Snowman Shooter - A 3D game in OpenGL

Name: David A. H. Kaan
Mail: dakaa16@student.sdu.dk
Supervisors: Rolf Fagerberg and Christian Kudahl
Course: DM842 Computer Game Programming

December 22, 2017

Contents

1 Introduction

In this project a 3D game is to be developed, that contains an implementation of different subjects, that have been presented during the course. The implemented game is a first person shooter style game, where you shoot snowmen. The development is done in C++ and OpenGL.
This paper will explore, how the 3D game was implemented, looking at design choices and improvements along the way. Each part of the program will be explained.

2 User Manual

This user manual is to inform the user how to open the game, as well as the goal and the controls of the game.

2.1 Opening the Game

The game is developed for the Windows operating system. Due to some of the external libraries' 64bit versions not working correctly in certain versions, there have only been made a 32bit **.exe* file. The *"Resources"* folder and certain **.dll* files, have to be in the same folder as the **.exe* file. The game **.exe* file can be found in the *"Game"* folder, under the name; *Snowman Shooter.exe*.

2.2 Goal

The goal of the game is to shoot snowmen. The more snowmen you shoot, the more points you will get. In order to survive you must stay clear of snowmen. If one snowman runs into you, you will be taken to the game over screen and your score will be written to the terminal. If the game window is covering the terminal you can use [ALT]+[TAB] to navigate to the terminal window. The longer you stay alive, the more snowmen will spawn.

2.3 Controls

In order to navigate the world, you use the [W] [A] [S] [D] keys to move forward, left, backwards and right. You look around in the world, by moving the mouse and shoot with the left mouse button. To get a closer look at what you are shooting at, you can align your head with the gun with [LSHIFT].

3 Libraries and Programming Language

The game is written in C++, using OpenGL. The choice of programming language is based on speed, which is a very important thing to keep in mind, when developing 3D games. Java has to be interpreted and thus it is genrally slower. However; it would most likely not have made a big noticeable difference, as the game is not very demanding. Java also comes with a lot more built-in libraries, that makes it much easier to manage a project of this size. Choosing C++ has proven, to be tedious, with all the external libraries that it needed. Managing the project, in regard to these external libraries, have taken time, that could have been spent on improving the game.

Other programming languages was considered as well, but the great debate was between Java and C++.

Three external libraries have been used, in order to get the game working: GLFW[10], glm[11] and Assimp[12]. GLFW is a multi-platform library, used in this project for creating a window, getting input from the mouse and the keyboard and getting the time the window has been open. The library glm(OpenGL Mathematics) is used for its matrix and vector data types when moving models. Assimp is used to import 3D models.

4 Implementation

This section will focus on the implementation of the *Snowman Shooter* game and describe the design choices. All parts of the implementation will be explained, to finally cover how everything fits together.

4.1 Resources

The resources for the game, can be found in the *Resources* folder. These files labeled resources, are files which are included in the game, but not recognized as code.
In this project there are two types of resources; shaders and models.

4.1.1 Shaders

Shaders are simple seperate programs, that turns inputs to outputs. The C-like language that they are written in, is usually run for the different GPU pipelines.[1] The two shaders used in this project, are a vertex shader and a fragment shader. The vertex shader gets matrices via *uniforms* from the render loop (*See section 4.8*) to translate the vertices, in relation to the virtual camera (*See section 4.2*). *Uniforms* are the only way to send and retrieve data to and from shaders, as they are completely seperate programs. In addition to translating vertices, the vertex shader implemented in this project also outputs texture coordinates, for the fragment shader to use.

Fragment shaders calculates the color of each pixel in the programs window. The fragment shader implemented in this game, gets the texture coordinates sent from the vertex shader, and sets the pixel colors according to the diffuse texture color.

Usually for bigger projects, you would have more than just two shaders, as you might want to render the models in your game differently - however, as lighting (which is handled a lot by the shaders) is not implemented in this game, there are not a lot of things to do differently in the shaders, that would still output the desired results. Thus the implementation of the shaders are very simple, but functional.

4.1.2 Models

Another resource are the models. These are the textured 3D shapes moving around in the game. All of the models in the game are **.obj*-files, due to the way they are loaded into the game *(see section 4.4)*. All the

models have at least one texture and a **.mtl*-file, located in the same folder as the **.obj*-file - again due to the method of loading. The texture is an image file and the **.mtl*-file, describes the material of the model, and (if multiple textures are present) which textures belong where.

4.2 Camera[5]

The implementation of the camera is a 'first person shooter'-style camera. This means that we want to be able to move and look around with the camera. OpenGL does not have any built in support for a camera, meaning we can not move around in the scene. What we can do is move everything according to the camera, to simulate that the camera is moving. However, before being able to move and look around, we need to define a camera in OpenGL.

First we need to define what the camera should see, and how it should be seen. We want the camera, to be able to see the perspective in the scene - meaning; models that are further away, appear smaller, than models that are closer. The library *glm* contains a function called *perspective*, which does exactly this. By passing a *fov* (field of view) value, we define how much of the scene we can see at once. Then by passing the width and height of the window the game is playing in, the *fov* will adapt to this size. The function takes two more parameters, defining how near and far things are before the camera can see it. The standard *fov* value is 45, the standard *near* value is 0.1 and the standard *far* value is 100. This means, that all vertices that are in the *fov* of 45 and are at least 0.1 away from the camera and maximally 100 away from the camera can be seen. The *perspective* returns a matrix, that makes sure, that these rules are kept. This matrix is called the *projection* matrix. The matrix is send to the vertex shader, in order to regulate the projection by multiplying it with the other matrices defining *gl_position*.[13].

The camera first of all needs a position. This position then functions as the origin of a local coordinate system. This coordinate system is the direction the camera is looking (z), up in relation to this direction(y) and right in relation to the direction(x). These are gotten as seen below:

$$\vec{Direction} = Normalized(\vec{Position} - \vec{Point}),\ where\ Point\ is\ what\ we\ are\ looking\ at$$

$$\vec{Right} = \vec{UP} \times \vec{Direction},\ where \quad \vec{UP} = (0, 1, 0)$$

$$\vec{Up} = \vec{Direction} \times \vec{Right}$$

This is called the Gram-Schmidt process.

Now a camera is defined, but we want everything to be moved around in relation to the camera. This is where the library *glm* is used. After having declared the direction, right and up vectors as the *glm* data type *vec3*, we can use the *glm* function *lookAt*. This function creates a 4X4 matrix based on the position of the camera, the point the camera is looking at and the *Up* vector, relative to the cameras direction. These are the only vectors

needed, as the function can calculate both the *Right* and *Direction* vectors based on the 3 arguments. The matrix returned, is calculated as follows:

$$lookAt = \begin{bmatrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Dir_x & Dir_y & Dir_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -Pos_x \\ 0 & 1 & 0 & -Pos_y \\ 0 & 0 & 1 & -Pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The reason why the camera position (*Pos*) is negative, is that we want everything to move opposite of the way the camera should be moving. This means, that when we tell the camera to move forward, everything in the scene moves backwards in relation to the camera.
In order to actually apply this matrix to everything in the scene, we send this matrix to the vertex shader just like we did with the *projection* matrix. This will change the *gl_position* by multiplying this matrix, with the other matrices that defines the *gl_position*.

This as well as movement, looking around and zooming is implemented in a class called *Camera*[6], which is mostly unchanged, from the original source.

When the camera is properly set up the simulation of the camera moving, can be implemented.

4.2.1 Movement

Implementing movement - meaning moving everything in the scene forward, backwards, left and right relative to the direction of the camera - is very simple. The $\vec{Direction}$ is used in order to define the $\vec{Point}$ used earlier in the *lookAt* function.

GLFW is used, to take input from the keyboard. When one of the specific keys are pressed, the camera position will change accordingly. The camera needs a movement speed, and also the time in between the last and current frame. The calculation for moving forward looks like this:

$$\vec{Position} = \vec{Position} + (\vec{Direction} * Speed * Time)$$

For moving backwards, you just subtract instead of adding.
For moving right and left, you do almost the same - only, instead of using $\vec{Direction}$ as moving direction, you use $\vec{Right}$ and add and subtract accordingly.

The movement of the the camera, could have been easily improved, by gradually lowering and rising the *velocity* relative to time. This would have made for a more realistic and fluid experience.

4.2.2 Looking around

In the former section, the implementation of moving in four directions relative to the direction of the camera, were explained. This section will explore how to change the camera direction based on input from the mouse, in order to simulate looking around in the world and how this is implemented.

In order to change this direction we use two values; *pitch* and *yaw*. These are part of *Euler angles* (*pitch*, *yaw*, *roll*). The value *pitch* is describing how much we look up and down. The *yaw* value describes how much we are looking left or right. The third angle; *roll*, is not used for these types of cameras.
A new direction vector for the camera is to be defined. It uses *pitch*, *yaw* and simple 2D trigonometry applied on different axes in a 3D coordinate system to calculate each coordinate.These are the coordinates for the new direction vector:

$$\vec{Direction} = \begin{pmatrix} Cos(pitch) \cdot Cos(yaw) \\ Sin(pitch) \\ Cos(pitch) \cdot Sin(yaw) \end{pmatrix}, \ where\ pitch\ and\ yaw\ are\ in\ radians$$

The values pitch and yaw are gotten from the mouse input. The *GLFW* library is used to hide the cursor, so it can not be seen and capture it, so it can not leave the screen. The library is also used to set a mouse callback, which is a function being called, every time the mouse is moved. Inside this function, the difference of the mouse position from the last frame to the current one is gotten. The cursor offset is gotten as *X* and *Y* coordinates. The offsets are then multiplied by a speed, defining how fast the camera shall turn. When the values of the offsets have been properly lowered, the *X*-axis offset is added to *yaw* and the *Y* offset is added to *pitch*. Having that implemented, the camera can simulate looking around in the world, but two things needs to be fixed. Firstly, the *pitch* value can get too high or low. If the value gets above 90 or below -90 the camera view reverses. This is easy to fix, by just limiting the *pitch* to values between 89 and -89. Due to floating point precision, the limiting numbers are one lower/higher than the actual limit.
The other problem is rather easy to fix as well; when moving forward, backward, right and left, we move relative to the cameras direction. As the *roll* angle is not implemented, this is not a problem when moving left or right. However, due to the *pitch* value, we can now freely move up and down relative to the game world. This is not something you would want the player to be able to do in a first person shooter. The ability to look up is wanted, but not the ability to walk upwards. In order to fix this, the camera $\vec{Direction}$ is to be changed when moving forward and backwards. In order to not move up or down, the Y-coordinate of the $\vec{Direction}$ is to be disregarded. This is simply done by multiplying it with a vector, as seen bellow:

$$\begin{pmatrix} Dir_x \\ Dir_y \\ Dir_z \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} Dir_x \\ 0 \\ Dir_z \end{pmatrix}$$

Having implemented all of this, the camera is working as it should, but one more feature have been implemented in the camera.

4.2.3 Zooming

The zooming is simply implemented, by changing the *fov* value in the *projection* matrix. The zoom is changed when a button is pressed, when it is released, the zoom, will return back to normal.

4.3 Using Shaders[1]

In order to use the shaders located in the resources folder, the program needs to get the GLSL code in the file, as a string. In order to do this and to use the shader, a class named *Shader*[2] is created in a header file.
The constructor of the shader class, takes the file paths to the vertex and the fragment shader as arguments and reads the files into two strings. Then both of the shaders are compiled, using the strings and finally a shader program is created.

Additionally a couple of functions are implemented in the class, to easily manage the shaders. The function *use* calls the function *glUseProgram*, given the shader program ID as argument. The function simply sets the shader program to be the program which is currently active. That means that any *gl* function calls to shaders (that does not take a shader as argument) will reference that shader program. For example when sending data to the shaders using uniforms, the function *glUniform* sends the given data to the currently active shader program. The class have functions, that sends data to the shaders, using uniforms as well as a function checking for compile errors for the shaders.

4.4 Importing and Drawing Models

OpenGL offers functions to draw different shapes, given an array of coordinates. The most common shape to draw, is a triangle, as you can create any imaginable 3D shape with triangles set to specific coordinates. However, just to describe a simple box with triangles, you would need to write 36 coordinates, each describing the position in the x-axis, the y-axis and the z-axis. Suffice to say, one does not want to create a snowman in an array, by simply typing in coordinates. This is where 3D modelling programs such as Blender[14] helps to abstract the incomprehensible data, to visual model manipulation. Saving the manipulated model as an **.obj*-file, saves the data to a simple file, that contains the vertices, the texture vertices, the vertex normals and the faces of the model.
In order to get the vertices as well as the texture coordinates imported from the **.obj*-file, a library called Assimp is used. This is a library that reads different 3D file formats, and makes it possible to use the recieved data in the implementation.

As loading models with textures is quite comprehensive, the loading of a model in this project, is divided into two classes: *Mesh* and *Model*.
The *Mesh* class[7] is responsible for drawing a mesh and applying a texture to it. This is done, by first creating buffer objects and a vertex array object. These then gets the vertices to draw and does some setup, for it to know how to handle the given data. These buffer and array objects, sends the vertices from the CPU to the memory of the GPU, so they are ready to be drawn.
The process of sending vertices to draw from the CPU to the GPU is not very fast. Thus sending lots of vertices at the same time a couple of times, is better than only sending a few vertices at a time relatively often.

Before drawing, the vertex array object is to be bound, for the vertex shader to know what to draw. Thus we can start drawing all the vertices by the function *glDrawElements*. This is fairly quick, as the verticies are already loaded into the memory of the GPU.[3]

The *Model* class[8] is where the models and textures gets loaded. The path to the **.obj*-file, is given as argument to the constructor. Then *Assimp* is used to read the file. *Assimp* does read many 3D-file formats, but the file format **.obj* was chosen as I am familiar with it. Also; the *Assimp* model loader, can not read all models even if they are the right file type. *Assimp* expects a certain format - thus a file format that I am familiar with was chosen.
The read **.obj*-file is stored as an *aiScene*, which is a struct from the *Assimp* library. It simply contains all information about the file. The *aiScene* has a tree-structure, where each node contains a set of indices. These indices describes data of the imported model.[4] Thus to set up all the meshes of the model, the tree is traversed recursively. Each mesh gets stored in a *vertex*-list as a *Mesh*(the class that was just discussed). The mesh then gets information about the texture and material of the mesh, from the *aiscene*. The texture image is imported, using the header file *stb_image*[9]. The class is set to look for the image file, in the folder of which the **.obj*-file is stored.
When all meshes has been stored in the *vertex*-list with textures and materials, the whole model can be drawn, by iterating over each mesh, and calling its *Draw* function.

The classes for importing models have been a great abstraction, from having to deal with the vertex array object and having to import models using *Assimp*. Having two separate classes also means, that you can use the *Mesh* without having to import a model. This could be useful, for loading very simple shapes, that you do not need 3D modelling software to describe.
The *Mesh* class is a direct copy from the source, but the *Model* class have been changed a little(see section 4.7).

4.5 Matrices

The implemented game uses matrices to scale, translate and rotate models. These matrices are generated by functions in the *glm* library. The functions for scaling and translating takes a *mat4*(4X4 matrix) and a *vec3*(vector 3) as arguments and outputs a *mat4* with the translation data corresponding to the function and the input. The function for rotating, takes a float as an argument on top of the *mat4* and the *vec3*. This is the rotation in radians.

An example of the use of matrices in the implementation is the gun model. In the game the gun is always in front of the camera. This is achieved by first scaling, translating and rotating the gun to be at the right position in front of the camera. The matrix returned, is then multiplied by the inverse of the cameras view matrix (see section 4.2.2). Thus, when the matrix is multiplied with the camera view matrix in the vertex shader, it is not translated relative to the camera, but rather just stays stationary in front of the camera.

This could also have been done by creating a new vertex shader, where the cameras view matrix is not multiplied with the rest. Then, when the gun was to be drawn, this should be the bound shader. However, as there are only two models (the other model being the 'Game Over' plane), where this behavior is needed, it did not make sense.

4.6 Snowman AI

The only AI(Artificial Intelligence) that is implemented in the game, is the AI for the snowmen. Seeing as though there are no obstacles for the snowmen to navigate around, a simple implementation of *seek* can be seen in the game. The snowmen simply moves in a straight line towards the player. This behavior is achieved, by subtracting the position of the player from the position of the snowman (disregarding the Y-coordinate). This vector is then normalized and multiplied by *speed* and the time from the last frame, to the current one, just like when the camera was moved around (see section 4.2.1). The resulting vector is then added to the position of the snowman, which is used to get the translation matrix. In order to make the snowmen face the player a function is declared, using *atan2* to get a new orientation, based on the X and Z coordinates of the vector that was added to the snowmans position.

The implemented AI is not very impressive, which is bound to happen, when there are nothing more for the snowmen to do, than seeking the player. Seperation could have been implemented in this simple world, to keep the snowmen from just merging into one snowman. If trees and rocks and maybe some houses were added to the game, the snowmen could use pathfinding to find the best way to the player. Additionally, the snowmen could try to avoid standing in a line, as one bullet can hit all snowmen, if they were all standing in a line.

4.7 Collision Detection

Collision detection can be done in many ways. Choosing one of the many methods is usually easy though, as they all - more or less - have a specific use-case. The use-case for the method implemented in the project, is not entirely fitting for the game, but was mostly fitting due to time constraints.
Collision detection has been implemented, using Axis Aligned Bounding Boxes (AABB for short). These are boxes wrapped around each model, that moves with the model, as it moves. The idea is to check if two specific boxes collide, instead of checking a collision between all the vertices of one model against another. Having bounding boxes makes the process much faster.

To implement this, the highest and lowest coordinate for each axis is needed for each model. These coordinates, describes the bounding box's dimension. These coordinates is gotten in the *Model* class, while exporting the data *Assimp* loaded to a data type, that OpenGL can draw. This is a small addition to the *Model* class, that makes sure that you can get the maximum and minimum coordinates of all models loaded with *Assimp*.

The bounding box is of course not a model that are being drawn and translated each frame. The box is actually just coordinates describing a box relative to the position of the given model. The box coordinates are calculated as seen below:

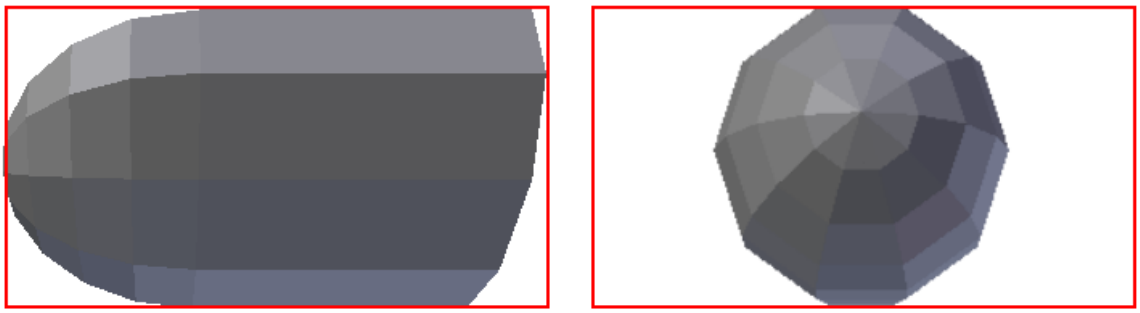
$$\vec{maxBox} = \vec{Position} + (\vec{MAX} * scaling)$$

$$\vec{minBox} = \vec{Position} + (\vec{MIN} * scaling)$$

In the equations seen above, $\vec{MAX}$ and $\vec{MIN}$ is the highest and lowest coordinates of the model. The value $scaling$ is a *float* describing how much the model have been uniformly scaled.

The collision detection method that has been implemented in this project is sufficient, but not very precise. It does wrap the models relatively good, but the bounding boxes does not rotate with the models. Axis Aligned Bounding Boxes, are meant for models that will not rotate, but both the snowman and the bullet rotates, making for imprecise collision detection, as visualized bellow:

Figure 1: The effect of rotating a model with Axis Aligned Bounding Boxes



As seen on the figure above, when rotating the bullet model 90 degrees on the Y-axis, the bounding box does not follow the bullet. Seen from this angle, the bounding box is too wide, however, if we looked from another angle, the bounding box would be too short.

A better way to have implemented the collsion detection, would have been by implementing bounding capsules, and making them rotate with the model. Capsules would fit much better around the models, and the added rotation, would make them somewhat precise.

4.8 Making the Game

Almost every last header and resource, that has made this game possible have now been discussed individually. This section will try to explain how everything ties together. Additionally some general improvements of the game, will be discussed.

The whole game comes together in the *main* class. This class is responsible for creating a window, taking keyboard and mouse input and also the rendering loop.
Before rendering anything, the window needs to be created. The library *GLFW* is used for this. The callback functions (input from mouse and keyboard) needs to be set. The depth buffer is to be enabled in order for the perspective to work properly. Then the shader (using the *Shader* class) and all the models, that do not have a class is initialized, using the *Model* class. And finally the render loop can be entered.
The render loop runs as long, as the game window is running. Each iteration the loop checks for input from the mouse and the keyboard, handles buffers, the camera and renders the game. First the *projection* vector is created and sent to the vertex shader using the class *Shader*. The *projection* matrix is often a constant, as you do not want to change what you see, and what you do not see, during runtime. Because of this, you would not usually have the *projection* matrix inside the render loop. However, in this project the *fov* value is changed for a zooming effect. Thus it is moved into the rendering loop.
The *view* matrix is gotten from the *Camera* class, gotten from the *lookAt* function. This matrix is also sent to the vertex shader.

Before rendering all the models, we check to see if the player is dead. If the player is dead, the "Game Over" plane model is rendered, but if the player is not dead, the game will start rendering. The order of which the models are rendered is of no importance, but each model will be explained in the order of which they appear in the code. Each model of course sends its created matrix to the shader, and get drawn.
The skybox model, is the box where the game takes place. The box is scaled to be big enough to create the illusion, that there is as sky and mountains in the distance. The model is translated to always be at the same position as the camera.
The snow ground that is being walked on, is scaled to a fitting size, that does not make the texture too pixelated and translated down the Y-axis, to appear under the camera.
The gun has been covered in detail (see section 4.5), thus the bullets are the next target. In order to keep track of all the bullets flying around, a class named *Bullet* is implemented. This class has a constructor that gets the initial position of the camera and the initial $\vec{Direction}$ of the camera. The constructor of course imports the model, as well as sets values in the class. The class also contains a function called *getShootingMatrix* that gets the shooting matrix for the bullet. The matrix translates the bullet according to the camera $\vec{Direction}$ gotten in the constructor, multiplied by a *velocity* and a time, just like previously decsribed when moving the camera (see section 4.2.1). The bullet is also scaled down to a fitting size. A function that updates the coordinates of the Axis Aligned Bounding Boxes is also implemented.
Back in the *main* class, an array of bullets is created. Every time the right mouse button is clicked a new *Bullet* is created and added to the array. In the render loop, a for-loop is created, that iterates over all the *Bullets*. Before sending the matrix gotten from the *getShootingMatrix* function and drawing the model, it is checked, whether or not the bullet is inside the skybox. If it is not, we skip drawing the bullet.
A class named *Snowman* is made in a header. This class is very similar to the *Bullet* class, only it creates the

matrix described in the AI section (see section 4.6).
Each iteration of the render loop, it is checked whether a new *Snowman* is to be instantiated and added to an array of *Snowmen*. This is based on a a dynamic spawn rate, that changes over time, spawning snowmen more and more often.
Just like with the bullets, all snowmen in the array is iterated over. Instead of checking where the snowman is, it is checked whether the snowman is dead. If it is, it is not wanted to either draw or check for collisions with this *Snowman*. However if the snowman is alive, the seek matrix is gotten and the *Snowman* is drawn.
Additionally; inside of the iteration of snowmen, bounding boxes are updated and a collision between the player (the camera) and a *Snowman* is checked. If there is a collision, the boolean *playerDead*, will be set to true, and the "Game Over" plane will appear in the next iteration of the render loop. If the player does not collide with the *Snowman* in the *Snowman* loop, a new loop is defined inside the *Snowman* loop. This loop again iterates over every *Bullet* in the array, but this time collisions between *Bullet*s and *Snowmen* are being checked. If a collision is happening, the *Snowman* is set to be dead, meaning it will not be drawn in the next iteration of the render loop.
When the rendering loop ends, the *main* function calls the function *glfwTerminate*, that clears all the memory allocated for the game and then it returns 0, to terminate the program.

Throughout the implementation sections, different improvement ideas have been mentioned. These improvements was of course on the topic of that section. In this section, more general improvements of the game will be proposed.
Having lighting would make the game look much more realistic. This could easily have been implemented, if it wasn't for the time constraint.
As mentioned in the AI section (see section 4.6), having trees, stones and houses would have made a much more convincing world. With some added fog to hide the rough edges and some snow particles, the world would look much better.
The texture or the ground is obviously tiled badly. This could easily be fixed with some image editing.
It is possible to reach the end of the ground plane. This could either be fixed by having some kind of fence near the end or by making the world infinite by drawing more planes, when the player is near an edge.
The bullets are partly rotated the right way relative to the gun, but not entirely. This could be fixed by adding some simple elements from the *Camera* class to the *Bullet* class.

5 Conclusion

In this project a 3D game has been implemented in C++ and OpenGL with other external libraries. The choice of language turned out to be an annoyance due to the small scope of the project. It seems that C++ is better for big projects, where this project, with a strict time restraint, might have been easier to implement in Java.
The implentation of the game have been fine, but a lot of things could have been improved. It has been achieved

to have implemented movement of models, texture loading, camera movement, AI and collision detection.

References

[1] Joey de Vries: Shaders,
https://learnopengl.com/#!Getting-started/Shaders

[2] Joey de Vries: Shader,
https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/shader$_s$.h

[3] Joey de Vries: Hello Triangle,
https://learnopengl.com/#!Getting-started/Hello-Triangle

[4] Joey de Vries: Assimp,
https://learnopengl.com/#!Model-Loading/Assimp

[5] Joey de Vries: Camera,
https://learnopengl.com/!Getting-started/Camera

[6] Joey de Vries: Camera,
https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/camera.h

[7] Joey de Vries: Mesh,
https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/mesh.h

[8] Joey de Vries: Model,
https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/model.h

[9] Sean Barett: stb image importer,
https://github.com/nothings/stb

[10] External library: GLFW,
http://www.glfw.org/

[11] External library: glm,
https://glm.g-truc.net/0.9.8/index.html

[12] External library: Assimp,
http://assimp.sourceforge.net/

[13] Joey de Vries: Coordinate Systems,
https://learnopengl.com/#!Getting-started/Coordinate-Systems

[14] 3D modelling software: Blender,
https://www.blender.org/