



**WWU - Computer Science Department
CSCI 440/540 Virtual Worlds
Final Report– Spring 2024**

Voxel Terrain Generation (a.k.a. Re:Voxel)

Parker Gutierrez, Ian Cambridge, Saunder VanWoerden, Samuel Turney

1. Introduction

Motivation

Virtual worlds come in many shapes and sizes. Some are meticulously detailed, photorealistic environments planned and crafted with hours of dedication. Others are pixelated, procedurally generated landscapes created in just seconds. Our project falls into the latter category, aiming to contribute to the world of procedurally generated virtual environments.

Our goal is to develop an efficient, customizable, and unique voxel-based world generation engine using Unity.

In the end, our project will serve several purposes. For entertainment purposes, allowing users to explore or alter their world at will. For instance, if someone wanted to, they could build a castle. For development purposes, others can use our engine as a base for their own projects. They might want to generate some terrain using our engine, save it, then use that terrain in their own world. It can also be used as a base for testing implementations of a voxel engine or terrain generation, whether it be a different implementation of voxel storage, different mesh generation techniques, or an overhaul of a certain component of our engine. and stress testing all of the above.

A key priority for this project is optimizing performance, ensuring that our engine can serve as a robust foundation for other projects without imposing a significant overhead. Since Unity currently lacks a tool like this, we believe our project will be a valuable addition to the community.

With these goals in mind, we hope to provide a powerful resource for developers looking to create dynamic and engaging virtual worlds.

Quick Definitions:

- Voxel: the 3D analogue to a pixel. We're representing these as cubes in the world.
- Noise: a number generated by giving specific input parameters to a noise algorithm, designed to return values similar to its neighbors.

Challenges

Chunk mesh generation presents its own difficulties, especially math regarding the vertices of each mesh and the technical implications of keeping those meshes updated and synchronized across clients. We'll use Mirror for our networking implementation, which will eventually result in connecting remote components to the host. We need to make sure the chunks are memory efficient while at the same time retaining 100% detail at a close distance, while also being quick to generate and render. This is by no means a new problem but implementing it takes work. We're going to be implementing greedy meshing^[0] to solve these problems.

The problem of how to handle terrain generation (or how to make terrain *look* a certain way) will most definitely require some detailed research and applications of algorithms. The most relevant/challenging aspect here will likely be implementing these algorithms; implementing them will require intimate knowledge of both our own codebase/voxel engine (which is itself a challenge) as well as the algorithms themselves. In the end, we used spline graphs and multi-octave perlin noise as our terrain generation method of choice. See later on for the description of spline graphs and multi-octave perlin noise.

Networking is a very important problem that needs to be addressed. Challenges will include synchronizing worlds across the network including chunk data, world data, player data, and interactions. Furthermore, server-side commands may become a challenge, along with getting the UI associated clean and functional. We'll also have to handle and synchronize localized player render distance and negotiating with the host for data. The final challenge is a stretch goal: disk persistence. While it is great (and super fast) to read from the host's memory, it would be even better to have the host save unloaded chunks to disk. We didn't quite make it to this, as we had issues with reading and writing the VoxelRuns, but the host itself can function as something of a persistent server even without this addition.

An auto profiler is crucial in standardizing the way that performance testing is done so that you can compare results against previous profiler runs reliably. It should cover all use-cases of the program and also do some stress testing to ensure that all parts of the engine work properly. The real challenge is that an agent in the world needs the ability to act automatically, and each possible task needs to be created and defined as its own atomic task or by grouping subtasks together. So, the engine needs a system in place to enable this. This specific description is like a behavior tree for AI.

The challenge of getting this engine to be efficient enough to not notice any lag at all is deceptively complex. First off there's simply making sure that your systems are efficient. On its surface it sounds simple, but when a system grows and integrates more components to itself, it can become ridiculously complicated. It can be argued that proper system design can alleviate these difficulties, but as a general case it still holds. This isn't the only reason. In our case we're using the Unity job system to perform tasks independent of other systems quickly (chunk generation & meshing), but we found that with our current implementation, shuffling the data to and from job-friendly data types was extremely expensive for the main thread. Not to mention, when many jobs complete at the same time, there needs to be a way for the job manager to limit itself so it doesn't completely hold the main thread hostage while cleaning up every single finished job. It needs to decide if it should wait until the next frame to continue finishing jobs, which isn't something we solved well in time.

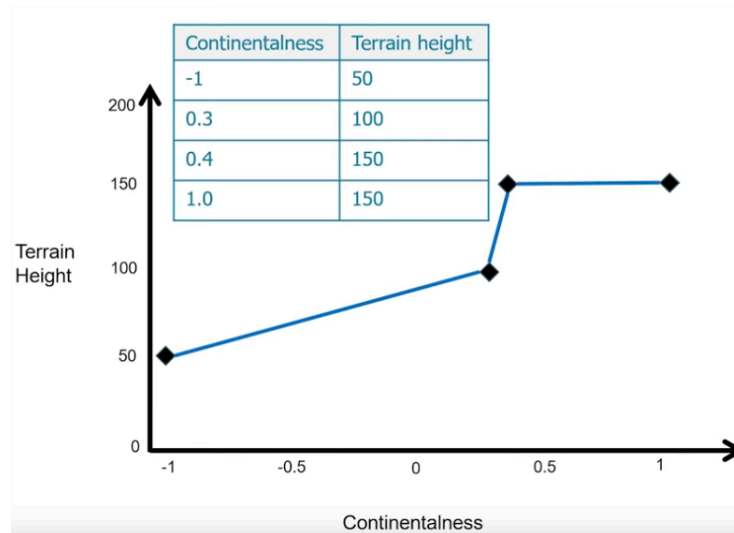
2. Related Work

Key Technologies

We will be using a few predefined technologies to assist us in our development. These technologies include the following:

- Mirror (Networking)
 - Mirror is a high-level networking framework that simplifies the process of adding persistence and multiplayer to our virtual world. Mirror is built on top of Unity's low-level networking API, and provides an easy-to-use solution for synchronizing game state across multiple clients and a server. Using this open source framework, creating a seamlessly networked gameplay experience will be much faster and easier.
 - The limitations of using this high-level framework are that, because it is high-level, we will have to follow the structure set by it, meaning that we will have less flexibility in the networking process. Even with these limitations, Mirror is a great choice here because of the simplicity that it provides for our project.
- *Mirror* Command, TargetRPC, ClientRPC patterns
 - These patterns are all immensely useful for communicating between server and client. The Mirror Command pattern allows running server code directly from the client (from objects with client authority, usually meaning players), whereas target and client RPC allow for running client code from the server. Target is a focused call, whereas client is a call that occurs on all connected clients.
- Shader Graph
 - Unity's Shader Graph is a Unity technology that allows developers to create shaders visually, using a node-based graph interface. By utilizing this technology, we are able to apply rgb color codes to individual voxels through the material applied by the mesh, bypassing the need for the voxels to be individual game objects with materials attached to them. This allows our voxels to look like individual objects without actually defining them as such.
- Greedy Meshing
 - Greedy meshing is a technique used to reduce the total faces of a square-based surface by merging voxel faces if they are exactly the same. It works by scanning all slices of a chunk and greedily growing quad faces where possible repeatedly, then adding those faces to the mesh.
- Spline Graphs/charts

- Spline graphs are a specific way to alter raw noise map values to specific ranges we define. Here's an image to illustrate the concept:



- If our raw continentalness noise generates a value between 0.3 and 0.4, we interpolate between the terrain heights for the two spline points on either side of the generated noise. We use spline graphs like these to customize each of our desired noise types which will interact with each other to add a layer of controllable complexity in the final output.
- It's a fancy way to filter a raw noise to our liking. A spline graph in our game can be as simple or as complex as needed to change the final output value.
- Multi-Octave Perlin Noise
 - Perlin noise for world generation on its own is good, but not good enough. Perlin noise on its own has obvious directional artifacts, and they look bad. by adding together multiple values of perlin noise at different scales, we can diminish this effect. The proper term for this is "octaves" and each successive octave has a higher frequency and a smaller amplitude.
- Unity Job System/Burst compiler
 - The Unity Job System is Unity's solution to multithreaded computing, allowing for the delegation of longer tasks away from the main thread in order to preserve performance. It bypasses the problem of mutexes & shared resources by supplying jobs with everything they will need for the duration of their task. The burst compiler is a special compiler for multithreaded tasks with this exact condition in mind, allowing for the generation of extremely efficient job code. We currently use jobs for the purpose of terrain generation & chunk mesh generation.
- Builder Pattern
 - Allows for easy object construction and ensures that those objects are constructed correctly.
- -

Related Class Projects

Here are some projects our project is related to:

- Procedural Terrain Project (Winter 2021) - By Brennan Drew, Paul Houser, and Carter Schmidt

- This project uses the marching cubes algorithm to generate terrain. They use meshes for terrain just like us, but they don't use voxels. We looked at the marching cubes algorithm and decided it wasn't for us.
- Villineage (Spring 2024) - By Group 7
 - This project utilizes mesh generation with Unity's terrain tools and perlin noise sampling.

3. Approach

Requirements

Engine / World Generation

- Chunks use the greedy meshing algorithm^[0] to merge equal voxel faces, saving memory. [Parker]
- The engine delegates chunk generation to separate threads, so as to not halt the main thread during play. [Parker]
- The engine generates a distinct world for each seed. [Parker]
 - Using the same seed results in the same world being generated.
 - Differences in other world features do not change the actual generation of the world. Voxel resolution only changes the base level of detail.
- The engine should be performant enough to render at chunks in a radius of 32 chunks in each direction where chunks are 32x32x32 voxels³/chunk at a reasonable framerate (≥ 60 fps), with no noticeable stutters on CF 420 lab computers. [Parker]
- Terrain generated is mostly realistic. For example, no large floating islands, no perfectly geometric features, and no sharp biome edges. [Saunders, Parker]
- The engine generates separate biomes, where each biome has unique generation and geological features. [Saunders, Parker]
- Biomes: [Saunders, Parker]
 - Mountains/cliffs
 - Plains
 - Deserts
 - Rivers
 - Lakes/oceans
 - Mesas/canyons
- Climate features overlap with biomes: [Saunders, Parker]
 - Neutral
 - Humid/dry
 - Hot/cold
 - Rainy/clear
- Biomes generate plant/foilage types depending on local climate. [Saunders]

World Interaction

- A user should be able to remove a voxel from the world [Ian]
- A user should be able to add a voxel to the world [Samuel]
- A destroy voxel tool should allow a user to destroy a voxel in the world that they have selected (ie. not determined by a hard-coded offset value). [Samuel]

- A place voxel tool should allow a user to place a selected voxel type at the selected location in the world. [Samuel]
- A bulk destruction tool should allow a user to destroy an area of voxels in the world that they have selected. [Samuel]
- A bulk placement tool should allow a user to place an area of voxels in the world that they have selected. [Samuel]
- A user should be able to define the boundaries of an area for destruction and placement of voxels (i.e. the world edit tool in MC Edit). [Samuel]
- Voxel placement and destruction should be efficient and should not impact gameplay for reasonably sized edits. [Parker, Samuel]
- When a player moves through the world: [Parker]
 - Chunks that get near enough are rendered
 - Chunks that pass a certain distance threshold are no longer rendered.

GUI Interaction

- A starting screen that allows you to input world generation parameters prior to generating the world. [Saunders]
 - Seed field
 - World height in chunks
 - Voxel resolution (Voxels/world unit) [Parker]
 - A higher voxel resolution means individual voxels are smaller, resulting in a world with a higher level of detail.
 - Chunk size (in amount of voxels)
 - Water level of the world [Ian]
 - A “Generate” button to begin generation.
- The UI indicates how many users are in the world [Ian]
- The seed is shown to the user upon request [Ian]
- Terrain brush modifier sliders can be used to change brush size for world interaction [Samuel]

Persistence / Networking

- Chunk data is synchronized between users [Ian]
 - Voxels broken in a chunk are broken for all users with that chunk loaded. [Ian]
 - Voxels placed in a chunk are placed for all users with that chunk loaded. [Ian]
- Users are able to save a seed to be used to generate their own copy of the world. [Ian]
- Connect to other clients via address, omitting port, in a functional interface. [Ian]
- When a user makes changes in a chunk, unloads the chunk such that no one has it loaded, the changes remain when someone else comes near it and loads it. [Ian]
- Multiple users are able to break and place voxels simultaneously [Ian]
- Terrain changes appear instantaneous to each end user [Ian]
- Option to save/load a world [Stretch]

Timeline

Weeks:

1. (Ends: Apr 26)

- Initial planning of project (All group members)
 - CRC cards
- Creating representation of voxels and chunks in game (Parker, Ian)
- Chunk meshing (Parker)
 - Only one voxel type supported

2. (Ends: May 3)

- Sophisticated Chunk meshing (Parker)
 - Naïve culling
 - Greedy meshing^[0]
 - Many voxel types + colors supported
- Main menu UI rough draft (Saunders)
- Basic world interaction (Sam)
 - Placing/breaking voxel
- Chunk generation, (Parker)
 - Chunk loading/unloading based on proximity

3. (Ends: May 10)

- Optimize chunk mesh update re-rendering
- World interaction (Sam)
- World seeds + Random generation (Parker)
- Water generation system (Ian)

4. (Ends: May 17)

- Continue world interaction (Sam)
- Polished UI (Saunders)
- Voxel resolution scaling (Parker)
- Begin networking implementation (Ian)

5. (Ends: May 24)

- Work on voxel interaction-related meshing issues across chunks. (Sam)
- Give different terrain features to biomes (Parker, Saunders)
- Implement large-scale climate patterns (Parker, Saunders)
- Continue networking implementation, code review and design improvements (Ian)

6. (Ends: May 31)

- Work on more terrain modification tools. (Sam)
- Continue implementation of terrain features, climate patterns (Parker, Saunders)
- Begin:
 - Persistent worlds (Ian)
 - Automatic profiling script (Parker)
- Robust networking implementation (Ian)

7. (Ends: June 7th)

- Finalize implementation from week 6. (All)

- Use case and general testing; this will include reviewing every individual use-case and ensuring across multiple users that each requirement is being met. (All)

8. (Ends: June 11th)

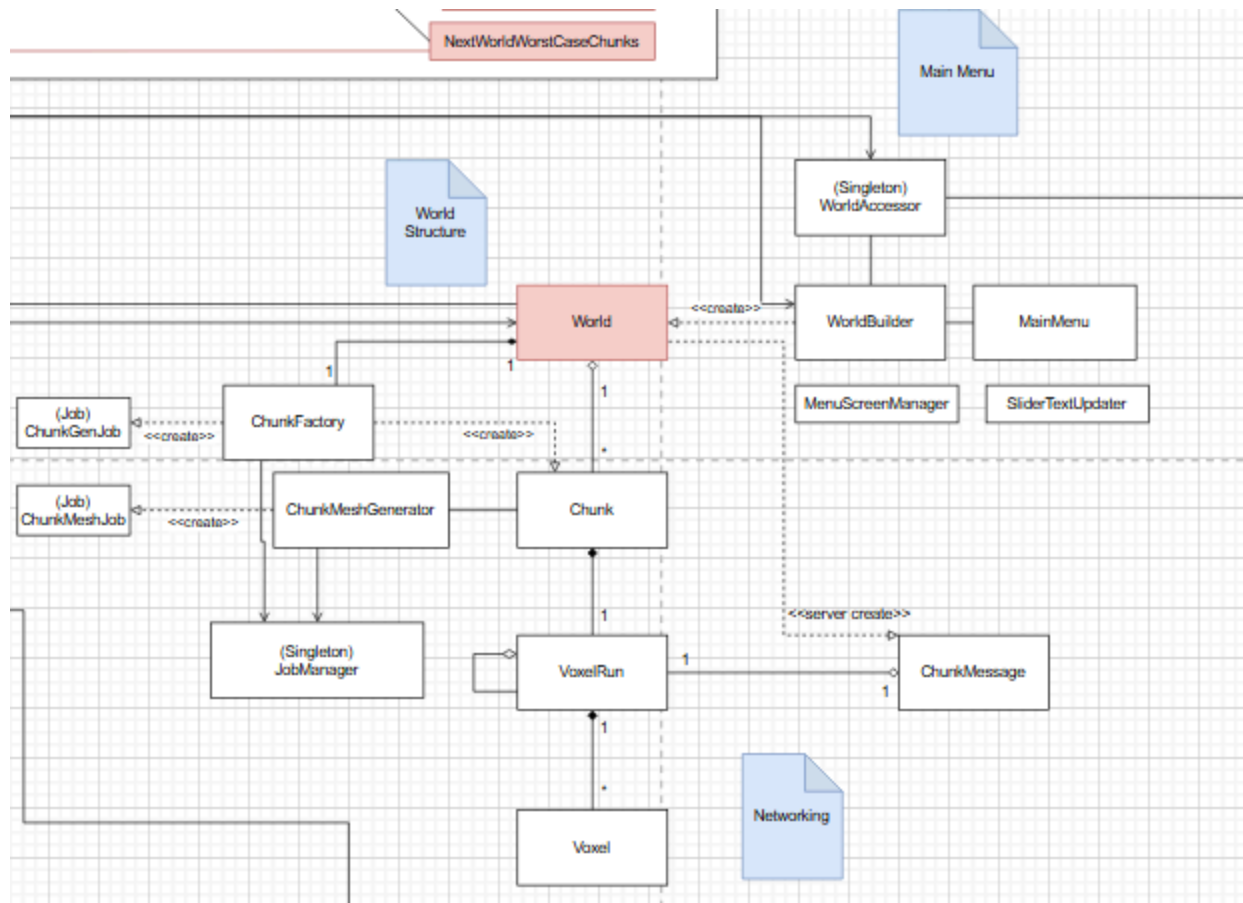
- Finish writeup (All / Parker (for graduate sections))

Architecture or Design Space

Overview

UML DIAGRAM LINK:

https://drive.google.com/file/d/1XMJtp-kpIA5F_RJUBQyu0IVOMI5i2Hj7/view?usp=drive_link



World Generation / Jobs

The world generation system is rather complex (and integrated thoroughly with networking), but here is a general rundown of how worlds work.

Worlds contain chunks at specific locations. Chunks are basically data structures with scripts and are also GameObjects because they need to render their own mesh.. They contain their own voxel data in the form of our own custom sparse voxel data structure, VoxelRun, which stores voxel types and for how many voxels that exact voxel runs on before pointing to another VoxelRun or null when the voxel type changes (if at all). WorldAccessor contains all the worlds and can be used to add or remove a world to this central registry. When a new world needs to be made, WorldBuilder is used.

VoxelRun is our custom sparse data structure. It's essentially a way to script the concept of run-length encoding^[3]. It's implemented as a linked list, where each node has a next field for the next different type of voxel and for how long it runs on.

When the player has its world set, it tells its new world where they are located, in chunk coordinates. The world, given this information, decides which chunks to load for the player based on their set render distance. Worlds do lots of bookkeeping to determine which chunks would need to be loaded into the world next. It keeps currently loaded chunks, unloaded chunks that neighbor loaded chunks, and it keeps any chunks that are in progress of being loaded by a job or some other asynchronous task. When an unloaded neighboring chunk comes within range of the player's render distance, it is queued to be

loaded. When we process our load queue and need to load a chunk, we tell the world's ChunkFactory to generate a chunk.

We use the Unity job system to parallelize long tasks that would otherwise freeze up the main thread and make our engine unusable. To utilize this in the way that we want, we needed to implement callbacks which weren't supported by Unity jobs by default. This led to the creation of the JobManager singleton, accessible from anywhere whenever you need to use a job and need a callback. This is so you can request a job be done and forget about it. Classes that need to use our custom job manager solution define their own data struct that will be passed to the job manager, and passed back to the object which scheduled the job so it can identify what the original job was called for and finish processing the data. We need the JobManager to be a singleton that is attached to a GameObject so that it can hook into the Unity lifecycle and repeatedly check if jobs are finished.

The ChunkFactory is the class contained by each World that dispatches ChunkGenJobs to generate terrain. ChunkGenJobs generate terrain based on our custom implementation of an algorithm described by someone who worked on Minecraft's newer terrain generation algorithms^[2]. In a nutshell, we define spline points (described in Key Technologies above), and use raw noise to generate values filtered by the spline points. When we finish, we have an array of voxels that gets converted to its memory-efficient representation, VoxelRun.

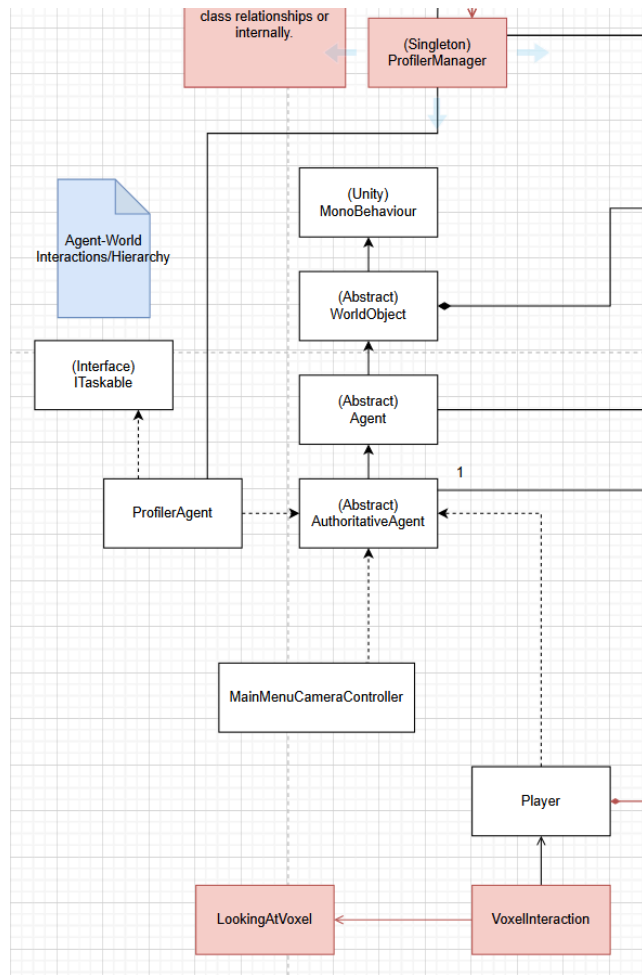
When chunks are finished, the callback returns to the world which dispatched it, finalizing the newly generated chunk, and doing bookkeeping regarding its chunks, as mentioned in the first description of worlds.

In a similar way, requests are made to the host client (server) from clients when they require additional chunks to be loaded. These are then either instantly called back to the client via a target RPC call, or dispatched themselves as jobs on the server which also eventually reach the client through inspection of server-side network identities holding the queue of chunks to be loaded by each player. After being received via target RPC, these chunks are rendered as normal on the clients.

When chunks are finished, they don't have a mesh. They only have terrain. So to compute the mesh information based on its contained voxels, we use the ChunkMeshJob. Data regarding the chunk as well as its immediate neighbor chunks are passed into the job so that the appropriate faces can be culled. The exact greedy meshing algorithm is listed as the first reference at the end [1].

ChunkMessage is a pattern to wrap everything relevant for chunks into a serializable message to send to connected clients. Mirror does the hard work and does some serialization magic.

The main menu when you load in is actually an automatically generated world where a MenuCameraController scrolls to the side, loading the world as it goes. Upon wake, the main menu calls upon WorldBuilder to load a default world for the camera. The main menu camera can actually interact with the menu world as you scroll. Try placing and breaking blocks for fun.

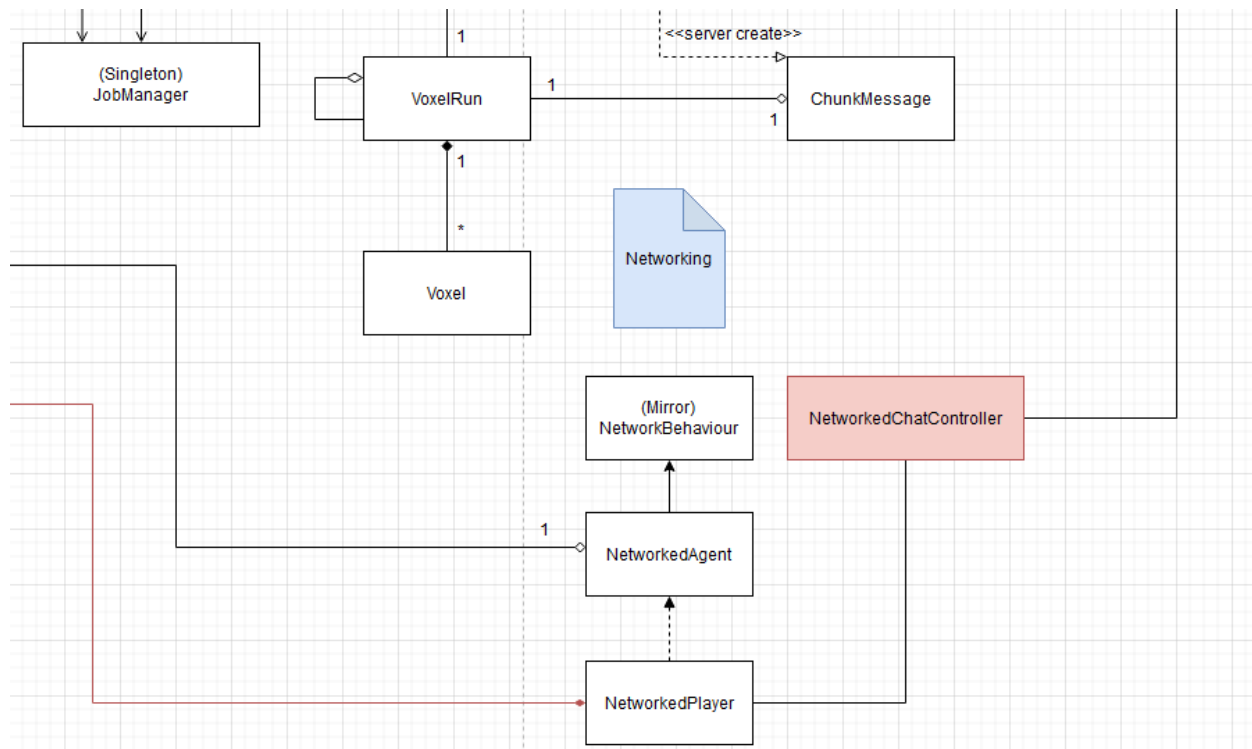


Object Hierarchy

We created our own hierarchy to represent different levels that objects and agents can exist, and what each entails. On a base level, we have WorldObjects which is any object that exists within a particular world within our engine. All WorldObjects are Monobehaviors because they must exist.

Agents are anything that can interact with the world. To be more specific, Agents are capable of moving through the world as well as breaking and placing blocks.

AuthoritativeAgents are the same but more important. AuthoritativeAgents have the added capability of making the world load chunks around them. Whenever any authoritative agent's chunk coordinates change, its world is notified so that it knows to load chunks or unload chunks around that coordinate. ProfilerAgents, MainMenuCameraControllers, and Players are AuthoritativeAgents.



Networking

Pictured is a diagram describing client-network relations. Generally, server-authority is prioritized; when a client wants to interact with the server, those interactions are fed into the host's world after being verified; depending on the action, however, the client will see that rendered first on their end. Some rendering occurs in the client and is checked ad-hoc for fluidity; in these cases, the server verifies legality after. In most cases, clients communicate their intentions and the server responds by eventual callback. The default behavior for Mirror is to give client-authority to players, but this can be mitigated via restrictions. Single-player utilizes a similar, "internal" server. Challenges are largely related to connecting chunk and world infrastructure to this system, as well as communicating and synchronizing rendering. The NetworkedChatController allows for numerous functions, including:

- Displaying Command Options
- Retrieving the World Seed
- Teleporting to Other Players
- List Other Players
- Change Player Name

These capabilities (and other ones such as networked voxel interactions) rely on the Command, TargetRPC, and ClientRPC patterns. These patterns, as explored earlier, are built into Mirror and allow for Players to run code remotely on the server (the Command pattern), servers to run code remotely on a target client (TargetRPC pattern), and servers to run code on all connected clients (ClientRPC pattern). The organization and structure of these communicative elements has some depth and requires some consideration so as to avoid attempting to run server code on the client, etc. Players, when requesting chunks, also submit a Cmd to the server and the server evaluates all clients' needs when loading chunks in. This all occurs at very fast speeds (requests are small and callbacks asynchronous) due to the networking system's implementation *mirroring* the existing ChunkFactory system. Chunks are wrapped

into their component parts (NOT as meshes, but as VoxelRuns, Vector3Ints representing coordinates, and the world accessor string) as ChunkMessages and sent to clients. NetworkAgents are merely extensions of NetworkBehaviours that handle tasks agents would pass off; in other words, they are linked to regular Agents and represent networked actions for those agents; anything that any Agent can do (like loading terrain) may need to be networked at some point, so that networking occurs through that connection so long as it is active. In the profiler, for example, local actions are valuable; in this case, when the server is not running, actions bypass their networked counterparts.

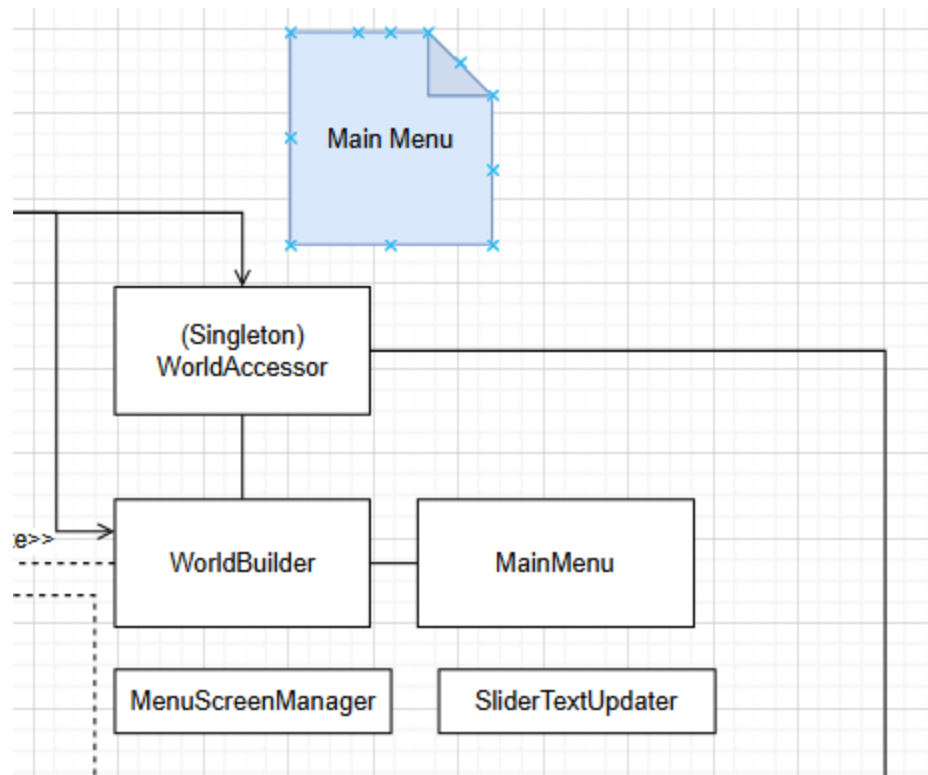
While it is possible to see the NetworkAgent and NetworkPlayer as tightly coupled with their localized counterparts (Agent, Player), and indeed they are, this is a natural result of the object hierarchy. Because most controllers are obligate MonoBehaviours and NetworkBehaviours are necessary for communication/network capabilities but cannot exist without NetworkIdentity, these components must be split but capable of interacting with each other on a deep level. In order to escape this paradigm, it is possible to construct a custom Model-Controller system that manages these connected, disparate parts without introducing overly much coupling.

UI

The first scene that is loaded when our project is started up is the Main Menu scene. This scene contains a canvas with 4 different screens: the initial menu screen, the create world screen, the join world screen, and the options screen. Handling swapping between these screens happens in the MenuScreenManager script. The MainMenu script is attached to the primary canvas. Within the main menu script lives the publicly implemented sliders and input fields that are used

in the create world screen to customize the world generation parameters: seed, world height, world resolution, chunk size, chunk height, and water height. Where sliders are being used, the values are updated on screen by the SliderTextUpdater script. There is also the ip address field in the join world screen that allows players to join another player's created world. The start method of the main menu script tells the world builder to initialize the menu world which generates a world with the default parameters.

When a player chooses the create world option, they can adjust the world generation parameters to affect various aspects of the world. Once they've done that and pressed generate world, the GenerateWorld method in the main menu script is called. This script saves the world generation



parameters to a WorldParameters struct. Then it loads the “Za Warudo” scene in single mode (unloading the main menu scene), and subscribes the PlayUsingPlayerAgent method to the event. PlayUsingPlayerAgent calls StartHost on the NetworkManager (worlds are always networked, except profiler worlds). Then, a new WorldBuilder is created, building the world with the appropriate world parameters. Then, the menu world is unloaded using the WorldAccessor and PlayUsingPlayerAgent is unsubscribed from the sceneLoaded event.

To allow players to join a world, the JoinWorld method is called when a player presses the join world button after entering an ip address of a host. The first thing this method does is to load the “Za Warudo” scene in single mode. Then it sets the networkAddress using the NetworkManager to whatever the user input in the ip address field. Finally, the NetworkManager.StartClient method is called to start a client session. This where a connection is attempted and if failed will return the client to the main menu.

To control the camera that slowly moves to the right in the main menu world, we have a camera that implements the AuthoritativeAgent class. This allows the camera to load new chunks in the menu world. The camera moves to the right with a simple update method.

The in-game pause menu can be activated by pressing the escape key, but it does not actually “pause” the game in the sense that it stops time. It brings up the chat window, which can be used to execute commands as well as converse with other players connected to the world. The commands can be seen by typing /help and include /list: lists the currently connected players, /nick: sets the players name which is displayed above the player model as well as in chat, /teleport [player] which teleports the executing player to the player indicated in the command, and /seed which displays the world seed. Quitting the game is also an option from this menu, carried out by the MainMenu method inside the NetworkedChatController script. This method unloads the current world, stops the host, server, and client, and quits the application.

```

classDiagram
    class WorldTask {
        <<interface>>
    }
    class ProfileGameTask {
        <<singleton>>
        <<itabletaskable>>
    }
    class JourneyScenario
    class WorldGenStressTestScenario
    class MassSingleReplaceScenario
    class MassMassReplaceScenario
    class WorstPossibleChunksScenario
    class LongMoveTask
    class WaitTask
    class SimpleMoveTask
    class SingleReplaceTask
    class MassReplaceTask
    class TeleportTask
    class WaitConditionTask
    class SwitchProfilerAgentWorld
    class SwitchProfilerAgentRenderDist
    class NextWorldWorstCaseChunks
    class ProfilerAgent {
        <<itabletaskable>>
    }
    class MonoBehaviour {
        <<unity>>
    }
    class World
    class WorldStructure

    WorldTask <|.. ProfileGameTask
    WorldTask <|.. JourneyScenario
    WorldTask <|.. WorldGenStressTestScenario
    WorldTask <|.. MassSingleReplaceScenario
    WorldTask <|.. MassMassReplaceScenario
    WorldTask <|.. WorstPossibleChunksScenario
    ProfileGameTask <|.. ProfilerAgent
    ProfileGameTask --> JourneyScenario
    ProfileGameTask --> WorldGenStressTestScenario
    ProfileGameTask --> MassSingleReplaceScenario
    ProfileGameTask --> MassMassReplaceScenario
    ProfileGameTask --> WorstPossibleChunksScenario
    JourneyScenario --> LongMoveTask
    JourneyScenario --> WaitTask
    JourneyScenario --> SimpleMoveTask
    JourneyScenario --> SingleReplaceTask
    JourneyScenario --> MassReplaceTask
    JourneyScenario --> TeleportTask
    JourneyScenario --> WaitConditionTask
    WorldGenStressTestScenario --> LongMoveTask
    WorldGenStressTestScenario --> WaitTask
    WorldGenStressTestScenario --> SimpleMoveTask
    WorldGenStressTestScenario --> SingleReplaceTask
    WorldGenStressTestScenario --> MassReplaceTask
    WorldGenStressTestScenario --> TeleportTask
    WorldGenStressTestScenario --> WaitConditionTask
    MassSingleReplaceScenario --> LongMoveTask
    MassSingleReplaceScenario --> WaitTask
    MassSingleReplaceScenario --> SimpleMoveTask
    MassSingleReplaceScenario --> SingleReplaceTask
    MassSingleReplaceScenario --> MassReplaceTask
    MassSingleReplaceScenario --> TeleportTask
    MassSingleReplaceScenario --> WaitConditionTask
    MassMassReplaceScenario --> LongMoveTask
    MassMassReplaceScenario --> WaitTask
    MassMassReplaceScenario --> SimpleMoveTask
    MassMassReplaceScenario --> SingleReplaceTask
    MassMassReplaceScenario --> MassReplaceTask
    MassMassReplaceScenario --> TeleportTask
    MassMassReplaceScenario --> WaitConditionTask
    WorstPossibleChunksScenario --> LongMoveTask
    WorstPossibleChunksScenario --> WaitTask
    WorstPossibleChunksScenario --> SimpleMoveTask
    WorstPossibleChunksScenario --> SingleReplaceTask
    WorstPossibleChunksScenario --> MassReplaceTask
    WorstPossibleChunksScenario --> TeleportTask
    WorstPossibleChunksScenario --> WaitConditionTask
    ProfileGameTask --> SwitchProfilerAgentWorld
    ProfileGameTask --> SwitchProfilerAgentRenderDist
    ProfileGameTask --> NextWorldWorstCaseChunks
    ProfilerAgent ..> MonoBehaviour
    MonoBehaviour --> World
    WorldStructure --> World
    World --> World : <<create>>
  
```

Red indicates a class or relationship that we think has some bad coupling associated with it somewhere, be it other class relationships or internally.

World Structure

World

We add the `ITaskable` interface, only to be added to an Agent. It forces classes to implement a `Perform()` method, so that the agent can progress on its current task. Agents that implement `ITaskable` each decide how to manage their tasks on their own.

The ProfileGameTask is the brain of the profiling tasks. The ProfilerAgent is simply assigned a ProfileGameTask and it works automatically. When a ProfileGameTask is created, it generates all possible combinations of world parameters with each scenario and assigns them to the player.

15

finished, the ProfilerManager writes to a file the data collected for each scenario, the scenario name, as well as the world parameters used. It collects the max and average CPU frame timings, and the max GPU frame timings. It also captures the memory used at the end of the scenario.

Integration Plan - Documentation to Help Future Developers Expand on Our Current Codebase

To ensure seamless integration and expansion, we are focusing on creating a modular and flexible architecture for our voxel-based world generation engine. Thankfully, our project lends itself well to extensibility due to its blank canvas-like nature. Our goal is to optimize the job of the terrain generation engine to a good enough degree that others can focus solely on adding even more life to the worlds it generates. Our project assets are organized as follows:

Within the assets folder, there are 3 folders that can be added to in order to expand on our project:

- Art - This folder contains the custom images we used for the buttons, the fonts for our project, as well as the material files.
- Scenes - This folder contains our two unity scenes, Main Menu and Za Warudo. More can be added here for customization.
- Scripts - This is where the bulk of our work lies. Within scripts lives the majority of our c sharp code that controls the logic of our world generator. The organization of scripts is as follows:
 - Agent - Contains code defining agents (players, profilers, main menu camera)
 - Player - Code defining the player class and other scripts relating to player actions
 - Autoprofiler - Contains the code that defines the agent that can perform tests for performance evaluation in a generated world
 - ProfilingScenarios - various scenarios for the profiler to perform
 - Results - output from the profiler to be analyzed for optimizations
 - Tasks - predefined tasks that can be added to new/existing scenarios for the profiler to carry out
 - Jobs - Contains the two main jobs we have defined (generating chunks and their mesh) as well as the JobManager script
 - UI - Contains all the main menu code (main menu world, world generation parameter handling, etc.)
 - Voxels - Contains the definition for voxel including the different types of voxels
 - World - Contains the definitions for the various world related scripts
 - Chunk - definition for a chunk as well as the ChunkFactory and ChunkMeshGenerator scripts
 - WorldObject - Script to handle setting/getting the current world. Used to handle swapping a players world from main menu world to newly created world to multiplayer world

Keybinds:

Key	Action
Left Click	Destroy Voxel
Right Click	Place Voxel
Shift + Left Click	Destroy Group of Voxels

Shift + Right Click	Place Group of Voxels
Control + Shift + Left Click	Select Corners for Region to Destroy
Control + Shift + Right Click	Select Corner for Region to Place
T	Teleport to Cursor Location
1	Set Placeable Voxel Type to Grass
2	Set Placeable Voxel Type to Dirt
3	Set Placeable Voxel Type to Stone
4	Set Placeable Voxel Type to Water
. (period)	Increase Mass Voxel Break/Place Size
, (comma)	Decrease Mass Voxel Break/Place Size
/ (slash)	Open/Close Chat Window with Prependded Slash
Esc (escape)	Open/Close Chat Window

Examples of How to Expand on Our Work:

- For examples of how to expand on our work, please refer to the Future Work section of this document.

4. Evaluation

Use Case 1

Title - Generate World

Description - The user starts our world generation engine and is greeted with the initial UI. There will be some parameters they can change that would affect how the terrain is generated. These can be altered or left default and then the user can begin the world generation.

Preconditions - The user must have correctly installed and compiled our application and have sufficient storage for the world they create to be stored on their system.

Basic Flow -

1. User opens our world generation engine application
2. User chooses “create world” button in initial UI
3. World is created
4. User is spawned into the world

Alternative Flow 1 -

1. User opens our world generation engine application
2. User adjusts a world generation parameter

3. User chooses “create world” button in UI
4. World is created with adjusted parameter
5. User is spawned into the world

Postconditions - The user now has a persistent, procedurally generated world to interact with and explore. The user can exit the world and come back to it at a later point.

Use Case 2

Title - Traverse World

Description - The user is inside of a world generated by our engine. They can now control their avatar and explore the terrain.

Preconditions - The world was successfully generated.

Basic Flow -

1. User sees the generated world terrain around their player avatar.
2. User uses the arrow keys/WASD to move their avatar around the terrain.
3. User stops to observe a notable feature of the terrain, staring wistfully into the distance.
4. User continues exploring the world, observing different biomes and natural features.

Postconditions - The user has explored the world they created and observed the natural features and biomes.

Use Case 3

Title - Interact with World

Description - The user is inside of a world generated by our engine and will use the world modification tools to break and place voxels.

Preconditions - The world was successfully generated.

Basic Flow -

1. User makes a selection of voxels to remove from the world.
2. User uses the UI to delete said voxel.

Alternative Flow 1 -

1. User chooses a voxel type to place into the world.
2. User selects the location they want to place the voxel.
3. User uses the UI to place the voxel into the world.

Postconditions - The world has been successfully modified. The modifications made by the player are saved to the world and will be persistent the next time the user loads up their world.

Methodology

This project was tested and evaluated based on performance. That is, memory used and frame timings. We have automated scripts that simulate player actions and measure performance during separate use case scenarios.

This manifests in different ways, such as automatically moving or teleporting the player to measure terrain generation impact with various render distances, or measuring frame times to ensure no spikes. We will also consider the simulation of breaking and placing many voxels to ensure minimal lag, the

maximum memory usage for different situations, and worst-case chunks where voxels alternate between solid and air which maximizes mesh size.

We evaluate based on these parameters. As this is a voxel engine, performance is above all else.

There are numerous ways to collect data on the performance of a Unity app. There's the performance profiler which collects all kinds of data on all metrics of the app. This adds a lot of overhead however, as it tracks many unnecessary things that you are probably not interested in. You can filter what you want by placing performance markers in the code which measure specific blocks of code, but this still incurs the overhead of the profiler since that is what it's based on.

I settled on using the [Unity frame timing manager](#) which has a tiny performance overhead compared to the classic profiler. This is used for CPU and GPU frame timings, while for memory, we captured snapshots of memory usage using Unity profiler methods which return the total allocated memory for the engine as it's running. We don't incur the whole cost of the profiler here because we're only using static methods to capture the current memory usage; The entire profiler isn't enabled.

To actually do some profiling, things need to happen in the world, and they need to happen consistently. I decided to rework our current object hierarchy to allow for Agents to exist, where agents are any entity that can interact with their current world. So both a player and my performance profiler agent are agents. My agent could interact with the world, but it didn't know what to do. So I also had to create a way for it to know what to do next. In comes the task system I created. The task system is akin to a behavior tree for AI, but without selectors.

When you want an agent to be able to perform some action automatically, you give an agent the ITaskable interface. This allows tasks to be assigned to the agent. Tasks can define their own logic, or be composed of other tasks, or both. When an agent tries to perform its current task, its current task logic dictates what things the agent does per attempt of the task's perform function, whether it be moving, breaking voxels, etc.

I define specific scenarios as tasks designed to model different use cases or stress different pieces of our engine.

Here are the scenarios measured and how they're defined, where each is measured with all possible world parameters:

Journey scenario (Designed to stress test the system which decides which new chunks to load and unload):

1. Spawn in the world with seed 200
2. Move 1000 blocks ahead
3. Finish

World gen scenario (Designed to stress test world generation):

1. Spawn in the world with seed 300
2. Change render distance to 10
3. Repeat 20 times:
 - a. Wait until all chunks in range are finished loading
 - b. Teleport 10,000 world units further away
4. Finish

Mass single replace scenario (Stress test single replace):

1. Spawn in the world with seed 400
2. Repeat 2,000 times:
 - a. Randomly place a stone voxel in a range in front of it.
3. Repeat 2,000 times:
 - a. Randomly break a voxel in the same range in front of it.
4. Finish

Mass Mass replace scenario (Stress test mass replace):

1. Spawn in the world with seed 500
2. Repeat 1,000 times:
 - a. Pick 2 random points ahead of the agent and do a mass place between them.
 - b. Pick 2 random points ahead of the agent and do a mass break between them.
3. Finish

Worst chunks scenario (Stress test memory usage of worst chunks):

1. Spawn in a world with worst chunk generation enabled
2. Wait until all chunks in range are finished loading.
3. Finish

New tasks can be created to suit new use cases as needed.

For each scenario, the following stats are measured:

- Total scenario duration (seconds)
- Max CPU frame time (ms)
- Average CPU frame time (ms)
- Max GPU frame time (ms)
- Memory used (MB)

From this data, we can determine weak points of our engine and optimize accordingly.

Though we have an auto profiler, it's still extremely important to use the built in Unity performance profiler and memory profiler. Our built-in profiler is good to measure performance at the very end of the engine creation process, performing high-level tasks rather than benchmarking code from the inside.

Results and Analysis

The auto profiler generates lots of data, some of which is not useful for certain scenarios. In our case, the auto profiler was beneficial because it indicated some optimized world parameters that we should be using by default.

There's lots of data and I didn't feel there was enough time to make something to automatically process the data, so for now you must parse it manually.

Data was generated on a computer with the following GPU and CPU:

CPU: Ryzen 5950X

GPU: RTX 3080 Ti

We didn't know if subdividing the world into vertical chunks was better or worse than having one singular tall chunk that spans the bottom of the world to the top. We decided to consult our data to check:

The relevant scenarios here are the world generation stress testing and the journey scenario.

For chunk sizes of 16, single tall chunks are approximately equal in performance than vertical chunks.

For chunk sizes of 32:

- Tall chunks:
 - Max CPU frame time: 61ms
- Vertically-stacked chunks:
 - Max CPU frame time: 90ms

For chunk sizes of 64:

- Tall chunks:
 - Max CPU frame time: 224ms
 - Avg. CPU frame time: 45ms
- Vertically-stacked chunks:
 - Max CPU frame time: 238ms
 - Avg. CPU frame time: 52ms

For chunk sizes of 128:

- Tall chunks:
 - Max CPU frame time: 900ms
 - Avg. CPU frame time: 240ms
- Vertically-stacked chunks:
 - Max CPU frame time: 792ms
 - Avg. CPU frame time: 201ms

For the journey scenarios, the tall chunks generally did slightly worse.

For mass single & mass mass replace scenarios, tall chunks did far worse. Here's mass mass replace for example:

For chunk sizes of 16:

- Tall chunks:
 - Max CPU frame time: 286ms
 - Avg. CPU frame time: 8ms
- Vertically-stacked chunks:
 - Max CPU frame time: 80ms
 - Avg. CPU frame time: 7ms

For chunk sizes of 32:

- Tall chunks:
 - Max CPU frame time: 1245ms
 - Avg. CPU frame time: 14ms
- Vertically-stacked chunks:
 - Max CPU frame time: 346ms
 - Avg. CPU frame time: 8ms

For chunk sizes of 64:

- Tall chunks:
 - Max CPU frame time: 7363ms
 - Avg. CPU frame time: 33ms

- Vertically-stacked chunks:
 - Max CPU frame time: 891ms
 - Avg. CPU frame time: 12ms

And it gets worse for chunk sizes of 128.

With this data in hand, this allows us to advise our decisions on default world parameters to enhance the user's experience with our engine, since they won't know anything about how the world parameters affect anything. This data tells us that there are certain situations which would be well-suited for taller chunks like in the case of specific parameters for world generation, but for other scenarios the tall chunks are not such a good idea. This happens to be the only detail we wanted to explore, but the strength of our auto profiler agent task system is that you can define any custom scenario that you want and measure the performance. Using this in tandem with built-in unity profiling results in an extremely flexible way to pinpoint performance bottlenecks.

It's important to mention that bottlenecks aren't always specific locations. Oftentimes, performance in applications degrades slowly over time. The phrase "death by a thousand cuts" comes to mind, and the solution is not always so straightforward. For small inefficiencies scattered around the project, I would recommend the built-in unity profiler, with deep profiling, or by manually placing profiler markers around in the code surrounding certain important blocks of code to measure the time spent as the project grows.

5. Progress

Milestone 1 Progress

Accomplishments:

- Base chunk & world data representation
- Limited world generation parameters
- Efficient chunk meshing

Group Member Contributions:

Ian:

- Added voxel removal option (digging)
- Reworked some meshing and voxel representation code for readability and extensibility 😊
- Added documentation; consulted and worked on design, OOP, Unity patterns, documentation, some low-level engine / .NET stuff.

Parker:

- Base chunk & world data representation
- Limited world generation parameters
- Efficient chunk meshing

Samuel:

- Camera line of sight based voxel interaction
- Cursor position based voxel interaction
- Voxel placement and destruction
 - Single voxels at a time
 - Bulk group of voxels centered around a point
 - All voxels between two selected points

Saunders:

- Rough draft of Main Menu design
- Can generate a world with modified attributes from Main Menu
- Quit app from Main Menu
- Misc. document contributions

Milestone 2 Progress

Accomplishments:

- Threaded chunk and mesh generation
- Frozen water
- More efficient and larger scale block breaking/placing
- More polished main menu screen
- Beginning networking implementation

We've made a significant amount of progress from milestone 1 to 2. The performance of our engine has greatly improved thanks to Parker's efforts on threading. Ian has added a basic implementation of water as well as the beginnings of networking, plus design/architectural changes. In terms of UI menus and tools, Sam has expanded the existing tools to efficiently destroy large batches of voxels and Saunders has been working towards getting the project to look like a finished product with nice looking menu buttons and backgrounds. The remaining tasks include seeds, biomes, finished networking and world serialization, and final logos/design. With this in mind, we feel we have effectively completed at least 60% of our project.

Group Member Contributions:

Saunders:

- Separated main menu into 3 screens and added custom made buttons based on the designs from [here](#)
- Added generated world that scrolls to the side as menu background
- Various milestone 2 document contributions

Ian:

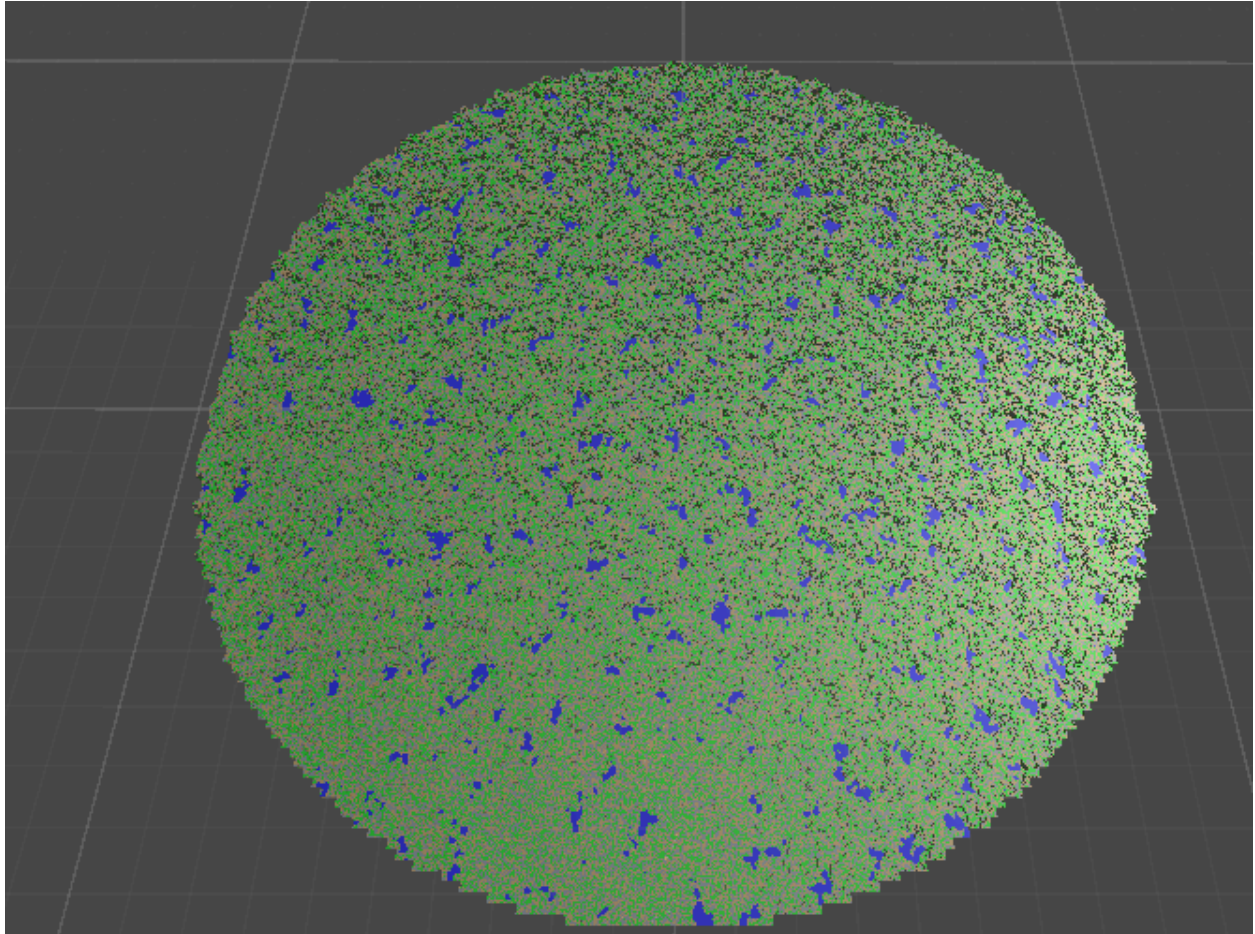
- Refactored code to follow a more logical structure; slashed a significant amount of unnecessary code and improved object relations, naming, and scene structure to be more logical and less error-prone (for example, generating a world if the player exists in an empty space), abstracting WorldController components into the builder pattern, including an accessor.
- Added "water".
- Began network implementation.
- Writing.

Sam:

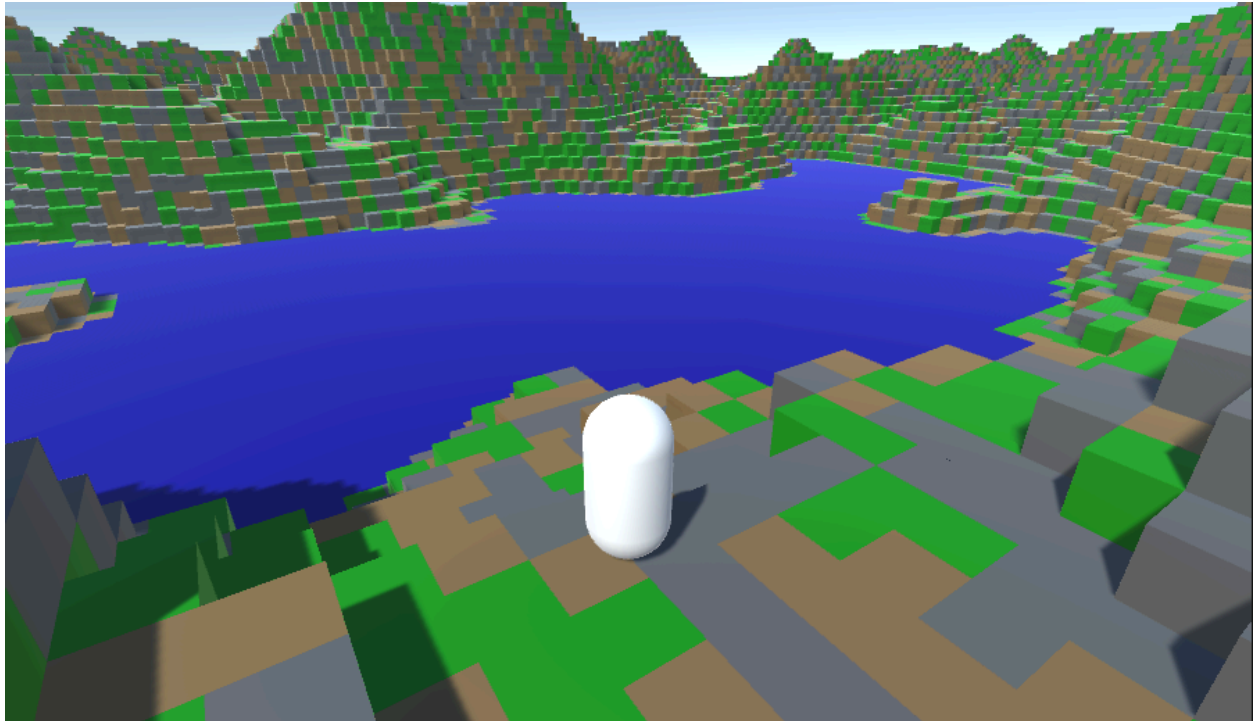
- Implemented voxel destruction and placement in large chunks, centered around a selected voxel.
- Added a terrain edit tool for selecting two points between which all voxels will be changed to the selected type, within reason (i.e. pre-existing terrain cannot be written over until the voxels have been destroyed).
- Increased efficiency of destruction/placement by minimizing the number of neighboring voxels checked.

Parker:

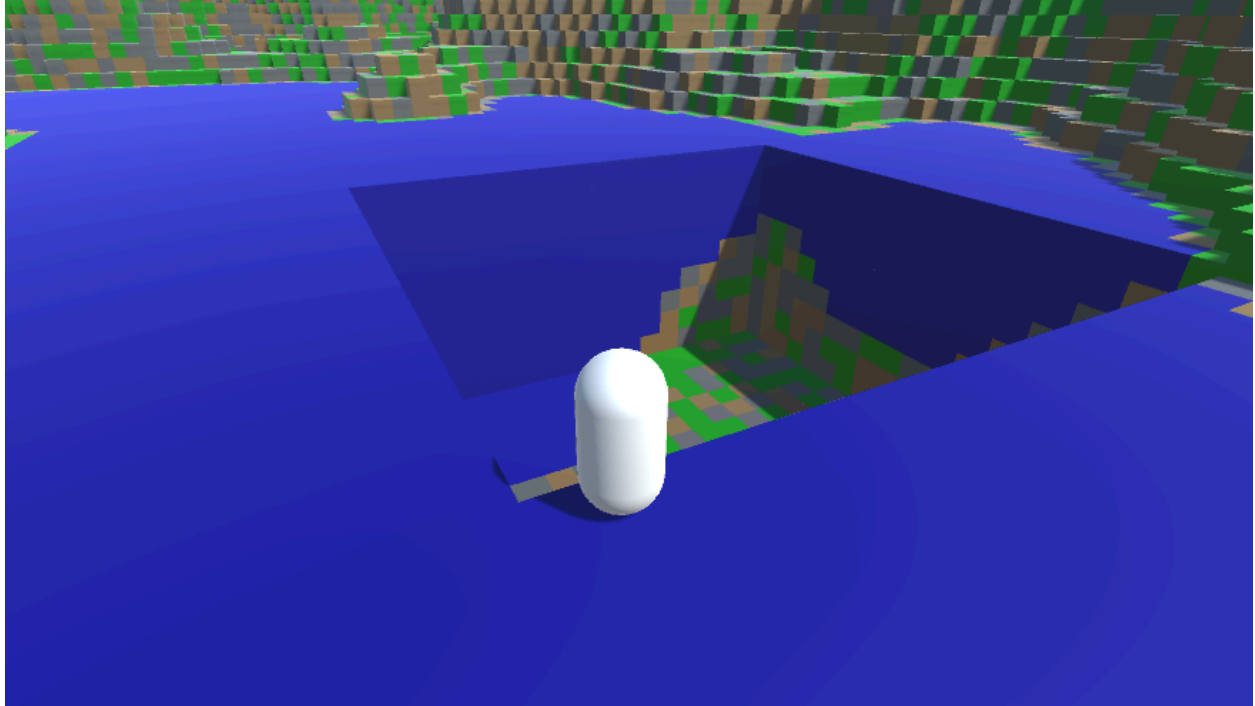
- Implemented threaded chunk generation & threaded chunk mesh generation using the Unity Job System
- Implementation of a simple perlin noise terrain generation algorithm



[MILESTONE 2] Pictured above is terrain in a render radius of 60 chunks with chunks being 16 voxels^3 . Unity consumed 18GB of memory for this. That is a lot (this is bad). At the moment, there is no sparsity in our data. I.e., chunks that have no blocks in them take up as much space as a chunk full of voxels. If we wanted worlds to accommodate tall mountains, then empty chunks would dominate our memory usage.



[MILESTONE 2] The player stares longingly at a body of “water”, knowing it’s solid and they can’t swim in it. This is due to the way meshes currently work in our game; chunks are represented as MeshColliders. The terrain is generated using a simple 3-octave perlin noise generator. It will be replaced, it was only chosen as a toy algorithm. The land being random voxels was chosen to make sure greedy meshing worked.



[MILESTONE 2] In a fit of wrath, the user lashes out at the “water” using the handy Mass Deletion Tool to remove a chunk of the terrain instantly.



[MILESTONE 2] The main menu as it scrolls to the right generating a world as it moves

Final Milestone Progress

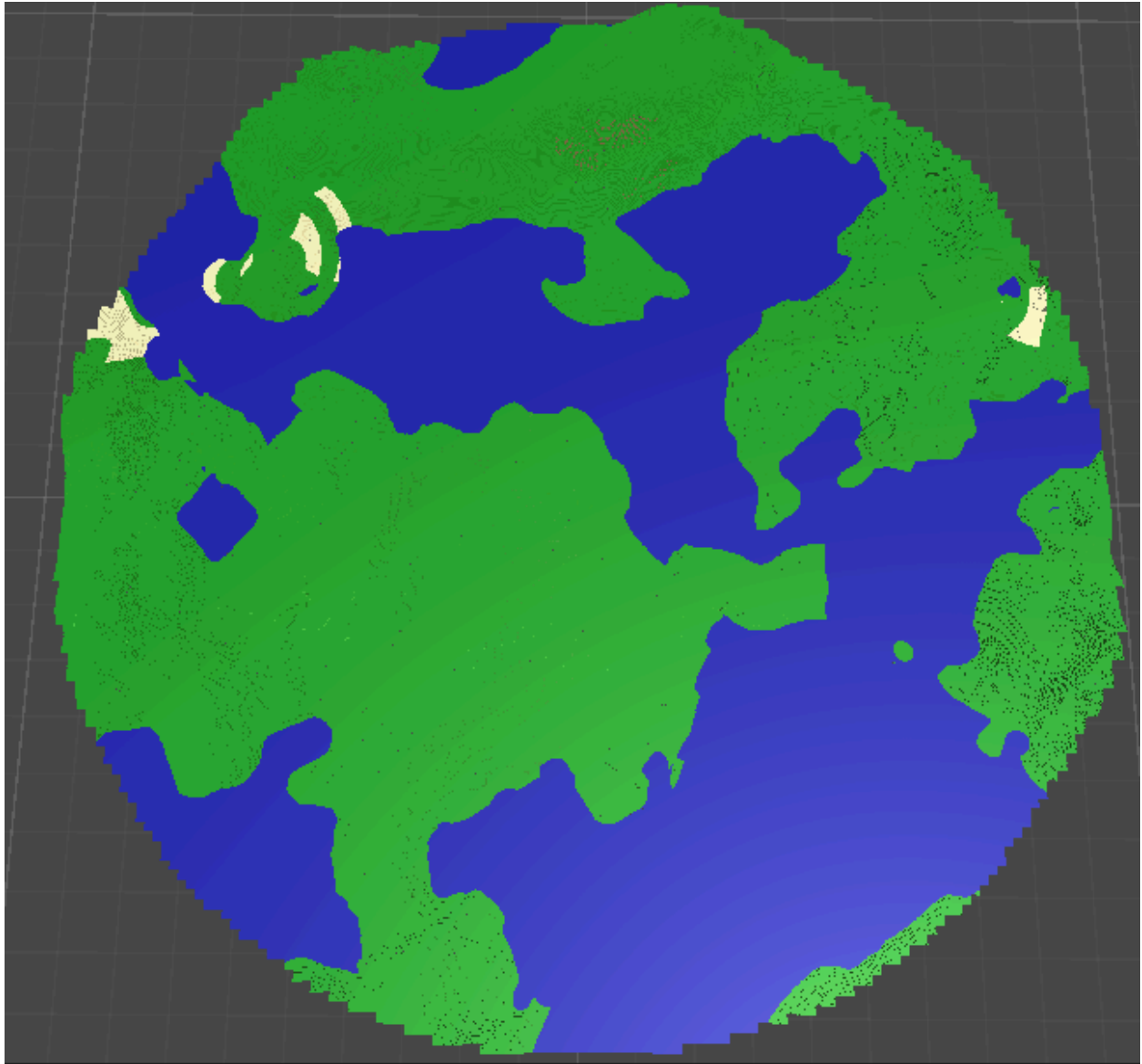
Accomplishments:

- Optimizations all around, both memory usage & run time.
 - Memory efficient chunk voxel storage using a custom linked-list implementation of the run-length-encoding of that chunk's voxels.
 - More efficient dispatching of chunk mesh requests; No more redundant mesh jobs dispatching.
 - More efficient job dispatching; do as little work on the main thread as possible.
 - Fixed various memory leaks
- Created an auto-profiler which directs an agent to perform various game scenarios at different world parameters, measuring frame timings and total memory usage.
 - This called for the creation of a system akin to a behavior tree, though without selector and sequence nodes.
- Overhauled the terrain generation algorithm
 - We use different types of noise which affect each other in different ways to generate more complex terrain:
 - Continentalness: Defines how far inland you are, typically resulting in higher terrain.
 - Erosion: Defines how eroded the terrain is. The more erosion, the flatter the terrain and the closer to sea level.
 - Peaks & Valleys: Defines a constant amount to add to or subtract from the terrain.
 - Biomes are not fully implemented and at the moment require refactoring/overhauling to make work. These noises are for classifying biomes and don't affect terrain:
 - Temperature: Only used in classifying biomes. For example, an area with high erosion noise & high temperature noise would be classified as a desert.
 - Humidity: Only used in classifying biomes. How humid an area is.
 - Each different type of noise has a "spline graph" which applies to the raw noise to shape the noise more to our liking.

In its current state, we would consider this project approximately 100% complete. Certain features have only limited functionality however all the important pieces are finished and usable. That is, networking, terrain generation, chunk meshing, and world interactions. We didn't have the time to finish extra features.

- Parker:
 - Optimizations all around, both memory usage & run time.
 - Overhauled the terrain generation algorithm
 - Overhauled object hierarchy to abstract away agents that can interact with their world
 - Created the auto-profiler & agent task system
- Ian:
 - Assisted with design/redesign of object/task hierarchy
 - Implemented networking in its entirety, including synchronized worlds, persistent (from host memory) chunks, and other features.
 - Implemented chat window with various commands.
- Saunder:
 - Helped add more noise functions and spline point implementation to terrain generation algorithm
 - Designed and created the project logos
 - Testing
- Samuel:

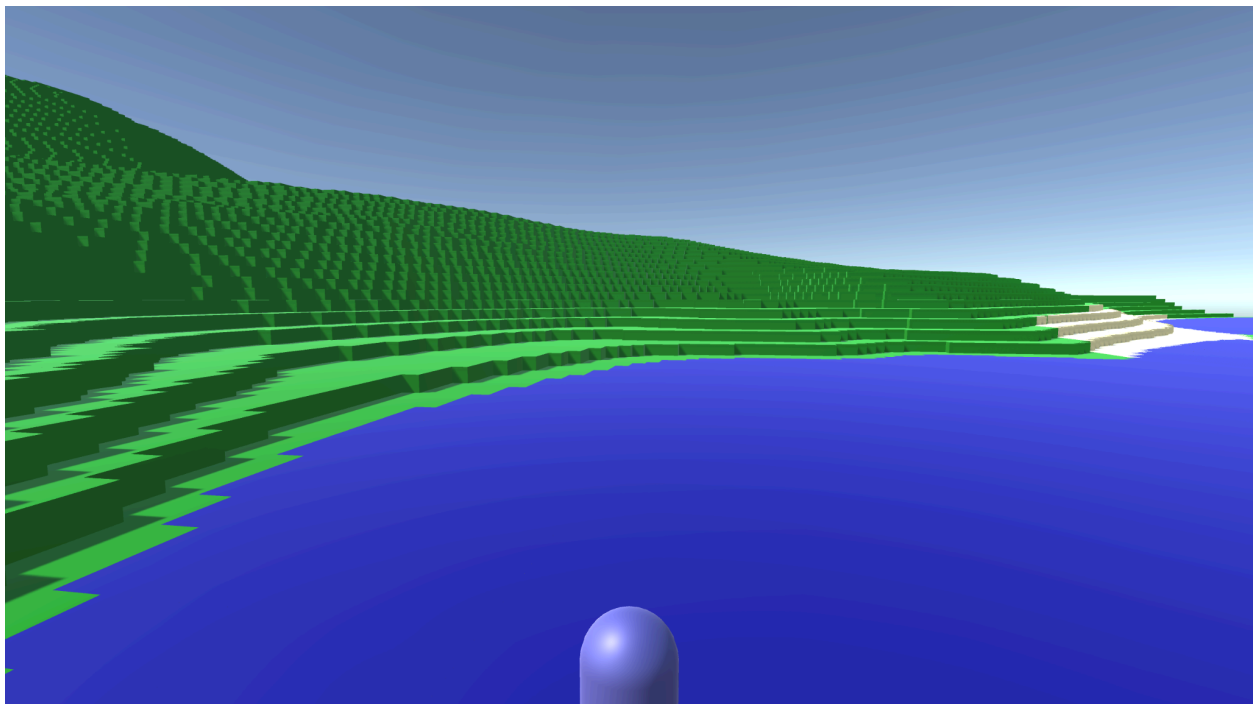
- Cleaned up mass voxel interaction
- Testing



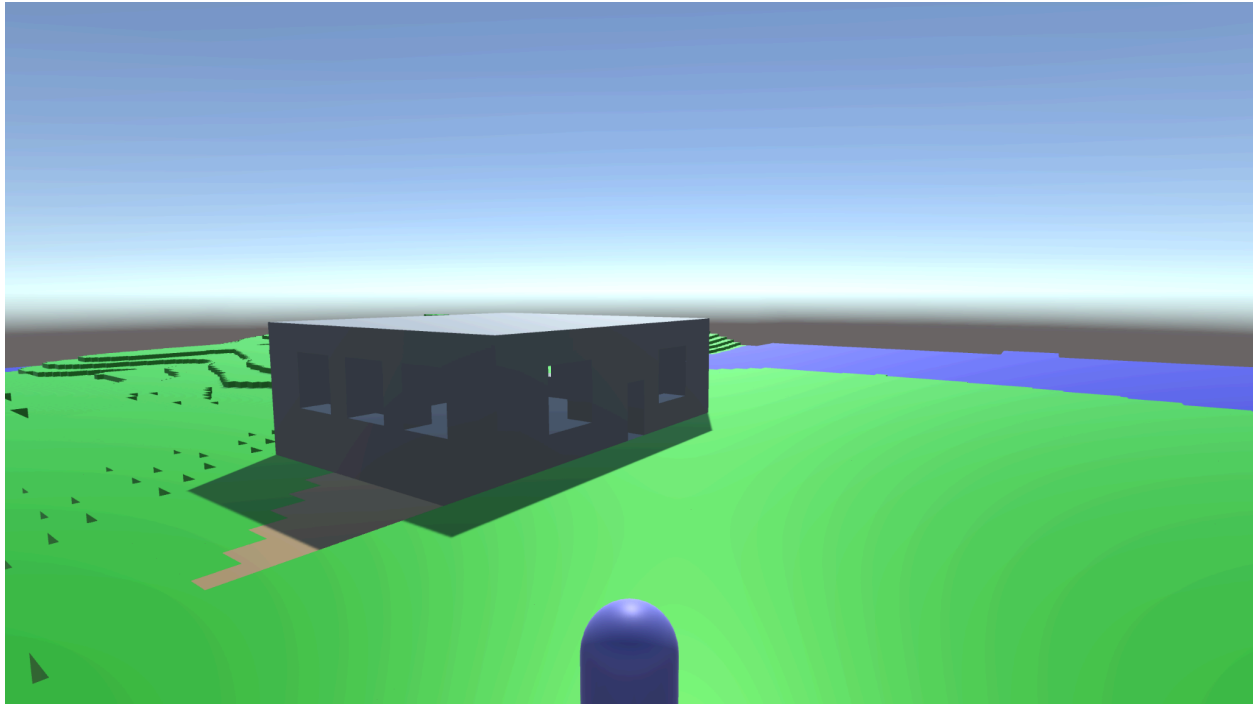
[FINAL MILESTONE] Pictured above is a birds-eye view of terrain in a render radius of 60 chunks with chunks being $16 \times 256 \times 16$ voxels³. Contrasted with this same world size from Milestone 2 (pictured above), this terrain only used up ~650MB and generated far quicker. This is due to the new data structure we use to store voxels as well as various other optimizations all over. You can see some “desert” which generated up in the top left corner of the map. Biome generation in its current state doesn’t work well.



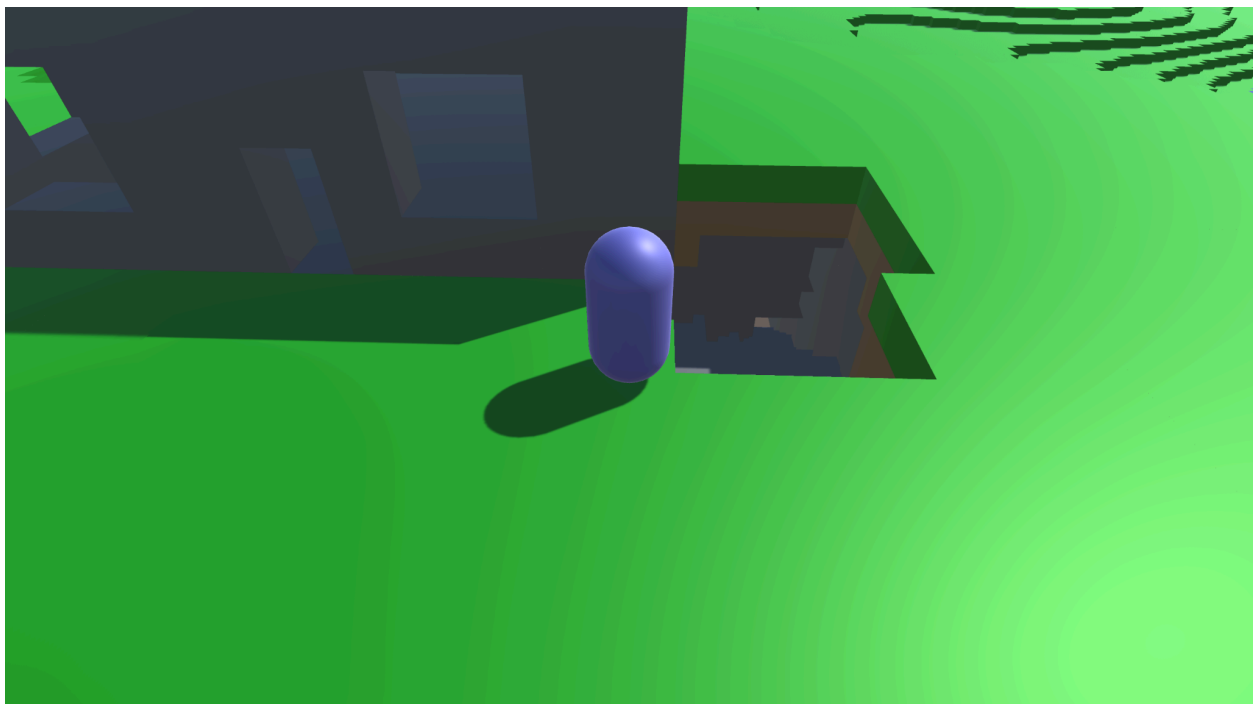
[FINAL MILESTONE] The current look of the main menu screen features a new logo as well as a new world in the background showcasing the updated world generation algorithm. You can edit the terrain you see in the title screen! Try breaking or placing blocks!



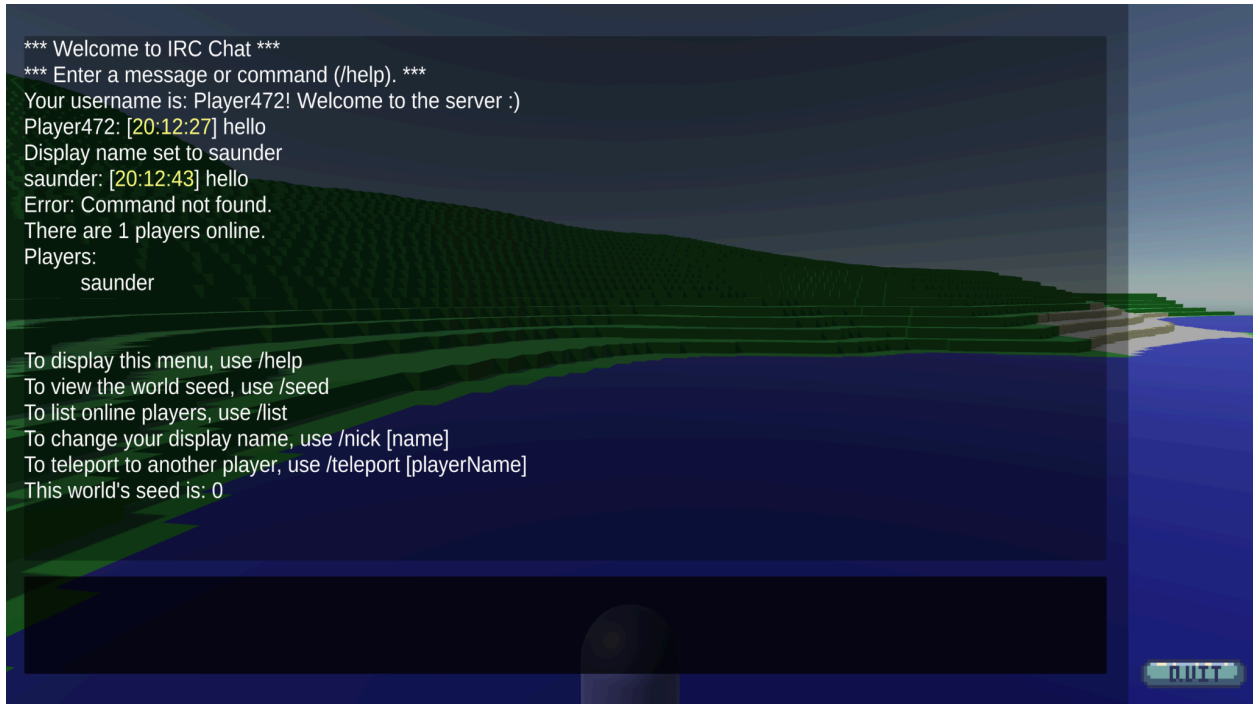
[FINAL MILESTONE] A look at what the player sees when inside one of the worlds.



[FINAL MILESTONE] A player admires their new home.



[FINAL MILESTONE] A player looks down into the tunnel they dug into the earth next to their home.



[FINAL MILESTONE] The escape menu that holds the chat window, showcasing some of the different commands available.

6. Conclusions

Summary

The focus of the Re:Voxel terrain generation engine was to develop an efficient, customizable voxel-based world generation engine in Unity. The project served multiple purposes; it provided an entertaining platform where users could explore and modify procedural worlds, offered a complete solution that other developers (and future groups) could build upon, and optimized performance for minimal cost incurred by the built-ins. Key technologies included Unity's Shader Graph, Mirror for networking, and the Unity Job system for task parallelization. Overall, the project addressed multiple challenges, such as chunk mesh generation, realistic terrain creation with spline graphs and multi-octave Perlin noise, and an efficient synchronization implementation through Mirror.

Notable contributions to this project included the successful implementation of threaded chunk and mesh generation, auto-profiling for performance testing, a fully networked chunk synchronization system, terrain modification tools, biomes, and an efficient voxel storage system through run-length encoding. The team also developed a user-friendly GUI for world customization and interactions. Results helped to indicate optimal chunk sizes and areas for future optimizations. The project achieved its goals, providing a versatile engine for voxel terrain generation.

Future Work

Certain components of our engine are either half-baked (the Agent Tasking system) or tightly coupled (Networking & the World class). Future work would be to flesh out these features and reduce coupling.

Customizing World Generation:

- In Scripts/World/Chunk/ChunkFactory.cs, the spline points for our 5 different noise curves are defined. These can be adjusted to affect how worlds look. Others can be added to further influence the terrain. We use two arrays for each, one to define the values of the noise, and the second to map those noise values to terrain height.

Adding Biomes:

- Currently, we have 2 biomes, desert and plains. More can be added by editing Scripts/Jobs/ChunkGenJob.cs. The DecideBiome method would need to be modified in order to assign different areas of terrain to the appropriate biome, which then could affect the type of voxels being generated inside of it as well as the terrain decoration. If someone wanted to add new biomes, they would also need to add to the Biome enum within ChunkGenJob and then appropriately modify the CarveTerrain method.

Adding Terrain Decoration:

- This has not been implemented, so it could be up to the new team to decide how to go about this. One way would be to add new methods to ChunkGenJob that would be called within CarveTerrain to generate different decorations (foliage, structures, etc.).

Additional Persistence:

- While there is currently some manner of persistence, implementing persistence that utilizes the VoxelRun data structure via writes to file would be beneficial. After implementing this (likely by extending the UnloadChunk method), it would be very easy to save worlds to disk (just exit the world). Then there would have to be some sort of list of loadable worlds in the registry.

Project Value

We learned the true value of project planning and the pitfalls of the lack of planning in the exact same project. Proper planning helped us get to where we are with a (mostly) robust engine. CRC cards, thinking, team meetings discussing high level ideas of the engine worked greatly to our advantage far into the future. Poor planning landed us in a pit when it came to networking. We didn't know enough about our networking solution (regarding object hierarchy) early enough and that led us, in the end, to tightly couple some of the networking code into the core of the engine. We should have taken networking into account sooner.

We also learned much about optimizations and that in some cases, performance concerns come before object-oriented design principles. For instance, in the case of the Unity job system, job code should be strictly data-oriented. We discovered and took advantage of profiling and came to realize it is invaluable to cornering performance concerns. We discovered a lot about optimal voxel implementation and chunk sizes. Most relevant and interesting is perhaps our implementation of VoxelRuns; this implementation allows for highly compact lists in a fashion that resembles closely lists of sparse matrices.


If nothing else, we gained experience in medium-scale project (large for a class project) design and management.

Bios

- **Dr. Wesley Deneke, Mentor** – Deneke is a professor in the Computer Science Department. He leads student research projects that are currently focusing on how to simulate Human Workflows using 3D virtual worlds.
- **Parker Gutierrez, Grad Student** - Gutierrez is a masters student in the Computer Science Department at Western Washington University. He has completed a wide range of relevant courses including Game Design (321), Object Oriented Design (345), Operating Systems (447), Analysis of Algorithms (511), and is expecting to graduate with an MS in winter 2026. Worked with Dr. Filip Jagodzinski in creating an application to visualize large amounts of protein mutations, computed from raw data. Responsible for underlying engine implementation as well as directing design choices for this project.
- **Ian Cambridge, Undergraduate Student** – Ian Cambridge is a senior Honors undergraduate student in the Computer Science Department at Western Washington University. He has completed Game Design (CSCI 321), Algorithms (305), Object-Oriented Design (345), Artificial Intelligence (402), Operating Systems (447), as well as other relevant courses; he is expected to graduate with a Bachelor's of Computer Science in Spring 2024. Ian previously completed an internship with VSP Vision in cloud and web development, and as of May 2024 is working remotely as an Apprentice Software Engineer also with VSP Vision, with his work largely focused in .NET/backend web development. Additionally, Ian was selected for a research fellowship at Western Washington University visualizing positional data from Dr. John Lund's (recently patented) long-range radio tags in Blazor .NET, and Ian's undergraduate research is focused in HCI—developing the autism mentorship site under Dr. Shameem Ahmed and Dr. Dustin O'Hara's guidance. For the procedural world generation project, Ian is responsible for various aspects of the implementation, including facets of software architecture and object-oriented design in the engine as well as more concrete implementation like the addition of water to the world generator and networking.
- **Saunder VanWoerden, Undergraduate Student** - Saunder is a student in the Computer Science Department at WWU. He is expecting to graduate with a BS in Spring 2024. Saunder has completed an internship at Wandke Accessibility through the Internet Studies Center at WWU working as a web developer on the infrastructure for a peer mentoring program. Saunder will be responsible for UI design and development as well as helping to implement biomes.
- **Samuel Turney, Undergraduate Student** – Samuel is a senior student in the Computer Science Department at Western Washington University. He has completed Game Design (CSCI 321), Algorithms (CSCI 305 and 405), Operating Systems (CSCI 447), as well as other relevant courses; he is expecting to graduate with a BS degree in Spring, 2024. Samuel has worked as a Cyber Security Analyst for Western Washington University's Information Security department, with his work largely focused in Microsoft Defender for Endpoint, bash scripting, and Splunk database deployment and use. Samuel will be responsible for implementing voxel interactions and networking.

References

[1] 0 FPS - Meshing in a Minecraft Game, <https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/>

[2] Henrik Kniberg - Minecraft terrain generation in a nutshell,  Minecraft terrain generation in a nutshell

[3] https://en.wikipedia.org/wiki/Run-length_encoding