Glasgow Caledonian University

University for the Common Good

Coursework
Games Programming 2

Student ID:	S1921018
Student Name:	Wiktor Zborowski
Course:	Games Programming 2 (M3I626039-21-A)
Date:	14 January 2022

Table of Contents

CHAPTER 1 - ABOUT THE PROJECT

The project is focused on rendering an application based on OpenGL. To make it possible there are dozens of different libraries and dependencies used. The majority of the program was set and programmed following the practical classes, which were taught by the module leader.

To make the project fun, the student decided to make it a playable game. The game is inspired by *Subway Surfers* (KILOO & SYBO, 2012), which is an infinite runner, where a player needs to avoid the trains. The game has a lot more features, but these were scrapped due to the complexity of the project.

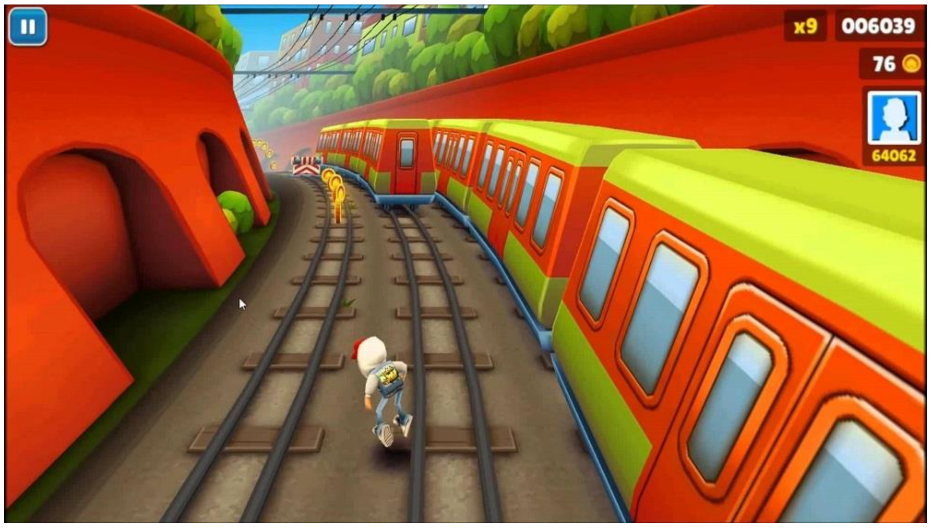

Figure 1. A screenshot from the Subway Surfers game (YILDIZ, S., 2017).

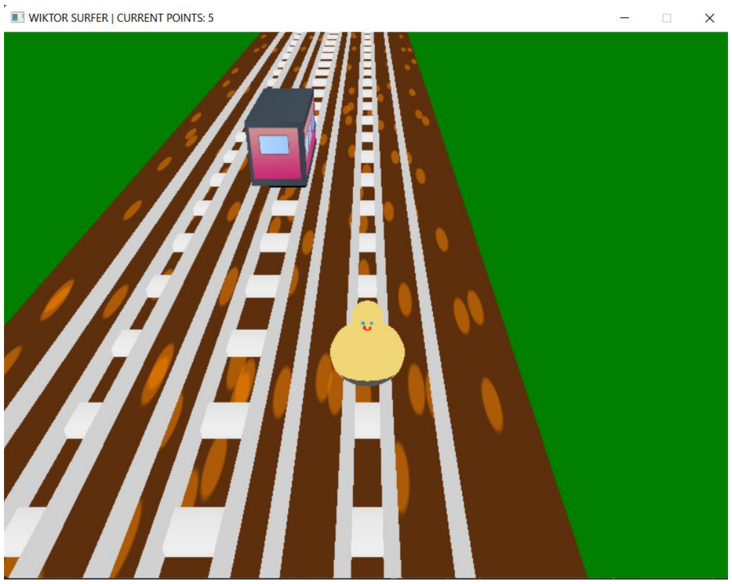

Figure 2. A screenshot from the game project.

CHAPTER 2 - SETTING UP THE ESSENTIALS

As mentioned earlier, the basics of the project were set on the practical session with the teacher. For the project, the required basics are loading files, mesh rendering, texture drawing, shader rendering, playing audio, collision detection, input keys.

Loading Files

This was achieved easily by implementing a class given by the lecturer in a practical class. The class used for this process is *obj_loader.* (Young, 2021).

Mesh, Texture & Shader Rendering

Making the visuals work properly can get tricky to be done. In this project, the code was generated based on the practical class from the lecturer, though for anyone who is desiring to copy the project on their own, it is recommended to follow a tutorial.

The classes that are responding to this process are Camera, Display, Mesh, Shader and Texture. Additionally, for Texture rendering there was a *stb_image.c* file required to load images properly. (NOTHINGS, 2021).

The Camera class is responsible for setting up the perspective of the game. Some functions allow moving the view.

The Display class is accountable for generating a window and initialising the attributes and buffers.

The Mesh class is obligated for calculating data of the vertices and loading the mesh properly. It also generates the vertex array and binds through each buffer, which is then passed to the GPU. It also stores spherical collisions.

The Shader class is used to load the shaders and update the view with the ones that were imported.

The purpose of the Texture class is properly loaded the texture file and buffering it.

Playing Audio

The audio is played using the Mixer functions from the SDL library.

Collision Detection

The collisions are detected by calculating the distance between two given meshes.

Input Keys

Each keystroke needs to be set to a function manually following the SDL documentation.

CHAPTER 3 - PROJECT RESOURCES

The external resources used in the project consist of different types of media.

Shaders

The shaders used in the game are ported from the practical classes of the module. The shader is a vertex-fragment.

Audio

There are three audio files in the game. Each of them follows the .WAV format.

The background music is *backgroundTrains* file (1 FINGER PIANO TUTORIAL, 2021). The music is a piano cover of the *Subway Surfers* (KILOO & SYBO, 2012) theme. It plays throughout the game until it is over.

There are two sound effects in-game. One is a train horn sound, which is played when a new train appears (SUSITHA SENANAYAKE, 2020). The other sound effect is played when the game is over, and this is a thud sound (SOUNDS, 2020).

To play the audio files in-game, the files need to be loaded by the audio manager, by setting the correct path of the sound file. Additionally, the files needed to be imported into the *Visual Studio* project “Resource Files” (MICROSOFT CORPORATION, 2021).

Models

The game models are saved as a .OBJ file. Each model used in-game is made by the student using the *Maya LT 2020* program (AUTODESK, 2019). The models were adapted between each other so they are scaled properly. The UVs were also set manually, so the models look interesting with proper texturing.

For the meshes to be rendered, each object file needed to be imported into the *Visual Studio* project “Resource Files” (MICROSOFT CORPORATION, 2021). Additionally, each file needs to be set as Excluded from the build, as otherwise, the program will try to compile these files, which will cause an error. Then the code needs to load the models and draw them with the specified position.

Textures

The textures for the models were made using *Adobe Photoshop* (ADOBE, 2020). Each texture is loaded in code only, and bound to a proper mesh. For an unknown reason, the UV textures were required to be flipped vertically to show properly in-game. This may be a sign that there is a mistake in the code, but this was left behind. An important note is that the size of the image files should not be too big, as initially, the textures were four times bigger, and this caused a big difference in the performance of the game.

CHAPTER 4 – INITIALIZATION

This section focuses on describing what is happening step by step.

Firstly, the main.cpp class is run. There is a `TrainGame` class instantiated, from which then the game is started with the `Start` function. The script of the class can be found in Appendix 1. From there, two functions are run. The first one is `InitialiseProcedures` which is a function that is meant to start loading the previously described essentials and resource files.

Firstly a display is initialised by calling a new Display instance. This initializes the SDL library functions that allow for the application to appear with proper OpenGL buffers. In the next step, the models are loaded. Each model is a separate object of the Mesh class and loads the model from the specified file path. Following that, the Camera is set up. There were different perspectives tried, until the student felt satisfied with the result. The camera is initialised to be a bit higher, and has set the field of view is 70. The perspective is also pitched with an angle of 0.6, to give a more three-dimensional, top-down look. In the next step, the shader is initialised. After setting the camera, there are default values set to the variables used in the game. The said variables are:

- `trainZ` – used to control the Z-axis position of the train
- `xMovement` – used to control the X-axis position of the player
- `speedTrain` – used to control the speed at which the train progresses (which increases progressively the longer the game is played)
- `trainLine` – used to control the X-position of the train

Following the set values, the track on which the train is appearing is randomised by calling the `RandomiseLine` function. The way the function randomizes the line is by calling out the C++ `srand` function which randomizes the seed used by the `rand` function. Without the `srand`, the lines are repeated between different instances of the game. After randomising the integer, there is a switch-case code, which sets the `trainLine` variable to different values based on the number generated.

The next step done by the `InitialiseProcedures` function is loading the audio files. This is happening with a reference to an `audioManager` which is an object created of the `AudioRelated` class. There are different functions used for the background music and the sound effects. This is because the background music is meant to be in a loop, and SDL Library has different memory management for constant music and temporary sounds. Once each track is loaded, the background music starts playing.

The `StartGameLoop` function is a loop that calls out different functions until the player quits from the game. The loop runs through three functions – `RecieveInputs`, `RenderGame`, and `CheckCollision`. The last function is only run while the game is not over, which is controlled by a Boolean.

The `RecieveInputs` function focuses on managing the player interactions. In the game there are four interactions programmed:

"A" / Left Arrow	Move the player left
"D" / Right Arrow	Move the player right
Escape	Quits the game
Space	Restarts the game (only if the game is over)

Figure 3. Inputs and their meaning in-game.

The movement actions are done by changing the value of the `xMovement` variable. Additionally, the camera is moved by calling the `MoveCameraX` function. To make the players not able to go out of bounds, there is a simple check performed before performing these actions, based on the value of the `xMovement`. The quit action is done by simply calling the `SDL_Quit` function, restarting the game works simply by reassigning the default values to the game variables, resetting the `gameOver` Boolean to false, and calling the `PlayBGMusic` function on `audioManager`.

The `RenderGame` function starts by resetting the display which is set to be green. This is to resemble the grass. After that, each texture is loaded from the specified file. Then, each mesh is set to generate based on set transform (position, rotation, scale). The two environmental meshes that don't change throughout the game are loaded at all times and differ only by the texture and scale of the model. The player and train models are loaded only if the game is not over and additionally to the mesh generation have the collisions data updated. They also have no static positions, as for the player the X-value of position is based on the `xMovement` variable, and for the train, the X-value of position is based on the `trainLine` variable, and Z-value is based on `trainZ` value. Other than these, the process for rendering a mesh is the same:

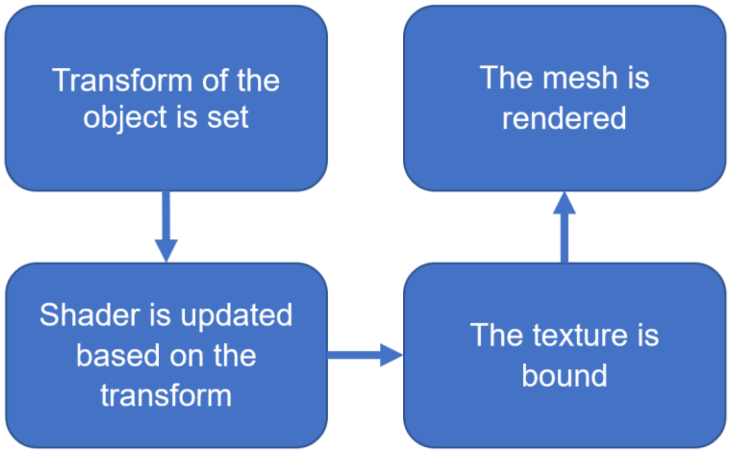
Transform of the object is set
Shader is updated based on the transform
The texture is bound
The mesh is rendered

Figure 4. Process of rendering the models in-game.

Once the meshes are rendered and the game is not over, the `trainZ` variable is decreased by the amount set in the `speedTrain` variable. If the value of `trainZ` is detected to be out of bounds, then its' value is set to the default value, which makes the train come from the top of the screen. The line is randomised, by calling the `RandomiseLine` function which gives the impression that the next train is coming, and a sound effect is played. These actions make the game infinite, as whenever the train comes out of bounds it is placed again in-bounds just outside of the view of the game.

Following this, there is a time function that is set to run every second. This is by getting the time in ticks using the `SDL_GetTicks` function and storing it in the `currentTick` variable.
If the current time is bigger than the previous time with 1000 ticks (which symbolises a second of difference), then the variables `points` and `speedTrain` increase. The points of the player are shown on the window title of the game. Finally, to make sure the check runs properly, the `currentTick` value is assigned to the `prevTick` variable.

The last function to be described is the `CheckCollision` function. It is called and has two objects' positions and radius passed, to calculate the distance between them. If it is lower than the sum of the radius of each object, then it means that the objects have collided. When this happens, the `audioManager` plays the game over sound effects and stops the music. Then the `gameOver` Boolean is set to true, which stops the game functions stop from running, sets the title of the game window to a game over and asks to press space to retry.

For the game, there would have been box-based collisions instead of spherical ones, but the student did not manage to get them working, and thus reused the sphere collisions code. The collisions were play-tested with many different values, and in the final version, the set collisions are the most optimal being not enormous, while detecting the player properly.

CHAPTER 5 – GAME

The game follows a simple loop of actions, described in Figure 5.

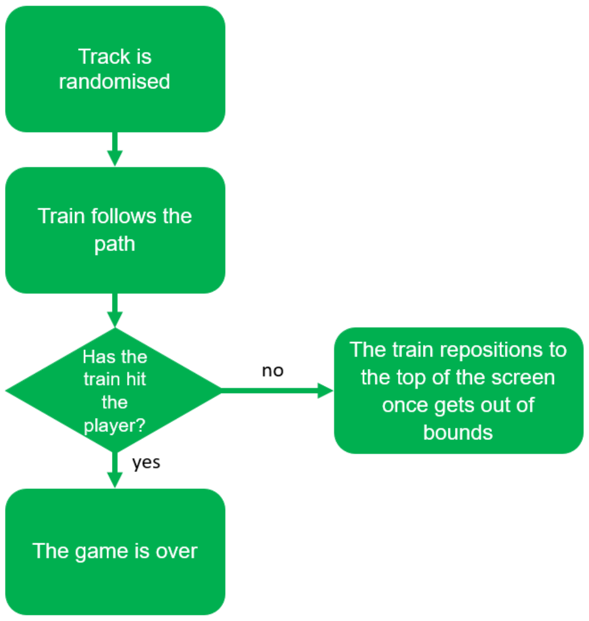
Track is randomised
Train follows the path
Has the train hit the player?
no
The train repositions to the top of the screen once gets out of bounds
yes
The game is over

Figure 5. "*Wiktor Surfer*" game loop.

The gameplay is in some way resembling the one in *Subway Surfers (*KILOO & SYBO, 2012), which was a planned outcome. A feature that could be implemented is increasing the hardness of the game, by adding a second train or more obstacles, like walls. Eventually, the number of tracks could be lowered to two instead of three, which would make the game harder. Currently, the highest score that has been achieved by the student is 1082 points. There is a recommendation from the student, to in order have more fun playing the game is doing something else while playing, making it more challenging to be multi-tasking.

REFERENCES

1. KILOO & SYBO, 2012. *Subway Surfers* [mobile game]. Sybo. [downloaded 08 November 2021].
   Available from: https://subwaysurfers.com/

2. YILDIZ, S., 2017. *How to Download Subway Surfers* [online]. [viewed 10 January 2022].
   Available from: http://appinformers.com/subway-surfers-guide/7471/

3. 1 FINGER PIANO TUTORIAL, 2021. *SUBWAY SURFERS – Theme Song – VERY EASY Piano tutorial* [video]. 20 March 2021. [viewed 28 December 2021].
   Available from: https://www.youtube.com/watch?v=gKBAnBL4kmc

4. SUSITHA SENANAYAKE, 2020. *Train horn sound – No copyright -Easy download link is in 1st comment* [video]. 04 December 2020. [viewed 28 December 2021].
   Available from: https://www.youtube.com/watch?v=0BcINnjrXbI

5. SOUNDS, 2020. *Dramatic Boom – Sound Effect* [video]. 05 December 2020. [viewed 28 December 2021].
   Available from: https://www.youtube.com/watch?v=DKoSh6J44SE

6. AUTODESK, 2019. *Autodesk Maya LT* (Version 2020.4) [computer program]. Autodesk. [downloaded 18 November 2021].
   Available from: https://www.autodesk.com/products/maya-lt/overview

7. 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/

8. ADOBE, 2020. *Adobe Photoshop 2020* (Version 21.2.4) [computer program]. Adobe. [downloaded 03 March 2020].
   Available from: https://www.adobe.com/products/photoshop.html

9. YOUNG, B., 2021. Games Programming 2 [class]. *Lab 4.* Glasgow, 27 October 2021.

10. NOTHINGS, 2011. *stb_image.c* (Version 1.33) [file]. [downloaded 20 October 2021].
    Available from: https://github.com/nothings/stb/blob/master/stb_image.h

APPENDIX 1 – TRAIN.CPP

```
#include "TrainGame.h"
#include "Camera.h"
#include <iostream>
#include <string>
#include "AudioRelated.h"
#include <time.h>


/* Creating transform variables for each game object */

Transform transform;
Transform playerTransform;
Transform envTransform;


/* Constructor */

TrainGame::TrainGame()
{
		trainGameState = GameState::PLAY;
		Display* gameRenderDisplay = new Display(); //pointing to display

    Mesh* playerModel();
		Mesh* envModel();
		Mesh* trackMidModel();
		Mesh* trainModel();

		AudioRelated audioManager();
}


/* Deconstructor */

TrainGame::~TrainGame()
{
		cout << "Deconstructing game....\n";
}


/* Public function called from main to start the game */

void TrainGame::Start()
{
		InitialiseProcedures();
		StartGameLoop();
}
```

```
/* Function to start the systems and setting default variables */
void TrainGame::InitialiseProcedures()
{
	gameRenderDisplay.initialiseDisplay();

	/* Loading models */
	playerModel.LoadModelFromFile("..\\res\\Models\\PlayerUV.obj");
	envModel.LoadModelFromFile("..\\res\\Models\\Ground.obj");
	trackMidModel.LoadModelFromFile("..\\res\\Models\\Tracksf.obj");
	trainModel.LoadModelFromFile("..\\res\\Models\\Train.obj");

	/* Setting camera */
	myCamera.InitialiseCamera(glm::vec3(0, 10, -15), 70.0f,
(float)gameRenderDisplay.getScreenWidth()/gameRenderDisplay.getScreenHeight(), 0.01f, 1000.0f);
	myCamera.Pitch(0.6f);
	shader.InitialiseShader("..\\res\\shader");

	/* Setting variables to default values */
	trainZ = 50.0f;
	xMovement = 0.0f;
	yMovement = 0.0f;
	speedTrain = 0.6f;

	/* Randomising the train line */
	RandomiseLine();

	/* Loading audio tracks and playing bg music */
	audioManager.AddBGMusicTrack("..\\res\\Audio\\backgroundTrains.wav");
	audioManager.PlayBGMusic();
	audioManager.AddSoundEffect("..\\res\\Audio\\TrainHorn.wav");
	audioManager.AddSoundEffect("..\\res\\Audio\\HitBoom.wav");
}


/* Function that loops the functions of the game */

void TrainGame::StartGameLoop()
{
	while (trainGameState != GameState::EXIT)
	{
		RecieveInputs();
		RenderGame();
		if (!gameOver) {
			CheckCollision(playerModel.GetCollisionPosition(),
playerModel.GetCollisionRadius(), trainModel.GetCollisionPosition(), trainModel.GetCollisionRadius());
		}
	}
}
```

```
void TrainGame::RecieveInputs()
{
	SDL_Event event; //calling out the events

	while(SDL_PollEvent(&event))
	{
		switch (event.type)
		{
		/* Keyboard inputs */
		case SDL_KEYDOWN:

			/* Movement keys inputs that check if player is in */
			if ((event.key.keysym.sym == SDLK_a) || (event.key.keysym.sym == SDLK_LEFT)) {
				//std::cout << "'A' Key or Left Arrow has been pressed \n";
				if (xMovement < 4.5f) {
					xMovement += 4.5;
					myCamera.MoveCameraX(4.5f);
				}
			}

			if ((event.key.keysym.sym == SDLK_d) || (event.key.keysym.sym == SDLK_RIGHT))
{
				//std::cout << "'D' Key or Right Arrow has been pressed \n";
				if (xMovement > -4.5f){
					xMovement -= 4.5;
					myCamera.MoveCameraX(-4.5f);
				}
			}

			/* When player is in game over, once the player presses space the game restarts with
ressetting the values aswell */
			if (event.key.keysym.sym == SDLK_SPACE) {
				//std::cout << "'Space' Key has been pressed \n";
				if (gameOver) {
					trainZ = 50.0f;
					yMovement = 0.0f;
					speedTrain = 0.6f;
					RandomiseLine();
					gameOver = false;
					points = 0;
					audioManager.PlayBGMusic();
				}
			}

			if (event.key.keysym.sym == SDLK_ESCAPE) {
				std::cout << "'ESC' Key has been pressed \n";
				trainGameState = GameState::EXIT;
			}
			break;

		case SDL_QUIT:
				trainGameState = GameState::EXIT;
				break;
		}
	}

}

/* Function that checks spherical collision between two objects based on their position and given radius. */
```

```
bool TrainGame::CheckCollision(glm::vec3 obj1Position, float Obj1Radius, glm::vec3 obj2Position, float
obj2Radius)
{
        float objectsDistance = ((obj2Position.x - obj1Position.x)*(obj2Position.x - obj1Position.x) +
(obj2Position.y - obj1Position.y)*(obj2Position.y - obj1Position.y) + (obj2Position.z -
obj1Position.z)*(obj2Position.z - obj1Position.z));

        if (objectsDistance*objectsDistance < (Obj1Radius + obj2Radius))
        {
                // Setting the game to be in "Game Over" state.
                audioManager.PlaySoundEffect(1);
                audioManager.StopBGMusic();
                cout << "___________________________\n\n     G A M E    O V E R    \n\n > Press Space to
Retry < \n___________________________\n\n";
                gameOver = true;
                return true;
        }
        else
        {
                return false;
        }
}


/* Function to randomise the train track on which the trains
will appear so they are not always on the same line. */

void TrainGame::RandomiseLine() {
        int a;
        srand(time(NULL)); //professional C++ technique to get proper random instead of always the same
random
        a = rand()%3;
        switch (a)
        {
        case 0:
                //cout << "Incoming: Right Line \n";
                trainLine = -4.5f;
                break;
        case 1:
                //cout << "Incoming: Middle Line \n";
                trainLine = 0.0f;
                break;
        case 2:
                //cout << "Incoming: Left Line \n";
                trainLine = 4.5f;
                break;
        default:
                cout << "Unknown randomise line value! \n"; //in case something went wrong used for testing
always for to keep in case of future expansion
                break;
        }
}
```

```
/* Big function that renders whole game and makes it work as intended */
void TrainGame::RenderGame()
{
	/* Setting green as bg because grass*/
	gameRenderDisplay.resetDisplay(0.0f, 0.5f, 0.0f, 1.0f);

	/* Loading textures used in game */
	Texture texturePlayer("..\\res\\Textures\\humanText.png");
	Texture textureTrain("..\\res\\Textures\\TextureTrain.jpg");
	Texture textureTrack("..\\res\\Textures\\track.jpg");
	Texture textureGround("..\\res\\Textures\\dirt.jpg");

	/* Train track Object*/
	envTransform.SetPos(glm::vec3(0, -30, 10.0));
	envTransform.SetRot(glm::vec3(0.0, 0.0, 0));
	envTransform.SetScale(glm::vec3(1.0, 1.0, 2.0));

	shader.BindTexture();
	shader.Update(envTransform, myCamera);
	textureTrack.BindTexture(0);
	trackMidModel.RenderMesh();

	/* Dirt model Object */
	envTransform.SetScale(glm::vec3(0.8, 1.0, 1.5));
	shader.BindTexture();
	shader.Update(envTransform, myCamera);
	textureGround.BindTexture(0);
	envModel.RenderMesh();

	/* Rendering the rest only if the game is not over */
	if (!gameOver) {

		/* Player Object */
		playerTransform.SetPos(glm::vec3(xMovement, 0, -3));
		playerTransform.SetRot(glm::vec3(0.0, 0.0, 0));
		playerTransform.SetScale(glm::vec3(1.0, 1.0, 1.0));

		shader.BindTexture();
		shader.Update(playerTransform, myCamera);
		texturePlayer.BindTexture(0);
		playerModel.RenderMesh();
		playerModel.UpdateCollisionData(*playerTransform.GetPos(), 10.0f); // for collisions
(environment didnt need that)

		/* Train Object */
		transform.SetPos(glm::vec3(trainLine, -2, trainZ));
		transform.SetRot(glm::vec3(0.0, 0, 0));
		transform.SetScale(glm::vec3(1, 1, 1));

		shader.BindTexture();
		shader.Update(transform, myCamera);
		textureTrain.BindTexture(0);
		trainModel.RenderMesh();
		trainModel.UpdateCollisionData(*transform.GetPos(), 240.0f);

		/* Moving the train */
		trainZ -= speedTrain;
```

```
			/* If train is over the screen reset it to top of the window and randomise line (giving impression
of next train) */
			if (trainZ < -15.0f) {

				trainZ = 55.0f;
				RandomiseLine();
				audioManager.PlaySoundEffect(0);
			}

			currentTick = SDL_GetTicks();
		}

		/* Function that runs every second if game is not over that adds points and speed of train. */
		if ((currentTick > prevTick + 1000) && (!gameOver)) {
			points += 1;
			speedTrain += 0.01f;
			cout << "POINTS: " << points << '\n';

			std::string a = "WIKTOR SURFER | CURRENT POINTS: ";
			std::string s = std::to_string(points);
			std::string b = a + s;
			char const* pchar = b.c_str();
			SDL_SetWindowTitle(gameRenderDisplay.window,pchar); //Setting the points in the title of
the windows as I could not get the SDL to render the text :(

			prevTick = currentTick; //setting the current time tick as the previous tick so it waits another
second
		}

		// Setting window title as game over.
		if (gameOver) {
			std::string s = "WIKTOR SURFER | GAME OVER. FINISHED WITH " + std::to_string(points) +
" POINTS. PRESS SPACE FOR RETRY.";
			char const* pchar = s.c_str();
			SDL_SetWindowTitle(gameRenderDisplay.window, pchar);
		}

		glEnableClientState(GL_COLOR_ARRAY);
		glEnd();

		gameRenderDisplay.bufferOpenGL();
}
```