David Grieco (dgri@itu.dk), Toto Christensen (totc@itu.dk)

Henrique Galvan Debarba

Game Programming (KGGAPRO1KU)

4 January 2022

Bézier Drive - A time-trial race game

1 Introduction

This report describes the result of the development process that we, David Grieco and Toto Christensen, went through to complete the final project in Programming at IT University of Copenhagen. The project aims to create a 2D top-down car driving game with procedurally generated tracks based on bézier curves. In the game, the player can drive a car on multiple race tracks to try and beat previous high scores by making better use of the realistic car physics present in the former. We used C++ as our programming language and Simple Render Engine (SRE) by Morten Nobel (*Nobel)*. The Simple Render Engine played an excellent foundation for our game as it provides an easy-to-access framework for building 2D (and 3D) games with the internal use of Box2D (*Box2D)* elements and OpenGL Mathematics (*GLM*).

2 Design

Our first design goal was to utilize bézier curves to be able to quickly generate a multitude of race tracks that feel smooth and interesting to traverse. We also set another essential design goal of using Box2D elements to create a believable simulation of car physics fundamentals. We achieved this through the use of the Joint elements.

2.1 Bézier Curves

We have chosen to employ bézier curves to construct our tracks and, therefore, the main brunt of our level diversity. A bézier curve results from linear interpolation and is used to describe a curve typically using two to four points, respectively a line, a quadratic- and a cubic bézier curve, the latter of which we have chosen to focus on in our project. Our choice to use bézier curves came down to several factors. Their computation is compact and lightweight. They can easily be scaled from 2D to 3D and further yet. This was important to us, as we originally intended on creating a 3D racing game but relished scaling down if needed, which we ultimately did. They furthermore provide an environment where the player can freely experiment and explore the physics of our game. Our reasoning for focusing on a cubic implementation is that their computation is sound no matter what points are used, that they are easily chainable, and that they can thereby form extensive paths needed for our tracks. We have, throughout this project, discovered that bézier curves are truly fascinating and a beautiful example of mathematics.

2.2 Joints

Joints are elements used to constrain bodies to the world or each other. Typical examples in games include ragdolls, teeters, and pulleys. Some joints provide limits so you can control the range of motion. Some joints offer motors that we can use to drive the joint at a prescribed speed until a prescribed force/torque is exceeded (Box2D Joints). We used Revolute Joints (Revolute Joint) and can be considered hinges. We define an anchor point for each body. The bodies are moved so that these two points are always in the same place, and their relative rotation is not restricted. Revolute Joints can also be given limits to only rotate in certain boundaries. We used this feature to limit the steering of the Tire bodies to simulate their behavior in the real world.

2.3 Simplified Architecture

At the start of the project, we started by laying down an initial Unified Model Language (*UML)* diagram to get a better hang on the development flow. The following image (*fig. 1.*) shows our starting project architecture's Class Diagram (N.B.: the diagram intentionally misses all the external libraries' classes as it would have been too convoluted to include them).

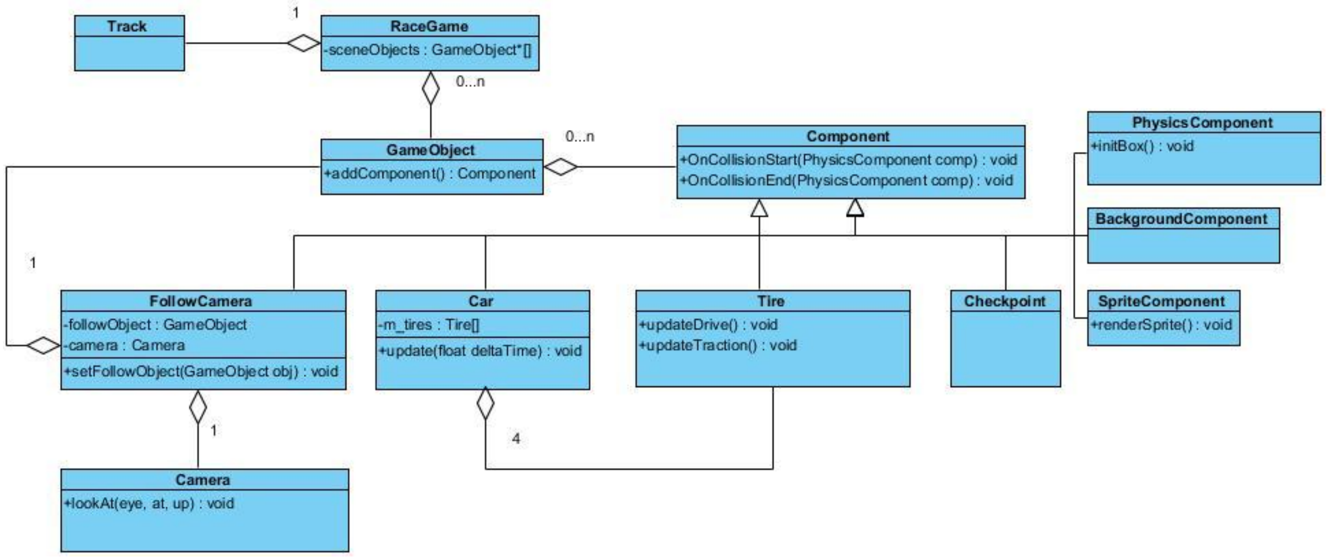

Track
1
RaceGame
-sceneObjects : GameObject*[]
0...n
GameObject
+addComponent() : Component
0...n
Component
+OnCollisionStart(PhysicsComponent comp) : void
+OnCollisionEnd(PhysicsComponent comp) : void
PhysicsComponent
+initBox() : void
BackgroundComponent
1
FollowCamera
-followObject : GameObject
-camera : Camera
+setFollowObject(GameObject obj) : void
Car
-m_tires : Tire[]
+update(float deltaTime) : void
Tire
+updateDrive() : void
+updateTraction() : void
Checkpoint
SpriteComponent
+renderSprite() : void
1
4
Camera
+lookAt(eye, at, up) : void

Fig. 1. Initial architecture's Class Diagram.

3 Implementation

3.1 Starting Point

The first playable version of our game utilized an incomplete simulation of the car physics that worked by having a single Box2D Box body to which the forces were applied. Although functioning, it was a clunky representation of car physics; most realism in creating a vehicle simulation comes from simulating its wheels separately. Although without all the functionality we were looking for, we used this version to test the track generation first.

3.2 First Results

With our first prototype, we generated a Track from a list of clicked points on the screen and rode on it with the first version of the car. At this point, we realized that the Track generation needed more work as it was failing in creating a smooth mesh on sharp turns. At this point, we also decided that the car was not realistic in how it was riding on the Track. Both solutions to those problems are better explained in the upcoming sections. We have to credit *iforce2d.net* for their well-documented tutorials on Box2D physics that gave us a starting point and a base project to build the car physics onto (*iforce2d)*.

3.3 Architecture

The final architecture of the project is close to the initial one we had in mind, apart from the implementation and structure needed for the car physics and saving/loading functionality. We got rid of the old car controller, which tried to achieve realism far too simplistically, and created a new car component. This car component, in turn, owns and controls four-tire components but controls the speed of neither them nor the car, as the tire components themselves do this while the car component only controls the steering. With this change, the car physics started to feel realistic even though they still needed to be tweaked.

In the following sections, we will explain how our main features work in detail, our roles in making the latter, and the tools we used for it.

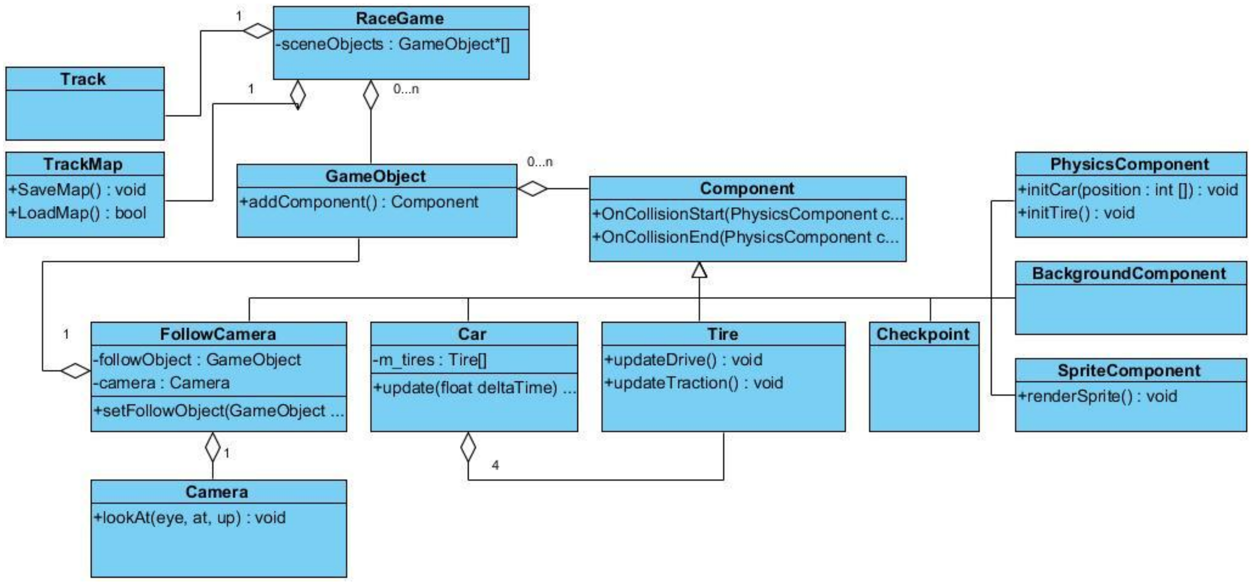


Fig. 2. Final Architecture.

3.4 Track Generation and Loading

To begin generating a new track/level, a bézier path is created based on supplied mouse positions; we use the control points of the path to calculate the needed drawing- and tangent points upon which the track- mesh and barriers are based.

To ensure quick loading of our levels, we decided to save this, among other information, in JSON format to file. This format offers information that is easily readable and editable by hand and can also be accessed logically through scripting. For each level of our game, we store how many laps need to be driven before the level is won, which scale the track has, its name, identifier, as well as drawing- and tangent points as described. Our scoreboard is also saved as and retrieved from JSON when a level has been finished. Loading to and saving either is adherent to the current track and is therefore handled in TrackMap.

3.5 Physics

Our project's primary approach to physics was sticking to the assets already made and provided by our teacher Henrique Galvan Debarba in most of the Game Programming course through the PhysicsComponent. Even though we had to go deeper and work directly with the *Box2D* components (*Dynamics Module*) in some cases. This was the case with implementing the car- and tire components and initializing their corresponding bodies in the PhysicsComponent.

3.5.1 Car Physics

The car component is the main entry point in the car physics simulation as it holds a reference to the four-tire components needed to make the car move. It has to be attached to a GameObject whose physics have been initialized through the "*initCar"* method of the PhysicsComponent. We created this method to initialize a different type of shape from the ones already present in the script. This method makes the polygon shape for the car physics and sets its density accordingly. When the car component is attached to the latter, its constructor is called. All tire objects are then created and initialized by attaching a PhysicsComponent and calling the *"initBox"* method to construct and set the box shape for each. During this initialization process, the car component also initializes its revolute joints relative to each tire to use them in the update function of the former. During the update cycle, the car component calls each wheels' update function and controls the cars' steering by updating the angle of the joints and their limits. This way, the force impulses are added on the tires' bodies and not on the car body directly, while the latter is only in charge of updating the tire components' scripts and controlling the steering, mimicking what happens in the real world.

3.6 Input & Controls

The player can interact with the game world by using the keyboard. The car accelerates using the 'Up Arrow' and brakes/goes backward using the 'Down Arrow'. The 'Left-' and 'Right Arrows' are used for steering. The 'B' key can be used when at high speed to drift in the direction the car is steering. The ' Spacebar ' is the main interact button used to change the game state from Ready / Won / Finished to Running. The player can also press the 'R' key to reset the track during the gameplay. For Debug purposes, we also left the functionality that lets the user hide graphics, and it can be activated by pressing the 'G' key. The 'D' key toggles the doDebugDraw function that draws every PhysicsComponents' shape present in the game world.

During production, we also constructed a track editor mode. This mode can be accessed by inputting 'Editor' as 'player name' when the game starts and will result in a blank screen where the following keys are available. '1' to save the current mouse position, and if enough points are generated, draw the resulting bézier curves. '2' to prompt saving of the current track and displaying it. '3' to clear all points and restart track-creation. 'S' to save the current track without displaying it, and 'L' to load the desired track. This mode is not a finished game feature, and caution should be taken to avoid overwriting existing levels. It should further be known that it is quite possible to create impossible levels using the current editor implementation.

3.7 Work Division

The project consists of two members, David Grieco and Toto Christensen. We mainly worked on the project online due to covid restrictions using audio- and video sharing software such as Discord. We had initial fixed objectives, but we helped each other in code writing to speed up the process during later stages of development. The following list contains the features each one of us mainly worked on.

- **David Grieco**
  Car Physics
  Tire Physics
  Tire Sprite Asset Creation and Animation
  Code Maintenance and Refactoring
  Performance Optimization
  Class Diagram
  Camera Management

- **Toto Christensen**
  Sound Implementation
  Visual Asset Creation
  Code Maintenance and Refactoring
  Checkpoint System
  Bézier Curve-, Track-, Track Barrier-, and Track Mesh Creation
  Track Creation Editor
  Scoreboard Functionality
  Saving and Loading to/from JSON
  Performance Optimization
  Class Diagram
  Camera Management

3.8 Tools

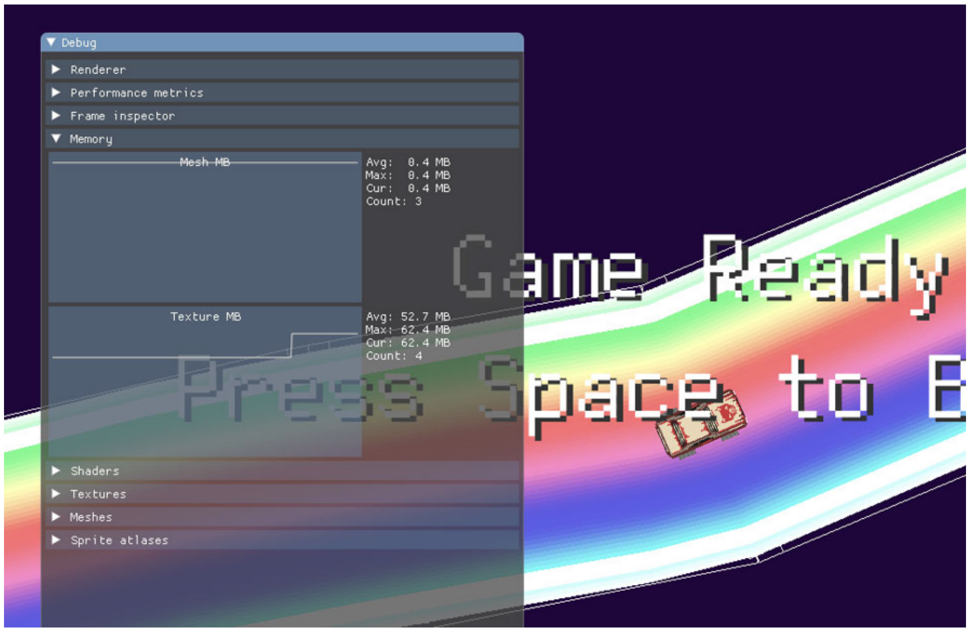
We used the CMake software (*CMake*) to build and generate project solutions. As its creators' website states: "*CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner*". As a starting platform, we utilized the scripts found in the Bird game project of the course (Exercise 8) and modified it to fit our needs. We further included the png and JSON files relative to the sprite sheet of our game. To get the latter, we used TexturePacker and added all of the needed PNGs to a single sprite sheet that we can then access in the RaceGame class (*TexturePacker*). We incorporated RapidJSON (*RapidJSON*) to quickly and easily save to and load our tracks and scoreboard in JSON. Since we worked on Windows, we used Microsoft Visual Studio (*Visual Studio*) for code editing, and we made CMake build Visual Studio solutions. Finally, we used Git to work simultaneously on different game features and share our versions (*GitHub*). Moreover, Git let us save each version we finished during our development process so that we always had the chance to return to a more stable state of the game whenever some new feature broke the gameplay flow.

4 Performance & Bottlenecks

During the development process' lifetime, we discovered several performance issues regarding memory management and framerate, as well as difficulties in generating smooth turns and mesh in our tracks. The following sections explain the problems in detail and go through the solutions we used to fix them.

4.1 Memory Allocation Optimization

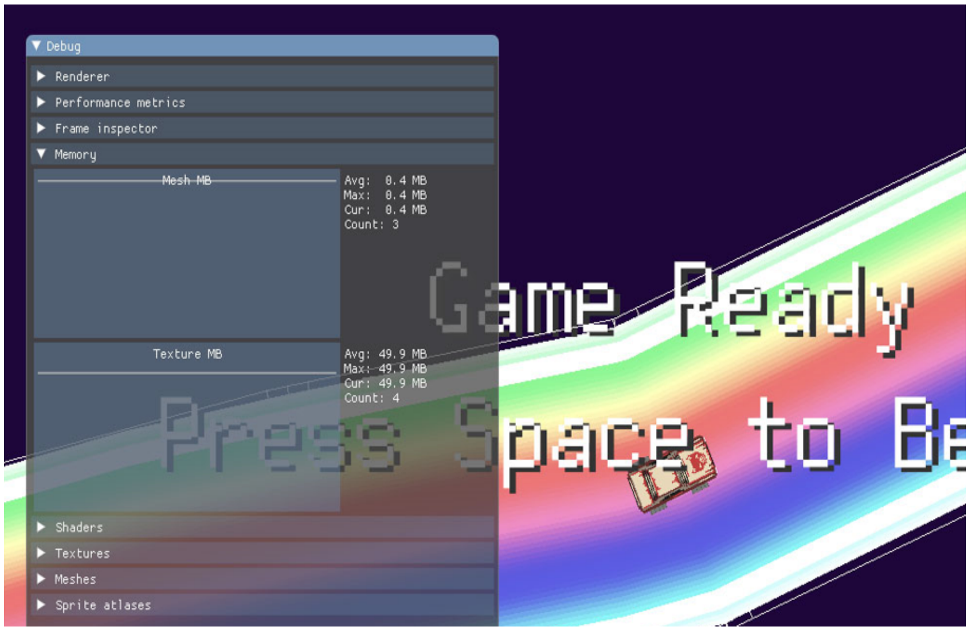
We incurred a critical memory leak issue regarding texture and mesh creation during the development process. Whenever we started/restarted a track, we created a new texture and mesh, resulting in the allocated portion of the memory heap growing at every reset. To solve the problem, we added a check before the texture and mesh creation to prevent more copies from being created when a new level was loaded or reset.



```
Avg:  0.4 MB
Max:  0.4 MB
Cur:  0.4 MB
Count: 3
```

```
Avg: 52.7 MB
Max: 62.4 MB
Cur: 62.4 MB
Count: 4
```

Fig. 3. Memory leak when loading a Track.



```
Avg:  0.4 MB
Max:  0.4 MB
Cur:  0.4 MB
Count: 3
```

```
Avg: 49.9 MB
Max: 49.9 MB
Cur: 49.9 MB
Count: 4
```

Fig. 4. Loading a Track after the memory leak fix.

We also found out that the cleanup function for the Material component, which was already present in the SRE assets, was called at each update cycle without apparent usefulness.

Since this was of no detriment to our project, we left it untouched. This is further true due to time constraints and us being unsure if this was its intended functionality.

4.2 Removed Sound Related Lag

When first including sound effects in the game, we encountered an issue concerning framerate spikes in our game. Every time a sound would play, the frame rate would drop simultaneously, with the memory allocation spiking up. This was due to the initial sound library and its lack of sound pre-loading features. When we played our sounds, this library would just load the sound in the memory as if it was the first time playing it. We switched to the Simple DirectMedia Layer library to solve this issue, which is able to pre-load the sound in memory once and then reuse it without reallocating memory, among many other valuable features (*SDL*).

4.3 Track Curvature

To utilize as few points and physics components as possible for our collision, we have constructed a function to calculate the curvature of our track and construct only a single track barrier where the curvature does not exceed what is allowed. The procedure enables us to change how rigid the track boundaries should be and ensures that the minimum number of collision objects is drawn (*fig. 5.*) while still showing a crisp track mesh. We would be able to further optimize this concept by utilizing *ChainShape* of Box2D (*ChainShape*); this would decrease the number of points drawn quadratically and barrier objects to two. As touched upon in the 'Future Work' chapter, we were unable to do this.

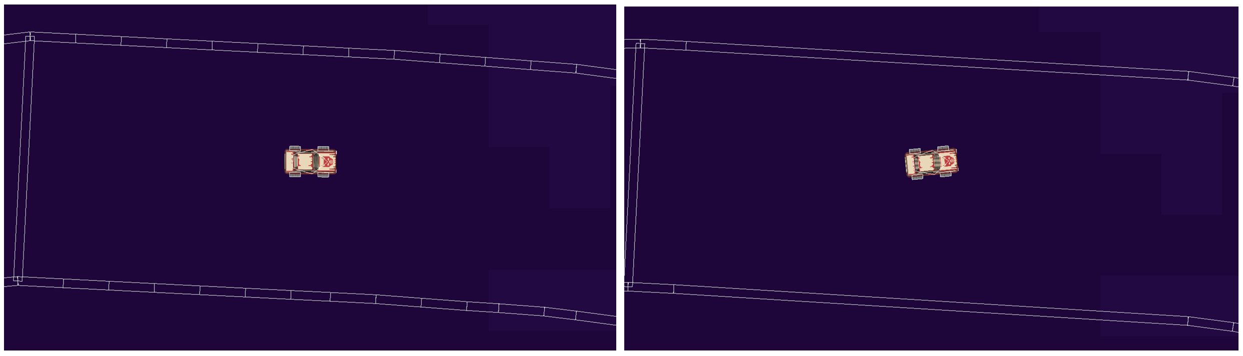


Fig. 5. Left; barriers without curvature checking. Right; curvature checking enabled.

4.4 Curves and Mesh

While generating our track and the resulting mesh did not result in performance issues, it is nonetheless essential to touch upon one of the more significant difficulties we faced during its' implementation - namely generating an even mesh. The problem occurred due to how we generate our bézier curves which are described as a multitude of points, depending on the curvature of the curve. This meant that some curves would have twice the amount of points compared to others, creating a much smoother mesh. We resolve this by finding the smallest distance between points on a given track and interpolating every section to the same degree. In cooperation with our curvature check function, this solution gave us the ability to create a smooth track mesh while keeping barriers as rigid as needed.

5 Discussion

We have largely achieved the primary goals we set at the start of the design process, even though we have had to limit some features and cut others out. The bézier curves-based track generation works as intended, with some caveats, and can extend the game's replayability by creating new tracks. The car physics feel realistic and demanding, making the player involved in the gameplay while still delivering a fun experience. The leaderboard system of the game provides the player with new challenges the more they play it, stretching the game time even further. The project can, of course, be improved upon and is by no means complete, but we feel very optimistic and proud of what we were able to achieve in the time available to us. The next chapter will discuss the main features we cut out and their characteristics.

6 Future Work

Due to a shortage of development time, we had to cut out some of our initial ideas for the project. That said, we started thinking of ways to implement some of them, and we will use this section to describe some of those features and their hypothetical implementation.

6.1 Enemies

The first draft of our design for the game included enemies to race against. Unfortunately, we could only make the car physics work as intended days before the hand-in date, so applying those mechanics to a whole new entity would have been far too time-consuming. The basic idea for the enemy AI was to make them follow points along the track (precisely a random point along a checkpoint) so that the player could feel challenged and forced to improve their driving, not to be beaten by the enemies. Another idea was to have enemies chasing the player to make them explode by driving head-first into them while trying to complete the track. This could have been done by deactivating the Box2D joints of the wheel to simulate a car explosion, but we were still missing the AI movement script.

6.2 Items

Another idea we had to spice gameplay up was to spawn Items on the track randomly. These Items would have behaved similarly to the ones in *Mario Kart* and would have given the player a random Power-Up that, on activation, would have modified some of the Car/Tire attributes (Super *Mario Kart*).

6.3 Obstacles

During the final stages of development, we thought about adding obstacles to the tracks. Those obstacles would not have had static bodies but dynamic ones so that the player could have had different interactions when driving into them. This had the purpose of making the gameplay feel more challenging for the player. The implementation for creating these obstacles would have been straightforward because we could have used the simple shapes that the PhysicsComponent already provided. Still, the problem lay in saving these obstacles in the loaded JSON of the track. It would not have been a difficult feat but a time-consuming one, so we decided to scrap it altogether.

6.4 Chains

As touched upon previously, while creating collisions for the track barriers, we tried to use the logically correct *ChainShape* of Box2D. Still, we could not make it work properly with the procedural generation of the track. We fixed it by creating Box shapes on the curve connecting each point forming the tracks' barriers. This solution works from a gameplay point of view. Incorporating *ChainShape* would cut occupied memory space by a significant amount considering that with our solution, each line of the barriers becomes a 4-sided box of which only one side is used.

6.5 Multiplayer

The first draft also included online multiplayer, but we scrapped it in the first stages of development. It would have taken too much time away from more exciting features for the project's purposes. The idea was to use Sockets to create a Client/Server connection and transfer all the physics logic on the Server with the Client only interacting through input data and receiving the new state as output data from the Server.

Ludography

Super Mario Kart (1992). *Nintendo Co. Ltd*

References

Nobel, *Simple Render Engine*, 2016,

https://github.com/mortennobel/SimpleRenderEngine.

Box2D, *Box2D Documentation*, 3 Aug 2009,

https://box2d.org/.

GLM, *OpenGL Mathematics*, 26 Jul. 2015,

https://glm.g-truc.net/0.9.9/index.html.

iforce2d, *Top-down car physics*, 19 Mar. 2014,

https://www.iforce2d.net/b2dtut/top-down-car.

Dynamics Module, *Box2D Dynamics Module*, 3 Aug 2009,

https://box2d.org/documentation/md__d_1__git_hub_box2d_docs_dynamics.html.

CMake, C*Make Overview,* 2000,

https://cmake.org/overview/.

TexturePacker, *TexturePacker Features Page,* 19 Sep. 2010,

https://www.codeandweb.com/texturepacker.

RapidJSON, *RapidJSON Documentation,* 2015,

https://rapidjson.org/

Visual Studio, *Microsoft Visual Studio,* 1997,

https://visualstudio.microsoft.com.

GitHub, *GitHub Main Page,* 19 Oct. 2007,

https://github.com/github.

SDL, *Simple DirectMedia Layer Homepage,*1998,

https://www.libsdl.org/.

ChainShape, *Box2D Documentation,* 3 Aug 2009,

https://box2d.org/documentation/md__d_1__git_hub_box2d_docs_collision.html#autotoc_md40.