Master thesis : Minecraft Clone in Rust and Vulkan

Auteur : Leclipteur, Henry
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/22449

UNIVERSITE DE LIEGE
FACULTE DES SCIENCES APPLIQUEES

Minecraft Clone in Rust and Vulkan

Leclipteur Henry

Thesis presented to obtain the degree of :
Master in Computer Science, professional focus in computer systems security

Thesis supervisor :
Mathy Laurent

Academic year: **2024 - 2025**

Acknowledgments

I would like to express my heartfelt gratitude to Professor Laurent Mathy for granting me the opportunity to work on this exceptional project, enabling me to present my thesis in January, and recognizing my efforts throughout this journey.
My sincere thanks also go to Benoît Knott, assistant to Professor Mathy, for his invaluable dedication in testing my game and providing constructive feedback that significantly enhanced the quality of my work.
I am deeply grateful to my family, especially my mother, for their unwavering support, encouragement, and belief in me throughout the course of this project.
A special thanks goes to my girlfriend, Marie, for her steadfast support, understanding, and motivation, which were vital during this process.
Lastly, I extend my appreciation to my friend Alexandre for his insightful discussions, which offered me fresh perspectives, and for his constant encouragement, which was a source of great inspiration.

Abstract

This master's thesis describes the development of a Minecraft clone using the Rust programming language and the Vulkan graphics API. The primary objective was to establish a foundational framework for a voxel-based game, allowing students to expand upon it for their personal projects or master's thesis. My personal goal was to explore and understand the various mechanisms involved in game development, ranging from 3D rendering to game physics.

The decision to use Rust and Vulkan was guided by my supervisor, Professor Laurent Mathy. Rust offers significant advantages in game development, particularly in its robust handling of memory safety, which eliminates many of the most challenging bugs associated with memory management. Vulkan, as a low-level graphics API, provided valuable insight into the intricacies of the rendering pipeline, as it requires explicit control over all stages of the process.

The project culminated in a functional game composed of four distinct modules:

1. **Renderer**: The rendering system built in rust and using the Vulkan API for efficient graphics processing.
2. **Game core**: Responsible for modeling the game world and mechanics.
3. **Main**: Coordinates the overall game execution.
4. **Draw element**: Facilitates interaction between crates to maintain modularity.

The most significant challenges of this project was developing the rendering system using Vulkan. As a low-level graphics API, Vulkan requires precise specification of every aspect of the rendering process, presenting a steep learning curve for beginners. Despite its complexity, I successfully implemented a rendering system capable of rendering up to 225 chunks around the player while maintaining a stable framerate of 144 FPS.
Achieving this performance required multiple optimizations, including minimizing delays during chunk loading and implementing a greedy meshing algorithm to reduce rendering overhead.

The resulting Minecraft clone, while rudimentary, is fully functional. Players can navigate a procedurally generated world and interact with the environment by placing and removing blocks. However, the current implementation is in its early stages and lacks several advanced features such as animations, multiple entities, and multiplayer functionality.

Future improvements will focus on further optimizing the rendering system and expanding the game's feature set with additions such as sound, crafting mechanics, animations, and improved user interfaces. This project serves as a foundational resource for anyone interested in learning about voxel-based game development.
The code and documentation are available on my GitLab, where they are presented in a clear and easily understandable manner. This provides students and developers with a straightforward framework for exploring game design and graphics programming.

Contents

List of Figures

List of Tables

Part I
Introduction

1 Context

1.1 Original game

Minecraft, developed by Mojang Studios and initially released in 2011, is a sandbox game that has become one of the best-selling video games of all time. The game offers players a procedurally generated, voxel-based 3D world where they can explore, mine resources, craft tools, and build structures. Its core mechanics revolve around breaking and placing blocks, enabling creative expression, survival challenges, and exploration.

The game features two primary modes: Creative Mode, where players have unlimited resources and the freedom to build without constraints, and Survival Mode, where players must gather resources, manage health, and fend off hostile creatures. Additionally, Minecraft's open-ended gameplay is complemented by multiplayer features, a thriving modding community, and educational applications.

Minecraft's technical foundation includes efficient chunk-based terrain generation, procedural algorithms for creating diverse landscapes, and a modular system for managing game components. This design has inspired many clones and adaptations, making it a prime example of combining simple mechanics with profound depth and scalability.

1.2 Existing copies

Numerous Minecraft clones already exist in nearly every programming language, making them excellent resources for learning about game engine development, especially for voxel-based games. Many content creators have documented their development process in insightful videos, which have proven to be valuable sources of inspiration.

Notable examples include `I Remade Minecraft But It's Optimized` by FinalForEach [3], and `How to Make Minecraft in C++ or Any Other Language` by Low Level Game Dev [4].

2 Objectives and Background

2.1 Project objective

The concept of creating a Minecraft clone was proposed by Professor Laurent Mathy.
His vision was to establish a platform for collaborative student projects, enabling students to work on innovative initiatives as part of their personal projects or master's thesis.
As part of this initiative, he envisioned a series of Minecraft clones that students could iteratively enhance by introducing new features and optimizations.
Currently, there are two versions of the clone in development: one developed in C++ utilizing the OpenGL rendering API, and another built in Rust, leveraging the Vulkan rendering API.
I was tasked with developing the Minecraft clone in Rust and Vulkan, laying the groundwork for future improvements and contributions.

The primary objective of this master's thesis is to create a solid, extensible framework for the Minecraft clone in Rust and Vulkan. This framework should include core functionalities, such as basic rendering, block-based world mechanics, and modular extensibility, while maintaining clear and comprehensive documentation. By ensuring accessibility and clarity, this project seeks to empower future students to understand, modify, and build upon this work, driving continued innovation in the field.

2.2 Personal background

Minecraft is a game that has been a significant part of my experience, as it has been for many others.
Spending considerable time playing it sparked my curiosity about the intricacies of game development.
I have previously created several small games, including a Geometry Dash-inspired game in Python, a Temple Run-like game in C++ during my university studies, and a physics simulation in Python as a personal project.
Prior to undertaking this project, I had no experience with Rust, 3D graphics programming, or working with the Vulkan low-level API. However, the challenge of learning new skills and exploring these uncharted areas was incredibly motivating, ultimately driving me to pursue this ambitious endeavor.

3 Game Presentation

I developed a simple Minecraft-like game that allows the player to explore an infinite voxel-based world, focusing on the core mechanics of a first-person exploration game. The implementation does not include a menu system.

The implementation details and source code are available in my GitLab repository [5]. A screenshot of the game is shown in Figure 3.1.

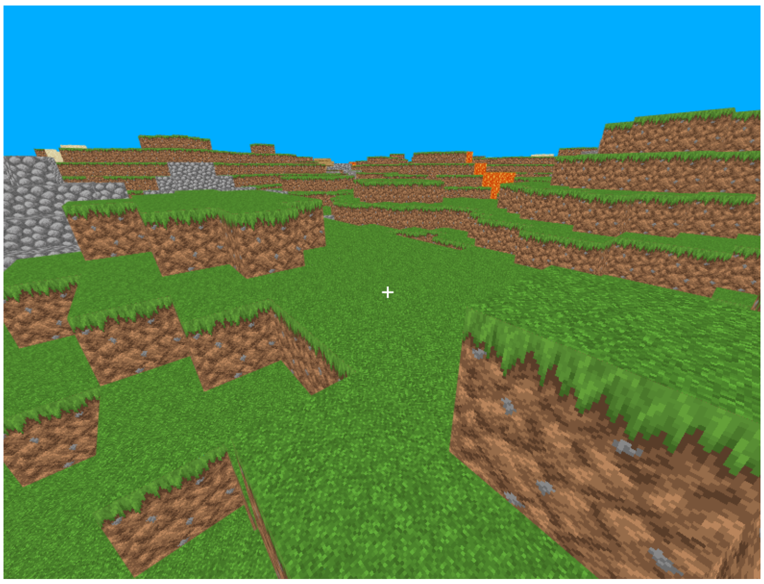


Figure 3.1: Minecraft clone gameplay screenshot

3.1 Controls

The gameplay allows for basic movement and interaction with the environment.
The controls are as follows:

- Use the **ZQSD** keys to move around.
- The **mouse movements** enables the you to look around and control the camera.
- Press the **space bar** to jump.
- Use the **right mouse button** to place a block and the **left mouse button** to remove a block at the position of the cursor.
- Press the **esc** key to quit the game and save the progression.

Part II
Architecture

To ensure modularity, the game is divided into four distinct crates.
In Rust, a crate is a modular compilation unit, similar to a library in other programming languages, except for the `main` crate, which contains the main function.
The four crates are:

- **main:** Contains the main loop that drives the game.
- **draw_element:** Defines a communication format used by the `renderer` and `game_core` crates.
- **renderer:** Implements the rendering system.
- **game_core:** Implements the model and logic of the game.

The two principal crates are `renderer` and `game_core`. Their roles and the algorithms they implement are explained in detail in Section 4 and Section 5, respectively. The remaining crates, `main` and `draw_element`, are discussed in Section 6.

4 Core Game logic

The core game logic resides within the `game_core` crate.

The main responsibilities of the `game_core` crate include: updating the position and speed of entities, detecting and resolving collisions, loading and saving chunks, handling user input, and calculating the visible faces of blocks.
Essentially, it encapsulates the core gameplay mechanics and logic. This design choice facilitates rapid prototyping, as it allows for easy refactoring and the implementation of quick improvements.

All calculations in the `game_core` crate are performed using relative coordinates, which represent positions within a chunk.
The coordinate ranges are $[0, 16[$ for X, $[0, 512[$ for Y, and $[0, 16[$ for Z.
By operating in relative coordinates, the system minimizes floating-point errors, which can arise when floating-point values become excessively large. This issue is particularly relevant in games like Minecraft, where the coordinates of blocks can grow very large in an infinite world, leading to precision loss.

The `game_core` crate defines a structure that models the game state, allowing function calls to notify the system of user inputs and to update the state based on elapsed time and received inputs. The game state is updated through the following sequence of actions:

- **Handle chunk loading and unloading:** This step involves a non-blocking receive to check if any chunks have completed loading or unloading in other threads, ensuring that the game can handle large areas of the world efficiently.
- **Update the view angle:** The player's view angle is adjusted based on mouse movement.
- **Set the obstacle target:** Using the ray tracing algorithm for voxel games [6], this step identifies the nearest obstacle in the player's line of sight. This information is crucial for interactions with the world, such as placing or breaking blocks.
- **Update the player's speed:** The player's speed is adjusted according to in-game factors, including movement or falling.
- **Update the player's position:** The player's position is recalculated based on its speed, taking into account interactions with the environment. This step involves the collision system, which ensures the player moves realistically and does not pass through solid objects. Additionally, if the player changes chunks, threads are launched to handle the unloading and loading of neighboring chunks.
- **Return the `SurfaceElements`:** At the end of the update process, some `SurfaceElements` are returned. These specialized elements are used for exchanging formalized data between the renderer and the `game_core` crate. The `SurfaceElement` structure is described in Section 6.1.

4.1 The collision system

In the Minecraft Clone, all blocks, obstacles, and entities are defined using Axis-Aligned Bounding Boxes (AABB). The player is the only entity capable of movement, so the collision system focuses exclusively on the moving player.

A collision system has two primary tasks: collision detection and collision resolution. While some algorithms excel at collision detection, they may not contribute effectively to collision resolution, which is equally important.
My main objective was to design a robust collision system that could be generalized to handle all entities in the future.

4.1.1 Tunnelling

Tunneling is a phenomenon where an entity, under certain conditions, can pass through an obstacle. This occurs when the collision detection system fails to identify the collision.

For instance, consider a simple collision detection system: We check if, after moving, the player's Axis-Aligned Bounding Box (AABB) overlaps with the AABBs of nearby obstacles. This approach detects collisions a posteriori, after they occured. While this method generally works well, it can fail in specific scenarios, such as:

- When the player moves at very high speeds and the obstacle is very thin.
- When the time interval between two updates is unusually long.

The issue arises because the player's position update is too large, causing it to 'skip over' the obstacle during the time step.

Tunneling can be prevented by either discretizing the update or employing a priori collision detection.

Discretizing the Update

This method involves breaking a single update into smaller sub-updates, allowing the system to detect collisions more effectively. However, this approach requires defining thresholds for the maximum distance traveled per update and the minimum thickness of obstacles.

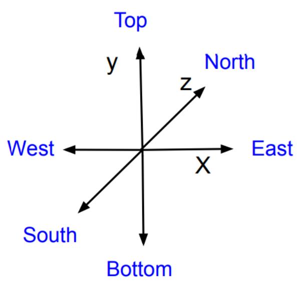
This creates a trade-off: accommodating very thin obstacles and small distances per update increases the number of sub-updates, which can become computationally intensive.

A Priori Collision Detection

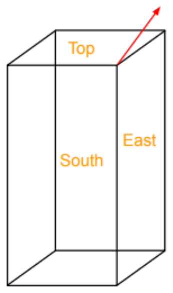
Alternatively, a method that predicts collisions before they occur, such as ray tracing or swept volumes, can be used. These techniques are proactive, examining the path of movement to determine if a collision will happen. Since this game uses AABBs, ray tracing is not applicable, but swept volumes can serve as an effective solution.

A swept volume is the 3D space occupied by the player as it moves during an update. It is created by translating the player's AABB along the velocity vector. This results in a new shape, a polyhedron of six to twelve faces depending on the direction of the velocity vector. This polyhedron encompasses the player's movement. The collision detection system then tests this polyhedron against all obstacles to check for intersections.

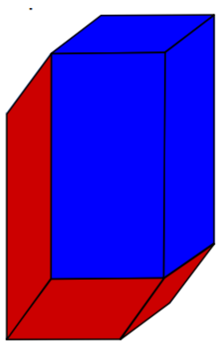
As an example you can see in Figure 4.1b the AABB of the player and in Figure 4.1c the volume created by its velocity vector.
My minecraft clone uses the face convention described in Figure 4.1a.



(a) Face convention



(b) Player AABB



(c) Volume created

Figure 4.1: Swept volume example

4.1.2 Strategy discussion

Method choice

I aimed to design a generalized collision system applicable to all entities with minimal constraints.
In developing the system, I considered the two approaches: discretizing updates and using the swept volumes strategy.

Collision detection refers to identifying whether, in a given game state, two Axis-Aligned Bounding Boxes (AABB) overlap. In this Minecraft clone, collision detection focuses on checking the player's AABB against nearby obstacle AABBs.
For that purpose, both methods are functionally equivalent for collision detection. However, discretizing updates has a significant drawback: it imposes constraints on the minimum thickness of obstacles and the maximum distance an entity can travel in one update.

Collision resolution involves determining the appropriate response to a collision, such as repositioning the player or defining the post-collision velocities of both the player and the obstacle. Both methods can easily calculate the penetration depth along the three axes, which is useful for approximating the collision point during resolution. However, in the Minecraft clone, relying solely on penetration depth proved to be problematic.

Consider the 2D example in Figure 4.2:
When the player falls onto an obstacle from above, the expected behavior is for the player to stop on top of the obstacle. However, using only penetration depth and resolving the collision along the axis of least penetration depth, the player would be pushed sideways to the left edge of the obstacle instead of coming to rest on top. This issue frequently arises in Minecraft, especially when walking along a wall or dropping onto a block.

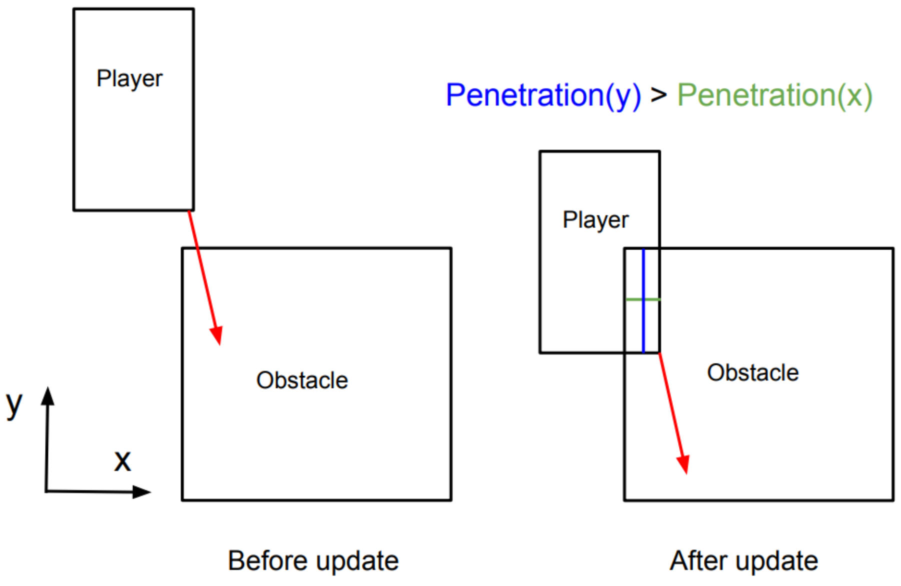


Figure 4.2: Penetration depth resolution example

To ensure robustness, I decided to calculate which face of the player collided with the obstacle. This additional information is crucial for accurate and flexible collision resolution.

Ultimately, I decided to use the swept volumes approach for my collision detection system to avoid subupdates that could lead to a loss of performance during high-speed scenarios. Choosing this approach required selecting a specific algorithm to implement. In the following section, I will discuss the algorithm selection process and explain how to calculate the collision face, a crucial element for ensuring accurate and robust collision resolution.

Algorithms

One possible approach for building a collision detection algorithm using the swept volume method is to apply the SAT (Separating Axis Theorem). This theorem states:
Two closed convex objects are disjoint if there exists a line ("separating axis") onto which the two objects' projections are disjoint. [7] .

There are many implementations of this theorem available online [8].

The advantage of using SAT lies in its ability to generalize easily to more complex shapes. However, it has a notable limitation: it cannot calculate the exact point of collision. Instead, it only provides the penetration depth along the three axes. As discussed earlier, this information alone is insufficient for building a robust collision system.
Faced with this limitation, I considered two options: develop a new, specific algorithm to calculate the exact position of the collision or rethink the way collisions are detected entirely.

My final solution is an innovative algorithm that not only checks for collisions but also identifies the face of the player's AABB that collided and determines the exact position of the collision.
I called it *Face casting*.

It works by considering 3 differents 6 faces polyhedron formed by the three face in the direction of the velocity and the velocity vector itself.
An example is shown in Figure 4.3, it is the six-face polyhedron formed by the player top face from the example in Figure 4.1b.

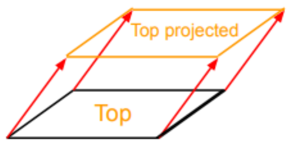


Figure 4.3: Projection of the top face

For each six-face polyhedron generated, the obstacles are sorted by the position of their opposing face. In the case of the top polyhedron shown in Figure 4.3, obstacles are sorted in ascending order based on the position of their bottom face, as this is the face that might collide with the player AABB.

Note that if the velocity vector were pointing downward, there would be no top polyhedron, but instead, a bottom polyhedron, and the obstacles would be sorted in descending order by the position of their top face.

Once sorted, the obstacles are checked one by one by projecting the player's face (top face in the example) to the plane of the obstacle face (bottom face in the example) along the velocity vector. The projected face is then compared to the obstacle face.
If the two faces overlap, a collision is detected. The distance along the velocity vector from the initial

face to the obstacle face is calculated and returned, along with the face of the player (top face in the example). If there is no overlap, the algorithm continues to check the next obstacle.

After performing the tests on all three polyhedra, we obtain three distances and three corresponding faces. The smallest distance is selected, providing the necessary information to resolve the collision.

In this game, the collision resolution is simple: we stop the player from entering the obstacle and set the velocity along the direction of the face to zero.

Limitations

The Face Casting algorithm has some limitations.
One issue is that the obstacles must be sorted, which can become problematic when there is a large number of obstacles.
Additionally, in the current version of the game, too many obstacles are tested. All obstacles within the AABB of the swept volume in Figure 4.1c, are checked for collision.
Nevertheless, the collision system remains very fast, and there are no noticeable performance issues.

A potential problem could arise if we consider multiple moving entities. Currently, the system treats all obstacles as static, making collision checks relatively simple. However, this becomes more complicated when several entities are moving simultaneously as we would check collision not only on simple AABB but also on other swept volumes.

One possible solution could be to consider all moving entities as static and check and resolve collisions starting with the fastest entity and then proceed to the slower ones. This would help prevent slow-moving entities from colliding with faster ones that should have already moved past them.
It is still an approximation as a fast entity could go through a slower one because the game would consider that the slower one hasn't move when calculating the collision of the fast entity.

4.2 Chunk storage and generation

In the Minecraft Clone, chunks are structured similarly to those in the original game, with the terrain divided into chunks along the x and z axes. For the Minecraft Clone, I have chosen the dimensions of a chunk to be 16x512x16 blocks, extending from Y=0 to Y=511. In contrast, the original game uses chunks of 16x384x16 blocks.

4.2.1 Chunk generation

In the original Minecraft, terrain generation is a sophisticated process driven by multiple layers of Perlin noise, each serving a distinct purpose: [9]

- Low Noise: Responsible for creating smooth and rolling terrain, ideal for gradual elevations and valleys.
- High Noise: Produces rough, jagged surfaces, adding rugged features to the landscape.
- Selector Noise: Acts as a decision-maker, determining whether a given area uses low or high noise, resulting in diverse and dynamic terrain.

This multi-layered approach enables the original game to create highly detailed and varied environments, contributing significantly to its immersive experience.

In contrast, the terrain generation in the Minecraft clone is significantly simplified. It relies on a single Perlin noise function to determine the maximum height of the terrain at each point. Blocks are then filled beneath this maximum height, creating a solid landscape. While efficient, this method lacks the complexity and richness of the original game's terrain.

To achieve terrain generation similar to the original game, the next step would be to integrate multiple Perlin noise layers. However, before this enhancement can be implemented, an essential prerequisite is optimizing the rendering system.

Jagged terrain generated by high noise leads to a significant increase in visible faces, which in turn demands more rendering power. This effect is shown in Section 7.4.
To address this, the algorithm in the `game_core` crate responsible for calculating rectangles from visible faces must be improved (Section 4.3). Currently, this algorithm calculates individual lines instead of larger rectangles, leading to inefficiencies. Optimizing this algorithm to group visible faces into rectangles will significantly reduce rendering overhead.

Additionally, the rendering system as a whole requires further optimization to handle the increased complexity of jagged terrain more efficiently. Once these improvements are implemented, the clone will be better equipped to adopt the advanced terrain generation techniques of the original game.

4.2.2 Chunk saving

In the original game, chunks are saved to disk when they are unloaded, which occurs when the player moves far away from a chunk or exits the game. These chunks are stored in region files [10], with each file containing a 32x32 grid of chunks, collectively known as a region.

The game uses a compressed Anvil format [11] to store chunk data efficiently. This format organizes chunks into smaller subsections (16x16x16 blocks) to optimize storage and retrieval. Rather than saving all blocks within a chunk, the system only records changes from the base seed, relying on procedural generation to recreate unmodified terrain dynamically.

In the Minecraft Clone, chunks are also saved when unloaded, but unlike the original, every block in each chunk is saved, with one file per chunk. This approach is less efficient and would require optimization to improve storage and performance.
To avoid delays during chunk loading or saving, these operations are performed asynchronously by separate threads. This ensures a smooth player experience while chunks are processed in the background.

4.3 Visible face optimization

Efficient rendering of the Minecraft world requires to display only the visible faces of blocks. A face is considered visible if it is not adjacent to another block; otherwise, it is occluded and does not need to be rendered.
This approach minimizes unnecessary rendering, but further optimizations are necessary when dealing with large numbers of chunks.

4.3.1 Greedy meshing and algorithms

Greedy meshing [12] is a technique used to optimize rendering by grouping adjacent, similar block faces into larger surfaces. By rendering these grouped faces as single entities, the number of vertices processed by the rendering system is significantly reduced, improving performance.

Implementing an optimal algorithm for this task is not straightforward.
While finding the exact solution is challenging, I explored two heuristic approaches:

- **Maximum Rectangle Removal:** This method identifies the largest possible rectangle of contiguous, similar faces, removes it, and repeats the process until all visible faces are grouped [13].
- **Greedy Expansion:** Starting from a random visible face, this method expands in both directions to form the largest possible rectangle, repeating this process until all faces are grouped.

These heuristics are conceptually simple, but they become more complex when applied to dynamic updates. For example, adding or removing blocks requires recalculating both the visible faces and their groupings to ensure that the meshing remains accurate.

4.3.2 Current implementation

In the current implementation of the Minecraft clone, a simplified version of the greedy expansion heuristic is applied. This method groups faces into lines by extending rectangles in two directions, reducing the number of rendered faces by a factor of four, from 1800 to 450 faces per chunk on average (with both Perlin constants set to 10).

Further optimization could be achieved by grouping faces into rectangular regions instead of just lines. This enhancement would likely reduce the number of vertices even further, allowing for the simultaneous rendering of additional chunks.

Another limitation in the current implementation is that all faces at the edges of chunks are marked as visible, even if adjacent chunks contain blocks that would occlude them.
This can be improved by marking those faces as non-visible when the adjacent chunk is loaded.

4.3.3 Fragmentation

A key issue arises from the fragmentation of face groups when blocks are frequently modified. Over time, this can lead to suboptimal face groupings, reducing the effectiveness of the greedy meshing technique.

A potential solution to this problem is to recompute the greedy meshing whenever a chunk is loaded or after a certain number of block modifications, ensuring that the face groups are always optimized.
This approach is not yet implemented in the Minecraft Clone, as the meshing is currently only calculated during chunk generation and reused when the chunk is loaded.

4.4 Batching surface elements

Each time a chunk is loaded, unloaded, or when a block is added or removed from the world, `SurfaceElements` are generated. These elements represent the face groupings produced by the greedy meshing algorithm.
A `SurfaceElement` is a structure defined in the `draw_element` crate, and it serves as a shared format to reduce translation overhead in the rendering system.
When multiple chunks are loaded simultaneously, a significant number of `SurfaceElements` can be created.
This process can have a notable impact on game performance, as the rendering system is required to buffer all these elements and write them to the vertex buffer.

To address this issue, the `game_core` crate batches the `SurfaceElements` and returns a single batch per update call. This approach reduces the workload on the rendering system within a single frame, minimizing lag. However, it comes at the cost of chunk loading and unloading not being completed instantly.

5 Rendering System

I spent the majority of the development time working on the rendering system, which is implemented in the `renderer` crate. The primary responsibility of this crate is to draw elements onto a window created by the main crate.

Currently, the rendering system uses absolute coordinates, which is not very efficient. This approach can lead to floating-point errors when dealing with large world coordinates, and I plan to address this limitation in the future.

In this section, I will summarize the purpose of each component of the rendering system and explain the key design decisions made during its development. The detailed implementation of the rendering system can be found in my GitLab repository [5].

5.1 Vulkan wrapper

Using the Vulkan API in Rust posed its own challenges. Since Vulkan was originally designed for use in C++, it requires a wrapper to integrate effectively with Rust.
Surprisingly, there are many wrappers available for the Vulkan API in Rust, each catering to different levels of abstraction and developer preferences:

- **Ash:** A thin wrapper around the Vulkan API. It provides near-complete control over Vulkan's low-level operations but requires familiarity with Vulkan's complexity. Ash is a popular choice for developers seeking maximum flexibility and minimal abstraction.
- **Boson:** A relatively new wrapper that builds upon Ash. It offers a similar low-level interface but introduces some abstractions to simplify common tasks, striking a balance between control and usability.
- **Vulkania:** Another newer option that aims to balance low-level API access with abstractions that reduce verbosity. It seeks to make Vulkan more approachable without sacrificing the ability to fine-tune performance-critical operations.
- **Vulkano:** A high-level wrapper that provides Rust-like abstractions over the Vulkan API. It retains low-level control where necessary but abstracts away much of the boilerplate, enabling developers to focus on essential functionality. Vulkano also integrates Rust's safety guarantees, making it a safer and more beginner-friendly option.

As a beginner in both Rust and Vulkan, I chose to start with Vulkano. Its higher level of abstraction and adherence to Rust's safety principles provided a more accessible entry point for learning Vulkan. Additionally, Vulkano's comprehensive feature set made it a suitable choice for this project, allowing me to focus on implementing functionality rather than dealing with Vulkan's verbosity. I continued using Vulkano for the entirety of the project due to its completeness and ease of use.

I found several valuable resources that greatly assisted in the development of the rendering system using Vulkano.
Initially, I based my system on the introductory tutorial provided by the Vulkano team [14]. In addition, I made frequent use of the comprehensive guide by taidaesal [15], which, despite being based on an older version of Vulkano, was incredibly helpful in grasping essential concepts and constructing the system.

5.2 Rendering basics

In this section, I will present the various components of the basic rendering system example provided in the Vulkano tutorial [14]. I will break down its functionality and introduce the fundamental concepts of graphics programming.
The Vulkano tutorial culminates in a fully functional program that, when executed, opens a window displaying a triangle rendered at its center.

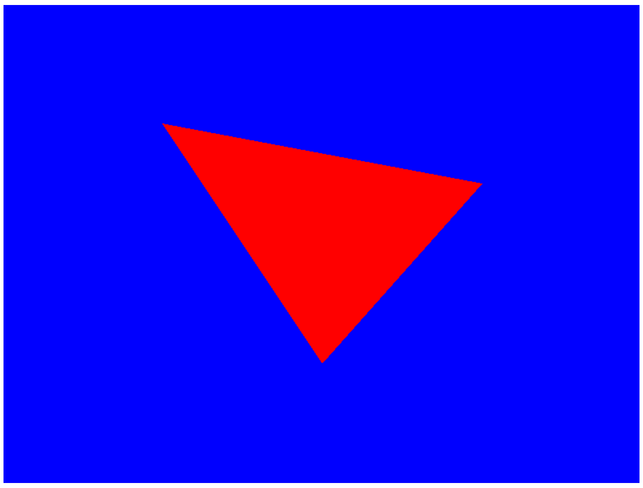


Figure 5.1: Vulkano triangle tutorial

To begin, Figure 5.2 provides a visual representation of the key graphical components discussed in the tutorial. Each component will be summarized to clarify its purpose and functionality, and its specific role in the rendering process will also be explained. Components highlighted in red represent elements that are memory-allocated on the GPU.

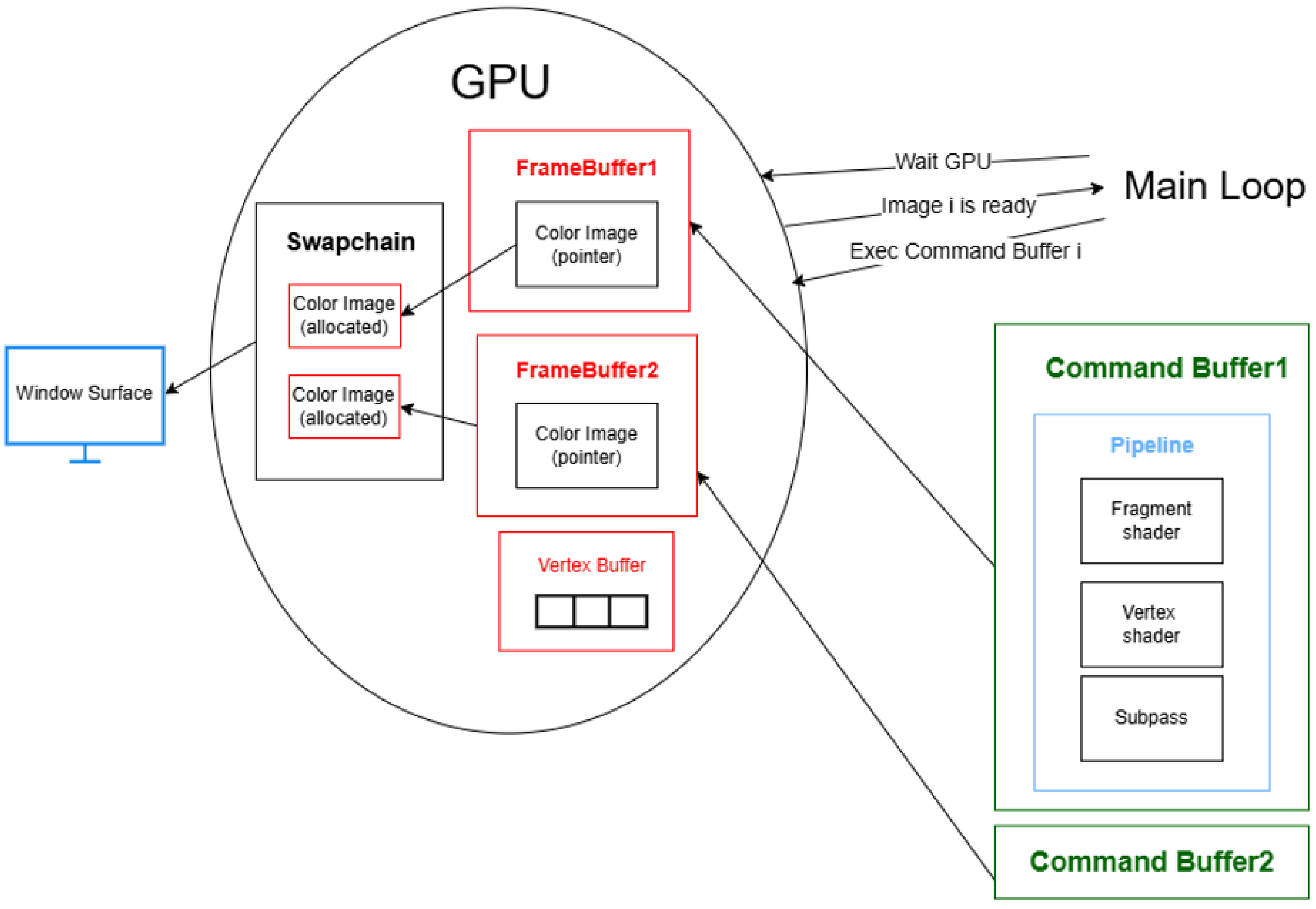


Figure 5.2: Visual representation of the vulkano tutorial

5.2.1 Rendering components

Window and Surface

The `Window` represents a standard window in the operating system and is created using the `winit` crate. This crate, recommended by the Vulkano team, also facilitates the creation of an event loop for handling user input efficiently.

The `surface` serves as an abstraction provided by Vulkano, linking the `winit` window to the Vulkan ecosystem. It acts as the interface between the operating system's windowing system and Vulkan, enabling the rendering process to target the visible screen.

Swapchain

The `Swapchain` manages a collection of color images used for rendering and presenting to the `surface`. Typically, a swapchain contains two images for double buffering, as illustrated in Figure 5.2. Double buffering (or triple buffering, or more) ensures smooth rendering and eliminates flickering in graphics. In a double-buffered setup, one image is displayed while the other is being rendered, allowing for continuous operation.

The images within a swapchain are called color images because they store RGB color data. These images are allocated directly on the GPU to support high-performance rendering.

Frame Buffers

A `Frame Buffer` is a structure that encapsulates one or more attachments used during rendering. These attachments can include:

- **Color Images**: Represent images with colors, such as RGB values. These can be the color images from the swapchain or other images.
- **Depth Images**: Represent images with a single value per pixel, indicating depth. They are used to determine the visibility of objects in 3D space.
- **Multisample Images**: Represent multisampled images, often used with color images for anti-aliasing to smooth jagged edges.

In Figure 5.2, each frame buffer contains only the color image from the swapchain. It is important to note that each swapchain image must be associated with a frame buffer to be used in rendering.

Render Pass

A `render pass` is a construct that defines the sequence of rendering operations to be performed on frame buffers. While it does not store data itself, it specifies how the frame buffer attachments are used, including how they are read from, written to, and how data transitions between different subpasses. A render pass is composed of one or more subpasses, each corresponding to a distinct stage in the rendering process.

In Figure 5.2, the render pass is not explicitly shown, as its main function is to define the sequence of operations that are executed during the rendering process, which is done through its subpasses. In the tutorial, the render pass contains a single subpass, which simplifies the setup, and this subpass is used by both pipelines.

In the tutorial, the role of the subpass is straightforward: it is defined to 'clear', initializing all pixel colors in the target color image to a specific value (dark blue in this case) before the rendering process begins. Once the rendering operation is complete, the subpass stores the resulting data in the same image. This clearing and storing procedure ensures that the image is properly initialized before rendering and that the final result is saved for display.

Vertex Buffer

A `Vertex Buffer` is a GPU-allocated buffer used to store vertex data, representing the vertices of primitives to be drawn. Vertices are typically organized in groups of three, forming triangle primitives, with each group occupying contiguous locations within the vertex buffer.

In addition to the position of each vertex, the vertex buffer can store other attributes that are specific to each vertex, such as color, normal vectors, texture coordinates (UVs), or texture indices. These attributes provide essential data for more complex rendering tasks, such as lighting calculations or texture mapping.
In the Vulkano triangle tutorial, a vertex is simply composed of 2D coordinates and the vertex buffer contains three vertex representing the three points of the triangle.

Vertex and Fragment Shaders

A `Shader` is a program designed to run on the GPU, allowing for parallel execution of thousands of tasks simultaneously. Shaders are often written in GLSL (OpenGL Shading Language), all the shaders in this project are written in GLSL.

A vertex shader is a type of shader that processes each vertex in a vertex buffer. Its main role is to apply transformations, such as rotation, translation, scaling, and projection, to each vertex. After processing, the vertex shader outputs the transformed vertex data for the next stage of the pipeline. Additionally, the vertex shader can pass computed data (such as texture coordinates, normals, or colors) to the fragment shader for further processing.
In the Vulkano triangle tutorial, the vertex shader simply outputs its position without any transformations.

The fragment shader is responsible for computing the final color of each pixel of the triangles defined by the vertex shader. It is used to determine the color and appearance of each pixel, taking into account lighting, texture, and other factors. The fragment shader can also utilize data passed from the vertex shader to perform these computations.
In the Vulkano triangle tutorial, the fragment shader simply sets all pixels to red.

Pipeline

A pipeline consists of several stages that transform vertex data into a final image ready to be displayed. While a pipeline can include many stages, I will focus on the relevant ones for the Vulkano triangle example:

- **Input Assembly Stage (IA)**: This stage groups the vertices into primitives, typically triangles (though other shapes can be used).
- **Vertex Shader Stage (VS)**: In this stage, the vertices are transformed, as explained in the previous Section.
- **Rasterization Stage (RS)**: This stage converts the primitives (e.g., triangles) into pixel fragments.
- **Fragment Shader Stage (FS)**: The fragment shader assigns color to the pixel fragments.
- **Writing Results to the Framebuffer**: The resulting pixel colors are written to the final color image in the framebuffer.

In the Vulkano triangle example, only the vertex shader, fragment shader, and subpass need to be explicitly specified to create the pipeline. The other stages are automatically included by default.
A last important component to define for the pipeline is the viewport. The viewport specifies how the final image will be mapped to the frame buffer by determining which range of coordinates will be displayed in normalized device coordinates (NDC). I will discuss coordinate transformations in the pipeline in more detail in Section 5.2.3.

Command Buffers

A `command buffer` is a structure designed to batch multiple commands for efficient execution on the GPU. By grouping commands into a single unit, command buffers minimize the CPU overhead associated with issuing individual commands.
A command buffer is defined on the CPU side and is meant to be sent to the GPU for execution.
Commands recorded in a command buffer can include:

- **Graphical Commands**: Such as binding pipelines, setting vertex buffers, and issuing draw calls.

- **Transfer Commands**: For copying data between buffers or images to or from the GPU.
- **Compute Commands**: To perform operations in compute pipelines.

In the Vulkano triangle example, two command buffers are created, each containing a single graphical command. Both command buffers use the same graphical pipeline and vertex buffer but target different framebuffers for the output image. This approach enables double buffering, ensuring smooth rendering without flickering by allowing the GPU to render one image while the other is displayed.

The command buffers used in the example are reused multiple times to save the CPU from the overhead of repeatedly building new command buffers. However, it is also possible to create one-time command buffers for specific cases that are used only once.

5.2.2 Main loop logic

All the components described earlier are initialized before the main loop starts. Once everything is set up, the main loop begins, and we submit work to the GPU, alternating between two command buffers. This process renders a beautiful red triangle thousands of times per second.

To submit work to the GPU, it is essential to ensure that the GPU has finished processing any previous submissions. This prevents conflicts on shared resources, such as the vertex buffer. For example, if the GPU is still using a resource from a prior frame, overwriting or modifying that resource could lead to undefined behavior.

To handle this, `fences` are used. It is a type of synchronization primitive that coordinates operations between the CPU and GPU. A fence allows the CPU to track the completion of GPU tasks. By waiting on a fence, the application ensures that the GPU has finished its associated operations before submitting new ones. Importantly, when a fence signals completion, it does not mean the GPU is idle; it simply indicates that all resources used for the work associated with that fence are no longer in use.

In the main loop, two key steps are performed:

1. **Waiting for the GPU:** This step corresponds to the "Wait GPU" arrow in Figure 5.2. Before submitting new work, we must ensure that the GPU has finished processing the previous frame. This involves two actions:
    - Acquiring an available image from the swapchain. This step ensures that the application targets a valid, unused swapchain image for rendering.
    - Waiting on the fence associated with the swapchain image to ensure that its resources are free.
2. **Executing Commands on the GPU:** This step corresponds to the "Execute Command buffer i" arrow in Figure 5.2. After acquiring the next swapchain image and ensuring the associated fence has been signaled, the command buffer linked to that image is submitted for execution. The GPU then begins processing the commands to render the next frame.

A final point to address is what happens when the window is resized.
When the window is resized, the dimensions of the swapchain images change, necessitating the recreation of several Vulkano components:

- **Swapchain:** The swapchain must be rebuilt with images of the updated size.

- **Frame buffers:** Since frame buffers are tied to the swapchain images, they must also be recreated to match the new dimensions.
- **Command Buffers:** The command buffers must be re-recorded to include commands targeting the newly created frame buffers.
- **Pipeline:** The graphics pipeline, particularly the viewport needs to be updated to accommodate the new image dimensions.

This process, while necessary, is pretty complex. It underscores a trade-off in Vulkan's design: the explicit control it provides enables high performance and flexibility but requires the application to manage these low-level details manually.

5.2.3 Coordinate system in Vulkan

In this Section, I will discuss Vulkan's coordinate system and how it is used in the different stages of the graphical pipeline.
First, let's take a look at Figure 5.4, which is taken from an article by Sean Harmer [1], explaining the coordinate system in Vulkan.
The figure shows how coordinates are transformed in a graphical pipeline from the model space to the frame buffer space, which represents the final image displayed on the screen.

The first step occurs in the vertex shader. It applies transformations to the 3D coordinates of the each vertex, converting them into 4D clip space coordinates called homogeneous coordinates.
Then, clipping and the perspective divide are performed to obtain the 3D Normalized Device Coordinates (NDC) . These steps are not explained in detail in this paper, but you can refer to Sean Harmer's article [1] for an in-depth explanation.
Finally, the viewport transformation discards NDC coordinates that are out of bounds and maps the remaining NDC coordinates into the 2D frame buffer coordinates, which correspond to the actual screen space for rendering.
You can see in Figure 5.3, the coordinate system of each space and their bounds.

In the Vulkano triangle example, we don't apply any transformations to the vertices, nor do we make use of the 4D clip space. Instead, in the vertex shader, we directly output the 2D coordinates with z = 0 and w = 1. This simplification allows the triangle to be rendered without the need of defining a Model View Projection matrix, making the example easier to understand.

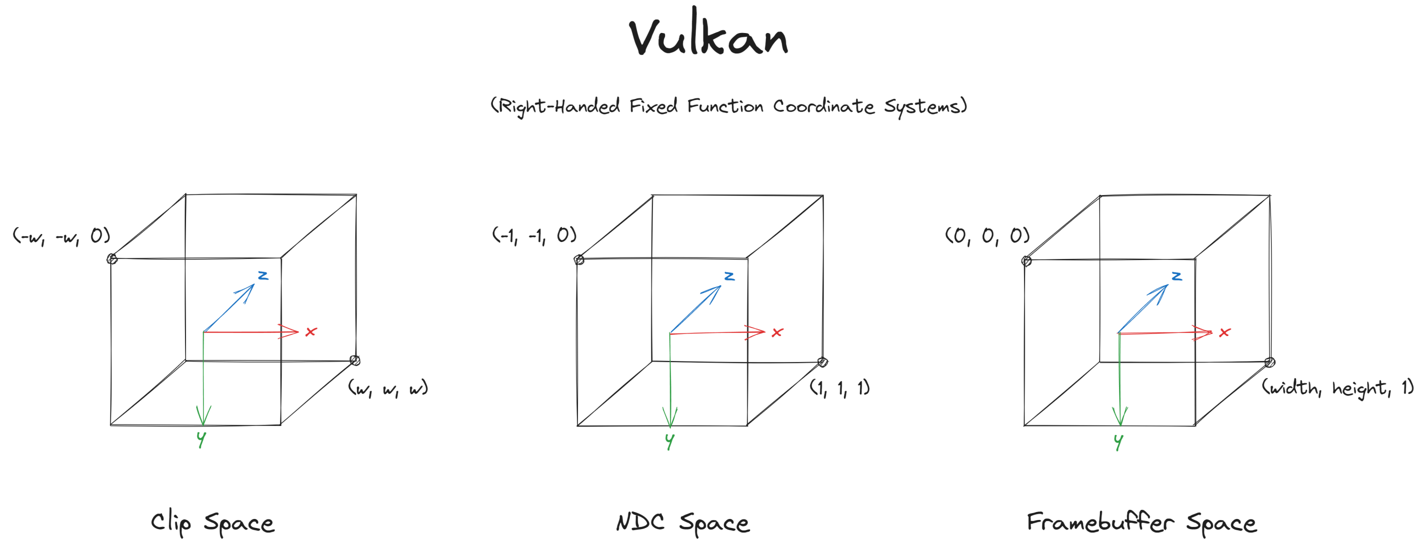


Figure 5.3: Coordinate system in vulkan from [1]

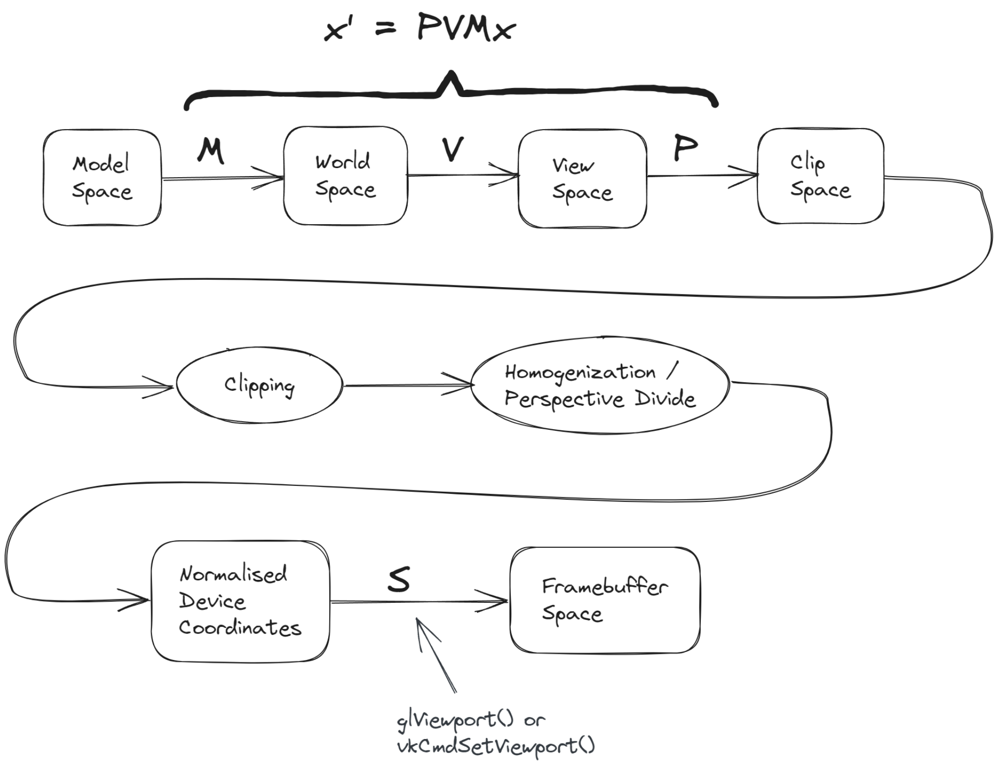

x' = PVMx
Model Space
M
World Space
V
View Space
P
Clip Space
Clipping
Homogenization / Perspective Divide
Normalised Device Coordinates
S
Framebuffer Space
glViewport() or vkCmdSetViewport()

Figure 5.4: Coordinate transformations in the pipeline from [1]

5.3 Renderer characteristics

In this section, I will outline the modifications made to the Vulkano triangle example, presented in the previous section, to develop a functional rendering system for the Minecraft clone. I will detail the new features added and describe the implementation process. A visual representation of the rendering system in the Minecraft clone is shown in Figure 5.5.

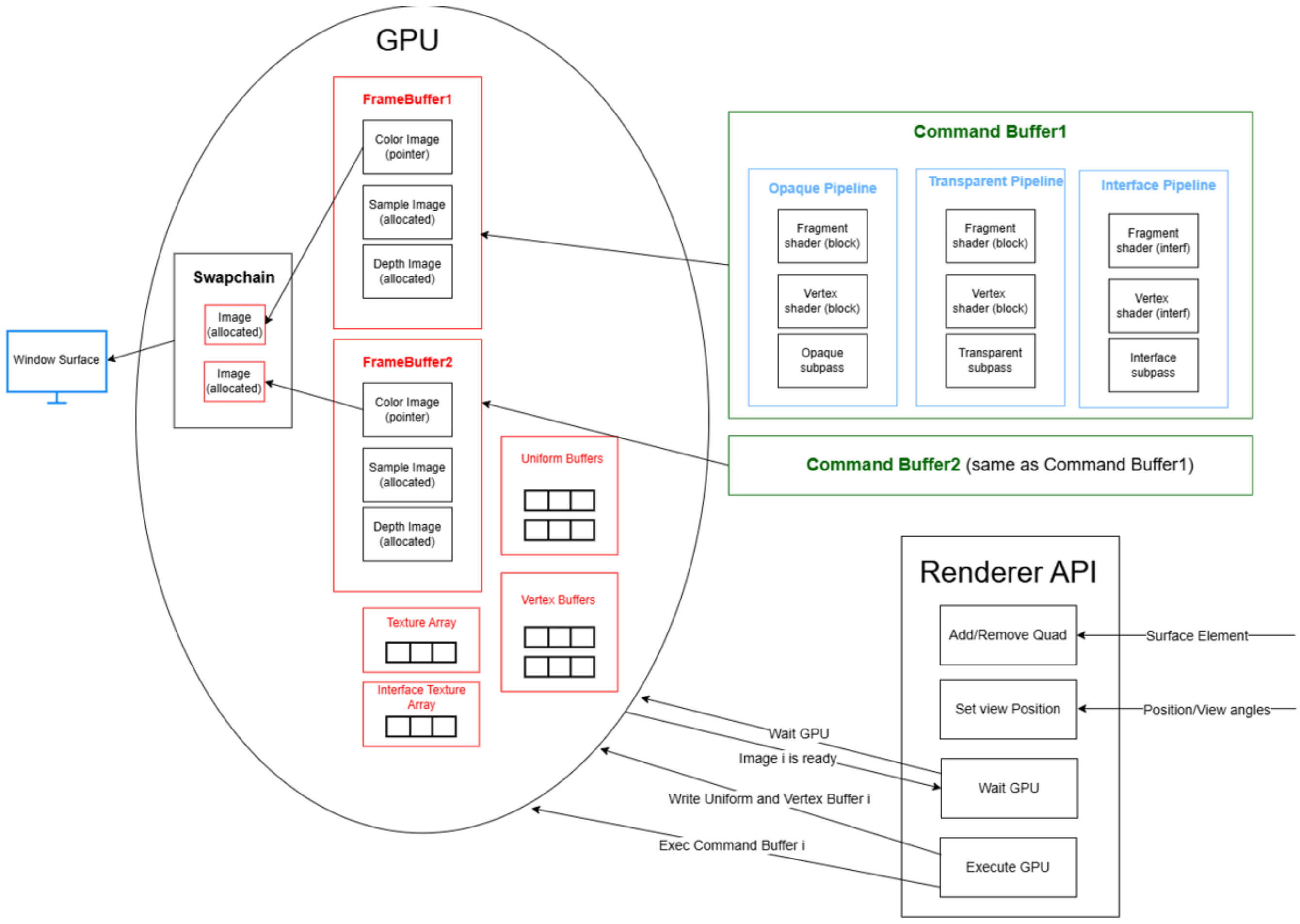


Figure 5.5: Visual representation of the Minecraft Clone rendering system

5.3.1 Dynamic changes

When creating a rendering system, one of the first considerations is the ability to dynamically modify the vertex buffer to apply changes. However, in the Vulkano example shown in Figure 5.2, both command buffers use the same vertex buffer. This can lead to difficulties accessing the vertex buffer, as the GPU is continuously using it.

In the Minecraft clone, this limitation is addressed by creating a second vertex buffer as a duplicate of the first. To ensure both buffers remain synchronized, two modification buffers are defined on the CPU side, one for each vertex buffer.
These modification buffers temporarily store vertex changes before they are applied to the corresponding vertex buffers.

5.3.2 Perspective

In Section 5.2.3, I mentioned that transformations can be performed in the vertex shader stage. I applied these concepts to render 3D surfaces instead of 2D triangles by first modifying the vertex buffer format to store 3D coordinates.
From now on, the term "surface" will be used to refer to a surface in 3D composed of two triangle primitives.

Next, I implemented the transformation of these 3D coordinates within the vertex shader. This transformation is achieved using an MVP (Model-View-Projection) matrix, which converts the 3D coordinates into 4D clip space coordinates. This step is essential for the correct rendering of objects in the subsequent stages of the pipeline.
Fortunately, in Rust, there are libraries that make it easy to set up the view position, near and far planes, field of view, as well as translation, scaling, and rotation. For this, I used the `nalgebra-glm` crate, which is similar to the one used in the tutorial [15].

The MVP matrix needs to be updated almost every frame, especially to implement a first-person view where the camera can rotate and move in all directions. To allow the MVP matrix to be updated at each frame, two uniform buffers are allocated in GPU memory.
A uniform buffer, unlike a vertex buffer, is a buffer designed to store data that remains consistent across all vertex calculations within a single draw call.
For the same reason explained in Section 5.3.1, two uniform buffers are required to facilitate continuous updates. While one buffer is being used by the GPU, the other can be modified.

If 3D surfaces are displayed without additional configurations, the triangles may not be rendered in the correct order, leading to visual artifacts or incorrect shapes. To resolve this issue, `depth testing` must be enabled.
To enable depth testing, a `depth image` must be allocated in each frame buffer, and the graphical pipeline must be configured to perform a depth test before the fragment shader colors the pixels. A depth image has the same dimensions (number of pixels) as the color image but stores depth information for each pixel instead of color.
Initially, all pixels in the depth image are set to a default value, typically representing the furthest possible depth (1.0 in Vulkan). During rendering, the depth test compares the z-coordinate (in normalized device coordinates) of each fragment against the corresponding value in the depth image. If the fragment's depth is less than the stored value, it passes the depth test, is rendered, and its depth is updated in the depth buffer. Otherwise, the fragment is discarded, ensuring that only the closest visible fragments are rendered.

By implementing the MVP transformations in the vertex shader, which can be updated dynamically based on user inputs, and enabling depth testing, The rendering system is capable of displaying a responsive first-person game.

5.3.3 Textures

After the implementation of the dynamic vertex update and the perspective, the next step was to add textures, enabling the rendering of blocks similar to those in the original Minecraft.

Textures were implemented by allocating a texture array in the GPU memory during the initialization of the rendering system. Since the textures are not modified during rendering, a single array is sufficient. The vertex format was then modified to include additional data beyond the 3D position of each vertex. Specifically, each vertex now includes:

- **UV coordinates:** These are 2D coordinates that map points on the texture to the corresponding points on a 3D object. The **U** and **V** values range from 0 to 1, where $(0, 0)$ represents the top-left corner of the texture, and $(1, 1)$ represents the bottom-right corner. This mapping ensures that the texture is correctly positioned and scaled on the object's surface.

- **Texture index:** This specifies which texture in the array should be applied to the vertex.

The vertex and fragment shaders were also updated to handle these changes and correctly apply the textures.
Additionally, a `sampler` was defined. Its role is to interpolate the texture stored in the texture array to match the UV coordinates, which are floating-point values rather than integers.

These changes enable the rendering system to assign unique textures to different surfaces, providing the necessary visual variety for a Minecraft-like environment.

5.3.4 Visual features

After textures were added, the game still looked unappealing, particularly due to jagged edges and poor texture quality at steep viewing angles.
To address these issues, two improvements were implemented:

Multisample Anti-Aliasing (MSAA): To smooth out jagged edges, Multisample Anti-Aliasing (MSAA) was implemented. This technique involves adding a multisampled image to each framebuffer and configuring the graphics pipeline to utilize multisampling. Additionally, the subpass was modified to resolve the final color image from the multisampled image, resulting in a smoother visual output.
In the rendering system of the Minecraft Clone, MSAA with a 4x sample rate is implemented.
The difference is evident when comparing the tower of sand in Figure 5.6a, where the edges appear jagged, to Figure 5.6b, which shows significantly improved visuals.

Anisotropic Filtering: To improve the quality of textures viewed at steep angles, anisotropic filtering was enabled. This was achieved by setting the sampler's anisotropy level to its maximum value, providing better texture visuals.
In the rendering system of the Minecraft Clone, maximum anisotropy setting of the device is automatically selected. The improvement can be clearly observed when comparing the grass texture rendered without anisotropic filtering in Figure 5.7b, which appear pixelated, to Figure 5.7a, offering smoother textures and a much more realistic appearance.

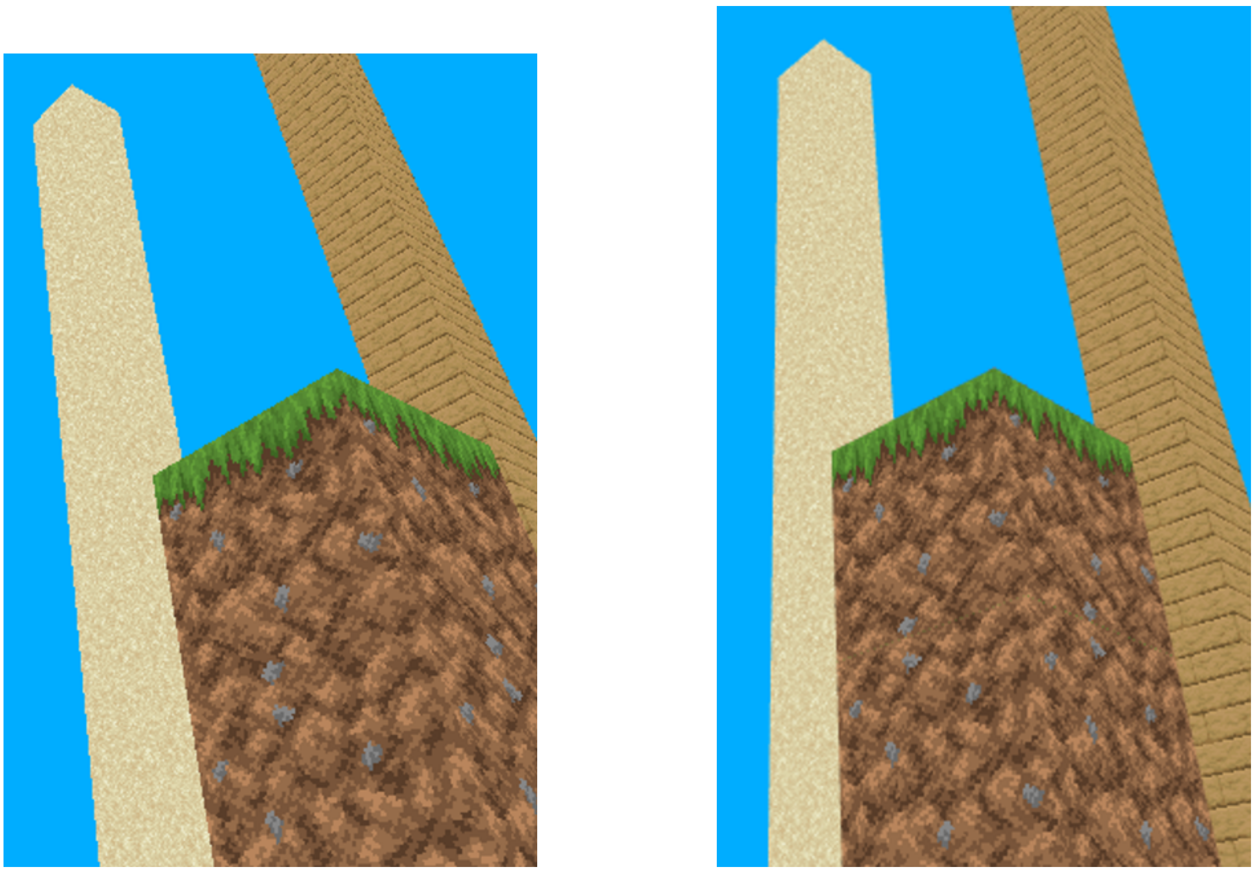

(a) Game without MSAA (b) Game with MSAA

Figure 5.6: MSAA visual changes

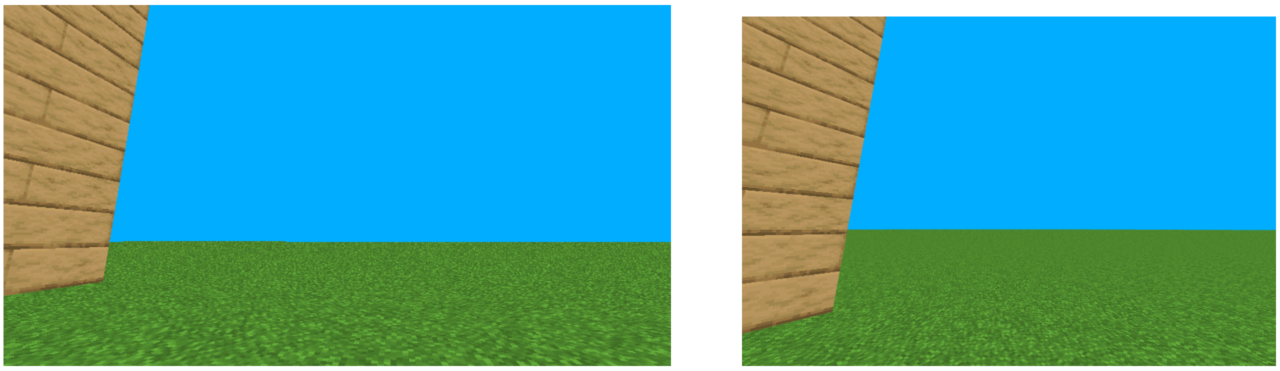

(a) Game without Anisotropy (b) Game with Anisotropy

Figure 5.7: Anisotropy visual changes

5.3.5 Dynamic textures

To implement dynamic textures in the rendering system, such as for water or lava blocks, surfaces with dynamic textures are updated every 50 milliseconds. This time interval is inspired by the "tick" in the original Minecraft game.

Every 50 ms, all the vertices of the dynamic surfaces are updated by incrementing their texture index from the CPU.

The animation speed and the number of frames for each dynamic texture are stored in a constant array. This array also specifies the texture name for each block texture.

While this approach works well when only a few dynamic surfaces are present, performance degrades significantly when many dynamic surfaces are present. This slowdown occurs because the CPU frequently performs random accesses to the vertex buffer in GPU memory to update the texture indices for a large number of vertices.

An effective optimization would eliminate the need to update each dynamic surface individually. Instead, the uniform buffer could store a value that is incremented with every game tick (every 50 ms) by the CPU. The fragment shader would then use the modulo operation on this value to determine the correct texture index to display. This approach would reduce the CPU's interaction with the GPU, shifting from frequent random accesses in the vertex buffer to a single modification in the uniform buffer.

This solution is not implemented in the Minecraft Clone and dynamic textures have been disabled due to their severe impact on overall performance.

5.3.6 Transparency

In the Minecraft Clone, an attempt was made to render transparent or partially transparent block by dividing the vertex buffer into two parts: 90% of the buffer is allocated for opaque surfaces, while the remaining 10% is dedicated to transparent surfaces. Since transparent surfaces require different rendering settings, a separate rendering pipeline is created for them. This new pipeline reuses the same shaders as the opaque surface pipeline but introduces a new subpass. The new subpass is necessary because there is now a second step in the rendering process. It modifies the color images created by the first subpass for opaque surfaces to include the transparent surfaces.

The newly created pipeline requires an important adjustment: enabling `blending`. Blending ensures that the color written to the frame buffer is combined with the existing color, rather than overwriting it. I implemented simple `alpha blending`, which is one of the most common blending strategies.

An other necessary change is to sort the transparent surfaces by their depth, from furthest to closest. This sorting ensures that transparent objects are blended correctly. However, this sorting part is not implemented in the Minecraft clone, meaning transparency is currently an experimental feature limited to water blocks, which can produce incorrect results.

A good solution could involve storing transparent objects in a separate vertex buffer and using an algorithm to sort the primitives (triangles) by depth. Since the array of transparent objects typically does not change much between frames, one could opt for a fast sorting algorithm that works well on a mostly sorted array.

In the current implementation, transparent objects are rendered after opaque objects, with depth writing disabled for transparent objects. This means that while the depth test still discards transparent objects that are behind opaque ones, transparent objects do not write to the depth buffer. As a result, transparent objects behind others can still be rendered. However, this can lead to issues with transparency, as alpha blending is not commutative, and without proper depth sorting, the final result can be visually incorrect.

5.3.7 User interface

The user interface includes elements such as menus and the cursor. To render these interface elements on top of the game, a third rendering pipeline is required.

Additionally, a new texture array is needed to store the textures used for the interface elements.
Two new vertex buffers are also necessary. While two vertex buffers aren't strictly required for the interface (we could use only one), I chose to include them for consistency with the overall architecture of the rendering system. This also provides the flexibility to dynamically adjust the position of the user interface elements.

I have also defined new vertex and fragment shaders. The vertex shader reuses the projection component from the MVP matrix stored in the uniform buffer, while the fragment shader is responsible for selecting the appropriate texture for the interface element being rendered.

In the current Minecraft clone, the interface texture array only includes the cursor. However, a potential improvement would be to add additional interface elements, such as menus.

Another enhancement could be to modify how the cursor is rendered. Using the projection component for positioning the cursor is not ideal, as it causes the cursor to stretch when the window is resized. This issue does not occur in full-screen mode, but it would be better to implement a more flexible solution for handling the cursor's position and scaling.

5.3.8 Culling and vertex optimization

Building an efficient rendering system requires minimizing the number of vertices processed, as this allows for more chunks in the minecraft Clone to be rendered and enhances the user's experience. While I have not implemented all of these strategies, I am aware of the significant performance improvements these optimizations could provide and describe them in this section.

Back face culling

One of the first optimizations that can be made is back-face culling. This straightforward technique prevents the rendering of primitives that are not visible to the camera, as they face away from the viewer's point of view.

Back-face culling determines visibility by analyzing the winding order of a face's vertices in screen space. Faces are considered front-facing if their vertices are ordered counterclockwise relative to the viewer, while faces that are clockwise are culled and not drawn. This culling occurs before the fragment shader processes the vertices, reducing unnecessary computations and enhancing performance.

In the Minecraft clone, back-face culling is enabled, and all functions responsible for constructing surfaces adhere to this convention. This ensures that only visible faces of blocks are processed and rendered, improving overall performance.

Index buffer

An other efficient way to reduce the number of vertices processed is by optimizing how surfaces are described. In the rendering system, only groups of faces are drawn, and these are represented as rectangles. To define a surface using only a vertex buffer, two triangles must be drawn, which results in six vertices per surface as shown in Figure 5.8.

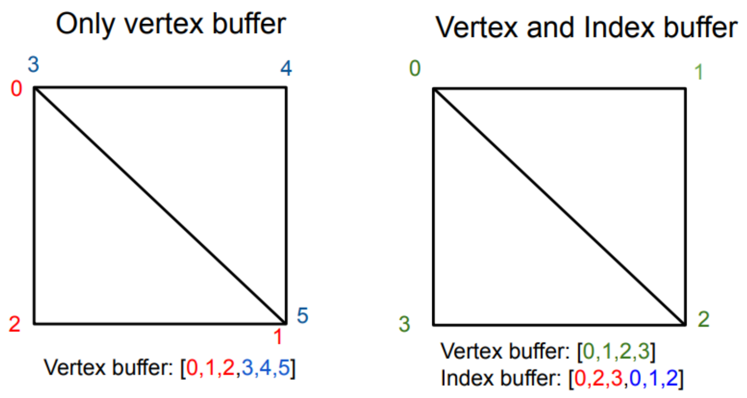


Figure 5.8: Surface definition with and without index buffer

This approach is inefficient because two of these vertices are duplicates, and they are processed twice by the vertex shader. A better solution is to use an index buffer, which references the vertex buffer to define triangle primitives. By employing an index buffer, the number of vertices required per surface is reduced by 33%, dropping from six to four. This would have the effect of reducing the workload of the rendering system by the same percentage.
A representation of this optimization is shown in Figure 5.8.

It is important to note, however, that this strategy cannot be applied across different surfaces. This is because each surface may have unique textures or varying normals in their vertex data, which prevents vertices from being shared between them.

Unfortunately, this optimization has not yet been implemented in the Minecraft clone, as managing the vertices and their indices proved to be a complex task. Nevertheless, integrating this technique would greatly improve the rendering system's performance.

Frustum culling

Frustum culling is a technique used to avoid processing vertices that lie outside the camera's view frustum, which is illustrated in Figure 5.9. This optimization is crucial in scenarios where a large number of objects need to be rendered, such as in the Minecraft clone.

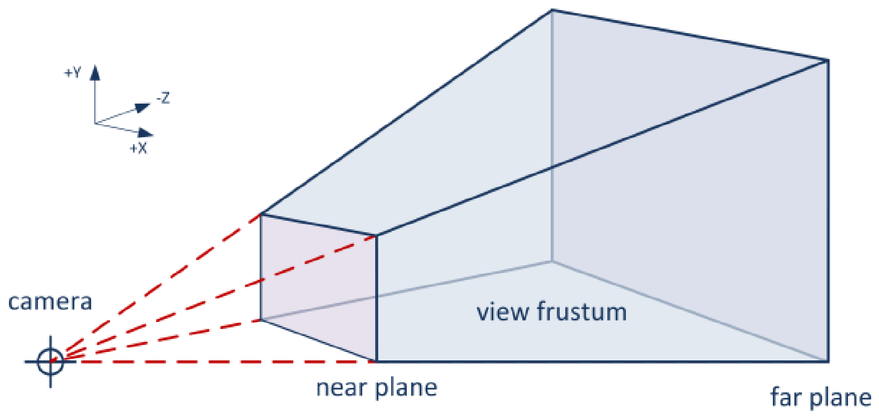


Figure 5.9: View frustum from [2]

The graphical pipeline includes a form of frustum culling called "clipping," but this occurs after vertex transformations, as illustrated in Figure 5.4. Clipping helps save resources by skipping depth testing and avoiding writes to the framebuffer for primitives outside the view frustum. However, it does not prevent the vertex shader from processing these vertices.

In contrast, frustum culling is typically performed on the CPU and can significantly reduce the number of vertices processed by the vertex shader in each frame. When implemented effectively, this technique can greatly improve rendering performance.

This strategy has not been implemented in the Minecraft clone, as it requires much more flexibility from the rendering system to add and remove an enormous number of vertices every frame.

A potential solution could involve performing frustum culling at the chunk level. This approach would enable an efficient CPU-side algorithm, as chunks can act as coarse-grained units of culling. The rendering system could be extended with new APIs to enable or disable the rendering of specific chunks dynamically. Each chunk could be stored in separate vertex buffers, allowing for quick removal or addition to the rendering pipeline. This separation would minimizes overhead and avoids costly vertex buffer updates while maintaining rendering flexibility.

5.4 Renderer API

To use the rendering system, the first step is to create a new instance, which initializes all the components described earlier.
Once the rendering system is created, four primary functions can be called:

- `wait_gpu` and `exec_gpu`: These functions, similar to those in the Vulkano triangle example, must be called in each iteration of the main loop in the `main` crate.
- `add/remove Surface`: The rendering system allows surfaces to be added or removed. These changes are buffered and will be applied at the next `exec_gpu` call. These surfaces are defined in the `draw_element` crate, which is discussed in detail in Section 6.1.
- `Set View Position`: This function adjusts the view angles and the camera position. It must be called between `wait_gpu` and `exec_gpu`, as these changes are not buffered and are written directly to the uniform buffer. This approach was chosen to make the game more responsive.

While the current approach prioritizes responsiveness, buffering view changes would improve clarity and modularity. By implementing buffering, the rendering system could consolidate `wait_gpu` and `exec_gpu` into a single `update` function. Additionally, any future APIs added to the system would also be buffered, leading to a more streamlined and flexible design.

5.5 Important consideration

The vertex buffer has a fixed size and does not dynamically grow. If a surface is added while the vertex buffer is full, it will not be added, potentially causing chunks to fail to render. Currently, there is no mechanism in place to prevent or handle this behavior.

A possible solution could involve limiting the number of chunks rendered or updating the vertex buffer's size. However, it is important to note that resizing the buffer is a very costly operation and should be approached with caution.

Another possible solution would be to define a separate vertex buffer for each chunk. This approach would not only alleviate the limitations of a fixed-size global vertex buffer but also enable the possibility of implementing frustum culling effectively.

6 Communication Modules

This Section discusses the crates responsible for enabling communication between the various components of the game. These crates are essential for the seamless functioning of the game, facilitating interaction between the game logic and the rendering system, while also ensuring the overall modularity of the game architecture.

6.1 Draw Element definition

A draw element refers to any element that the game logic sends to the rendering system for drawing or removal.
This crate is designed to formalize the exchange of information between the rendering system and the game model. Its primary goal is to promote modularity by defining structures that both systems can use to communicate about what needs to be rendered.
By doing so, it allows for the flexibility to swap or modify either the rendering system or the `game_core` crate without impacting the overall functionality of the game. In essence, it facilitates modularity and separation of concerns between the game logic and the rendering components.

In this version of the Minecraft Clone, the crate defines a single structure: the `SurfaceElement`.
A `SurfaceElement` is a group of contiguous block faces that together form a rectangular surface. These faces are grouped to optimize rendering, as they represent a visible portion of the world that can be drawn as a single rectangular unit. The algorithm used to achieve this grouping is explained in Section 4.3.

Additionally, the crate defines conventions for both blocks and faces. The convention for faces is illustrated in Figure 4.1a.

6.2 Main crate

With the rendering system, game core, and draw element well defined, the `main` crate primarily contains the main loop, which does input detection and sequentially updates both the rendering system and the game core. This loop ensures that the game progresses by continuously updating the game state and rendering the changes to the screen.

An improvement to this design would be to run the rendering system and the game core asynchronously. By doing so, input lag could be reduced, and delays in one component such as game logic updates would not block an other component such as rendering, leading to a smoother overall experience.

Part III
Performance Measurements

In this part, I will assess the performance of the Minecraft Clone game by analyzing key factors that affect both its gameplay experience and overall efficiency. This evaluation will include an examination of the sequential steps in the main loop, along with an analysis of how various variables impact the number of surfaces generated and the writing delay in the vertex buffer
This comprehensive approach will help identify potential bottlenecks and areas for optimization within the game.

All performance tests were conducted on my personal machine, which has the following specifications:

- **Model**: ACER PREDATOR PH315-53
- **Processor**: Intel®Core™i7-10750H CPU @ 2.60 GHz
- **RAM**: 16 GB
- **GPU**: NVIDIA GeForce RTX 2070 with Max-Q Design

7 Measurements

7.1 Execution time

In this section, I will measure the worst-case execution time for each step in the main game loop. The GPU rendering time for a frame is not measured here, as it is difficult to capture. Instead, an approximation is provided by evaluating performance based on the vertex buffer size in Section 7.2.

The tests were conducted using the following constants from the `game_core` crate, while moving in the game for 20 seconds to load and unload chunks:

- `PERLIN_FACTOR` = 10: This factor influences the steepness of terrain changes; a higher value results in smoother terrain.
- `PERLIN_MAX_HEIGHT` = 10: This defines the maximum height of a block in the world.
- `RENDER_DISTANCE` = 7: This is the distance (in chunks) at which the game will render terrain around the player.
- `MAX_DRAW_BATCH` = 300: This is the maximum number of `SurfaceElements` returned during each Game Core update.

The results are shown in Table 7.1.

Step	**Worst Delay Experienced (ms)**
Game Core Update	0.25
Renderer Buffering Updates	1.5
Renderer Vertex Buffer Writing	2

Table 7.1: Maximum delay experienced by different steps

We can observe that the worst delay occurs due to the writting operations on the vertex buffer, followed by the buffering of surface changes. The Game Core update, on the other hand, has a significantly lower delay compared to the other steps.

In conclusion, the main bottleneck in the Minecraft Clone is caused by the I/O operations performed by the rendering system on the vertex buffer and the process of buffering surface changes for additions and removals.

7.2 Vertex buffer size

In the rendering system, the size of the vertex buffer is fixed during its initialization. Since there is no index buffer, all the vertices in the vertex buffer are processed, even if some of them contain 0s data. As a result, the size of the vertex buffer plays a crucial role in the performance of the rendering system and provides a close approximation of the GPU rendering time.

The tests were conducted by launching the game without moving, to prevent chunk loading or unloading, and focusing solely on the performance of the GPU.
The following constants were used:

- `PERLIN_FACTOR` = 10
- `PERLIN_MAX_HEIGHT` = 10

7.2.1 Results

The impact of the vertex buffer size on the average frames per second (FPS) is presented in Table 7.2.

Vertex Buffer size	**FPS**	**Surfaces**	**Max chunks**	**Max Render Distance**
1,000,000	1000	160,000	370	9
2,000,000	950	330,000	740	13
5,000,000	160	830,000	1,850	21
8,000,000	110	1,330,000	2,960	26
10,000,000	40	1,660,000	3,700	29

Table 7.2: Average FPS on vertex buffer size

We can observe that the FPS drops significantly when the number of vertices to process exceeds 5 million. Since most games are designed to run at 144 FPS, we can conclude that computers with specifications similar to mine should be able to handle a vertex buffer size of up to 5 million vertices.

The other columns assign the theoretical maximum number of chunks that can be loaded, along with the theoretical maximum render distance, from the vertex buffer size.
The number of surfaces is calculated by dividing the number of vertices by 6, as the rendering system uses 6 vertices to represent a single surface.
The theoretical maximum number of chunks that can be loaded is estimated by considering that a chunk, on average, generates 450 surfaces (for the parameters of the test), which correspond to 2,700 vertices needed to represent the visible surfaces.
The theoretical maximum render distance is determined using the formula $(\sqrt{\text{max chunks}} - 1)/2$, as the render distance represents the number of chunks that can be loaded from the chunk where the player is located.

7.2.2 Discussion

The rendering system can achieve very good results under these conditions, with the theoretical maximum render distance for a smooth 144 FPS game being 21 chunks, enabling it to process up to 1,850 chunks per frame.
However, other factors, such as the Perlin constants used to generate the chunks, affect the number of surfaces per chunk, which can lead to varying results. These factors are tested in the Section 7.4.

On the other hand, optimizations that reduce the number of surfaces rendered are crucial for further improving performance. These future optimizations are discussed in Section 8.2.

7.3 Batch size

As explained in Section 4.4, the `game_core` crate returns `SurfaceElements` in batches.
This section will analyze the maximum delay experienced when writing to the vertex buffer of the rendering system for different maximum batch sizes.
The tests were conducted using the following constants from the `game_core` crate, while moving in the game for 20 seconds to load and unload chunks:

- `PERLIN_FACTOR` = 10: This factor controls the steepness of terrain changes; higher values result in smoother terrain.
- `PERLIN_MAX_HEIGHT` = 10: This defines the maximum height of a block in the world.
- `RENDER_DISTANCE` = 7: This specifies the distance (in chunks) at which the game will render terrain around the player.

7.3.1 Results

The effect of the maximum batch size on the worst delay experienced when writing to the vertex buffer is shown in Table 7.3.

Max batch size	Renderer vertex buffer writing worst delay (ms)
200	4
300	6
500	10
Not Limited	50

Table 7.3: Renderer write delay on batch size

We can observe that the worst delay experienced when writing to the vertex buffer increases linearly with the maximum batch size.

7.3.2 Discussion

The maximum batch size constant in the `game_core` crate plays a crucial role, as it represents a tradeoff between slower chunk rendering when loaded and reduced delay when writing to the vertex buffer.

Based on the results, the optimal batch size for a machine with specifications similar to mine is a maximum batch size of 300. This results in a delay of 6 ms, which does not cause any lag in a game running at 144 FPS.

7.4 Perlin constants

This section focuses on evaluating the number of surfaces generated based on the Perlin constants used for terrain generation.
The tests were conducted using the following constant from the `game_core` crate:

- `RENDER_DISTANCE` = 4: Specifies the distance (in chunks) at which the game renders terrain around the player.

7.4.1 Results

Table 7.4 presents the results of tests where the `PERLIN_MAX_HEIGHT` constant was varied, showing its impact on the number of surfaces generated, with a fixed `PERLIN_FACTOR` = 10.
Similarly, Table 7.5 summarizes the results of tests where the `PERLIN_FACTOR` constant was modified, illustrating its effect on the number of surfaces generated, with a fixed `PERLIN_MAX_HEIGHT` = 30.

Perlin Max Height	Vertices	Surfaces
10	180,000	30,000
15	245,000	41,000
30	420,000	70,000
50	650,000	108,000
100	1,000,000	160,000

Table 7.4: Vertex count based on perlin max height

Perlin Factor	Vertices	Surfaces
10	420,000	70,000
5	700,000	117,000
3	900,000	150,000

Table 7.5: Vertex count based on perlin factor

It can be observed that increasing the `PERLIN_MAX_HEIGHT` significantly increases the number of vertices generated.
Similarly, lowering the `PERLIN_FACTOR` also results in a substantial increase in the number of vertices.

7.4.2 Discussion

This phenomenon can be attributed to the increase in the number of similar faces stacked on top of each other when `PERLIN_MAX_HEIGHT` is increased or `PERLIN_FACTOR` is decreased. The current greedy meshing algorithm struggles to optimize these cases effectively.

The algorithm, implemented in the `game_core` crate, attempts to create lines from visible faces. Specifically, it processes faces along the X direction for the NORTH, SOUTH, TOP, and BOTTOM faces, and along the Z direction for the EAST and WEST faces. However, this approach is ineffective for optimizing stacked faces, as it does not group or reduce redundancy in such configurations.

7.5 Summary of the results

These performance tests highlight that the rendering system performs efficiently on its own, capable of processing 1850 chunks with a stable frame rate. The main bottleneck arises from large modifications to the vertex buffer, which occur during chunk loading. To address this limitation, further work is needed, including:

- Reducing the overall number of vertices per chunk by improving the greedy meshing algorithm and utilizing an index buffer.
- Preloading chunks that are not visible and rendering them as visible once they are complete.

These improvement strategies are discussed in Section 8.

Part IV
Conclusion

8 Future Work

This section outlines potential future improvements for the Minecraft clone, which could serve as a personal project for a student or as a Master's thesis topic. It also acts as a reference for my ongoing development of the clone.
The section includes a list of optimizations aimed at enhancing the game's robustness, as well as ideas for features that could improve the overall user experience.
I highly recommend that developers or students aiming to improve or modify this work start by addressing the optimization issues in the rendering system. As discussed in Section 7, the rendering system is the bottleneck of the Minecraft Clone.

8.1 Game Core Logic

8.1.1 Collision system

As explained in Section 4.1.2, the collision system needs improvement.

An important enhancement would be to modify the collision system to support multiple moving entities. This improvement is crucial, as it must be completed before adding multiple players or other moving entities to the game. The implementation of this improvement should not be overly complicated and could use the proposed solution in Section 4.1.2.

An optimization that could also be considered is to gather obstacles more efficiently. However, as discussed in Section 4.1.2, the current system for obstacle gathering is not a significant bottleneck.

8.1.2 Chunk storage and generation

The chunk storage of the Minecraft clone is rather rudimentary, as it stores the entire chunks in separate files.

An improvement should be made at this level to implement a system where chunks are stored based on changes to the base seed, rather than storing all blocks. It would also be beneficial to store chunks in region files, similar to the original game. Additionally, compressing the saved chunks to save space would be an important optimization.

The chunk generation system is also rudimentary and would benefit from the use of multiple Perlin noise layers. However, as discussed in Section 4.2.1, optimizations to the rendering system must be completed prior to implementing this improvement.

8.1.3 Greedy meshing

As explained in Section 4.3.1, the Minecraft clone currently groups visible faces by line. An important improvement would be to modify the algorithm to group visible faces into rectangles.

Alternatively, it may be possible to implement a different algorithm, such as the max rectangle removal or an exact solution, provided it is not too computationally intensive. The key consideration is that the solution must remain efficient for dynamic changes.

Another potential improvement to resolve the "badly loaded chunks" bug mentioned in Section 8.3.3, and to reduce fragmentation in the face groupings, could be to avoid saving these groupings within the chunks.
Instead, they could be recalculated each time the chunk is loaded.

8.1.4 Placing blocks

Currently, it is possible to place a block on the player.
While this feature is useful for testing the rendering system's functionality, it needs to be addressed eventually by checking if an entity is present before allowing a block to be placed.

8.1.5 Dynamic batch update

Currently, the batch size of returned `SurfaceElements` is hardcoded as a constant. A valuable improvement would be to dynamically adjust its value based on the render distance and the number of FPS.

8.2 Renderer

8.2.1 Dynamic textures

The dynamic textures in the Minecraft clone are currently poorly managed and were even disabled due to causing a significant loss in performance.
A major change is required to properly handle texture animations.
The problem and a possible solution are described in Section 5.3.5.

8.2.2 Relative coordinates

The current rendering system operates using absolute coordinates, transforming the relative coordinates of the `SurfaceElements` into absolute coordinates. It also uses absolute coordinates for the camera position.

This is considered poor practice, as all calculations in shaders are performed using 32-bit floating point values. This can lead to precision issues when objects are positioned far from the origin.

Defining relative coordinates within the renderer would be a significant improvement. It could use the same chunk size as in the `game_core` crate, or alternatively, define a different size, as the `draw_element` crate is flexible and designed for this purpose.

8.2.3 Toggling visual features

In the current version of the Minecraft clone, all visual features are initialized when creating an instance of the rendering system.

A useful feature addition would be to provide an API to toggle settings such as MSAA, anisotropy, and fullscreen. This is not a trivial improvement, as it would require refactoring the `renderer` crate to make it more modular.

8.2.4 Index buffer

The addition of an index buffer would immensely benefit the rendering system, as it could significantly reduce the number of vertices processed by the system.
As discussed in Section 5.3.8, redefining the surface using an index buffer would reduce the number of vertices by 33%.

8.2.5 Transparency

As discussed in Section 5.3.6, the transparency feature is still experimental and lacks a reliable method to render transparent objects correctly. A challenging but necessary improvement would involve resolving this issue by implementing a proper technique to sort transparent objects or by exploring alternative strategies for rendering them correctly.

Additionally, separating the vertex buffers for transparent and opaque objects could improve the clarity and performance of the rendering system, as it would allow for more efficient handling and sorting of the transparent objects.

8.2.6 Visual amelioration

In the current state of the game, block textures appear pixelated from a distance when standing still, even with the maximum anisotropy setting.
An improvement is necessary to enhance the game's visual appeal.
A potential solution would be to create a new subpass to smooth the color image written in the frame buffer.

8.2.7 Expanded user interface

The only interface present in the Minecraft clone is the cursor. As explained in Section 5.3.7, the cursor is rendered using the same projection matrix as the blocks, which can cause visual issues when the window is resized.
A valuable addition would be to define a uniform buffer for the interface elements and introduce menus through a new API.

8.2.8 Animations and particle effects

Another great addition would be to implement block-breaking animations and particle effects.
Additionally, providing visual feedback for the player's hand would further enhance the experience.

8.2.9 Water and lighting shaders

Currently, water is treated as a surface like any other block in the game. A significant, though not essential, improvement would be to implement specific shaders for water rendering through a new API for the renderer.
Furthermore, there is no lighting or shadows in the current rendering system. Adding lighting and shadow effects by defining new shaders would greatly improve the visual experience and enhance the user interaction with the game.

8.2.10 Vertex buffer

The vertex buffer size is currently hardcoded, which could lead to performance loss if the buffer is too large and empty in certain areas, or too small, causing some chunks to not be displayed. This should be modified to account for the number of chunks loaded, which is determined by the render distance.

Another solution would be to define a separate vertex buffer for each chunk, instead of using a global vertex buffer. This would enable the possibility of implementing frustum culling and also allow preloading chunks into the rendering system.

8.3 Other possible improvements

8.3.1 Asynchronous execution

The current Minecraft clone updates the state of the game and the rendering system in a sequential manner as explained in the Section 6.2.
Switching to an asynchronous approach for both the rendering system and the `game_core` crate could be beneficial and lead to a more modular structure.
This would ensure that the delay of one module does not impact the others.

8.3.2 Sound

The Minecraft clone currently lacks sound. A valuable addition would be to implement a module responsible for handling the game's sound.

8.3.3 Bugs

I have noticed some visual bugs when chunks are loaded, sometimes some surfaces do not render.
The issue may arise from a bug in the greedy meshing algorithm or from how blocks are currently stored.
Modifying the way greedy meshing works will likely resolve these problems.

Another bug occurred once where a chunk did not generate, but the cause of the problem is unknown.

9 Conclusion

This thesis marks a significant step in exploring the development of voxel-based games using modern tools such as Rust and Vulkan. By creating a functional Minecraft clone, I have gained valuable insights into the complexities of game development, including 3D rendering pipelines, game mechanics, and performance optimization.

The project demonstrated the feasibility and advantages of using Rust for game development, particularly highlighting its memory safety features, which substantially reduce the risk of critical errors and enable rapid development cycles. Additionally, employing Vulkan as the rendering API provided an in-depth understanding of the rendering pipeline, despite its steep learning curve. These experiences have not only expanded my technical expertise but also honed my problem-solving skills in addressing real-world challenges in game development.

While the game remains in its early stages, the foundational framework established here serves as a robust starting point for further development. It also acts as a valuable resource for students and developers interested in game design, particularly within the realm of voxel-based environments.

One of the key achievements of this project was the successful implementation of a Minecraft clone capable of managing up to 225 chunks at a stable 144 FPS, all while maintaining a functional game with basic mechanics

Looking ahead, there are numerous opportunities to enhance this project. One crucial improvement involves implementing advanced rendering strategies, such as frustum culling, to optimize performance further. Such enhancements could lead to a more sophisticated chunk generation system, potentially enabling procedurally generated worlds akin to the original Minecraft. Additionally, integrating features such as sound, animations, and multiplayer capabilities would significantly elevate the user experience and showcase the scalability of the framework.

The open-source nature of this project encourages collaboration and innovation, providing a platform for students and developers to explore, learn, and contribute to game development.

In conclusion, this thesis fulfills its dual purpose of serving as a foundation for further development and as a personal journey into the intricacies of game development. While the project is still in its infancy, it lays a solid groundwork for future improvements and expansions. I hope this work inspires others to delve into the potential of voxel-based games, fostering further exploration and innovation in this field.

Bibliography

[1] Sean Harmer : Projection matrices with vulkan - part 1. https://www.kdab.com/projection-matrices-with-vulkan-part-1/. Viewed on the 30th of December, 2024.

[2] Frustum culling. https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling. Viewed on the 31th of December, 2024.

[3] FinalForEach : I remade minecraft but it's optimized. https://www.youtube.com/watch?v=fz3Td2zlgro. Viewed on the 28th of December, 2024.

[4] Low Level Game Dev : How to make minecraft in c++ or any other language. https://www.youtube.com/watch?v=ebeRZVHsl00. Viewed on the 28th of December, 2024.

[5] Henry Leclipteur : Gitlab. https://gitlab.uliege.be/Henry.Leclipteur/minecraft-rust-clone.

[6] Amanatides John et al : A fast voxel traversal algorithm for ray tracing. 1987. http://www.cse.yorku.ca/~amana/research/grid.pdf.

[7] Wikipedia : Hyperplane separation theorem. https://en.wikipedia.org/wiki/Hyperplane_separation_theorem. Viewed on the 27th of December, 2024.

[8] David Eberly : Intersection of convex objects: The method of separating axes. https://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf. Viewed on the 27th of December, 2024.

[9] Minecraft Wiki : Noise generator. https://minecraft.fandom.com/wiki/Noise_generator. Viewed on the 28th of December, 2024.

[10] Minecraft Wiki : Region file format. https://minecraft.wiki/w/Region_file_format Viewed on the 28th of December, 2024.

[11] Minecraft Wiki : Anvil file format. https://minecraft.wiki/w/Anvil_file_form. Viewed on the 28th of December, 2024.

[12] dphfox Elttob : Consume everything - how greedy meshing works. https://devforum.roblox.com/t/consume-everything-how-greedy-meshing-works/452717. Viewed on the 28th of December, 2024.

[13] Geeks for geeks : Maximum size square sub-matrix with all 1s. https://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/. Viewed on the 28th of December, 2024.

[14] Vulkano, safe rust wrapper around the vulkan api. https://vulkano.rs/01-introduction/01-introduction.html. Viewed on the 29th of December, 2024.

[15] taidaesal : Vulkano tutorial. https://github.com/taidaesal/vulkano_tutorial. Viewed on the 29th of December, 2024.