Contents

1. Introduction

This report presents the end project for the course Game Programming *(KGGAPRO1KU)* at IT University of Copenhagen. For this course, we, Mads Engberg, Frans Peter Larsen and Sven Santema, have created and optimised a voxel game. The project is done in *C++* and makes use of the Simple Render Engine (SRE) [1] by Morten Nobel. The focus of this project is on resource management, memory leaks, physics libraries.

The Simple Render Engine project presents a good starting point for a game in *C++*. It has 3D capabilities using Simple DirectMedia Layer (SDL) [2], Box2D [3] or Bullet Physics [4], OpenGL Mathematics (GLM) [5]. Our project builds upon SRE and creates a basic voxel-based game much like *Minecraft* (Mojang, 2009). In *Minecraft* players move around in a world completely made out of blocks. Players can destroy and place blocks, thereby, making buildings and shaping the world to the players liking. Our project aims to create and implement our own version of a voxel-game very similar to Minecraft. In short, a game in which the player moves around in a world that exists out of blocks, and where players can place and destroy blocks.

We have chosen for a voxel-based game since we think it presents many interesting challenges. First of all, the world exists out of many blocks that all have to be rendered. Secondly, what blocks are being displayed change during run-time since players can add and remove blocks as they like. These two challenges require a flexible and well-optimised game system, since meshes cannot be pre-generated or loaded from the disk, but have to be generated and modified during run-time. Lastly, the game is 3D which generally adds to the complexity of the implementation.

The report will first go into the design, in which we will establish the requirements and ideas for the game. In the second chapter, the overall architecture of the implementation, in-depth information regarding class structures and implementations together with work division and tools used for the project. After this, a chapter is dedicated to the performance and bottlenecks we encountered during development. This chapter also goes into depth on how these bottlenecks were solved. Chapter five presents a discussion on the end result. Lastly, in chapter six, future work, we present ideas that we think would further optimise our project and add value to it.

2. Design

The overall design goal is to get a proper voxel-based structure running in which you can add and remove blocks from the world in real-time without major performance issues. While the game could be represented in basic colours for the blocks we want to have per-block textures and several different types of blocks with different textures. The player needs to be able to move around freely in the world, interact with blocks such as colliding with the blocks, jumping on top of blocks, going underneath blocks (or into caves) and of course add and remove them to/from the world.In order to establish a greater understanding of the features of the design, one has to understand some terminology that is common for voxel-based engines: voxels and chunks.

A. Voxel

ame Programming (Autumn 2017) (KGGAPRO1KU-Autumn 2017 A voxel (also referred to as Block in the context of our project), which comes from *Volumetric* and *Pixel* is a single point in 3D space. Much like a Pixel, it has a colour (or type) and a position within the space [6]. Both are visualised in some way and all are uniform in their size but do not directly have a size as such. Of course, due to the nature of computer graphics, the voxel needs to have some sort of size but is in most cases always the same, or just 1 unit in width, height and depth, so by representation they are discrete and uniform [6].

B. Chunk

In voxel games, a large number of voxels have to be rendered at the same time, while they are interactable and mutatable. Having all of these voxels being rendered and interactable at the same time can be a giant task for the CPU if the number of voxels is not restricted. To solve this issue many voxel engines split their voxels into larger structures: the chunk.

A chunk is some amount of voxels often, but not always, arranged in a cube-like shape. By splitting voxels up like this it is possible to only make the chunks that the player is closest to interactable and rendered. This can severely improve the performance of the game depending on how the chunks are implemented.

The size of chunk differs from game to game. For example; in Minecraft, [7] chunks are 16 blocks wide, 16 blocks long, and 256 blocks high, which is 65,536 blocks total. In this project, the dimensions of a chunk have changed often and been fitted to the optimisations that were implemented.

C. Initial Architecture

Using the mentioned design and its requirements we established an initial *Unified Model Language* (UML) diagram for the overall established architecture of the game. This formed a starting point when we started working on the project (See figure 1). The source code can be found at https://github.com/Jalict/voxel-game together with a build which can be found at https://github.com/Jalict/voxel-game/releases/tag/1.0.

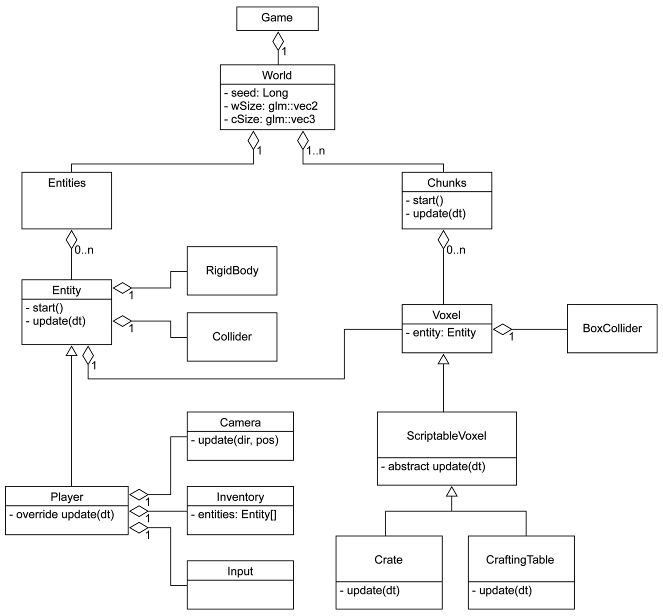


Fig. 1: Initial UML class diagram and relationships.

3. Implementation

This chapter explains the implementation and the overall architecture of the project. Besides this, the tools that were used during the project are described in short. Furthermore, the chapter contains the work division between all the members of the team.

A. Starting Point

As a starting point for the project, we have used exercise 6 - Wolf3D. In this exercise, we worked on a very simplistic clone of Wolfenstein 3D (id Software, 1992) in SRE. In this exercise, we had to complete a first person controller. Load the map from a JSON file and create 3D boxes with textures on them. Since our project requires a first person controller and is 3D with textured cubes this exercise formed a good starting point. With that said, most of the code used in the exercise has been replaced over the course of the project.

B. Result

The result of the project is a game with basic mechanics of voxel-based games, such as *Minecraft*. The player can place, remove and replace blocks within the world. They are free to move around in the world. The player can switch between several types of blocks that can be placed in the world. Furthermore, they can mine every block in the world, except for bedrock, which forms the bottom layer of the world. The terrain is generated using a very simplistic procedural method. See figure 2, 3, 4 and 5 for screen captures of the game. The game performs well and runs stable, even in debug mode in Visual Studio a steady 60 frames per second is achieved.

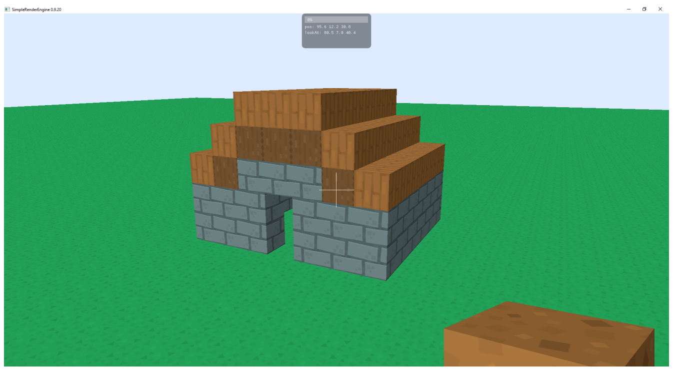


Fig. 2: Basic houses and a few blocks placed.

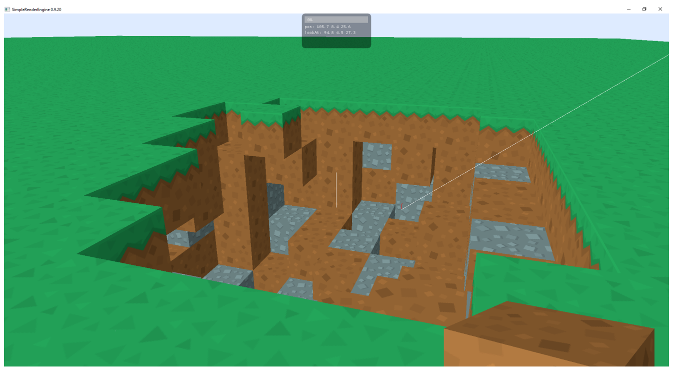


Fig. 3: Block removal from terrain.

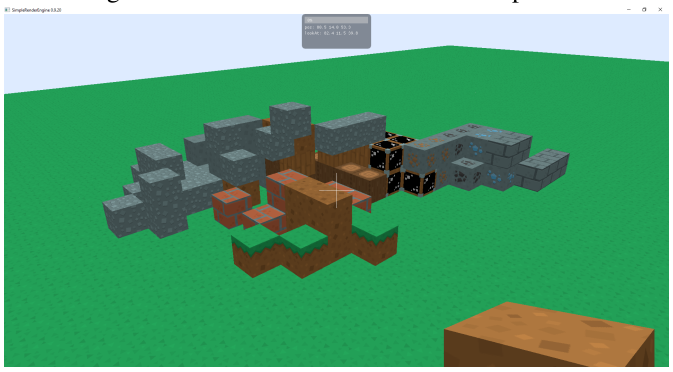


Fig. 4: Blocks can be placed as the player wishes.

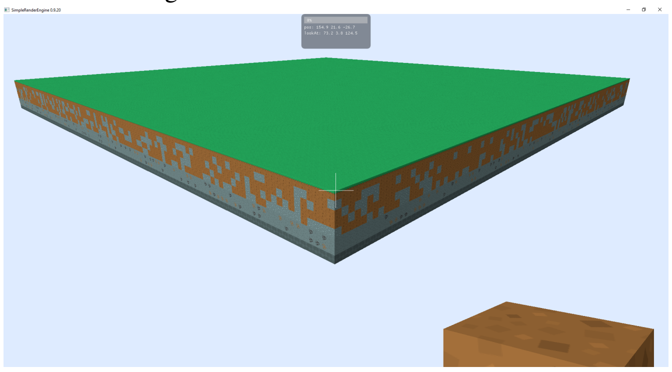


Fig. 5: An overview of the game world.

C. Architecture

The game uses a very basic class structure due to the scope of the project and the focus on optimisation. Most of the work have been spread out to to get basic Voxels working well together with basic interaction. This means that there is a only a few classes in the project (See figure 6):

- **Game** : *Handles instantiation, game-, render-loop, input and container for chunks, graphical user interface (GUI)*
- **Chunk** : *Container for voxels and partial voxel handling*
- **Block** : *Voxel data*
- **FirstPersonController** : *Camera handling, player input & movement, world interaction*
- **ParticleSystem** : *Container for particles and handling*
- **Particle** : *Particle data*
- **Physics** : *Holds the physics world, also serves as a wrapper for Bullet Physics*

Here is a description of the general instantiation and loop of the game.

The game starts instantiating everything, this goes for physics, render, textures/materials, particle system and chunks, light and the player. It proceeds to subscribe four functions, one for updating, one for rendering, one for keyEvents and one for mouseEvents. After that, the rendering finally starts.

All objects are instantiated before the game so that the least amount of objects have to be instantiated during run-time. Since instantiated during run-time can be quite costly to performance. Instead, the game uses flags for disabling objects whenever they should be inactive or destroyed, this gives much more responsive gameplay and stable frame rate. The drawback is memory usage can become quite high.

When game loop starts physics is updated first. After this, all the game logic is updated. FirstPersonController handles all the player input, movement and actions (such as mining and placement of blocks). Furthermore, the physic colliders for chunks are disabled or enabled based on the player's position (This is described in more detail in the chapter "Performance and bottlenecks"). The update proceeds to update all the chunks, updating the chunks only involves the recalculation of chunk mesh if it was flagged to do so. This only happens on a few occasions such as when a block has been added, removed or replaced within the chunk or a block that is an immediate neighbour of this chunk. As all the chunks have been updated the particles are updated. Particles occur when the player removes a block, at the location of this block particles get emitted for a short period. The particle system makes sure all the particles move and resizes over time.

For the render update, there are two passes. The first pass renders the whole 3D world, this includes the chunk meshes, the block that is in the player's hand and the GUI. The second pass is used to draw a crosshair on the centre of the screen.

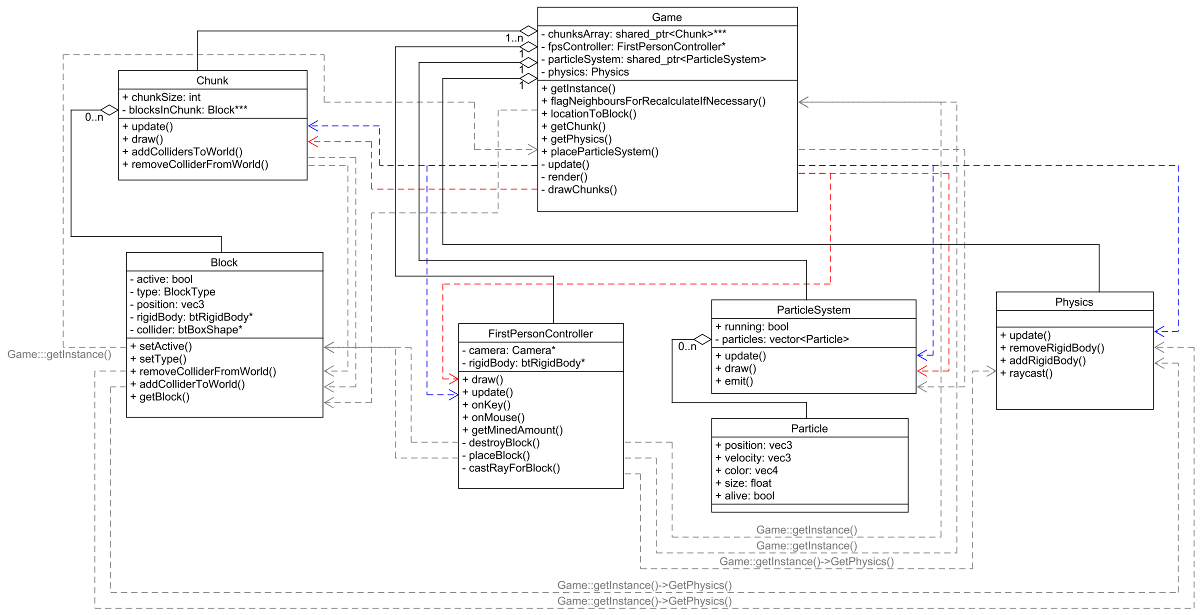


Fig. 6: UML diagram of the classes and the relationships between them. Blue lines indicates update calls, Red lines indicates render calls.

D. Block Structure

In this project, the block class contains a lot of important information. Since every block is a physical object in the game world they all have physics components, namely a rigid body and a collider. This allows them to be interacted with and collided by with physics objects. Also, all blocks hold their own position in world space. This is used when placing the physics components and is also useful when finding neighbouring blocks since one can find it by adding one on an axis of the current block. All blocks also store what type of block they are. All possible types of blocks are stored in one enum. The type of the block decides what texture is used for the block along with some type-specific block behaviour [1]. Lastly, all blocks have a flag that tells whether they are active or not. This flag determines whether the block should be rendered. Furthermore, only active blocks have their collider added to the world.

E. Chunk Structure

The most important function of the chunk is to hold a fixed portion of the blocks that the game world is made up of. This is done in a 3D array of blocks. The size of the chunk has been determined before runtime and is fixed. Chunks on their own do not represent anything, it only displays the blocks inside it. In other words, if there are no blocks in a chunk, the chunk is empty and does not display anything. Together, all the chunks hold all of the blocks in the world since blocks only exist within chunks. When a new chunk is instantiated it expects to get a world position passed in. It then proceeds to instantiate all the Blocks, a 3-dimensional array of Block, named *blocksInChunk* (See figure 6). The chunk uses randomness and conditioning in order to generate layers of blocks, such as grass on top, bedrock on the bottom, and random types of rock/ores in between which makes a basic procedural terrain generation.

Besides storing blocks the chunk is also responsible for rendering all the blocks it holds. The process of how this is done has been heavily optimised over the course of the project. The final version of chunk contains a mesh that is displayed. This mesh is the outer layer of blocks inside the chunk. It only contains the outer faces of the blocks inside the chunk. Furthermore, this mesh does not contain any faces that neighbour a block in any neighbouring chunk. Thus, in short, it only contains faces that are visible to the player. The mesh is calculated once and is only recalculated when it is flagged to do so. So, the only moment when a chunk mesh is recalculated is when a player places, destroys or replaces a block.

F. Physics

To simulate physics the game makes use of the Bullet Physics Library [4]. The Bullet physics library is a free and open-source 3D physics engine written in *C++*. Inside the project, elements are added to the physics world through the Physics class. The physics class forms a wrapper for Bullet. It instantiates and updates the physics of the game.

There are two main uses for physics in the project. First of all, it handles the collisions between the player and the blocks that make up the world. Secondly, it is used to detect what blocks the player is looking at and interacting with.

Collisions between players and blocks are achieved by adding colliders to each of them. The first person controller has a capsule collider of roughly two blocks high. It is a non-static collider and fully affected by physics, in other words, a non-kinematic rigid body character controller. The controller has, however, no friction to get smooth jump behaviour and is disabled to set to sleep. This is to prevent the controller from freezing when the character has not moved in a while. The blocks have a static box collider. Since blocks never move they can be made static. Blocks are simply removed from the physics world when they are destroyed by the player and added back to the physics world when a block is placed in that location.

In order to detect what block the player is looking at the first person controller uses raycasts. Whenever a player holds down a mouse button a ray is being cast from the centre of the screen in the direction the player is looking at. The position the raycast hits is the point the character controller is looking at. This position can be translated to a block, which can then be destroyed. Furthermore, for the placements of blocks, we make use of the hit normal. A small fraction of the hit normal is added to the position the raycast hits, to get empty spots to place blocks. Lastly, raycasts are also used to see if the character is grounded. By casting a ray straight down, and checking how far away the hit is we can know if the character is grounded. This is used to prevent the character from double jumping.

G. Input & Controls

Players can interact with the game using a keyboard and a mouse.The WASD keys control the player characters movement on the XZ plane, while the mouse decides which direction the player character is looking. These controls are very common for first person controllers. The character can rotate a full 360 degrees around the y-axis. However, rotations around the x-axis are clamped, to prevent the camera from turning upside down. It is also possible to sprint, thereby doubling the player characters movement speed by holding down shift. Lastly, the character can also jump by pressing space.

[1] A grass block turns into a dirt block when another block is placed on top of it and bedrock cannot be destroyed.

Interaction with the world mainly takes place with the use of the mouse. The player can destroy blocks by holding down the left mouse button. The progress of destroying the block is shown in the top of the screen with a yellow bar. Blocks can be placed with the right mouse button. To place different types of blocks, the player can press 'Q' or 'E' to cycle through the different types of blocks. The chosen type can be seen in the player characters hand in the lower right corner of the screen. Furthermore, the crosshairs displayed in the middle of the screen show what block the character is currently aiming at.

Furthermore, we have implemented various other keys that can be used to make it easier to explore and interact with the game world. First of all, the escape key can be used disable mouse lock. Thus, when pressing this button the mouse is released from the window. Pressing '1' enables the debug drawing of physics colliders and '2' toggles the profiler which displays information about the game's performance.

To make it easier to navigate the world it is possible to turn off collisions between the player and the blocks by pressing the 'T' key and flying can be turned on with the 'Y' key. When flying you can press the spacebar to go upwards and left-control to go downwards. It is also possible to toggle instant destruction of blocks by pressing 'U'. Lastly, you can change the placement mode with 'R', when enabled you replace blocks instead of placing new blocks on top of them.

H. Work Division

The project consist of a team of 3 members, *Mads Engberg*, *Frans Peter Larsen* and *Sven Santema*. Nearly all work has been completed by working together in the same room. This allowed for quick communication and problem-solving. Members often helped each other out with issues and all major decisions concerning architecture have been discussed with all members. Each member has taken equal part in the implementation. If a division has to be made the general workload is as follows:

- Mads Engberg
  - Blocks
  - Chunks
  - Chunk Mesh optimisations
  - First Person Controller
- Frans Peter Larsen
  - System Architecture
  - Blocks
  - Particle System
  - First Person Controller
- Sven Santema
  - Physics
  - System Architecture
  - Code Maintenance and Refactoring
  - First Person Controller
  - Blocks
  - Chunks
  - Chunk Mesh optimisations

I. Tools

The project uses CMake [8] for the generation of project solutions. CMake can be used to manage the build process for different platforms and environments. All of our project members develop on Windows, thus CMake was used to generate Microsoft Visual Studio solutions. The CMake lists included in the project are from exercise 6. With some minor modifications to suit the project. First of all, the CMake files have been adjusted to include Bullet Physics by default. Secondly, the block texture map and a JSON file are copied to the build folder.

Another important tool used over the course of the project was Git. Git allowed us to work on the project simultaneous and kept the versions synchronised between members. Versions pushed to Git should always build and we tried to the best of our ability to keep them bug-free. The Git that contains all the code and information necessary to compile the game.

The project is rather low on asset usage. However, the textures that are used for the blocks were obtained from *Kenneys Voxel Pack* [9]. The texture in the pack come in form of single *PNG* files which were then packed into a spritesheet using *TexturePacker* [10] in order to reduce the number of loaded resources and draw calls.

4. Performance & Bottlenecks

This chapter discusses all optimisations that have been implemented in the game. Throughout the project, the performance of the game has been a big challenge. Initially, before any optimisations were implemented the game ran poorly as soon as any meaningful number of blocks were loaded in.

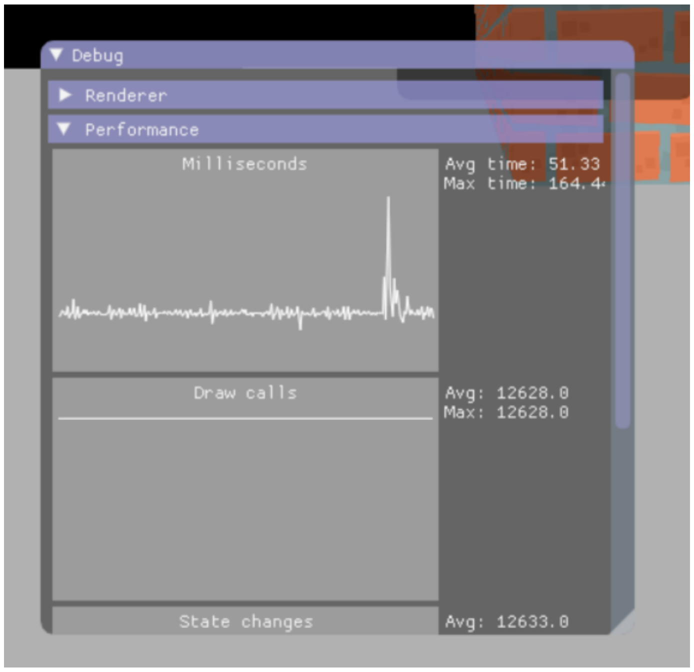
A screenshot of the first version can be seen in figure 7. In figure 7 we are rendering 12,625 blocks (101 chunks with each chunk being 5x5x5 blocks) along with a few test objects. As witnessed every single block has its own draw call. This results in an average time spent per frame of 51.33 milliseconds; this corresponds to a frame rate of 19.48 frames per second which is much lower than the goal of 60 frames per second (*16ms*). Besides this test showed that a scene with this amount of blocks had a memory consumption of 272MB. Compared to Minecraft (Mojang, 2009) this is a very poor performance, as Minecraft loads 65,536 blocks per chunk and has several chunks loaded while the frame rate remains stable [7]. This initial version of our project does not even come close to the number of blocks in one chunk in Minecraft.

All the measurements in this section are taken in Visual Studios Debug Mode and are therefore not representable for how the game will perform in Release Mode. However, the measurements are still good for individual comparisons. Since they have all been taken on the same machine.

A. Reducing the Number of Blocks Drawn I

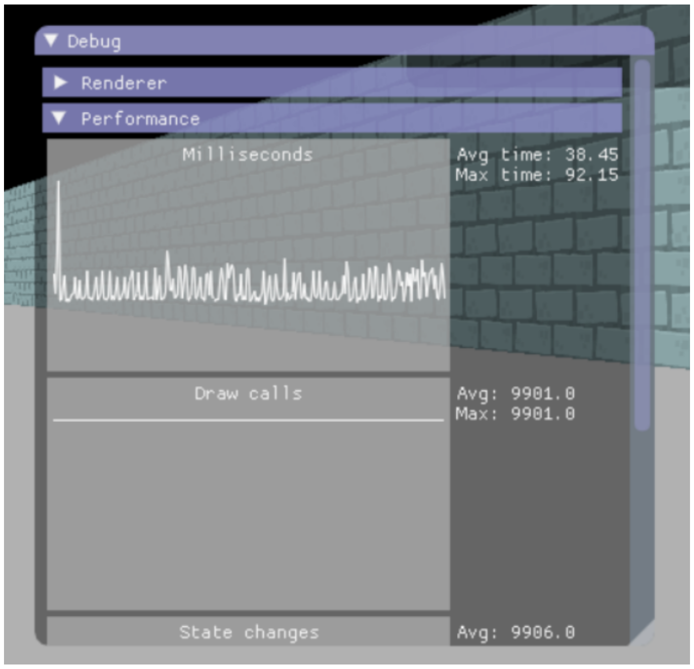
The first bottleneck seemed to be that every single block in the scene was being drawn, even when they are completely hidden from view by other blocks. To reduce the number of blocks that are drawn we decided to only draw blocks in chunks that are not obscured from view, e.g. they are surrounded by active blocks on all sides.

This small optimisation already had an increase in performance. The result can be seen in figure 8. The number of blocks in the scene is still the same as in figure 7, namely 12,626, but now the average time per frame has been reduced to 38.45 milliseconds, resulting in a frame rate of 26.00. the amount of render calls has also been reduced from 12,628 to 9,901. This means that 2,727 fewer blocks are being rendered. During this test all blocks in a chunk were active. Thus this is the best case scenario for this optimisation.



```
Avg time: 51.33
Max time: 164.4
Avg: 12628.0
Max: 12628.0
Avg: 12633.0
```

Fig. 7: A screen capture of the game's performance for the initial block generation before first optimisation pass.



```
Avg time: 38.45
Max time: 92.15
Avg: 9901.0
Max: 9901.0
Avg: 9906.0
```

Fig. 8: A screen capture of the game's performance after first optimisation pass.

B. Reducing the Number of Loaded Meshes

After this initial optimisation it had become apparent that the program was using a lot of memory. The reason for is was that every block in the scene stored its own mesh even if all the blocks had the same mesh. Therefore, the more blocks a scene has the more (unnecessary) memory it starts to consume.

To reduce the memory usage, one copy of each of the different types of blocks were instantiated once and then instead of each block having a mesh, each block had a pointer to one of these meshes. This also made it easier to change the type of a block since it was simply a matter of switching the mesh that a blocks pointer pointed to.

This optimisation resulted in a reduction from 272MB memory usage to 138MB. This was a very substantial optimisation.

C. Reducing Draw Calls

Reducing the number of blocks drawn, as presented previously, still did not have a satisfactory result. This was in large part because every single block still had their own draw call. This meant that the draw function was called and data was sent back and forth between the CPU and GPU many times a frame and created a bottleneck.

To solve this issue, we tried to reduce the number of draw calls. This was accomplished by constructing a combined mesh out of all the blocks inside one chunk and sending their collective geometry to the GPU in one call. This also has the added benefit of making it possible to only include selected faces in a mesh and therefore avoid rendering faces that are part of an active block, but not visible since they have a neighbouring block in that side. This optimisation has the effect that can be seen in figure 9 when using the same 12,626 blocks as the above sections. The amount of render calls per frame is reduced drastically from 9,901 as seen in figure 8 to 107. Still, the time per frame is increased drastically from 38.45 to 151.96 resulting in the FPS going from 26.00 to 6.58.

This might seem counter-intuitive and appear as a step back, since performance has not increased but decreased. However, since the number of draw calls has been reduced drastically we still considered it a performance improvement. Our hypothesis at the time was that the decrease in performance was due to the fact that we recalculate the mesh for each chunk every frame, this results in a net loss of FPS.

Another side effect of this optimisation is that every block no longer needs to have a pointer to their mesh as described in the section above because blocks no longer contain their own meshes. This results in an increase of memory usage from 138MB to 146MB which is a slight decrease, but nothing significant.

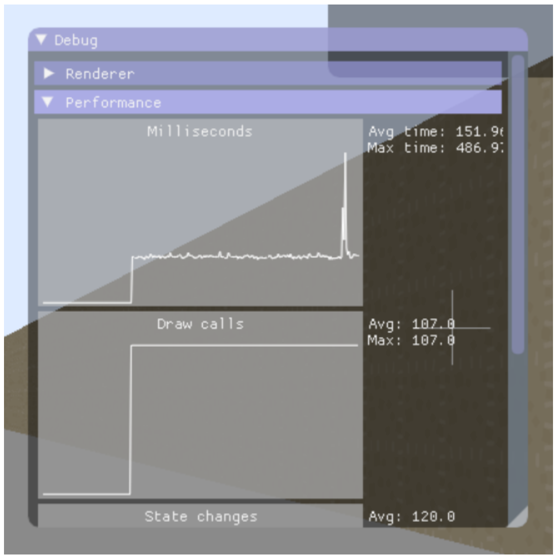


Fig. 9: A screen capture of the game's performance after third optimisation pass.

D. Reducing the Number of Blocks Drawn II

Another optimisation that was implemented after this, was a further reduction in blocks that are being drawn. In the section Reducing the Number of Blocks Drawn I, the number of blocks that are drawn were reduced. However, this implementation failed to search nearby chunk. Therefore the outer blocks of a chunk were always drawn when active. To further optimise this blocks could check if their neighbour in another chunk is active. This is done by inputting world space position of the neighbour block into a function that looks in the neighbour chunk to find the block in that position.

With this functionality, only the faces of the world that are facing outwards will be drawn.The mesh is ”hollow” so to say and seen from the inside it will even be invisible because of back-face culling.

As seen in figure 10, when using the same 12,626 blocks as the sections before, this optimisation has a decent effect on the time per frame lowering it to 104.62 from the previous 151.96. However, this still results in a frame rate of 9.55 FPS, which is much lower than the target of 60 FPS. This discrepancy even after many attempts to optimise is the result of re-calculating the mesh of a chunk every time we render it, as will be seen in the next section.

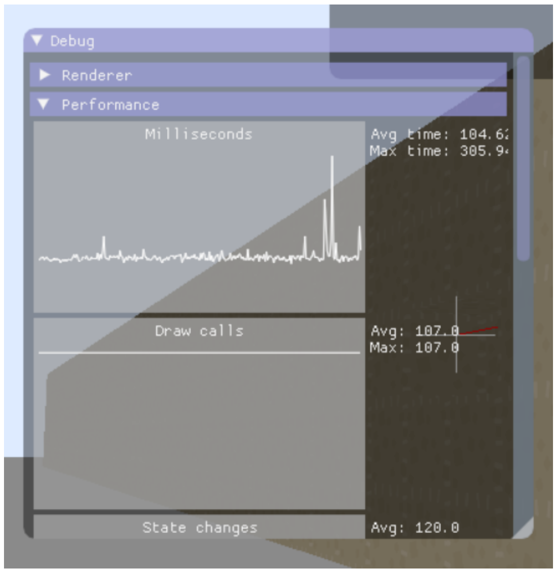


Fig. 10: A screen capture of the game's performance after fourth optimisation pass.

E. Only Recalculate Meshes When Changes Occur

This is the final optimisation that was done to the rendering of chunks in the project. This optimisation removed the huge performance hit that appears when the mesh of a chunk is re-calculated every frame. Instead of recalculating all chunk meshes every frame, chunks now have a flag that can be raised to recalculate the mesh. Chunks only recalculate the mesh when this flag is raised. This flag is only raised when a block has changed. If this block was in the centre of a chunk only that chunk needs to be recalculated. However, if the block was on an edge of the chunk the neighbouring chunks will also be flagged. This is necessary since there are no faces between chunk borders.

The recalculation of chunk meshes still hits performance. Although, now, it is barely noticeable. These spikes can be seen in figure 11. The two largest spikes in figure 11 were caused by the player first placing and then removing a block, causing that chunk's mesh to be recalculated both times.

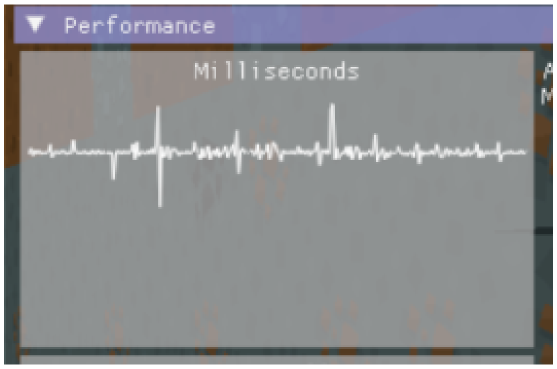
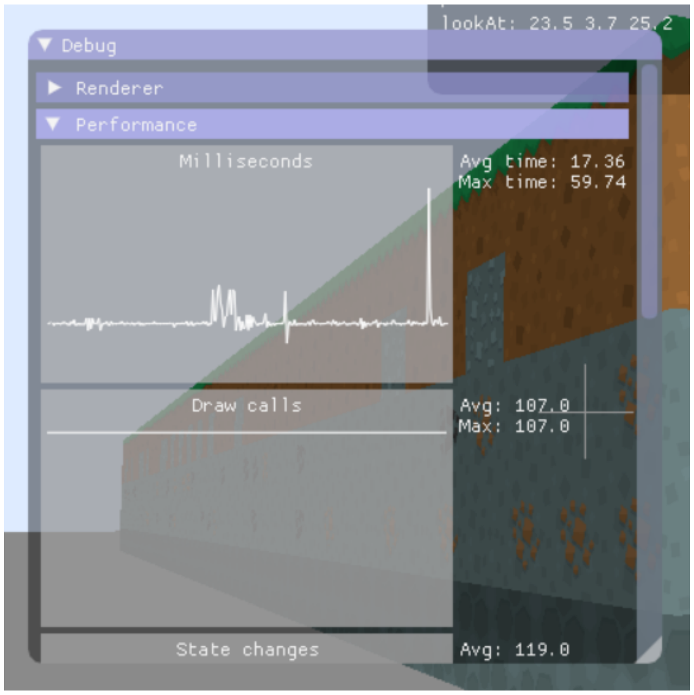


Fig. 11: A screen capture of the game's performance spikes as a result of recalculating the mesh of a chunk.

The overall increase from this optimisation is astonishing as seen in figure 12. The same 12,626 blocks as the above sections are still used, but now the time per frame decreases to 17.36 milliseconds which results in a frame rate of 57.60 FPS, which only fluctuates slightly and hangs around 60 FPS. Besides this, a test with 819,200 blocks was done and the game still ran at a stable 20.37 FPS. This performance is much closer to that of *Minecraft*.The final memory usage of the program ended as 115MB for the usual 12,626 blocks.



```
lookAt: 23.5 3.7 25.2
Avg time: 17.36
Max time: 59.74
Avg: 107.0
Max: 107.0
Avg: 119.0
```

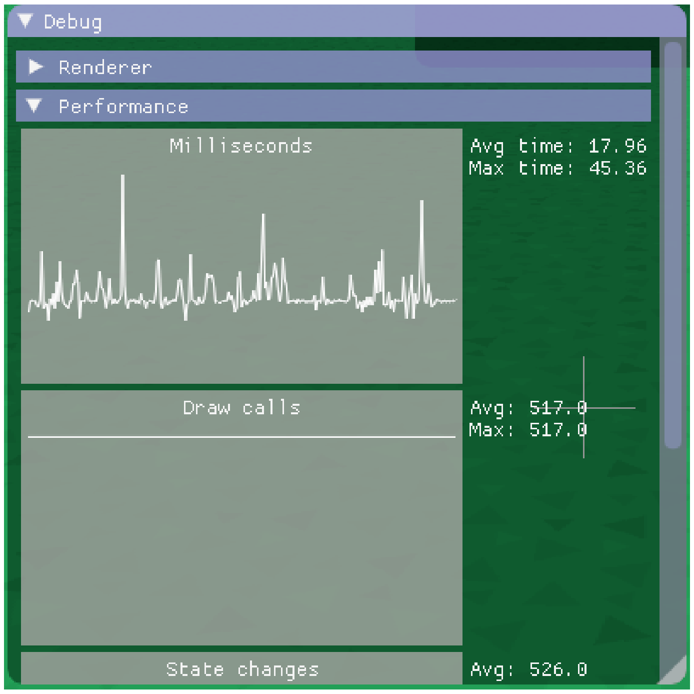
Fig. 12: A screen capture of the game's performance after fifth and last optimisation pass.

F. Physics Optimisation

After having optimised the meshes and render speed, we started to test the game with more chunks and blocks. Up to that point, physics had not really formed a bottleneck. However, as we started to increase the number of blocks it turned out that the rendering no longer formed a bottleneck. Physics started to become one, more specifically on startup. Up to this point, every block added its colliders to the physics world in the constructor. In tests with 16x2x16 chunks of 8x8x8 blocks, a total of 262,144 blocks, this resulted in a startup time of 2 minutes and 32 seconds. Furthermore, once the game has started physics reacts rather slow.

The solution we came up with is to only have colliders active in the chunk the player is on and its immediate neighbours. Blocks still create their rigidbodies when the world is created, however, they do not add their rigidbodies to the world until they are told to. Furthermore, they remove their rigidbodies from the world when they are told to. In the example from before this results in 18 chunks (3x2x3) that have their colliders active in the world as opposed to 512 chunks (16x2x16). This results in a startup time of 6 seconds, which is much more manageable and realistic. The downside of this optimisation is that startup time is decreased, however, the rigidbodies are now added and removed from the world during run-time. When walking around, upon entering a new chunk the colliders of a new chunk will be added to the physics world and those of another will be removed. The performance hit can be seen in the profiler in figure-13. The three big spikes are moments where the player switches to a new chunk.

Even though, this optimisation is far from being the most optimal solution it was deemed sufficient. Since, firstly, it reduces the start-up time significantly and secondly, the performance hit is neglectable. The performance hit that occurs only affects one or two frames, which end up being barely noticeable.



```
Avg time: 17.96
Max time: 45.36
Avg: 517.0
Max: 517.0
Avg: 526.0
```

Fig. 13: A screen capture of the game's profiler when the player walks over to a new chunk.

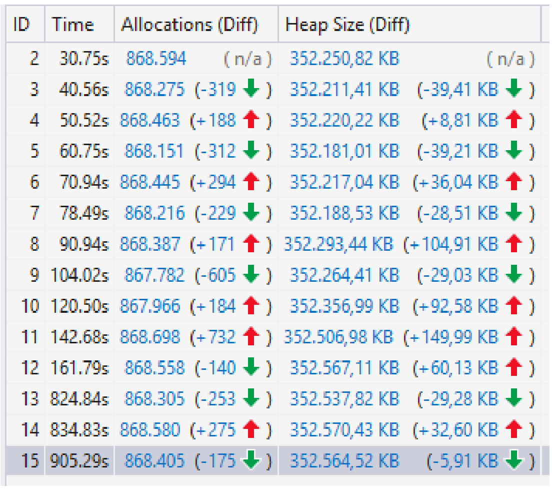
5. Discussion

Overall the project achieves the goals it set out to reach. We have realised a game in which players can place and destroy blocks. Furthermore, there are various types of blocks that can be used in the world. The first version of the game was poorly optimised, but after many iterations of optimisations the intended performance was achieved. Most optimisations have been focused on the rendering and drawing of the chunks since this was the main bottleneck. Near the end of the project, however, the rendering was optimised to such an extent that other bottlenecks began to become visible. Mainly, physics started to form a problem as the number of blocks and chunks in the world increased. By dynamically adding and removing physics colliders to and from the world this problem was solved. All these optimisations resulted in a game that performs very well. In table-I a performance overview can be seen of the game for various sizes of the game world. All these tests were run with a chunk size of 8x8x8 or 512 blocks per chunk. These tests were run on a Dell XPS 15 9560, with an Intel i7 2.80GHz CPU and a GTX 1050 graphics card. The tests where run in Visual Studio in debug mode.

TABLE I: Performance Statistics of the Game for Various Numbers of Chunks.

Blocks	**Chunks**	**Start-up Time (s)**	**Memory Usage**	**Draw Calls**	**Average Frame Time (ms)**
512	1	3	57 (MB)	6	16.68
51,200	100	4	174 (MB)	105	16.74
512,000	1000	9	868 (MB)	1005	18.22
1,024,000	2000	10	1,4 (GB)	2005	18.44

To test for memory leak we have run memory diagnostics in Visual Studio. The results can be seen in figure 14. We did not start to record until 30 seconds after start-up. This is to make sure that all elements of the game are initialized fully. After this we just let the game run until snapshot 7. Up to this point, we have not moved, destroy or placed any blocks. Snapshot 8 & 9 were taken after removing several blocks with instant mining on. Snapshot 10 to 12 were taken whilst placing blocks. Lastly, snapshot 13 to 15 were taken after the game had run for 15 minutes. The results show that the heap size is fluctuating whilst the game is running. However, there does not seem to be a trend of increasing memory as is expected with memory leaks. The lowest heap size recorded is 352.181,01 KB. This was whilst doing nothing at the start of the game. The biggest heap size was 352.570,43 KB after the game had run for almost 15 minutes. The difference between the maximum and minimum is 389,42 KB. One possible explanation for the fluctuations in memory could be that the chunk meshes vary in size. Destroying blocks or placing blocks could result in a mesh which exists out of more or fewer points, which in turn takes up more memory for the chunk mesh. This is however not directly visible in the measurements taken during memory diagnostics.



ID	Time	Allocations (Diff)	Heap Size (Diff)
2	30.75s	868.594 (n/a)	352.250,82 KB (n/a)
3	40.56s	868.275 (-319 ↓)	352.211,41 KB (-39,41 KB ↓)
4	50.52s	868.463 (+188 ↑)	352.220,22 KB (+8,81 KB ↑)
5	60.75s	868.151 (-312 ↓)	352.181,01 KB (-39,21 KB ↓)
6	70.94s	868.445 (+294 ↑)	352.217,04 KB (+36,04 KB ↑)
7	78.49s	868.216 (-229 ↓)	352.188,53 KB (-28,51 KB ↓)
8	90.94s	868.387 (+171 ↑)	352.293,44 KB (+104,91 KB ↑)
9	104.02s	867.782 (-605 ↓)	352.264,41 KB (-29,03 KB ↓)
10	120.50s	867.966 (+184 ↑)	352.356,99 KB (+92,58 KB ↑)
11	142.68s	868.698 (+732 ↑)	352.506,98 KB (+149,99 KB ↑)
12	161.79s	868.558 (-140 ↓)	352.567,11 KB (+60,13 KB ↑)
13	824.84s	868.305 (-253 ↓)	352.537,82 KB (-29,28 KB ↓)
14	834.83s	868.580 (+275 ↑)	352.570,43 KB (+32,60 KB ↑)
15	905.29s	868.405 (-175 ↓)	352.564,52 KB (-5,91 KB ↓)

Fig. 14: A screen capture of memory diagnostics run on the game.

6. Future Work

The work presented in this report presents the initial steps of our project to a flexible and robust framework for a voxel-based game. It is, of course, noteworthy to present that there can be many more features that can be added to the game such as *Inventory Management*, *Item Drop*, *NPCs* and many more, those features are of course desired but have more to do with game design than the actual aim of this project. This chapter highlights various optimisations and other topics we feel would have added value to this project but unfortunately did not have enough time for to implement.

A. Frustum Culling

One major optimisation that could be done is frustum culling [11]. With frustum culling, you only render meshes that are in the view of the game-camera. All objects that are not in the game cameras frustum are culled, in other words not rendered. When applying this to our game this would reduce the draw calls even further. Since there are chunks all around the player and the player cannot look at all of them at once.

One way this could have been implemented for the project is by creating functions that check whether simple geometrical shapes such as spheres, points and cubes are within the frustum and then cull all objects that do not pass this test. All chunks in the game would obviously be tested with the cube-function, while objects such as particle effects could be tested with the point-function. Keeping the test-functions to using simple geometric shapes makes sure that the check does not end up taking more calculation time than what would be saved by the frustum culling. For all objects that intersect one of the sides of the frustum and therefore are partly inside the frustum, the simple solution is to render those completely. In this project specifically, this is assured to never be a large performance decrease since the largest object possible is a chunk.

B. Physics optimisation

As described in the section *Physics Optimisations* the physics inside the project have been optimised in such a way that it no longer forms a bottleneck or a significant hindrance to performance. However, the implemented solution is probably not the most optimal. Therefore, we suggest further improvements to this system. Perhaps a solution with physics layers could be more efficient or instead of having colliders for each block, the chunk mesh could be used as a collider. Though, if the chunk mesh is used, the mesh collider changes a lot when blocks are placed and destroyed. Furthermore, it is no longer a primitive shape. Therefore, it is hard to say whether this is more efficient without having done any of the testing.

Additionally, the expertise of the project members in the Bullet Physics Engine is relatively low. None of the members had any prior experience with Bullet physics. Furthermore, the main bottlenecks during the project have not been with the physics. Therefore, most of the time was focused on resolving the bottlenecks rather than optimising physics after the physics were working.

C. Procedural generation

Minecraft, the game that this project mimics, uses procedural generation techniques to produce the game world. In *Minecraft* the player can walk around in an open world and wherever the player goes the surrounding terrain is generated. This feature would have been a natural fit for this project, since the generated hills and valleys would have made the terrain look much more impressive and interesting than just a flat plane of blocks. The shifting terrain would have also given a good impression of perspective, showing off just how many blocks the game is able to render.

This procedural generation would be accomplished by the use of a procedurally generated height map. A height map can be understood as a 2D matrix with numbers that, when laid out over some terrain, describes the height of this terrain. Such a height map could be generated with random numbers from Perlin Noise; noise that is known for giving a natural appearance [12]. In this project, these height values in the height map would determine up to what height, or world-y value, blocks would be active. Blocks above this height would be deactivated and thus not drawn

D. Serialisation

A desired feature, that would fit very well with procedural generation, would be the ability to save and load your world. This would require serialisation and deserialisation of the chunk, block and player data. If this was implemented, chunks could be loaded and unloaded dynamically and therefore we would not have to keep all chunks in memory anymore. Furthermore, the world could become (theoretically) infinite, since you would just load new chunks in the direction that the player is going and unload chunks the player has left.

7. Ludography

Minecraft (2009). Mojang/Mojang
Wolfenstein 3D (1992). id Software/Apogee Software & FormGen

8. References

[1] M. Nobel-Jørgensen, *Simple Render Engine*, 2016.
[2] *Simple DirectMedia Layer*. SDL Community, 1998.
[3] E. Catto, *Box2D*, 2007.
[4] E. Coumans, *Bullet*, 2005.
[5] *OpenGL Mathemathics*. G-Truc Creation, 2005.
[6] M. Jan, “Pixels and voxels, the long answer,” https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190, 2016.
[7] “Chunk - minecraft wiki,” https://minecraft.gamepedia.com/Chunk, [Online; accessed 10-12-2017].
[8] *CMake*. CMake, 2000.
[9] Kenney. Voxel pack. http://kenney.nl/assets/voxel-pack. [Online; accessed 09-12-2017].
[10] Codeandweb. Texturepacker. https://www.codeandweb.com/texturepacker. [Online; accessed 09-12-2017].
[11] “Camera frustum,” https://en.wikipedia.org/wiki/Viewing_frustum, [Online; accessed 10-12-2017].
[12] “Perlin noise - wikipedia,” https://en.wikipedia.org/wiki/Perlin_noise, [Online; accessed 10-12-2017].