Silesian
University
of Technology

FINAL PROJECT

Implementation of 3D game MakeFarm with infinite terrain model and basic physics in OpenGL API

Dawid Szymon GROBERT
Student identification number: ■■■■■■

Programme: Informatics
Specialisation: Informatics

SUPERVISOR
dr hab. inż. Agnieszka Szczęsna
DEPARTMENT of Graphics, Computer Vision and Digital Systems
Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2023

Thesis title

Implementation of 3D game MakeFarm with infinite terrain model and basic physics in OpenGL API

Abstract

The purpose of this thesis is to implement a voxel-based game containing procedurally generated terrain, and the essential world interactions of a computer game inspired by the game Minecraft. The thesis describes how the project was created without using an existing 3D game engine on the market, so it solves the most basic problems related to physics, computer graphics, and the game logic itself.

Key words

opengl, procedural terrain generation, voxels

Tytuł pracy

Implementacja gry 3D MakeFarm z nieskończonym modelem terenu i podstawową fizyką w API OpenGL

Streszczenie

Celem tej pracy jest implementacja wokselowej gry zawierającej proceduralnie generowany teren, oraz podstawowe założenia gry komputerowej zainspirowane grą Minecraft. Praca opisuje sposób powstawania projektu bez użycia istniejącego już na rynku silnika gier 3D, więc rozwiązuje przy tym najbardziej podstawowe problemy związane z fizyką, grafiką komputerową, oraz samą logiką gry.

Słowa kluczowe

opengl, proceduralnie generowany teren, woksele

Contents

Chapter 1

Introduction

Developing computer games is an extremely difficult and complicated process that consumes huge amounts of money. If one were to think about how many different scientific fields computer games use, we could easily list fields such as computer graphics, art, psychology, economics, or social sciences. Computer games are definitely driving the development of technology. Companies are outdoing themselves in terms of artificial intelligence, photo-realistic graphics, efficient networking systems, incredibly sophisticated systems to catch cheaters in online games, and much more.

Many players don't dive into what's going on inside the game they're playing. Even many developers who create games don't necessarily know what goes on inside the game engine. It's a huge collection of tools that try to hide all the low-level issues of creating computer games from the scratch.

The goal of this project is to descend into low-level computer game design and implement using the OpenGL API a voxel-based (see Section 2.1.2) game containing procedurally generated terrain, and the essential world interactions of a computer game inspired by the game Minecraft. Minecraft released its first version in 2009 [34] and has given voxel-based games worldwide recognition among gamers. Initially, through its voxel world and the blocks with which to build, it reminded people of LEGO [1]. Now 13 years later, anything that consists of blocks, and allows to build is being compared to Minecraft, and younger and younger generations are most likely comparing LEGO to the Minecraft game.

In the years in which Minecraft was created, game engines were not as widely available to the individual developer [8]. Many of them were not as easy to use, and often required the purchase of a license. Nowadays, nothing prevents the individual developer from creating a game similar to Minecraft using a popular game engine from the market. However, projects like this one allow to peek „under the hood" and learn a lot. Many developers even nowadays decide to abandon the benefits of the already existing game engines, saying that their own solutions give them more freedom. This often results in

[1] The toy company, which gained its worldwide recognition on blocks that could be combined with each other and create various constructions.

amazing solutions that exceed all expectations. An example of such a game is Teardown, with beautiful graphics and efficient destruction of the environment based on voxels.

The next chapters will include descriptions of the subsequent stages of the development of this project. The second chapter of this work will clarify what voxels are and will analyze the problems associated with the creation of the game from scratch and the generation of infinite terrain. The third chapter will focus on assumptions related to the game as well as requirements. The chosen development tools, used libraries and development methodology will be explained. The fourth chapter will allow the reader to understand how to play the game, and build the project. The fifth chapter will describe the internal architecture and the implementation of the most important mechanisms. The sixth chapter will focus on the quality assurance of the game. The final, seventh chapter will summarize the results achieved and describe future expectations for project development.

Chapter 2

Analysis of the problem

Computer games create a huge market numbering up to hundreds of billions of dollars, and computer games themselves are so complex that they can significantly drive technology development. Computer graphics and computer games go hand in hand holding countless specialists in many fields in their grasp. Considering that a computer game is a large undertaking that must be planned well in advance to carry out the project in a linear approach from start to finish, there are many decisions to be made at the very beginning.

2.1 Game engine

To reduce production time and minimize costs, companies often decide on an existing game engine already on the market. Usually, game engines already include such elements as a rendering engine for 2D or 3D graphics. A physics engine, multithreading, networking, and toolkits to help create the game itself. Games that choose not to take advantage of the benefits of existing game engines must face additional problems to solve. Due to the fact that this project focuses on the process of creating a voxel-based game including the most basic issues of creating computer games from scratch then no video game engine will be used, and the problems associated with it will be explained and solved.

2.1.1 Game loop

Games that are created from scratch must take care of even the smallest details. Such one detail is precisely the game loop, which is often neglected by developers and this leads to many problems.

The execution of the game is based on a game loop [16]. Beginning programmers creating their first games often create primitive game loops like the one shown in Listing 1.

```
def gameLoop():
   while window.isOpen():
       updateGame()
       handleEvents()
       render()
```

Listing 1: Pseudocode of the primitive game loop.

The first problem is the lack of elapsed time since the previous frame. In this case, it is not possible to make a correct system of movement inside the game. This is because the distance is described by the following formula:

$$distance = speed \times time \tag{2.1}$$

This leads to situations in which a programmer working on one piece of hardware does not notice the problem that a game on other hardware works in a completely different way. In this case, if the game runs at 30 FPS it will update the distance 30 times within one second. For 60 FPS it will already do it 60 times within one second moving the player much further. If the game has no concept of time then it does not know how to adjust the distance traveled in relation to the time elapsed. In this case, it will just blindly calculate the distance traveled based on frames, the more of which the farther the player has moved. The time dependence of certain variables can be forgotten even by the greatest creators. Thus, during the release of the PC port of Red Dead Redemption 2, players noticed that the cycle of day and night, and weather phenomena passed them by in the blink of an eye [36]. For a performance of 120 fps compared to 30 fps, the game's main character lost weight four times faster. This problem was not noticed by the developers in the console version, as there the frame counter was permanently locked at 30 FPS. In the case of the port, the frame rate depended on the capabilities of the player's hardware.

After improving the game loop, it should look similar to the one shown in Listing 2.

```
def gameLoop():
    Clock clock
    frameTimeElapsed = Time::Zero
    while window.isOpen():
        frameTimeElapsed = clock.restart()
        updateGame(frameTimeElapsed)
        handleEvents()
        render()
```

Listing 2: Pseudocode of the improved game loop.

In this way, the game is able to depend on time, and it should do so. Another problem may occur during collision checking conditions. Let's analyze an example in which the time gap between drawing two frames becomes noticeable.

In figure 2.1, it can be seen that the object in motion is headed toward the wall. In the next generated frame it is inside the wall. A collision is detected and the object can be blocked.

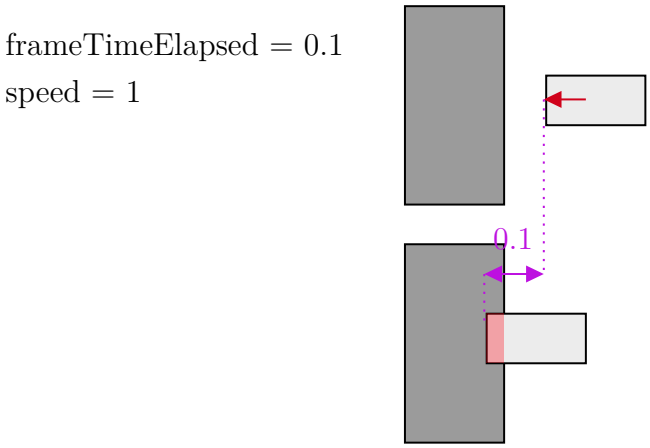

Figure 2.1: The object moves at a constant speed of 1, and the time gap between drawing the frame at the top part of the figure and drawing the frame at the bottom part of the figure is small.

The problem arises when, as a result of performance degradation and longer unresponsiveness of the game, the time between frames is so large that the calculated distance far exceeds the width of the wall as well as the width of the object. In this case, the collision is never detected. Such a situation is shown in Figure 2.2.

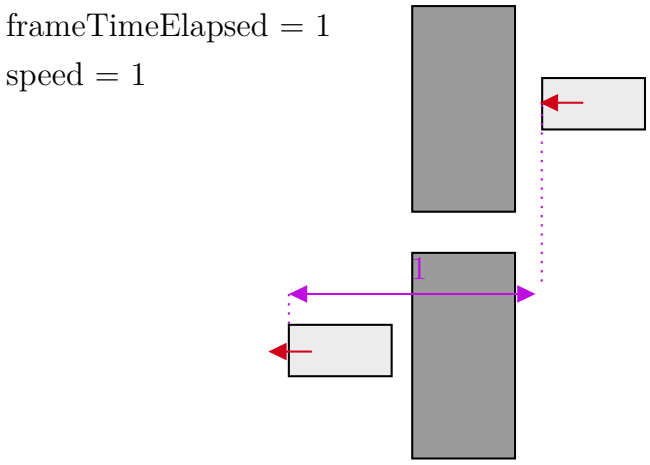

Figure 2.2: The object moves at a constant speed of 1, and the time gap between drawing the frame at the top part of the figure and drawing the frame at the bottom part of the figure is noticeable.

The solution does not necessarily result in increasing the complexity of the collision detection algorithm. What is also worth considering is improving the game loop in such a way that it would break large time gaps between frames into smaller ones. In other words, it means that it updates at least once in a given time frame. An example of such a solution can be seen in Listing 3.

```
def performFixedUpdateAtLeastMinimalNumberOfTimes(Time frameTimeElapsed):
    if frameTimeElapsed > MINIMAL_TIME_PER_FIXED_UPDATE:
        do:
            frameTimeElapsed -= MINIMAL_TIME_PER_FIXED_UPDATE
            fixedGameUpdate(MINIMAL_TIME_PER_FIXED_UPDATE)
        while frameTimeElapsed > MINIMAL_TIME_PER_FIXED_UPDATE:
    fixedGameUpdate(frameTimeElapsed)

def gameLoop():
   Clock clock
   frameTimeElapsed = Time::Zero
   while window.isOpen():
       frameTimeElapsed = clock.restart()
       updateGame(frameTimeElapsed)
       performFixedUpdateAtLeastMinimalNumberOfTimes(frameTimeElapsed)
       handleEvents()
       render()
```

Listing 3: Pseudocode of the game loop improved with a function that ensures the minimum number of updates in the given time.

The function presented in Listing 3 can be used for such things as just checking for collisions. It is still not a complete solution to the problem, but it is a function that is

predictable and gives us calls with a time frame that is never excessive. This means that when the function should execute every 10 ms, but the game was last updated 23 ms ago then on the next occasion the function will be called 2 times with the elapsed 10 ms argument passed and once with a 3 ms argument.

What is also worth mentioning, a very common loop is yet a loop with fixed time intervals. It differs from the loop in Listing 3 in that, it does not perform an additional update at the end with the rest of the remaining time. Time is also not lost as it serves as an indication of when the next update should occur. An example of such a function can be seen in Listing 4.

```
class Game:
    Clock mFixedUpdateClock
    Time mTimeSinceLastFixedUpdate

    def fixedUpdateAtEqualIntervals():
        mTimeSinceLastFixedUpdate += mFixedUpdateClock.restart()
        if mTimeSinceLastFixedUpdate > TIME_PER_FIXED_UPDATE_CALLS:
            do:
                mTimeSinceLastFixedUpdate -= TIME_PER_FIXED_UPDATE_CALLS
                fixedGameUpdate(TIME_PER_FIXED_UPDATE_CALLS)
            while mTimeSinceLastFixedUpdate > TIME_PER_FIXED_UPDATE_CALLS:

    def gameLoop():
       Clock clock
       frameTimeElapsed = Time::Zero
       mFixedUpdateClock.restart()
       while window.isOpen():
           frameTimeElapsed = clock.restart()
           updateGame(frameTimeElapsed)
           fixedUpdateAtEqualIntervals()
           handleEvents()
           render()
```

Listing 4: Pseudocode of the game loop improved with a function that updates the game with fixed intervals.

The function presented in Listing 4 can be used for collision detection as well as physics calculations. It is a very predictable function and provides updates at a constant intervals. This means that when an update should execute every 10 ms, but the last time it was executed 23 ms ago, then on the next execution it will be called 2 times with the elapsed 10 ms as an argument, and the remaining 3 ms will be kept saved for the next call. If the next call will be in the next 7 ms then the function will execute once with 10 ms as an argument.

2.1.2 Voxels

A voxel may sound unfamiliar, which is why it is much easier to compare it with the more widely known concept which is Pixel. However, the two are not the same thing. A voxel is not a pixel displayed in 3D as some might think. What then is it? To make the comparison complete, let's start by explaining what a pixel is. Pixel is a conglomeration of two words:

$$\text{Picture} + \text{Element} = \text{Pixel}$$

Pixel refers to raster graphics, in which it is the smallest element of a digital image [37]. A 2D raster graphic is a 2D grid on which individual x and y coordinates are assigned a value. Each such grid element is called a pixel. In the case of the voxel, the case is somewhat analogous. Voxel is the conglomeration of two words:

$$\text{Volume} + \text{Element} = \text{Voxel}$$

The voxel is the basic element representing volume in 3D [20]. In its own way, we can say that a voxel for 3D is what a pixel is for 2D [11]. A voxel represents the smallest addressable element. It is a representation of a single value inside 3D grid on which individual x, y, z coordinates are assigned a value.

This project will use voxels to store terrain data. There are many games such as Minecraft and Roblox that use voxels to store terrain data. Such worlds made of voxels are called voxel worlds.

2.1.3 Physics

Physics in a simple voxel world (see 2.1.2) does not require being either very accurate or sophisticated. Often in such cases, axis-aligned bounding boxes, or in other words AABB, are used, which are boxes whose axes are aligned with the coordinate axes [30]. Such a box most often surrounds an object, or a group of objects and it works as a simplified overlay that is used to test intersection with other figures designed for it. For 2D, AABB will be a rectangle, and for 3D, AABB will be a cuboid. The case in which the intersection between one AABB and another AABB is checked is often called AABB vs AABB. In this project, due to the fact that the entire game world consists of cubes, the simple AABB vs AABB collision detection will be used [2]. AABB vs AABB collisions in the 3D world are analogous to collisions in the 2D world. For this reason, it is much easier to represent them in the 2D plane. An example representation of an AABB collision in 2D space is shown in Figure 2.3. To check whether an AABB intersects another AABB it takes only one test per axis using the boundaries of the rectangles. An example of collision checking of two rectangles in 2D space can be seen in Listing 5.

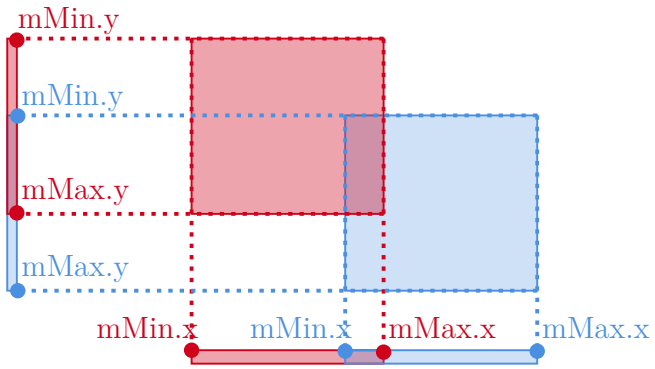

Figure 2.3: AABB collision for two 2D rectangles.

```
class AABB:
    vector2 mMin
    vector2 mMax

    def intersect(aabb):
       return mMin.x <= aabb.mMax.x and
              mMax.x >= aabb.mMin.x and
              mMin.y <= aabb.mMax.y and
              mMax.y >= aabb.mMin.y
```

Listing 5: Pseudocode for collision detection of two rectangles in two-dimensional space.

In case we want to transfer the collision test to a three-dimensional form, we add only one more test for the Z axis. An example of collision checking of cuboids in 3D space is shown in Listing 6.

```
class AABB:
    vector3 mMin
    vector3 mMax

    def intersect(aabb):
       return mMin.x <= aabb.mMax.x and mMax.x >= aabb.mMin.x and
              mMin.y <= aabb.mMax.y and mMax.y >= aabb.mMin.y and
              mMin.z <= aabb.mMax.z and mMax.z >= aabb.mMin.z
```

Listing 6: Pseudocode for collision detection of two cuboids in three-dimensional space.

Collision detection is nevertheless just part of the job. A collision must also be responded to appropriately. For example, zeroing out the player's speed at the time of a collision is already too late a reaction, because that means the player is already inside the wall and shouldn't be. Therefore, a much better approach is to react „forward".

The simplest option is to add the player's speed to his current position and then check if the player is colliding. If the player does not collide then everything is fine, otherwise we retract the player's position by the indicated velocity and reset the player's velocity. An example of such a solution can be seen in Listing 7.

```
class TransformObject:
    vector3 mPosition
    vector3 mVelocity

    def tryUpdatePosition():
        mPosition += mVelocity
        if doesObjectCollide():
            mPosition -= mVelocity
            mVelocity = 0
```

Listing 7: Pseudocode showing collision response for all axis at once.

Even better when the check is performed on all three axes separately (see Listing 8).

```
class TransformObject:
    vector3 mPosition
    vector3 mVelocity

    def tryUpdatePosition():
        mPosition.x += mVelocity.x
        if doesObjectCollide():
            mPosition.x -= mVelocity.x
            mVelocity.x = 0

        mPosition.y += mVelocity.y
        if doesObjectCollide():
            mPosition.y -= mVelocity.y
            mVelocity.y = 0

        mPosition.z += mVelocity.z
        if doesObjectCollide():
            mPosition.z -= mVelocity.z
            mVelocity.z = 0
```

Listing 8: Pseudocode showing collision response for each axis separately.

These are not very accurate collisions, but they can definitely be sufficient.

2.1.4 State Stack

What programmers should also keep in mind when creating a video game is *state stack*, which will allow to properly manage the flow of the application. *State* in this case describes independent parts of the entire application that encapsulate all the logic of many functionalities and information [15]. For many games, such states would be the game's main menu, title screen, or pause. The stack is a certain linear data structure that works according to LIFO (Last-In-First-Out) [38], which is a data handling method in which the newly added element is processed first and the oldest added element is processed last. Such a collection of elements has only two operations:

- Push, which adds an element to the collection.
- Pop, which removes the last added element from the collection.

The combination of these two concepts creates a stack that stores states, which in this project is called the state stack. The application will have the state flow as presented in Figure 2.4.

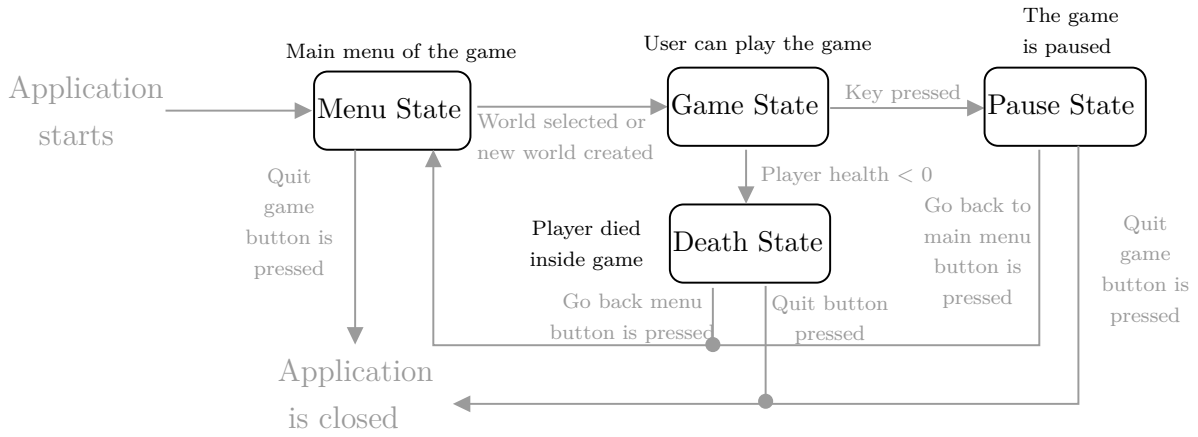

Figure 2.4: Finite-State Machine showing the application flow.

For example, when the player is in the main menu, selects the world, and starts the game, then the specified operations shown in Figure 2.5 are performed on the stack.

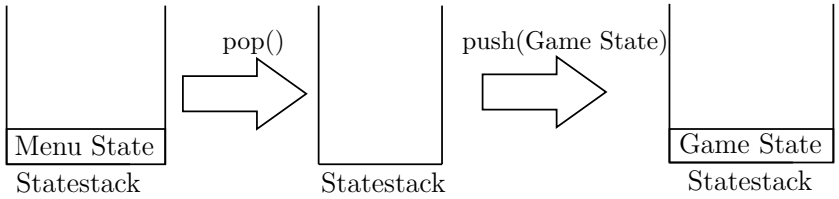

Figure 2.5: Transition from Menu State to Game State.

When a player decides to pause the game then a Pause State is put on top of the stack. It thus covers the execution of the state under it until the player resumes the game again thereby popping the Pause State from the stack (see Figure 2.6).

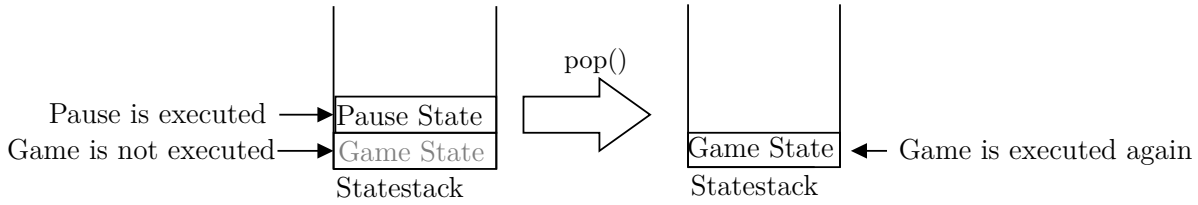

Figure 2.6: Resuming the game by turning off the pause.

The state stack is very easy to extend with additional functionality in which states can be transparent. For example, not in all video games pause means stopping the game. Some video games continue their gameplay despite the pause state. In such a case, a pause could be specified as a transparent state, and put on top of the stack would perform all its functions as before, but without blocking the execution of the state below it.

2.2 Infinite terrain generation

There are many games on the market that offer an infinitely large world model. Of course, these are infinitely large worlds only in theory, because practically there are always some limitations. For example, both big games with an infinite world model, namely No Man's Sky and Minecraft have some technical limitations. Minecraft has a world of about 3.6 billion km^2 [19] assuming that one block inside the game is 1 meter. This is an area more than 6 times larger than the surface of the Earth, and this is just one of the possible game worlds. Minecraft in Java Edition can create up to 2^{64}, that is 18446744073709551616 different worlds [1]. This is because the seed for the Java version of the Minecraft world is stored as a „*long*", which is an integer consisting of 64 bits. This raises the question, where are these worlds kept? Where do we get the space to store every single world of these 2^{64} worlds? We don't have the space for it and we don't store these worlds. What is used is called **procedurally generated terrain**. This means that games with infinitely large terrain just know the recipe for creating terrain. The terrain is created on the fly around the player on his horizon. Due to the fact that this project is based on a game set in a voxel world, this work will discuss the approach of world generation typical for voxel worlds.

2.2.1 Chunks

What is really generated around the player are *chunks*. Chunks are made up of a predetermined number of columns of blocks. In Minecraft, chunks come in the size of

16x16 with up to 384 blocks high, which gives 98304 blocks in each chunk [5]. The sizes of chunks in the Minecraft game have increased over time. For earlier versions of the game, it was 256 and even 128 blocks in height. The size of the chunk translates directly into the loading time of the world and its responsiveness.

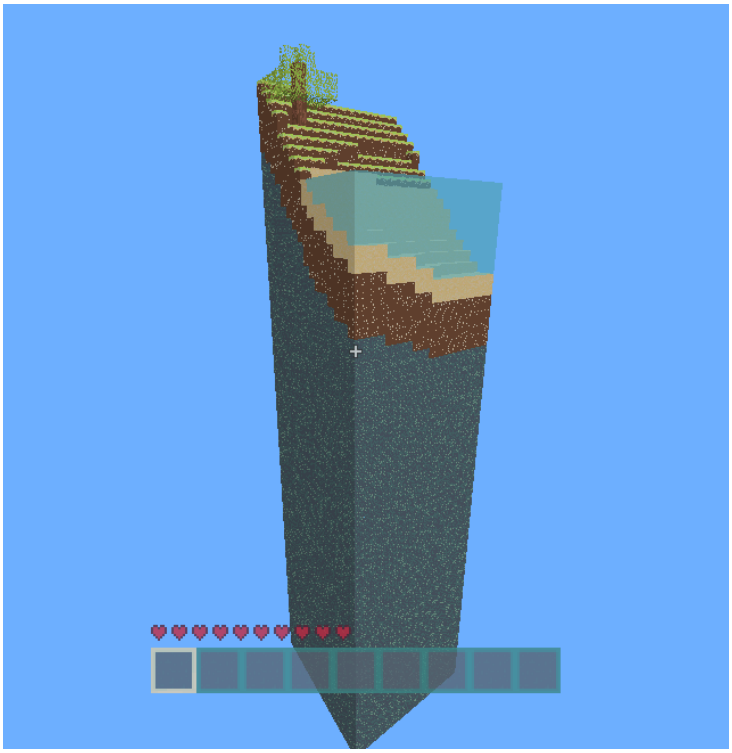

Figure 2.7: Chunk of size 16x128x16, with the upper part mostly filled with air blocks.

Creation of chunk

The chunk for each block within it has to answer the question „what block should go here?". To do this, very often at a perlin noise function [31] is used, or even several perlin noise functions to obtain the terrain height. Perlin noise is a type of gradient noise, which is a function that takes a coordinate for any dimension and returns a real number between -1 and 1. What distinguishes it is that this function generates coherent noise — that is, noise for which the transition between two points occurs smoothly without any discontinuities. Perlin noise is ideal for generating terrain, due to its very „smooth" sequences of pseudo-random numbers that give the impression of terrain continuity. The perlin noise itself is also easy to customize. The most commonly used parameter is *octaves* which are the result of adding several noise functions of different frequencies and amplitude, which add to the noise the illusion of complexity, depth and make it look more natural [35].

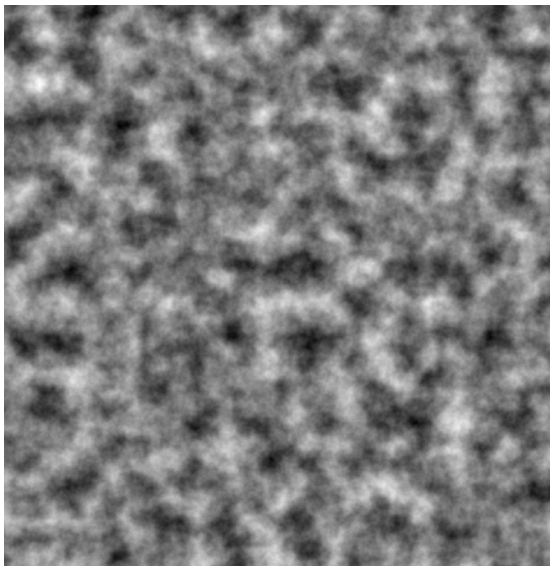

Figure 2.8: Perlin noise obtained from the FastNoiseLite Gui application [3].

The original Minecraft terrain generation system has much more complexity, and it is too complex for this project. What is worth knowing is that the original Minecraft uses as many as three perlin noises to generate the overall terrain height, and each noise configuration is also assigned its own spline function, which determines the height of the terrain [19]. Their use is also more complicated and this way will not be used in this project. When it comes to the terrain shape itself, the original Minecraft uses 3D perlin noise. It allows to generate structures that are wider at the top than at the bottom. These can also be beautiful mountains floating above the flat terrain below. Using only 2D noise as a heightmap will lead to a smooth terrain. Such a situation that the mountain „leans" over the flat terrain underneath will never occur.

3D perlin noise does not create such structures out of place, and it needs to be adjusted accordingly. The most common way to do this is by setting a terrain level and then increasing the chance of an air block above that terrain, and decreasing that chance below the terrain level. However, due to the complexity, this technique also will not appear in this work, and instead 2D perlin noise will be used for the basic terrain.

After determining the height of the terrain, the chunk must determine what block is at what location. These blocks depend on the biome (see 2.2.2) that is located in the area. An example function showing the assignment of blocks to the terrain can be seen in Listing 9.

```
    for y in range(BLOCKS_PER_Y_DIMENSION):
        if y == surfaceLevel:
            chunkBlocks[x][y][z] = BlockId::SnowyGrass
        elif y < surfaceLevel - 5:
            chunkBlocks[x][y][z] = BlockId::Stone
        elif y < surfaceLevel:
            chunkBlocks[x][y][z] = BlockId::Dirt
        elif y == SEA_LEVEL - 1:
            chunkBlocks[x][y][z] = BlockId::Ice
        elif y < SEA_LEVEL - 1:
            chunkBlocks[x][y][z] = BlockId::Water
        else:
            chunkBlocks[x][y][z] = BlockId::Air
```

Listing 9: Pseudocode, showing an example of block assignment in chunk depending on the position of Y.

Chunk Container

Each chunk must be in a container that allows easy access to other chunks. The most common implementation is a hashmap that returns the individual chunk based on two parameters x and z. Most often it works the following way: when there is a need to get a block with given world coordinates, the chunk coordinates of a chunk that contains the given block are calculated. Then in the chunk itself, the local coordinates of the block depending on the dimensions of the chunk itself are used.

Chunks are created according to the spiral motion around the player (see Figure 2.9).

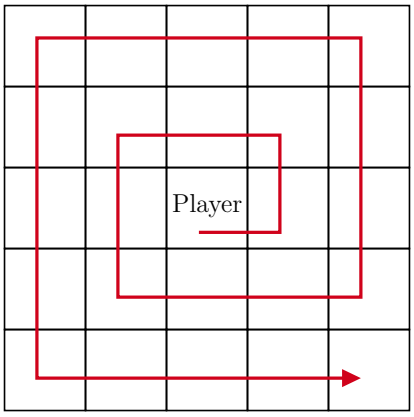

Figure 2.9: The order in which chunks should be created around the player.

Such a function queuing chunks should have some limit on how many chunks should be queued. Assuming that chunks are created in a separate thread then its completion should not take too long, so one thread can't have too many chunks queued together. Otherwise,

the player will unnecessarily wait for a huge set of chunks to appear at once. This may even lead to a situation where the player has already moved from the place where the new chunks will appear. The game must have a chance to generate those chunks closest to the player including when the player is in motion. That's why it's a good idea, for example, to periodically repeat the entire spiral around the player and queue only the limited number of new chunks that have not yet been built.

Mesh Generation

If one wanted to display the entire contents of the chunk on the screen, the performance will be very low. The mesh of the chunk should contain only its outer walls from which it is built. An example algorithm that achieves this in the simplest way is for each face of each block to check whether its neighboring block is an air block or any other transparent block. If this is the case, such a block face can be noticed by the player during the game, so it is added to the overall chunk mesh. An example of such an algorithm can be seen in Listing 10.

```
1 for block in chunk:
2     for blockFace in block:
3         if doesBlockFaceHasTransparentNeighbor(blockFace):
4             meshBuilder.addQuad(blockFace)
```

Listing 10: Pseudocode showing the generation of the chunk mesh.

The problem arises when we want to check the blocks on the edge of the chunk. Blocks on the edge of the chunk very often also should not to be drawn, because as a result, underground chunks will be separated by solid drawn walls, which are not visible to the player anyway. For this reason, the chunk must be able to refer to the container it is in to know about blocks located in a neighboring chunk. This will make the chunks fit together perfectly as if they form one perfect whole.

Mesh Rebuilding

Mesh rebuilding is the process of building a chunk's mesh again. A game of this type rebuilds meshes very often and these are very expensive operations. The mesh has to be rebuilt whenever the structure of its blocks is violated. This can be a destroyed block, or a new block placed on it. Rebuilding the mesh should be done as soon as possible. If it takes too long to rebuild the mesh, the player may experience a visual delay in which the block is still in place for a while after being destroyed.

The smaller the chunks are, the shorter time it takes to rebuild them. To speed up rebuilding, a chunk can be divided vertically into smaller chunks. It is also possible to create

chunks that resemble a cube (30x30x30 chunks for example), but the approach in which chunks are generated in all directions of the world also has its disadvantages. Limiting the bottom and top is better at controlling the player's gameplay than, for example, allowing infinite digging into the ground. The player would also rather see everything below and above them than wait for the world to generate at the bottom and top.

The rebuilding operation becomes all the more costly when it happens on the edges of the chunk. In the worst scenario, the destruction of one block can rebuild as many as three chunks.

In Figure 2.10, four chunks can be seen. The orange color indicates the destroyed block. Destroying it reveals the face of the two neighboring blocks it previously covered. If the two chunks containing these blocks are not rebuilt then there will be a visual hole in the map. Therefore, the destruction of one block leads to the rebuilding of three chunks.

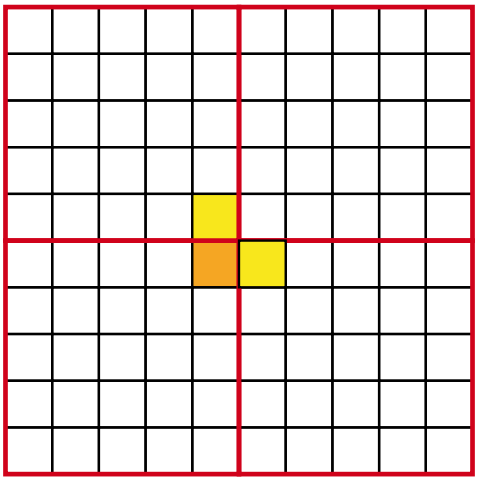

Figure 2.10: Four adjacent chunks, where the orange block shows the destroyed block, and the yellow blocks are blocks that reveal the previously invisible part of them.

2.2.2 Biomes

The biomes in this project depend on two noises: temperature and humidity. For example, for high temperature and low humidity, it will be a desert biome. For very low temperatures, it is the winter biome. Each biome has its own heightmaps, and decorate chunks according to its own pattern. The two different heightmaps, however, create the problem of a sudden terrain jump between two different biomes.

The problem does not occur in chunks that have only one biome in them and up to one block around the chunk, so calculating the terrain height consistent with a given biome for each x and z coordinate is sufficient. The problem appears on the transitions between biomes, as they need to be smoothed. In this case, chunks that contain more than one biome have the terrain height calculated differently.

At chunk corners, and adjacent corners of other chunks, the height of the terrain is calculated (see Figure 2.11). Based on these values, an average is calculated and is assumed to be the height of the terrain at the corners of the chunk (see Figure 2.12).

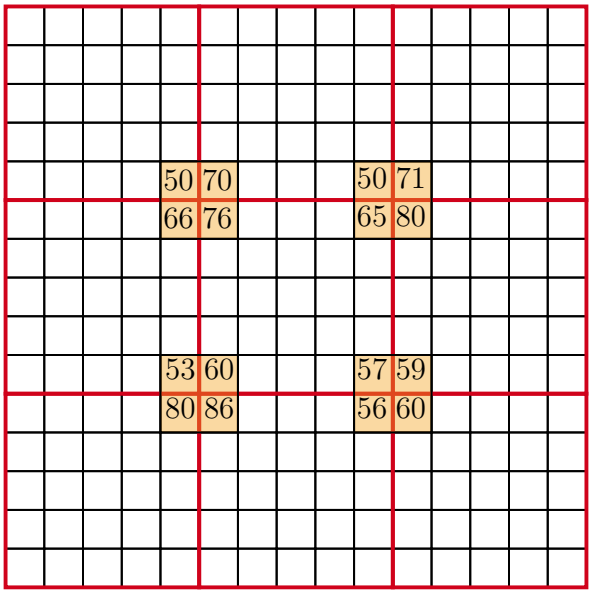

Figure 2.11: In the corner of a chunk, the height of the chunk's terrain is calculated as well as the height at the corners of neighboring chunks.

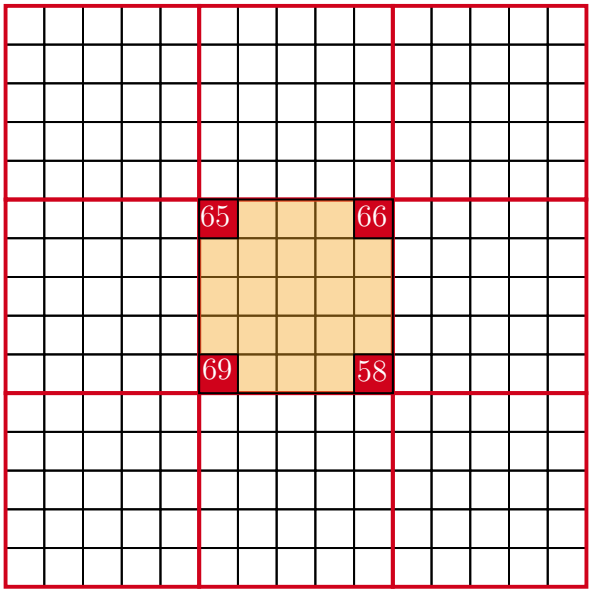

Figure 2.12: The red corners show the average of the four terrain elevations. The orange boxes will be filled in linearly creating a smooth transition between the values from the four corners.

Then, using bilinear interpolation, the transition values between the corners are calculated and are used as the terrain height between the corners of the chunk. Perlin noise is no longer used here to determine the height of the terrain.

Chapter 3

Requirements and tools

3.1 Functional requirements

3.1.1 Saving and loading game

The player should be able to create a new world and be able to save the game state at any time in order to load the saved world later and start the game again from the same point in time while maintaining, among other things, the state of the inventory.

3.1.2 Main Menu

The player should be greeted by the main game menu from which they should have the options to leave the game and to start a new game. The player should be able to return to the main menu at any time during the game.

3.1.3 Pause

The player should be able to pause the game at any time by blocking its execution in the background while displaying the corresponding pause user interface. If the game is paused, the player should be able to restore the game again, or save it and return to the main menu, or save it and quit the game.

3.1.4 Basic physics

The player should be able to walk around the game world through simple in-game physics that make the player move on the surface of the game world. If there is no ground beneath the player, the player should start falling. On contact with any surface, the player should stop on it, not go through it.

Swimming

When in contact with water, the player's movement should behave differently than when on the surface. The player should be able to move in all directions, swim back to the surface, or sink to the bottom of the water in a slow manner.

3.1.5 Infinite world generation

The game world should be continuously generated as the player approaches the map boundary. The world beyond the player's reach should not be drawn. Parts of the map that have been modified by the player should remain in place in case the player later returns to that location and loads it again (see 3.1.1).

3.1.6 Diversity of the game world

The game world should be diverse. The terrain should have decorations such as trees and should have changing terrain elevation and lakes.

Biomes

The terrain should be divided into biomes, which have their own way of generating terrain as well as decorating it. For example, a desert terrain could be flat, and full of sand and cacti, while a forest terrain could contain uplands and simple trees.

3.1.7 Interaction with the game world

The player should be able to interact with the game world by destroying and placing new blocks in the game world.

3.1.8 Inventory

The player should have an inventory to store the items they have collected and which they can use to place them in the game world.

3.1.9 Health bar

The player should have a health bar that indicates their current health status. The game should reduce the life level depending on interactions in the game world such as:

- A player falling from a height.
- Staying underwater too long.

3.1.10 Oxygen bar

The player should be able to see the oxygen bar while underwater. When the oxygen bar runs out the player starts to lose health as a result of which they may die. Oxygen levels should return to maximum levels as soon as the player is back on the surface (out of the water).

3.1.11 Crafting System

The player should be able to create other items based on currently owned items. The crafting system should allow the player to arrange items in a 3x3 grid and when the player guesses the correct arrangement they can get another item in exchange for the arranged items.

3.2 Non-functional requirements

3.2.1 Intuitive controls

The controls should be intuitive and resemble standard controls for the first-person perspective type of games.

3.2.2 Stable performance

The game should have a constant framerate. For example, it would be unexpected for the player's game to freeze when the world is generated. Such things should happen in the background and not block the main execution of the game. The game should limit the drawing of parts of the map that are not visible to the player, such as all the blocks underground.

3.2.3 Modifiable

The game should be modifiable by the game files. For example, changing texture packs should be possible and game configuration files should be accessible.

3.2.4 Modern

The project should not use outdated, abandoned technologies or software engineering techniques.

3.3 Use case modelling

The use cases are divided into two diagrams related to:

- the main menu — shown in Figure 3.2,
- the gameplay — shown in Figure 3.1.

3.3.1 Actor candidates

Actor Candidate	Role Description
Player	The player in the main menu may wish to start or close the game. In the case of starting the game, the player must create a world, or select an existing one (see Figure 3.2). Once the game has started the player has the option to (see Figure 3.1): • Open the inventory and optionally create a new item, or move existing items around in the inventory. • Place a block. • Destroy a block that may or may not allow itself to be collected. • Pause the game and can return to the main menu, or quit the game which will result in the game world being saved. • The player can get damage that can kill them which will result in a respawn, or a return to the main menu and a world save.

Table 3.1: Actor candidates and their role descriptions.

3.3.2 Use case diagrams

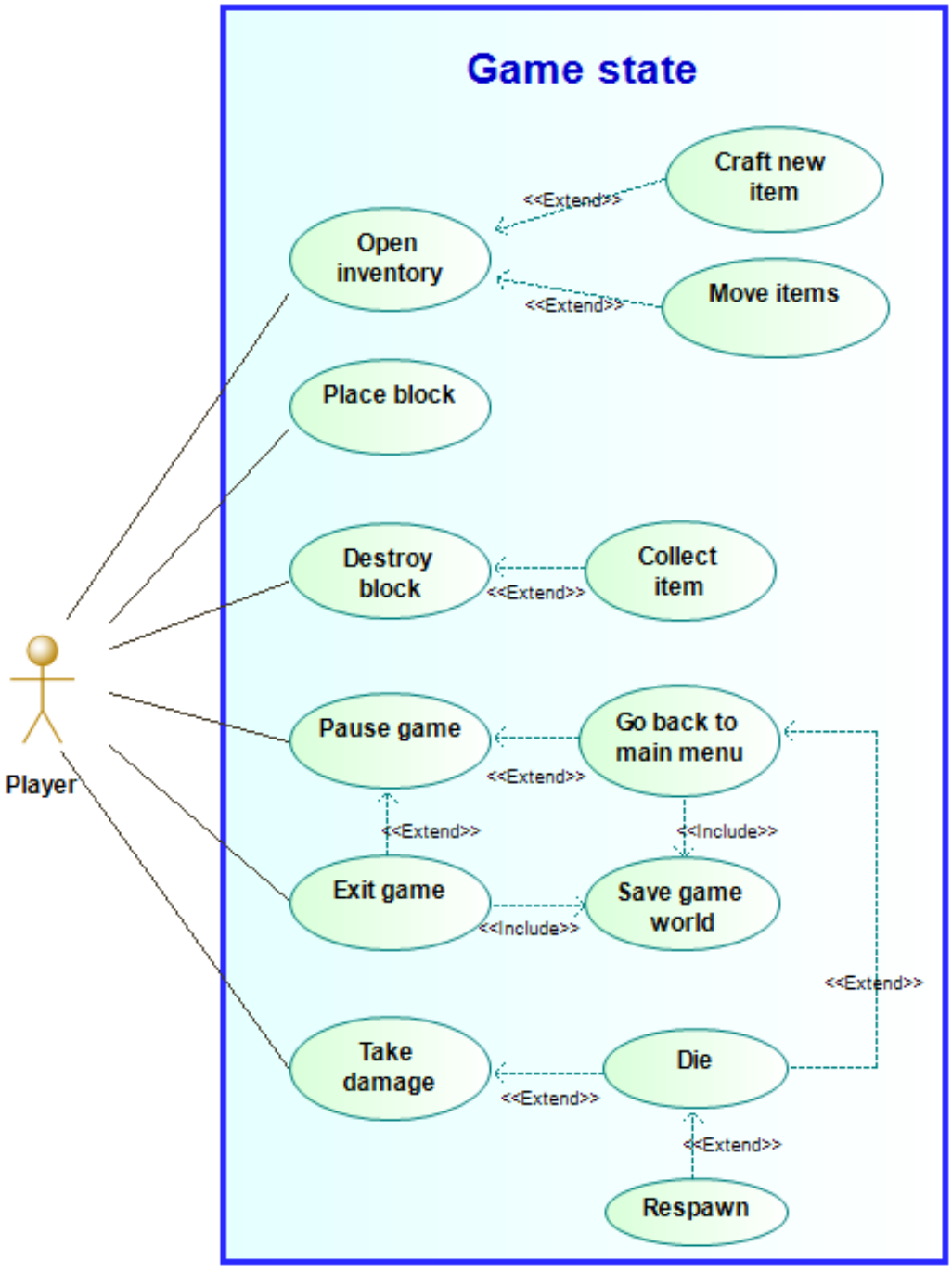

Figure 3.1: Use case for the gameplay.

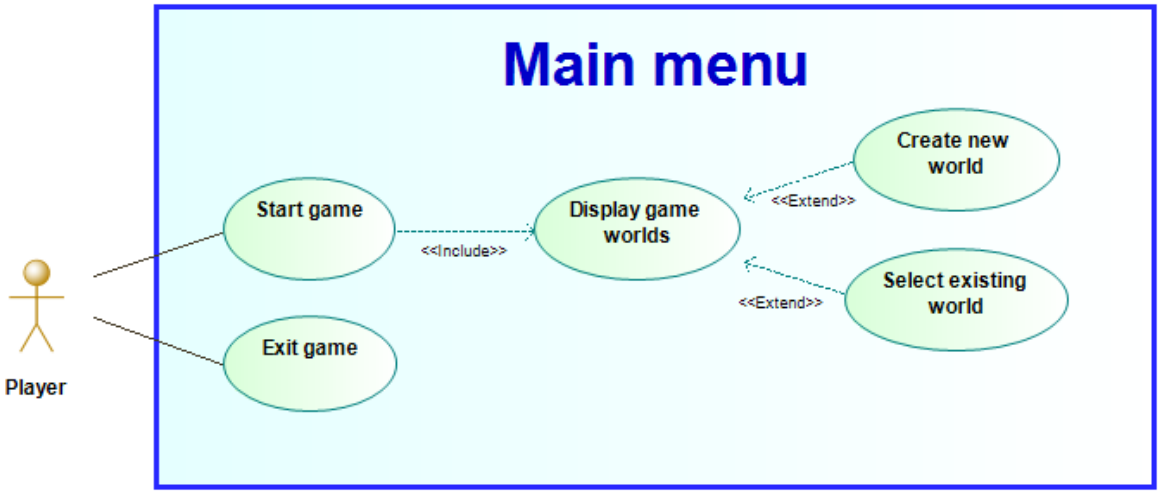

Figure 3.2: Use case for the main game menu.

3.4 Implementation Assumptions

3.4.1 Libraries

- **FastNoise Lite** — A noise generation library focused on high performance and a wide range of noise algorithms. Library needed to implement world terrain generation [22].
- **GLEW** — A library that allows checking at runtime which extensions are present and which function can be used. It provides OpenGL functions as well as OpenGL extensions [26].
- **GLM** — It is a mathematical library based on the OpenGL Shading Language (GLSL). It contains a collection of classes and functions useful for creating vectors or matrices for OpenGL applications [7].
- **ImGui** — A library for creating a user graphical interface. It is very often used for tool development, or for debugging purposes. This project uses it precisely for debugging purposes [28].
- **ImGui-SFML** — A library that allows to easily combine the *SFML* library with the *Dear ImGui* library [9].
- **Serializer** — It is a serialization framework that allows saving the objects directly to files in binary format. In this project, it will be used to implement a system for writing and reading the game world [10].
- **SFML** — It is a fast and simple cross-platform library that allows easy access to windows, graphics, audio, or network. In the project, it will be used to create a

window, manage user input, or even draw 2D elements to the window [32].

- **TGui** — This is the GUI library that will be used to draw user interface in-game such as buttons, or sliders [33].
- **GoogleTest** — It is a library that allows writing tests, and mock classes. It will be used to test the application [13].

3.4.2 Technologies and tools

- **CMake** — A multi-platform tool that allows to automatically manage of the compilation process. With this tool, other libraries in use do not have to be kept in one repository together with the project. CMake allows them to be downloaded at build time and automatically linked into the project [6].
- **C++20** — C++ standard to be released in December 2020 [4].
- **MSVC Compiler 14.34.31933** — A compiler developed by Microsoft. The project will be built and run on this compiler [25].
- **CLion 2022.2** — JetBrains' multi-platform integrated development environment for C/C++ languages. The project is created using this IDE [21].
- **Clang-Format** — C++ code formatter to ensure a stylistically consistent code style and format [23].
- **Git** — A distributed version control system for tracking code changes [18].
- **Github** — A web hosting service that allows hosting development projects using the Git version control system [12].

3.4.3 File structure

As can be seen in Figure 3.3, the file structure is divided into resources used by the game (resources), the game's source code (src), tests, and external libraries (vendor). The vendor folder only contains folders with CMakeLists.txt only, as they have the information on where each of the libraries needed to build the project should be downloaded from.

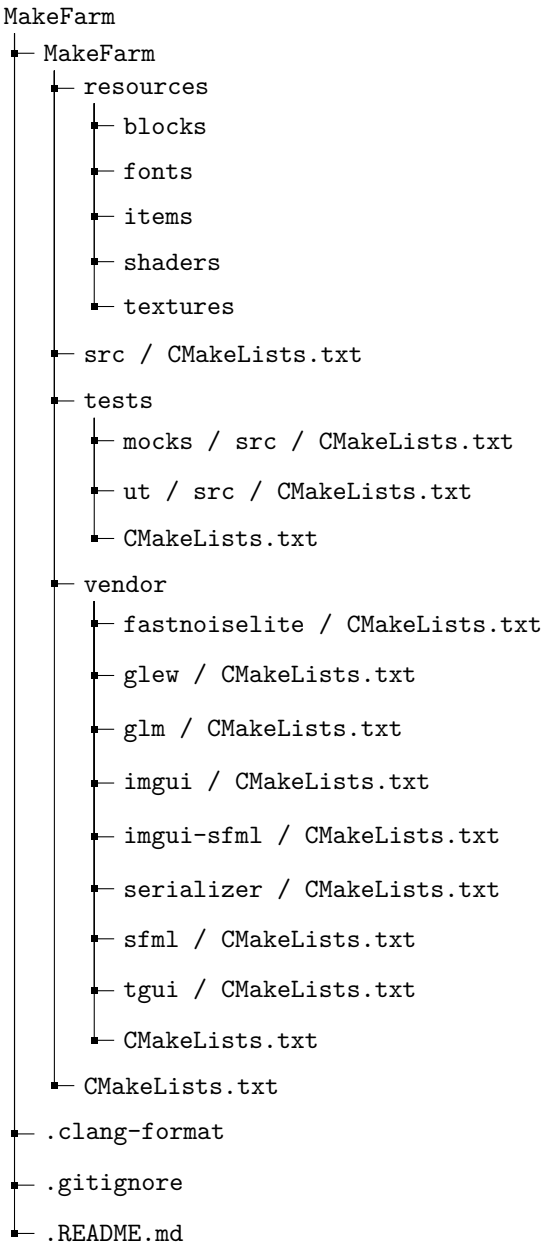

Figure 3.3: File structure of the project.

3.5 Development methodology

All development was carried out using the git version control system. During development, 97 commits were created, of which 51 commits were on master. Feature work was carried out on separate branches and then merged and squashed to master. Through the use of *CMake* and *Github Actions* together, each commit was verified to ensure that it was building correctly and that the tests were passing. Such confirmation would appear next to the commit in the form of a green pipe when everything succeeded and in the form of a red X when something failed. Commits were named according to the generally known commit message guidelines [29].

Chapter 4

External specification

4.1 Hardware and software requirements

The game supports building under Windows, so Windows 7 or higher is recommended. The game was tested on hardware with a configuration of:

- **Processor:** Intel Core i5 13600k.
- **Motherboard:** Gigabyte Z690 Gaming X DDR4.
- **RAM:** 48 GB DDR4.
- **Graphics Card:** AMD Radeon RX 6700 XT.

On which no performance issues were noticed and the framerate was locked at 120 fps.

4.2 Installation procedure

4.2.1 Pre-built executable file

If the game has been supplied in an already pre-built form, all that is required is to run the *MakeFarmApp.exe* executable file.

4.2.2 Building the project using CLion 2022.2

To build a project, the user will need a CMake version of at least 3.16, an internet connection, and an MSVC compiler that supports the C++20 standard.

1. Start CLion and select the option to open a new project (see Figure 4.1).

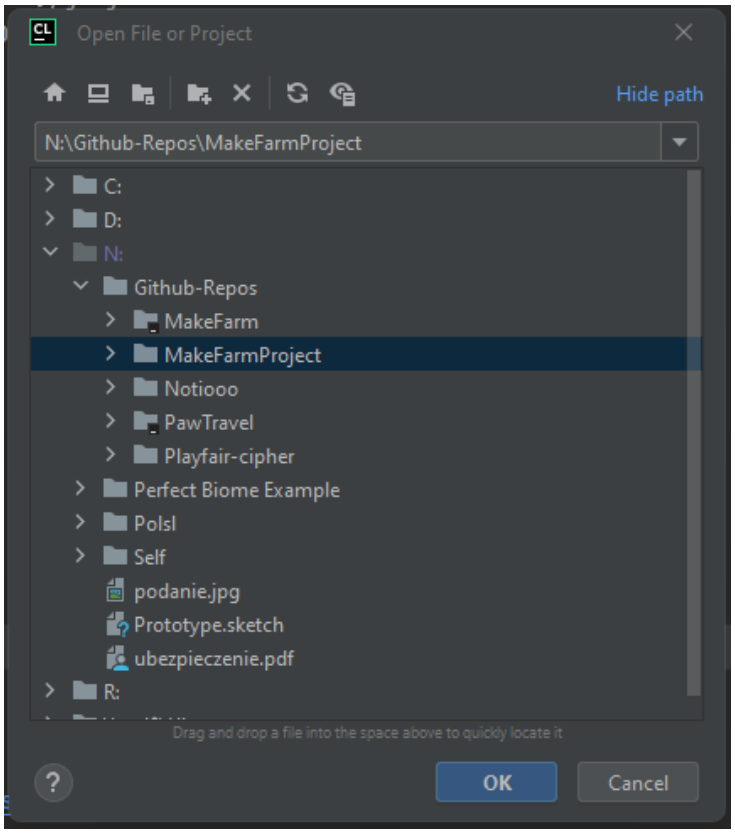

Figure 4.1: Selection of a new project in the „Open File or Project" tab.

2. There is a CMakeLists.txt in the main MakeFarm folder, which the user has to right-click on and select „Load CMake Project" (see Figure 4.2).

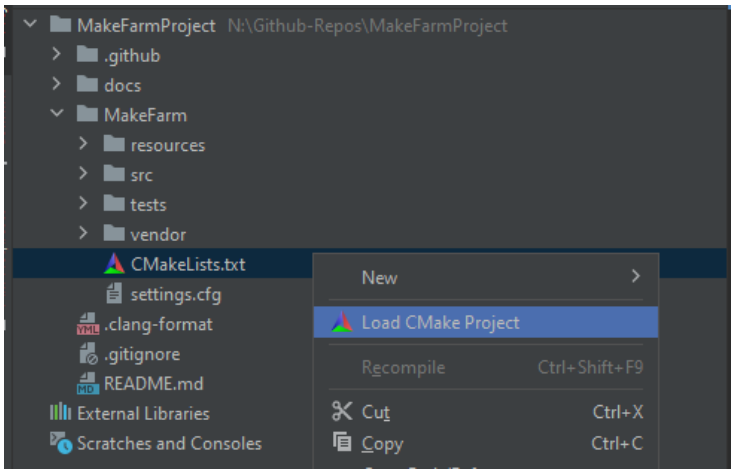

Figure 4.2: Loading the MakeFarm project from the main CMakeLists file.

3. Downloading external libraries and building them will begin (see Figure 4.3).

```
"C:\Program Files\JetBrains\CLion 2022.2.4\bin\cmake\win\bin\cmake.exe" ...
-- Fetching SFML...
-- SFML Fetched!
-- Fetching ImGui...
-- ImGui Fetched!
-- Fetching ImGui-SFML...
-- Found ImGui v1.89 WIP in N:/Github-Repos/...
-- ImGui-SFML Fetched!
-- Fetching GLM...
-- GLM: Version 0.9.9.9
-- GLM Fetched!
-- Fetching FastNoiseLite...
-- FastNoiseLite Fetched!
...
```

Figure 4.3: Sample output displayed in the „CMake" tab when the project is being loaded.

4. The build should end with the line: *[Finished]* (see Figure 4.4).

```
...
-- Configuring done
-- Generating done
-- Build files have been written to: N:/Github-Repos/...

[Finished]
```

Figure 4.4: The final output in the „CMake" tab showing the correct loading of the project.

5. Then select *MakeFarmApp* from the list of available configurations and click „Run MakeFarmApp" (see Figure 4.5).

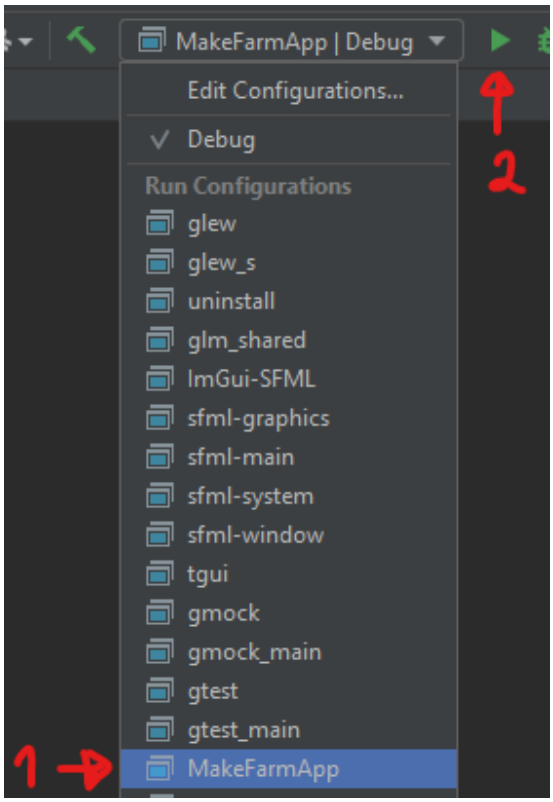

Figure 4.5: List of available configurations to build.

6. Once the compilation is complete, the game window should launch. If there any problems occurred please check in the CLion settings that under *Build, Execution, Deployment > CMake* the Toolchain is set to *Visual Studio*.

4.3 User manual

4.3.1 How to start

To start the game, simply run the .exe executable file. The game window with the main menu should appear on the screen (see Figure 4.6).

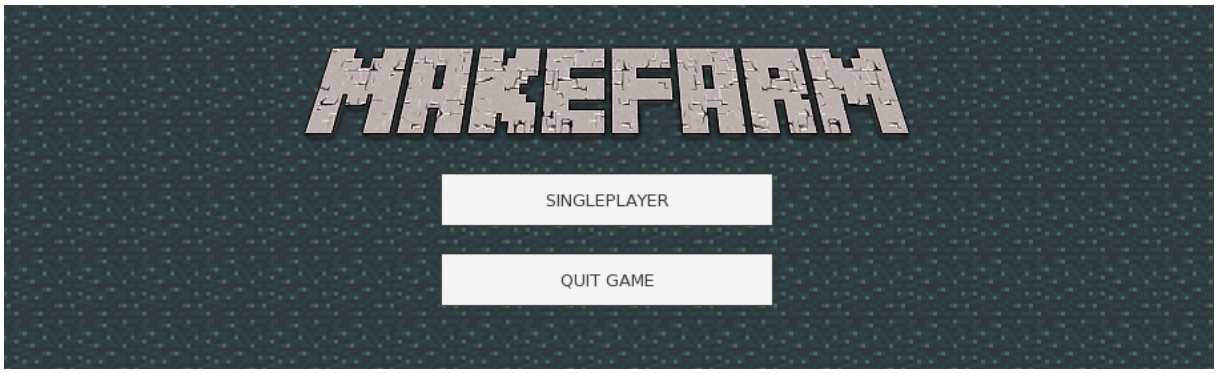

Figure 4.6: The main game menu.

From this level, it is possible to exit the game by clicking „Quit Game". Selection, or creation of a world is possible by clicking „Singleplayer" (see Figure 4.7).

4.3.2 Loading world

If the game already has worlds created, they will be visible in this view. In Figure 4.7 there can already be seen the two created worlds „Custom named world" and „New world". In this case, all that is needed is to click on an existing world and the game will start.

Figure 4.7: View of world selection after clicking on the „Singleplayer" option.

4.3.3 Creating new world

When creating a new world, all that have to be done is to click „Create New World". If the world is not named in any way, it will simply be called „New world" and any subsequent worlds created will be called „New world (1)", „New world (2)" and so on. In Figure 4.7, the newly created world would be called "My new world name" thanks to the name entered in the field to the left of the button.

4.3.4 Control basics

To move around the game world use the **W, A, S, D keys,** which allow to move in the four directions of the world. To jump press the **spacebar**. The spacebar also allows to swim upwards when the player is in the water. In the game, player can look around using the mouse.

4.3.5 Hotbar

Hotbar (see Figure 4.8) is a quick access bar around which there is a lot of additional information. The hotbar contains 9 slots for storing items. The currently selected slot is marked with a white and gold border. Each item has its own texture, and in the upper left corner of the slot appears in what quantity the item is in the slot. In the middle above the hotbar displays the name of the currently selected item.

Health

Above the left side of the hotbar the current health status can be seen. If all the hearts are light-red then the player has the maximum amount of life, if all the light-red hearts disappear and are replaced by dark-red hearts then the player dies. One example of possible loss of life is a fall from a great height onto a hard surface. Water in this case cushions the fall.

Oxygen

In addition, when the player is underwater, the current oxygen level is displayed above the hearts. If all the bubbles disappear and the player is still underwater then the player will start to lose life. Emerging from the water renews the oxygen level to the maximum possible level.

Figure 4.8: An in-game hotbar that contains items for quick access.

4.3.6 Destroying blocks

To destroy a block, all it takes is pointing the crosshair at the block to be destroyed and then pressing the **left mouse button**. Depending on block properties, the destroyed

block may or may not appear in the inventory. An example of this operation can be seen in Figures 4.9 and Figure 4.10. It is worth noting that in these Figures can be seen that after destroying the block, the number of blocks in the inventory increased by one.

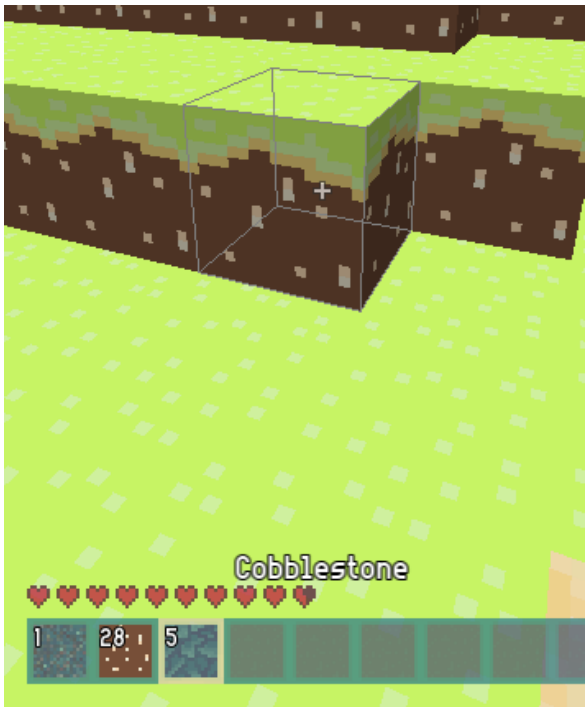

Figure 4.9: Screenshot of the game before the block was destroyed.

Figure 4.10: Screenshot of the game after the block is destroyed.

4.3.7 Placing blocks

To place a block, the player must first have a block in the inventory (see 4.3.5 and 4.3.8). Then use the mouse scroll to select the block in the hotbar that should be placed (see 4.3.5). To place a block, simply aim with the crosshair inside the game at the place where the block should be placed. Then simply click the **right mouse button**. The block will appear adjacent to the indicated block wall. See the Figure 4.11 before the block is placed and the Figure 4.12 after the block is placed. It is worth noting that on the figures (see 4.12 and 4.11), the amount of blocks in the inventory decreased by one after placing the block. Each time a block is placed, it always requires having a block and then losing one quantity of it from the inventory.

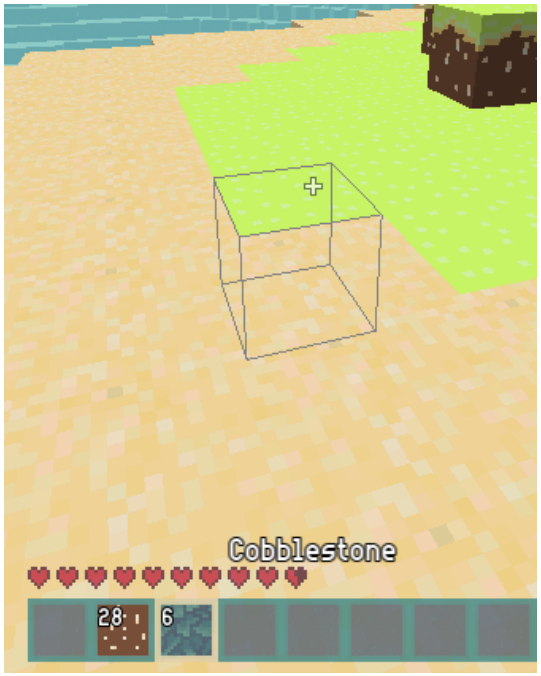

Figure 4.11: Screenshot of the game before the block was placed.

Figure 4.12: Screenshot of the game after the block is placed.

4.3.8 Inventory

The full inventory can be found under the **E** key. While the hotbar only shows 9 slots of space, the entire inventory allows to store 36 different items. The equipment is shown in Figure 4.13.

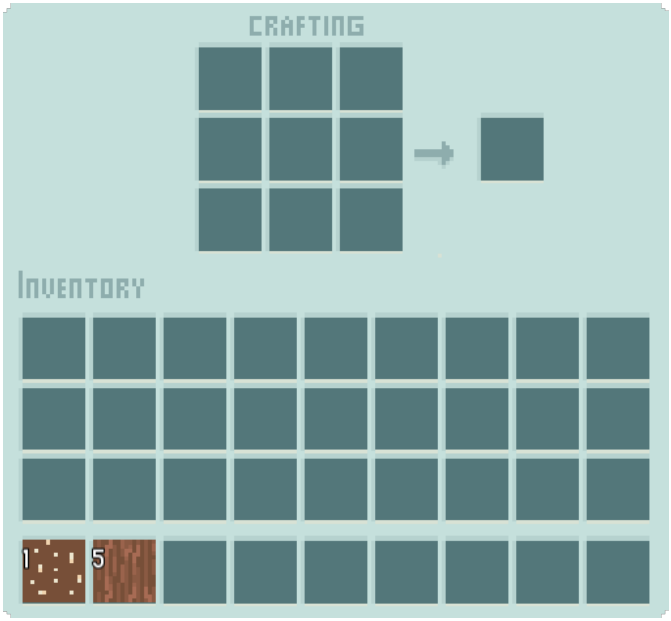

Figure 4.13: In-game equipment.

Moving around the inventory

To take and move the blocks with the mouse, press the **left mouse button**. To put away all the blocks being held by mouse, click with the left mouse button on a free slot. If player left-clicks on an item of the same type as the one being held by the mouse, their quantities will add up. Use the **right mouse button** to insert only one item at a time into a slot. Example operations can be seen in Figure 4.14.

Figure 4.14: Examples of operations performed in the inventory.

Items and quantities

In most cases, blocks and items can be stored in a maximum of 64 units. This means that one slot can usually only hold 64 blocks, or other items. Some items nevertheless have their own limitations and sometimes a slot can hold fewer items than 64. An attempt to combine two items with a total quantity greater than 64 is shown in Figures 4.15 and 4.16

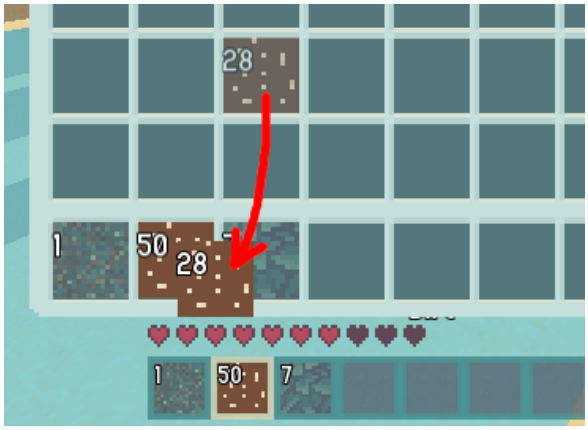

Figure 4.15: Player attempts to combine two blocks with a total of more than 64.

Figure 4.16: As a result of merging, 64 blocks are inside the slot and the mouse holds the remaining 14 that did not fit.

4.3.9 Crafting

Crafting can be found directly in the inventory under the **E** button. Crafting makes possible that using different types of items with the right arrangement in crafting-zone (see Figure 4.17) results in obtaining another item in different quantity.

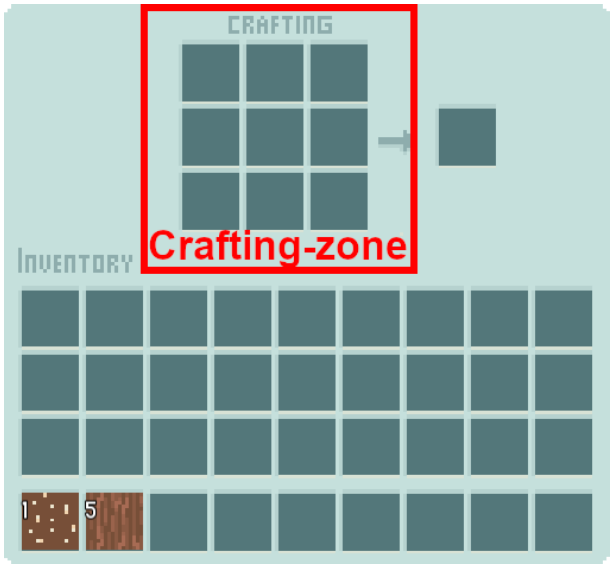

Figure 4.17: Crafting-zone as part of the inventory.

An example of the crafting process can be seen in figures 4.18 and 4.19.

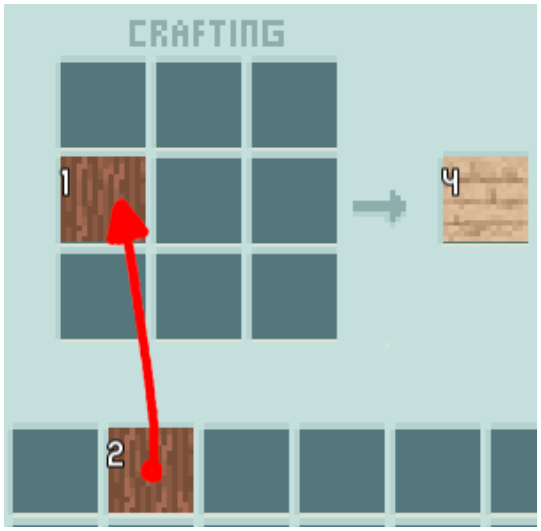

Figure 4.18: The player inserts one block of wood into the crafting-zone, which is the correct arrangement for creating the planks. The planks appear as the proposed crafting result.

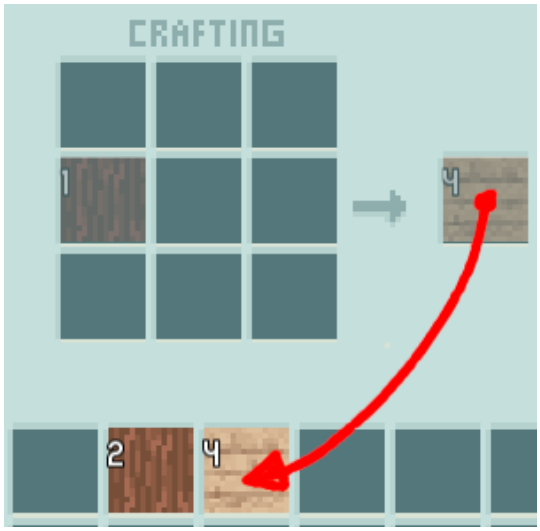

Figure 4.19: The player takes away the result of crafting. When the resulting crafting item is caught with the mouse, the items in crafting-zone responsible for creating the new item disappear.

4.3.10 Death and respawn

In the event of death, the game screen turns red and the player has two options: save the game and return to the main menu or respawn (see Figure 4.20). In the case of a respawn, the player appears at the starting point from where they started and the player's inventory is cleared. Of course, by selecting the options to return to the main menu and returning to the world later, the player will be in the same situation.

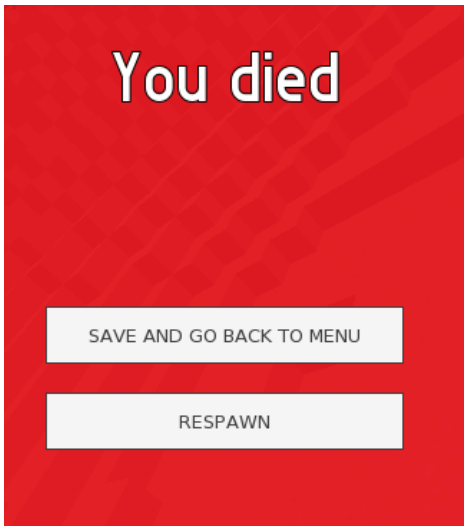

Figure 4.20: Death screen displayed in the event of a player's death.

4.3.11 Pause

To pause the game, simply click the **ESC** (Escape) button. The game in the background stops and the screen becomes darker and two buttons appear in front of the player: save and exit the game, and save and return to the main menu. The pause window can be seen in Figure 4.21.

Figure 4.21: Pause screen.

4.3.12 Developer options

When the **F4** button is pressed, additional developer options are activated. These are the player's current position, additional drop-down menu (see Figure 4.22), and the frame rate in the top right corner (see Figure 4.23). In addition, the **F1** button allows the cursor to be detached from the game, so that the developer interface can be used. The **F2** button takes half heart of life away from the player. The **F3** button allows the player to fly, where in addition the **spacebar** is responsible for flying upwards and the left shift for descending.

Figure 4.22: Additional developer options.

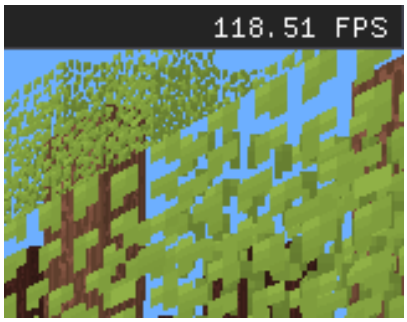

Figure 4.23: Frame rate counter.

Texture Packs

By choosing from *MakeFarm* > *Texture Packs*, textures can be changed at runtime. An example of the effect of changing from default to alternative textures can be seen in Figure 4.24.

Figure 4.24: Switching from default textures to alternative textures.

Wireframe

By selecting *OpenGL > Switch Wireframe*, it is possible to switch the view to the default or wireframe at runtime. An example of such a switch can be seen in Figure 4.25.

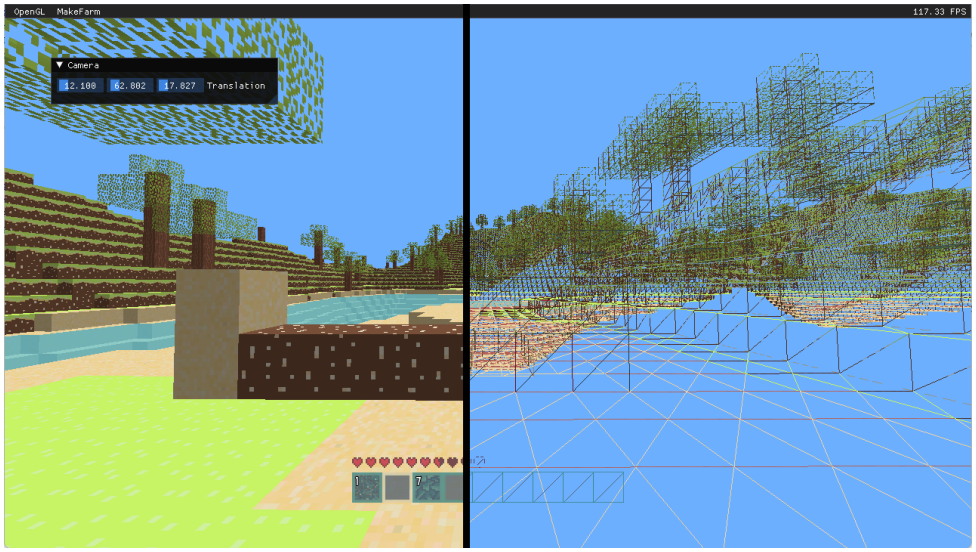

Figure 4.25: Switching on and off the wireframe mode.

4.4 Working scenarios

Most of the functionality has already been shown in the Section 4.3, so mostly the scenery of the game world will be presented here.

Figure 4.26: A very dryland biome, it is most often a transitional biome between forest and desert. Dead trees are present. Its terrain is very flat.

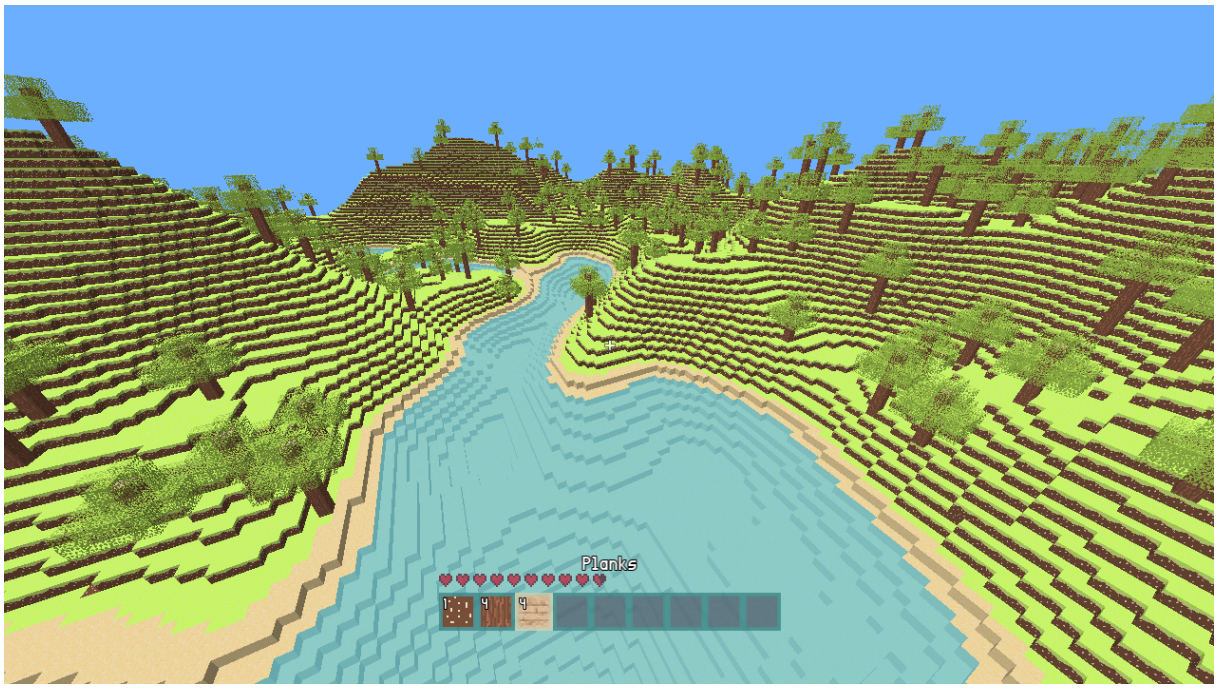

Figure 4.27: A forested biome, its terrain has quite a few hills and lakes. Trees are present.

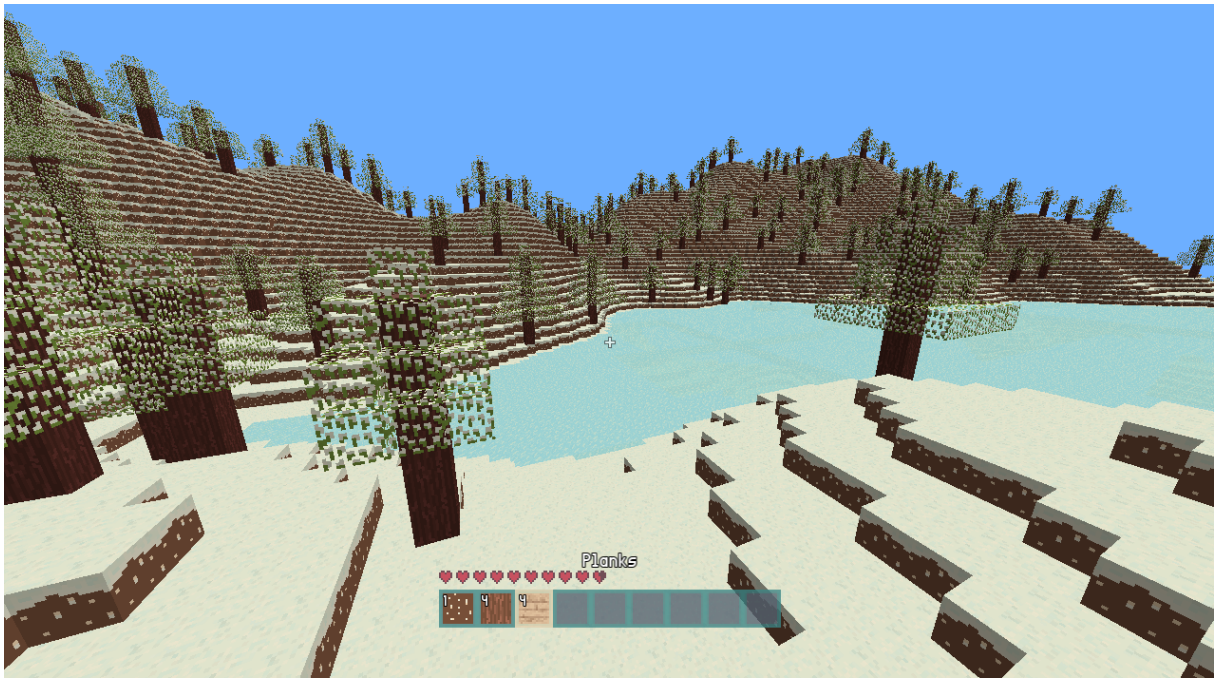

Figure 4.28: A winter biome, it features the darker type of trees whose leaves are covered in snow. The water is frozen and the ground is covered with snow.

Figure 4.29: The desert biome has very flat terrain where cacti grow and there is plenty of sand and almost no water.

Figure 4.30: Small house built on forest biome.

Chapter 5

Internal specification

5.1 General architecture of the game

The main class, that is *Game* is based on two main pillars. One pillar is to manage the main flow of the game and it does this in collaboration with the state stack. The second pillar are the resources it holds and distributes further into the game. Presenting a single class diagram of the entire program would be impossible in this work, due to its complexity and overall size, so this chapter will focus on the most important aspects of the application implementations.

5.1.1 State Stack

The implementation of the state stack is based on the book *SFML Game Development* [15] which was a tremendous help in writing the project. Looking at Figure 5.1, one can see that there are two design patterns going on here. All states inherit from the *State* class, which serves as an interface. Using the **strategy design pattern** here allows *StateStack* class to hold all states in a single container. Each state implements four methods:

- **draw** — a function that draws graphics for the game screen.
- **fixedUpdate** — a function that updates the game a fixed number of times.
- **update** — a function that updates the game every game frame.
- **handleEvent** — a function that accepts player inputs such as a mouse, keyboard, window focus.

All four methods are sufficient to program the appropriate game logic. A second design pattern also appears here, which is the **factory**. It does not appear in its most familiar form, but *StateStack* can also create states when needed.

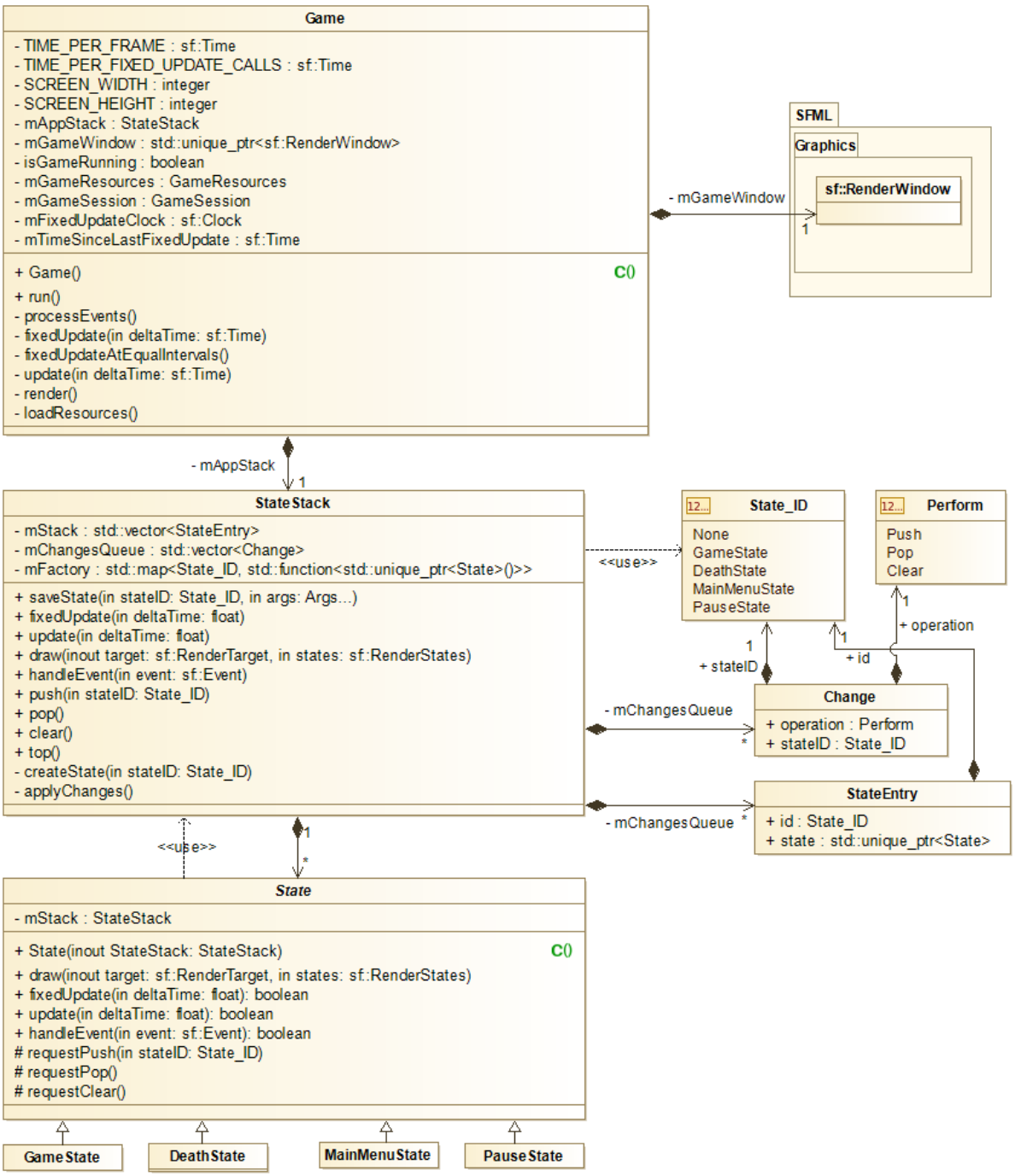

Figure 5.1: Class diagram of *Game* class and *StateStack* class.

When a player starts a game and the *Game* class is created, it starts by passing information to the *StateStack* class on how it should create each state, and then it passes on information about which state should be first on the stack (see Listing 11).

```
Game::Game()
{
    ...

    mAppStack.saveState<GameState>(State_ID::GameState, *mGameWindow,
                                   mGameResources, mGameSession);
    mAppStack.saveState<DeathState>(State_ID::DeathState, *mGameWindow,
                                    mGameResources, mGameSession);
    mAppStack.saveState<PauseState>(State_ID::PauseState, *mGameWindow,
                                    mGameResources);
    mAppStack.saveState<ExitGameState>(State_ID::ExitGameState);
    mAppStack.saveState<MainMenuState>(State_ID::MainMenuState, *mGameWindow,
                                       mGameResources, mGameSession);

    mAppStack.push(State_ID::MainMenuState);
}
```

Listing 11: Code responsible for configuring the states making them available for creation in the future, and pushing the first state onto the stack.

When the *saveState* method is called, no state is created yet. A state identifier to state creation lambda function entry is created in the map container (see Listing 12). The template allows to pass the class of the state inheriting from the *State* class, and the variadic template allows to pass any arguments needed to construct any state whatever arguments it needs.

```
template<typename State, typename... Args>
void StateStack::saveState(State_ID stateID, Args&&... args)
{
    mFactory[stateID] = [&args..., this]()
    {
        return std::unique_ptr<State>(
               std::make_unique<State>(*this, std::forward<Args>(args)...));
    };
}
```

Listing 12: The way „recipes” of how to create states are stored by the *StateStack* class.

When push is called, no state is created yet either. An immediate reaction to push, pop or stack clearing is undesirable, as these methods are usually called directly from the active state on the stack. Popping the state when the state has not yet finished job is not the best idea, which is why these operations are queued and called at the next opportunity (see Fig 5.2)

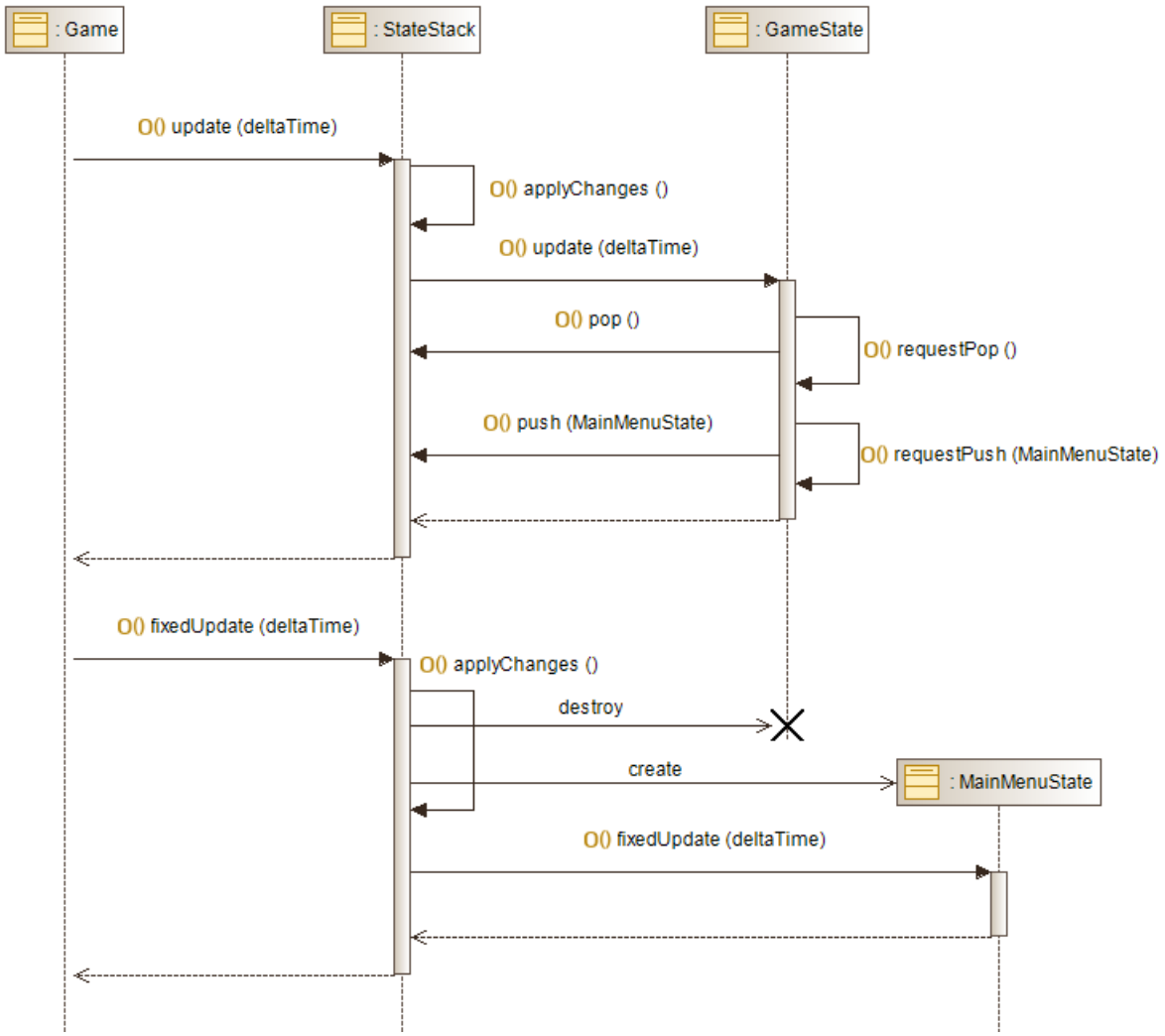

Figure 5.2: Sequence diagram showing how states are popped and pushed by the active states on the stack.

The biggest positive of this solution is that the state currently on the stack does not need to have an idea of what arguments the state that constructs onto the stack needs. All these things have already been prepared using the *saveState* method. If the state on the stack wants to perform any operations on the stack, it only send requests at most by passing only the identifier of the new state.

5.1.2 Game resources

For resources, it is very good practice when using the SFML library to create a resource manager class. This is a class that is actually very often just a wrapper around map of resource identifiers to a resource. Textures, audio, or fonts in SFML are very large objects, so loading them from a file into memory in each class would be an unnecessary waste of resources. Thanks to *ResourceManager* class in this project it is only necessary to load the resource once, and then using the identifier pick it up in any class. *ResourceManager* is a

typical **flyweight** design pattern implementation. In this project, the *ResourceManager* is a template class, so the individual instances of the *FontManager* and *TextureManager* are created from it (see Figure 5.3).

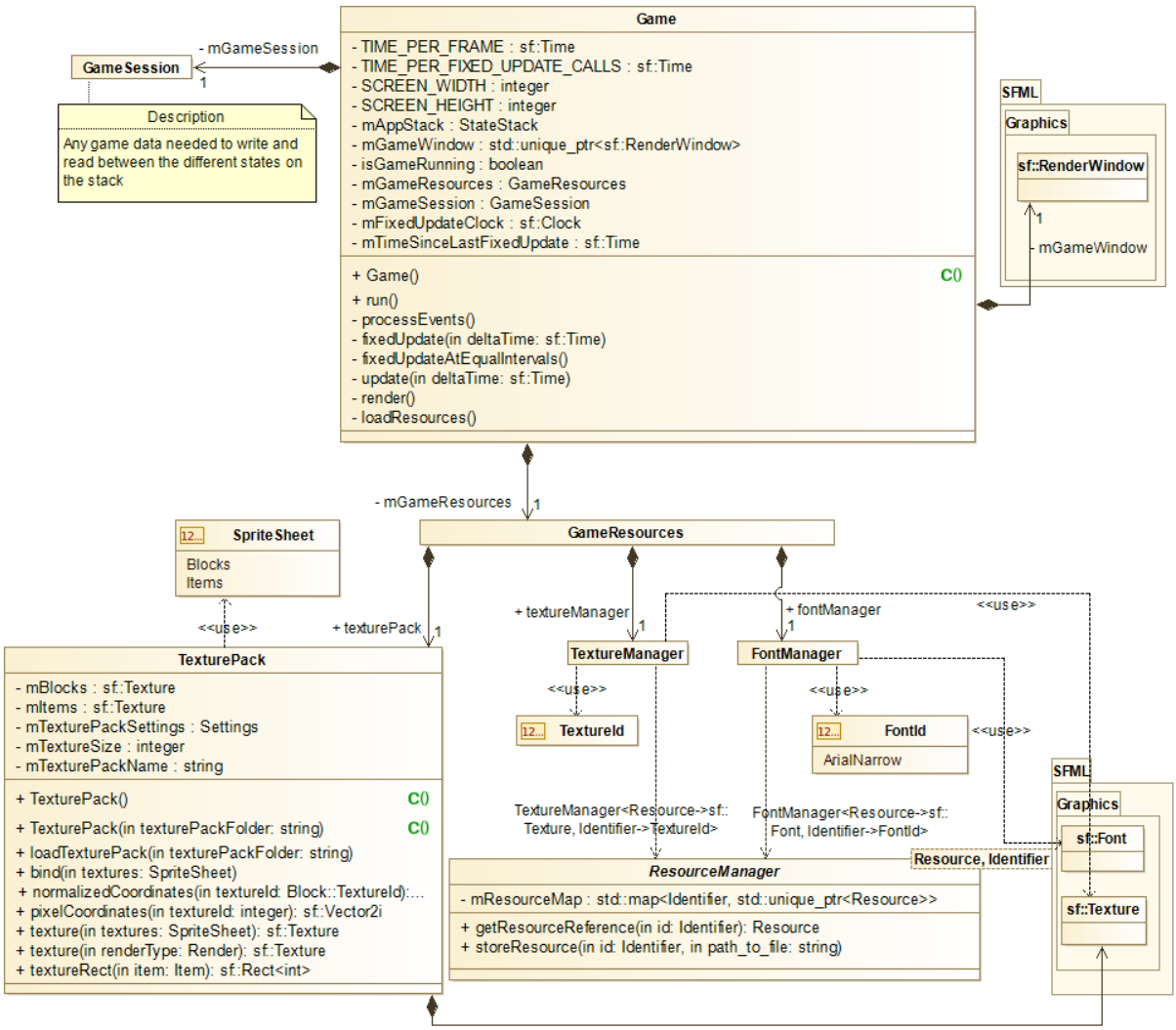

Figure 5.3: Simplified class diagram showing the resources held by the game.

This project usually keeps larger textures in the *ResourceManager*, the exception, however, are so-called texture packs. These are two spritesheets intended for blocks and items in the game. The *TexturePack* class allows those two textures to be changed during the game, and returns the normalised coordinates for each texture ID which makes them easy to use together with OpenGL.

5.2 Terrain generation

Terrain generation proceeds according to previous analysis in Section 2.2.2. Figure 5.4 shows the classes involved in generating terrain.

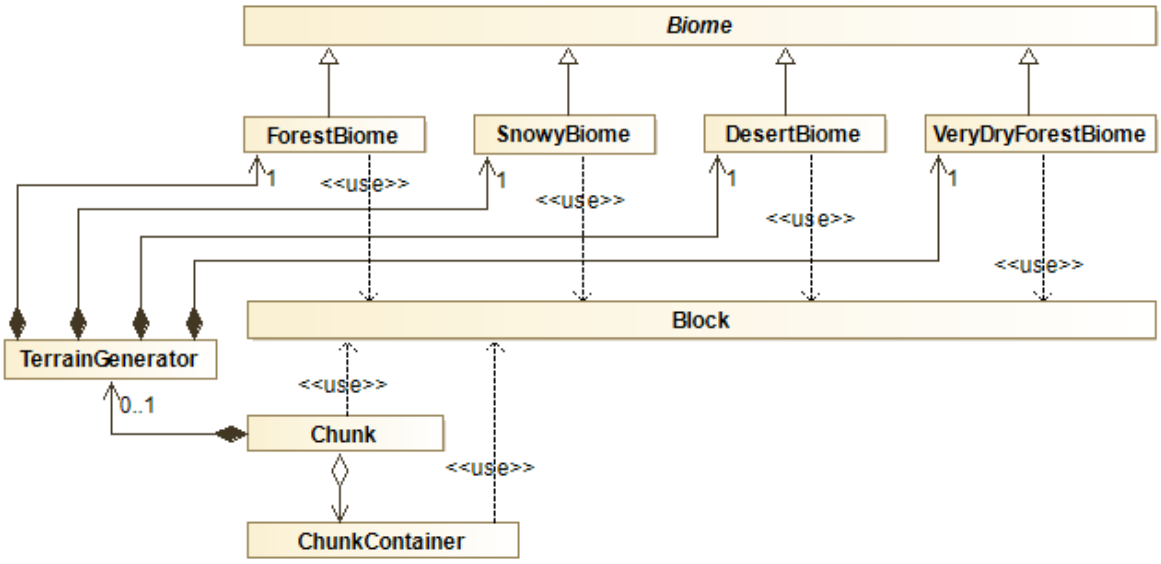

Figure 5.4: Very high-level class diagram of classes involved in terrain generation.

Each biome implements an abstract *Biome* class, the implementation of which looks as shown in Figure 5.5.

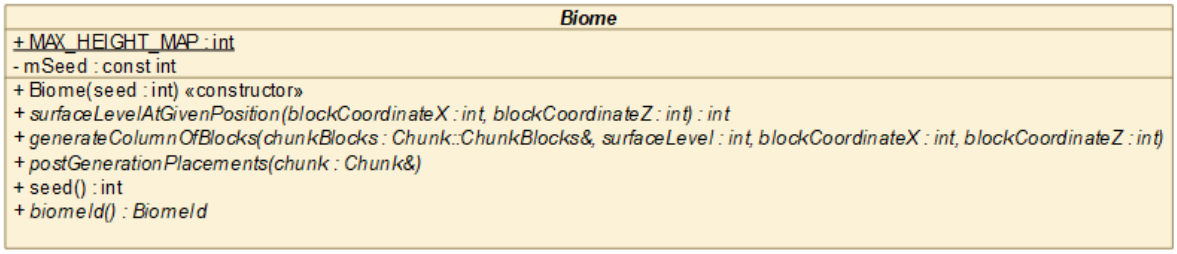

Figure 5.5: Implementation of an abstract *Biome* class.

Each new biome has three methods to implement, which provide suitable customization options to create new unique terrains:

- **surfaceLevelAtGivenPosition** is a function that determines how high the terrain should be at the given X and Z coordinates. Depending on how it is implemented the terrain can be very mountainous, or flat. Low-set, or high-set.
- **generateColumnOfBlocks** is a function that generates a column of blocks on given X and Z coordinates with a given terrain height. For a desert, it could be a column that consists of sand at the top and stone below, while in a forest the first block would be grass, the next would be dirt, and the still lower would be stone.
- **postGenerationPlacements** is a function that carries out the decorative procedure on an existing terrain. This can include putting up trees, flowers, grass, bushes, cacti.

- **biomeId** is a function that allows to distinguish and compare biomes with each other when there is only a reference or pointer to the base class.

The biome is deduced directly inside the *TerrainGenerator* and the **strategy** design pattern is used. The implementation of the biome deduction method can be seen in Listing 13.

```
Biome& TerrainGenerator::deduceBiome(int globalCoordinateX, int globalCoordinateZ)
{
    auto& x = globalCoordinateX;
    auto& z = globalCoordinateZ;

    auto humidity = humidityAtGivenCoordinates({x, 0, z});
    auto temperature = temperatureAtGivenCoordinates({x, 0, z});

    if (humidity <= 20 && temperature >= 20)
    {
        return mDesertBiome;
    }
    else if (humidity <= 30 && temperature >= 15)
    {
        return mVeryDryForestBiome;
    }
    else if (temperature < -5)
    {
        return mSnowyBiome;
    }
    else
    {
        return mForestBiome;
    }
}
```

Listing 13: The method responsible for deducing the biome on the x and z coordinates.

The fact that biomes are used here as a reference to the base class makes it possible for each coordinate inside the chunk to be assigned a corresponding biome, keeping everything in one container. The biomes are very easily compared using the virtual *biomeId()* method.

5.3 Physics

At this stage of implementation, the only collision in the 3D world that can occur is the block-player collision. For the purpose of collisions, the *AABB* class was created, which further simplifies this process and makes it more readable (see Figure 5.6).

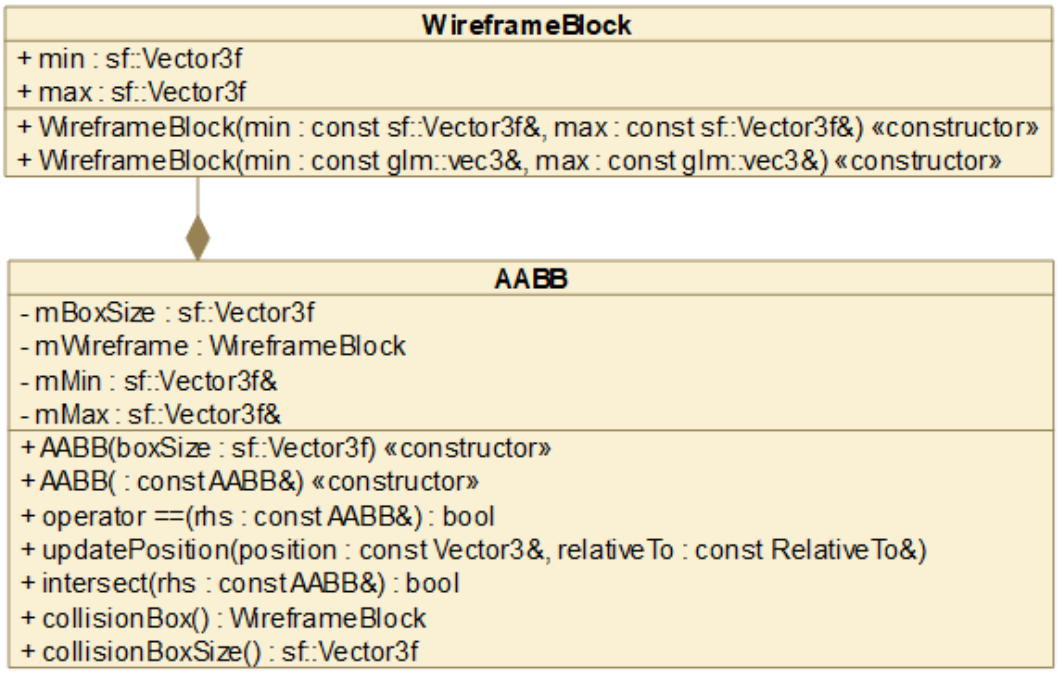

Figure 5.6: *AABB* class implementation.

Inside the *AABB* hides the *WireframeBlock* class, which is a cuboid represented by two points in space (see Figure 5.7).

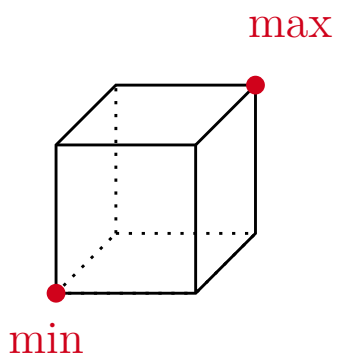

Figure 5.7: Representation of a cuboid in space based on two points.

For every fixed update the player performs a position update and checks for collisions. This works according to the analysis in section 2.1.3. First, the player's collision box is set to player position and then the *ChunkContainer* is asked whether a collision has occurred with any block (see Listing 14). Based on this, the player's new position based on velocity is either retained or discarded and the player stopped.

```
bool Player::tryUpdatePositionByApplyingVelocityIfCollisionAllows(
    const ChunkContainer& chunkContainer, float& position, float& velocity)
{
    position += velocity;
    mAABB.updatePosition(mPosition, AABB::RelativeTo::BottomCenter);
    if (chunkContainer.doesItCollide(collisionBox()))
    {
        position -= velocity;
        velocity = 0;
        return false;
    }
    return true;
}
```

Listing 14: Detecting if a player collides with any block and changing the player's position based on that.

Assuming that each block may have a different collision box then the appropriate behavior is to make the collision check with the blocks with which the collision box is already in contact. To check all the blocks with which there is contact, it is enough to go from the min point to the max point with an increment of block side, every time checking the collision with a block at a given point. This is shown in Listing 15 and Figure 5.8.

```
bool ChunkContainer::doesItCollide(const AABB& aabb) const
{
    auto incrementPoint = [](float& point, float& dimension)
    {
      if (dimension < point)
        dimension = point;
      else
        dimension += Block::BLOCK_SIZE;
    };

    auto [minPoint, maxPoint] = aabb.collisionBox();
    for (auto x = minPoint.x; x <= maxPoint.x; incrementPoint(maxPoint.x, x))
      for (auto y = minPoint.y; y <= maxPoint.y; incrementPoint(maxPoint.y, y))
        for (auto z = minPoint.z; z <= maxPoint.z; incrementPoint(maxPoint.z, z))
        {
          if (isThereCollisionBetweenBlockAtGivenPoint(aabb, sf::Vector3f(x, y, z)))
            return true;
        }
    return false;
}
```

Listing 15: Code that checks for collision with any block that is within the collision box.

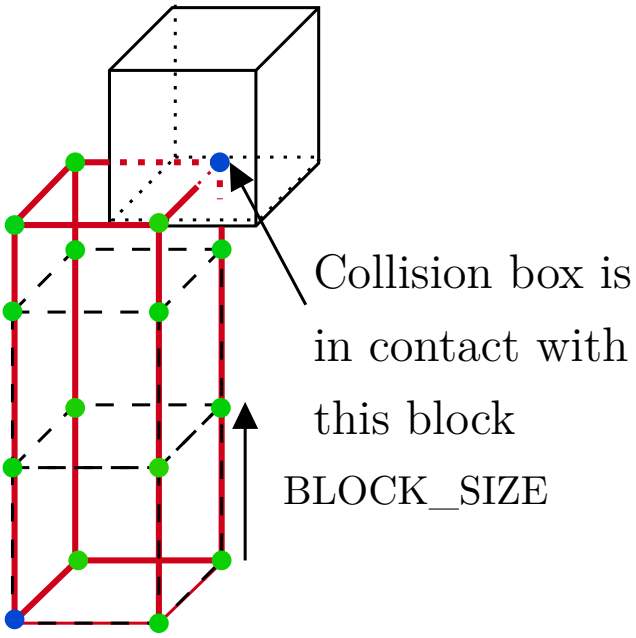

Figure 5.8: Subsequent points checked to find blocks that are within the range of the collision box.

In Figure 5.8, it is worth noting that it is important to preserve the bounds of the intervals. The last top points are no longer extended by a side of the block further and are bounded by the coordinates of the maximum point.

In the current implementation, the aforementioned step is sufficient to detect collisions, it is not even necessary to compare the collision box of the player with the collision box of the block. Determining the blocks within the collision box and then checking if the block is collidable would work in the current version of the game, but would not be correct. In the current version, all blocks fully fill their entire space, but when non-standard-sized blocks such as half-blocks or fences are added to the game, the above mechanism is crucial in detecting collisions correctly around the player.

5.4 Chunks

Chunk-related classes are some of the largest classes in the entire application. In Figure 5.9 a simplified class diagram can be seen. All three classes have their own separate uses.

- **ChunkManager** removes chunks that are far from the player, adds new chunks around the player, and queues the rebuilding of chunks.
- **ChunkContainer** stores chunks and allows easy access to blocks, or chunks with given world coordinates.
- **Chunk** stores blocks, allows to edit them, destroy them, and place new ones.

Unfortunately, all three classes are tightly coupled. This is due to the fact that:

- *Chunk* is asking *ChunkManager* to queue its rebuild.
- *Chunk* is asking *ChunkContainer* to find out about blocks in neighboring chunks. This happens, for example, when building a mesh.

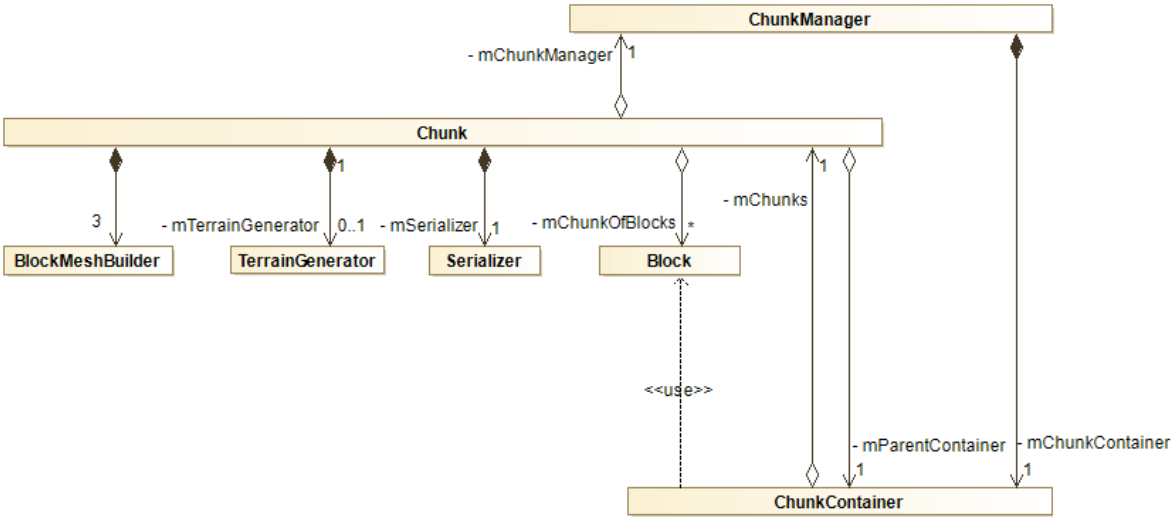

Figure 5.9: Simplified class diagram showing Chunks.

In addition, *TerrainGenerator* allows to get terrain information, *BlockMeshBuilder* builds mesh, and *Serializer* saves chunk to file.

5.4.1 Problems of post-generation

When the *Chunk* is finished preparing the terrain and has generated all the blocks in the *Chunk* there comes a moment when *Biome* may want to decorate the *Chunk* it was working on and this turns out to be quite a problematic moment. Let's follow the process based on *ForestBiome* class. When *ForestBiome* generates terrain and generates a block one block higher than the ground level, then there is a small chance that a tree will be created at that place (see Listing 16).

```
1 else if (y == surfaceLevel + 1 && mTreeDecider(gen))
2 {
3     chunkBlocks[x][y][z] = std::make_unique<Block>(BlockId::Log);
4     mTreesToPlace.emplace_back(x, y, z);
5 }
```

Listing 16: Part of *ForestBiome* code that determines what is a block above terrain level.

As can be seen in Listing 16, such a block coordinate is stored in the container that is used to place a tree later in the post-generation phase on that given coordinate. When these are single blocks it is not problematic, it becomes problematic when a larger element such as a tree with a crown several blocks large is on the edge of the chunk, and the crown goes beyond the chunk. If in this case the algorithm simply would not place such blocks then the map will produce many trees similar to the one that can be seen in Figure 5.10.

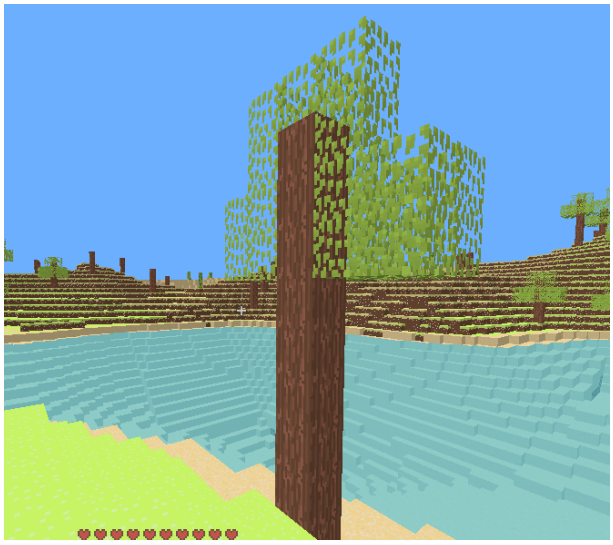

Figure 5.10: Half of the tree was cut down, as the other half was outside the boundaries of the chunk at the time of its creation.

The fact that the *Chunk* class can communicate with the *ChunkContainer* class and ask it to place blocks in a neighboring chunk also does not solve the problem, because at this stage of chunk creation very often the chunk does not yet have a neighbor. For this reason, *ChunkContainer* allows placing blocks in places that don't yet exist by storing them in a queue and in specific moment checking to see if perhaps they belong to one of the newly created chunks. Such information is stored in the structure shown in Figure 5.11.

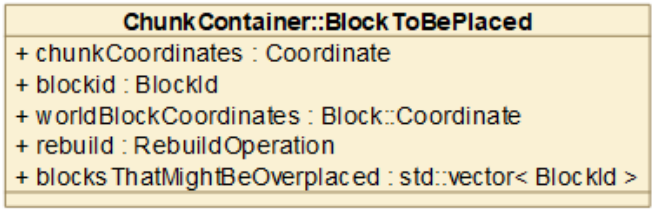

Figure 5.11: Implementation of a structure storing blocks to be placed.

It is worth explaining here two important variables that may not be understood:

- **rebuild** determines whether the chunk should be rebuilt after this operation of block placement. For example, when placing multiple blocks at the same time, it may be undesirable to rebuild the chunk each time.

- **blocksThatMightBeOverplaced** are blocks that can be overwritten. If the argument responsible for all blocks is passed here, then when the chunk is generated no matter what is in the place this block will overwrite it. In other cases, for example, it may be desired that the block will overwrite only the air block, and not necessarily „push" the leaves in the rocky mountain.

The block queuing mechanism can be seen in Figure 5.12, which in high-level attempts to show a scenario in which a biome place blocks on a chunk.

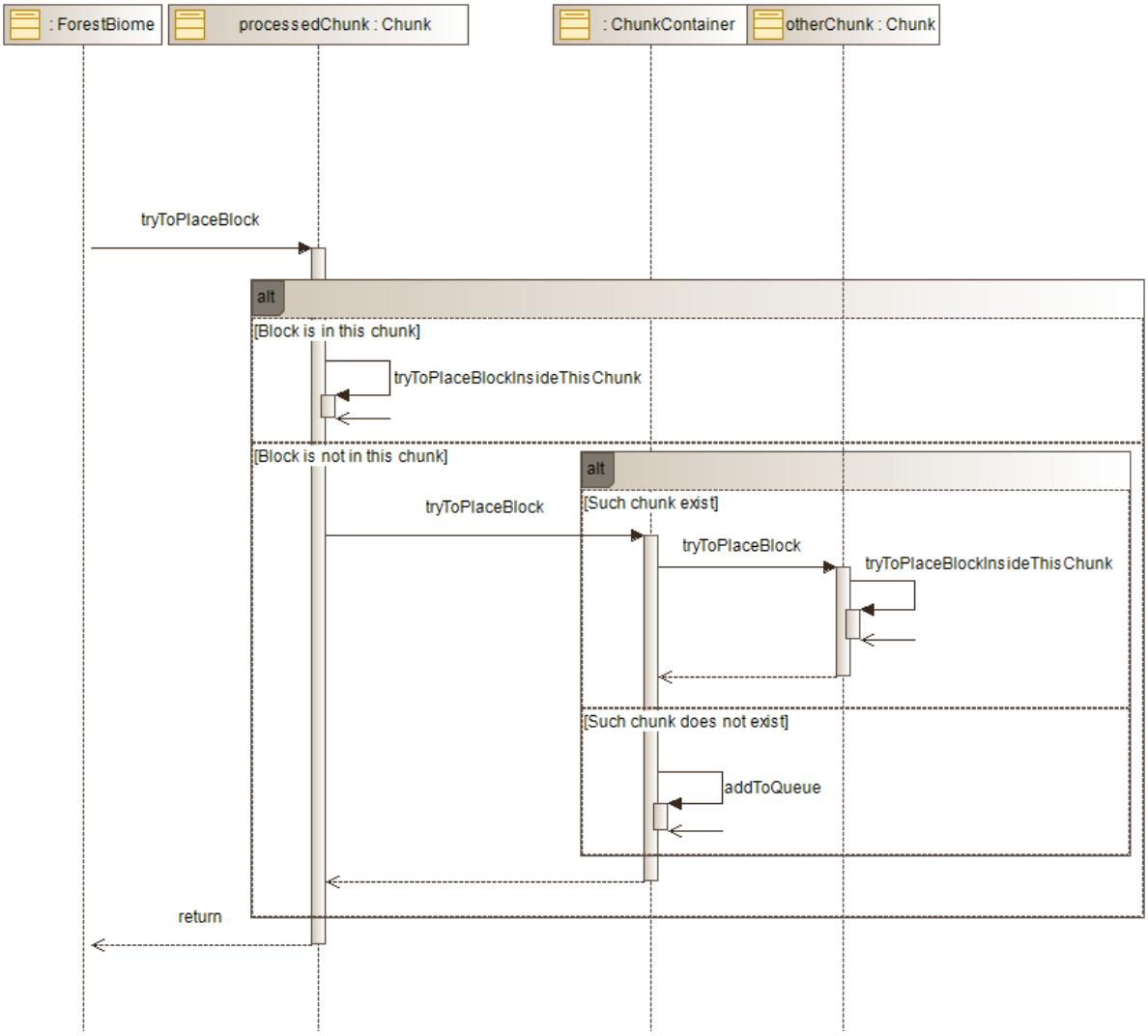

Figure 5.12: Mechanism of placing blocks depending on whether it is outside the chunk and whether the chunk exists.

The block queue can then be checked every update, or better yet, after every chunk insert, or emplace operation called on the container. If any of the waiting blocks have lived to see the chunk on which it is supposed to be placed, it is placed and then removed from the queue.

5.5 Player

There is a lot going on in the *Player* class, as it is the connecting point of many mechanisms. The class itself also contains a lot of its own internal logic responsible for in-game controls. A simplified class diagram focusing on the most important relationships related to the *Player* class can be seen in Figure 5.13, and the public implementation interface of the *Player* in Figure 5.14.

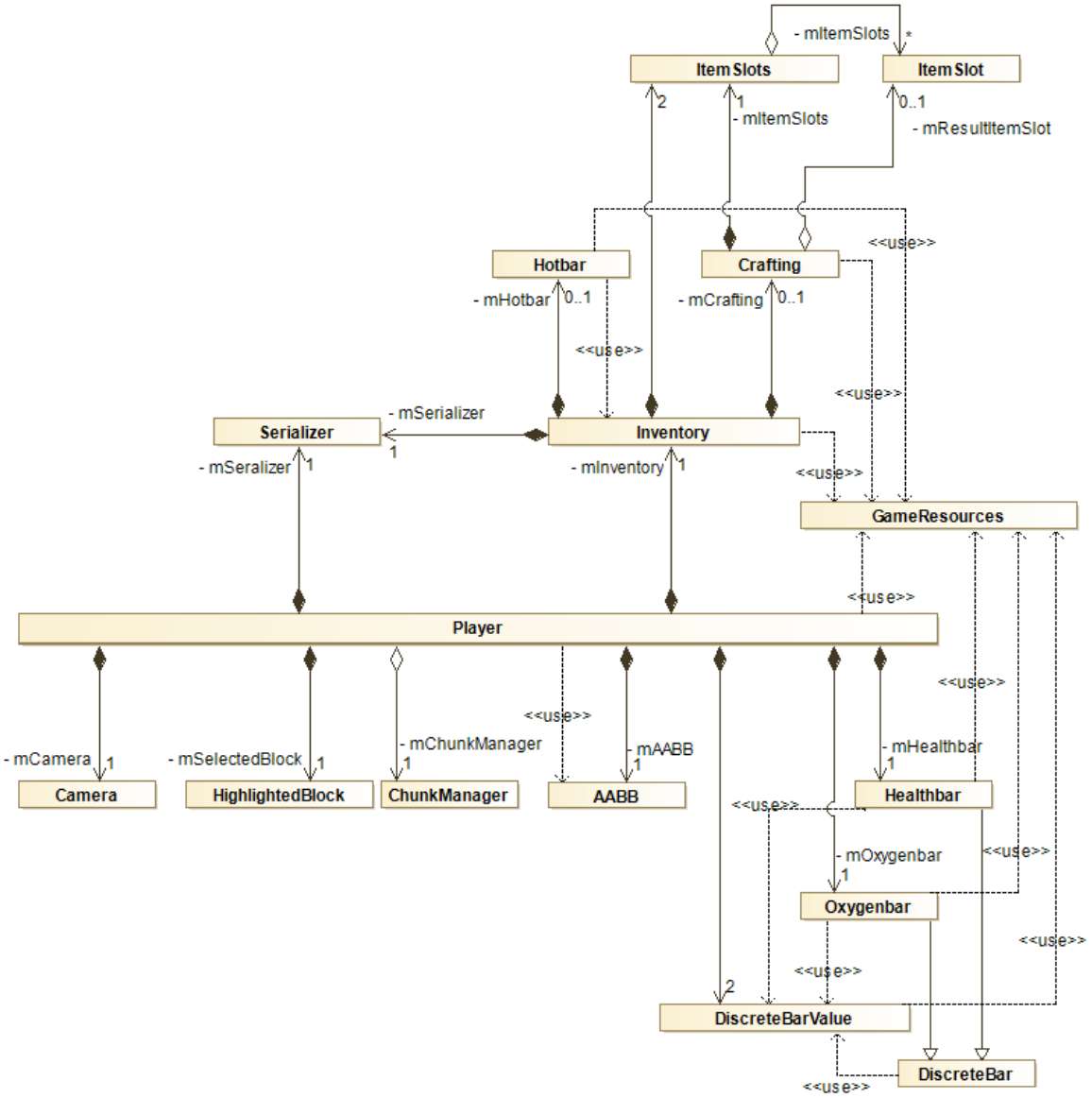

Figure 5.13: Simplified high-level class diagram focusing on the *Player* class.

Figure 5.14: Public interface implementation of the *Player* class.

Internally, the *Player* class is constantly updating such information as the player's life and oxygen, for example. It calculates and stores the player's velocity and position. It is the *Player* class that measures how long the player has been underwater and checks if the player's head is underwater using an additional auxiliary collision box located at camera eye level that can be seen in Figure 5.15.

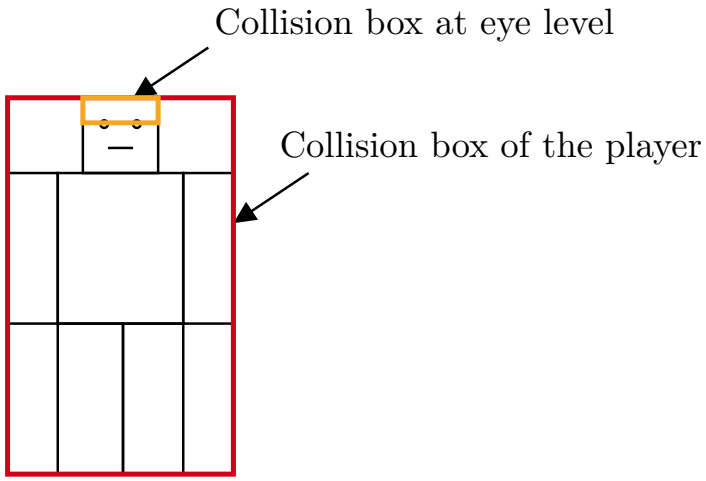

Figure 5.15: Collision boxes of the player.

Chapter 6

Verification and validation

6.1 Testing

Testing is a very important part of software production. Lack of proper testing and validation can lead to loss of customers, image damage, or, in the worst cases, even death. Imagine a situation in which a large telecommunications services company buys a radio unit from another large radio manufacturing company. There are a lot of connected devices in one radio cell due to a large event. Suddenly there is a huge disaster and in result a big panic. Lots of people are injured and need help, but the radio that serves their phones has not been properly tested and validated to handle such a huge throughput and unexpectedly crashes and shuts down. People are unable to call for help, the event echoes in the press. A huge telecommunications services company loses the trust of its customers and loses its image as a decent company for many years. It's even worse for the company producing this particular radio unit, as it will lose business partners and those affected by the event may file numerous lawsuits against it. And worst of all, people can die in all of this, and the main result is poor validation and testing of the software on their radio. In an age of autonomous cars, a plethora of life-support devices, can we afford such recklessness?

6.1.1 What does testing provide?

Testing is a way of making sure that the product being delivered is of good quality and that it is consistent with the assumptions and requirements. Without proper testing and validation, there can be no talk of reliability and quality, because it is testing and validation that proves this quality.

Testing creates self-documenting code. Tests are the best documentation because it is documentation that must be up-to-date and not outdated. Well-written tests show what works and how it works.

When testing goes hand in hand with clean, quality code it significantly accelerates

development work. Tests allow to find bugs and, most importantly, are like shields that don't allow sneaking new bugs into existing code. In the words of *Robert C. Martin*, an American programmer and author of many books on software engineering:

> The only way to make the deadline — the only way to go fast — is to keep the code as clean as possible at all times. [24].

Testing is an extremely important aspect of working with legacy code that programmer do not know well. If such is untested the first rule of working with such code is to write tests for it.

6.1.2 Where is testing critical?

The best answer to this question is „everywhere". Nowadays, the software is part of our daily lives. It is software that makes us, hundreds of miles away from each other, still, talk to each other. It's software that washes our laundry and manages our bank accounts.

However, it must be admitted that it is possible to distinguish areas that need it more. Such examples include, for example, autonomous cars, life support devices, radio units, and critical systems that manage huge amounts of cash. In the case of computer games, one of the worst things that can happen is financial loss and disappointed players. Poor code and lack of testing in the video game industry can lead to poor performance, lost saves, and game crashes. We can't compare it to a situation in which someone loses their life. In which a car suddenly starts to accelerate as a result of a software error, or in which the radio unit does not allow to call an emergency number.

6.1.3 Manual Tests

Manual testing in the case of the computer game development process is a very common solution. Game development companies employ huge numbers of testers–players who look for bugs in the games the company develops. Such bugs are then reported with detailed information on the reproduction of the bug.

In this project, it was very common that every feature has been manually checked with every change whether everything works correctly and whether it can be broken in any way. This is the kind of test that is not automated by any means but can be quite effective especially when the game is still small. Not all errors can be caught by automated methods, especially in computer games.

6.1.4 Unit Tests

Unit tests are the most popular method of automation testing. They are distinguished by their simplicity and speed of execution. They are fast enough so entire software development techniques like *Test-driven development* can be based on them in which the

developer first writes tests that do not pass, then implements the functionality making the tests pass, and at the very end refactors the code.

They easily allow refactoring of even another programmer's code without fear of introducing an error into the code. This is because the idea is that unit tests should only test the public interface. Because of this, the entire internal implementation can be changed and the tests should correctly indicate whether public interface works correct or not.

An example of very simple units tests can be seen in Listing 17. They test using only the public interface whether states appear and disappear from the state stack (see Section 5.1.1). Messing up any of these functionalities will momentarily throw an error.

```
TEST(StateStackTest, StateShouldNotBePushedBeforeUpdate)
{
    StateStack s;
    EXPECT_TRUE(s.empty());
    s.saveState<State>(State_ID::GameState);
    EXPECT_TRUE(s.empty());
    s.push(State_ID::GameState);
    EXPECT_TRUE(s.empty());
}

TEST(StateStackTest, StateShouldBePushedAfterUpdate)
{
    StateStack s;
    s.saveState<State>(State_ID::GameState);
    s.push(State_ID::GameState);
    s.update(0);
    EXPECT_FALSE(s.empty());
    EXPECT_EQ(s.top(), State_ID::GameState);
}

TEST(StateStackTest, CorrectStateIdShouldBeOnTopAfterPushTwice)
{
    StateStack s;
    s.saveState<State>(State_ID::GameState);
    s.saveState<State>(State_ID::PauseState);
    s.push(State_ID::GameState);
    s.push(State_ID::PauseState);
    s.update(0);
    EXPECT_FALSE(s.empty());
    EXPECT_EQ(s.top(), State_ID::PauseState);
}
```

Listing 17: Sample unit test code that tests the correct behavior of *StateStack*.

Now when someone accidentally makes the unexpected change seen in Listing 18, the tests

will alert right away by telling what is working incorrectly (see Figure 6.1 and Figure 6.2).

```
void StateStack::push(State_ID stateID)
{
    mChangesQueue.emplace_back(Change{Perform::Push, stateID});
    applyChanges(); // <------ OH NO! THAT IS A BAD LINE!
}
```

Listing 18: An example of incorrectly modified *StateStack* code.

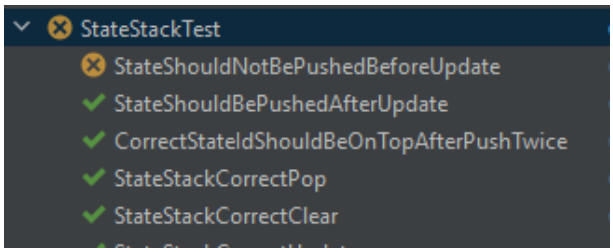

Figure 6.1: A test that does not pass after the change in Listing 18.

```
N:\...\tests\ut\src\StateStackTest.cpp(29): error: Value of: s.empty()
  Actual: false
  Expected: true
```

Figure 6.2: The message that the failed test from Figure 6.1 returns.

However, to make testing fast and simple it is necessary to actually keep things small here. When there is test testing the behavior of the state stack, there is not much point in using the real game state, which will actually create the whole game in the background and really update it for that time. In all of this it is just a matter of checking the state stack as a unit, no other dependencies are needed. For this purpose, it is possible to use a *Fake* object, which will be a simplified implementation of the state. It will be an object that will not involve huge dependencies. Or it is possible to use a *Stub* object, which will hold already prepared data and simply just respond with it. Both are great, but there is another type of object that will further allow to better test the code — *Mock*.

Mocks

Mock is an extremely useful object in testing, as it registers calls and even allows to simulate certain behaviors. For example, a very important behavior of the *StateStack* class is that every update should also call update method of the state on top of the stack. Mock makes this very thing possible to verify. For this purpose, the *MockState* class shown in Listing 19 was prepared.

```
class MockState : public State
{
public:
    using State::State;

    MOCK_METHOD(void, draw, (sf::RenderTarget & target, sf::RenderStates states),
                (const, override));
    MOCK_METHOD(bool, fixedUpdate, (const float&), (override));
    MOCK_METHOD(bool, update, (const float&), (override));
    MOCK_METHOD(bool, handleEvent, (const sf::Event&), (override));
};
```

Listing 19: A mock of the *State* class.

Unfortunately, the factory that is the *StateStack* class is very reluctant to give a peek into its internals and the public interface very efficiently defends access to the internal states. For this reason, the tested class required a small extension of the behavior with a helper method for the test. So an additional method was created in the test file to forcefully implant the mock and access it at the same time that can be seen in Listing 20.

```
class UglyTestableStateStack : public StateStack
{
public:
    template<typename State, typename... Args>
    State& forcePushStateAndReturn(State_ID stateID, Args&&... args)
    {
        auto state = std::make_unique<State>(*this, std::forward<Args>(args)...);
        auto& stateRef = *state;
        mStack.push_back({stateID, std::move(state)});
        return stateRef;
    }
};
```

Listing 20: *StateStack* extended with additional functionality for testing to allow implantation and retrieval of the mock.

Now, thanks to the mock, It can be check whether calling an update on a stack of states will also make an update to be called on a state on top of the stack. A test to verify this behavior can be seen in Listing 21. And if ever this functionality is broken, this test will let know about it.

```
TEST(StateStackTest, StateStackCorrectUpdate)
{
    UglyTestableStateStack s;
    auto& mock = s.forcePushStateAndReturn<MockState>(State_ID::GameState);
    EXPECT_CALL(mock, update(15));
    s.update(15);
}
```

Listing 21: Test to see if an update called on a stack calls out an update on a state on top of it.

More importantly, more complex behavior can also be tested. For example, the state stack allows states to block, or not block, the execution of the state below. For the update method, this happens based on whether the method returns true or false. Thus, by asking the mock to return the value needed when the method is called, it is possible to get any scenario that is needed. Such a test can be seen in Listing 22.

```
TEST(StateStackTest, StateStackUpdateBlockingStateBelow)
{
    UglyTestableStateStack s;
    auto& mock1 = s.forcePushStateAndReturn<MockState>(State_ID::GameState);
    auto& mock2 = s.forcePushStateAndReturn<MockState>(State_ID::GameState);
    EXPECT_CALL(mock2, update(15)).WillOnce(testing::Return(false));
    EXPECT_CALL(mock1, update(15)).Times(0);
    s.update(15);
}

TEST(StateStackTest, StateStackUpdateNotBlockingStateBelow)
{
    UglyTestableStateStack s;
    auto& mock1 = s.forcePushStateAndReturn<MockState>(State_ID::GameState);
    auto& mock2 = s.forcePushStateAndReturn<MockState>(State_ID::GameState);
    EXPECT_CALL(mock2, update(15)).WillOnce(testing::Return(true));
    EXPECT_CALL(mock1, update(15));
    s.update(15);
}
```

Listing 22: Tests to verify functionality in which more than two states on the stack can block or not to block further execution of the states below them.

6.1.5 Detected and fixed bugs

A great amount of bugs were spotted during the production of the game. One of such unexpected bugs was the problem described in the Section 5.4.1. Among other smaller

ones, it is possible to mention:

- The player was able to place the block directly where they stands blocking themselves.
- The player was able to place a block where a block already stands, effectively overwriting it.
- The player's life was not based on discrete values such as 0.5, 1.0, or 1.5, but was determined by a floating number, and also the damage player could receive was a floating number too. This resulted in a slow death as the player descended from small heights and received little damage over time.
- Seed of the world did not save correctly, causing the player to see mismatched terrain on the borders of chunks after reloading in new locations.
- All collision boxes of AABB collisions were moved by some offset creating a mismatched collision in the blocks.
- When an item resulting from crafting was taken from the crafting-zone, all items in the crafting-zone were removed.
- The transition between biomes was steep and often with ugly layered the blocks, for example, causing the forest mountain next to the desert to have sand all along its transition from the top of the mountain to the desert.

6.2 Clean Code

Since this is a large project from the beginning, it tries to avoid making a mess of the code for fear of the cost associated with having a mess. When the mess in the code is uncontrolled and allowed to build up then the performance asymptotically starts to approach zero [24]. The cost of change becomes huge, one change begins to destroy other parts of the code. At some point development becomes impossible. To make matters worse, the working time cannot be estimated correctly, so the completion of the project faces a big question mark and it is impossible to predict what the future will bring. Code that is clean tries to carry a fixed cost of change, making it much easier to estimate the working time on different parts of the application.

- The code for this project has a huge number of comments in header files written in the style of doxygen comments. Each method has its own comments.
- Functions try to be small, do one thing, and have descriptive names.

- The code has one consistent style defined by the clang-format file, and checked by the clang-format tool in both the development environment and CI (see Section 6.3).
- The design patterns are in use.
- The polymorphism is prefered over if/else or switch/case.
- The KISS (keep it simple stupid) rule is followed.
- Each class name tries to define what to do and do only one thing.
- The code avoids duplication.

6.3 Continuous integration

Continuous Integration (CI) workflows (Github Actions) in Github have proven to be an extremely useful tool. In this project, it is set up in such a way that with each commit, three important things are checked:

- whether the project is building correctly,
- whether all tests pass,
- whether clang-formatting throughout the project is correct.

The tools on which one works can fail. A developer can fail to check tests before pushing out a change, or some change is not pushed out in full and the project doesn't build. This continuous integration setup allows to make sure that everything that is committed builds, passes tests and formatting is correct. This avoids the situation in which a developer uploads a lot of changes, and then the project on another computer does not work, because for example, the library used was present on the previous computer locally, but could no longer download correctly in the current project. In Figure 6.3, it can be seen two checks that are executing. The next Figure 6.4 shows the build and run tests in the „CMake Windows Tests / build" check.

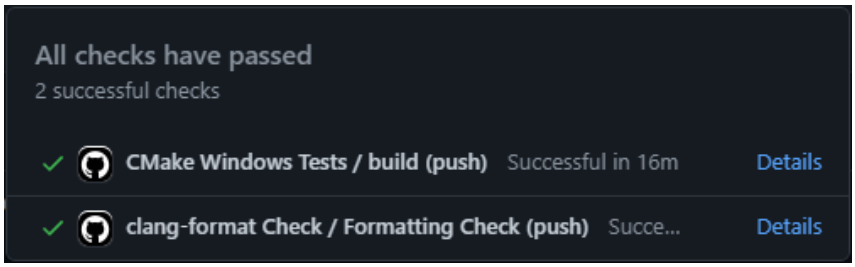

Figure 6.3: Checks executed on the pushed commit.

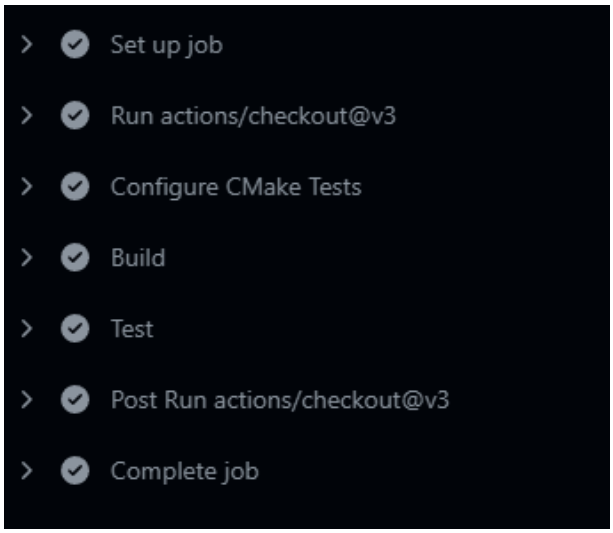

Figure 6.4: Job building the project and checking that tests pass.

Chapter 7

Conclusions

The main goal of this project was to implement a voxel-based game containing procedurally generated terrain, and the essential world interactions of a computer game inspired by the game Minecraft. These assumptions and functional requirements have been met as proven by the External Specification (see Chapter 4) and the provided recordings (see Chapter 7).

The game still has huge potential for further development. The original Minecraft is a game that is 13 years old since its first public release and is still being developed. It can be said that this project is a bottomless pit, and it will never run out of possible features to add to it. Nevertheless, in the near future, the most needed and likely improvements would be:

- Mobs such as animals or monsters.
- A cycle of day and night.
- Colorful lighting.
- Ambient occlusion.
- Rewrite the game logic in such a way as to use Entity-Component-System.
- Chests to store items.
- Furnaces for melting materials and roasting meat.
- Blocks of non-standard shapes.
- Caves made using perlin-worms.
- Collection of underground materials.
- Animations of block destruction.

- Replace the current chunk meshing algorithm (Culling method) with a more efficient method like Greedy Meshing.
- Increasing the distance of chunk rendering by adding LOD (level of detail).
- Rewriting the game to Vulkan API.

Working with OpenGL, which is a machine is a hard challenge, especially hard when operating on multiple threads. Multithreading in itself is a very heavy task and its use should still be improved in this project in the future.

Voxel games of this type have a very uneven degree of difficulty. They require a lot of knowledge of programming, optimization, algorithms, and computer graphics. For example, the process of building a mesh using the Culling method is definitely more straightforward than doing it by Greedy Meshing — but at the same time, it is definitely far from being an optimal solution.

This is a large project that grows to a huge size and requires a tremendous amount of software engineering knowledge to run it well. For this reason, this is a project worth continuing in order to gain knowledge and experience. But most of all, it's great fun to create a computer game from scratch.

Bibliography

[1] URL: https://www.quora.com/How-many-seeds-exist-for-Java-Minecraft (visited on 19/12/2022).

[2] *3D collision detection - game development: MDN*. URL: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection (visited on 18/12/2022).

[3] Auburn. *Release Fixed Linux MacOS GUI binaries · Auburn/Fastnoiselite*. URL: https://github.com/Auburn/FastNoiseLite/releases/tag/v1.0.3 (visited on 19/12/2022).

[4] *C++20*. URL: https://en.cppreference.com/w/cpp/20 (visited on 03/01/2023).

[5] *Chunk*. URL: https://minecraft.fandom.com/wiki/Chunk (visited on 19/12/2022).

[6] *CMake*. URL: https://cmake.org/ (visited on 03/01/2023).

[7] G-Truc Creation. *GLM*. URL: https://github.com/g-truc/glm (visited on 03/01/2023).

[8] Mark DeLoura. *Choosing your game engine (2009)*. URL: https://www.slideshare.net/markdeloura/choosing-a-game-engine (visited on 08/01/2023).

[9] eliasdaler" Elias Daler. *ImGui-SFML*. URL: https://github.com/eliasdaler/imgui-sfml (visited on 03/01/2023).

[10] eyalz800. *Serializer*. URL: https://github.com/eyalz800/serializer (visited on 03/01/2023).

[11] *From pixel to voxel. A dimensional leap.* URL: https://www.ledpulse.com/blog-posts/from-pixel-to-voxel-a-dimensional-leap (visited on 26/12/2022).

[12] *Github*. URL: https://github.com/ (visited on 03/01/2023).

[13] Google. *GoogleTest*. URL: https://github.com/google/googletest (visited on 03/01/2023).

[14] Jason Gregory. *Game Engine Architecture*. Taylor and Francis, CRC Press, 2018. ISBN: 978-1138035454.

[15] Jan Haller, Artur Moreira and Henrik Vogelius Hansson. 'Diverting the Game Flow – State Stack'. In: *SFML game development: Learn how to use SFML 2.0 to develop your own feature-packed game*. Packt Publishing, 2013, 130–153. ISBN: 978-1849696845.

[16] Jan Haller, Artur Moreira and Henrik Vogelius Hansson. 'Making a Game Tick'. In: *SFML game development: Learn how to use SFML 2.0 to develop your own feature-packed game*. Packt Publishing, 2013, 24–45. ISBN: 978-1849696845.

[17] Jan Haller, Artur Moreira and Henrik Vogelius Hansson. *SFML game development: Learn how to use SFML 2.0 to develop your own feature-packed game*. Packt Publishing, 2013. ISBN: 978-1849696845.

[18] Junio Hamano and Linus Torvalds. *https://git-scm.com/*. URL: `http://gts.sourceforge.net/` (visited on 03/01/2023).

[19] henrikkniberg. *Minecraft terrain generation in a Nutshell*. 2022. URL: `https://www.youtube.com/watch?v=CSa5O6knuwI` (visited on 19/12/2022).

[20] Matej 'Retro' Jan. *Pixels and voxels, the long answer*. 2017. URL: `https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190` (visited on 25/12/2022).

[21] Jetbrains. *CLion*. URL: `https://www.jetbrains.com/clion/` (visited on 03/01/2023).

[22] Auburn" Jordan Peck. *FastNoise Lite*. URL: `https://github.com/Auburn/FastNoiseLite` (visited on 03/01/2023).

[23] Chris Lattner and LLVM Developer Group. *Clang and Clang-Format*. URL: `https://clang.llvm.org/docs/ClangFormat.html` (visited on 03/01/2023).

[24] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[25] Microsoft. *MSVC*. URL: `https://visualstudio.microsoft.com/pl/vs/` (visited on 03/01/2023).

[26] nigels-com" Nigel Stewart. *glew*. URL: `https://github.com/nigels-com/glew/` (visited on 03/01/2023).

[27] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN: 978-0990582908.

[28] ocornut. *Dear ImGui*. URL: `https://github.com/ocornut/imgui` (visited on 03/01/2023).

[29] Robert Painsi. *Commit Message Guidelines*. URL: `https://gist.github.com/robertpainsi/b632364184e70900af4ab688decf6f53` (visited on 03/01/2023).

[30] *Ray tracing acceleration methods.* URL: https://watkins.cs.queensu.ca/~jstewart/454/notes/33/33a-axis-aligned-bounding-boxes.html (visited on 09/01/2023).

[31] *Szum Perlina (artykuł) | Szum.* URL: https://pl.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise (visited on 19/12/2022).

[32] SFML Team. *SFML — Simple and Fast Multimedia Library.* URL: https://github.com/SFML/SFML (visited on 03/01/2023).

[33] texus. *TGUI - Texus' Graphical User Interface.* URL: https://github.com/texus/TGUI (visited on 03/01/2023).

[34] *The history of Minecraft.* URL: https://www.thescienceacademystemmagnet.org/2019/12/20/the-history-of-minecraft/ (visited on 08/01/2023).

[35] *Understanding Perlin Noise.* URL: https://adrianb.io/2014/08/09/perlinnoise.html (visited on 27/12/2022).

[36] Richard Wakeling. *Red Dead Redemption 2's PC framerate causes unusual issues.* 2019. URL: https://www.gamespot.com/articles/red-dead-redemption-2s-pc-framerate-causes-unusual/1100-6471484/ (visited on 18/12/2022).

[37] *What is a pixel? - definition from Techopedia.* URL: https://www.techopedia.com/definition/24012/pixel (visited on 27/12/2022).

[38] *What is stack in data structure? - javatpoint.* URL: https://www.javatpoint.com/data-structure-stack (visited on 26/12/2022).

Appendices

Index of abbreviations and symbols

Crafting-Zone	A 3x3 grid used to create a new item using the correct arrangement of other items in the grid (see Section 4.3.9).
API	Application Programming Interface.
Voxel	The basic element representing volume in 3D.
FPS	Frames Per Second.
LIFO	Last In First Out.
FIFO	First In First Out.
Seed	The number used to initialize the pseudo-random number generator.
Hotbar	Quick access bar.
AABB	Axis-Aligned Bounding Box.
CI	Continuous Integration.

List of additional files in electronic submission

Additional files uploaded to the system include:

- source code of the application,
- a video file showing how the game works and its key mechanics,
- a video file showing a 5-minute flight over the game world map.

List of Figures

List of Listings

List of Tables