Abstract

This aim of this project was to create a sandbox racing game in where the player can create their own car to drive on a track generated on top of a spline.

A track tool is provided to give designers an easy way to create the tracks and a car editor has been created to give the players the ability to easily create their cars.

Table of Contents

1. Progress Report

1.1. Introduction

1.1.1 Background

Splox Racers is a 3D sandbox racer created as the major project of the Southampton Solent University. The tracks will be generated on splines. Those splines can be created by the developer or a random generator and can be saved to the disk to load at a different time. The AI uses the splines as a guideline to figure out, how to drive along the track. Different points on the spline have different attributes to make different areas, like boosters or obstacles, on the track available. Each point also has attributes to control the tracks banking, thickness, width and whether or not it has guard rails available.

The players can build their own cars via a sandbox editor. There will be different block shapes available to create the chassis in any form. Blocks can be coloured to further improve the look of the car. Special blocks will be needed to change the cars behaviour, like speed, handling and so forth.
The editor makes sure, that the car has all necessary parts, exactly one seat, wheels and an engine.
Players can test their cars in a special track or a chosen/generated track to help fine-tune the cars behaviour.
If they are happy with the car, they can name it, save it and choose to drive it in races.

As of my research, no one ever combined a random spline track generator and a vehicle builder into a racing game.

1.1.2 Challenges

There will be quite a lot of challenges to overcome.

The mesh of the track has to be generated, so there are no holes or other artefacts in it and all the textures line up without any seems.

The random generator has to generate valid tracks by making sure the spline does not overlap itself and keeps enough distance, so that cars will not be blocked by the track. Loops have to be placed in a way that the players can build up enough speed to be able to make it through the loop. Obstacles have to be placed in a way that makes them fun i.e. make sure the distance between obstacles is not too low, as that will make the track boring, and not too high, as that will make the track too frustrating.

The car editor has to be easy to use and make sure the build cars are always valid. Therefore, it has to check for disconnected blocks and indicate them to the player. It has to make sure the player places all the mandatory parts and constraint the placement of blocks to keep the cars size to a certain maximum size.

Another challenge will be for the AI to drive along the spline in an intelligent way. Cars controlled by the AI have to be able to avoid obstacles.
They also need to be able to drive cars created by users without knowing how the car behaves in terms of acceleration, max speed and handling.

1.1.3 Aims & Objectives

The main aims are:

- Implementing a sandbox racing game
- Create a racing track on top of splines

The main objectives are:

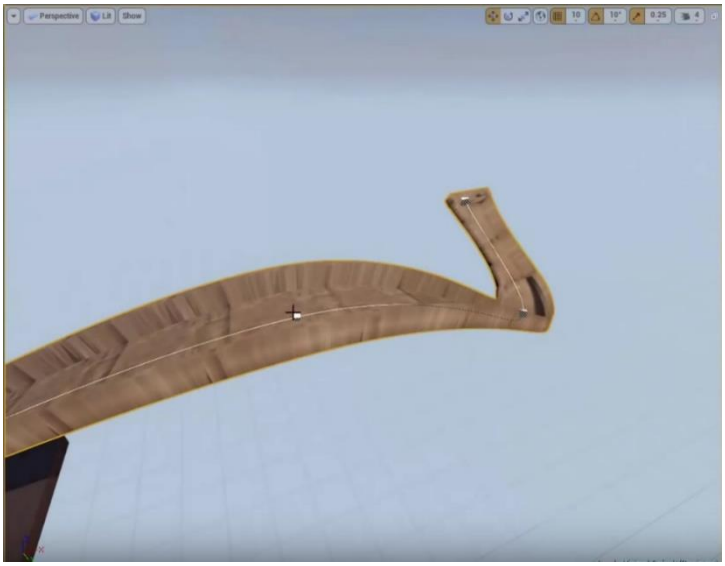
- Create car editor
- Create splines
- Generate track on splines

- Combine previous objectives into one game
- Implement racing game rules

1.2. Research

The YouTuber DokipenTech [1] created an unreal engine 4 tool to create a road from a spline as shown in figure 1. He made four video tutorials about how he has done it using blueprints.

In addition, an employee of Epic made a live training [2], which is still available on

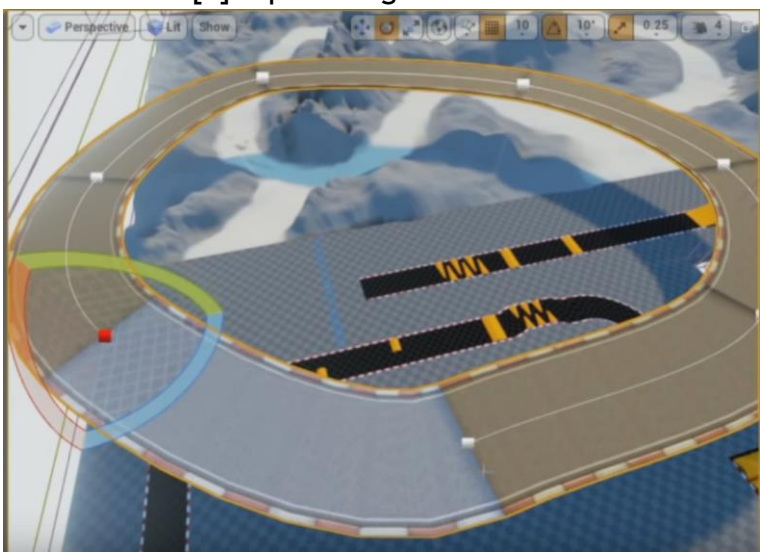


YouTube. In that training video they show a sophisticated way to create meshes on splines, also using unreal engine 4 blueprints. See figure 2 for a preview of his work.

Figure 1: Road created on a spline with the unreal engine 4.

Figure 2: Spline road as created in a live training from Epic.

A sandbox vehicle editor has been done by several developers. Freejam LTD uses it in the game Robocraft [3] as seen in figure 3. Other games like Autocraft developed by Alientrap Games [4] also have the ability to create vehicles in a sandbox mode (see figure 4), though Autocraft does not use an external editor, just a mode switch in the current loaded level.
Another game that allows the player to create vehicles using blocks is Space Engineers by KEEN SWH LTD [5]. Space Engineers also does not switch into a



separate editor and does not switch to a different mode to build as seen in figure 5.

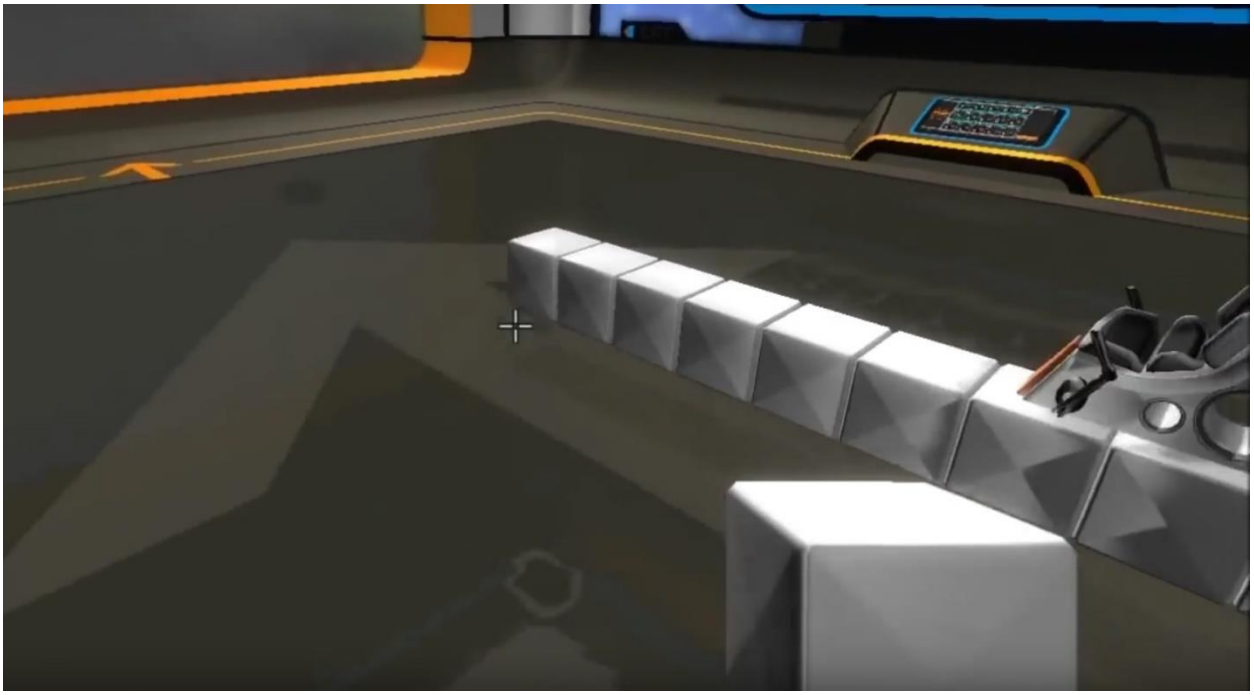


Figure 3: Robot builder used in the game Robocraft.

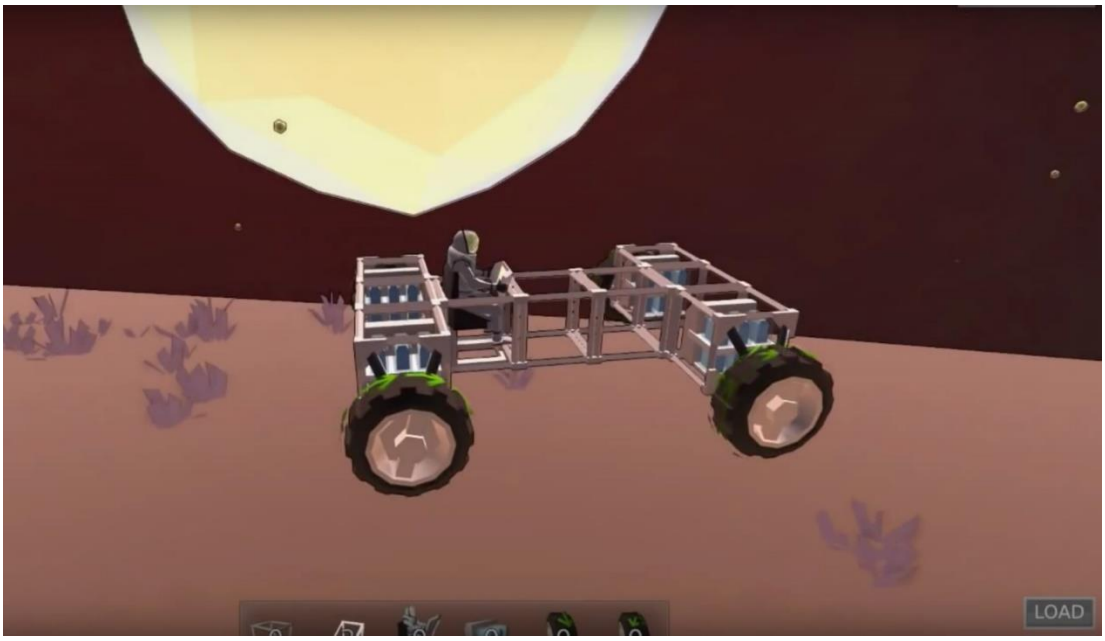


Figure 4: Building mode in the game Autocraft.

Figure 5: Building a car in Space Engineers.

I could not find any implementation details of their editors. Therefore all information I have are from images and trying them out to get a feel of their functionality and controls.

1.3. Solutions

I will use the Unreal Engine 4 to implement the game, as it already provides helpful tools to expand upon. It already has built-in spline components and a sophisticated car controller to use as a base for the project.

To give players the ability to create their own cars, I will provide them with a separate editor mode as it is done in Robocraft. This allows me to constraint the cars size and make sure it is valid before it will be used to race. A separate editor makes also makes it easier to come back later and edit previously created cars. There will be a virtual 3D-grid to place the blocks on and the editor will provide the player with a starting block. The player will only be able to place blocks attached on any other block and not free floating in the air. If they remove a block from the car, all blocks neighbouring the removed one will check if they still have a connection to the seat block. I will implement the check with a 3D version of the A* algorithm [6], which will find a way to the seat block if there is one. If there is not a route, it will notify the player by tinting the block red.

To create the spline roads, I will follow the tutorials mentioned before and extend upon it. The tutorials only show the basics to make it work, but there is more work needed to make it look “good” and to make sure that curves cannot be too steep. Furthermore, I decided to switch from blueprints to C++ as the extensions to the road generator will get very complex very fast if done in blueprints.

The AI will at first just use the spline as a path to drive on and will calculate the optimal speed by the steepness of curved, which it will detect through the angle of the tangent of a spline node point. They will drive along the spline using the steering algorithm, which also makes sure they avoid collisions.
Later in the development I plan to improve the AI so that it calculates a better

path and better detect the optimal speeds also taking into consideration the cars specs.

1.4. Project Management

1.4.1 Task breakdown

I created tasks for the different parts of the game (see table 1). Everything to do with the track generator gets an ID starting with TG, car editor tasks have an ID starting with CE, the AI tasks begin with AI, asset tasks with AS and lastly the tasks which combine all the different parts into a game got IDs starting with GA. In addition, I gave each task a priority to indicate whether a feature is necessary (must have), will improve the game further (should have) or are entirely optional and might not be implemented due to the time needed to implement the game. In the task table, I also provided dependency to show which tasks have to be implemented in order to implement a certain other task. This ensures a smooth workflow, as I will not start tasks that cannot be completed at that point of work. Lastly, I provided a simple description of the task and what its implementation is supposed to achieve.

ID	Priority	Dependency	Description
TG001	Must Have		Generate the road mesh along the spline
TG002	Must Have	TG001	Apply road texture to road mesh
TG003	Optional	TG001	Make looping's possible
TG011	Must Have	TG001	Add attributes to spline points to control width, thickness, bank and element type
TG012	Must Have	TG011	Add a start line onto the track
TG013	Must Have	TG011	Add goal line onto the track

TG014	Should Have	TG011	On closed loop make start to also be goal
TG015	Should Have	TG011	Add booster to speed up car when they are on it
Completed up to here			
TG101	Optional	TG001	Randomly generate spline
TG102	Optional	TG101	Randomly change spline point attributes
CE001	Must Have		Level to build car in
CE002	Must Have	CE001	UI for editor
CE003	Must Have	CE001	Place start block which can't be removed
CE004	Must Have	CE001	Let the player place blocks on other blocks
CE005	Must Have	CE004	Let the player delete placed blocks
CE006	Must Have	CE005	Check for car validity
CE011	Must Have	CE001	Let player select block types from menu
CE012	Should Have	CE011	Add different weight chassis blocks
CE013	Must Have	CE011	Add wheels
CE014	Should Have	CE011	Add engines
CE015	Optional	CE011	Add fuel tanks
CE021	Should Have	CE011	Add colour picker
CE022	Should Have	CE021	Colour chassis blocks

CE031	Must Have	CE002	Allow naming of car
CE032	Must Have	CE006	Allow saving of car to a file
CE033	Must Have	CE032	Allow loading of car from a file
CE041	Optional	CE033	Test drive car
CE051	Optional	CE032	Bake car data to use less colliders and meshes
GA001	Must Have	TG;CE	Main Menu
GA002	Must Have	GA001	Car selection
GA003	Must Have	GA001	Level selection
GA011	Must Have	GA002;GA003	Load Level
GA012	Must Have	GA011	Spawn player
GA013	Must Have	GA012	Controls
GA014	Should Have	GA011;AI	Spawn AI
GA015	Must Have	GA013	Win condition
GA021	Must Have	GA011	UI
GA022	Must Have	GA021	Show speed
GA023	Should Have	GA021	Show lap/laps
GA024	Should Have	GA021;GA012;GA014	Show place
AI001	Should Have	GA	AI cars drive along spline
AI002	Optional	AI001	Avoid collisions with other cars
DC001	Must Have		Create task list
DC002	Must Have		Project proposal

DC003	Must Have	DC001;DC002	Progress report
DC011	Must Have	DC003;GA	Final report
AS001	Must Have		Road mesh and texture
AS002	Must Have	AS001	Start mesh and texture
AS003	Should Have	AS002	Goal mesh and texture
AS011	Must Have		Chassis block mesh and texture
AS012	Must Have		Wheel mesh and texture
AS013	Should Have		Engine block mesh and texture
AS014	Should Have		Fuel tank mesh and texture
AS021	Must Have		UI textures

Table 1: Task list with IDs, priority, task dependencies and descriptions.

1.4.2 Time Assessment

I estimated the time each task will take to be implement. For that, I created another table that shows the tasks category, ID and time estimate. The very last row is the sum of the time each task will need to be completed. The overall time estimate for all tasks amounts to **411** hours of work time, presumed there will not be any major problems or difficult to find errors, who increase the tasks time much.

Track Generator		**Car Editor**		**Game**		**AI**		**Assets**		**Documents**	
Task	**Time**	**Task**	**Time**	**Task**	**Time**	**Task**	**Time**	**Task**	**Time**	**Task**	**Time**
TG001	30,0 h	CE001	15,0 h	GA001	4,0 h	AI001	8,0 h	AS001	5,0 h	DC001	4,0 h
TG002	5,0 h	CE002	15,0 h	GA002	4,0 h	AI002	10,0 h	AS002	2,0 h	DC002	6,0 h
TG003	8,0 h	CE003	6,0 h	GA003	4,0 h			AS003	1,0 h	DC003	8,0 h

TG011	20,0 h	CE004	8,0 h	GA011	2,0 h			AS011	2,0 h	DC011	12,0 h
TG012	15,0 h	CE005	3,0 h	GA012	1,0 h			AS012	6,0 h		
TG013	10,0 h	CE006	20,0 h	GA013	3,0 h			AS013	3,0 h		
TG014	6,0 h	CE011	7,0 h	GA014	2,0 h			AS014	2,0 h		
TG015	6,0 h	CE012	2,0 h	GA015	1,0 h			AS021	6,0 h		
TG101	30,0 h	CE013	6,0 h	GA021	3,0 h						
TG102	10,0 h	CE014	2,0 h	GA022	1,0 h						
		CE015	2,0 h	GA023	1,0 h						
		CE021	16,0 h	GA024	2,0 h						
		CE022	3,0 h								
		CE031	1,0 h								
		CE032	8,0 h								
		CE033	8,0 h								
		CE041	10,0 h								
		CE051	36,0 h								
	140,0 h		168,0 h		28,0 h		18,0 h		27,0 h		30,0 h

Table 2: Time estimation to complete every individual task.

Since the car editor is the biggest feature of the game, it will also take the most time to complete. CE001 and CE002 have gotten a bigger amount of time than most tasks in the car editor category, since they provide the basic for the entire car editor and everything builds up on those. CE006, which is the car validity checker, has also gotten a large amount of time allocated, since it has to be very efficient as it might get executed several times per frame and should not increase the frame time by a noticeable amount. To implement and optimize the algorithm

to the desired degree will take a long time, which is why the allocated time is set to 20 hours.
The largest time allocated to the car editor is the mesh and collider baking of the car, which will decrease computation and render times in the races considerably. It is also set to optional, since it might not be implemented in the time I have gotten to complete the game. It will implemented last, if there is time left, or after the project submission.

The second biggest category is the track generator. Again the first task has the longest time allocated to it, since it will provide the basis for each other task to build upon. TG101 and TG102 are also optional tasks, which might not be implemented, since they will need a lot of work and fine-tuning, but are not necessary for the game to function properly.

The basic implementation of the AI will probably not take longer than 18 hours, since they will only drive along the spline, which is already given and try to avoid other cars, which should not be much of a problem using the steering algorithm.

Assets will mostly be gathered from free sources or created by myself; they will be plain looking assets, as they are not necessary for the games functionality. They will be improved after the project submission.

The Game category will consist of the combination of all other categories and also provide menus, win conditions, detection of the games states and so on. Combining the previous parts should go rather quickly, since they already have everything necessary to load the correct data.

1.4.3 Risk Analysis

To prevent the loss of the project due to technical errors, I am using version control. I am using GitHub as a version control service (See Appendix A for the

link). This allows me to keep a copy on a different server and it allows me to revert changes to the code that may cause errors in the game.

To make sure, the game will reach a certain point of completion in that it can actually be player, I will complete the task according to their priorities and dependencies. Task with the priority of optional will be done as the very last, since they are high risk and do not provide enough return to do them earlier.
The “Must Have”-priority tasks are the ones to provide a huge return and should therefore be completed at first.

Code optimization and AI improvements are not listed in the task list, since they will only be done if there is time left, after all other tasks have been completed, since especially the optimization may break a lot of code and is therefore very high risk.
Changes to the AI will change the entire game balance and experience and should be thoroughly tested, which will cost a lot of time.

1.5 References

[1] *Unreal engine 4 road tool in blueprints with Spline and SplineMesh - part 1 Spline components*, 2014 . YouTube

[2] *Using Splines & Spline components | live training | unreal engine*, 2014 . YouTube

[3] MIKECODEJAMMER and FREEJAM LTD, 2017. *ROBOCRAFT* [viewed 3 March 2017]. Available from: http://robocraftgame.com/

[4] ALIENTRAP GAMES, n.d. *Autocraft - early alpha access* [viewed 3 March 2017]. Available from: http://www.autocraftgame.com/

[5] KEEN SWH LTD, 2013. *Space engineers* [viewed 3 March 2017]. Available from: http://www.spaceengineersgame.com/

[6] Cui, X. and Shi, H., 2011. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, *11*(1), pp.125-130.

Appendix A

Link to the GitHub repository: https://github.com/NeoIzen/SploxRacers

2. Implementation

2.1 Project Progression

I have been working on the project in an agile methodology. I broke the project down into small tasks, which are listed in the section 1.4.1. Still I didn't plan every single step, so that I can still be flexible, therefore I reacted to changes as new problems have been occurring. I have also used the prototyping approach to create and test small systems of my project without having to change the main project. This was helpful in the case where a prototype was not successful. The unused prototypes did therefore not influence the main project. Prototyping was also helpful to create systems that can work on their own as separate modules. I created a prototype for the track tool and the custom vehicle implementation (See Appendix A).

The project has been developed on a notebook with 16GB DDR3 Memory, an Intel Core i7 3630QM and an NVidia GeForce GTX 675MX using the Unreal Engine 4.14 and Visual Studio 2015. Though at a later point of the development process, the engine version has been updated to 4.15 and visual studio has been updated to version 2017.

2.1.1 Track tool

I started the project implementation with the track tool. The track tool allows the level designer to create a spline and adjust the different spline point positions and orientations. The default spline implementation of the unreal engine 4 (EPIC GAMES, 2017) also allows the designer to adjust the tangents of each spline point. The road tool uses the spline point configuration to generate road elements between two spline points. The meshes on top of the spline are generated in real-time while the designer is tweaking the spline. This enables the designer to completely create the track in the editor without having to switch to another view or to have to start the

game to see results. Each spline point also has additional attributes that can be adjusted as seen in Figure 1.

Figure 1: Attributes of the spline point of the current spline. With spline point 0 and 1 expanded.

In the current version of the track tool it is possible to enable and disable the guard rails for the track element starting on the configured spline point. It is also possible to adjust the roads banking value, width and thickness at the configured spline point. Banking, track width and the track thickness affect both track elements touching the spline point. The values then get interpolated through the track element. There is no automatic checking whether the values makes sense, so that the designer has full flexibility in creating the race track. Lastly the designer has to choose the type of the track element currently available are ‘start’, which adds a start line to the track, ‘road’ which is a default road piece and ‘speed’, which adds speed arrows on the ground and adds force to the car driving on it to speed it up. Some values are locked to make sure the track will be valid. Therefore the first

spline point gets locked to be the ‘start’ type with banking, width and thickness locked to their default values, which are 0 for banking and 1 for width and thickness. The start and speed elements are not fully implemented yet, since I didn’t value them high enough to prioritize them over other tasks. To further complete the track tool there should also be an ‘end’ element and ‘checkpoint’ elements. To help the designer build and configure the track. Spline point numbers can be enabled or disabled as seen in Figure 2 and 3. This is especially helpful for large tracks with many spline points, as it prevents the designer from having to count through them or try values until the correct spline point is found. Instead they can immediately figure out which spline point index belong to which spline point in the world.

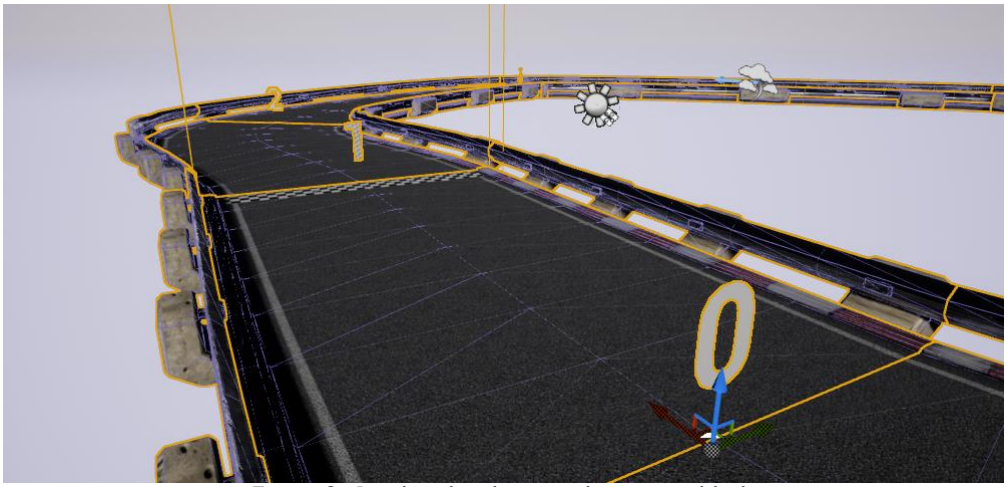


Figure 2: Road with spline numbering enabled.

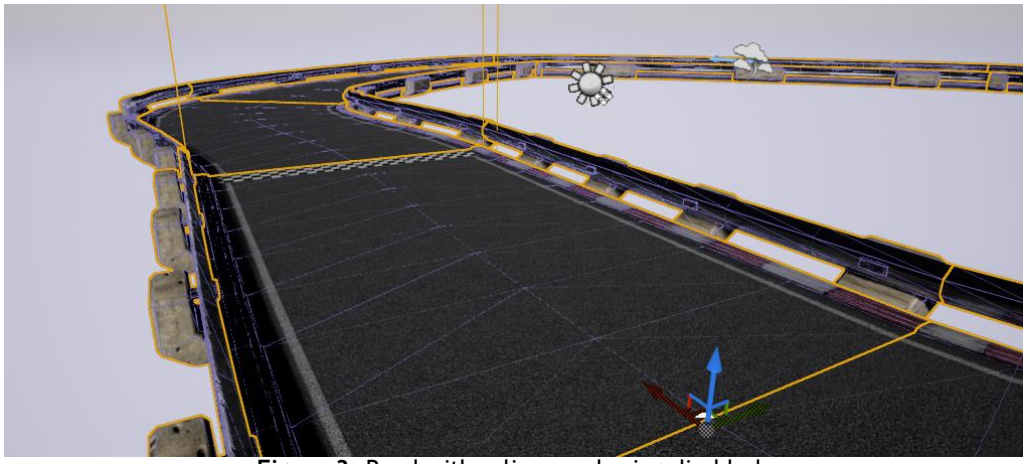


Figure 3: Road with spline numbering disabled.

I created a very basic implementation of the track tool as a prototype using blueprints. For the implementation inside the main project I decided to switch to c++, because the road tool has quite a lot of functionality and blueprints can get unorganized and unreadable very fast. Also since the road updates in real-time, I wanted it to have the best performance as possible, so changes can be seen immediately and the designer does not have to wait every time a value gets tweaked. That is why I switched to c++, since it is around ten times faster than blueprints are (Irascible, 2014).

I fully converted the track tool prototype from blueprints to c++, just to find out, that asset loading should not be done in c++ code. Asset loading does work in c++, but it has some complications when assets get renamed, moved or deleted. When an asset gets renamed or moved, the editor will create what is called a redirector so that every class that tries to access an asset via the old reference gets redirected to the new one (EPIC GAMES, 2017). Every once in a while those redirectors should get fixed up (EPIC GAMES, 2017). The fix up process will adjust all blueprints that use the reference and then delete the redirector, but it can't do it with c++ classes which have the path hardcoded as a string. After learning that, I created only base classes in c++ to provide the functionality and created a blueprint derived from

these to handle asset loading. Since blueprint classes can't be directly referenced from c++ I created a track element library class, which maps an ID to a class. By creating a blueprint from the library class, I was able to fill out the map with IDs and the associated blueprint classes. With that map, the c++ code is able to instantiate and use the blueprint object through the c++ interface, since the blueprint overload c++ classes and therefore c++ can use them using polymorphism.

2.1.2 Car Editor

The next system I started working on was the car editor. The car editor always starts off with a seat in the middle of the car, as shown in Figure 4.

Figure 4: Car editor with seat already placed.

The player can then start placing other blocks attached to the seat. Different blocks can be placed on either side of another already placed block to create the desired look of the car. There is a library with all available blocks to choose from and create the car. To open the library, there is a button on the top right corner of the screen. The library is separated in different categories to organize the blocks available and make it easy to find the block the player is looking for. After they selected a block,

they can also set the blocks colour with the colour sliders on the upper left corner of the screen. The colour sliders start off hidden, but can be shown by clicking the colour-wheel button on the upper left corner of the screen. After the player is done creating the car, they can click on the save button in the lower left corner of the screen and give it a name to save it to a file. It is also possible to load a previously created car by clicking on the load button and enter the cars name. The entire GUI can be seen expanded in Figure 5.

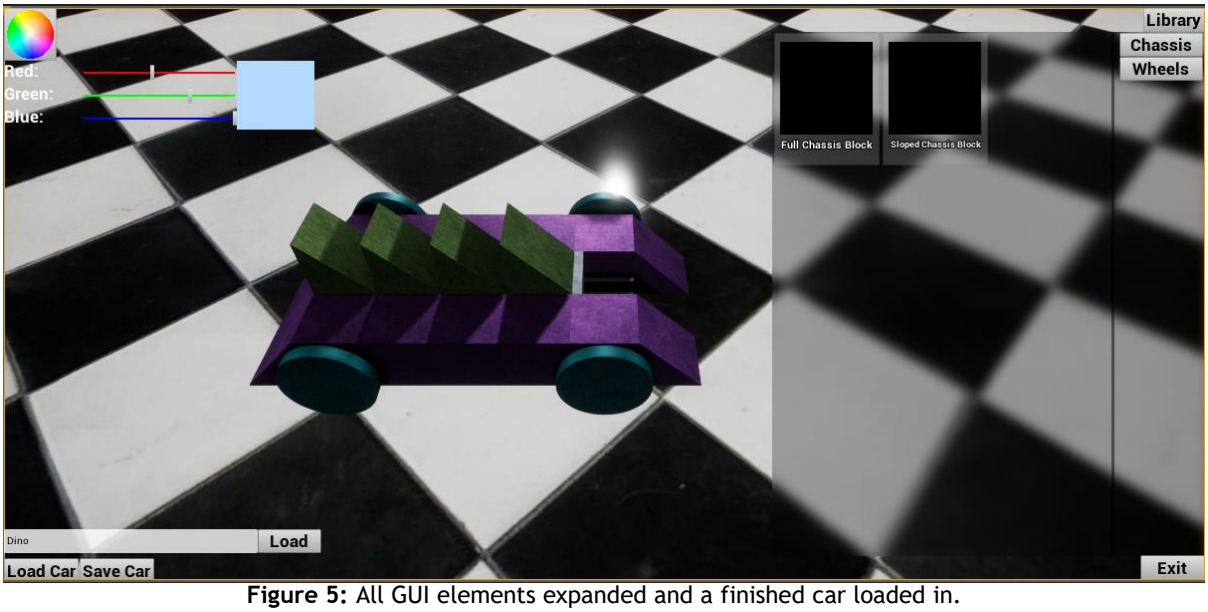


Figure 5: All GUI elements expanded and a finished car loaded in.

To build a fully functioning car, all you need in the current version of the game is the seat, which is already placed, some chassis blocks and enough wheels to give the desired balance to the car. Different blocks have different weights, which can change the cars behaviour, performance, balance and handling drastically. Additional tasks to improve the car editor would be a block for the engine to give more options to create an individual car and to tweak the behaviour and performance of the car. Wheels should get the option to decide whether they can be used to steer, to what degree they can be steered and if they can be rotated by the engine to make the car accelerate. Also a thruster block would be a good

extension to provide the driver with the ability to give the car a small boost for a short amount of time. The thruster could be configured to focus on strong short boosts or long weak boosts.

To implement the car editor I started by creating the base class for all blocks in c++, which holds all common properties like a name, category it belongs to, weight and an ID to help with saving and loading. Each block has a static mesh, which is assigned by creating a blueprint from the base class or another block class. I decided to derive the base class from the 'StaticMeshComponent' class of the unreal engine 4, since every block has to have a static mesh to be able to be rendered and is a component, which is why it makes sense to derive from the 'StaticMeshComponent' which provides all the functionality to render them and also provides the physic components to make the blocks a functioning physical object in the game world. The blueprint editor allows for the blocks properties to be configured. I decided to go about it in that way, so that a designer can add multiple blocks of one type without having to touch the source code. A designer could easily add new chassis blocks by creating a blueprint from the chassis block base class and then they would only have to set the desired properties, like the mesh, weight, ID and so on. The same holds true for wheels or any other block. The designer only has to derive from the corresponding base class to give a block the desired functionality. This way all blocks can also be stored in the same place in categorized folders in the project, to make them easy organized and editable. Since c++ can't directly access those blueprint classes, I had to create a library like I did for the track tool, to hold all blueprint block classes. By deriving the library class as a blueprint I was able to fill in the blueprint block classes with corresponding IDs, which can then be used in c++ through their c++ base class. The block library gets created automatically through the 'GameState' of the unreal engine 4. The 'GameState' gets created by the 'GameMode' which is always available for a loaded level. The 'GameMode' can be

overridden to change the classes it will create. Through an overloaded version of the ‘GameMode’ I could set my own overloaded ‘GameState’ to get created on the start of the level. I overloaded the default ‘GameState’-base class to create and store the block library, so it will be available for the car editor or the game loader to create and load cars. More information about ‘GameMode’ and ‘GameState’ can be found in the official documentation (Epic Games, 2017).

To store the placed blocks in memory, I created a virtual three dimensional grid. A single grid stores a single car and each player has their own grid. Every time a block is added to the grid, it gets a hash value based on the position of the block to accurately identify a block and to be able to quickly search for a block, given only its position. The class also provides helper methods to snap a block into a grid cell, to make sure they are always placed at valid grid cells and never occupy more than one cell or be offset outside of the cell they are meant to occupy. Since the grid knows about every placed block, it is used to save the car and it can load a file to recreate its values and spawn the blocks which make up the car. To save a car to a file, a special structure is used to only save the data needed to recreate the car and leave out everything that does not need to be saved, like the meshes, as they can be identified through the block ID. The save file gets compressed using the zlib algorithm implementation of the unreal engine 4 (Rama, 2016). The grid would also be able to error check the car, to make sure that no block exists which isn’t directly or indirectly connected to the seat and to make sure every necessary part has been placed. The error check is not implemented in the current version, but will be in the future as it has been an optional task and therefore had a very low priority.

2.1.3 Main Menu

The main menu, which is also the entry point of the game, gives access to all features currently available. It has a button to start the game, which will get you to

a configuration menu to select a track and a car from the lists provided, to start a game. The main menu also has a button to start the car editor. Below that button is a button to open up the options menu, which is not implemented yet, as it has not been planned in the tasks. Configurable options will be necessary before the game gets released, which is why the button already exists. Lastly there is a button to end the game. The main menu can be seen in Figure 6.

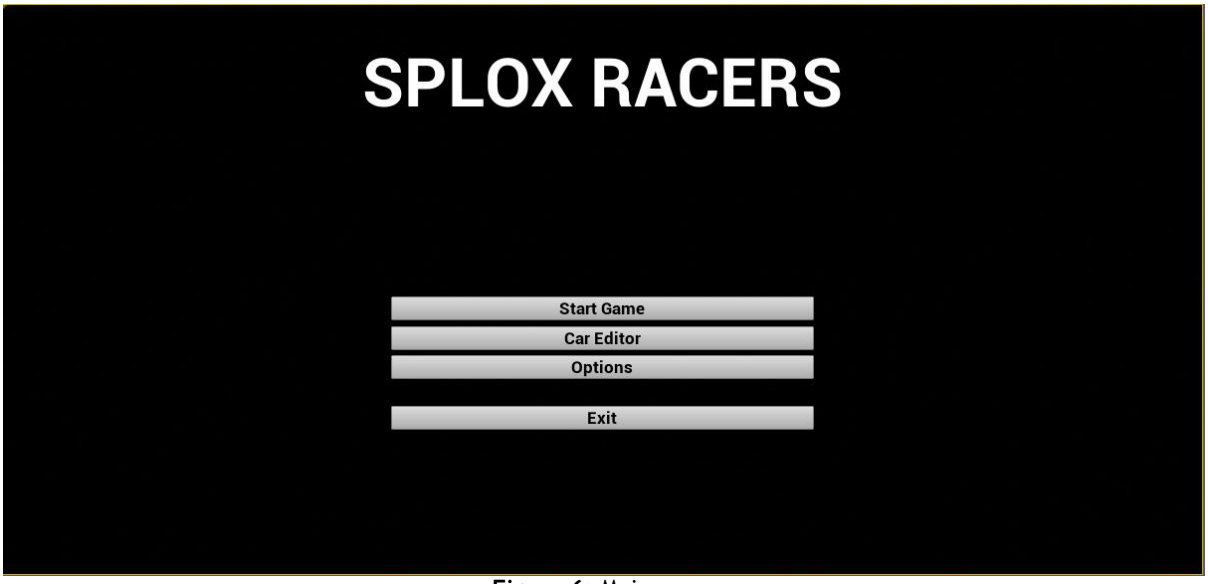


Figure 6: Main menu

The entire main menu has been created using widget blueprints, as they are required to create a GUI using the unreal engine 4 UMG system (EPIC GAMES, 2017). To fill out the level list, I simply created a blueprint structure which lists all tracks I want to make available. The list stores the names of the level files, as the unreal engine 4 only requires a level name to find and load it. Manually setting the levels available allows for debug or test tracks to exist without the players having access to them. The start game menu can be seen in Figure 7.

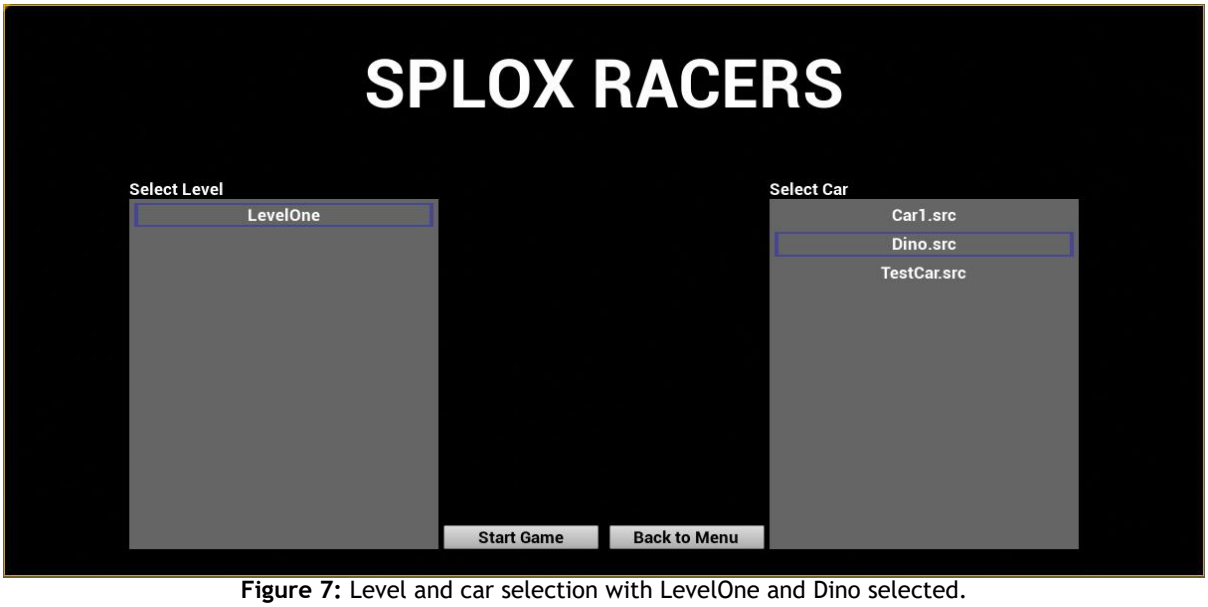


Figure 7: Level and car selection with LevelOne and Dino selected.

The car list gets filled out by reading the contents of the car save folder. To access the file system I had to create a helper method in c++ as the c++ API provides a way to access the operating systems file system (Rama and Zyr, 2016). After selecting a level and a car, the track level gets loaded and the car name gets transferred using the 'GameInstance' class the 'GameInstance' object does not get deleted on a level switch and can thereby be used to carry data between levels (Rama, 2016). For future releases it will also be used to provide other players data and the number and strength of AI players to the level.

2.1.4 Game

For the actual game, all previously described modules have to be functioning. The game gets started from the main menu. After the level has been loaded, it goes into its initialization phase in where it reads out the data provided by the 'GameInstance' class. It then spawns a pawn for every player and tells each players grid to load a car with the specified name. The tracks spline provides possible spawn locations on where the cars get placed. After all cars have been initialized, game variables will

be initialized and lastly control will be given to the players for their own cars. The game can then start by counting down and allowing every player to drive. Depending on whether the spline is a closed loop or open ended, the race would be a point A to point B or a lap based race. Each human player has a GUI to display their speed, position and current lap. As in most racing game, the player who first crosses the finish line or completes all laps, wins.

I started by providing the data for the first player only, but I prepared the initialization blueprint for more players, by using a 'foreach' loop. The loop only goes through the one player, since more data will not currently be provided. My next step was to load the car and create the vehicle movement component of the unreal engine. Unfortunately the provided component does not work with the system this project uses. The vehicle movement component provided by the unreal engine 4 uses a skeletal mesh to place virtual wheels on bones that have been placed where the wheels are. It then uses those bones for animation data and to drive the car. For more information on the vehicle movement component see the official documentation (EPIC GAMES, 2017). Since the cars in this project get build up with multiple static meshes, rather with a single skeletal mesh, the given component does not work, therefore the animation, movement, physics calculations and engine behaviour have to be implemented by myself. For the replacement car behaviour system, I started working on a prototype (See Appendix A) and got a very basic car functioning. Unfortunately it does not work in the game yet and therefore neither the UI nor game rules have been implemented yet.

2.2 Changes to the Schedule

I followed the schedule I made as much as I was able to. The schedule can be seen in sections 1.4.1 and 1.4.2. At some point in the development process, tasks took longer to complete as planned or a prototype had to be created outside of the

schedule. There have been some tasks, that I didn't think of before and didn't know have been necessary respectively. Also some glitches did cost me a lot of time, which again put the schedule behind.

Sometimes I had to redo a certain task. The first time I did tasks more than once and had to create additional tasks was while I was implementing the track tool. I started a prototype of the track tool using blueprints and switched it to c++, as mentioned in the track tool section 2.1.1 of this document. The switch to c++ created new tasks as I had to create a track element library to properly assign assets to the track elements and make them available to c++ code to give designers the ability to create and manage track parts.

One glitch that had cost me a lot of time was triggered by the garbage collector of the unreal engine 4. The grid object that holds the car data got randomly deleted, creating a bug in where the block to place did not get rendered on the position it should have been, since the variables to calculate the blocks position based on the grid snap had been invalidated. After trying to interact with the grid, by placing or removing a block, the game crashed, since it tried to access memory which was not longer available. It took me a long time and lots of testing and research to figure out what caused the bug and how to fix it.

Another case where I had to create many more tasks was when I figured out, that the vehicle movement component provided by the unreal engine 4, is not suitable for this game. I had to find a way to create my own vehicle system, had to create a prototype for it and do more research. I ended up with more tasks to create physics constraints, manage the welding of the static mesh physics bodies and create the behaviours to drive according to the users input (UnrealTek, 2016).

2.3 Things Learned From Milestones

Working on all the systems of this project gave me a lot of insight on several programming techniques and the unreal engine 4 in particular.

2.3.1 Track Tool

Working on the track tool I learned about the spline system of the unreal engine 4 and about asset loading. I learned that, even though it is possible, it is not recommended to load assets via c++ code. When assets get moved, renamed or deleted a so called redirector will be created which purpose is to make sure that a class trying to load an asset still loads the correct asset, even if the path of the asset is no longer valid (EPIC GAMES, 2017). However you should regularly use the fix up 'commandlet' to remove all redirectors (EPIC GAMES, 2017). Doing so will automatically update the references where ever it can. It is easy for the editor to change the blueprints using those references to point to the new file, but it can't change the c++ code, since c++ has them hardcoded in a string and it is impossible to find the correct strings and not accidently invalidate other strings. This makes it so that c++ might still use references which are not valid anymore. After I learned about the asset loading issues, I changed the code so I would use blueprint classes, deriving from c++ classes, to set references to materials and meshes.

2.3.2 Car Editor

While I was creating the car editor I learned about the garbage collector of the unreal engine 4, as I ran into a problem with it. The garbage collector deleted an object I was still using, which caused the game to crash (Rama, 2017).

The car editor required me to set up communication between blueprints and c++ classes as it uses a GUI to configure some properties of the blocks, like which block to place and what colour to paint it with. As GUIs have to be created using widget blueprints (EPIC GAMES, 2017), I needed a way to transfer data from it to c++ and

vice versa. I also needed to be able to call c++ methods from blueprints (EPIC GAMES, 2017).

Furthermore I learned how to properly map one numeric range onto another, which was needed to snap the blocks into grid cells, as blocks exist in the world space of the unreal engine 4 and the grid has a virtual local space. To properly place the blocks I had to map the world space onto the local space of the grid (Singhal, 2011). Lastly I had to change the default rounding behaviour of c++, since I still needed to round down at half way between numbers instead of rounding up. Meaning for a value of 1.5 I needed to round to 1 instead of 2. To achieve that behaviour, I used a simple implementation of the round function (dan04, 2010) and added a small value onto the rounding limit to change the exact point the rounding up occurs at.

2.3.3 Game

Working on the game itself, I learned how to transfer data between different unreal engine 4 levels using the game instance class (Rama, 2016).

Furthermore I learned how the vehicle system used by the unreal engine 4 works and about the limitation that it requires a single skeletal mesh to function (EPIC GAMES, 2017). That caused me to have to look into how to create my own vehicle system using physics constraints (UnrealTek, 2016).

2.3.4 Conclusion

Sometimes the initial research turns out to not work for the exact problem that is to be solved, which makes it so more research than initially planned is needed and that costs more time as planned. Some tasks seem easy at first but can cause problems that could not have been expected before, which is why a lot more time has to be planned in than it might seem at first. Even a good break down of a project into tasks will not encompass everything needed by the program, so it is essential

that you are prepared to back away from the initial planning and react to new challenges as they appear.

2.4 Difficulties

Not everything went as well as I would have hoped for and there have been a few difficulties that cost me a lot of time. The first big problem I encountered was when the grid object used to place and save blocks in the car editor got randomly destroyed by the garbage collector. It did not always happen and the timespan in where it happened differed every time, which mad it difficult to reproduce and to detect it earlier. I did not add the 'UPROPERTY' keyword before the pointer that holds the grid object, which is why the unreal engine 4 had no way to keep track of used references to that object. After the garbage collection system had been triggered it deleted the grid, since no references where counted and the object was considered to not be in use. The object was still needed even after it got deleted, which caused some variables to contain invalid data. Trying to place a block would show it in the wrong place as shown in Figure 8. And calling the method to save the block into the grid would crash the game, since the memory location was no longer valid.

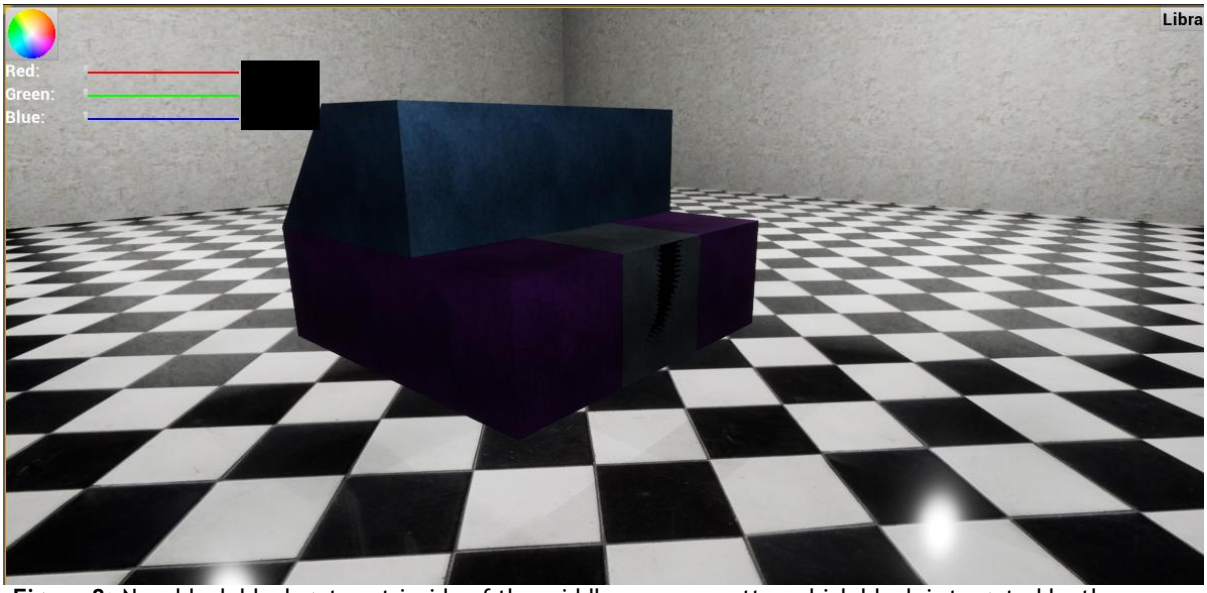


Figure 8: New black block gets put inside of the middle one, no matter which block is targeted by the mouse pointer, after the garbage collector deleted the grid object.

Not previously knowing about how the garbage collector of the unreal engine works made it difficult to find the problem. Debugging the code just showed me that some constants have suddenly gotten invalid data. After some research I learned that every pointer used in the unreal engine 4 should be accompanied by the 'UPROPERTY' keyword (Rama, 2017), so that the garbage collector knows about the object and can keep track of used references.

Another big difficulty I encountered was the previously mentioned problem that the vehicle system of the unreal engine 4 did not work as I had hoped it would. At first I thought, I just need to configure it correctly, that I had to create a virtual skeletal mesh or that I had to override the component to make the necessary changes to it. I had to read into the source code of the vehicle system a lot, just to realize it is not possible to use it the way I wanted to. After realizing it won't work, I had to start working on my own system. For that I had to do additional research to figure out how to move different rigid bodies, which are part of the same actor, separately

but still having them be dependent on each other. I found out about the physics constraint component and how to use it (UnrealTek, 2016).

The last big difficulty I had encountered, happened when I tried to implement and configure the physics constraints. In the prototype, which only uses blueprints, everything was working as expected, but my c++ implementation has a few issues, which I have yet to solve. In the blueprint of the prototype, I prebuild the car with all components already created and attached, to test the components behaviours and interactions, where in c++ everything gets constructed and configured in runtime. While everything gets constructed, rigid body welding, component attachments, component transformations and physics settings have to be managed. At first the rigid body welding was set so that the physics constraints could not be attached. I solved that by creating welding rules to not weld certain parts of the car with the rest. After that was solved, component attachments have been lost after the enable physics simulation flag was set. Simply enabling the physics simulation before the attachments have been made, solved that problem. The only problem that is still to be solved, is that after the physics constraints have been configured, added and enabled, the wheels seem to get moved to another place, but still are attached. I have not yet found out why that happens, but I have a few ideas to prototype and I will be able to solve that problem, given a bit more time.

Loading the car blocks or track elements from the respective libraries, also had a hard to find issue in where the object creation did sometimes fail. I had already created and configured objects inside the library and used the ‘NewObject’-method of the unreal engine 4 (EPIC GAMES, 2017). The ‘NewObject’-method has several overloads, which are not documented, since the official documentation of the unreal engine 4 is somewhat out of date. The ‘NewObject’-method has several overrides in where you can plug in an already created object and copy over the data to the newly created one. That can cause problems, when pointers are involved. Through

debugging that method I noticed, that first the new object gets created and its constructor gets called as expected. After that, data is copied over, overriding the pointer addresses, making them point to possibly invalid locations. To fix that problem, I had to use references to classes instead of already instantiated objects, allowing me to plug in preconfigured blueprint classes, which correctly will construct with the desired values. Finding the problem with the 'NewObject'-method, did cost me quite a lot of time.

Because of all the mentioned difficulties, which cost me more time than I had hoped for and because I did estimate too little time in the task breakdown beforehand, I was not able to fulfil all my aims and objectives. The project did not become a fully playable game, but there is not much missing for it to be considered one. I did manage to create a racing track on top of a spline and I was able to implement a car editor. I failed at fully combining everything together, because I did not fully manage to make the car drivable yet. As soon as the car is drivable, implementing the game rules will be easy to do and therefore one more week of work would fulfil all aims and objectives I had.

2.5 Implementation of Backup Plans

As mentioned before I needed to implement my own vehicle system, since the previous plan did not work. Using the physics constraints that I mentioned before, many different systems are possible. Physics constraints provide many options, but all I needed to accelerate the car was to create a constraint between the wheels and the cars chassis. The constraint had to be set to lock the relative positions of the different parts, so that if one moves, the other moves as well, but allow rotation of the wheel around the world y axis. Since the world y axis always stays the same, but the cars orientation changes, it was only able to drive in one direction. That is

why I had to switch to the physics constraints own right vector instead the worlds y axis, as it starts off as the same, but the right vector rotates with the car.

Steering the car is a bit more complex to set up. To steer a wheel a second physics constraint and a dummy object is necessary (UnrealTek, 2016). The steering constraint has to lock the dummy object to the body and limit one rotation axis to a desired range. The second constraint then has to lock the wheel to the dummy object and free only the one rotation axis needed for acceleration.

To improve the wheel behaviour, it helps to create physics materials. Physics materials set the friction and restitution factors of an actor. If two actors interact with each other the friction and restitution values get combined in a way defined by the material or a project default (EPIC GAMES, 2017). Correct values for the physics materials make the driving experience more realistic.

3. Evaluation and Reflections

3.1 Evaluation

3.1.1 Evaluation of Aims and Objectives

As it can be seen from the product, some of the objectives, which can be seen in section 1.1.3 of this document, have been have been met whereas other have not. The issue that has not been solved yet is the problem where the wheels get moved away from the car as the car is build and set up.

3.1.2 Strength and Weaknesses of the Product

I don't know of any bugs or errors in the game, besides the not completed car functionality, therefore the product can be considered robust and does not crash, even though it is still in its development phase and far from being the finished product I'd like it to be one day. The controls, when known, feel very intuitive and are easy to use, still they would probably be changed and tweaked in the finished product to give an even better experience while playing the game.

The physic behaviour of the car prototype does not feel the way it should do, therefore more tweaking is required. As of now, the game still lacks options to individualize the cars, which makes it so it is not much fun in the long run, but then again, it is still in development and more options to individualize and tweak the cars behaviour will be provided in the finished product.

3.1.3 Strength and Weaknesses of the Process

At the beginning I created categories of tasks, as they can be seen in section 1.4.2 in this document, and filled these with small tasks. That helped a lot in getting an image of the completed product and it was helpful during the development phase, to come back to check that everything has been implemented and what should be implemented next. Creating these tasks did not cost much time, therefore it was

worth doing, since it was able to save time during the rest of the development. The only issues I encountered with the task list, was that the time estimates have been very wrong for a lot of them and some tasks should have been split up even more.

Creating prototypes for different sub-systems of the game had its advantages and disadvantages on its own. On the plus side, the prototypes have been helpful to develop and test a subsystem without interference from other code. Since the prototypes are their own project, they start in a clean environment and can focus on the only task they are designed for. If a prototype does not deliver the desired results, it can simply be deleted and recreated without leaving any dead code in the main project. Sometimes the code of the prototype did not always fit perfectly into the main projects code and therefore did not always work as expected at first. Creating and integrating prototypes costs time, but in the long run it can save a lot of trouble.

Due to the lack of experience with the unreal engine 4 I started with, some tasks I completed had to be redone several times. I didn't know about redirectors at first, which led me to load the assets I needed inside the c++ code. After running into problems with it I had to make a lot of changes to load the assets with blueprints but provide functionality and the interface to use over c++. That problem occurred with the track tool and the car editor, which is why I lost time in redoing those tasks.

3.2 Reflections

3.2.1 What went right?

I have been happy with the tasks I created, since they were easy to follow and I could easily see what comes next. Most of the tasks also have been small enough to complete them in a short time. Categorizing the project into its main categories was

very helpful, since it allowed me to concentrate on one area of the game at a time without jumping around too much.

Creating the prototypes was helpful to create and test independent systems without having to experiment inside the main code. Creating ad testing that code in the main project without a prototype could have left some code that would need to be deleted and some changes to the code would have been difficult to undo.

Most of the time the project progressed well, as a lot of information was easy to get from different sources and I am satisfied with most of the progress I made.

3.2.2 What went wrong?

Unfortunately not everything went as smooth as I had hoped. I lost time from redoing tasks or hard to find glitches. Most of the time loss stems from lack of experience, which on the other hand helped me learn.

The worst thing that went wrong has been the issue with the vehicle movement. Had I known beforehand, that I have to create a vehicle system from scratch, I had spent less time at the other systems. The other systems would not have been as robust as they are now, but the game could have been completed.

3.2.3 What was learned?

Through everything that went wrong and that went right, I learned a lot. I started with a very basic knowledge of the unreal engine 4 and now I feel very comfortable in using it. I learned about the garbage collector of the unreal engine, asset loading, communication between blueprints and c++ and much more.

Due to creating different prototypes I learned about how far to create them and how to focus them on the tasks they are created for. Also I learned that some tasks should include more time to create prototypes or to research the task and that tasks should really be as small and defined as possible. That way less gets missed and

many things can be researched beforehand, since the tasks, that might have been missed can create problems that have not been thought of before.

3.2.4 If the project were to be repeated, what changes would be made to improve the project?

Due to the experience I gained in using the unreal engine, I would make less mistakes and save a lot of time by doing things right from the beginning, if I were to repeat the project.

Yet there are things that I would do differently if I were to repeat the project. I would create more robust prototypes and try to get the prototype code closer to the main project code, before I transfer it. This would help me discover errors sooner and I can fix them in a smaller, more isolated environment. Had I done that the first time, I would probably have a working car.

I would better plan the tasks, in that I would create even smaller tasks, which would increase the accuracy of my time estimates and I would notice tasks that are missing more easily.

3.3 Future Expansion

The project as it is can be expanded a lot to create a high quality and fun game. To really finish and polish it, the assets should be professionally created and the car editor should be designed by a professional designer. With a completed track tool, designers can create several different tracks for players to drive on.

3.3.1 Track Tool

The track tool can be expanded by adding more track element types, like different obstacle types, slowdowns and different road types, like asphalt, dirt and many more. To make the different road types interesting and realistic, they should get physic materials to represent the material they are made of and ensure correct car

behaviour while driving on them. Intersections are not supported in the current version, making that an improvement which can add a lot to the tracks. The spline point attributes should get limited to certain values to make sure that tracks are always drivable. The track tool should gain attributes to allow different racing types to be handled, for now it only supports lap races.

3.3.2 Car Editor

To improve the car editor, many new blocks should be added. Blocks should weight differently, which should be taken into consideration for the physics calculations. Additionally the car editor does not yet have engine blocks, which could allow for much more tweaking of the car. Different fuel blocks would help in making the car design more interesting by giving the player more to think of, while balancing out every attribute of the car. Currently there is only one wheel type available, but more wheels with different designs and properties can strongly increase the user experience of the editor.

The ability to change some settings on blocks, like the engines performance, whether a wheel can be accelerated or steered and to what degree, is not available yet, but should be added to help the player create the perfect car.

To help the player tweak their car, there should be an easy to access test track build into the car editor. This would allow the player to build a car, test it without loading times and change it until it performs as they want it to.

3.3.3 Game mechanics

First of all the game needs to be completed in order to expand on it, but after it has been gotten to the point which was planned for this project, there are still many great ways to expand it.

Different game modes, like drag race, drift race or races from point A to B would give the game more variety. One optional task I created mentioned random track generation, which would be an excellent expansion to make the game a new experience every time. Through using splines it should not be too difficult to randomly create tracks. The random track generator would need a sophisticated rule set to ensure good tracks. That of course would require more research and testing.

Adding multiplayer support would not be very difficult, as the track tool and car editor are all designed for a single player and only the main menu and the main game would need to be adjusted. The main menu would need a lobby to allow for game creation and to join games. The game needs to spawn pawns for every player and provide every player with the car data of each other. Network replication (Zoid, Rama and RoyAwesome, 2014) needs to be added, to give everyone the updates they need.

To make the game more interactive, usable items could be added. There are many possibilities for these items. Items to help yourself, like speed boost or shields, items to hinder opponents, like mines or homing missiles and many more. Allowing players to enable or disable items at the creation of a game session gives many more options to individualize the game and keep it diverse. Also it gives the players more reasons to create different cars. Some cars might be very suited for certain game modes with certain items, where others might be better at other modes with no items and so on. This allows for many different racing experiences and players can choose to play the game how they like it the most.

To improve the visuals, animations and particle effects can be added and especially for this game it would be easy to make cars fall apart if they get damaged too much.

4. References

dan04 (2010). *Concise way to implement round() in C?*. [online] Stackoverflow.com. Available at: http://stackoverflow.com/questions/4572556/concise-way-to-implement-round-in-c/4572591#4572591 [Accessed 5 May 2017].

Epic Games (2017). *Game Mode and Game State*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/GameMode/index.html [Accessed 4 May 2017].

EPIC GAMES (2017). *Blueprint Splines*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Engine/BlueprintSplines/index.html [Accessed 8 May 2017].

EPIC GAMES (2017). *C++ and Blueprints*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Gameplay/ClassCreation/CodeAndBlueprints/ [Accessed 5 May 2017].

EPIC GAMES (2017). *Creating and Displaying UI*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Engine/UMG/HowTo/CreatingWidgets/ [Accessed 4 May 2017].

EPIC GAMES (2017). *Physical Materials User Guide*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Engine/Physics/PhysicalMaterials/PhysMatUserGuide/ [Accessed 7 May 2017].

EPIC GAMES (2017). *Redirectors*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Engine/Basics/Redirectors/ [Accessed 4 May 2017].

EPIC GAMES (2017). *UObject Instance Creation*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Objects/Creation/ [Accessed 5 May 2017].

EPIC GAMES (2017). *Vehicle User Guide*. [online] Docs.unrealengine.com. Available at: https://docs.unrealengine.com/latest/INT/Engine/Physics/Vehicles/VehicleUserGuide/index.html [Accessed 4 May 2017].

Irascible, [. (2014). *[Twitch] Fortnite Developers Discussion - Apr. 17, 2014.* [online] Forums.unrealengine.com. Available at: https://forums.unrealengine.com/showthread.php?3035-New-Twitch-livestream-with-Fortnite-developers-Thursday-April-17&p=19464&viewfull=1#post19464 [Accessed 4 May 2017].

Rama (2016). *Game Instance, Custom Game Instance For Inter-Level Persistent Data Storage - Epic Wiki*. [online] Wiki.unrealengine.com. Available at: https://wiki.unrealengine.com/Game_Instance,_Custom_Game_Instance_For_Inter-Level_Persistent_Data_Storage [Accessed 4 May 2017].

Rama (2016). *Save System, Read & Write Any Data to Compressed Binary Files - Epic Wiki*. [online] Wiki.unrealengine.com. Available at: https://wiki.unrealengine.com/Save_System,_Read_%26_Write_Any_Data_to_Compressed_Binary_Files [Accessed 4 May 2017].

Rama (2017). *Garbage Collection & Dynamic Memory Allocation - Epic Wiki*. [online] Wiki.unrealengine.com. Available at: https://wiki.unrealengine.com/Garbage_Collection_%26_Dynamic_Memory_Allocation [Accessed 4 May 2017].

Rama and Zyr (2016). *File Management, Create Folders, Delete Files, and More - Epic Wiki*. [online] Wiki.unrealengine.com. Available at: https://wiki.unrealengine.com/File_Management,_Create_Folders,_Delete_Files,_and_More [Accessed 4 May 2017].

Singhal, A. (2011). *Mapping a numeric range onto another*. [online] Stackoverflow.com. Available at: http://stackoverflow.com/questions/5731863/mapping-a-numeric-range-onto-another/5732390#5732390 [Accessed 4 May 2017].

UnrealTek (2016). *Custom Vehicles in Unreal Engine 4 - YouTube*. [online] YouTube. Available at: https://www.youtube.com/playlist?list=PLXAwKe8ok5isn3wy5J5_adkg0DTOnLsq2 [Accessed 4 May 2017].

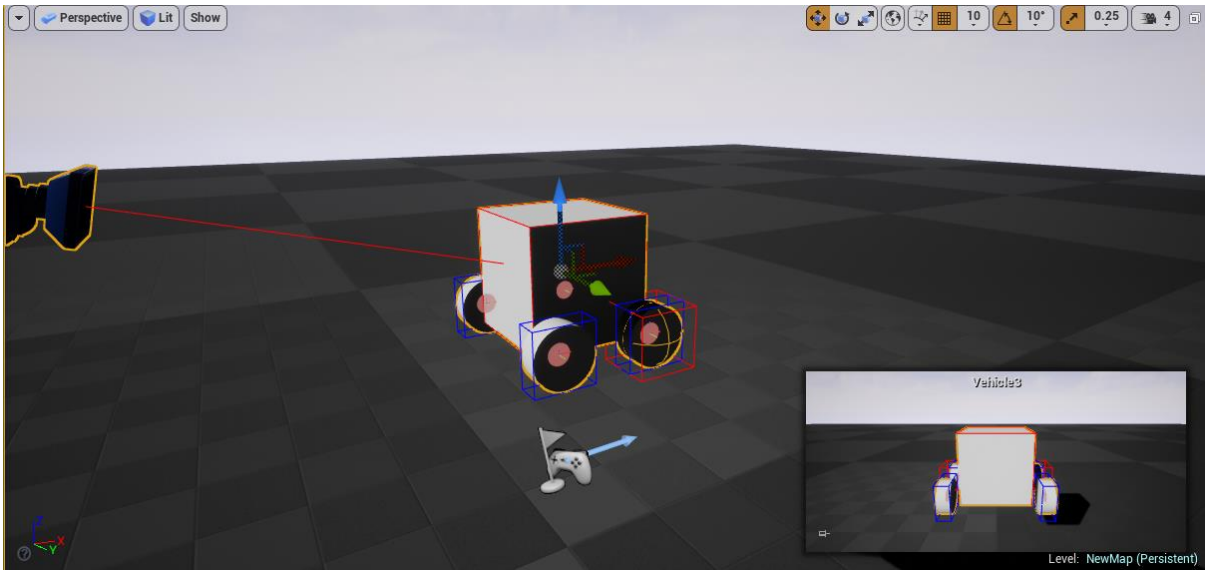
Zoid, Rama and RoyAwesome (2014). *Replication - Epic Wiki*. [online] Wiki.unrealengine.com. Available at: https://wiki.unrealengine.com/Replication [Accessed 7 May 2017].

5. Bibliography

Boughen, M. (2012). *Marble and Tiles*. [online] Mb3d.co.uk. Available at: http://www.mb3d.co.uk/mb3d/Marble_and_Tiles_Seamless_and_Tileable_High_Res_Textures_files/Tiles_05_UV_M_CM_1.jpg [Accessed 5 May 2017].

Goodtextures (2011). *8 Seamless Plaster Textures*. [online] Brusheezy. Available at: https://www.brusheezy.com/textures/27382-8-seamless-plaster-textures [Accessed 5 May 2017].

Appendix A



Prototype for the custom created car controller. Constraints are shown with the red and blue boxes.