University for the Common Good

Coursework
Games Programming 3

Student ID: S1921018
Student Name: Wiktor Zborowski
Course: Games Programming 3
(MHI625659-22-A-GLAS)
Date: 14 January 2023

By submitting this assignment, I agree to the following statement:

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Wiktor Zborowski

Table of Contents

Chapter 1. Introduction. 3
Chapter 2. Spaceship Movement 4
Chapter 3. Camera Modes & Movement 6
Chapter 4. Game Conditions 9
Chapter 5. Conclusion 11
References 14

Table of Figures

Figure 1. An image showcasing the three-dimensional Cartesian coordinate system (WIKIMEDIA COMMONS, 2007). 4
Figure 2. A snippet of code showcasing the movement programming, found in MainGame.cpp class. 4
Figure 3. The vectors found in MainGame header file, where each has the shipSpeed in the proper axis. 5
Figure 4. Part of the key detection code in MainGame.cpp file. The camera is set to move around in proper directions by multiplying desired speed with the time. 6
Figure 5. Mouse keys detection found in MainGame.cpp file. 7
Figure 6. The function for changing the camera based on the set boolean. 7
Figure 7. Comparison between the ship-focused and asteroid-focused camera modes. 8
Figure 8. A snippet of fireMissiles function, found in MainGame.cpp file. 9
Figure 9. A snippet of updateMissiles function, found in MainGame.cpp file. 9
Figure 10. On the left - a game screenshot showcasing asteroid, which after getting shot at gets replaced with a monkey space base - seen on the right. 11
Figure 11. The victoryActions() function present in MainGame.cpp file. 11
Figure 12. A screenshot of game, showcasing the change of scenery after victory has been achieved. 12
Figure 13. A snippet of checkGameOver() function in MainGame.cpp file. 12
Figure 14. A snippet from MainGame.cpp file portraying gameLoop() function. 13
Figure 15. On the left the firstITimeSetup() function is presented, which is then portrayed in the game on the right. 13

Chapter 1. Introduction

The project is focused on developing and rendering a '*Space Explorer'* game. The code for the project came up mostly from the continued lab sessions (YOUNG, 2022), where the author of the project was learning different OpenGL concepts. This document focuses on the extension part of the project, hence for the main part of the code, refer to the previous Games Programming 3 content (YOUNG, 2022).

The project is being developed in Visual Studio 2019 (MICROSOFT CORPORATION, 2021). The developer chose to expand the code into a playable game. To achieve that, there were different aspects developed, such as spaceship movement, camera movement, different camera modes, missiles firing, and win-and-lose conditions.

Chapter 2. Spaceship Movement

As the first part of the extension, the researcher has been exploring the movement of the spaceship. Since the project is three-dimensional based (3D), there are 3 axes across the movement that can be done.

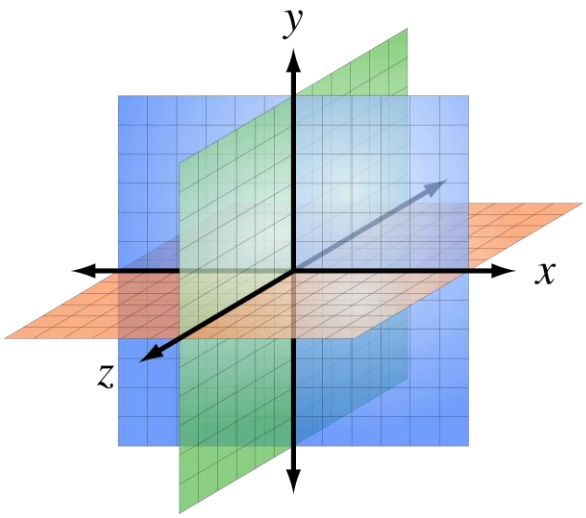

Figure 1. An image showcasing the three-dimensional Cartesian coordinate system (WIKIMEDIA COMMONS, 2007).

Hence, to be able to fully explore the space around, the player should be able to move in all these directions.

```
        case SDL_KEYDOWN:
            switch (evnt.key.keysym.sym)
            {
            /*
            Ship Movement:
            W/S - Moving on Y axis
            A/D - Moving on X axis
            E/Q - Moving on Z axis
            */
            case SDLK_w:
                ship.moveTo((*ship.getTPosition() + yMovement * deltaTime));
                break;
            case SDLK_s:
                ship.moveTo((*ship.getTPosition() - yMovement * deltaTime));
                break;
            case SDLK_a:
                ship.moveTo((*ship.getTPosition() + xMovement * deltaTime));
                break;
            case SDLK_d:
                ship.moveTo((*ship.getTPosition() - xMovement * deltaTime));
                break;
            case SDLK_e:
                ship.moveTo((*ship.getTPosition() + zMovement * deltaTime));
                break;
            case SDLK_q:
                ship.moveTo((*ship.getTPosition() - zMovement * deltaTime));
                break;
            /*
```

Figure 2. A snippet of code showcasing the movement programming, found in MainGame.cpp class.

The game processes inputs in a game loop, and if any of the designed movement keys (WSADEQ) are used, the ship moves in the set direction. The position of the ship is taken and then it is moved with the proper vector, with multiplication over the delta time, which is the time between each update of the game. The movement vector uses the set speed of the spaceship as a variable, hence they are automatically updated if the developer decides to change the speed manually.

```
glm::vec3 xMovement = glm::vec3(shipSpeed, 0.0, 0.0);
glm::vec3 yMovement = glm::vec3(0.0, shipSpeed, 0.0);
glm::vec3 zMovement = glm::vec3(0.0, 0.0, shipSpeed);
```

Figure 3. The vectors found in MainGame header file, where each has the shipSpeed in the proper axis.

The developer also wanted to work on working out the rotation of the spaceship, however, the idea was discontinued, due to no need for such action for the game.

Chapter 3. Camera Movement and Modes

One of the bigger parts of the extension material revolves around the camera. The project allows for moving the camera around the focused object, using arrow keys. The camera speed can be changed to either slow or fast by pressing Left Shift.

```
Camera Movement:
Left/Right Arrow - Moving on X axis
Up/Down Arrow - Moving on Y axis
Shift - Change speed
*/
case SDLK_LEFT:
    myCamera.MoveLeft(cameraSpeed*deltaTime);
    break;
case SDLK_RIGHT:
    myCamera.MoveRight(cameraSpeed*deltaTime);
    break;
case SDLK_UP:
    myCamera.MoveUp(cameraSpeed*deltaTime);
    break;
case SDLK_DOWN:
    myCamera.MoveDown(cameraSpeed*deltaTime);
    break;
case SDLK_LSHIFT:
    if (cameraSpeed == 10.0f) {
        cameraSpeed = 50.0f;
    }
    else
    {
        cameraSpeed = 10.0f;
    }
    break;
```

Figure 4. Part of the key detection code in MainGame.cpp file. The camera is set to move around in proper directions by multiplying desired speed with the time.

Additionally, the player can change the camera to follow asteroids instead of the spaceship. To switch between the camera modes, the middle mouse button is used. For the asteroid camera, there are also two more actions – left and right mouse buttons – that allow players to focus on different asteroids.

```
        /*
        * Asteroid Camera Mode
        * Left/Right Mouse Button - Change to next/previous asteroid
        * Middle Mouse Button - Change camera mode
        */
    case SDL_MOUSEBUTTONDOWN:
        switch (evnt.button.button)
        {
        case SDL_BUTTON_LEFT:
            cameraAsteroidCount++;
            break;
        case SDL_BUTTON_RIGHT:
            cameraAsteroidCount--;
            break;
        case SDL_BUTTON_MIDDLE:
            if(cameraAsteroidMode){
                cameraAsteroidMode = false;
                myCamera.MoveBack(50.0f);
            }
            else {
                cameraAsteroidMode = true;
                myCamera.MoveForward(50.0f);
            }
```

Figure 5. Mouse keys detection found in MainGame.cpp file.

The way different camera modes work is by looping a function that detects which camera is being used. There is an additional variable used, that holds which asteroid should be displayed. Additionally, if the player goes over the asteroid amount rendered in the game, it loops back to the first asteroid.

```
void MainGame::changeCamera()
{
    if (cameraAsteroidMode)
    {
        if (cameraAsteroidCount < 0)
        {
            cameraAsteroidCount = 19;
        }
        if (cameraAsteroidCount > 19)
        {
            cameraAsteroidCount = 0;
        }
        myCamera.setLook(*asteroids[cameraAsteroidCount].getTM().GetPos());
    }
    else
    {
        myCamera.setLook(meshSpaceship.getSpherePos());
    }

}
```

Figure 6. The function for changing the camera based on the set boolean.

Both camera modes allow for moving the camera around and using setLook function, which was initially developed as the main part of the code.

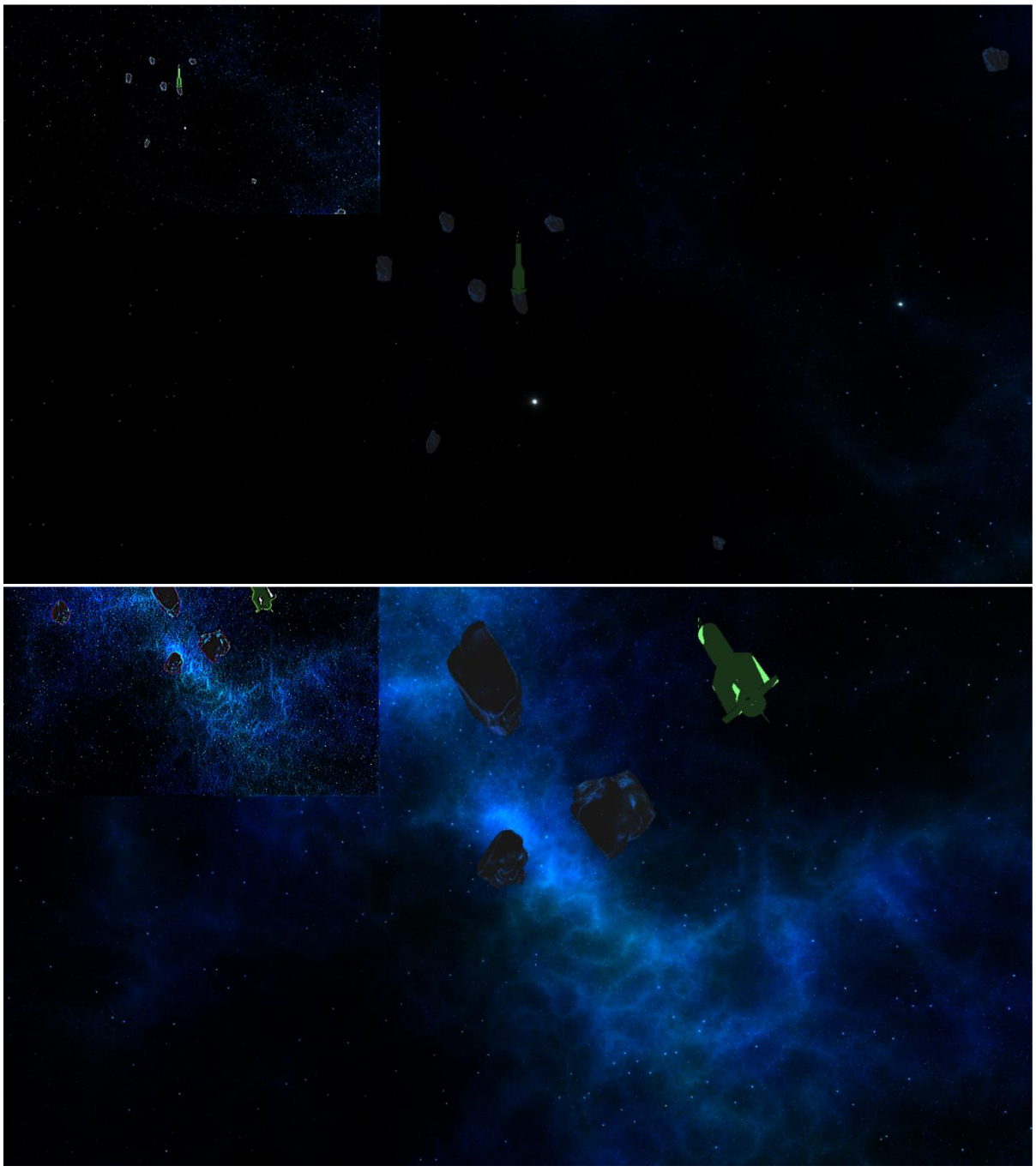

Figure 7. Comparison between the ship-focused and asteroid-focused camera modes.

While changing the camera mode is not required for the win condition of the game, it helps check the current progress.

Chapter 4. Missiles

A crucial mechanic for the game is the ability to fire off the missiles. The way the player can do that is by pressing the space button. Only one missile can be fired at a time. If the game is not over, function fireMissiles() will be called.

```
void MainGame::fireMissiles()
{
    missiles[missileLaunchNumber].moveTo(*ship.getTM().GetPos());
    missiles[missileLaunchNumber].setActive(1);
    canShoot = false;
    updateMissiles();
}
```

Figure 8. A snippet of fireMissiles function, found in MainGame.cpp file.

The function is set to pick the current missile number and transport it to the position of the spaceship, which shows that the missile is sent from the spaceship. The missile is then made active so it can be seen in the game, and then another function is called that will update the position of the missile over time.

```
void MainGame::updateMissiles()
{

    while (canShoot == false)
    {

        glm::vec3 shootVector = *asteroids[missileLaunchNumber].getTPosition() - *missiles[missileLaunchNumber].getTPosition();
        glm::vec3 shootVel = normalize(shootVector) * missileSpeed * deltaTime;


        missiles[missileLaunchNumber].moveTo(*missiles[missileLaunchNumber].getTPosition()+shootVel);

        if (checkCollision(*missiles[missileLaunchNumber].getTPosition(), missileScaleFloat*0.01, *asteroids[missileLaunchNumber].getTPosition(), asteroidScaleFloat))
        {
            missiles[missileLaunchNumber].setRotation(glm::vec3(rand(), rand(), rand()));
            asteroids[missileLaunchNumber].setActive(0);
            playSound(audioBoom, *missiles[missileLaunchNumber].getTPosition());
            missileLaunchNumber++;
            if (missileLaunchNumber == 20) {

                victoryActions();
                canShoot = true;

            }
            else {
                cout << "destroyed asteroid " << missileLaunchNumber << ". \n";
                canShoot = true;
            }

        }
    }
}
```

Figure 9. A snippet of updateMissiles function, found in MainGame.cpp file.

Since there can be only one missile shot at a time (controlled by canShoot boolean), the function consists of a while loop, based on the same boolean. The vector of the missile is calculated by subtracting the positions of the asteroid and missile, which is then normalized and multiplied by missile speed and delta time. This velocity is applied over the loop, by adding it to the current position of the missile. Then there is a collision check,

as when the missile hits the asteroid, the asteroid is ‘destroyed’ by deactivating it, the missile explodes leaving the monkey trail behind, audio is played at the final missile position, and then the next missile is loaded. After the action, another missile can be shoot. There are twenty missiles for the twenty asteroids in the game.

Chapter 5. Game Conditions

For the project, the author decided to expand the code into a proper game. For that purpose, the win-and-lose conditions were developed.

To win, the player needs to shoot all 20 asteroids. Every successful asteroid destroyed gets replaced with a monkey space base in its place.

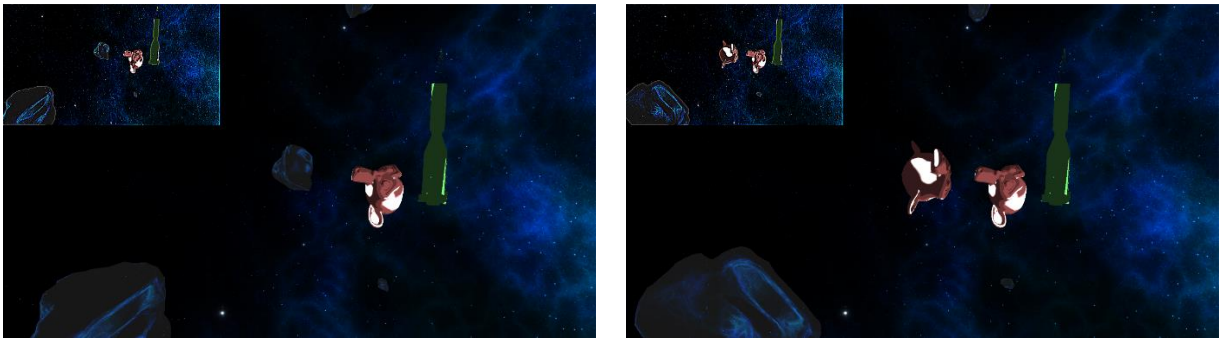

Figure 10. On the left - a game screenshot showcasing asteroid, which after getting shot at gets replaced with a monkey space base - seen on the right.

After all twenty asteroids are destroyed, a player lands on earth and can travel around. For this purpose, a victoryActions function is called. The function is to expand the filter screen created with Frame Buffer Object to the whole window (by changing the Boolean), as well as disable all monkey bases created beforehand, and change the scenery to Earth, by switching the faces used for rendering the skybox.

```
void MainGame::victoryActions() {
    cout << "VICTORY!\n";
    smallFilter = false;
    createScreenQuad();

    for (int i = 0; i < 20; ++i)
    {
        missiles[i].setActive(false);
    }
    vector<std::string> facesLand
    {
        "..\\res\\skybox\\right1.jpg",
        "..\\res\\skybox\\left1.jpg",
        "..\\res\\skybox\\top1.jpg",
        "..\\res\\skybox\\bottom1.jpg",
        "..\\res\\skybox\\front1.jpg",
        "..\\res\\skybox\\back1.jpg"
    };


    skybox.init(facesLand);
}
```

Figure 11. The victoryActions() function present in MainGame.cpp file.

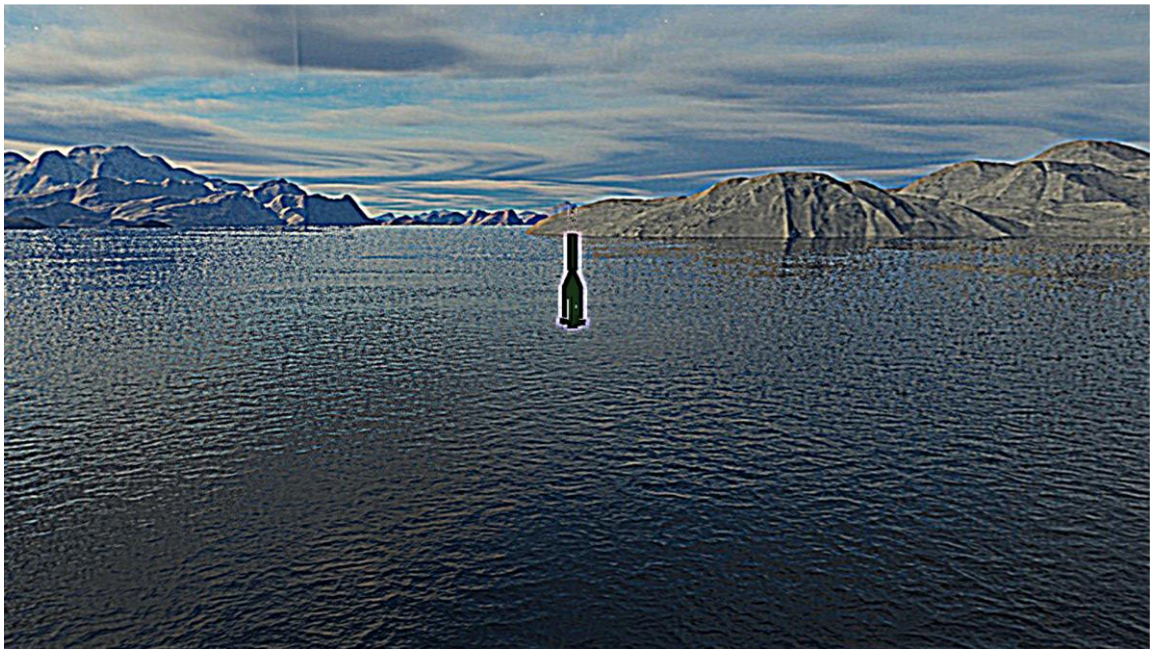

Figure 12. A screenshot of game, showcasing the change of scenery after victory has been achieved.

The player is left in the game to be able to explore the space around them. To exit the game, the player needs to press the Escape key.

The game can be lost if the player touches any of the asteroids with their spaceship. Although the player is still able to travel around the galaxy, they won't be able to reach earth and shoot missiles. Additionally, the galaxy map filter will start shaking. The way it has been achieved was by developing a checkGameOver() which is connected to the game loop.

```
void MainGame::checkGameOver()
{
    for (int i = 0; i < sizeof(asteroids); i++)
    {
        if (checkCollision(*asteroids[i].getTPosition(), asteroids[i].getSphereRadiusinGameObject(&meshAsteroid) * 3, meshSpaceship.getSpherePos(), meshSpaceship.getSphereRadius()))
        {
            if (asteroids[i].getActive()) {
                audioDevice.setlistener(myCamera.getPos(), meshSpaceship.getSpherePos());
                playAudio(audioBoom, meshSpaceship.getSpherePos());
                shakeCamera = true;
                life--;
                if (life == 0) {
                    gameOver = true;
                    SDL_ShowSimpleMessageBox(0, "Space Explorer - GAME OVER", "You lost the game! Press ESC to quit.", _gameDisplay.getWindow());
                }
            }

        }
    }
}
```

Figure 13. A snippet of checkGameOver() function in MainGame.cpp file.

Function loops over all asteroids and checks whether the spaceship hit a collision with any of them. This only applies if the asteroid is an asteroid itself, by checking the state of activity. If it is, then a sound effect is played, the top-left corner filter starts shaking by changing the Boolean of shakeCamera to true, and life gets taken. Initial idea was to give the player more than one hit point, but this was changed to only one due to the easiness

of the game. When the player runs out of health, a boolean gameOver is true, which stops the player from being able to shoot missiles, and a message is shown.

To make sure, that players will know how to play the game, the author of the project designed an initial message. It was desired by the developer that this message would show when the game has the skybox and models rendered, hence it needed to be put in the game loop. To run functions only once in the loop, another boolean was used called first, that is changed after these functions are run which in the end stops these functions to be run any further.

```
void MainGame::gameLoop()
{
    while (_gameState != GameState::EXIT)
    {
        processInput();
        currentCamPosition = myCamera.getPos();
        drawGame();
        updateDelta();
        playAudio(audioBackgroundMusic, glm::vec3(0.0f,0.0f,0.0f));
        if (first)
        {
            firstTimeSetup();
        }
        checkGameOver();
    }
}
```

Figure 14. A snippet from MainGame.cpp file portraying gameLoop() function.

The functions called on the first rendered frame are disabling all the missiles, as well as showing up the welcome message, which also describes the controls.

```
void MainGame::firstTimeSetup() {

        first = false;
        for (int i = 0; i < 20; ++i)
        {
            missiles[i].setActive(0);
        }
        SDL_ShowSimpleMessageBox(0, "Space Explorer", "Welcome to t
}
```

Figure 15. On the left the firstITimeSetup() function is presented, which is then portrayed in the game on the right.

Moreover, for improving the performance of the game, some parts of the rendering code were reorganised. Additionally, unused parts were deleted.

References

1. MICROSOFT CORPORATION, 2021. *Microsoft Visual Studio Community 2019* (Version 16.11.5) [computer program]. Microsoft Corporation. [downloaded 03 March 2020]. Available from: https://visualstudio.microsoft.com/vs/older-downloads/

2. YOUNG, B., 2022. Games Programming 3 [class]. *Lab 7.* Glasgow, November 2022.

3. WIKIMEDIA COMMONS, 2007. *Visual representation of 3D Cartesian coordinates on 2D sheet* [image]. Available from: https://commons.wikimedia.org/wiki/File:3D_coordinate_system.svg