Web and Desktop Native Voxel Game Engine

Final Report

For a Computer Science BSc.
Reported by Jo Null ############
This report has been anonymised for publication on my personal website, where my name is not publically available.

10257 words — ***Word count calculations are done excluding content of code listings, bibliographies and any appendices.***

1. Preface

This report is submitted as part requirement for the degree of Computer Science at the ##########. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I hereby give permission for a copy of this report to be loaned out to students in future years.

Signed, Jo Null ##########.

1.a. Acknowledgements

Thanks to Dr. ###############, who has been been a great supervisor. He has offered lots of encouragement, support, and enthusiasm in the development, structure, and documentation of my project. I would not have been able to write so extensively without him.

Thank you to the various students, members of faculty, and friends who have also supported me in my work.

2. Abstract

In this report, I build a game engine that can quickly render infinite worlds as a 3D grid of blocks. The engine runs on Windows, Linux, MacOS, and in web browsers. The engine is built from the ground up in a cross-platform GPU API: WebGPU. This allows me to build graphics functionality without the typical abstractions required to target conventional graphics APIs like DirectX and Vulkan. I build these features for my program: a 3D camera that can be controlled by keyboard and mouse, a sun that lights and casts shadows on the terrain, infinite world generation by a terrain “chunk” abstraction, and the ability to save and load changes from disk. All of these features work on the desktop and the browser.

Contents

3. Introduction

In this project, I aim to make a voxel game engine: A game engine that renders 3D cubes instead of meshes. It will aim to be performant on the web and on the desktop. The engine will run and be tested on modern desktop platforms: Windows 11, MacOS Sequoia, and common Linux distributions. It will also run in the Web using the experimental WebGPU API.

3.a. Motivation

Rust is a relatively new programming language. Rust was released in 2015 with a focus on correct, fast programs (Rust Foundation, 2024). Since developers needed to create new libraries for this language, it has been a greenfield site for emerging technologies, upcoming standards, and better re-implementations of existing technologies to thrive. Talented developers have put a lot of effort into making the building blocks for great games to stand tall upon. This is why I think it's a great language to make games with.

3.a.i. A case study: Minecraft Java Edition

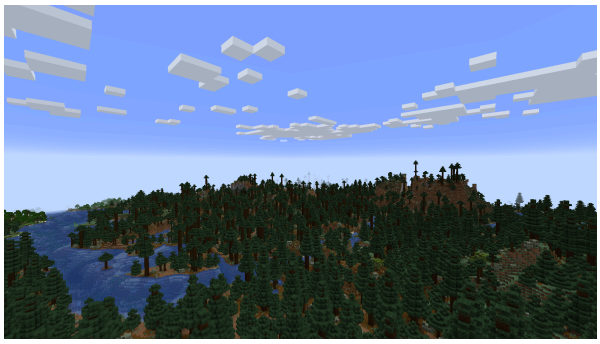


Figure 1: Minecraft: Java Edition with no shader

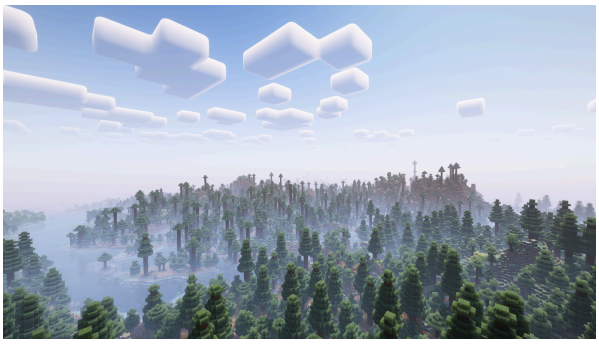


Figure 2: Minecraft: Java Edition with a community made shader, 'Complementary'

Minecraft: Java Edition (MC: JE) (Figure 1, Figure 2) is one of the most popular games that exist. Despite this, it has a very poor graphics implementation. There have been community efforts to ameliorate this, with modifications (mods) that optimise the rendering pipeline and add capabilities like complex shading (See Figure 2). These methods all fall short of solving a lot of issues however, due to a fundamental issue with the game's implementation: it uses Lightweight Java Game Library (LWJGL) to render graphics with Open Graphics Library (OpenGL). (Mojang, 2024)

OpenGL was released in 1992, and had its final release in 2017 (Michaud, 2017). Since then, efforts have been moved to creating and supporting modern graphics libraries. The OpenGL maintainers, Khronos Group, have moved onto Vulkan (Ware, 2023), while Apple and Microsoft develop Metal (Apple Inc, 2024) and DirectX (DX) (Nvidia, 2024). These libraries move work that was previously done on the Computer Processing Unit (CPU) to the Graphics Processing Unit (GPU), improving performance. (Iliev, 2023). This is called "driver overhead" (Imagination Technologies, 2018) and is one of the many reasons there have been efforts by large vendors, such as Apple, to deprecate OpenGL (Apple Inc, 2018).

The solution here is a graphics library that bridges these gaps and provides performance everywhere. This is where WebGPU – and the WGPU library I will be using – comes in.

3.a.ii. WebGPU

> Graphics Processing Units, or GPUs for short, have been essential in enabling rich rendering and computational applications in personal computing. WebGPU is an API that exposes the capabilities of GPU hardware for the Web. The API is designed from the ground up to efficiently map to (post-2014) native GPU APIs.
>
> — *WebGPU Working Draft* (World Wide Web Consortium, 2024)

WebGPU is a JavaScript (JS) Application Programming Interface (API) worked on by teams at Google, Mozilla, and Apple that lets a website use the GPU to render text. These vendors have a vested interest in maintaining working mappings as they all maintain browsers Chrome, Firefox, and Safari, respectively. While not a direct replacement, WebGPU surfaces much more direct access to the GPU than the other API: Web Graphics Library (WebGL). This means a programming utilising WebGL could replace all of its calls with WebGPU, and theoretically get much better performance out of it than it had previously (Trofymchuk, 2023).

WGPU is based on WebGPU, and gains the strength of mapping to all modern graphics APIs. (Rust Graphics Mages, 2024) However, it can also run outside of the browser, so you can write native desktop or mobile applications with it. Before this, developers would either have to rewrite code for multiple APIs or use inefficient OpenGL implementations. With WGPU, developers do not need to duplicate functionality to have good performance. I believe this will be the future of game engines as graphics code will be performant everywhere.

3.b. Relevance to the field

I believe that currently all voxel game engines suffer from age and poor rendering implementations (michael314, 2018; Anonymous User, 2019; fewturns, 2022; Agtrigormortis, 2023). They tend to utilise very old graphics libraries, and need to evolve with the times. Other features, like game mods, are using outdated methods and could be updated as well. This not only effects performance, but also security.

These have real impacts on users. With the use of WGPU, games that have previously been locked to one platform are free to be ran on many platforms (Rust Graphics Mages, 2024). This can allow players to connect and socialise together where they were previously unable.

> […] It's so much easier to play with friends when you don't have to worry about who owns what device or console. You might stop playing a game entirely without an online friend to play with. Making every friend accessible to you, even if they're playing on a different platform, is a huge improvement to the user experience.
>
> — Corey Davis (Epic Games, 2021)

Malicious mods cause real harm to users. Insecure mod loaders can allow mods to infect people's computers, steal data (Prospector Dev and Jai, 2024), hijack accounts to spread further (Alon Rabinovitz, 2023), and many other possible actions. The use of sandboxed environments like WebAssembly (WASM) could help stop the spread of malicious software in mod distribution (TechTarget, 2021) , with benefits in ergonomics for developers as they can choose from a myriad of languages (WebAssembly Developers, 2024).

3.c. Objectives

- Research methods of rendering voxels
- Develop a simple cube renderer
- Use the large ecosystem of Rust libraries to help implement game engine features
- Implement a renderer of a 3D cube grid
- Research and develop methods of procedural world generation
- Develop a way of saving and loading a procedural world
- Implement a playable character

3.c.i. Extensions

I have quite a lot of ideas to expand the engine and game. I will surely not be able to get to all of them, but these are some of my ideas:

- Implement interesting ways to interact with the game world
- Implement loader for game modifications compiled for WASM to be loaded and ran
- Implement a server for connecting together game clients
- Optimise the renderer for large draw distances
- Run in a web browser at performance comparable to native desktop

3.d. Resources required

As my work is mostly writing software on device, I will probably not need many resources.

- I will need a computer to work on.
- I will need capabilities for research, e.g. a library or internet connection.
- If developed, I may need server infrastructure for networking.

4. Background research

Despite voxel game engines being quite easy to implement, and having a lot of visual interest, there are not many popular voxel games. Instead, I will be focusing on notable implementations.

Minecraft is an obvious case, being one of the most popular games ever (Parrish, 2023). Minecraft is actually two games: Minecraft: Java Edition and Minecraft: Bedrock Edition. Minecraft: Java Edition (MC: JE) is the first, and Minecraft: Bedrock Edition (MC: BE) came after Mojang tried to make Minecraft cross platform (Mojang, 2017). As a product of it's time, it was forced to use different graphics libraries for different platforms.

Luanti (f.k.a. Minetest) is an open source voxel engine. It has been used notably for education: Google has used it for an educational hacking competition (Faessler, 2019), and has been used to teach kids subject such as Trigonometry, History and Construction (Luanti Developers, 2024).

4.a. A further study into Minecraft: Java Edition

As previously mentioned in the Motivations section, Minecraft: Java Edition (MC: JE) uses OpenGL for rendering (LWJGL Developers, 2024). While this has the benefit of allowing for rendering cross-platform, its semi-deprecated status (Apple Inc, 2018) and lacklustre performance compared to other libraries (Ware, 2023) means that it should not be replicated.

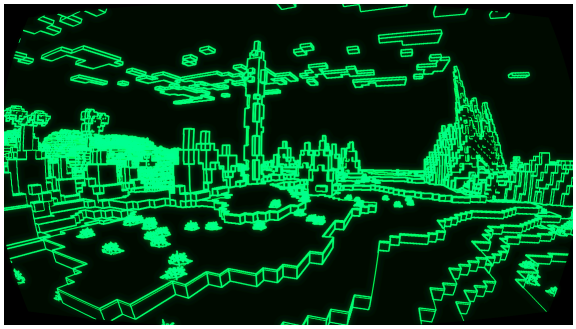


Figure 3: An out-there shader mod for MC: JE, 'Vector'

However, MC: JE does do some things right. It has a very interesting mod development scene. Since Java is a language that can be decompiled (JetBrains, 2024), the entire game can be understood and worked on by a dedicated community of developers.

Notably, there are community-made mod loaders, which hijack the game's launch sequence and inject custom code. These load packages of executable Java Bytecode, with some metadata, and perform operations to hook into the game's code while running. This uses reflection: a language feature that allows a program to operate on itself (McCluskey, 1998).

Mod loaders will usually abstract over this to save the developer some effort. They create custom APIs for interfacing with the game's classes. Recent innovations in mod loaders have introduced mixins, which allow developers to modify existing code in more ergonomic ways, and can reduce the chances of developers writing reflective code that clashes with another developer's (Mumfrey, 2017). This creates a *two-pronged approach* of making mods, where developers mostly work with abstracted APIs and fall back to mixins when the former falls short.

There is a potential issue of trust in the mod ecosystem. Due to how mod loaders are implemented, mods themselves are executable Java bytecode. While Java developers have attempted to make a sandbox, it is trivial to escape (Eauvidoum and disk noise, 2021) and people recommend against using it (rzwitserloot, 2021; RandomName8, 2023). Because of this, people have to trust mod developers to not include viruses or other malicious code. The community has somewhat of a solution to this: mod repositories. These are websites that check and collate community mods. These sites centralise trust, and review every mod submitted. However, processes for preventing malware at the moment seem to be mostly *reactive* rather than *proactive* (Alon Rabinovitz, 2023; Prospector Dev and Jai, 2024).

In 2023, a supply-chain attack called Mad Gadget (Justine Tunney, 2017) affected a large swath of Minecraft mods (Serializationisbad Developers, 2023). In 2024, a large mod repository, Curseforge, suffered an attack where malicious developers uploaded malware that stole other mod developer's credentials (Alon Rabinovitz, 2023).

4.b. Bedrock: The Other Minecraft

Minecraft: Bedrock Edition (MC: BE) is the sibling to MC: JE. It is mainly used as the edition for mobile devices and consoles, but it has a version for Windows. For most targets, it uses OpenGL for Embedded Systems (OpenGL ES), a simplified version of OpenGL, but for Windows it notably uses DirectX. (Mojang, 2019) This means that it can render with much more performance, and use complex ray-tracing shaders. This comes at the cost of only working on Windows, however.

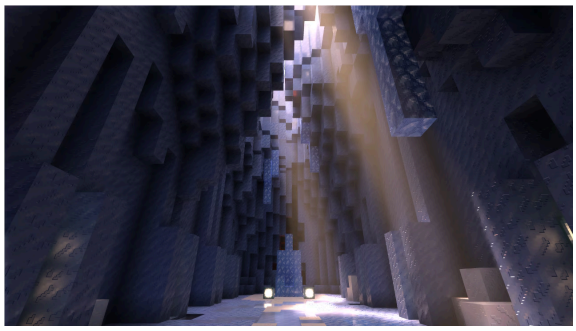


Figure 4: Minecraft: Bedrock Edition with the NVIDIA RTX add-on. Image courtesy of Mojang (Mojang, 2019)

This edition of Minecraft has a different mod loader system too. The mods are called 'add-on's and are officially supported by Mojang. These mods can contain 'behaviour packs' and 'resource packs' which can both modify the game in different ways (iconicNurdle, 2023a).

Unlike other mods, which are usually written in the same language as the game, or a scripting language like Lua, MC: BE 'add-on's are written in JavaScript Object Notation (JSON). This markup is then parsed by the games mod loader. Custom behaviour can also be defined in JS, but is not required for a mod. This presents a more traditional mod creation experience (iconicNurdle, 2023b). MC: BE contains an in-game download manager where players can purchase and download mods with digital currency called 'Minecraft Marketplace'.

The game uses a QuickJS variant to run JS code (Mojang, 2024). This could be secure, but care has to be taken to make sure any security holes are not created. Removing or reworking unneeded APIs would make sure the *attack surface* is small, which would increase security (Bono *et al.*, 2009). If a security hole is created, a sandbox could help prevent the hole from being exploited to run malicious code. Using an interpreted language makes code easier to audit as it is human-readable, as opposed to a language which compiles to machine bytecode.

4.c. Luanti

Figure 5: Luanti, running the community made content: 'Nonsensical Skyblock'

Figure 6: Luanti, now running the community made content: 'VoxeLibre'

Luanti is an open source voxel game engine. It uses Simple DirectMedia Layer (SDL) 2 for rendering (Luanti Developers, 2024). This is a library that can be used for games and other rendered or interactive software (SDL2 Developers, 2024). SDL 2 is quite old. Version 2.0 was released in 2013 (Sam Lantinga, 2014). Since then, it has not majorly upgraded its graphics libraries from OpenGL and Direct3D. The SDL team are working on 3.0, which will support DirectX 12, Vulkan and Metal libraries (Ethan Lee, 2024).

What differentiates Luanti from other voxel games is a sanctioned modding API, with an in-game mod browser. Developers can make 'packages' in a language called Lua (Ward, 2024). One thing Luanti lacks in its mod implementation is the lack of a lower-level fallback like MC: JE and MC: BE. This means that mods are less complex and therefore cannot deviate as much from the base experience. However, this does have an advantage as it makes it a lot less complex to maintain mods mixins make it harder to update a mod as internal code does not have the same stability guarantees as a public API (quat, 2023).

The choice of Lua for mods is a good choice for security (Bono *et al.*, 2009), and it provides a sandbox for scripts to run in. Luanti mods are sandboxed by default, but a mod can request that the sandbox is lifted (Ward, 2024). This means that in theory, malicious code could be distributed on Luanti mod repositories, though incidents have not been reported of this happening. This is maybe due to sandbox lifting being obvious in code review.

5. Requirements

My project will be a large monolithic program. It will be implemented in Rust, ideally with one "crate" (a Rust program or library). If needed for technical reasons, the project will be turned into a workspace with multiple crates.

Components are ordered in a *rough* order of implementation, so later components should be implemented after earlier ones.

Any library in `monospace text` is a Rust library. You can find these libraries on the internet at https://crates.io.

5.a. Renderer

This part of the program is the main focus, and will be first implemented. The renderer will be implemented in `wgpu`, and will render a voxel grid. It may render voxels not aligned in a grid, for example for a player entity. Ideally, this component will be very optimised, but at this stage of development it is unclear what techniques will be used for this.

Since the terrain should be theoretically infinite, the renderer should stop rendering past a certain range around the camera. One could call this the 'render distance'. The maximum render distance that doesn't push the system past the minimum Frames per Second (FPS) will be a good benchmark for measuring how effective the renderer is. Ideally, the maximum render distance *should* be greater than other implementations.

5.a.i. Lights and shadows

In order for the scene feel correct, I need to have good lighting and shadows. This will help convey things like depth to the viewer (Hess, 1961) and make the scene look more realistic. A good lighting implementation will mimic the properties of real objects, and not make any mistakes that would make the player lose their grounding in the world. A good shadow implementation will display crisp, accurate shadows cast by a sun, point lights in the world, or any other light.

Though, techniques like real-time ray-tracing are out of scope of this project. Though ray-tracing capable machines *seem* to be in the hands of most people who use game distribution platform Steam (Laird, 2023), it would not be advisable to release a game engine with only ray-traced lighting (Archer, 2024). Further complicating this: Writing ray-tracing software, while an interesting problem, would be a large undertaking (Shirley, Black and Hollasch, 2024) unsupported by `wgpu` (Ignaci, 2020).

5.b. Terrain generator

This will use noise algorithms to generate a cube grid. This will be rendered as voxels. Layering of different algorithms will be used to achieve landscapes like rivers, plains, and mountains. Care should be taken to ensure that an input seed always produces the same generated result.

This stage will require figuring out how to store the map data. This will probably be in a rudimentary form until the save manager interfaces with it. It is important that map data is somewhat global, as most of the engine will interface with it. I will likely at some point implement a write-lock system on the map data and make it "global".

5.c. Entity component system

Since the engine will have quite a few moving parts, an Entity Component System (ECS) will be important to manage interfacing between systems. An ECS is important as it will allow for abstractions over complexity, easing faster development. ECSes are usually more performant than traditional object oriented systems, proven by Unity's adoption of ECS massively increasing performance (Antich, 2023). This is because it can make use of multi-threading (Anderson, 2020; Amethyst Foundation, 2024). The design of ECSes are much better suited for Rust development as well (West, 2018).

There exist many approaches and libraries for this. There are a few options, `bevy_ecs`, `apecs`, `legion`, and others. Which will be the most suitable will have to be evaluated.

5.d. Save manager

This will read the contents of the cube grid, encode it in a binary format, then store it in the user's application data directory.

Since Rust has no built in capability to encode arbitrary structure, the library `bincode` could be used, which converts Rust data structures to binary data. Manually creating a binary format could also work for this, and may be preferable since Rust does not have a stable Application Binary Interface (ABI).

The manager will also have capabilities to load the saved file. This will decode the data using a similar method used to encode. It is likely that

The manager may automatically save at intervals (Auto-save functionality), and when the user requests.

Since the specific application data directory will depend per operating system, a library could be used, and testing should be done per supported OS.

5.e. Camera and character controller

This component will let the user use input systems to control an in-world character. This will let them interact with the world.

Keyboard and mouse input reading comes with the most used Rust windowing library: `winit`. However, game-pad support is more complex as it requires interfacing with other OS APIs. I have found a platform-agnostic library for this: `gilrs`. This library should support most game-pad scenarios, but workarounds may be needed. Notably, idle inhibition (stopping the system from entering a locked or sleep state) may need workarounds, to stop the system from going to sleep when the user is actively inputting controls. This will require testing.

A rudimentary controller may be implemented early in development to view generated terrain, but without a focus on user experience.

5.f. Plugin Loader

This component will load and evaluate plugins made in WASM. This will let any user implement an extension to the project, in any language that can compile to WASM, like Rust, C, C++, C#, AssemblyScript, or Java (Wasmer Inc., 2024).

This will require adding a WASM runtime. The most likely candidate is `wasmer`. I will then have to make a public API for plugins to use to interface with the engine. This will most likely let plugins do things such as: adding voxel block types, influencing terrain generation, and implementing custom functionality for blocks/items. This will also require that I guarantee a stable ABI, as this does not come with the language. This may require the use of a library like `stable-abi` (Manero, 2021).

The plugin loader should be secure. A WebAssembly runtime should contain its programs, and not allow them to access the surrounding system. `wasmer` claims to be secure and to take efforts to patch security bugs in their runtime. This should be hopefully sufficient to satisfy this requirement (Wasmer Inc., 2024).

For the scope of this project, demonstrating a plugin made in Rust and one other language would satisfy this requirement well.

5.g. Web support

Though supported by WGPU, web support is hard to implement as there are many factors (e.g. using WebGL or WebGPU, controller input, storage.) These are all issues that have to be dealt with differently on the web. To satisfy this requirement, demonstration that all implemented components work inside at least two different web browsers would be sufficient. WGPU should provide a WebGL fallback for browsers that do not currently support WebGPU. This is important as WebGPU is still experimental and support is patchy.

5.h. Multi-player support through networking

Almost every ‘sandbox’ game – games with no goal except to build and explore – has multiplayer networking. This lets users create and explore together. This would require either a peer-to-peer or a server-client infrastructure, and either would work well for this requirement. Players should be able to network together on clients running on different operating systems, desktop or web.

This requirement would be quite complex to implement. Potentially, the game would be separated into two binaries (server and client), and a shared library would hold most of the game logic. Rust libraries could be used to simplify implementation of server to client communications, e.g. `netcode-rs` or `renetcode`. A peer to peer implementation would only require one binary but makes implementation more complex. It could use `libp2p` or `matchbox_signaling` to simplify implementation. A peer to peer implementation may also need a server to establish and relay connections between peers.

6. Design and Production

My project was produced through Agile methodologies, a set of practices for efficient software development. Agile was created through the Agile Manifesto (Beck *et al.*, 2001) by a committee of project leads that wanted to reduce the iteration time and bureaucratic overhead in software development. There are many different Agile methodologies, but the most popular are Kanban, Scrum and Lean (UK Government, 2016).

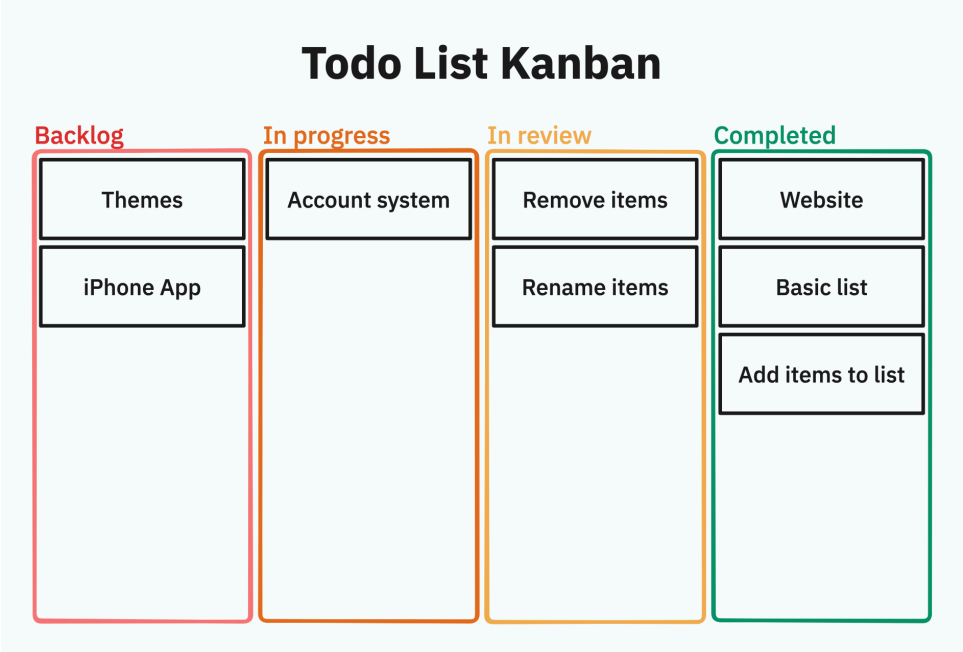


Figure 7: Example Kanban boards for managing tasks in development of “to-do list” software. There are four boards: “backlog”, “in progress”, “in review”, and “completed”. As tasks move to rightward boards, a task becomes more complete.

The method variant I used is Kanban, where projects are broken up into tasks and categorised based on completion, as shown in Figure 7 (Radigan, 2025). Kanban boards allow for easy visualisation of tasks in progress and help minimise doing too many things at the same time. GitHub added support for Kanban boards in 2016 (Botsford, 2017), and this allows for Agile methods to be performed next to your source control. I used this feature for my Kanban board (Figure 9). GitHub’s boards let you mark tasks with priority tags and complexity-size tags (Figure 8). This helps you weigh which tasks you want to tackle.

Figure 8: Screenshot of GitHub’s Kanban task management view. You can set markers on a task for priority, “size”, and other metrics.

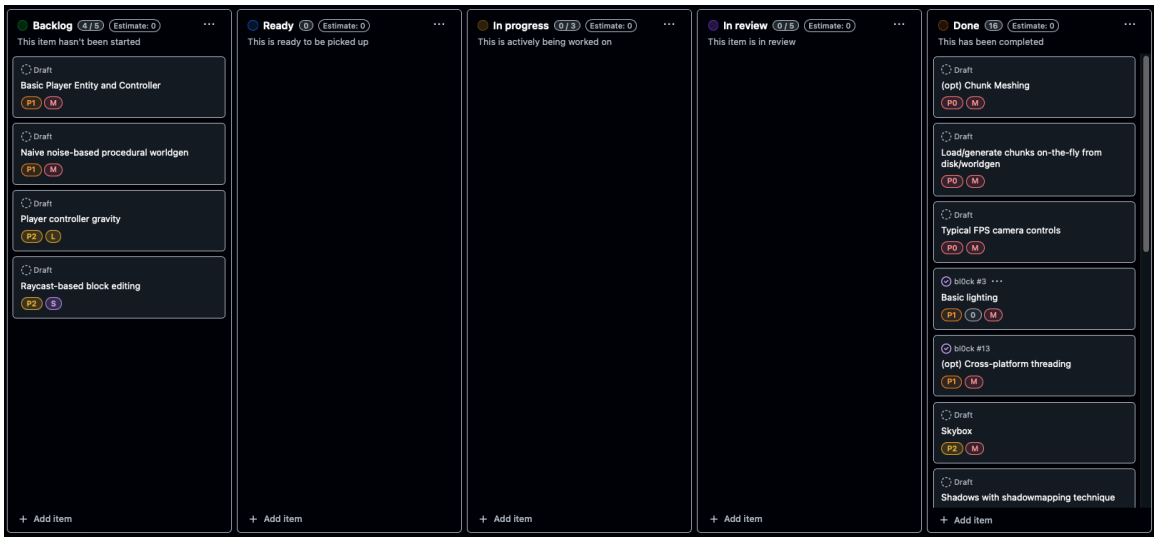


Figure 9: GitHub screenshot of Kanban boards of my project before submission. My project utilises an extra board, “Ready” for tasks to complete in the current sprint which are not being worked on yet. There are priority and size labels on each task.

In order to satisfy the Agile requirement of regularly taking feedback from the customer, I performed two-week sprints. Sprints are periods of time where you develop software. At the end of a sprint, a team member will present the work done to the customer to get feedback. Involving the customer is a key aspect of Agile (Beck *et al.*, 2001). As I did not have a strict “customer” in my project, I reported to my supervisor at the end of sprints. When I completed a task, I moved it from the “In progress” board to the “In review” board. If my supervisor was satisfied with the completion of my task, it was moved to the “Completed” board. We then moved new items from the backlog to the “Ready” board, starting the cycle again.

GitHub’s project management also let me construct a GANTT chart out of my Kanban tasks, like in Figure 10. GANTT charts plot your tasks in respect to time, and can help you find links between tasks. Critical paths — series of dependent tasks that would slow project development down if disrupted — can be identified by lines of tasks that start when the previous ends (Atlassian, no date).

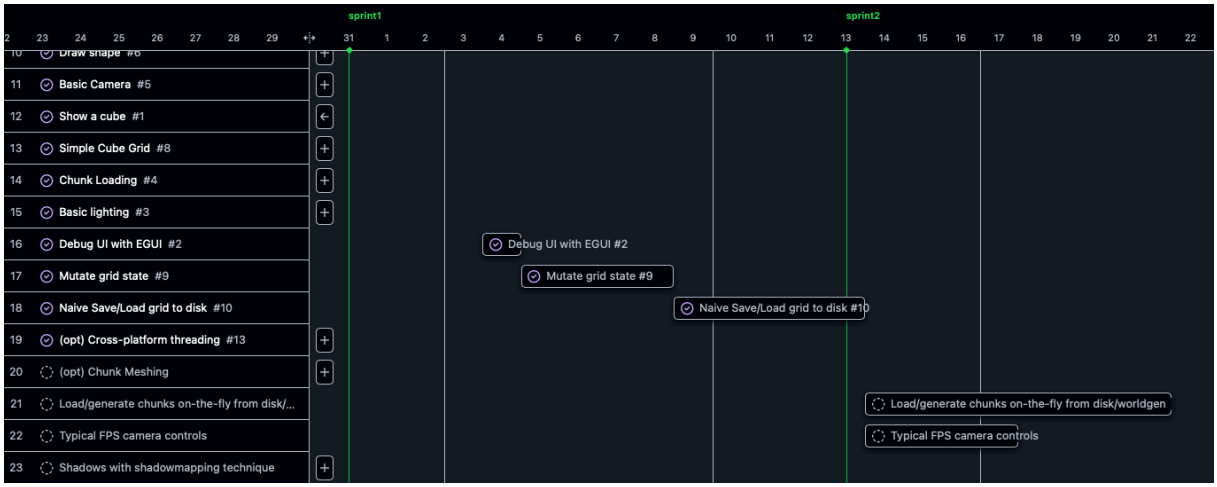


Figure 10: Screenshot of GitHub’s GANTT viewer. It shows tasks flowing into each other, and sprint deadlines. This lets me have a visual history of all of my tasks, and track if I am reaching deadlines or not.

6.a. Initial Architecture and Development

Most of my early work is referenced from the tutorial website *Learn WGPU* (Hansen, 2024). This series of tutorials helps a developer set up the architecture that is required to draw images to the screen. It also helps with axillary libraries, like:

- `winit` for instantiating an OS window,
- `pollster` for interacting with asynchronous code within a synchronous context,
- `wasm-bindgen` for generating bindings from my WebAssembly code to JavaScript,
- `web-sys` for facilitating interaction to JavaScript APIs, and
- `bytemuck` for turning my data structures into formats that GPUs can understand.

These libraries were essential for my development. I did use other libraries in the tutorial, but these were mainly for ease of development rather than giving me more capabilities.

One thing I had to be mindful of in the architecture is that web and desktop environments have different requirements. On desktop systems the computer can easily execute asynchronous code by either running it synchronously or using an executor. I chose to run these synchronously. On the web, I had to use a library called `wasm-bindgen-futures` to turn a Rust `Future` into a JavaScript `Promise`. This promise is ran "in the background" by the browser's asynchronous executor (MDN, 2025a).

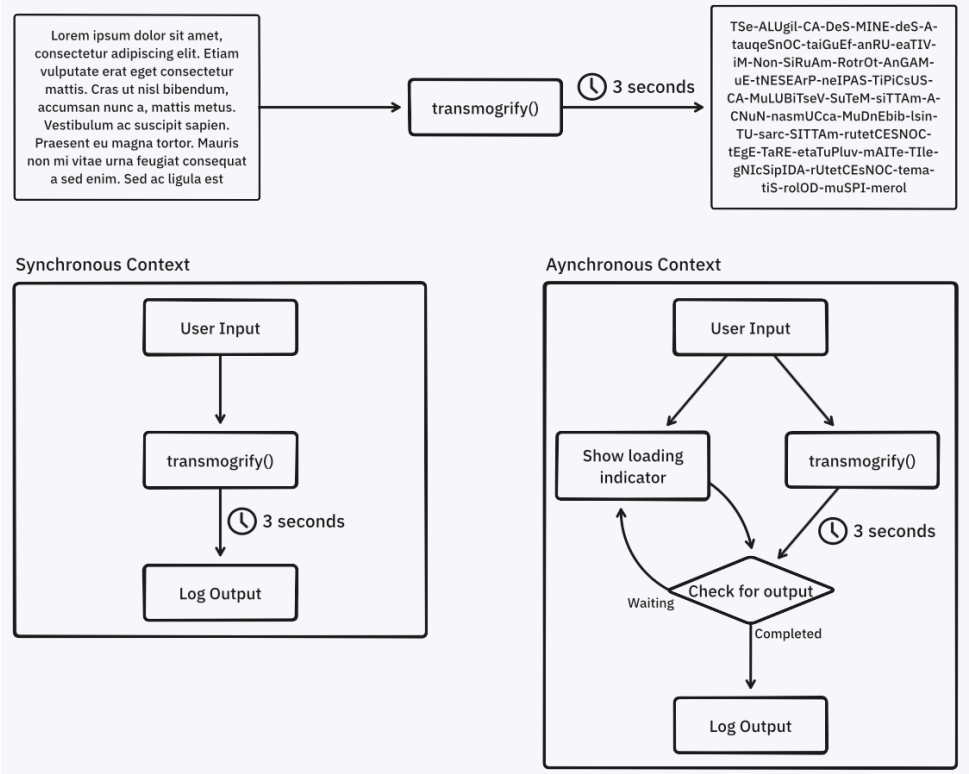


Figure 11: A diagram of a hypothetical function that takes time to finish computing. Two flowcharts are shown, one in a synchronous context, and another in an asynchronous context. In a synchronous context, execution is paused until the function is finished before continuing. In an asynchronous context, the flowchart lets the function resolve while it shows an indicator to the user. It waits in a loop for the function to resolve, then continues execution.

I ran into issues in the web because the promises can't return any values to the Rust code directly. If I want to return a value, I have to use *message passing* (Rust Foundation, 2024). This involved creating a **transmitter** and **receiver**, and then holding onto a receiver while giving the transmitter to the thread. This allowed for communication where it would otherwise be impossible[1].

[1] If you are still struggling to understand this concept, imagine giving the other function a *radio transmitter* before sending it away on another thread! It can talk to the function without "physical contact" through sending messages.

6.a.i. Complexity

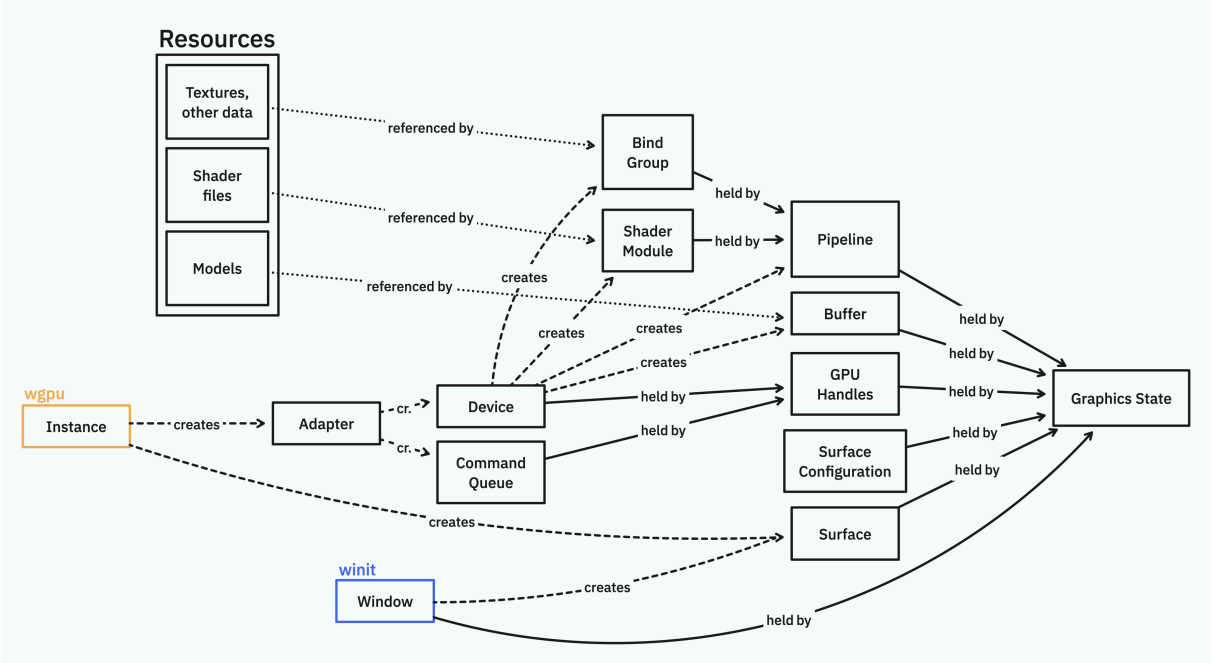


Figure 12: A diagram of what happens in the `State::new()` function. It starts with a GPU `Instance` and a `Window`, and ends with a large `State` structure.

All GPU rendering is centred around render pipelines. They contain all the tools one has to make and alter 2D images (Akenine-Möller *et al.*, 2018). In the context of WebGPU, render pipelines are steps that the GPU takes to turn a set of inputs into a set of outputs (World Wide Web Consortium, 2024). Everything one can do on the GPU has to go through a pipeline. Most of the building of the graphics state in Figure 12 is centred around constructing a pipeline.

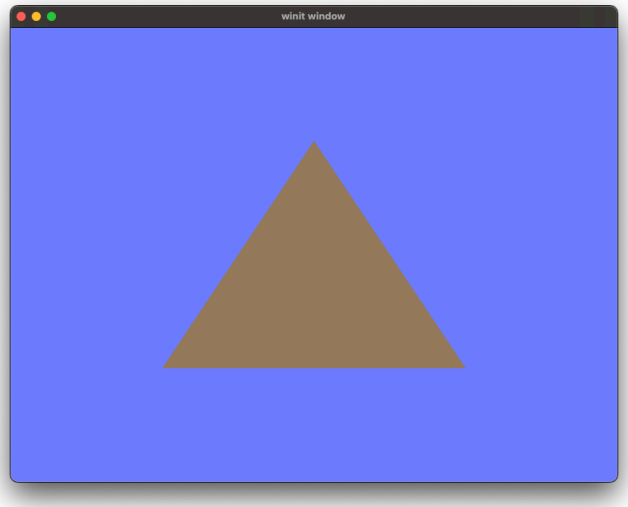


Figure 13: A macOS window. It shows a brown triangle with a blue background. This is one of the most basic programs one can make with WGPU.

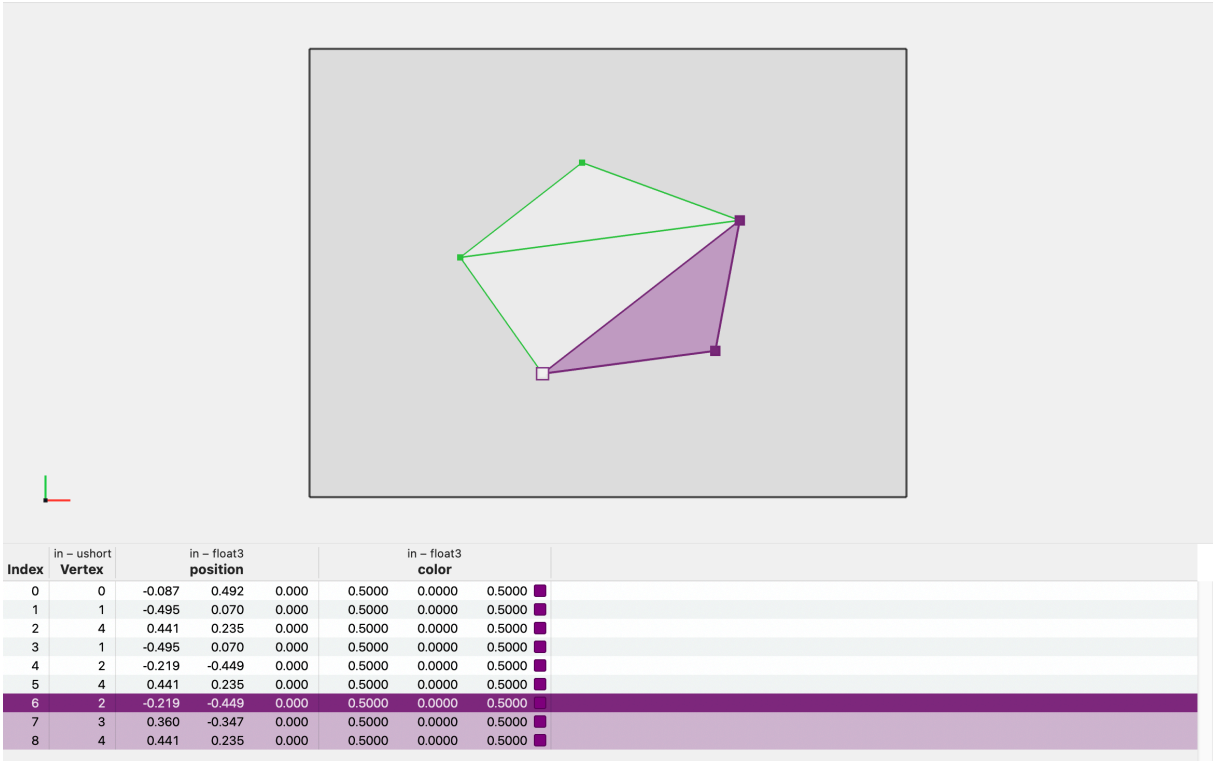
To render the triangle shown in Figure 13, I set up a pipeline. In the pipeline, I pass in a shader. I also tell the pipeline in which ways it should render triangles, with back face culling, triangle filling, and which side is the front. After instantiating a graphics state, the program enters the render loop. On each frame, it constructs a render pass, sets it to use the pipeline, tells it to render three vertices, then finally submits commands to the *command queue* and present the surface's texture.

I can start adding complexity to the program now, like rendering a different shape.

```rust
const VERTICES: &[Vertex] = &[
    Vertex { position: [-0.0868241, 0.49240386, 0.0], color: [0.5, 0.0, 0.5] }, // 0
    Vertex { position: [-0.49513406, 0.06958647, 0.0], color: [0.5, 0.0, 0.5] }, // 1
    Vertex { position: [-0.21918549, -0.44939706, 0.0], color: [0.5, 0.0, 0.5] }, // 2
    Vertex { position: [0.35966998, -0.3473291, 0.0], color: [0.5, 0.0, 0.5] }, // 3
    Vertex { position: [0.44147372, 0.2347359, 0.0], color: [0.5, 0.0, 0.5] }, // 4
];

const INDICES: &[u16] = &[
    0, 1, 4, // Triangle 0
    1, 2, 4, // Triangle 1
    2, 3, 4, // Triangle 2
];
```

Listing 1: Defining a set of constants for vertices and indices in Rust. The vertices contain positions and colours. The indices contain 3 sets of 3 index values. The indices point to the vertices and let me reuse vertices.



Index	in – ushort Vertex	in – float3 position			in – float3 color		
0	0	-0.087	0.492	0.000	0.5000	0.0000	0.5000
1	1	-0.495	0.070	0.000	0.5000	0.0000	0.5000
2	4	0.441	0.235	0.000	0.5000	0.0000	0.5000
3	1	-0.495	0.070	0.000	0.5000	0.0000	0.5000
4	2	-0.219	-0.449	0.000	0.5000	0.0000	0.5000
5	4	0.441	0.235	0.000	0.5000	0.0000	0.5000
6	2	-0.219	-0.449	0.000	0.5000	0.0000	0.5000
7	3	0.360	-0.347	0.000	0.5000	0.0000	0.5000
8	4	0.441	0.235	0.000	0.5000	0.0000	0.5000

Figure 14: A pentagon being rendered and debugged in Xcode. The debugger shows how the 9 indices defined in Listing 1 point to the 5 vertices. It also demonstrates that index groups of three form one triangle.

In order to send data to the GPU, you have to use buffers. In WebGPU, buffers are used in two ways: **Vertex** and **Index** buffers, that let you push *geometry* to the GPU, and **Uniform** buffers, that let you push *arbitrary data* to the GPU. In the case of Figure 14 and Listing 1, you can push vertices and indices directly in the render pass. If you want to use any data in a shader, you have to go through one of these two pathways.

For uniform buffers, you need to specify exactly how and what they are before you can push them. This is where bind groups are used in WebGPU. In a bind group, you need to specify how many items you are binding, what types those items are, and at what stages those items will be visible to shaders. In Figure 15, you bind a texture to apply to the pentagon. You also need to bind a sampler of that texture. You need to do this to specify the behaviour of what sampling a texture does. This is because sampling is a non-trivial operation, and edge cases need to be defined[2].

[2] There is a lot of configuration you can set with texture samplers (World Wide Web Consortium, 2024). For example:
- What happens if you try to access out of bounds?
  - You can either clamp the value to the texture edge, repeat the texture, repeat and mirror the texture, or clamp with a border.
- What does sampling do if the texture is grown or shrink?
  - You can define algorithms to use in each case, either “nearest neighbour” or linear interpolation.

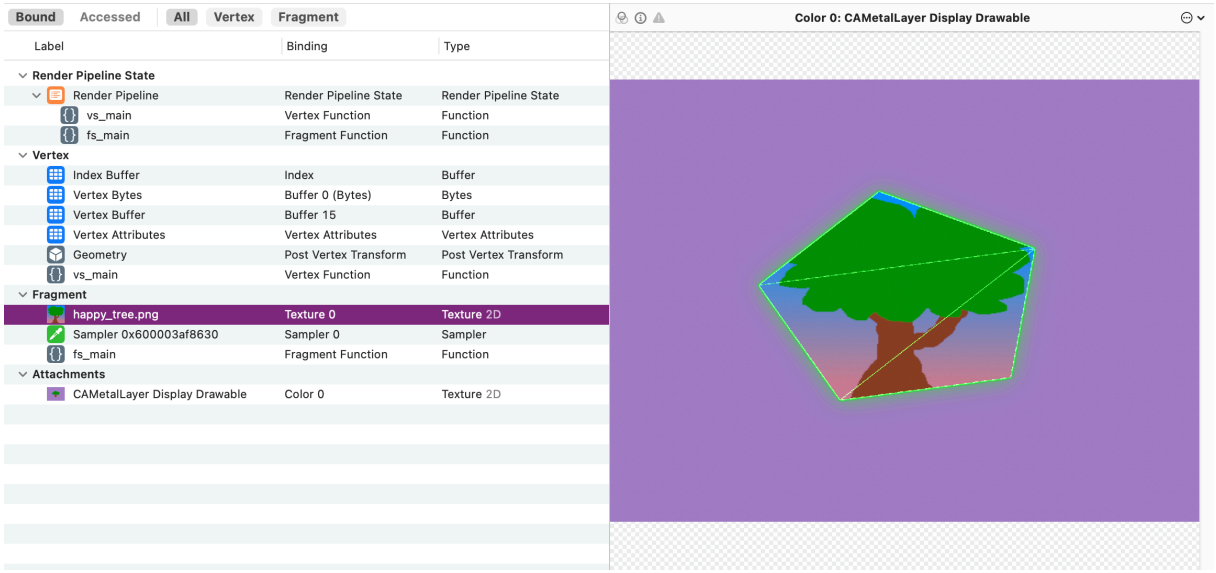


Figure 15: The pentagon from Figure 14, with a texture of a tree applied. The uniform data bound to the shader can be seen in the "Fragment" section. You can see that a texture and a sampler were bound.

6.a.ii. 3D Camera

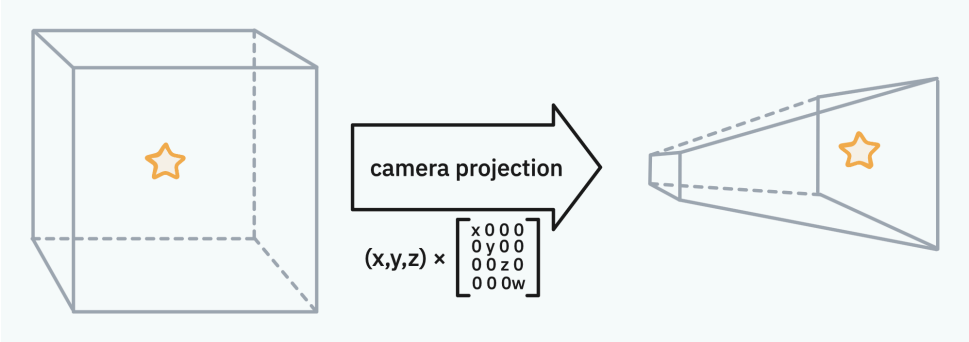


Figure 16: Demonstration of how I use a perspective projection matrix. By multiplying each vertex with the matrix, it effectively transforms the space in which the vertices reside in (Akenine-Möller *et al.*, 2018).

Perspective *projection matrices* are 4×4 matrices that I use to transform points in a space, as demonstrated in Figure 16. These data structures are key in implementing a camera for the game world.

To do vector maths, I use the Rust library `glam`. I use this library to build a projection matrix for the camera. In Figure 17, I specify the parameters to build a *view frustum*, which is a pyramid-shaped space that contains any geometry shown by the camera (Akenine-Möller *et al.*, 2018). This model is a simplified version of a camera.

To do user input for the camera, I have to talk to the *window*. This means I have to get inputs from `winit`. In the window's event loop, I add a hook to listen to any inputs the window has captured. Then, I can match on keyboard inputs. On a left or right arrow key input, the program can orbit the camera clockwise and anti-clockwise around the origin. On an up or down arrow key input, it can move closer or further from the origin. I have to make sure that I make these changes before rendering, to avoid adding a frame of lag to the camera controls.

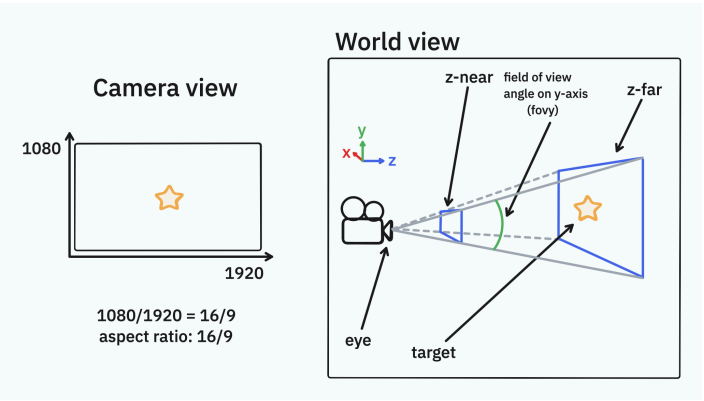


Figure 17: Diagram of what information is required for specifying a camera's frustum. The eye is the point where the camera is positioned in 3D space. The target is where the camera is pointed towards. The z-near and z-far specify where the pyramid is sliced on the z-axis. The `fovy` specifies how wide the pyramid is on the y-axis. `glam` uses this angle, with the aspect ratio, to figure out the x axis' `fov`.

6.a.iii. Instancing

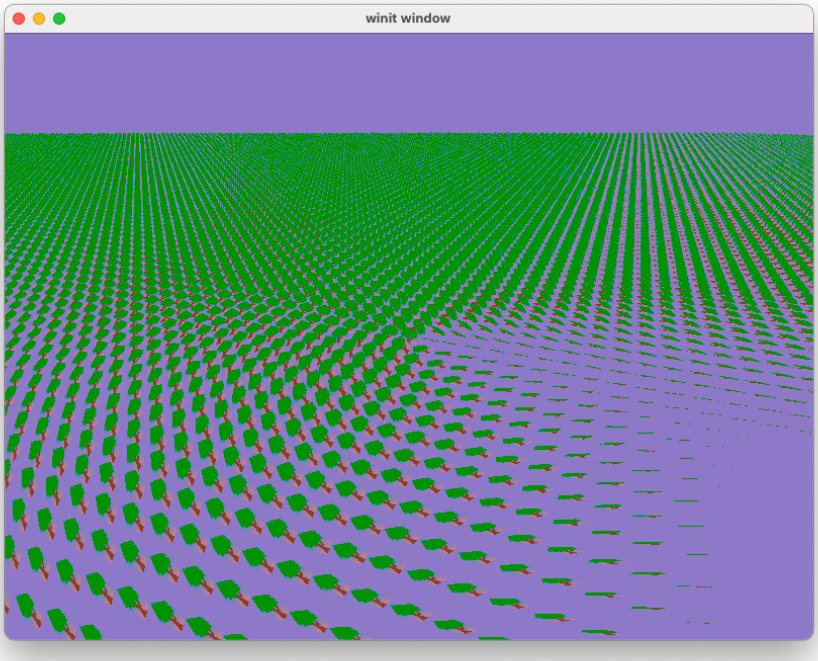


Figure 18: A “forest” of tree images on pentagons.

Instancing is a feature of GPUs where you can draw many objects in one draw command, and give varying inputs to each instance of the object (Akenine-Möller *et al.*, 2018). As you can see in Figure 18, the pentagon objects are varied in rotation and position. In an instanced draw, I can pass in the instance data as a vertex buffer. This means each object will have the appropriate data when running its vertices through shaders.

In Listing 2, I have a basic procedure that creates this instance data. In order to use this representation in a shader, I turn the position and rotation into 4×4 matrices, and multiply them together. This matrix representation lets me *compose* these operations into one. This simplifies the shader code, as I can add this matrix into the transform steps done for the camera in Section 6.a.ii.

```
let instances = itertools::iproduct!(0..NUM_INSTANCES_PER_ROW, 0..NUM_INSTANCES_PER_ROW)
    .map(move |(x, z)| {
        let position = vec3(x as f32, 0.0, z as f32) - INSTANCE_DISPLACEMENT;

        // this is needed so an object at (0, 0, 0) won't get scaled to zero
        // as Quaternions can affect scale if they're not created correctly
        let rotation = match position.try_normalize() {
            Some(position) => Quat::from_axis_angle(position, 45.0),
            None => Quat::from_axis_angle(Vec3::Z, 0.0),
        };

        Instance { position, rotation }
    })
    .collect::<Vec<_>>();
```

Listing 2: Procedure that creates the data required for varying object rotation and position. I need to turn the rotational data into a quaternion, a 4 dimensional data type for representing rotation.

You may notice that some instances are drawing over other instances. To solve this, I have to use a **depth buffer**. A depth buffer is a type of image where pixels get darker when a point is closer to the *eye* of the camera. I use this to decide when to draw triangles over other triangles – setting a *draw order* (Akenine-Möller *et al.*, 2018). You can see a depth buffer of the previous scene in Figure 19. I pass the pipeline a blank texture, and the pipeline writes to it, then reads from it for depth comparisons. This use of textures in this way is a feature in WebGPU called a “stencil” (Ammann, 2022).

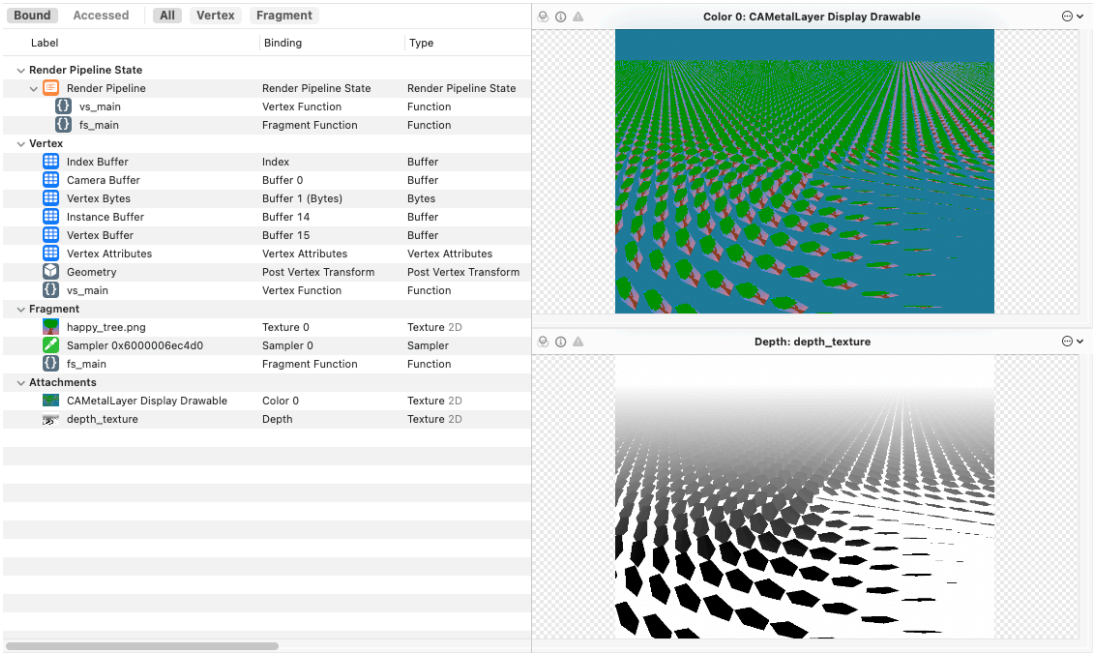


Figure 19: A screenshot of Xcode's graphics debugger. You can see the drawn image next to the depth buffer. A darker pixel means a closer point in space. A pure white pixel means the maximal distance from the eye of the camera.

6.a.iv. Model Loading

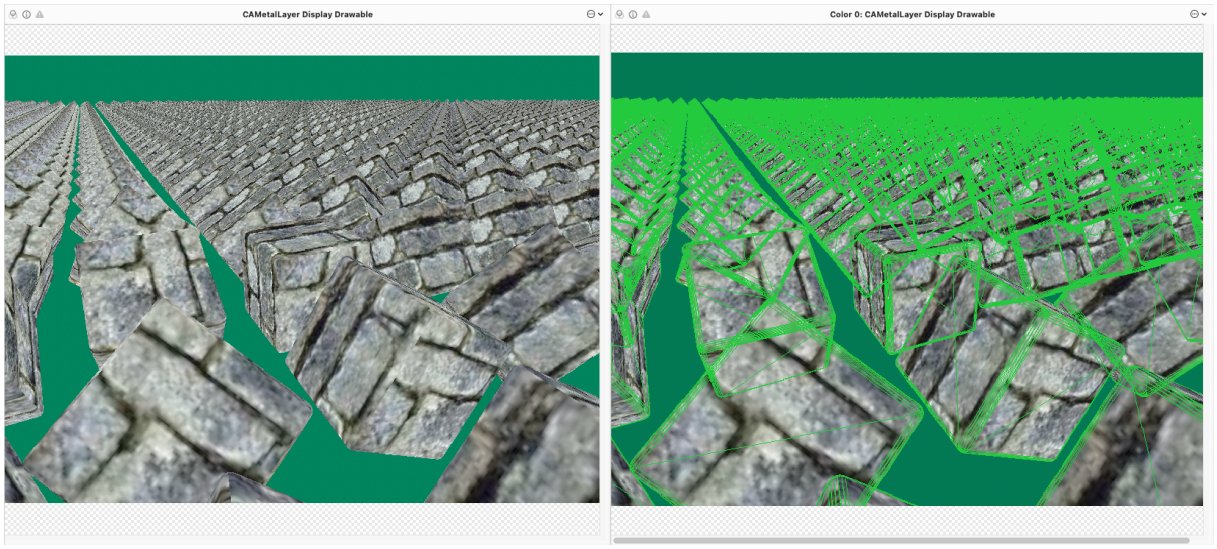


Figure 20: A screenshot of Xcode's graphics debugger. On the left, you can see a large number of stone blocks in a grid. On the right, you see the wireframes – the lines showing triangle edges on models – of the stone blocks.

The Wavefront `.obj` file format is rather simple: It is essentially a plain text list of numbers, like Listing 1. You can load a model with the rust library `tobj`. Since the program has to live on the desktop and the browser, I need to make some abstractions in loading models and textures. I make a resources module in the program for these file loading functions.

On desktop, the program can use the operating system to read a file. On the web, it have to use a library to make a HTTP GET request to the hosted files. After reading in a model, it can use `tobj` to parse the data for me. The object file refers to a texture, which the program can then fetch from the resource loading abstraction defined earlier.

6.a.v. Blinn-Phong lighting

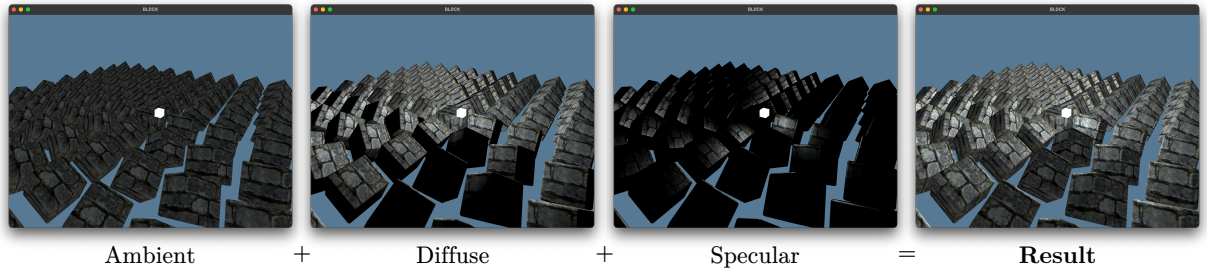


Figure 21: Demonstration of the three parts that make up Phong (and by extension, Blinn-Phong) lighting. You sum the values of each step to get a resultant value on every fragment.

Lighting is key to grounding a scene [[cite]], and thus is the last of the initial production. The Blinn-Phong lighting model (Blinn, 1977) is a simple method for realistic-looking lighting, as seen in Figure 21. The ambient layer is light that is scattered indirectly to all objects in the world. Think about how at night, you can still see things because some sunlight is still reflected (Hansen, 2024).

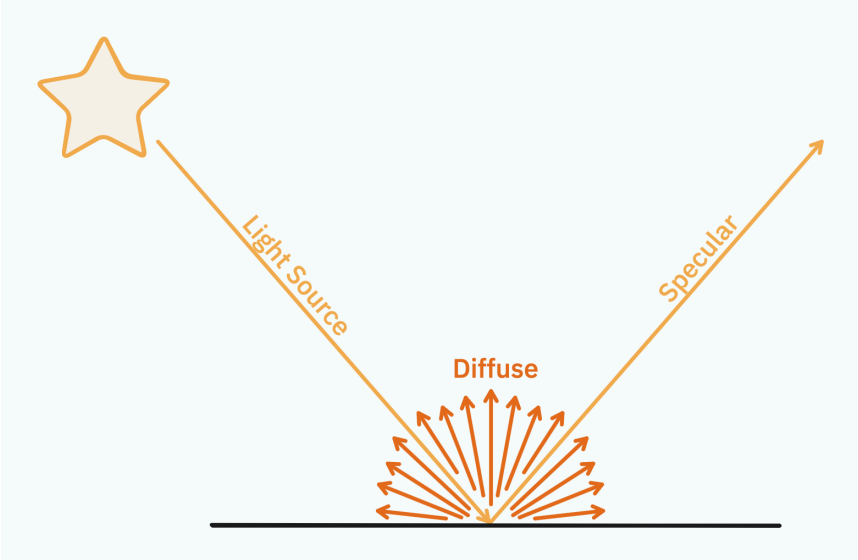


Figure 22: Diagram of the difference between diffuse reflections and specular reflections.

In Phong lighting, diffuse light is a measure of how much the face is perpendicular to the light source. As you can see in Figure 21, faces pointing towards the light are brighter, and faces pointing away from the light are darker. Specular light is the result of light rays bouncing directly from the object to your eyes (Wolff, 1994). You can see this in Figure 22. I implement this by checking how close a point is to pointing directly at the camera.

As you can see in Figure 21, adding together these lighting “layers” results in a somewhat convincing model of light reflection.

6.b. Updating `winit`

The WGPU tutorial project I was referencing, Hansen (2024), used an old version of the rust library `winit` (0.29). This library handles everything to do with windows: creation, destruction, input, etc.

I decided that I would update the version to 0.30 before starting any new work on the project. This version of `winit` changes the way you instantiate a window. Instead of creating an event loop with a closure, I pass in a structure that implements the `Application` trait. The event loop calls methods on this structure.

This has issues. Mainly, `wgpu` requires asynchronous code in initialising some types. In Rust, asynchronous functions return types called `Futures` with `Send` and `Sync` traits. These are explicitly unsupported within WebAssembly, which is how I can run my application in the web browser.

In the previous `winit` version, I got around this by using a library to handle these types with the web's APIs. I could safely make these types, that make up the GPU handles, and then get on without using any more asynchronous code. On desktop platforms, I didn't spawn a thread, and instead block on the asynchronous function.

The new structure of `winit` means that I can't just pass in this handle though. In Figure 23 you can see that I have to create a window before starting any interactions with the GPU.

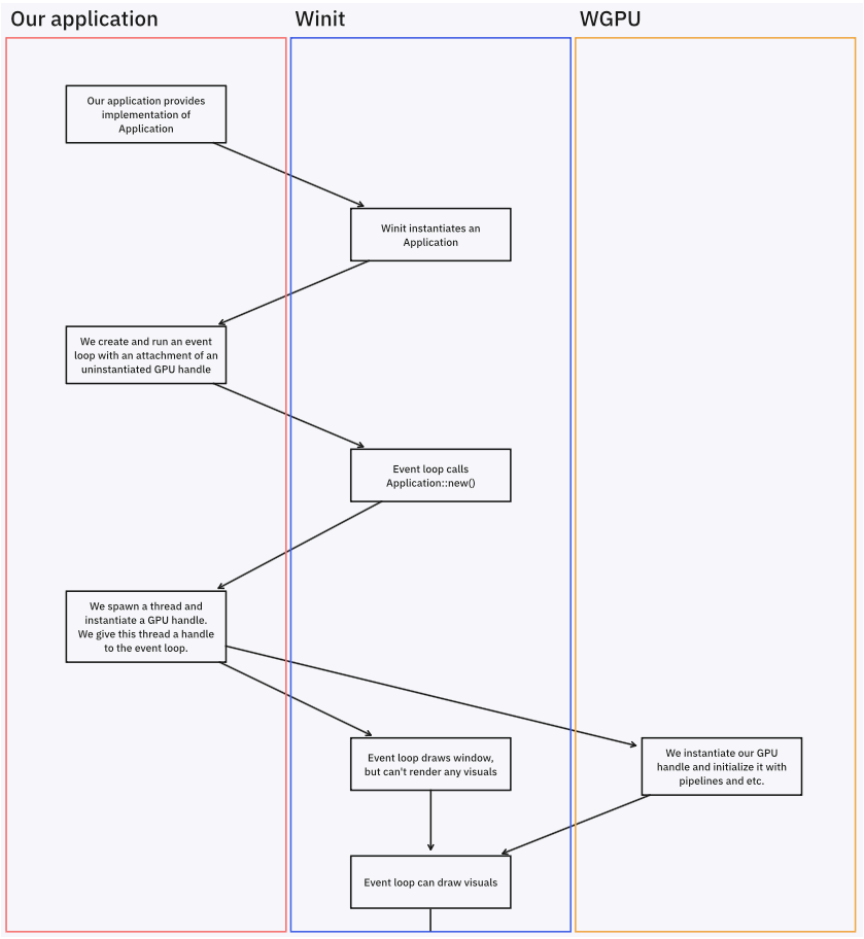


Figure 23: An interaction diagram of my application, `winit` and `wgpu`. I mainly followed this structure: erer1243 (no date), with some referencing from Bentebent (2024) in development.

6.c. Infinite Worlds

One might wonder, how do you represent an infinite world within a finite machine? Surely, the storage requirements would make a such a feat infeasible? The answer is that you don't store an infinite world. You *generate* an infinite world from a series of algorithms. But how do you load it?

Figure 24: The GameCube game *SSX 3*, running in the Dolphin emulator.

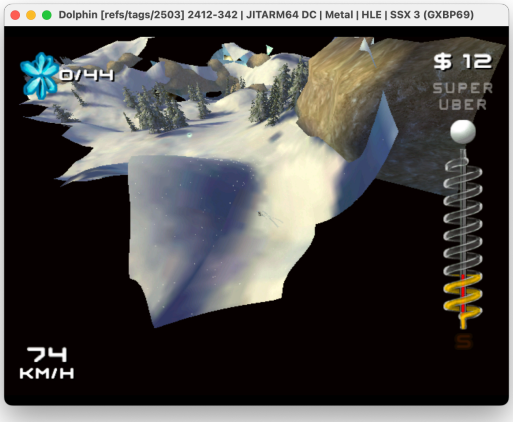


Figure 25: The game, as shown in Figure 24, but with the camera zoomed out. This reveals that the world is only rendered in fragments of level geometry.

SSX 3, shown in Figure 24, is a game that boasts a large continuous world map. This means you can start at the top of a mountain peak, and snowboard all the way to the base (John Linneman, 2018). Achieving this on a system like the GameCube is a challenge. With a low amount of RAM, it would be quite hard — if not impossible — to load all the level geometry at once. In Figure 25, you can see that the landscape geometry is cut off. This demonstrates that the game only shows what the camera is able to see. This takes advantage of the fast memory bandwidth of the GameCube (Nintendo, no date) to stream in "chunks" of the level geometry (John Linneman, 2018).

This is a technique that I will call "chunking." Chunking is the practice of breaking a world into distinct sections. It is used in games where the world geometry is too large to fit within the RAM of the hardware. Chunks in my game engine are a $16 \times 16 \times 16$ cube of voxels. Now that I have a representation of chunks in Listing 3, I need to define a way to generate chunks.

```rust
pub type Array3 = [[[Block; 16]; 16]; 16]; // In real code, we set these values with constants, as magic numbers are bad practice. I have made them numbers for readability in these demonstrations.

pub struct WorldMap {
    pub chunks: HashMap<IVec2, Chunk>,
}

pub struct Chunk {
    pub blocks: Array3,
}

pub enum Block { AIR, BRICK }

pub fn ar3get(ar3: &Array3, x: usize, y: usize, z: usize) -> Block {
    sl3[y][z][x] // Note that we go x/z/y. This is arbitrary.
}
pub fn ar3set(ar3: &mut Array3, x: usize, y: usize, z: usize, new: Block) {
    sl3[y][z][x] = new;
}
```

Listing 3: A basic Rust data structure that encodes a world map, containing chunks. These chunks contain a 3D array of blocks.

```rust
fn new_chunk(chunk_x: i32, chunk_z: i32) -> Chunk {
    let mut blocks = Array3::default();

    for (x, z, y) in itertools::iproduct!(0..16, 0..16, 0..16) {
        let (xf, zf) = (
            (x as i32 + (chunk_x * 16)) as f32,
            (z as i32 + (chunk_z * 16)) as f32,
        );

        let sines = f32::sin(xf * 0.1) + f32::sin(zf * 0.1);

        let n = ((sines / 2. * 16 as f32).round() as i32) <= y as _;
        ar3set(&mut blocks, x, y, z, {
            if n { Block::BRICK } else { Block::AIR }
        });
    }

    Chunk { blocks }
}
```

Listing 4: A chunk generation function.

In Listing 4 I use a function called `iproduct`. This is a function that comes in very handy in implementing "dimensional" operations. It is functionally the same as a procedure like Listing 5, but cleaner.

```rust
for x in 0..16 {
  for z in 0..16 {
    for y in 0..16 {
      // Do some work with coordinates.
    }
  }
}
```

Listing 5: A simple "3D" for loop. Though simple, it requires a lot of nesting. Nesting is heavily discouraged in large code repositories like the Linux kernel (Torvalds, 2016).

The `xf` and `zf` in Listing 4 are a transformed `x` and `z`. I use the formula $x_W = x_I + (x_C * 16)$ to perform the translation. I have to do this because of *coordinate spaces* (Dunn and Parbery, 2025). In this case, I need to transform from "in-chunk" (I) space to "world" (W) space. Figure 26 shows a visual demonstration of the translation I need to do in this case. Note that in-chunk space is really just the world space but with a $\bmod 16$ operation. It helps to think of it as a separate, finite space.

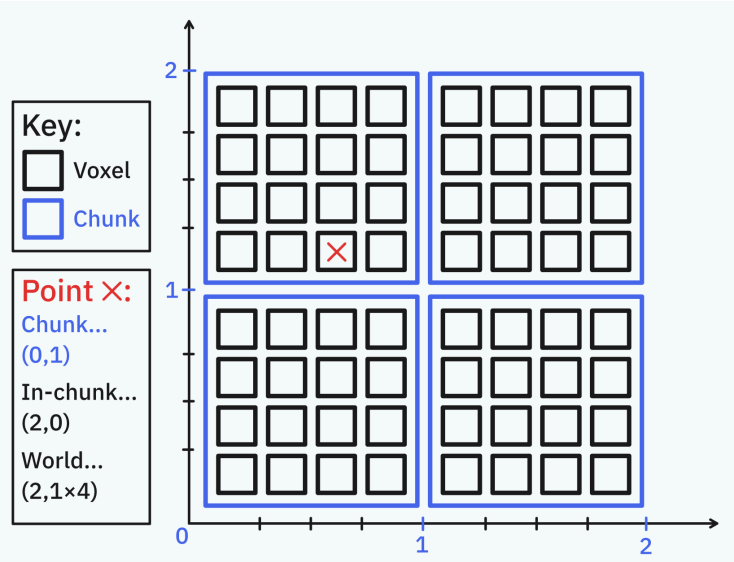


Figure 26: Diagram that demonstrates the concept of coordinate spaces with a 2D representation of an infinitely chunked world. It demonstrates three different coordinate spaces, chunk, in-chunk, and world. It shows how to convert between the three.

Once I have the transformed coordinates, I can apply a basic $\sin(x) + \sin(z)$ function to get a simple chunk generation algorithm in Listing 6. I build a world by filling it with chunks in a loop. This results in a world that looks like Figure 27.

```rust
pub fn new_map() -> WorldMap {
    const INITIAL_GENERATION_SIZE: i32 = 5;
    let iter = (-(INITIAL_GENERATION_SIZE / 2)..).take(5);

    let mut chunks = HashMap::new();

    for (x, z) in itertools::iproduct!(iter.clone(), iter) {
        chunks.insert(ivec2(x, z), new_chunk(x, z));
    }

    WorldMap { chunks }
}
```

Listing 6: Function that makes a new world map. It generates 25 chunks around the origin in a for loop, and inserts them into a new map.

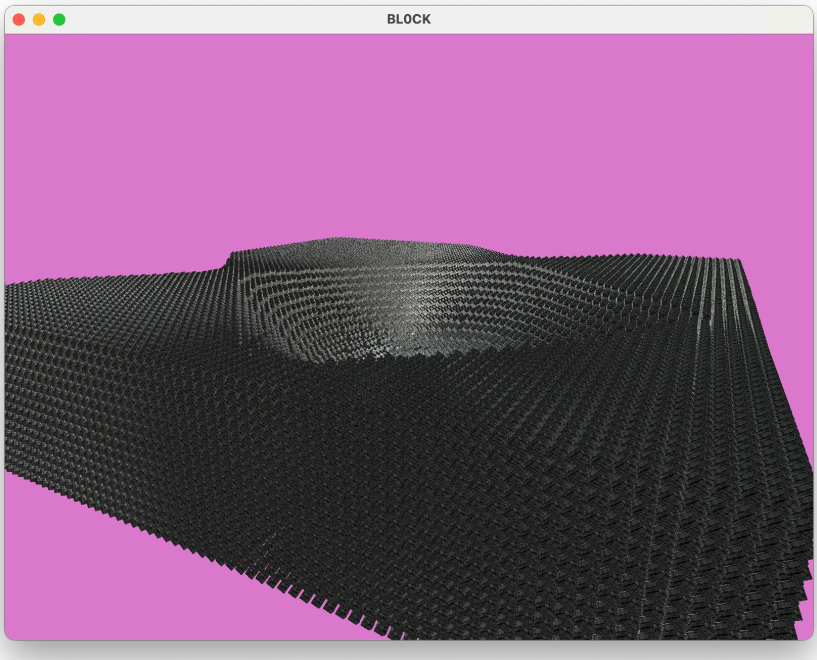


Figure 27: A large collection of cubes. The cubes are arranged to make a 3D wave. However, the top of the wave is cut off.

6.c.i. Getting a better model

The model from Hansen (2024) helped in initial development, but I need a simpler model. In Blender, I made a simple cube, and then made a texture that would help show any orientation problems (Figure 28).

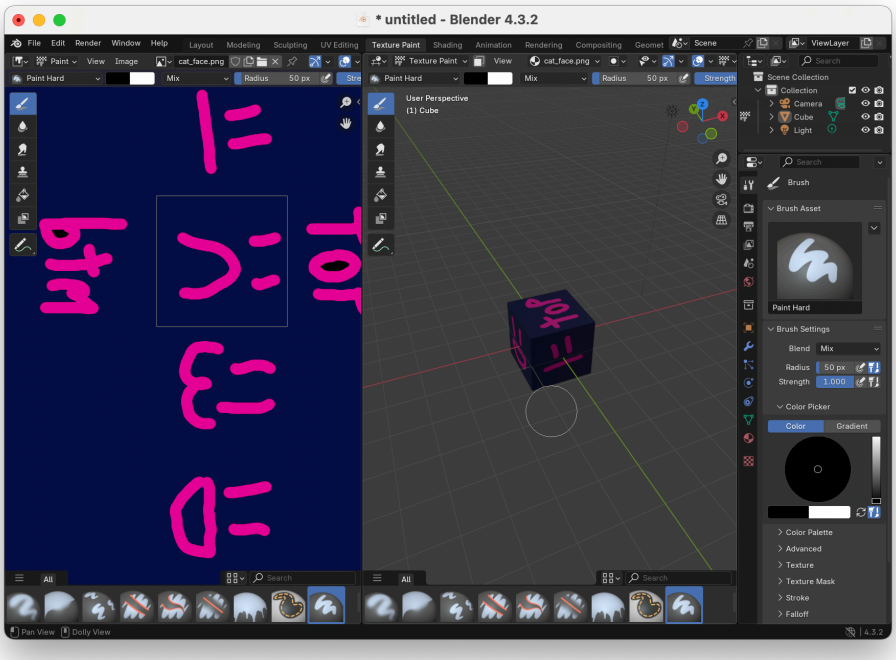


Figure 28: A cube in blender with a test texture on. Each face is labelled with a direction or a drawn face.

Due to abstracting earlier, I could easily integrate this new model. All I had to do is export the blender model as an obj, and change the path in the object loading function. With less polygons, I can render more cubes. Here I have bumped the "render distance" to an 11×11 square. As I am no longer testing rotation, I can set all of the cubes to a normal rotation. This results in a much more convincing world as shown in Figure 29.

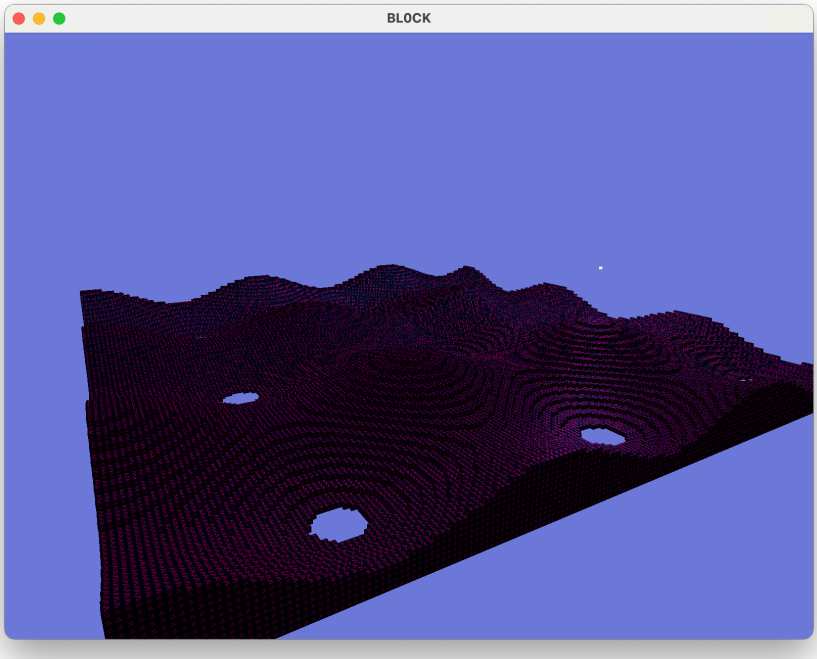


Figure 29: A large 3D grid of cubes. Together, they form large, cohesive waves of terrain.

6.c.ii. The Illusion of Infinity

To achieve the illusion of an infinite plane, you have to load chunks around the player. As I don't have a player yet in the engine, I use the camera as a substitute. I need a strong data structure to assist here, which is why I chose to add the library `rollgrid3d`. This library — used in Listing 7 — adds an abstraction where a world offset can be applied, and only show a small "window" section of the world.

```
pub struct WorldMap {
    pub chunks: RollGrid3D<Chunk>,
}
pub fn new() -> WorldMap {
    const SIZE: usize = 11;

    WorldMap {
        chunks: RollGrid3D::new(
            (SIZE as _, 2, SIZE as _), // Size of "window" to world state
            (0, 0, 0), // Offset from origin
            |(x, y, z)| Chunk::load(ivec3(x, y, z)).unwrap(), // Function to initialize a chunk.
        ),
    }
}
```

Listing 7: Initialising a RollGrid3D.

To achieve infinite loading, I need methods on `Chunk`: `load_from_file`, `save`, and `generate`. I need these to perform the functions listed in Figure 30. Due to the complexity involved in these operations, I will discuss them later in Section 6.c.iii. Once I have these functions, I can then "reposition" the window into the world. In Listing 8, I transform the camera coordinates to chunk space. I must then set a flag to remake the instance buffer.

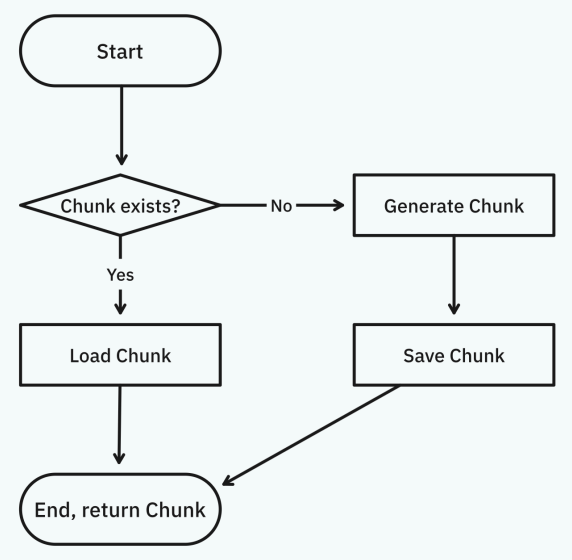


Figure 30: Flowchart that shows the logic of loading a chunk.

```
if self.load_chunks {
    const BLOCK_UNIT_SIZE: i32 = 32;
    let chunk_relative = IVec3::from(
        (camera.position.x as i32 / BLOCK_UNIT_SIZE,
        -(camera.position.y as i32 / BLOCK_UNIT_SIZE),
        camera.position.z as i32 / BLOCK_UNIT_SIZE,
    )) + IVec3::splat(-2);
    if chunk_relative != world.map.chunks.offset().into() {
        world.map.chunks.reposition((IVec3::from(chunk_relative)).into(), |_old, new, chunk| {
                *chunk = Chunk::load(ivec3(new.0, new.1, new.2)).unwrap();
            });
        *remake = true;
    }
}
```

Listing 8: The camera update function tells the world map to reposition if it has moved out of the chunk in the middle of the world map window.

The last thing I have to do to update the instance buffer. This is so the program can clear old triangles and push new ones. If it didn't do this, the logical world map would change, but the actual visuals would stay the same. Luckily, I can just extract the logic that makes the instance buffer into it's own function. With these changes, new chunks load and unload as the camera moves.

6.c.iii. Saving, Loading, and Generating Chunks

In order to store chunk data, I have to convert it to a textual format. I call this process serialisation. I can use the library `bincode` to turn the `Chunks` into bytes. I can then write these bytes to a file. As you can see in Figure 31, the `BlockKind` is turned into a `u8`. In Rust, I can use `u8` numbers to represent bytes, as they are 8 bits long.

Figure 31: `bincode` converts the data types into a minimal representation.

```rust
fn load_from_file(map_pos: IVec3) -> Result<Option<Chunk>, Box<dyn std::error::Error>> {
    let config = bincode::config::standard();
    let file_hash = calculate_hash(&map_pos);
    let file_name = format!("chunk_{}.bl0ck", file_hash);

    #[cfg(not(target_arch = "wasm32"))]
    {
        let file_path = Path::new("./save/chunk/").join(Path::new(&file_name));
        if file_path.exists() {
            log::warn!("Load Chunk!");
            let mut file = File::open(file_path).unwrap();

            let decoded = bincode::decode_from_std_read(&mut file, config)?;

            Ok(Some(decoded))
        } else {
            log::warn!("Chunk not loaded!");
            Ok(None)
        }
    }
    #[cfg(target_arch = "wasm32")]
    {
        let store = web_sys::window().unwrap().local_storage().unwrap().unwrap();
        if let Ok(Some(s)) = store.get(&file_name) {
            let s = BASE64_STANDARD.decode(s)?;
            let (decoded, _) = bincode::decode_from_slice(&s[..], config)?;

            Ok(Some(decoded))
        } else {
            Ok(None)
        }
    }
}
```

Listing 9: The “load from file” function. It acts as a useful abstraction from the architecture-specific file loading requirements.

In Listing 9, I have to have separate logic on the web or on desktop. The `cfg` annotations let me toggle code depending on the CPU architecture. On desktop, I try to load a file at `./save/chunk/chunk_POS.bl0ck`, where `POS` is substituted for a hash of the map position. On the web, I don’t have access to the system’s file system. I have to use browser APIs.

The web has multiple ways of storing items. Traditionally, cookies are used to store data. These present a few challenges, though. Mainly, cookies aren’t a persistent form of storage, as they have an expiry date. Also, cookies get sent back and forth between the server and browser within headers. If I used this to store large amounts of data, this would make the request sizes unnecessarily large (MDN, 2025b).

```javascript
window.localStorage.foo = "bar";
console.log(window.localStorage.foo); // "bar"
```

Listing 10: Basic `localStorage` usage in JavaScript

```rust
let store = web_sys::window().unwrap()
            .local_storage()?.unwrap();
store.set("foo", "bar")?;
println!(store.get("foo")?.unwrap()); // "bar"
```

Listing 11: Basic `localStorage` usage in Rust

The `localStorage` API solves these problems (WHATWG Community, 2025). I can store items in key and value pairs using the API like in Listing 10. In Listing 11, you can use the `web_sys` crate to call into JavaScript API bindings. Due to Rust’s strong typing and safety, you have to unwrap (or use the `?` operator [[cite]]) on a lot of `Option<T>` and `Result<T, E>` types that encapsulate where JavaScript might put a `null` or throw an exception [[cite]].

To mimic a hash map, I use Rust’s hashing features to encode the key (Rust Foundation, 2024). This allows me to look up keys without having to come up with a way to encode the coordinates in a string.

The saving implementation uses similar logic to Listing 9. For the chunk generation function, I adapt the function in Listing 4.

6.d. Shadow mapping

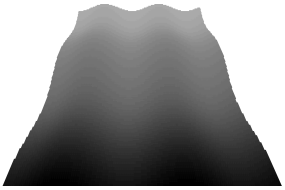


Figure 32: A shadow map exported using Xcode's Metal debugging tools.

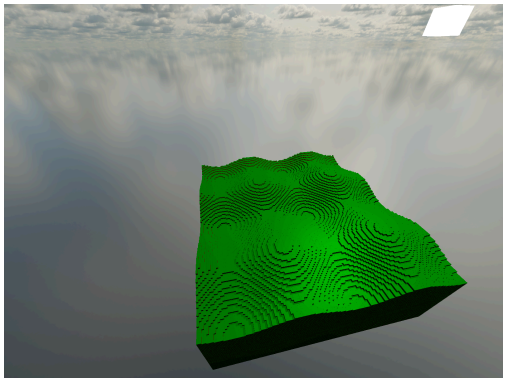


Figure 33: The world with shadows, mapped in Figure 32, projected onto it.

Shadow mapping is a method of simulating shadows in rendering. It works by treating a source of light as a camera. I aim this camera at the world, and take a depth buffer. I are using a depth buffer differently than in Section 6.a.iii, as I are using it as an input to another step, rather than used for setting the draw order of objects.

Once I get the buffer, I can sample the texture, using the projection of the light to transform the world coordinates into 2D texture coordinates. Any discrepancy in distance from the floor to the distance measured in the map is where a shadow is drawn (Akenine-Möller *et al.*, 2018).

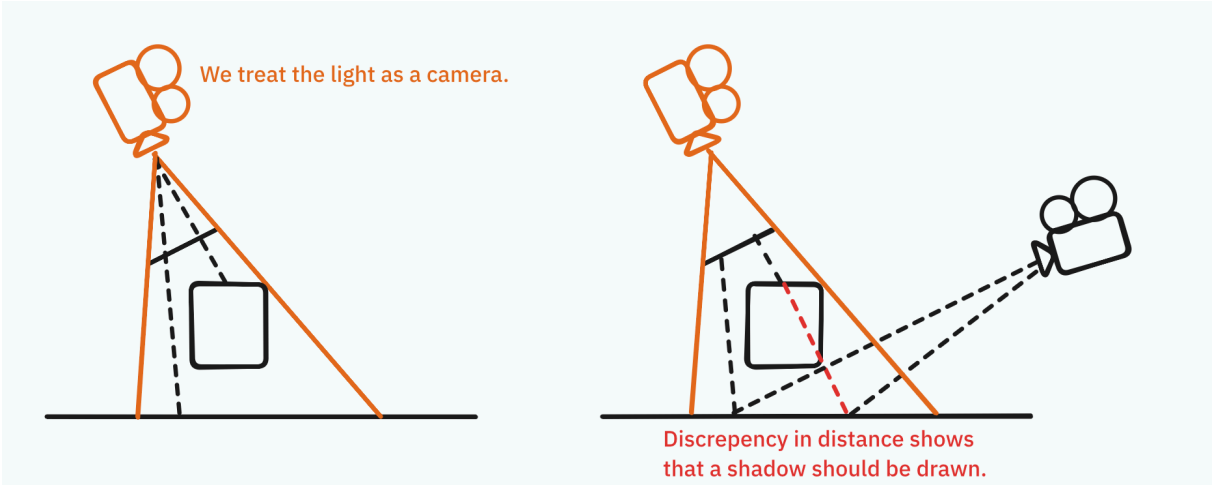


Figure 34: A simplified diagram of the process of shadow mapping. Dashed lines show "rays of light".

6.d.i. A Technical Explanation of Shadow Mapping

To achieve this in practice, I have to use render passes. Render passes let you group render pipelines into groups that can be ran in sequence. I construct a pipeline for the shadow map creation. I tell the pipeline that it's view is a texture view, instead of the screen's view. This texture can then be held onto, and a view can be created for the next pipeline. An abstract version of this process is diagrammed in Figure 35. I called these two render passes "shadow" and "forward."

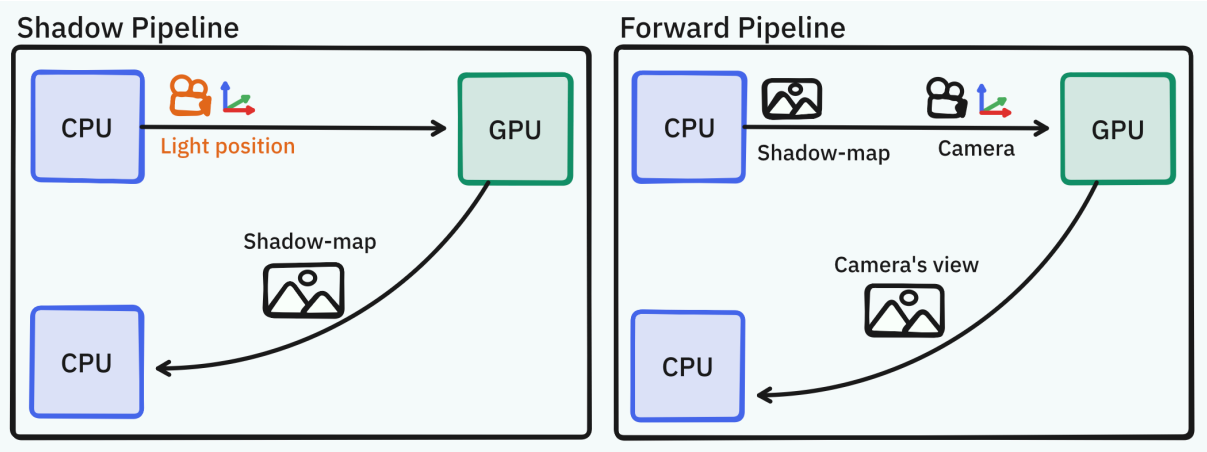


Figure 35: A simplified diagram of what happens in each pipeline. In the shadow pipeline, the position of the light, which is treated like a camera, is sent to the GPU. The GPU responds with a texture, which I treat as a shadow map. In the forward pipeline I send this texture, and the camera's positional data, to the GPU. The GPU then responds with a view of the camera.

In the forward render pass, I would like to reference the shadow map. To do this, I add the view from the previous pipeline to this new pipeline's bind group. This will then be accessible in the shader. I also pass in a comparison sampler, which will let me compare pixels in the texture (Listing 12). I then write a function that takes an input of the world position, projected onto the light's view (Listing 13).

```
@group(1) @binding(2)
var t_shadow: texture_depth_2d;
@group(1) @binding(3)
var s_shadow: sampler_comparison;
```

Listing 12: The texture view and sampler, accessible as variables in the shader.

```
fn fetch_shadow(homogeneous_coords: vec4<f32>) -> f32 {
    if (homogeneous_coords.w <= 0.0) {
        return 1.0;
    }
    let flip_correction = vec2<f32>(0.5, -0.5);
    let proj_correction = 1.0 / homogeneous_coords.w;
    let light_local = homogeneous_coords.xy * flip_correction * proj_correction + vec2<f32>(0.5,
0.5);

    return textureSampleCompareLevel(t_shadow, s_shadow, light_local, homogeneous_coords.z *
proj_correction);
}
```

Listing 13: This function was adapted from a WGPU example file (*wgpu-rs/examples/shadow/shader.wgsl at master · gfx-rs/wgpu-rs*, no date). Note that the flip correction is done because *Normalised Device Coordinates* are Y-up instead of Y-down, and range from $(-1 \text{ to } 1)$ instead of texture-space $(0 \text{ to } 1)$ (Carmen Cincotti, 2022).

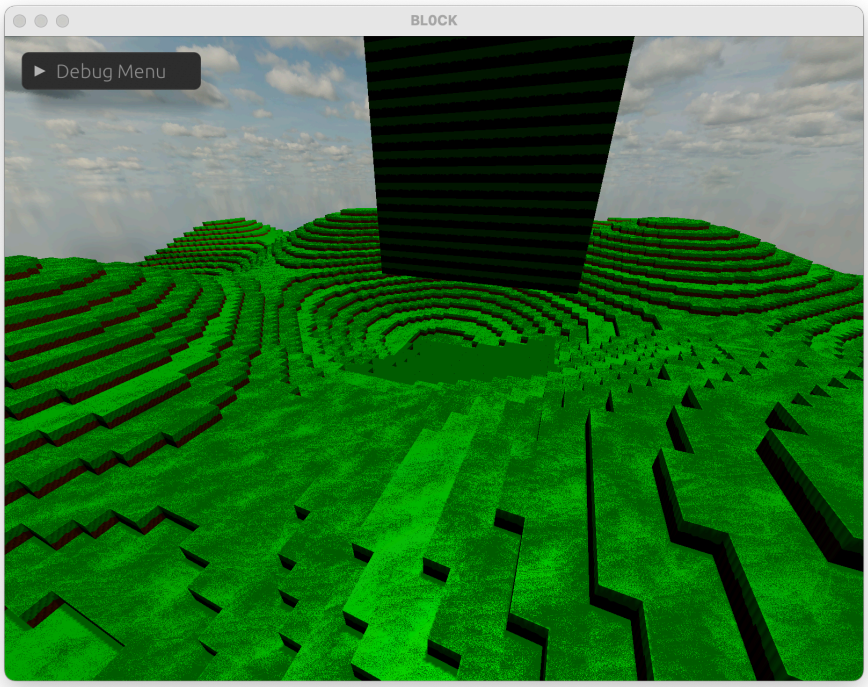


Figure 36: Naive implementation of shadow mapping. Note the large amount of noise left on objects meant to be in light. This is "shadow acne."

6.d.ii. Explorations of Fixing Surface Self-Shadowing

Surface self-shadowing, or colloquially "shadow acne", is an issue with a naive shadow map implementation. It happens because of inaccuracies in sampling, and inaccuracies in floating point maths. There are two ways to ameliorate self-shadowing: front-face culling, and biasing (Akenine-Möller *et al.*, 2018).

Biasing is where you add a slight positive offset to the depth lookup. This biases the algorithm to shadow slightly less. This is because a lot of shadows created because of a shallow depth differential self-shadow. I found that a bias of -0.001 would hide artefacts in most areas, but at the edges far from the light source there would be moiré-pattern artefacts (Zhao, no date) as seen in Figure 37. I believe this is because my lighting is somewhat of a "worst case" for shadow mapping, where acute angles to shadowed objects mean that precision issues get exacerbated.

```
let pipeline = device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    primitive: wgpu::PrimitiveState {
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        front_face: wgpu::FrontFace::Ccw,
        cull_mode: Some(wgpu::Face::Front),
        polygon_mode: wgpu::PolygonMode::Fill,
        unclipped_depth: false,
        conservative: false,
    }, ...
}
```

Listing 14: Enabling front-face culling in the shadow-mapping pipeline.

Face culling is a basic optimisation technique. It is built into WebGPU's primitive drawing: it lets me specify which, if any, side I want to be culled (Listing 14). Usually, people cull the back face, as this is a face that the camera cannot see. I choose to cull the front face in creating the shadow map as, theoretically, only the back face of an object would be casting a shadow. This technique is mostly successful, but leaves some grid-shaped shadow artefacts as seen in Figure 38.

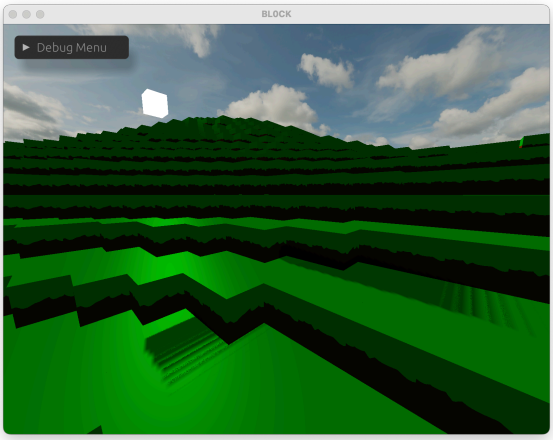


Figure 37: Shadows with only biasing implemented to prevent self-shadowing. Note that shadow edges far from the light source get repeated, and show moiré-pattern artefacts.

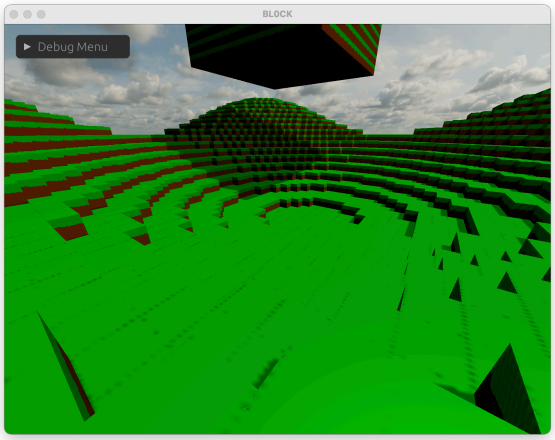


Figure 38: Shadows with front-face culling as an amelioration for self-shadowing. Dotted grid lines of shadows appear in lit areas. This could be a consequence of having a large voxel world.

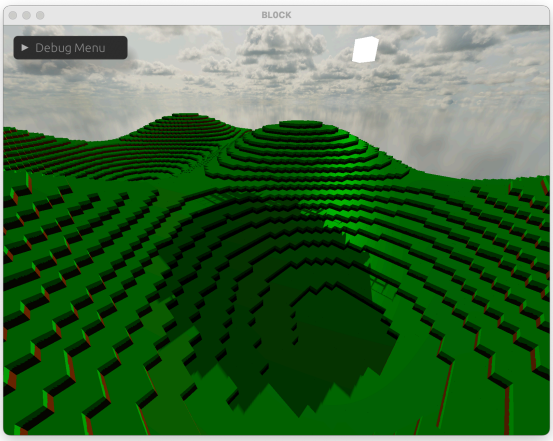


Figure 39: A shadow with front-face culling and biasing applied. Note that there block outline shaped artefacts near the edges of the shadow body.

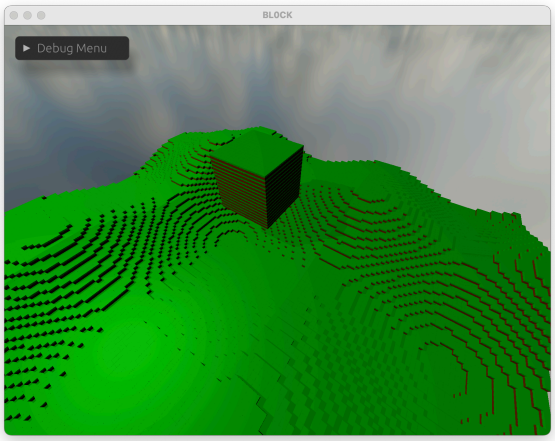


Figure 40: A demonstration of the final shadow implementation. Shadows are being cast by the terrain geometry, as well as the floating chunk of voxels. Some “shadow acne” can be seen in lit areas from Figure 38, but smoothed with a Gaussian blur.

I found that combining biasing and front-face culling did not produce a better result, as one would expect. Instead, it created sharp shadows, but shadows of voxels near the edge of were hollowed as seen in Figure 39. I believe this is because the offset caused depth lookups to be done on culled faces. I chose to go with the front-face culling approach, but I believe that with more research the biasing approach could also be a successful way to remove self-shadowing.

6.e. Trying to implement Chunk Meshing

One of my goals in this project was to have a large render distance. One of my largest issues with voxel games is that the low view distance makes these worlds feel claustrophobic. In my own project, I was finding that I couldn't go past about 9^3 chunks in render distance without experiencing frame rates under 60fps.

The first solution that most people would take is to *mesh* chunks. Instead of drawing cubes for every voxel, you instead only draw the faces that the camera would be able to view. This is a somewhat simple optimisation, and should have been easy to implement.

I had previously been loading a cube model from an OBJ file. This was good for quickly getting a cube loaded. However, in order to only load each face I wanted instead of the whole model, I needed to manually de serialise the cube object. This is something that I ended up doing manually, by comprehending the OBJ file format and transcribing the vertices, indices and texture coordinates.

```
pub fn primitive_model(&self, (cx, cy, cz): (i32, i32, i32), device: &Device, material:
&Arc<Vec<Material>>) -> Option<Model> {
    let mut mesh = PrimitiveMeshBuilder::new();

    for (x, y, z) in iproduct!(0..Chunk::X, 0..Chunk::Y, 0..Chunk::Z) {
        if self.get(x, y, z) == BlockKind::Brick {
            let faces = Faces::normals().map(|IVec3 {x: nx, y: ny, z: nz}| {
                self.get_opt(x + nx, y + ny, z + nz) != Some(BlockKind::Brick)
            }); // Simplified for brevity
            let faces = Faces::from_arr(faces);

            let offset = |x: usize, cx: i32, max: usize| {
                (cx * 2.0 * max) + x * 2.0 // Simplified for brevity
            };

            mesh = mesh.cube(faces, offset(x, cx, Chunk::X),
              offset(y, cy, Chunk::Y), offset(z, cz, Chunk::Z));
        }
    }

    mesh.build(device, material) // Simplified for brevity
}
```

Listing 15: Function that takes a chunk and returns a mesh of the chunk

I ran into an issue where seemingly random faces would be culled instead of the expected ones. I spent about two weeks scrutinising this code and the code around it. I rewrote it a few times and abstracted over concepts to help me understand my code. Eventually, I found the culprit:

```
pub fn cube_indices(n: u32, faces: Faces) -> Vec<u32> {
    let vertices_count = cube_vertices(faces, 0., (0., 0., 0.)).len() as u32;
    let front = vec![0, 1, 2, 0, 2, 3];
    let back = vec![4, 5, 6, 4, 6, 7];
    let top = vec![8, 9, 10, 8, 10, 11];
    let bottom = vec![12, 13, 14, 12, 14, 15];
    let right = vec![16, 17, 18, 16, 18, 19];
    let left = vec![20, 21, 22, 20, 22, 23];

    // Removed for brevity
}
```

Listing 16: The function that caused the bug. It takes a cube's faces and returns its indices.

I had found what I later called "essentially an out-of-bounds on the GPU." This `cube_indices` function was hard-coded to understand that a cube must have 24 vertices (which is the standard for a full cube, see Figure 41).

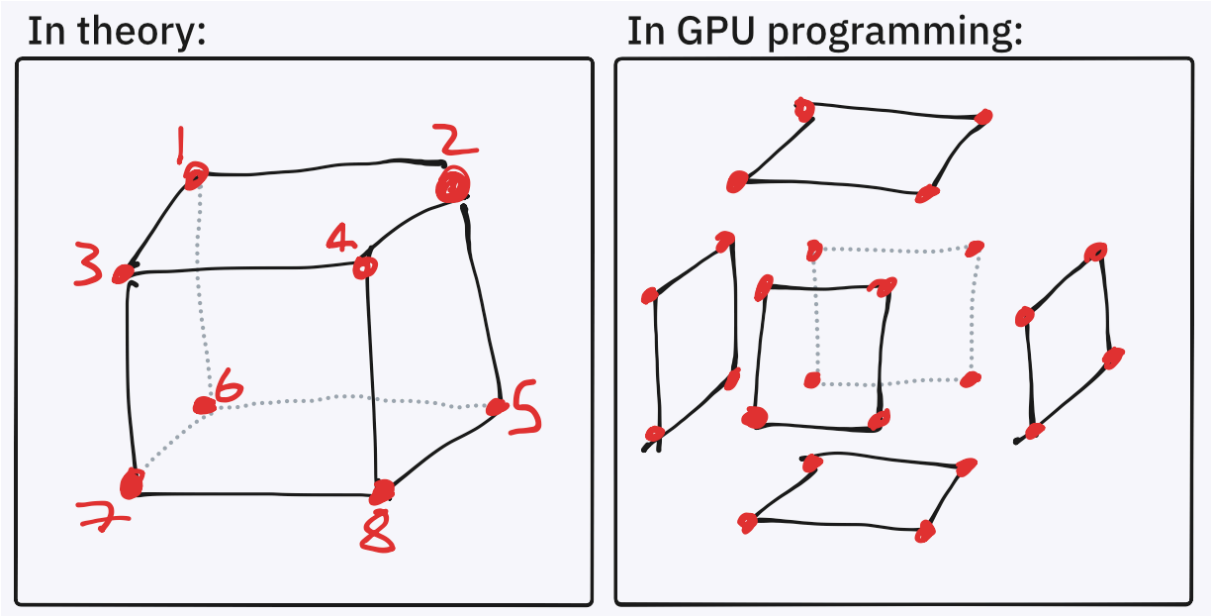


Figure 41: An illustration of why there are 24 vertices instead of 8 for cubes in GPU programming. The faces cannot “see” each other, so linked vertices have to be duplicated.

As you can imagine, if faces are removed, the amount of vertices will lower. This is what caused the odd behaviour. My struggles in reproducing it were because the out-of-range behaviour seems to be undefined behaviour, though I have not found any sources for this. In testing, the out-of-range vertices point to vertices on other cubes.

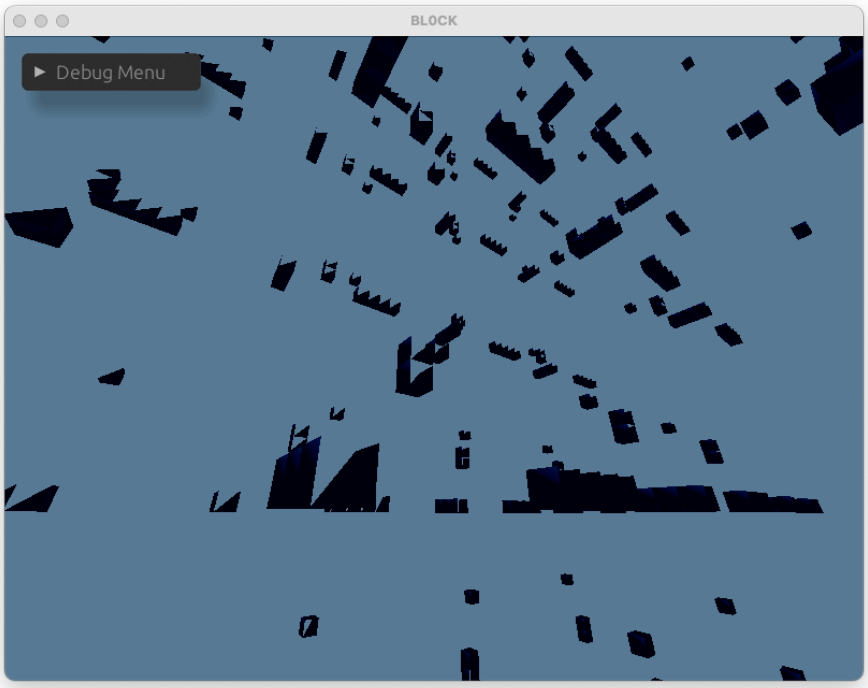


Figure 42: A recreation of the meshing issue.

```rust
let vertices_count = cube_vertices(faces, 1., (0., 0., 0.)).len() as u32;

const VERTICES_PER_INDICES: u32 = 4;
let face_indices = [0, 1, 2, 0, 2, 3];
let mut all_indices = vec![];
for (i, face) in Faces::arr(faces).iter().enumerate() {
    let indices = face_indices.map(|n| n + (i as u32 * VERTICES_PER_INDICES));

    if *face {
        all_indices.push(indices);
    }
}
```

Listing 17: The fixed section of code.

After fixing this code, my meshes almost looked good. I had implemented *simple chunk meshing*. My next step was to add logic to fix meshes where chunks met. After completing this, I weighed my options and decided to move work on this feature to future work to focus on implementing other goals. This code was forked to a git branch: `feat/chunk_meshing`.

I would have caught this bug much earlier if I had a better grasp on the concepts of GPU programming. For a long time in development, the concepts of vertex and index buffers were somewhat confusing to me. This could have been avoided if I had done deeper research into the fundamental data structures of GPU programming.

6.f. Improving the Camera

In a traditional first-person game the camera is controlled by movement of the mouse. In practice, this means modifying the *yaw* and *pitch* of the camera.

Previously, the camera was fixed at pointing towards the origin $(0, 0, 0)$ of the world. This heavily restricts how you can move around the world. To fix this, and allow mouse control of the camera, I have to change the camera model (Figure 43).

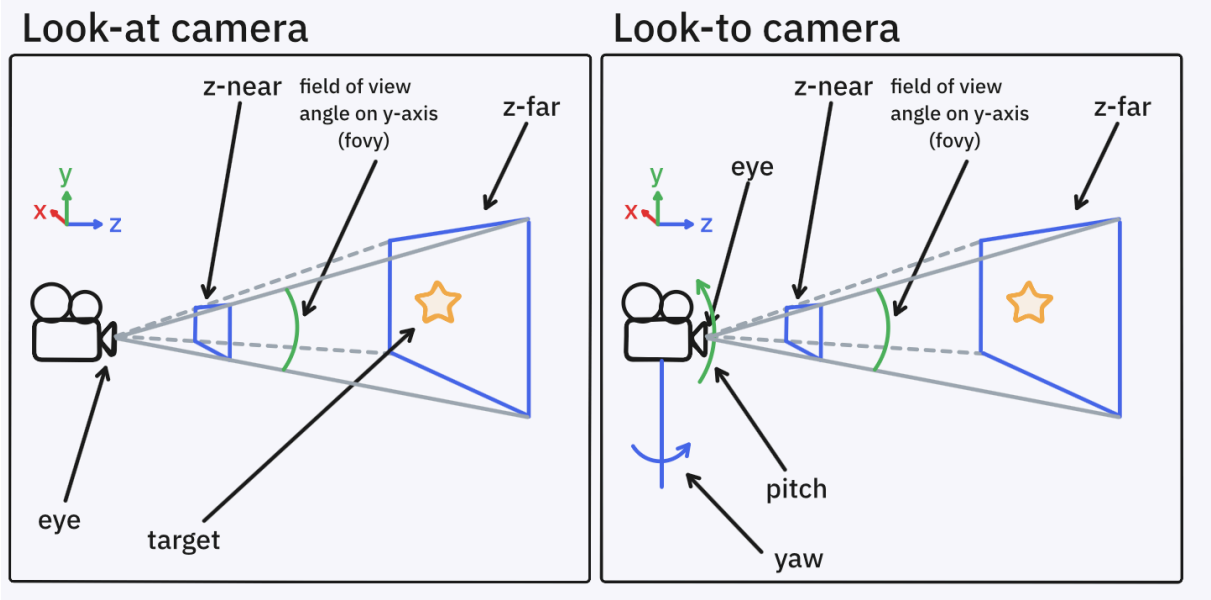


Figure 43: Two models of storing camera positional and frustum data. The “look at” model is defined with a target in mind. When calculating the position, the target is used to change how the camera is facing. The “look to” models is designed to give input in the facing direction of the camera, without a specific target. This model is more suitable for giving a user control of the camera.

Input was also changed. The traditional “WASD” arrow key movement allows movement on the x and z axes, and the space and shift allow movement up and down the y axis. Mouse movement is stored as a frame-by-frame difference of a 2D vector, which is added directly to the pitch and yaw rotations.

When updating my camera to a look-to model as in Figure 43, I did not notice that this broke the implementation for the sun light. It was not changing direction to look at the world, which meant that in some angles there were no shadows being cast. To fix this, I had to make my camera implementation generic over “look-at” and “look-to”, and allow choice in which model to use. In future work, this could be fixed by creating an “omnidirectional” light that contains six cameras facing in all directions. Care would have to be taken to make sure the frustums cover all parts of the light without causing any overlap issues (Akenine-Möller *et al.*, 2018).

6.g. Optimising Save and Load Operations in the Hopes of Instantaneous Chunk Loading

One issue that was plaguing my engine was freezes that occurred when the camera went into new chunks. These chunks have to be loaded from disk, turned into model data, then sent to the GPU. These freezes were about a second long. When I compiled with optimisations, they shrunk to about 100ms long. I first looked to my implementation of saving and loading chunks.

My naive saving implementation *on desktop* was to use `bincode` to serialise the chunk structure to a file. This means I had a bunch of files. Consider that every block is a 16^3 cube of either 0 or 1. A typical "render distance" in my engine is $7^3 = 343$ chunks. This means that every time I had to load a world in, I had to access 343 files.

I had not realised it at the time, but I had fallen into this perfect example of this benchmark from SQLite (2025). The chunks are small "blobs" of data, and I was doing a lot of lookups by coordinates. Shockingly, storing these in **one file** is 35% faster! The overhead of doing file system lookup outweighs the speed of a SQL-based database. As I was already doing lookup, it felt fitting to replace my code with this.

```rust
let mut stmt = conn.prepare_cached(r#"
    SELECT (data) from chunks
    WHERE (x,y,z) == (?,?,?)
"#,
)?;
let i: Vec<u8> = stmt.query_row((map_pos.x, map_pos.y, map_pos.z),
  |f| f.get("data"))?; // Simplified for brevity

let (decoded, _) = bincode::decode_from_slice(i.as_slice(), config)?;

Ok(Some(decoded))
```

Listing 18: Loading a chunk from a SQLite database.

However, this did not visually change the freezes that would happen when loading new files.

My next attempt at optimising the loading further was this: Instead of doing all of this computation on the main thread, I put this work on it's own thread. This would theoretically reduce the time that graphics rendering is frozen.

```rust
let thread = spawn(move || {
  let w = world;
  let mut conn = rusqlite::Connection::open("./save.sqlite").unwrap();

  loop {
      let Ok((x, y, z)) = rx.recv() else { break; };
      let Ok(ref mut w) = w.lock() else { break; };

      w.map.chunks.reposition(
          IVec3::from((x, y, z)).into(),
          |_, (i, j, k), chunk| {
              *chunk = Chunk::load(ivec3(i, j, k), &mut conn).unwrap();
          },
      );

      w.remake = true;
  }
});
```

Listing 19: Spawning a new thread that loads chunk data into the world.

This turned out to not be fully successful. This is due to the mechanisms of a mutex (mutual exclusion) lock (Steve Klabnik, Carol Nichols and Chris Krycho, no date). The mutex holds data – in this case, the World state – and will only let one entity access it at once. Other entities trying to access the data have to wait until the previous entity finishes before accessing.

What is happening, is that the main thread is waiting on this mutex. However, I had not benchmarked this, and did not know how badly it was affecting these freezes..

What I ultimately found out, when doing exploratory debugging, was that *chunk loading was not the bottleneck*. The mutex was freed almost instantly. My real bottleneck was in computing the instance buffer. This $O(n^3)$ function would have easily been able to exported to another thread, as it does not need any handles to windowing or GPUs.

```rust
fn remake_instance_buf(map: &WorldMap) -> Vec<Instance> {
    let mut instances = vec![];

    const SPACE_BETWEEN: f32 = 2.0;
    for (coords, chunk) in map.chunks.iter() {

        // Creates an iterator like this:
        // (0,0,0), (0,0,1)... (0,1,0)... (1,0,0)... (15,15,15)
        let _3diter = itertools::iproduct!(0..CHUNK_SIZE.0, 0..CHUNK_SIZE.1, 0..CHUNK_SIZE.2);

        let mut i = _3diter
            .filter_map(|(x, y, z)| { ... }) // Redacted loop for brevity
            .collect::<Vec<_>>();

        instances.append(&mut i);
    }

    // Redacted for brevity

    instances
}
```

Listing 20: The function that computes a new instance buffer from world data.

In testing, an un-optimised build takes roughly 450ms. An optimised build takes roughly 85ms. This large gap shows that this function is non-optimal, and should have been a big target for optimising. In future development, I hope to fix these freezes though

This shows that I am not experienced in debugging. This issue, visually, looks like the program is freezing on load. I would have found this out if I had used benchmarking tests like *flame graphs*, or just measured timings, before committing to writing an optimisation.

7. Conclusion

In summary, I believe that I have made a solid foundation for future game development on this engine. I have done this by meeting the needed requirements listed at the start of my project.

In the below sections I go into detail about which requirements that I have and have not met.

7.a. Renderer

I have implemented a renderer. It renders voxels in a grid. To satisfy the infinite world part of this requirement, the world has been broken up into voxel chunks (Section 6.c), which get selectively loaded at a set distance from the camera.

I have attempted a few methods of optimising the renderer. First is back-face culling (Section 6.a.i), which halves the amount of triangles required in a draw call by hiding faces that face away from the view. Secondly, work was done into updating dependencies to ensure I was benefitting from the latest updates (Section 6.b). Finally, chunk meshing (Section 6.e) drastically lowers the amount of triangles that need to be rendered.

Chunks shown	Mesh build time	Memory Usage
$21^2 \times 2 = 882$	$3.70\,\text{sec}$	1.08 GB
$25^2 \times 2 = 1250$	$5.66\,\text{sec}$	1.48 GB
$29^2 \times 2 = 1682$	$6.88\,\text{sec}$	1.94 GB
$33^2 \times 2 = 2178$	$8.90\,\text{sec}$	2.45 GB
$37^2 \times 2 = 2738$	$12.37\,\text{sec}$	3.07 GB

Table 1: Table showing the performance characteristics of the program when the render distance is increased. Chunks are added at a rate of $(21 + 4t)^2 \times 2$, where t increases once every new test.

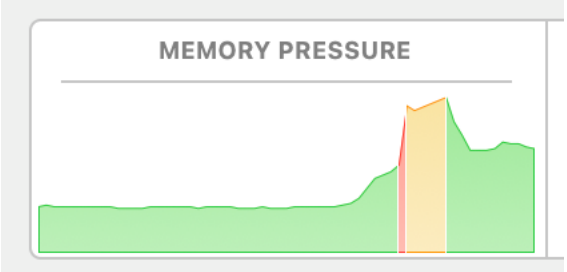


Figure 44: Memory pressure metrics when starting the program with a render distance of $37^2 \times 2$ in Table 1. As the program goes to present an image, the memory usage sharply spikes, and then relaxes. This is probably due to the need to copy buffers.

With chunk meshing I can show a lot of cubes, but it reaches a limit of RAM. While I can render a $21^2 \times 2$ cuboid of cubes, an optimised build of my program takes $3.70\,\text{sec}$ to build a mesh, as seen in Table 1. These metrics suggest that there is a need to optimise memory copies when going to present a new collection of meshes, so that huge memory spikes like Figure 44

don't affect a user's computer. To avoid these large spikes in future work, I could look into saving memory by using shared memory on platforms that support it. `wgpu` supports this through an extension for non-web platforms (nikitablack, 2021; Rust Graphics Mages, 2025).

I would say that this requirement is *mostly* satisfied. I would not say that the program performs better than current engines in use at this point.

7.b. Terrain Generation and Representation

Terrain in the current iteration of the engine is a basic continuous wave (Section 6.c). This is done through a very simple trigonometric function, and produces predictable results every time. This is great for testing the other parts of my engine; any erroneous presence or lack of cube could be due to a random variance in terrain *or* an issue in how I render, store, or create the mesh of that cube.

Noise algorithms are a large topic and — as I have learned — terrain generation is an imperfect science. Minecraft generates terrains through a very complex set of algorithm "passes." Some passes contain noise generation, but others contain things like edge detection, transformation, and smoothing (Rose, 2021; Zucconi, 2022). Refining a large pipeline of algorithms would take up a large portion of development. While researching this topic, I realised that implementing this would not be viable within the allocated development time. Therefore, I have mostly *not* met this requirement.

In future work, I would love to implement a less naive version of terrain generation. There are many different procedural algorithms to generate worlds, including: Value Noise, Particle simulations, Diamond Square, Cellular Automata, etc. (Nordeus, 2013; Taylor, 2022). Research would likely needed to be done into the most effective generation algorithm for my purposes.

This requirement also specified the need to represent terrain in memory. Implementation of this was done in Section 6.c.ii, where a data structure library let me dynamically load Chunk structures into memory when needed. However, as seen in Table 1, these chunk structures take a considerable amount of memory in the running program. It would be nice to optimise for space efficiency here. In future work, I could achieve this by using Rust features like *Packed Representation* (Desires *et al.*, 2025), which doesn't leave spacing between members in a structure. I could also try to find a more compact data representation for blocks in memory, but care would have to be taken to make sure that I can make additions like adding new blocks without issue.

7.c. Saving and Loading Terrain Data

Saving and loading involves the process of (de-)serialising data to a storage medium. In Section 6.c.iii, the `bincode` library handles most of this work, and storage is done though the path of least effort. However, there were *some* performance implications in storing each separate chunk in separate files, which is why I switched to using a file-based database in Section 6.g. I would say that I have mostly met this requirement.

In future work, auto-saving could be done to protect the user from exiting the program without saving any progress. On desktop, data storage could be done to a more standard location for application data instead of next to the binary. On web, certain browsers flush Local Storage, so true persistent storage would require server-side storage to be implemented.

7.d. Web Support

As a refresher, WebAssembly (Wasm) is a program bytecode that can run in a web browser. The work-in-progress World Wide Web Consortium (2025) specification defines it as a fast, portable way to distribute programs in the browser. The program, a renderer written in the Rust programming language, is compiled to Wasm, which is distributed as a blob that runs in the browser of a user. The library `wgpu` calls the native WebGPU browser API through JavaScript in this Wasm runtime.

The Wasm browser runtime presents various issues in development. Of concern in my project is: displaying content, taking input from the user, storing data, and running asynchronous procedures. All of these have to be done through bridging to browser-native JavaScript APIs, and crossing this bridge defines where the program can break in porting efforts.

Displaying content and taking input from the user is done by the `winit` library, which calls into JavaScript, creates a `<canvas>` element, and renders to it. Taking input can be similarly done through APIs. Storing data is done in Section 6.c.iii through the Local Storage API. Asynchronous execution is a hard problem: Section 6.a and Section 6.b discuss the complexity of having to either avoid asynchronous execution, or convert functions to use the JavaScript Promise API.

Due to the somewhat spotty support of the WebGPU API at the moment, the only browser that can run my program without modification is Chrome (and it's derivative Edge.) However, with a simple flag toggle in all major browsers can run my program without issue. Therefore, I would say that I have met this requirement.

7.e. Camera and character controller

The camera is functional, mostly easy to use through conventional game controls, and easy to extend. Building a character controller on top of this system would be a lot easier than building it from scratch, so this will make future work easier in this field.

7.f. Entity Component System, Plugin Loader and Multiplayer Networking

Since I have not implemented many entities in my game engine yet, I did not feel the need to start working on the entity component system. This would be a great next step for development, as it would create a useful abstraction to clean up the various large systems like: camera, lights, terrain worlds, etc.

Again, a plugin loader did not make sense to start working on at the current level of development in the engine. This would be a great future task, as it would help set out my engine from other engines in the game market.

Multiplayer networking is a big requirement, but one which a lot of voxel engines support. Future work on this would lead to a more “complete” engine. Unfortunately, a lot of work needs to be done to implement networking, and would likely be a large future research topic.

These larger goals are more auxiliary, and would turn this foundation of an engine into a larger piece of successful software.

8. Bibliography

Agtrigormortis (2023) *Minecraft's poor optimization, a basic issue which should have been gone a long time ago. - Discussion - Minecraft - Minecraft Forum - Minecraft Forum*. Available at: https://www.minecraftforum.net/forums/minecraft/discussion/3175130-minecrafts-poor-optimization-a-basic-issue-which (Accessed: 13 November 2024).

Akenine-Möller, T. *et al.* (2018) *Real-time rendering 4th edition*. Boca Raton, FL, USA: A K Peters/CRC Press.

Alon Rabinovitz (2023) *Safeguarding our community: CurseForge Fighting Malware Incident Report*. Available at: https://blog.curseforge.com/safeguarding-our-community-curseforge-fighting-malware-incident-report/ (Accessed: 5 November 2024).

Amethyst Foundation (2024) *Legion ECS*. Amethyst Foundation. Available at: https://github.com/amethyst/legion (Accessed: 13 November 2024).

Ammann, M. (2022) *Stencil Testing in WebGPU and wgpu | Max Ammann*. Available at: https://maxammann.org/posts/2022/01/wgpu-stencil-testing/ (Accessed: 9 April 2025).

Anderson, C. (2020) *Bevy 0.2*. Available at: https://bevyengine.org//news/bevy-0-2/ (Accessed: 13 November 2024).

Anonymous User (2019) *Vulkan for rendering instead of OpenGL*. Available at: http://feedback.minecraft.net/hc/en-us/community/posts/360043041791-Vulkan-for-rendering-instead-of-OpenGL (Accessed: 13 November 2024).

Antich, A. (2023) *Unity DOTS / ECS Performance: Amazing*. Available at: https://medium.com/superstringtheory/unity-dots-ecs-performance-amazing-5a62fece23d4 (Accessed: 13 November 2024).

Apple Inc (2018) *About OpenGL for OS X*. Available at: https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_intro/opengl_intro.html (Accessed: 19 October 2024).

Apple Inc (2024) *Metal Overview*. Available at: https://developer.apple.com/metal/ (Accessed: 19 October 2024).

Archer, J. (2024) ‘Mandatory ray tracing makes Indiana Jones and the Great Circle pretty yet brutal, like Indy himself’, *Rock, Paper, Shotgun* [Preprint]. Available at: https://www.rockpapershotgun.com/indiana-jones-and-the-great-circle-pc-performance-best-settings (Accessed: 16 April 2025).

Atlassian (no date) *Gantt Charts Explained [+ How to Create One]*. Available at: https://www.atlassian.com/agile/project-management/gantt-chart (Accessed: 15 April 2025).

Beck, K. *et al.* (2001) *Manifesto for Agile Software Development*. Available at: http://agilemanifesto.org/ (Accessed: 15 April 2025).

Bentebent (2024) *rita*. Available at: https://github.com/Bentebent/rita (Accessed: 30 March 2025).

Blinn, J.F. (1977) ‘Models of light reflection for computer synthesized pictures’, *SIGGRAPH Comput. Graph.*, 11(2), pp. 192–198. Available at: https://doi.org/10.1145/965141.563893.

Bono, S. *et al.* (2009) ‘Reducing the Attack Surface in Massively Multiplayer Online Role-Playing Games’, *IEEE Security & Privacy*, 7(3), pp. 13–19. Available at: https://doi.org/10.1109/MSP.2009.75.

Botsford, D. (2017) *How to Use GitHub’s New “Projects” Boards*. Available at: https://medium.com/@dawsonbotsford/how-to-use-github-projects-aa15a8411b72 (Accessed: 15 April 2025).

Carmen Cincotti (2022) *Homogeneous Coordinates, Clip Space, and NDC | WebGPU*. Available at: https://carmencincotti.com/2022-05-02/homogeneous-coordinates-clip-space-ndc/#the-projection-transformation (Accessed: 31 March 2025).

Desires, A. *et al.* (2025) *The Rustonomicon*. Available at: https://doc.rust-lang.org/nomicon/other-reprs.html#reprpacked-reprpackedn (Accessed: 16 April 2025).

Dunn, F. and Parbery, I. (2025) *3D Math Primer for Graphics and Game Development*. Available at: https://gamemath.com/ (Accessed: 5 April 2025).

Eauvidoum, I. and disk noise (2021) *Twenty years of Escaping the Java Sandbox*. Available at: http://phrack.org/issues/70/7.html#article (Accessed: 13 November 2024).

Epic Games (2021) *Why cross-play matters*. Available at: https://onlineservices.epicgames.com/en-US/news/why-cross-play-matters (Accessed: 7 November 2024).

erer1243 (no date) *wgpu v0.20 + winit v0.30 "hello triangle" on the web and native*. Available at: https://github.com/erer1243/wgpu-0.20-winit-0.30-web-example (Accessed: 30 March 2025).

Ethan Lee (2024) *FNA's GPU API for SDL3: Now in Alpha!*. Available at: https://icculus.org/finger/flibitijibibo?date=2024-06-15&time=13-14-16.

Faessler, F. (2019) *Minetest (Hardware) - Google CTF Quals 2019*. Available at: https://liveoverflow.com/minetest/ (Accessed: 7 November 2024).

fewturns (2022) *Minecraft lagging on a high-end pc :: Hardware and Operating Systems*. Available at: https://steamcommunity.com/discussions/forum/11/3278065083969030484/ (Accessed: 13 November 2024).

Hansen, B. (2024) *Learn Wgpu*. Available at: https://sotrh.github.io/learn-wgpu/ (Accessed: 19 October 2024).

Hess, E.H. (1961) 'Shadows and depth perception', *Scientific American*, 204(3), pp. 138–151. Available at: http://www.jstor.org/stable/24937397 (Accessed: 16 April 2025).

iconicNurdle (2023a) *Getting Started With Minecraft Add-Ons*. Available at: https://learn.microsoft.com/en-us/minecraft/creator/documents/gettingstarted?view=minecraft-bedrock-stable (Accessed: 6 November 2024).

iconicNurdle (2023b) *Introduction to Scripting*. Available at: https://learn.microsoft.com/en-us/minecraft/creator/documents/scriptingintroduction?view=minecraft-bedrock-stable (Accessed: 6 November 2024).

Ignaci, A. (2020) *Ray Tracing Support · Issue #1040 · gfx-rs/wgpu*. Available at: https://github.com/gfx-rs/wgpu/issues/1040 (Accessed: 16 April 2025).

Iliev, H. (2023) *OpenGL vs Vulkan*. Available at: https://thatonegamedev.com/cpp/opengl-vs-vulkan/ (Accessed: 7 November 2024).

Imagination Technologies (2018) *Blog post: The Benefits of Vulkan - Imagination*. Available at: https://blog.imaginationtech.com/stuck-on-opengl-es-time-to-move-on-why-vulkan-is-the-future-of-graphics/ (Accessed: 19 October 2024).

JetBrains (2024) *Fernflower*. Available at: https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine (Accessed: 13 November 2024).

John Linneman (2018) 'SSX 3 is a retro masterpiece - and it's even better on Xbox One', *Eurogamer* [Preprint]. Available at: https://www.eurogamer.net/digitalfoundry-2018-ssx-is-a-retro-masterpiece-and-its-brilliant-on-xbox-one (Accessed: 5 April 2025).

Justine Tunney (2017) *Operation Rosehub*. Available at: https://opensource.googleblog.com/2017/03/operation-rosehub.html (Accessed: 5 November 2024).

Laird, J. (2023) 'Nvidia's RTX 3060 is the new Steam Survey king but something fishy is going on', *PC Gamer* [Preprint]. Available at: https://www.pcgamer.com/nvidias-rtx-3060-is-the-new-steam-survey-king-but-something-fishy-is-going-on/ (Accessed: 16 April 2025).

Luanti Developers (2024) *Minetest for Education - Minetest*. Available at: https://www.minetest.net/education/ (Accessed: 7 November 2024).

LWJGL Developers (2024) *LWJGL - Lightweight Java Game Library*. Available at: https://www.lwjgl.org/ (Accessed: 19 October 2024).

Manero, M.O. (2021) *Plugins in Rust: Getting Started*. Available at: https://nullderef.com/blog/plugin-start/ (Accessed: 6 November 2024).

McCluskey, G. (1998) *Using Java Reflection*. Available at: https://www.oracle.com/technical-resources/articles/java/javareflection.html (Accessed: 13 November 2024).

MDN (2025a) *Promise - JavaScript*. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (Accessed: 15 April 2025).

MDN (2025b) *Using HTTP cookies*. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies (Accessed: 6 April 2025).

michael314 (2018) *Rendering Performance - Luanti Forums*. Available at: https://forum.luanti.org/viewtopic.php?t=19412 (Accessed: 13 November 2024).

Michaud, S. (2017) *The Khronos Group Releases OpenGL 4.6 - PC Perspective*. Available at: https://pcper.com/2017/08/the-khronos-group-releases-opengl-4-6/ (Accessed: 11 November 2024).

Mojang (2017) *Better Together FAQ*. Available at: https://www.minecraft.net/en-us/article/better-together-faq (Accessed: 13 November 2024).

Mojang (2019) *Render dragon and NVIDIA ray tracing*. Available at: https://www.minecraft.net/en-us/article/render-dragon-and-nvidia-ray-tracing (Accessed: 19 October 2024).

Mojang (2024) *Minecraft attributions*. Available at: https://www.minecraft.net/en-us/attribution (Accessed: 12 November 2024).

Mumfrey (2017) *Flippin' Mixins, how do they work? · SpongePowered/Mixin Wiki*. Available at: https://github.com/SpongePowered/Mixin/wiki/Flippin'-Mixins,-how-do-they-work%3F (Accessed: 5 November 2024).

nikitablack (2021) *How to properly write to a buffer every frame? · gfx-rs/wgpu · Discussion #1438*. Available at: https://github.com/gfx-rs/wgpu/discussions/1438 (Accessed: 16 April 2025).

Nintendo (no date) *Technical data: Nintendo GameCube*. Available at: https://www.nintendo.com/en-gb/Support/Nintendo-GameCube/Product-Information/Technical-data/Technical-data-619165.html (Accessed: 5 April 2025).

Nordeus, E. (2013) *How to generate random terrain*. Available at: https://blog.habrador.com/2013/02/how-to-generate-random-terrain.html (Accessed: 16 April 2025).

Nvidia (2024) *DirectX*. Available at: https://developer.nvidia.com/directx (Accessed: 13 November 2024).

Parrish, A. (2023) *Minecraft has sold over 300 million copies - The Verge*. Available at: https://web.archive.org/web/20231015232557/https://www.theverge.com/2023/10/15/23916349/minecraft-mojang-sold-300-million-copies-live-2023 (Accessed: 13 November 2024).

Prospector Dev and Jai (2024) *Malware Discovery Disclosure: "Windows Borderless" mod*. Available at: https://blog.modrinth.com/p/windows-borderless-malware-disclosure (Accessed: 5 November 2024).

quat (2023) *The Treadmill - Highly Suspect Agency*. Available at: https://highlysuspect.agency/posts/the_treadmill/ (Accessed: 13 November 2024).

Radigan, D. (2025) *Kanban — How the kanban methodology applies to software development*. Available at: https://www.atlassian.com/agile/kanban (Accessed: 15 April 2025).

RandomName8 (2023) *I can think of a few…*.

Rose, J. (2021) *Terrain generation in Minecraft*. Available at: https://blog.nornagon.net/terrain-generation-in-minecraft/ (Accessed: 16 April 2025).

Rust Foundation (2024) *Rust Programming Language*. Available at: https://www.rust-lang.org/ (Accessed: 11 November 2024).

Rust Graphics Mages (2024) *wgpu: portable graphics library for Rust*. Available at: https://wgpu.rs/ (Accessed: 19 October 2024).

Rust Graphics Mages (2025) *wgpu-types Documentation*. Available at: https://docs.rs/wgpu-types/latest/wgpu_types/struct.Features.html#associatedconstant.MAPPABLE_PRIMARY_BUFFERS (Accessed: 16 April 2025).

rzwitserloot (2021) *Just about everythin…*.

Sam Lantinga (2014) *[SDL] Announcing SDL 2.0.0*. Available at: http://web.archive.org/web/20140130104245/http://lists.libsdl.org/pipermail/sdl-libsdl.org/2013-August/089854.html (Accessed: 5 November 2024).

SDL2 Developers (2024) 'SDL2'.

Serializationisbad Developers (2023) *Unsafe Deserialization Vulnerability in Many Minecraft Mods*. Available at: https://github.com/dogboy21/serializationisbad/blob/master/README.md (Accessed: 5 November 2024).

Shirley, P., Black, T.D. and Hollasch, S. (2024) *Ray Tracing in One Weekend*. Available at: https://raytracing.github.io/books/RayTracingInOneWeekend.html (Accessed: 16 April 2025).

SQLite (2025) *35% Faster Than The Filesystem*. Available at: https://www.sqlite.org/fasterthanfs.html#approx (Accessed: 30 March 2025).

Steve Klabnik, Carol Nichols and Chris Krycho (no date) 'Shared-State Concurrency', *The Rust Programming Language*. Available at: https://doc.rust-lang.org/book/ch16-03-shared-state.html#using-mutexes-to-allow-access-to-data-from-one-thread-at-a-time (Accessed: 30 March 2025).

Taylor, M. (2022) *Fundamentals of Terrain Generation*. Carnegie Mellon University. Available at: https://www.cs.cmu.edu/~112-s23/notes/student-tp-guides/Terrain.pdf (Accessed: 16 April 2025).

TechTarget (2021) *What is a Sandbox? Definition from SearchSecurity*. Available at: https://www.techtarget.com/searchsecurity/definition/sandbox (Accessed: 13 November 2024).

Torvalds, L. (2016) *Linux kernel coding style — The Linux Kernel documentation*. Available at: https://www.kernel.org/doc/html/v4.10/process/coding-style.html (Accessed: 5 April 2025).

Trofymchuk, L. (2023) *WebGPU performance —is it what we expect?*. Available at: https://medium.com/source-true/webgpu-performance-is-it-what-we-expect-b1c96b1705e1.

UK Government (2016) *Agile methods: an introduction - Service Manual - GOV.UK*. Available at: https://www.gov.uk/service-manual/agile-delivery/agile-methodologies (Accessed: 15 April 2025).

Ward, A. (2024) *Luanti Modding Book*. Available at: https://rubenwardy.com/minetest_modding_book/en/quality/security.html (Accessed: 7 November 2024).

Ware, R. (2023) *Vulkan vs. DirectX 12: Which Should You Choose?*. Available at: https://www.howtogeek.com/884042/vulkan-vs-directx-12/ (Accessed: 19 October 2024).

Wasmer Inc. (2024) *Wasmer: Security Policy*. Available at: https://github.com/wasmerio/wasmer/security (Accessed: 6 November 2024).

WebAssembly Developers (2024) *I want to… - WebAssembly*. Available at: https://webassembly.org/getting-started/developers-guide/ (Accessed: 13 November 2024).

West, C. (2018) *Using Rust For Game Development*. RustConf. Available at: https://www.youtube.com/watch?v=aKLntZcp27M (Accessed: 13 November 2024).

wgpu-rs/examples/shadow/shader.wgsl at master · gfx-rs/wgpu-rs (no date). Available at: https://github.com/gfx-rs/wgpu-rs/blob/master/examples/shadow/shader.wgsl (Accessed: 31 March 2025).

WHATWG Community (2025) *HTML Living Standard*. Available at: https://html.spec.whatwg.org/ (Accessed: 6 April 2025).

Wolff, L.B. (1994) 'Relative brightness of specular and diffuse reflection', *Optical Engineering*, 33(1), pp. 285–293. Available at: https://doi.org/10.1117/12.149144.

World Wide Web Consortium (2024) *WebGPU*. Available at: https://www.w3.org/TR/webgpu/ (Accessed: 19 October 2024).

World Wide Web Consortium (2025) 'WebAssembly Core Specification'.

Zhao, R. (no date) *Computer-automated shadow moiré method and applications for 3D surface profiling*.

Zucconi, A. (2022) *The World Generation of Minecraft*. Available at: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/ (Accessed: 16 April 2025).