Vrije Universiteit Amsterdam

Bachelor Thesis

Voxelsim - Designing a System for Programmable Environments within a Modifiable Virtual Environment

Author: Zain Munir (2730761)

1st and daily supervisor: Jesse Donkervliet
2nd reader: Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 20, 2024

Abstract

Minecraft, one of the most popular games of the last decade is a sandbox game with a Modifiable Virtual Environment (MVE). In an `MVE`, a player can choose to change the environment at will with fine granularity. By itself, an MVE has many societal benefits like in education with the existence of Minecraft Education, but these systems do not scale well to a large number of players or construct complexity.

This thesis explores the design and implementation of a programmable environment into Opencraft 2 (an existing open-source MVE project) with programmable components that allows players to create and manipulate logic circuits. We evaluate its performance, scalability, and trade-offs between player interactions and circuit complexity. Our findings demonstrate that our implementation supports over 200 concurrent circuits with low impact on server performance (fractions of a millisecond on total frame time), as well as handling up to 100 players while maintaining over 120 frames-per-second, making our design a promising base for such systems in the future.

Contents

1

Introduction

Soon after the creation of the first electronic programmable computers in 1945 (1) came the creation of the first interactive video games (2). Starting in the 1950s, digital games have become one of the largest entertainment industries, generating over $180 billion in yearly revenue, and cater to a wide range of use-cases for an audience of over 3.3 billion players globally (3). One such genre is educational games, with the first being created as early as the 1960s with PLATO (4). These games have evolved from largely text-based visuals into many genres as well, incorporating 2D and 3D graphics as they became mainstream. Because such a large proportion of the population play games, it is important for the learning tools to also adapt and make use of this medium for learning as an engaging supplement to standard teaching methods. A recent literature review (5) found that in the past several years, there has been increase in research into this field, specifically in the field of computer science. They found game-based learning where students played the game to be the most common approach and that game-based learning spans every educational level, from primary all the way to university.

1.1 Problem Statement

Minecraft, one of the most popular games of the last decade with over 300 million sales (6), is a sandbox game with a Modifiable Virtual Environment (MVE). In an `MVE`, a player can choose to change the environment at will with fine granularity. For example, in Minecraft, one aspect of its creation is using the `redstone` mechanic which is both an item itself as well as an all-encapsulating term used for anything technical created with it, its many variants and related interactive blocks. By itself, this kind of MVE has many societal

benefits like in education, but these systems do not scale well to a large number of players or construct complexity (7).

One solution to this problem would be to extend an MVE type game with a set of core components based on this idea of `redstone` that fit most scenarios, with one such system of components being logic circuitry, and then investigate its limitations through real-world load-based experiments. It is important for such a game to scale with this added complexity to both support more simulation load as well as more concurrent players.

In this thesis, we extend Opencraft 2, a voxel-based existing open-source project which is currently a limited Modifiable Virtual Environment with only terrain generation and manipulation, to include a design of a programmable environment which incorporates logic circuitry. We then investigate the feasibility and effectiveness of this design from a performance and scalability perspective.

1.2 Research Questions

Questions surrounding this topic are related to all three steps of the project, namely the design, implementation and evaluation.

Q1: How to design an MVE with a programmable environment that supports hundreds of users?

A programmable MVE that scales is important as through it, we can show the potential for larger interactive platforms that allow for many users to participate.

Such an environment would be generally useful. First for researchers, because by extending an open-source MVE application with such features, we provide a base for future research into similar topics as well as a prototype that can be built upon to include more features. Second, MVEs provide societally beneficial features with large potential in areas such as education and entertainment. For example, a scalable MVE would allow for more engaging approaches for large-enrolment classes.

However, existing systems are unable to fulfil this potential due to limitations on their performance and scalability. Designing such a system is challenging because no such system currently exists. It is also not known how to simulate real-time virtual environments efficiently and there is no known design process that guarantees a successful outcome.

Q2: How to implement such a system in a state-of-the-art MVE?

Having a prototype enables exploration in to new use-cases for such a programmable environment in education, entertainment or any others. A prototype also helps to highlight limitations and subsequently potential improvements to the original design.

However, selecting the correct tools to implement such a system is challenging because, although the amount of tools in the gaming ecosystem is large, their design goals differ both from each other and from the design proposed in **Q1**, making their integration difficult in achieving the end goal.

Q3: How to evaluate such a system?

To evaluate the implementation and show its efficacy, we need to design representative workloads that allow us to provide evidence through metrics to show the performance of the implementation. However, designing such workloads is challenging as we are building a large-scale system which does not already exist and so there are no examples of relevant work-loads we could adapt.

1.3 Research Methodology

Through our methodology, we hope to follow the `@Large` vision for massivising computer systems (8), as well as the computers systems and networking research manifesto (9).

M1 Design: By creating a design, we broaden our understanding and work towards engineering useful computer ecosystems through a scalable MVE that can support complex interactions and hundreds of users (8).

M2 Implementation: By implementing a prototype of our design, we can evaluate the efficacy of our approach through real-world experiments, showing a practical solution to our proposed problem.

M3 Experimental Evaluation: By using real-world experiments (which include using good performance metrics, representative workloads, a representative deployment environment, and a representative system configuration) to evaluate an implementation, we can see the feasibility of such a design.

1.4 Thesis Contributions

C1: A novel design for an MVE with a programmable environment that supports hundreds of users and can be used as a foundation for teaching logical circuitry concepts in Computer Systems education.

C2: We implement a prototype of our design using Unity and Opencraft 2, a state-of-the-art research platform and MVE. Our prototype is publicly available on Github (10).

C3: An extensive set of real-world experiments evaluating our approach. We design a set of experiments with novel workloads and performance metrics to evaluate the efficacy of our approach, conduct these experiments in a representative environment, and report on our findings.

1.5 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment (excluding the text in section 1.5).

2

Background

2.1 Modifiable Virtual Environment

As defined in (12), a modifiable virtual environment is a real-time, online, multi-user environment which allows its users (i.e., players) to modify the virtual world's objects (e.g., player apparel) and parts (e.g., terrain), create new content by connecting components, and interact with the world through programs.

This kind of environment can be found in both 2D and 3D games such as Terraria and Minecraft which are described later. Because an MVE allows for granular modification of the environment, a large quantity of information is stored related to the game state in static terms. Then, factoring in distinct actions from every connected player and the synchronisation of state between players and the environment based on all actions, the server state receives many updates. These updates then need to be propagated back to each player with minimal latency for a satisfying user experience. However, by increasing the number of players, we increase the load on the server and therefore increase latency which frames the main challenge with such a system: scalability.

A general model of an MVE for our purposes can be seen in Figure 2.1. Because the focus of this thesis will be the programmable environment ❶, we abstract away the exact communication protocol between the player and hosting environments ❷ and while present, we do not focus heavily on terrain generation ❸. Instead, our main focus will be the player-defined constructs that make up the programmable environment and the internal server programming needed to fulfil such a system.

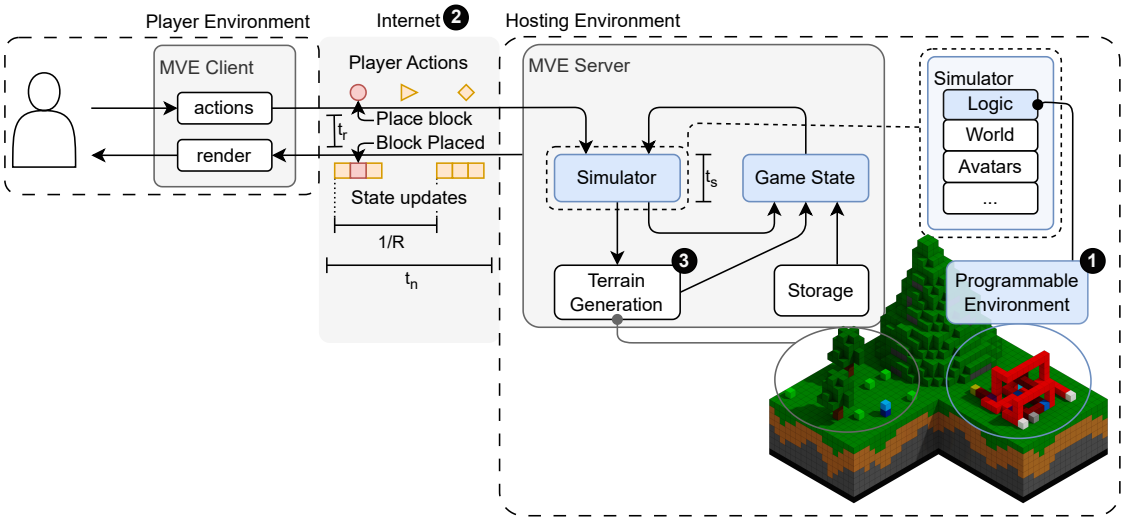


Figure 2.1: Model of MVEs with a focus on programmable environments highlighted in blue, adapted from (11).

2.2 Minecraft

Minecraft is an MVE-type game developed by Mojang Studios which can be considered the genre-defining game for its category in 3D sandbox games. Over many years, increasing numbers of features have been added to the game to make it endlessly re-playable. Outside of varying block types generated in the environment naturally, even more can be derived and created using the in-game crafting system. Minecraft also has many other features such as non-player characters (NPCs), both friendly and not, various player interactions such as flying and swimming and the complex `redstone` system.

Naturally as one of, if not the most, popular games on the planet, lots of research has been done on its capabilities both functionally and socially. Socially, Minecraft is already a great tool for education, especially with the existing Minecraft Education Mode which has catered lessons in many topics such as those in STEM as well as resources for teachers. While this system is closed-source, it shows that MVE's are a viable tool for education, with existing research showing the same (13, 14).

2.3 Terraria

Terraria is a 2D sandbox game developed by Re-Logic. Similar to Minecraft, there is a vast array of features, with many block types, NPCs and player interactions. However, unlike Minecraft's seemingly infinite world, Terraria has tiered world sizes. The world is

populated with varied biomes both above and below ground. Terraria is more survival focused because it does not have a creative mode natively, meaning players need to progress through the game, gathering resources and defeating bosses along the way.

2.4 Logic Circuits

Logic circuits are the electronic circuits made of various logic gates that perform logical operations on binary inputs to produce binary outputs. Using boolean functions, increasingly complex computations can be done which is the backbone of how computers process information. Having an understanding of logic circuits is beneficial because it showcases how larger problems are broken down into smaller ones to perform some function, as well being essential to understanding the foundation of digital electronics.

A special class of Logic gates are universal ones, such as a `NAND` gate (15) which is comprised of an `AND` and a `NOT` gate. A universal gate can be used to construct any other type of gate and subsequently any circuit. `NOR` gates are another such gate. However, only using universal gates result in larger circuits. Therefore, in our implementation described in Chapter 4, we decided to implement `AND`, `OR`, `NOT` and `XOR` gates instead to allow for easier circuit creation while also being able combine either of the `AND` or `OR` gates with the `NOT` gate to retain this universal property.

3

Design

In this section, we establish our design for a programmable environment within an MVE to answer **Q1** and briefly discuss how this design will be realised by using an existing MVE system.

3.1 Stakeholders

This project has two main stakeholders, fellow researchers and educators. For researchers, we're expanding the field by adding to an existing open-source MVE application which can be built upon and improved - in this instance by adding a visual system rooted in logical circuits. For educators, seeing the success of MVE like games, such as Minecraft Education, making use of an interactive environment, students can have an engaging way to learn about computer science topics. Also, allowing many students to occupy the same environment encourages collaboration.

3.2 System Requirements

This design should be incorporable in any MVE. As such, it has the following requirements:

1. The system should update the logic state and the environment within one second.

   Circuit creation is inherently player interaction based, and placing blocks takes time. This one second is chosen as to not overload the system with unnecessary work as well as remaining in the timescales of player behaviour.

2. The system should support hundreds of players concurrently and allow for hundreds circuits to be created and evaluated.

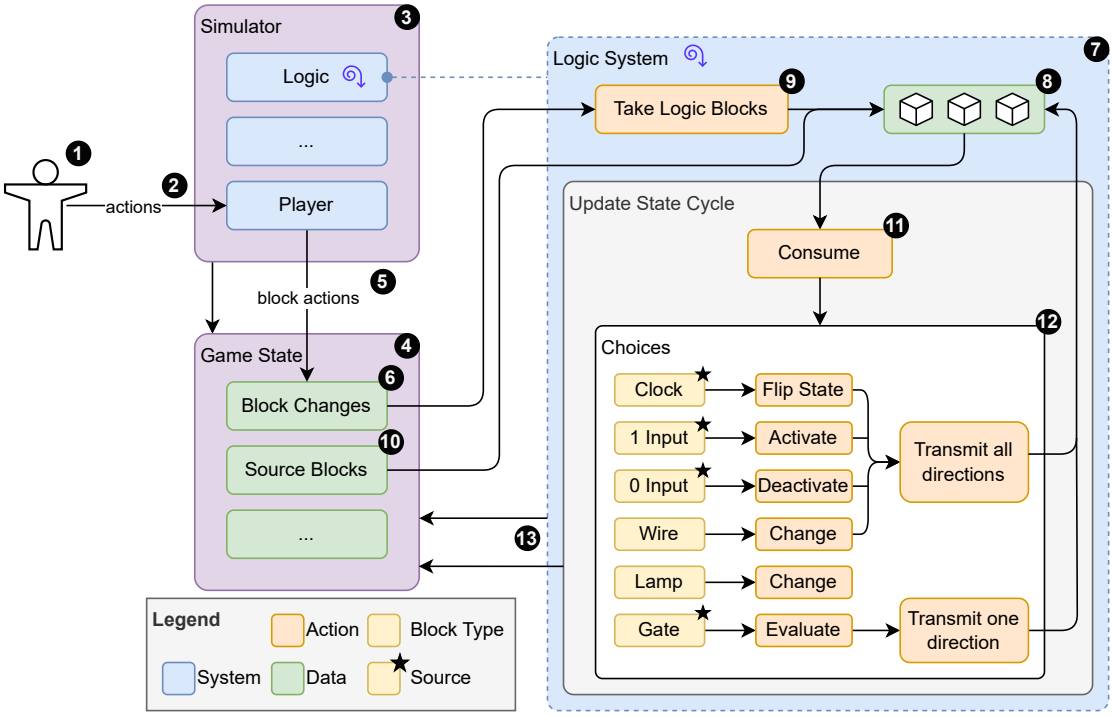


Figure 3.1: Design Overview of a programmable environment within an MVE.

With a future goal of the design being used as a tool to teach logic circuit concepts, we want to be able to support many players for large class sizes. The benefit of such a system would be allowing players to interact with the environment and explore how circuits work, meaning we want to support just as many circuits.

3. The logic system should have low additional resource usage on top of the existing system by ignoring redundant unnecessary work and updating state based on events.

   Existing MVE's, such as Minecraft Education which is limited to at most 40 players with the minimum requirements supporting around 4 players and the recommended hardware struggling with upwards of 10 (16), struggle to support hundreds of players. Our system needs to use the available resources optimally to be able to reach large players counts while also allowing for the programmable environment to be used.

3.3 Design Overview

The design overview is shown in Figure 3.1.

A player (1) is placed in the MVE and can perform a variety of actions (2) which are captured and sent to the MVE simulator (3). These actions include movement, and block placement, destruction or interaction. All actions cause some change in the state of the game (4) kept in memory, but the latter group of block actions (5) are tracked temporarily in the block changes (6) list.

Within the MVE simulator is the logic system (7) which is responsible for updating the logic states and attached visuals of the environment based on the presence and evaluation of circuit type blocks. It does not run on every update of the simulator, but rather has a configurable interval time. This behaviour is achieved by creating a queue of logic type blocks (8). This queue is comprised of a filtered list of block changes (9) and a tracked list of all logic source blocks (10) which include clocks, active and inactive blocks as well as gates.

This logic block queue is then drained one block at a time (11) in the update cycle. Every placed block has some properties associated to it, including the block type, logical state and directional placement. Different choices (12) are made depending on the block type to update internal block state and decide on the next action. If the block type allows for transmitting state, all six directions (for each side of a cube) are assessed and if a logic block is found, that block is added to the end of the queue. Gate blocks behave slightly differently than other blocks. They use their directional placement to specify which sides are inputs and which side is the output. Then depending on the type of gate, it evaluates its internal logical state before adding the output direction to the queue if a logic block is present there.

Throughout the duration of the update cycle, changes are made to the game state (13). The updated logical states are transferred back into game state. Additionally, after the block changes have been fetched, the list is cleared to prevent redundant future work. The logic system runs throughout the duration of the simulation at fixed intervals to keep the programmable environment consistent with the circuitry placed within it.

3.4 Opencraft 2

In this thesis, to realise our design, we extend an existing project for our prototype implementation. Opencraft 2 (17, 18) is an open source bare-bones implementation of an MVE developed under the `@Large` research group (19) using the Unity game-engine (20). It was created to facilitate research into the network and performance scalability of `MVE` type systems. The current implementation supports on-the-fly terrain generation and basic player

interaction such as movement, jumping, as well as placing and destruction of a single type of block. We extend this original project with our design which requires some extra features such as more block types, the ability to place specific blocks and block-to-block interaction as well as player-to-block interactions.

This implementation is created using the entity component system (ECS) (21). This is software-architectural design pattern commonly used in games to represent objects. Entities are unique identifiers for objects in the game world which contain components comprising of data that define some value for the entity. The components and entities themselves do not hold any logic or specific behaviour, but systems that operate on them define the behaviour of the game using the information from the entities.

4

Implementation

In this section, we showcase our implementation of a programmable environment in an MVE to answer **Q2** as well as highlight other relevant systems needed in its creation.

4.1 What Are Blocks? Our Smallest Unit of the MVE

The game environment is a world which is arranged in chunks of 16 blocks in all 3 dimensions for a total of 4096 blocks per chunk. Each chunk has an `Entity` and data for each chunk is stored in various `Component`s and `DynamicBuffer`s that are attached with the corresponding chunk's entity. Each of these can be obtained from various lookups on the `SystemState` using the original chunk `Entity`.

A subset of these buffers are related to blocks. They each have a capacity of 4096 and so by using the local coordinates which is a set of three numbers (x, y, z), we can obtain the unique index of a specific block within its chunk. With both the index of a block and the property buffer for the containing area, we are able to obtain properties associated with that block.

In short, each block is not independent, i.e. every block does not have its own `Entity` but rather an entry in the chunk to which it belongs.

The original project only incorporated one block property buffer which was used to define what the type of block at each index was for texturing purposes. We've extended this with two additional buffers for the logic state of each block with a simple `Boolean` of either true or false, and the directional placement of the block which is any of the 6 directions corresponding to the 6 faces of a cuboid block, as seen in Figure 4.1 with label ❶.

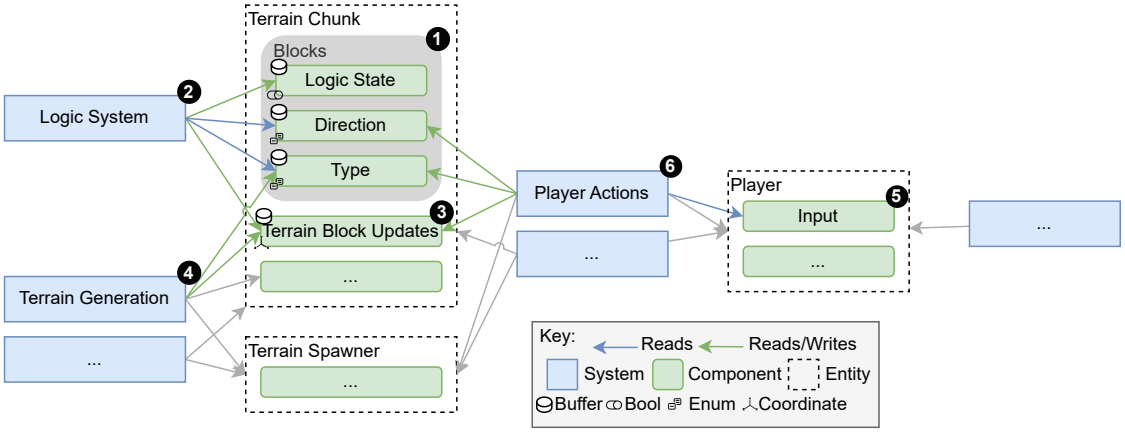


Figure 4.1: Implementation Overview, adapted from (22).

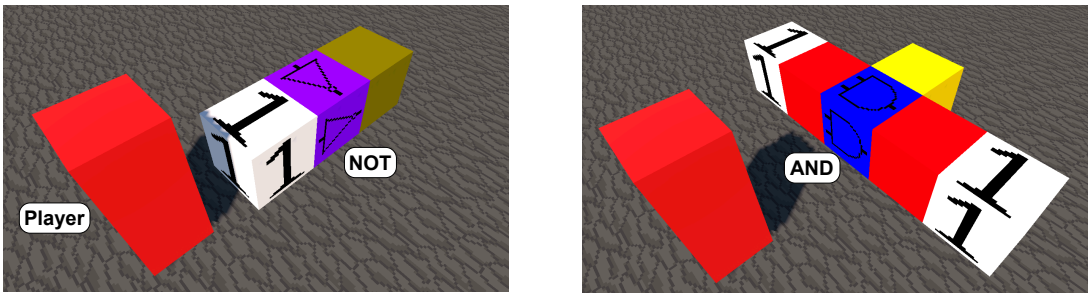
Name	Type	Description
Input	Source	Placed in off state. Toggleable by player between on and off
Clock	Source	Flips logic state every update cycle
Wire	Transmitter	Transfers logic signal along any face
Lamp	Receiver	Receives logic signal but does not transmit it
AND	2-Gate	Outputs on if two inputs are on
OR	2-Gate	Outputs if one or more inputs are on
NOT	1-Gate	Flips input signal for output
XOR	2-Gate	Outputs if one and only one input is on

Table 4.1: New logic block types.

4.2 New Block Types

The original set of blocks had various designs to mimic aspects of an environment such as stone, dirt, grass, wood, leaves etc. All of these blocks behaved functionally the same, in that they had a specific texture and could be broken. We add several new blocks which can do more, as shown in Table 4.1 by extending the `BlockType enum`. We also add several helper functions to the `BlockData` class to streamline certain reused conditional checks.

Gate blocks behave similarly to their logic counterparts. When placed, appropriate faces of the block become inputs and the face away from the player becomes the output as shown in Figure 4.2.



(a) One input gate. **(b)** Two input gate.

Figure 4.2: Gate placement visual.

4.3 Logic System

After the game initialises, the logic system starts ❷ in the `SimulationSystemGroup`. This group also contains other terrain related systems like the generation of new chunks, as well player interactions with the terrain as all of these systems are handled on the server. To facilitate the logic system's operations, we add another `DynamicBuffer` to each terrain chunk ❸.

Every update cycle for every chunk, an environment-level update is performed by looping through this buffer and extracting relevant coordinates into local state and clearing the buffer. This can be seen adapted from the design in Figure 3.1. Originally, we abstracted block changes into being a directly accessible portion of game state. However, in the actual implementation, each chunk keeps track of its own changes and we combine and then clear all the buffers. This update cycle happens at a configurable rate based on command-line arguments when running the game.

Any source-type blocks need to propagate their state to adjacent blocks. This is done through a `Queue` where the gathered input-blocks propagate their current state to their adjacent blocks - active gate blocks have direction and so only consider a subset of their adjacent blocks. If the adjacent block can receive logic, change its state and can transmit state, they are added to the queue. This queue is created and exhausted during each run of the logic system.

Gate-type blocks have specialised conditions to be evaluated to the on state. Depending on their placed direction, they consider certain sides to be inputs. If conditions based on the inputs are met, they evaluate accordingly and can update their adjacent output during the next cycle.

Any modifications in the terrain prompted by player interactions such as destroying or placing logic-type blocks result in the transmission of an off-signal down all available paths. This is done because any destroyed blocks may result in a break in the chain of logic such as removing a wire block in the middle of a length of wire. Similar reasoning applies to placing blocks because they may complete a chain and so need to be reevaluated themselves and subsequent blocks in the chain.

4.4 Terrain Generation System

The terrain in the world is structured in layers and generated ❹ in chunks of volume whenever a player crosses a threshold where they are within 3 chunks of distance from an unexplored area. This is done by the `TerrainToSpawn` system. It loops over all connected players and calculates the chunks that are within the players' location and if they have not been already generated, it marks the location of that chunk. Then, the terrain generation system loops over these marked chunks to create and initialise all of the relevant chunk entities. This includes adding all of the components and buffers to the new entity itself as well as filling in the values for block type based on the terrain configuration set.

The terrain configuration layers have different types which specify how the terrain should be generated.

Absolute - with a specified height and depth, flat layers of blocks of a specific type are placed horizontally.

Additive - uses noise derived from a `seed` - random or otherwise - to generate varying heights of blocks between two values.

Surface - one layer of blocks over the top of the existing terrain, matching the variation in the terrain.

Structure - used to create shapes such as trees which have specific rules that govern how they are formed.

Calculated - created for this thesis, allows for a systemic placement of blocks to create basic patterns of circuitry.

These layers can then be combined into terrain layer collections to create a configuration which is used in the terrain generation of the world. Multiple of these collections can be added to a single configuration and using specific command-line arguments during start-up, one can be selected. In the original project, the stand-out collection generates varying complex terrain and structures to represent hilly terrain. For the purposes of this thesis,

we create several new collections that allow for varying amounts of circuitry blocks within a small amount of chunks.

4.5 Additional Utilities to Support Development and Evaluation

Some features were not strictly necessary to achieve the intended design, but were nice to have during the implementation, testing and evaluation.

4.5.1 Player Interaction

The player character spawns into the world on top of any existing terrain at the global zero position (0, y, 0). They have various actions such as looking around, movement, jumping and placing or destroying blocks in the environment around them.

From the original research, because it was focused on terrain generation more than it was on player creation, the only placeable block was stone. We extend the functionality here by adding two more inputs ❺ which then translate into actions ❻ causing a traversal through a list of available blocks either forwards or backwards. Then, when the player tries to place a block in the world, they will instead place their selected block.

Certain blocks in the world can also be interacted with after they have been placed, such as the input block. To allow for this interaction, we've added a `Third Action` button which causes a toggle update on the currently selected block. For the input example, it toggles an off state to be on and an on state to be off.

The player has a constantly updating location in the environment and are also able to face in any direction. Blocks which operate as gates interact differently depending on orientation. By using the coordinate of the destination block when placing and the current location of the player, we can calculate the direction based on the greatest difference in coordinate out of the three directions (x, y, z).

4.5.2 Unbreakable Block

With the base game, player jumping is tied to the y-position of the player being over another block in the terrain. Because every block was breakable, a player could end up at the "bottom" of the terrain where they have no block under them and so are unable to jump to get out of that predicament. As such, an unbreakable block was added which cannot be destroyed using a simple conditional when trying to break blocks. This block is best used as a bottom layer to the world.

4.5.3 Command-Line Arguments

During the evaluation, being able to alter the way either the server or client instances behave is essential to simulate different workloads. For the server, in order to see the effect of having a logic system versus it being absent, an argument was added to disable the system. Also, configuring the specific terrain generation to allow more granular specifications of the circuit size was necessary. Existing arguments allow selecting different terrain configurations which we use to create a layered approach to adding more circuits. However, when evaluating the effect of other systems as compared to the logic system, limiting the number of circuits to a constant per run was necessary. To that end, arguments were added to select the area of chunks that circuits would be allowed to generate in. Any chunks outside of that area would generate without any circuitry.

4.5.4 Statistics System

For our evaluation, we mostly focus on the composition of server frame times. The existing statistics system tracked the overall time per frame but had no extra granularity. We extend the tracked statistics with relevant sub-portions of the total frame time such as various stages of the logic system and the terrain generation which allows us to see the effect of different systems on the overall system. We also keep a counter to track the number of input type and gate blocks to see the effective sizes of circuits.

4.6 Other Implementation Approaches

Certain parts of the translation of the implementation could have been done in multiple different ways, each with their own benefits and drawbacks. We discuss a few of these below.

4.6.1 Logic System Loop

We thought of two main approaches when implementing our design. The first is the simple and somewhat naive approach described above where the logic system operates sequentially on the main process. All blocks are considered one after another, regardless of the terrain area. This approach scales worse with an increasing number of terrain areas as each one adds an additional one that needs to be checked in sequence.

The alternative approach is a job based one. Instead of operating on the whole world, each terrain area is evaluated separately in jobs. These jobs can then be distributed among

multiple threads which on a multi-core system would be much faster, like the system used in the evaluation. There are more things to consider when using this approach such as circuits across chunk boundaries as multiple jobs may interact with the same blocks in their queues and could lead to abnormal results. Other systems in the existing implementation already use this approach such as the terrain generation system.

We did attempt to implement the latter approach as well, but due to time constraints, our results in the evaluation are entirely based on the simple approach.

4.6.2 Block State

In the original project, the extent of individual block state was just the block type. Every position in a chunk has a block type, including empty space. Therefore, having a fixed-size buffer attached to each chunk was the cleanest approach as indexing is easy and we need all of the memory space of the buffer.

For the logic system, we added two additional buffers for logic state and direction which operate on the same principle where each position in the chunk indexes into the buffer exactly. However, not every block needs these properties so we ultimately are trading memory for performance. The alternative here would be to link the positions to dynamically allocated values while introducing some overhead in this added complexity.

5

Evaluation

In this section, we evaluate our implementation to answer **Q3** by performing a variety of experiments on representative hardware and discuss the results obtained. We also discuss some pitfalls of our results.

5.1 Main Findings

MF1: Our implementation supports over 200 concurrent circuits with a 1% impact on the server performance. The decrease in performance is comparable to normal fluctuations when running the process which is within 0.5 ms §5.3, §5.6.

MF2: Increasing the number of players does increase the additional load caused by the logic system with or without circuits present proportional to the complexity of the terrain and number of circuits present as the FPS of the server decreases §5.4.

MF3: Increasing the complexity of the terrain with the logic system active leads to increased resource usage of CPU, memory and network bandwidth up to a maximum 10%, regardless of circuits being present §5.5.

MF4: There is an expected negative correlation between the maximum number of circuits and the maximum number of players §5.6.

MF5: The impact of the logic system on the overall system is low, but does cause a spike in frame time of up to and over 10 times during its evaluation §5.7.

5.2 Experimental Setup

We run our experiments on the DAS-6 supercomputer (23). This hardware is representative for a real-world MVE deployment because it consists of many compute nodes

Section	Focus	Workload	Duration / min
§5.3	Performance	Number of circuits	5
§5.4	Performance	Number of players	2
§5.5	Overhead	System statistics	2
§5.6	Player/Circuit trade-off	Players and circuits	2

Table 5.1: Experiment overview.

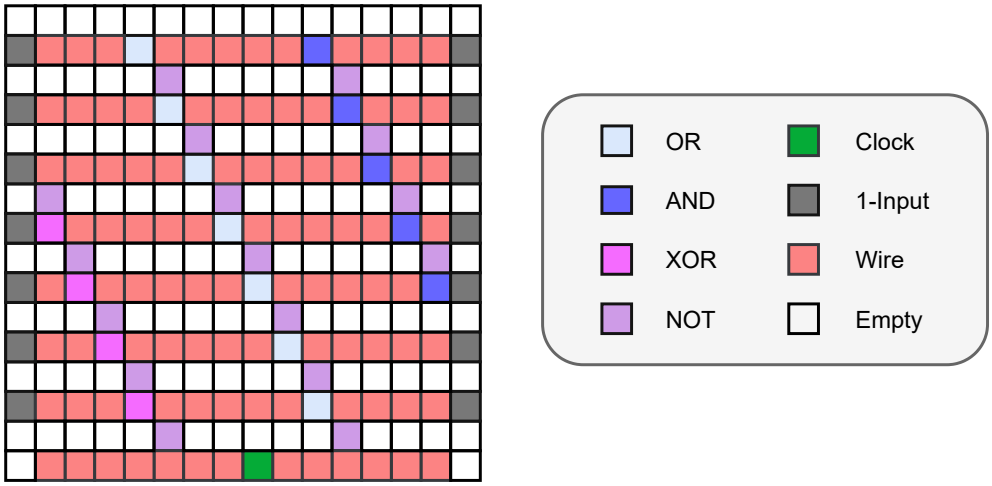
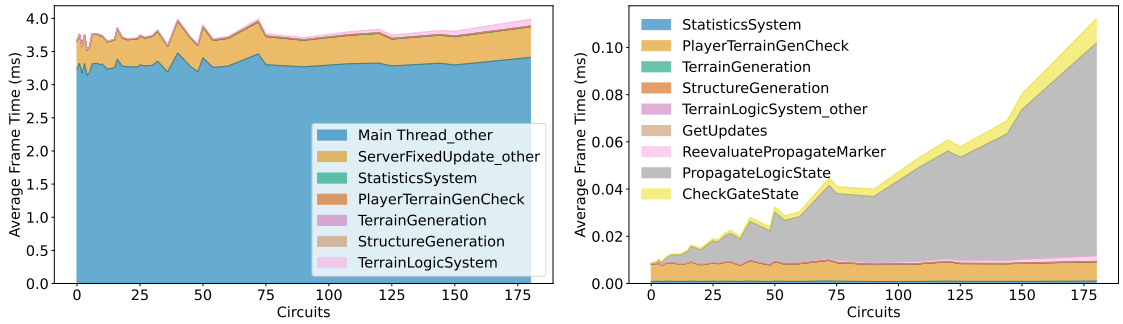


Figure 5.1: A schematic overview of the circuit used for each experiment.

throughout which we can distribute our workload such that the server has its own dedicated node and clients are interspersed evenly throughout other nodes. This system was designed for research and is the sixth version - the previous fives' history can be read about further here (24).

The smallest unit of terrain generation is a single chunk, which is a 16 by 16 by 16 cube of blocks. We have created a circuit which fits in one layer of a chunk as shown in Figure 5.1. Each circuit then contains 32 gate blocks and 15 input blocks, one of which is a clock to cause constant updates. Using a combination of different terrain types for multiple layers and command-line arguments, we can specify how many circuits are created per layer, how many layers are added and whether or not the logic system is active. We also use an argument to set the intervals in which the logic system updates at 1 second as discussed in Section 3.2.

We run the circuit experiments for 5 minutes because only 1 player is spawned for each run. The other experiments last for 2 minutes per run because initialising increasing numbers of players takes more time. This duration starts after the server instance has



(a) Including all systems.

(b) Logic system as compared to other similar impact systems.

Figure 5.2: Breakdown of frame time

properly initialised, the relevant number of clients have connected and once the terrain generation has stabilised.

For any circuit related terrain, we keep the generation basic with just a flat layer of stone and circuits placed on top. For terrain marked as `RollingHills`, we use the existing configuration from Opencraft 2 for generating complex terrain.

Using a combination of application-level and system-level statistics, we create graphs and discuss insights into the data.

5.3 Scalability in Number of Circuits

The goal of this experiment is to evaluate the performance of the server when the number of circuits in the world increases while only one client is connected. Through this experiment, we show that Voxelsim is able to support hundreds of circuits while having a small impact on server performance. The frame time increases but is similar to normal fluctuations in the system as it is within 0.5 ms of additional time.

Our result is depicted in Figure 5.2. The figure shows the frame time of the game for an increasing number of circuits. The horizontal axis shows the number of circuits used. The vertical axis shows the frame time, and the area in the plot shows how much time was spent on each ECS system.

This result shows that simulating blocks only takes up to 1% of the total frame time, and that over 90% of time is spent on tasks unrelated to circuit simulation. The former can be observed by the pink area (`TerrainLogicSystem`) in Figure 5.2a taking at most 0.1 ms. The latter can be seen in the figure by the light-blue area (`Main Thread_other`) consistently exceeding 3 ms, and the total frame time consistencly being less than 4 ms.

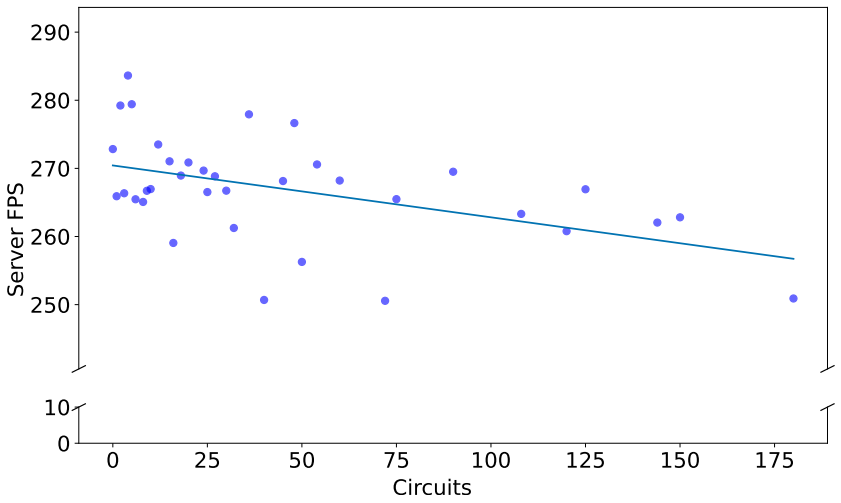


Figure 5.3: Server FPS with increasing circuits

To analyse how the `TerrainLogicSystem` scales with the number of circuits, we show a zoomed-in version of this plot, excluding ECS systems unrelated to our workload, in Figure 5.2b. In this figure, we see a constant portion for checking if new terrain needs to be spawned (`TerrainGenCheck`) as the numbers of players throughout the runs is always 1. The statistics system and terrain generation are negligible here as the maximum effect of the former is around 0.001 ms and new terrain isn't being generated throughout the experiment. The other portions in this graph break down the logic system into pieces. We can see that `GetUpdates` and `ReevaluateProgagate` are very small which is because there is no player interaction or terrain generation to prompt any collection of updates. The main portion is `PropagateLogicState` where the majority of the time is spent because this function transmits all the logic from inputs and active gates throughout the terrain. As the number of circuits increases, the load increases as well by a linear amount to the number of circuits.

In Figure 5.3, we show the frames-per-second (FPS) of the server. The horizontal axis shows the number of circuits used and the vertical axis shows the FPS. The points show each configurations' FPS with a line-of-best-fit to show the trend. From this figure, we can see that the FPS trends lower at a slow rate but remains well above 200 FPS.

Based on this evidence, we find that Voxelsim can support hundreds of circuits under this workload, assuming a minimum required FPS of 200 which results in a great user experience (25).

Terrain	Logic Active	Circuits	Description
Flat	False	0	Flat world with only 2 layers of blocks
Flat	True	0	Flat world with only 2 layers of blocks
2-Layer	True	72	Flat terrain with 2 layers of circuits
RollingHills	False	0	Complex terrain generation
RollingHills	True	0	Complex terrain generation

Table 5.2: Player experiment loads.

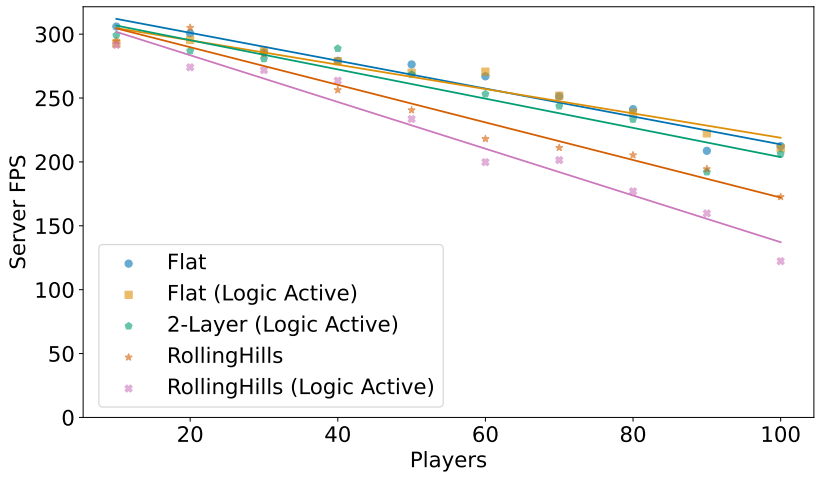
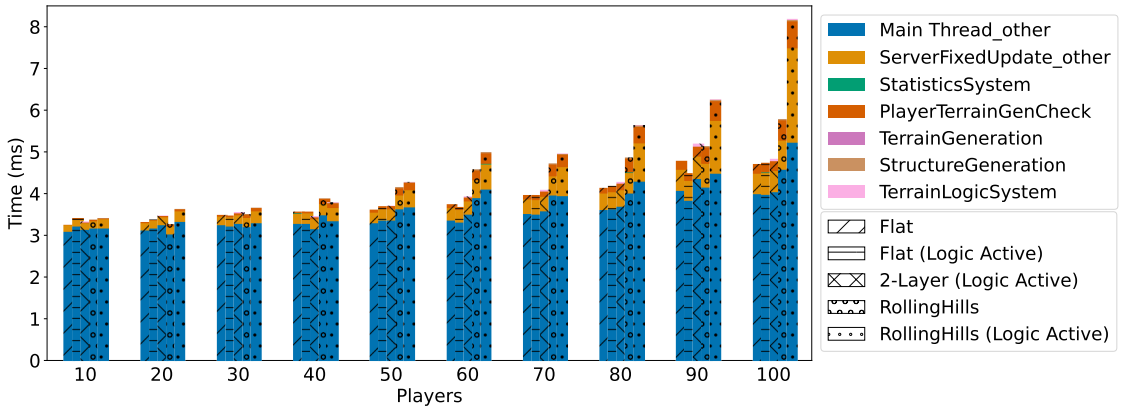


Figure 5.4: Increasing players FPS comparison.

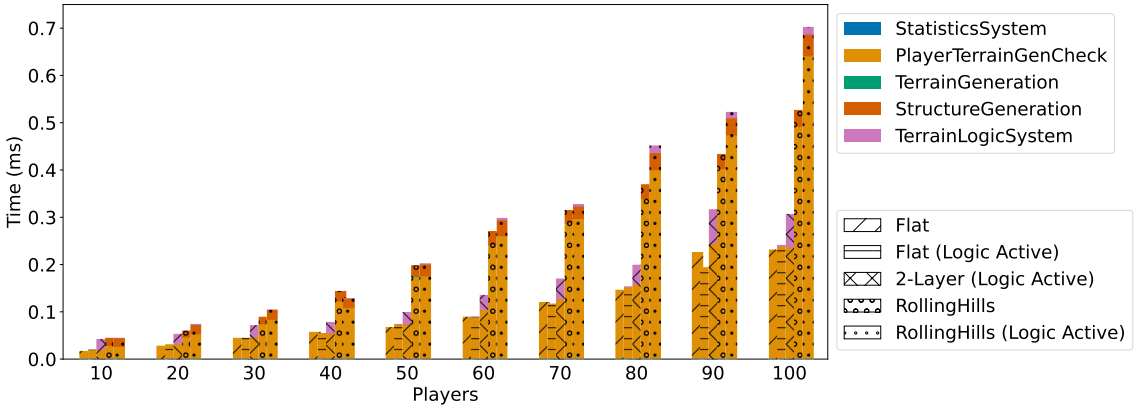
5.4 Scalability in Number of Players

The goal of this experiment was to evaluate the performance of the server as the number of players increases across different terrain and logic loads. The players have simulated movement confined to the first few chunks of the world. Through this experiment, we show that Voxelsim can support at least 100 players with logic active, complex terrain and circuits present.

Our result is shown in Figure 5.4, This figure shows the FPS of the server for an increasing number of circuits. The horizontal axis shows the number of players connected and the vertical axis shows the FPS. The points show each configurations' FPS with a line-of-best-fit to show the trend for each load. We have 5 different loads in order of complexity, as described in Table 5.2. As we can see from this figure, the general trend is that the more complex the terrain, the fewer players that can be supported. Having the logic system active also reduces performance at a constant amount without circuits and a



(a) Including all systems.



(b) Logic system as compared to other similar impact systems.

Figure 5.5: Breakdown of frame time with increasing players and varying terrain loads.

heavier amount proportional to the number of circuits, as we saw in Section 5.3.

To analyse the causes for the decrease in FPS, we show the breakdown of average frame time in Figure 5.5. The vertical axis shows the frame time. The horizontal axis shows the number of players connected with each bar in a grouping representing a different load according to the hatching in the legend. In Figure 5.5a, we see that the majority of frame time is spent on tasks unrelated to circuit simulation, which is the portions in blue (`Main Thread_other`) and orange (`ServerFixedUpdate_other`), the latter of which is increasing non-linearly with the amount of players. This is because it contains systems involved in simulating player movement and handling updating the clients, which makes sense because increasing the number of players increases the workload for these systems.

Zooming in on specific systems in Figure 5.5b, we can see that the majority of load here is caused by the `PlayerTerrainGenCheck` - this is proportional to the number of players because it needs to check for each player's position whether new terrain needs

to be spawned. Wherever the logic system is active, the length of updates increases as well. This is caused by the time taken to update state increasing due to overall load on the server and more terrain to check when complex terrain is generated. When there are circuits present, the effect is compounded with the additional time to propagate changes into the environment as well.

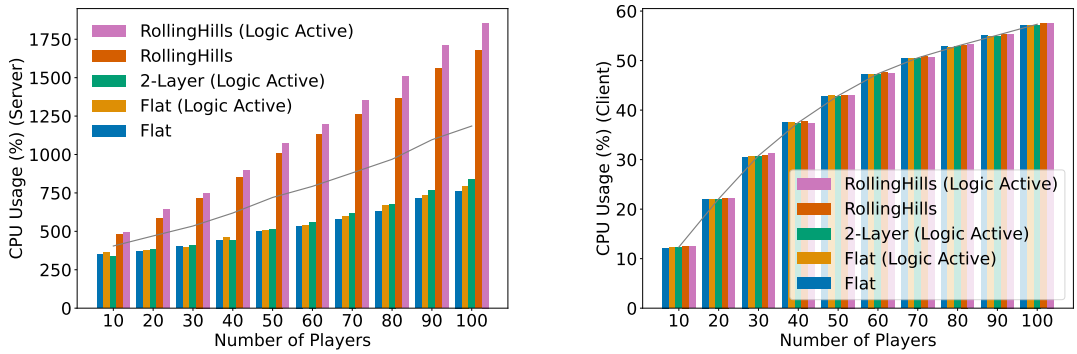
Based on this evidence, we find that Voxelsim can support up to 60 players while maintaining above 200 FPS but can support up to and beyond 100 players above 60 FPS under varying complex loads.

5.5 Overhead of Logic on the System

The goal of this experiment was to measure and compare the overhead of the logic system as compared to other systems in the project. We re-use the runs used in Section 5.4 in this experiment but instead take the system logs of both the server node and the average of the client nodes for each run. Through this experiment we show that having the logic system active has a maximum increase effect of around 10% for similar terrain on CPU, memory and network traffic but a more substantial effect to memory usage and outbound traffic when circuits are present. The logic system being inactive means that the simulation loop for logic does not occur, but additional buffers associated with block state for the logic state and direction have not been removed.

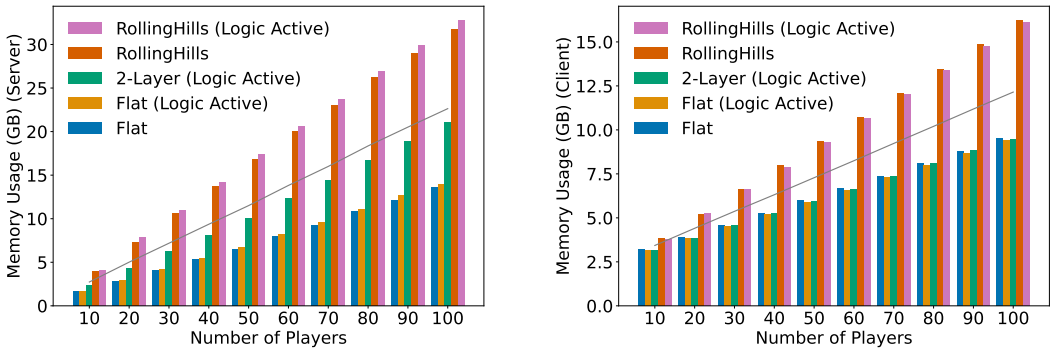
Our results are shown in Figures 5.6, 5.7 and 5.8. For each figure, the horizontal axis shows the number of players connected with each bar in a grouping representing a different load. The vertical axis shows the relevant measured metric being described.

In Figure 5.6, we can see the CPU usage. For the server, we were able to measure application-specific system metrics and so have the exact usage across multiple cores. For the client nodes, we have the total CPU usage across all cores. Naturally, for both the server and the clients, in all cases the CPU usage trends upwards. For the server, this is because it has to process more movement simulations for each additional connected player. For the clients, this increase is because each client node connects a tenth of the total players because we are using 10 nodes total to distribute the clients. For the server, the more complicated terrain generation results in additional load on the CPU because more terrain is generated overall and various calculations are used to generate it. The overhead of the logic system increases with this increase in terrain generation, even without circuits, as the logic system needs to check for updates throughout all the terrain even if there are none.



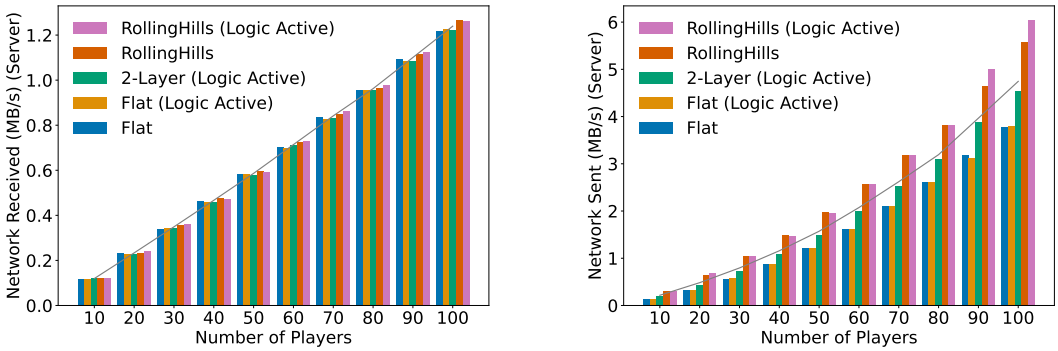
(a) Server. **(b)** Averaged Clients.

Figure 5.6: CPU usage.



(a) Server. **(b)** Averaged Clients.

Figure 5.7: Memory usage.



(a) Megabytes received by server every second. **(b)** Megabytes sent by server every second.

Figure 5.8: Server Network Bandwidth.

Looking at Figures 5.7a and 5.8a, we have the server's memory usage and inbound network bandwidth. Both of these increase linearly with the amount of clients connected. The more complicated terrain requires more space to store in memory which is why those bars have a higher baseline. When circuits are present, the memory usage increases as we store information about the circuits in state. Unlike the memory, the inbound network traffic is fairly close as each client only sends updates in the form of player actions which then change game state on the server.

Unlike the other graphs, 5.8b shows a non-linear increasing trend. The outbound network traffic increases at a faster rate as more players are connected. This is because each player gets state updates based on actions from every other player. For example, with 10 players each doing a single action, the server needs to send out 10 updates to each player for a total of 100 updates. With 20 players, this would be 400 updates, 30 would be 900 etc. When circuits are present, we are also sending out terrain updates based on the changes in logic state of the blocks.

5.6 Player/Circuit Trade-off

In the first experiment, we investigated the effect of the logic system by increasing the number of circuits in a world with one player and comparing the server frame time, finding it to have a minimal overall effect with the frame time trending upwards but at a slow pace. In the second experiment, we compared the effect of different systems and terrain types when increasing the number of players. Here we found that the logic system's impact increases with the number players, with or without circuits present. We also saw how more complex terrain compounds the slow down caused by more players.

With this experiment, we used the information gathered previously to directly compare the circuits and players to find out the optimum configurations that maximise the amounts of both while still performing sufficiently. Through this, we find that Voxelsim can support more circuits than players, with the numbers of each decreasing when the other increases.

Figure 5.9 shows the results condensed into a pareto front with FPS thresholds of 120 and 200. The horizontal axis is the number of players connected and the vertical axis is the number of circuits present. The gradient colour of the points show the FPS of that combination of players and circuits, with the exact number written next to the points as well. Looking at this figure, there is an expected negative correlation between increasing either circuits or players. We can see a sharper decline in performance based on increasing

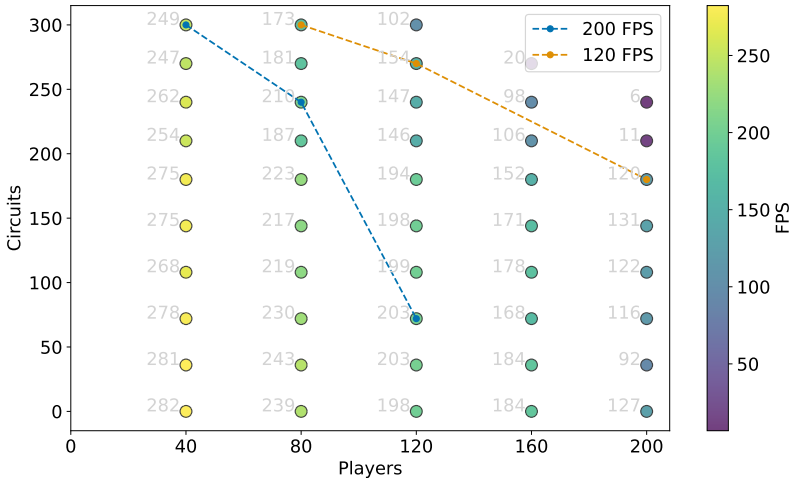


Figure 5.9: Trade-off between players and circuits based on FPS. The values in grey are the exact FPS values for that combination of players and circuits.

the number of players than increasing the number of circuits, implying the load of a single heavy circuit is less than the load of a player.

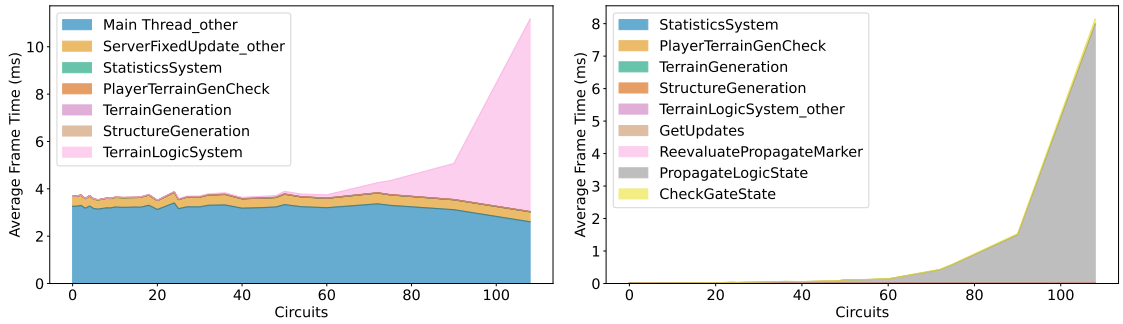
Based on this evidence, we show that there is a clear trade-off between the number of players supported and the number of circuits supported, but we are still able to support hundreds of circuits which have thousands of blocks with over one hundred players.

5.7 Result Caveats

As discussed in Section 4.6.1, our implementation of the logic system loop uses a simple approach where all circuits are evaluated sequentially in one loop. While the results shown in the previous sections are promising, this approach causes certain drawbacks.

The method of reporting our experiment results usually use average values with the main ones being for portions of the overall frame time and subsequently, the FPS. With this, we show that the impact of the logic system is low as compared to the overall system. However, the logic system and many others do not run every frame, with the logic system specifically running slower according to our configuration, based on the design requirements in Section 3.2. In the frames that the logic system operates, the frame time spikes to over 10 times as much duration because of our single-threaded simple approach, which can lead to stuttering behaviour on the visuals.

We discovered this impact after completing all of our experiments and decided to do a deep dive into the composition of time spent in the logic system during these specific



(a) Including all systems.

(b) Logic system as compared to other similar impact systems.

Figure 5.10: Old implementation composition.

frames using Unity's built-in Deep Profiler and found that the majority of the frame time was spent operating on buffers. This includes getting the buffers, reinterpreting buffers for modifications and getting entries from the buffers. These operations are directly related to the amount of logic blocks present, and so as the amount increases, so does the impact of these operations. In order to mitigate this, we could use the more advanced approach by spreading the load across multiple threads or using something other than buffers as discussed in Section 4.6.2.

5.8 Improvements throughout Evaluation

All of the evaluation results shown above, as expected, were not from our first build of the implementation. The first version of our implementation had some glaring issues, which become obvious when comparing the graphs in Figure 5.10 and Figure 5.2. The main culprit is the `PropagateLogicState`, which is where the bulk of the logic system work is. As we use a clock, this function has to do the maximum amount of work possible each cycle it runs for the corresponding configuration. It originally had several problems.

Active signals for blocks come from two sources, active inputs or active gates. These are stored separately to make operating on them easier. However, in the original implementation we would concatenate the two lists containing these blocks together and then loop over them, which is quite slow (26) compared to other methods. As we did not want to modify either list, we split the propagation of this signal into two function calls instead, which leads to the same effect.

With enough of these small improvements, the effect on the frame time became much less, by over a factor of ten for a comparable number of circuits. This resulted in us

increasing the number of circuits we tested with, both in the first experiment and the second where we tripled the amount of circuits used in `2-Layer` with the overall impact remaining about the same.

6

Related Work

In this thesis we design, implement and evaluate a programmable environment created in an MVE. This concept of an MVE was was defined in (12). This previous research also discusses existing models of deployment of such an environment and the limitations and challenges in scalability they present. The authors go on to explore a conceptual server-less model that uses differentiated deployment, which means it would adapt and scale the system according to the available resources, allowing the deployment to service more users.

This project relied heavily on and was built on top of the existing open-source Opencraft 2 implementation (22). In their research, they go over the existing infrastructure for online game deployments, the system model for online games and how networking plays into it. They then go on to explain, design and develop an aforementioned differentiated deployment system called PolkaDOTS. They used this deployment when creating Opencraft 2 and evaluated it to showcase its, and differentiated deployment's, performance.

There has also been research into the best way to benchmark MVE type systems in terms of both scalability and network usage, one of the first of which was Yardstick (27). They designed a benchmark whose load consisted of different complexities of virtual worlds and player emulation which would gather system and application-level metrics to then evaluate the performance of an MVE. Another benchmark, Meterstick (7), builds on the findings and systems of Yardstick to investigate the performance variability of different workloads on MVE-type games.

The authors of Servo (11) designed a server-less backend architecture for MVEs and then evaluated their prototype on commercial server-less platforms to show in practice the potential for server-less designed deployments of MVEs in how they can outperform other deployments like those found in Minecraft.

7

Conclusion

The goal of this thesis was to investigate the scalability of incorporating programmable environments into MVEs. We ultimately decided that the concepts related to logical circuits were the best fit for this programmable environment and so designed, implemented and evaluated a mapping of logic circuits into a 3-dimensional space and found it to be scalable to both hundreds of circuits and hundreds of players, making it a potentially useful foundation for a tool used in teaching these concepts because many students can join and participate concurrently.

7.1 Answering Research Questions

Q1: How to design an MVE with a programmable environment that supports hundreds of users?

In Chapter 3, we adapted the original design from the Opencraft 2 project to incorporate a programmable environment system grounded in the same logic components such as gates, inputs and outputs that make up the logical circuits found in computer hardware. We discussed the benefits this design had to its potential stakeholders, the requirements our system should have and the most relevant aspects of the system in more detail.

Q2: How to implement such a system in a state-of-the-art MVE?

In Chapter 4, we realised our design by implementing it within the existing MVE, Opencraft 2. We discussed the existing system to help contextualise the places where our design lives within the project, as well as going into the most relevant portions in detail to explain one approach.

Q3: How to evaluate such a system?

In Chapter 5, we saw how our system performed under load compared to other systems found in a typical MVE by running it on representative server hardware, DAS-6. By seeing the effect on the system as a whole, we showed the low impact of our system and how it scaled with increasing numbers of circuits and players.

7.2 Limitations and Future Work

This work was predominantly limited by both time due to its nature as a Bachelor Thesis and lack of experience working with both Unity and existing large projects. Time played a factor in the way it gradually limited scope from the initial lofty ideas to what we ultimately created with the logic system. With an original focus on MVEs' use in education, many additional features such as proximity voice chat, nametags for players, saving of worlds were considered as well but became quickly out of scope of the core idea. Inexperience also played a factor, especially in the beginning of the project where time was spent learning the basics of both Unity and C# to then also spend time learning how the existing project worked to be able to add to it. This inexperience also means that while we created a specific approach to implementing typically 2D logic gates into a 3D environment, we did not do an exhaustive search into all approaches using more complex Unity paradigms. This project also took the meaning of a programmable environment to only include logic gates because universal gates allow any circuit to be created - however, programmable environment can be taken to mean many more things, such as creating structures in game using scripts applied to blocks, moving existing blocks and potentially infinite more examples.

In terms of future work, the existing project can be built upon. Currently, we have basic circuitry but adding features to allow saving and miniaturisation of circuits down to a single or a few blocks would allow more complex constructs to be made, making the environment more useful.

From a technical standpoint, our system is currently sequential and therefore, slower than it can be through instead using a job-based approach. We also make specific performance-memory trade-offs in our implementation that can be investigated in more detail. Both of these are explored conceptually in Section 4.6.

Additionally, in our thesis, we focus in our evaluation on a quantitative approach. However, because we intend for our system to be used in education as well as research, a qualitative evaluation with feedback from users would be worthwhile to obtain. To do such an investigation, many features could be added. For example, better UI showing

which block a user has selected to place, player-to-player interaction such as proximity voice chat, directional textures for better responsive visuals and the ability to save the modified environment.

References

[1] Thomas Haigh and Paul E. Ceruzzi. *A new history of modern computing*, pages 9–11. The MIT Press, September 2021. 1

[2] Steven L Kent. *The Ultimate History of Video Games, Volume 1: From Pong to Pokemon and Beyond... the Story Behind the Craze That Touched Our Lives and Changed the World*, **1**. Crown, 2010. 1

[3] Newzoo. **Newzoo Global Games Market Report 2023**. Accessed on August 9th, 2024. 1

[4] Stanley G Smith and Bruce Arne Sherwood. **Educational Uses of the PLATO Computer System: The PLATO system is used for instruction, scientific research, and communications.** *Science*, **192**(4237):344–352, 1976. 1

[5] Maja Videnovik, Tone Vold, Linda Kiønig, Ana Madevska Bogdanova, and Vladimir Trajkovik. **Game-based learning in computer science education: a scoping literature review**. *International Journal of STEM Education*, **10**(1), September 2023. 1

[6] Wikipedia. **List of best-selling video games**. Accessed on April 18th, 2024. 1

[7] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In *Proceedings of the International Conference on Performance Engineering, Coimbra, Portugal, April, 2023*, 2023. 2, 33

[8] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent van Beek, and Lucian Toader. **Massivizing Computer Systems: a Vision to Understand, Design, and Engineer Computer Ecosystems through and beyond Modern Distributed Systems**. *CoRR*, **abs/1802.05465**, 2018. 3

REFERENCES

[9] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, 2022. 3

[10] **Voxelsim Prototype**. 4, 43

[11] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L. Abad, and Alexandru Iosup. **Servo: Increasing the Scalability of Modifiable Virtual Environments Using Serverless Computing**. In *43st IEEE International Conference on Distributed Computing Systems, ICDCS 2023, Hong Kong, China, July 17-21, 2023*. IEEE, 2023. 6, 33

[12] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems**. In Amar Phanishayee and Ryan Stutsman, editors, *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020. 5, 33

[13] Steve Nebel, Sascha Schneider, and Günter Daniel Rey. **Mining Learning and Crafting Scientific Experiments: A Literature Review on the Use of Minecraft in Education and Research**. *Journal of Educational Technology Society*, **19**(2):355–366, 2016. 6

[14] Noelene Callaghan. **Investigating the role of Minecraft in educational learning environments**. *Educational Media International*, **53**(4):244–260, 2016. 6

[15] Wikipedia. **NAND Logic**. Accessed on June 27th, 2024. 7

[16] Mojang. **How To Set Up A Multiplayer Game**. Accessed on August 9th, 2024. 10

[17] AtLarge. **OpenCraft Website**. Accessed on May 16th, 2024. 11

[18] AtLarge. **OpenCraft GitHub Repository**. Accessed on May 16th, 2024. 11

[19] AtLarge. **@Large Research Group**. Accessed on May 16th, 2024. 11

[20] Unity. **Unity Website**. Accessed on May 16th, 2024. 11

[21] Wikipedia. **Entity component system**. Accessed on August 9th, 2024. 12

[22] Jerrit Eickhoff. **Polka: A differentiated deployment system for online and streamed games, meta-verses, and modifiable virtual environments**, March 2024. Accessed on June 24th, 2024. 14, 33

[23] Advanced School for Computing and Imaging. **DAS-6**. Accessed on July 31st, 2024. 21, 43

[24] Henri Bal, Dick Epema, Cees De Laat, Rob Van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. **A Medium-Scale Distributed System for computer science research: infrastructure for the long term**. *Computer*, **49**(5):5463, May 2016. 22

[25] Shengmei Liu, Atsuo Kuwahara, James J Scovell, and Mark Claypool. **The Effects of Frame Rate Variation on Game Player Quality of Experience**. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery. 24

[26] Henri Hein and Henri Hein. **Performance comparison using three different methods for joining lists**. 31

[27] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the International Conference on Performance Engineering, Mumbai, India, April, 2019*, 2019. 33

[28] **Voxelsim Evaluation Results**. 43

[29] Anaconda Inc. **Miniconda Installation**. Accessed on July 31st, 2024. 44

REFERENCES

Appendix A

Reproducibility

A.1 Abstract

Voxelsim is an addition to Opencraft 2 which adds a programmable environment on top of the already existing MVE. In this section, we go over relevant details of this project.

A.2 Artifact check-list (meta-information)

- **Program:** Voxelsim
- **Compilation:** Unity builder
- **Run-time environment:** Linux / Windows
- **Hardware:** DAS-6 (23) for experiments / Local for testing
- **Experiments:** see Table 5.1
- **How much disk space required (approximately)?:** 2GB for the build
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 4 + 3 + 3 hours
- **Publicly available?:** Yes, Project (10) and Results (28)

A.3 Description

A.3.1 How to access

The Opencraft 2 repository with the modifications made during this research can be found on GitHub (10). The experiments run and the raw data can also be found on GitHub (28). Using the Unity editor, a Linux/Windows build of the project can be made. Directories marked with "do not ship" or similar are not required for running the project.

A.3.2 Hardware dependencies

All experiments were run on DAS-6, so similar hardware would be needed to replicate these experiments.

A.3.3 Software dependencies

The project was implemented on a Windows 11 machine but evaluated exclusively and headlessly on a Linux system.

A.4 Installation

Experiments are designed to be run on the DAS-5/6 machines.

- Many of the scripts are dependent on your `student_id`, so change the value found in `config.cfg`
- Move your linux build of Opencraft to `/var/scratch/student_id/`
- Move the `das/` directory to `/home/student_id/`
- Follow the linux install instructions for miniconda (29), but change the install directory to be at `/var/scratch/student_id/`
- Navigate to the `das/` directory and create the conda environment with `conda env create -f environment.yml`
- You can manually start it with `conda activate experiment-env`
- Add the above line to the end of your `.bashrc` file so it is active on start-up for every node

A.5 Experiment workflow

There are 3 main experiment scripts: base, players and pareto. Each have different configuration options at the top of the file, with the values used in this thesis in comments. Wherever "Empty" terrain occurs below and in the repositories has been replaced with "Flat" in the explanations throughout the rest of this thesis as after conducting our experiments, we found that name to be more informative to the actual terrain configuration.

- Potential terrain types include "Empty", "1-Layer", "2-Layer", "3-Layer", "4-Layer", "5-Layer", "RollingHills"
- Active logic can either be "" or "-activeLogic"
- CircuitX / CircuitZ / Radius specify the dimensions of circuit generation

Nodes on DAS should be reserved via `preserve -np _ -t _` where `np` is the number of nodes and `t` is the duration they are reserved for. You may need to run `module load prun` beforehand, or you can add it to your `.bashrc` so it loads on startup. After that, you run `preserve -llist` to see which nodes you received and fill in `config.cfg` with the appropriate information.

Different experiments were run for different purposes and so loop over different parameters to run experiments in one go.

Logs from every client and the server are collected into `opencraft_logs/` directories. The internal application statistics are collected into `opencraft_stats/`. Each node's system statistics are collected into `system_logs/`.

After running the experiments, you can then run the `get_data.sh` script locally and it will pull all the recent runs to the `data/` directory. After doing so, you can then run `clear_data.sh` on DAS so you aren't pulling the same data again.

A.6 Evaluation and expected results

Our evaluation and results were discussed in Chapter 5. After running the experiment, there are a few steps to be able to use the data collected.

The `shared_config.py` file contains some parameters that most other scripts in the `scripts/` directory are dependent on. The main one to consider is the `experiment_name` as that dictates which directory other scripts will look at when managing data. Our end directory structure can be seen in Figure A.1, where `player_Dummy` already exists and raw results are moved to specific directories based on the prefix.

A.6.1 Data Sanitisation

The server statistics collected start from when the server is run, rather than when the experiment begins after clients are connected. The following 3 scripts can be run in order to format the data, sum it all together and finally give an averaged `csv` file that other plotters will use: `data_formatter.py`, `data_summer.py` and `data_averager.py`.

A.6.2 Plotting

All plots output to the plots directory. Plots were created and adapted for the specific experiments run, so may not be compatible for other variations. Certain scripts are designed for specific experiments while others work for all.

Figure A.1: Structure of data directory with experiment results.

`stacked_line_graph.py` and `stacked_bar_graph.py` work for any experiment and have an optional command-line argument `all` to include more statistics in the output. `fps_scatter.py` works for most experiments.

`bar_combined.py` and `fps_scatter_combined.py` were made specifically for players experiments and `pareto.py` for the pareto experiment.

Overhead

`resource_usage.py` is used to plot graphs related to overhead but `system_logs_extracter.py` should be run first to move the relevant `system_logs` to the `data/overhead/` directory.

Appendix B

Additional Experiments

Our evaluation chapter includes most of our experiments. However, in deciding which experiments were most interesting for our thesis, we also investigated another approach involving terrain generation. Additionally, we also ran our players experiment with a slightly different configuration.

B.1 Terrain Generation vs. Logic System

The original project, Opencraft 2 had one main feature which was its implementation of terrain generation. With our thesis, we added an additional feature with the logic system and believed comparing the two would be interesting. However, the only way to invoke terrain generation is by proximity to the player. As such, we created an experiment script that would start four clients moving in four different fixed directions. We ran this experiment with different configurations of terrain alongside some circuitry to see the performance but found that the terrain generation system did not have a great impact on overall performance and so did not pursue it further. Based on surface-level analysis, we believe that the player speed in reaching new terrain areas was not fast enough to overload the job-based multi-threaded approach taken in implementing this system.

B.2 Players and the Performance Overhead

In Section 5.4, we conducted a number of players focused experiment in which we compared the impact on the system of players across different terrain types. We then used this experiment's system level statistics in Section 5.5 to see the overhead of the logic system. However, in both of these sections, only the looping portion of the logic system was

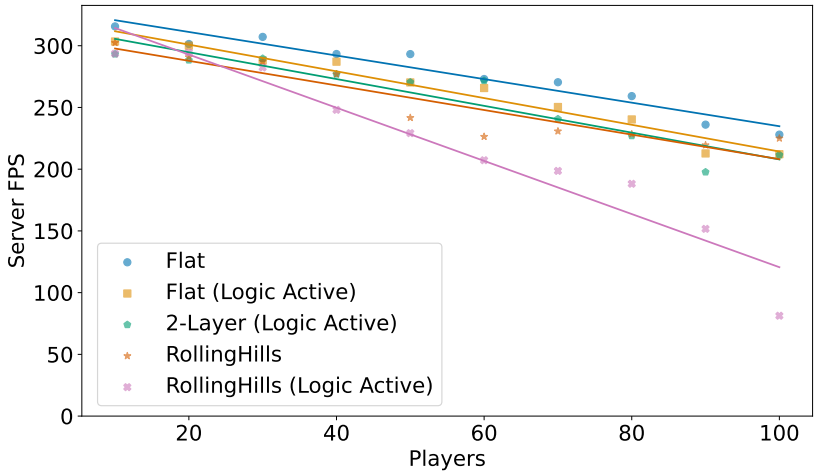


Figure B.1: Increasing players FPS comparison.

compared and not the overhead caused by having additional buffers, especially on system memory used.

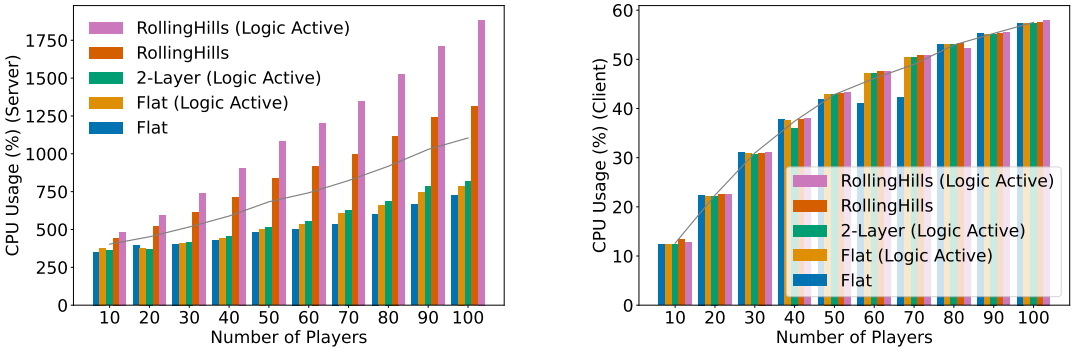
For this additional experiment, we instead compared the original implementation of Opencraft 2 without any modifications against Voxelsim, the results of which can be seen in similar figures to the ones used in the previous experiments. Here, any configurations labelled in the legend with "Logic Active" are Voxelsim and those without are the original project.

In Figure B.1, we can see that the performance of both `Flat` and `RollingHills` improve without the system, as expected because there is less processing required to add and manage additional buffers on each terrain area.

With CPU usage in Figure B.2, the averaged client usage remains about the same as before. The difference in CPU usage of the server is more drastic, with the run of `RollingHills` without logic requiring significantly (up to 30%) less CPU percentage.

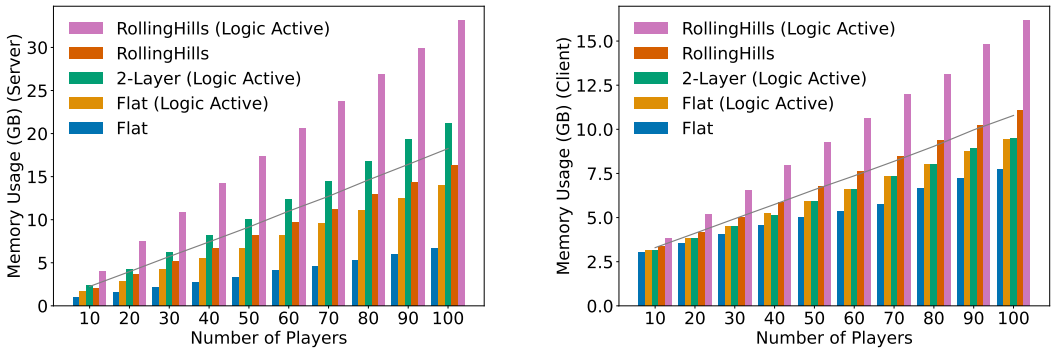
For memory usage in Figure B.3, both the server and clients use significantly less memory. This is because of the trade-off for performance we made with storing potentially unnecessary state in fixed-size buffers for block state, as described in Section 4.6.2. A similar reasoning is present for network bandwidth in Figure B.4 with a higher effect on the server as it needs to send out more information

However, while this version of the experiment is also interesting, we decided to keep the original in Section 5.4 as we were unable to implement a way to disable a buffer from being added and so believe this comparison isn't as indicative of the real-world system.



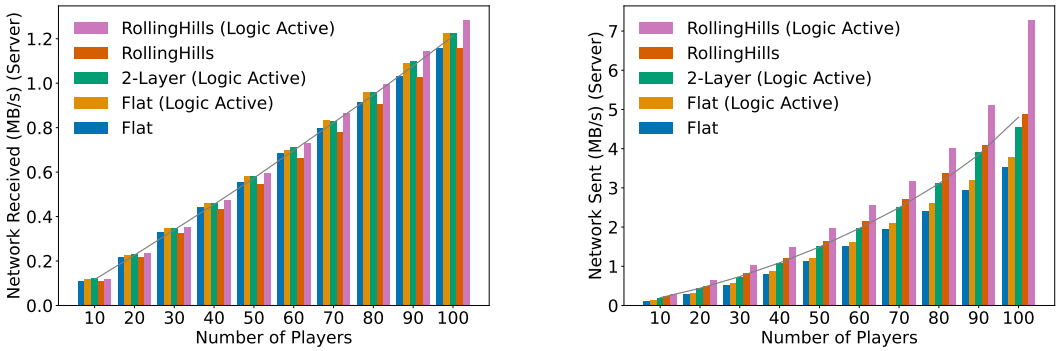
(a) Server. **(b)** Averaged Clients.

Figure B.2: CPU usage.



(a) Server. **(b)** Averaged Clients.

Figure B.3: Memory usage.



(a) Megabytes received by server every second. **(b)** Megabytes sent by server every second.

Figure B.4: Server Network Bandwidth.