps.

Target time for completion: February 2020 (Completed)

**Milestone 3**

Adding efficient voxel storage system.

Target time for completion: April 2020 (Completed)

# Major Technical Achievements

For me personally, as a huge Minecraft fan, I was very pleased to be able to make a map generator that functions in a similar way to the method that Minecraft uses to generate maps. Utilising noise functions to create terrains I

I understand that Minecraft is much more advanced in the way it does things, but this project has shown me what is necessary to go forward and improve upon it, and possibly add more advanced methods of reading and rendering voxel data.

# Project Review

Overall, I found the project to be challenging. One thing that did not work correctly was the ability to delete and add voxels to the world using a mouse. This had been working correctly before the optimised storage system was implemented, but unfortunately it stopped working after that and was causing errors that I could not fix, so I had to disable it.

If I were to start the project again, I would possibly use the Vulkan API instead of OpenGL. This would make it possible to utilise an RTX GPU to render the map using ray tracing.

# Conclusions

On completion of this project, I found that ray tracing is unrealistic unless you are using an Nvidia RTX GPU and utilising its ray tracing cores.

# Future Work

If I were to add anything to this project, I would implement a storage system similar to the one already implemented for storing voxels, but it would be for storing the matrices that need to be sent to the GPU. Currently, if any changes are made to the map (and the map needs to be updated), then the entire map is copied to the instancing arrays and then copied to the GPU buffer. If the instancing arrays were part of the Chunk class, only the Chunk’s arrays would need to be updated and then sent to the GPU.

# References

**Voxels**

* <https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190>
* <https://en.wikipedia.org/wiki/Voxel>

**OpenGL**

* <https://learnopengl.com/>
* <https://www.khronos.org/opengl/wiki/Compute_Shader>

# Appendices

**This project was compiled and tested on a PC with the following specifications:**

*Processor* Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz

*Installed RAM* 16.0 GB

*GPU* Nvidia RTX 2060 6 GB

**Voxel Storage System**

The storage system used for storing the voxels is relatively straightforward. The system consists of three classes: The **World** class, the **Map** class and the **Chunk** class. When the project is run, a World object is created. The World object then creates an array of pointers to Map objects and the Map objects each create an array of pointers to Chunk objects.

After the terrain and scenery has been generated, the voxel storage is populated. The terrain map is split up equally so that each section fits into each existing Map in the World object and then once the data has been converted to voxel positions, the Chunks are populated with the terrain data. Then the scenery is added in the same way.

Once this has been completed, a check is done to optimise the storage. If a Chunk is completely empty – as in it is full of voxels representing air, it will be deleted. As a lot of the map will be made up of air, this greatly reduces memory usage. If a voxel is then added to a region of the World where no Chunks exists, a Chunk will be created for the voxel to be placed. Also, voxels are represented by a byte, which makes memory usage even less.

Using this system also speeds up copying voxel positions to the GPU, as when the copying is taking place, any null pointers in the Map can be ignored as this means there is no Chunk there to check.

Scroll further down to see a diagram of the storage system.

