OULUN AMMATTIKORKEAKOULU

Toni Keränen

Creating a hovercar racing game with Unreal Engine

Creating a hovercar racing game with Unreal Engine

Toni Keränen
Bachelor’s Thesis
Spring 2024
Business Information Systems
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Business Information Systems

Author: Toni Keränen
Title of bachelor's thesis: Creating a hovercar racing game with Unreal Engine
Supervisor(s): Matti Viitala
Term and year of completion: Spring 2024
Number of pages: 33

This thesis describes the development process of a game I was making for my own company. The goal was to create a protype hovercar racing game for PC using Unreal Engine.

The thesis describes the development steps of this game starting with basic introduction to the tools used, Unreal Engine as the game engine and Blender for 3D-modeling. Secondly going through the creation of the game prototype, which consists of player movement, game physics and race progression. Finally, some thoughts about the prototype and plans regarding further development.

Keywords: Game Programming, 3D

TABLE OF CONTENTS

1 INTRODUCTION

The reason for choosing the game as the subject of my thesis was the fact I was already working on it during my studies. This meant it was a convenient way of combining work and school into one project. I also see this thesis as a way of promoting the game during further development.

The game itself was born out of love for Nintendo's F-Zero (1) game series and the desire to see a sequel. The last mainline entry in the series, F-Zero GX was at the time of writing this thesis over 20-years old (2). The goal of this game is to carry on the spirit of F-Zero, while creating a product that's appealing to both F-Zero fans and people unfamiliar with it.

Before starting work on the thesis, I already had some amount of experience with Unreal Engine and Blender through independent studies. I also had game development experience through smaller demo projects and one actual mobile release, on which I worked mainly as a programmer.

The goal of this thesis is to document the steps taken during the prototypes development and at the end of the thesis the reader should have a good understanding of the development process of this kind of game.

By the end of this thesis, besides the systems being developed, the content of the prototype will consist of 1 test track, made with a 3D modeling software, that the player can race around alone.

2 TOOLS

This chapter goes through the tools used in this project, namely Unreal Engine and Blender. In the case of Blender this chapter will give a very general summary of the application, but later go into more detail as the more specific parts become relevant.

Unreal Engine

Unreal engine is a game engine developed by Epic Games. It was first showcased in the 1998 game Unreal, with which it was simultaneously released. The latest release at the time of writing this thesis was Unreal Engine 5 and is the version this game is using. In Unreal Engine the programming language is C++, but it also has a visual scripting system called Blueprints (3). Unreal engine is free to use until the lifetime gross revenue of the product exceeds $1 million USD, from which point onwards a 5% royalty fee applied to any further sales (4). There are 2 main reasons for choosing Unreal over something like Unity, which is a more commonly used engine in indie game development (5). First, Unreal Engine has better graphical options, which makes it easier to get the game to look good. Secondly, Unreal Engine has built-in multiplayer support (6).

Blender

Blender is a 3D computer graphics tool set originally released in 1994 by a Dutch animation studio, NeoGeo. As of 2002 blender has been a free and open-source application, maintained by the Blender Foundation, as well as the community (7). Blender offers a large variety of different tools to their users. Some of these include 3D modeling, animation, video editing and even game development. This thesis only focuses on the 3D modeling, as it becomes relevant in the development of the game physics.

3 DESIGN

This chapter goes more into detail about the various systems being developed and explains their purpose. The chapter begins with explaining the idea of the game, followed by the players movement, game physics and finally tracking the progression of the race.

The basic game concept is a high speed, high risk, high reward racing game. The player races at over 1000 km/h on twisty rollercoaster-like roads, where every mistake could result in a crash. The players take part in 30 competitor races on enormous racetracks in a futuristic alien setting.

3.1 Player Movement

Forward Movement

This system is responsible for moving the vehicle forward. Its main components are Max Speed, Acceleration and Deceleration. These components are by default static numbers, but Max Speed and Acceleration are modified by other systems.

Weight

Like the components in the previous entry, Weight is a static number. It influences Handling and Grip.

Turning

Turning is the system that turns the vehicle left and right. The main system affecting it is Handling. Turning also slows the vehicle down, by decreasing its max speed and acceleration. The amount is determined by the angle and the length of the turn.

Handling

The Handling system determines how much the vehicle turns per input. It is negatively affected by Weight, but also the vehicle’s current speed. This means a heavier vehicle and a high speed have a negative effect on Handling, which means the vehicle turns less per input. The vehicle also has a handling stat, which is the primary factor in calculating how much the vehicle turns.

Strafing

Strafing moves the vehicle left and right without turning it. The purpose is to adjust its horizontal position with minimal speed loss. This action also tilts the vehicle slightly in the direction of the strafe.

Boost

Boosting gives the vehicle a short burst of speed, temporarily increasing its Acceleration and Max Speed.

Grip

The Grip system, like Handling, affects the vehicle’s turning, but where Handling affects how much the vehicle turns, the Grip system is designed to limit the players turning. The system punishes players for turning too much at high speeds by breaking its grip and causing oversteer. Oversteer means the vehicle turns more than the player expected and can often lead to crashing. It is positively affected by the Weight stat, meaning it is harder to lose grip. It is also affected by the vehicle’s current speed, which means the faster the vehicle is moving, the easier it is to cause grip loss. Similar to Handling, Grip also has its own stat, which determines how prone the vehicle is to losing its grip.

Quick Turn

The quick turn allows the vehicle to clear turns quickly, without a need for a braking system. It is essentially a controlled grip loss, where making a turn greatly exceeds the default amount the vehicle would turn.

Air Movement

From the perspective of player movement air movement is largely the same as its ground counterpart, with only Forward movement, Strafing and Quick turn missing. Unlike on the ground, in the air the player can tilt the vehicle forward and backwards. Tilting the vehicle forward increases the vehicle's falling speed, while tilting it backwards slows it down. Falling down, or gravity will be explained in more detail in the Game Physics section.

3.2 Game Physics

Hover

Hover lifts the vehicle off the ground and keeps it at a constant height above it. This includes adjusting its position when encountering uneven terrain.

Vehicle Rotation

This is by far the most important system in the game. It rotates the vehicle to match the road's rotation, meaning the vehicle sticks to the road's surface like a magnet. This allows driving on roads that are, for example, upside down, or unconventional road shapes, like cylinders. If the vehicle is strafing, the rotation will be slightly adjusted in the direction of the strafe. The vehicle also rotates so its bottom is facing the ground when entering air.

Gravity

Pushes the vehicle down when it is in the air. Falling speed is determined by the vehicle’s speed at the start of the fall, then further adjusted by tilting the vehicle forward or back, as described in the Air movement section. This system also allows the vehicle to lift off the ground, past the boundaries set by the Hover system, in some scenarios, like certain types of uphill. Finally, it calculates the vehicle’s velocity to ensure it does not immediately start falling towards ground, but rather the velocity gradually adjusts from forward to a downwards velocity.

3.3 Race Progression

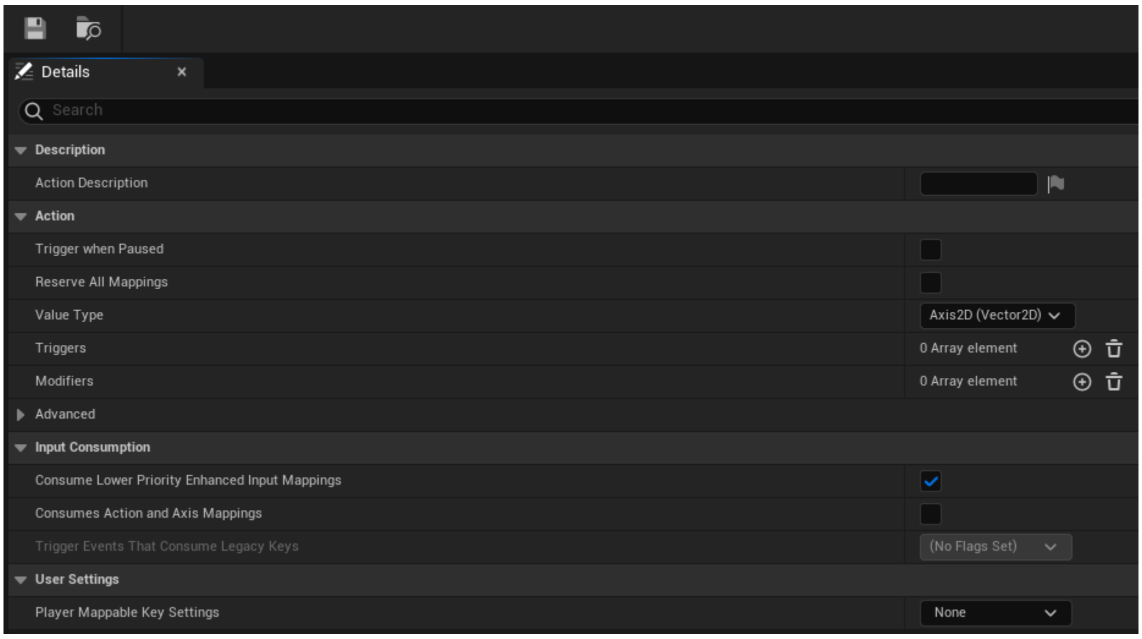
This system keeps track of the players progression throughout the race by counting the laps the player has completed. The system consists of checkpoints, which the player must go through to count their laps, a starting line, where the laps progress and a race manager, which handles the actual logic of the system.

4 DEVELOPMENT

This chapter goes through the hands-on development of the game. It starts with some Unreal Engine basics, through which the player movement is mainly done. It then dives into the game physics and lastly tracking the race progression.

4.1 Enhanced Input

Unreal Engine 5 introduced a new input system called Enhanced Input, which replaced the old input system (8). The 2 main components of this system are Input Action, displayed in figure 1 and Input Mapping Context, displayed in figure 2. Input Actions represent an action the player can do, like move forward or boost.



Description	
Action Description	
Action	
Trigger when Paused	☐
Reserve All Mappings	☐
Value Type	Axis2D (Vector2D)
Triggers	0 Array element
Modifiers	0 Array element
Advanced	
Input Consumption	
Consume Lower Priority Enhanced Input Mappings	✓
Consumes Action and Axis Mappings	☐
Trigger Events That Consume Legacy Keys	(No Flags Set)
User Settings	
Player Mappable Key Settings	None

FIGURE 1. Input Action.

"Input Mapping Contexts are a collection of Input Actions that represents a certain context that the player can be in" (8). This is where things like key bindings, triggers and modifiers are set for Input Actions.

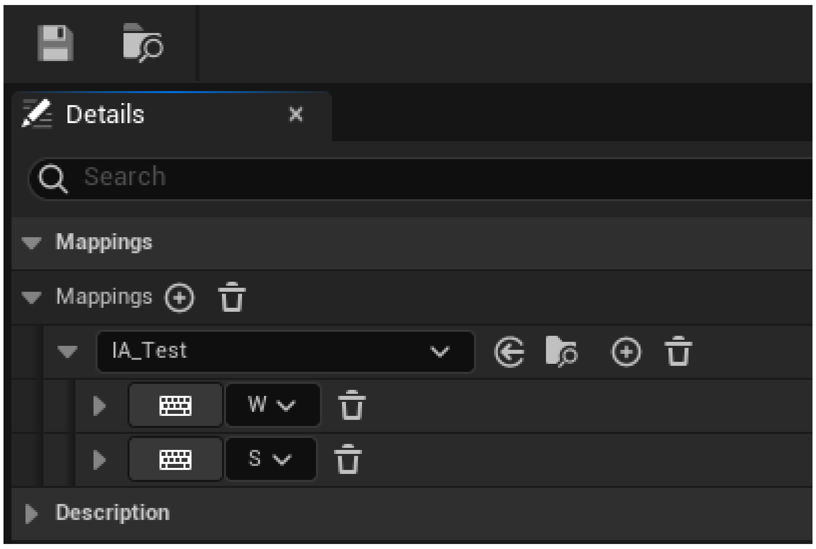


FIGURE 2. Input Mapping Context.

4.2 Blueprint

The blueprint system is Unreal Engines own visual scripting system. The basic building blocks of this system are the nodes, which can be compared to a C++ function. Some of the most commonly used nodes are BeginPlay, Tick and Input Action nodes. BeginPlay is executed when the object the blueprint is a part of is spawned into the game. Tick is executed every frame, which means it can be used to continuously execute something. The input action node is where the game functionality is bound to the button set Input Mapping Context. A typical blueprint node consists of an execution pin, which controls the order in which the node actions are executed and value pins, which contain, for example, variables.

The following figure is of a blueprint graph, in which a player controller is retrieved, and the input mapping context is added to it. A player controller takes the input data from the player and translates that into actions (9).

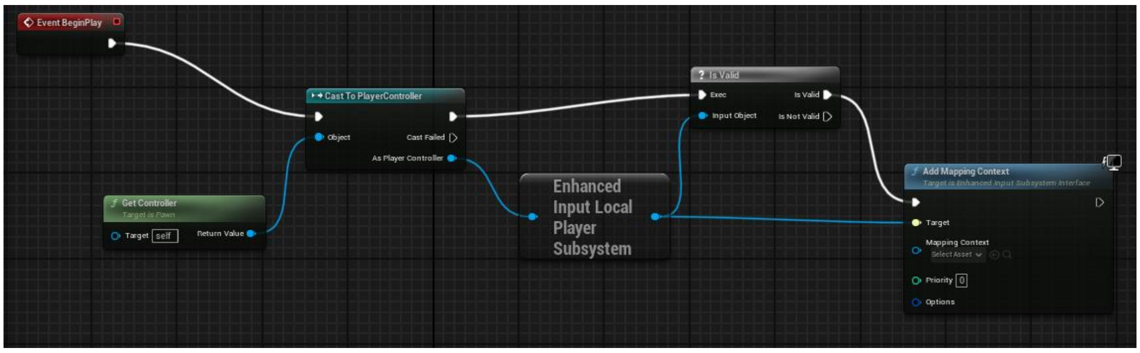


FIGURE 3. Blueprint nodes.

4.3 Player Movement

Forward Movement

Developing player movement starts with moving the player forward. First an Input Action is created. The value type for this input is Axis2D (Vector2D). It is a type of input that can capture 2D directional values, e.g. forward and back or left and right. Vectors in Unreal Engine are a data type used to represent points or directions in 2D or 3D space. The Input Action is then added to Input Mapping Context, where it is given a keybind. After that the actions enhanced input node is added to the blueprint event graph. There its executor pin is connected to an Add Movement Input through a branch node, that checks if the vehicle is on the road.

The road check is done through line trace, also known as raycast. It is a beam that upon hitting an object can return information about said object. In this case it is shot directly down from the vehicle. If the line trace hits something, it means the vehicle is on the road and is free to move forward.

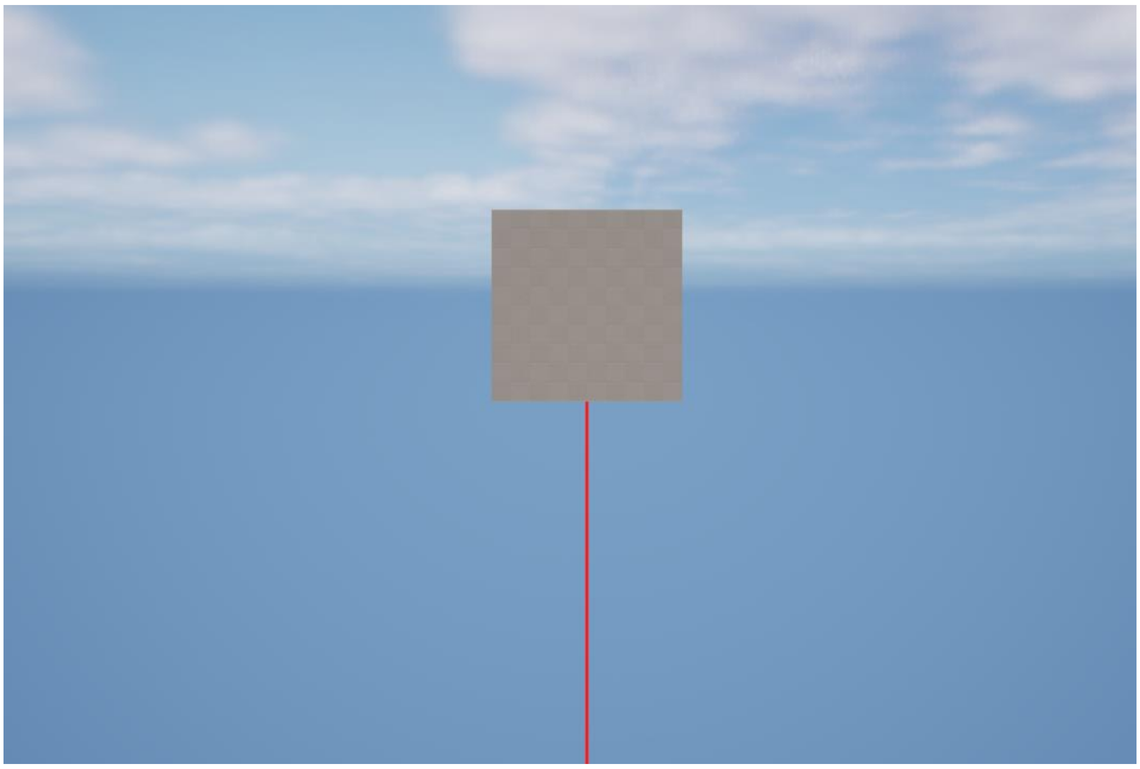


FIGURE 4. Line trace(red).

The base movement amount is calculated by taking the Action Value Y pin from the enhanced input node, which is the axis on which back and forth movement happens and multiplying that by an input modifier.

The following figure shows an enhanced input node. It contains an executor pin (Triggered) and Action Value X and Y pins.

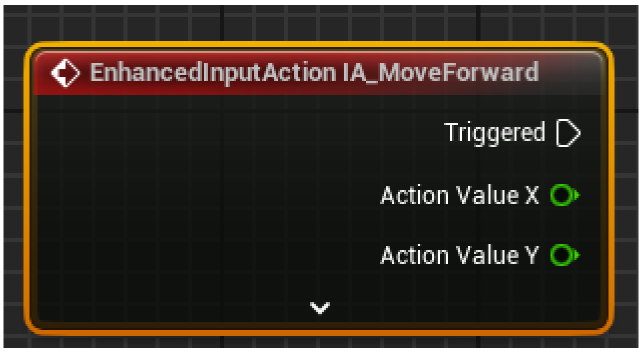


FIGURE 5. Enhanced Input node.

The result is then multiplied by DeltaTime. DeltaTime refers to the amount of time since the last frame. It is used to make actions frame rate independent. This result is then added as the scale value of the Add Movement Input node. The result is *Base movement amount* = *Input value * input modifier * DeltaTime.* Finally, to get forward direction, the vehicle's rotation is combined with a Get Forward Vector node.

Turning

The first steps of the turning system are very similar to Forward Movement. An Input Action is created then added to the blueprint graph through Input Mapping Context and the enhanced input node executes the movement, with an Add Actor Local Rotation node being used instead of Add Movement Input. The turning system has 2 main components: Calculating how much the vehicle turns, from here on out referred to as Handling and how turning affects the vehicle's speed.
The first half of the turning system, Handling, uses input X instead of Y, which is the axis for sideways movement. The input value is multiplied by a base turn amount. The base turn amount is then modified by the vehicle's current speed, weight and most of all, its Handling stat.
The speed modifier is calculated by multiplying it with a speed impact modifier. The handling modifier is split by a handling impact modifier and finally the weight modifier is calculated by multiplying weight with a weight impact. The result is *Base turn amount * speed modifier * handling modifier * weight modifier* = *turn amount.* The turn amount is then multiplied by delta time and made into a rotator with the Make Rotator node.
The Make Rotator node ensures that the turning happens on the correct axis. 3D computer graphics have 3 axes: X, or roll, Y, or pitch and Z, or Yaw. Using a car as an example, rotating on the X axis rotates the car towards its side. Rotating on Y tilts the vehicle's nose up and down and doing it on Z is how a car is normally turned and is what this function is also doing. The multiplied turn amount is connected to the Z pin in the Make Rotator node and the result is used as Delta Rotation in the Add Actor Local Rotation node.

The second half of this system is slowing down the vehicle when it turns. The first step is acquiring the angle of the turn. This is calculated by subtracting the vehicle's current Z rotation from its rotation when the turn was initiated. Turn angle is then split by the maximum angle at which the vehicle's speed loss keeps growing. Acceleration and max speed are then multiplied by the result and adjusted accordingly.

Boost

The initial steps of the boost system are like in the previous 2. When a boost is activated by the player, a check is made to see if it is already active. This is to prevent the player from activating the boost before it is finished. If the boost is not active, it is marked as so.
Following that the vehicle's acceleration and max speed are increased from their default amount to their boosted versions, for the duration of the boost. The duration of the boost is handled by a timer and timeline nodes. These nodes track the duration of the boost and once it has ended, returns acceleration and max speed to their default values, as well as marks boost as not active to allow it to be activated again.

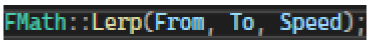
Strafe

The initiation of strafe is done in blueprint and the actual functionality in C++. Strafe is split into 2 Input Actions, 1 for strafing left and 1 for right. Before being triggered, the following conditions must be met: The vehicle's speed must be over 0, meaning the action cannot be performed while stationary. Quick turn cannot be active, as these 2 actions are not compatible with each other. Strafe in the opposite direction must not be active and finally the vehicle must be on the road. If these conditions are met, strafe is marked as active, and the C++ function is executed.

The C++ function handles the actual movement, as well as the speed loss.
The speed loss part works similar to the 1 in the Turning system, but instead of using the angle of the turn to determine how much the vehicle's max speed and acceleration are decreased, it is calculated using the duration of the strafe. The duration is acquired from the input action node, which has a Triggered Seconds pin, which contains the duration of the action in seconds. This information is passed to the function when triggered, where it is multiplied by DeltaTime for frame rate independence.

The vehicle's max speed and acceleration are then decreased using a Lerp function. Lerp stands for Linear Interpolation (10). Lerp can be used to create a smooth transition between 2 points, in this case it smoothly adjusts the vehicle's current max speed and acceleration towards their reduced states over a period of time determined by the duration of the action.

Unreal Engine C++ Lerp function. *From* is the value at the start, *To* is the value being lerped to and *Speed* is the speed at which the lerp happens.



```
FMath::Lerp(From, To, Speed);
```

FIGURE 6. Lerp C++ function

The second part, moving the vehicle left or right begins by getting the vehicle's right vector. A new vector is then created, which represents the strafe movement. This vector is calculated by multiplying the right vector with strafe speed, which is a static number. Then multiplied by strafe direction, which, like the strafe duration is also acquired from the input node. This is then finally multiplied by DeltaTime. The result is *Right Vector * speed * direction * DeltaTime.* This is then added to the vehicle's current location and finally the result is set as the vehicle's location. The rotation part of this system is covered in the Vehicle Rotation chapter.

Grip

The grip system is done entirely in C++. The reason for this is some more complex functions are easier to do in C++, which is a more controlled environment, and one not reliant on blueprint nodes.

The stat modifiers are calculated in a similar fashion to the handling system. Weight is multiplied by a weight modifier; the grip stat is split with the stat modifier and speed is multiplied by the speed modifier.

How the system works in practice, is the players' input value is checked. By default, a normal button press has the value of 1. Things like controller stick however, range between 0 and 1 or -1 depending on the tilt of the stick. This behavior can be replicated with things like keyboard keys, but that is not in the scope of this prototype and will only focus on the controller part. The system then calculates 2 points between 1 and -1, which when the input value crosses triggers grip loss.

The point at which grip loss occurs is calculated by adding the modifiers together and clamping the value between 0 and 1. It then checks if the players input value is at or has crossed that point, after which the amount the vehicle turns is multiplied by an oversteer multiplier, to which the amount the input value has gone over the point of grip loss is added. For this system to work in both turn directions, the absolute value of the input is used, meaning for every value between 0 and -1, the minus is ignored.

Quick Turn

Unlike the previous actions, quick turn is initiated by pressing down 2 buttons simultaneously. The initial setup is the same, but additional steps need to be taken to check that both buttons are pressed. This is done through the button event node, which checks the status of the button in question. The node has 2 executor pins. 1 for pressing the key and 1 for releasing it. Pressing the key marks it as true and releasing it marks it as false.
When the quick turn is initiated, a check is made to see if the status of both buttons is true, meaning they're both pressed. If the check is passed, the vehicle's base turn amount is then multiplied by a quick boost modifier, which results in the vehicle being able to turn at a speed exceptionally higher than its default amount.

Air Movement

Air Movements initiation is similar to Forward Movement, using the same line trace, but instead is triggered when the line trace does not hit anything, meaning the vehicle is in the air. Air movement has 2 components: Turning, which behaves exactly as turning on the ground, and tilting the vehicle.

The tilting initiation is identical to moving forward and turning, where the input value is first multiplied by a modifier then multiplied a second time by DeltaTime. After the second multiplication the result is added together with the vehicle's Y-axis rotation, which is acquired through a break rotator node. That result is then clamped between a minimum and a maximum value, stopping the vehicle from rotating past those values. The return value is then used to make a rotator, which is finally used to set the vehicle's new rotation. As mentioned in the design chapter, tilting the vehicle forward and back adjusts the vehicle's falling speed, which will be covered in the Gravity chapter.

4.4 Game Physics

This chapter covers the games physics, Hover, Rotation and Gravity. These systems are mostly done via C++, with a few parts done through blueprint. The chapter begins with explaining some of the key concepts used in the creation of the game physics, especially the Vehicle Rotation system. These concepts include the structure of 3D objects, which will be created in Blender and how those objects are rendered in Unreal Engine.

Game engines such as Unreal or Unity typically have a built-in physics engine (11). In addition to these, standalone physics engines are also available, that are solely designed for physics simulation (12). This game uses neither and instead all the physics are done manually. The reason for this is the game has quite A unique physics system and doing it manually gives more control over it.

4.5 3D geometry basics

A 3D object consists of 4 components: Vertices, Edges, Faces and Triangles. Vertices are the points in which 2 or more edges meet. Vertices also have something called vertex normals. Vertex normals represent the average direction of all the faces that share said vertex. Edges are the lines that connect vertices together and form the edges of a polygon. Face is the flat surface of the polygon, in the middle of the vertices and edges. Faces consist of triangles, for example a face with 4 vertices has 2 triangles. Triangles are also how game engines like Unreal Engine render 3D objects. Faces, like vertices, also have normals. Face normals are used to determine the direction a surface is facing.

The following figure shows a default Blender plane. The plane consists of Vertices (orange dots), edges (orange lines) and a face (blue).

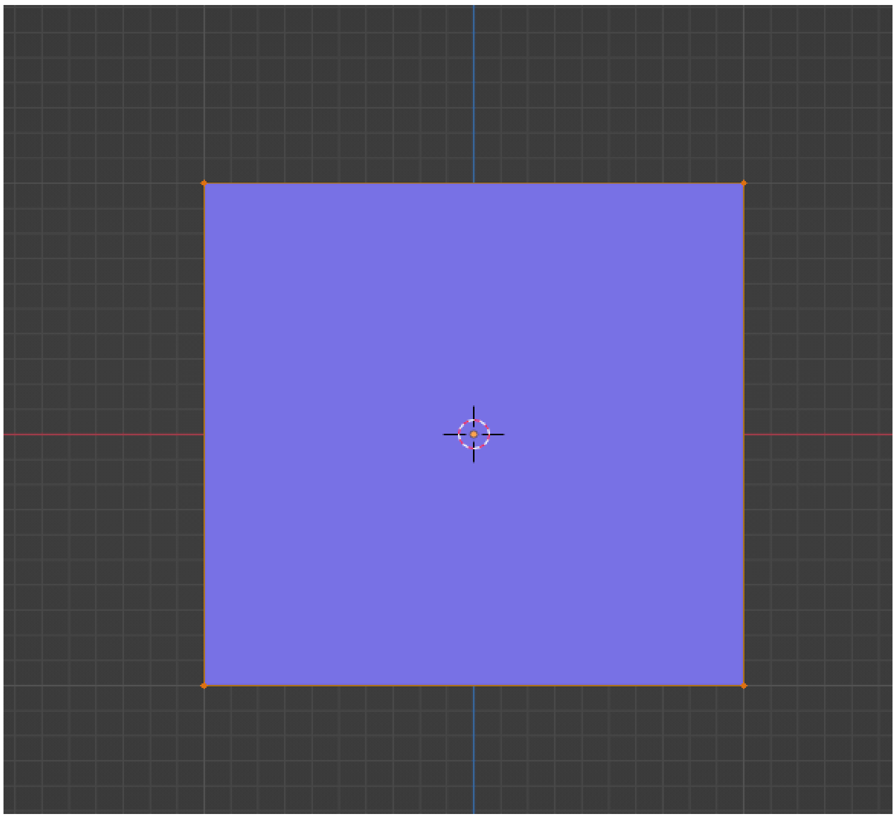


FIGURE 7. 2D plane in Blender.

This figure shows the plane in the previous figure rendered in Unreal Engine, using the wireframe view. Wireframe in Unreal Engine is a visual representation of the 3D object where only the edges that define the objects geometry are shown (13). The point is to be able to preview objects' geometry more accurately.

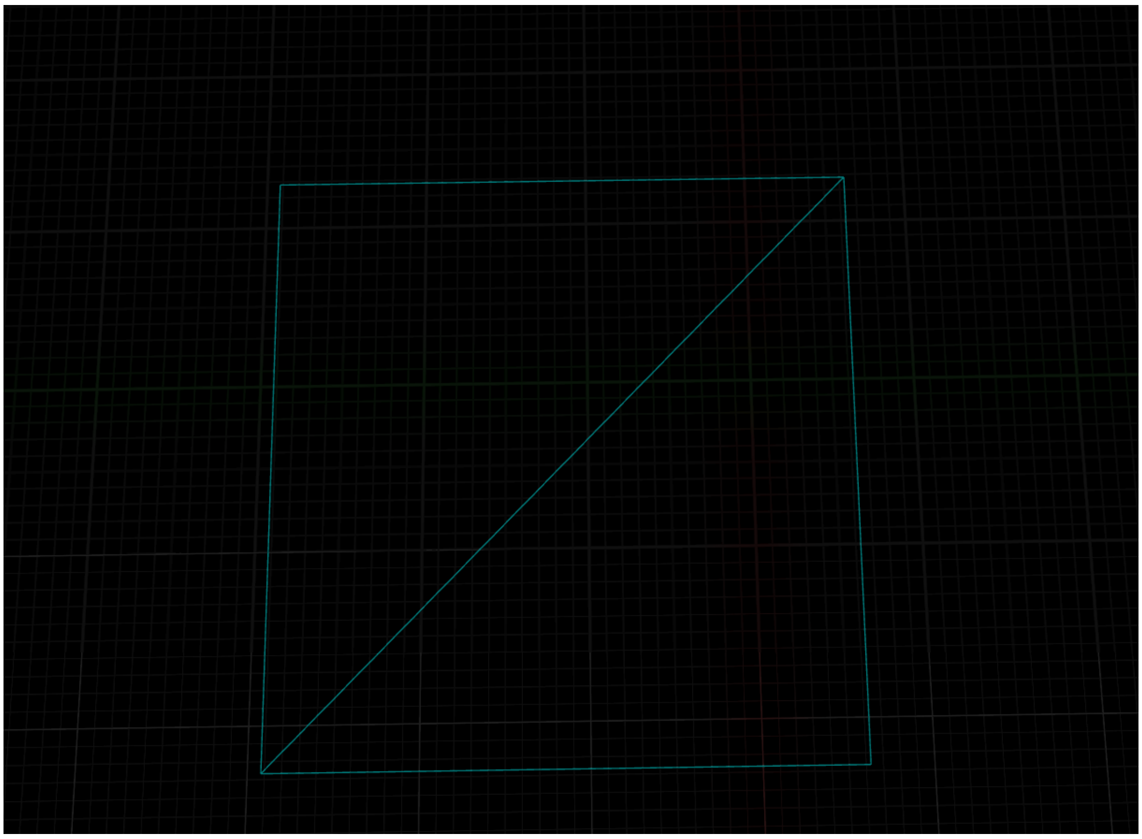


FIGURE 8. Plane in Unreal Engine

Unreal Engine, like many other game engines has a system called LOD (14), which is short for Level Of Detail. It is used to manage how complex objects are rendered at different distances. It does this by reducing the polygon count (triangles and vertices) of far away objects to improve performance.

4.6 Hover

Development of game physics starts off with the Hover system. The system uses the same line trace used before to measure the distance to the ground. This is acquired through Unreal Engines Distance function, which as the name implies, measures the distance between 2 points, in this case the vehicle and the ground.

First the system checks if it is on the ground and whether the vehicle's position is being adjusted through the Gravity system. Passing that, a check is made to see if the vehicle's distance to the ground is more or less than the desired hover height. If either one of these conditions are true, the vehicle's position is then adjusted to the desired hover height. First the direction in which the vehicle needs to adjust is acquired by getting the vehicle's up or down vector. Then, depending on if the vehicle needs to lower or lift a target height is calculated: *Lift target height -= direction of adjustment * (distance to the ground – hover height). Lower target height += direction of adjustment * (hover height – distance to the ground).*
Finally, the vehicle's location is adjusted according to the result.

4.7 Vehicle Rotation

As mentioned in the design chapter, this system is the most crucial one in getting the game to work as it should. The chapter begins with explaining some mathematical concepts and calculations through which the rotation is calculated, followed by the development part.

The vehicle is rotated using quaternions. A quaternion is a number system that can be used to represent rotations and orientations in 3D space (15). It has several advantages over other kinds of rotation systems, namely avoiding gimbal lock.
Gimbal lock is a phenomenon where one degree of rotational freedom is lost. What this means in practice is 2 axes are aligned in parallel, which results in the system going from 3 dimensions to 2.

The calculation for the vehicle's rotation was done using barycentric coordinate system. The barycentric coordinate system specifies a point inside a triangle and since everything in Unreal Engine is rendered as a triangle, it can be used to get a normal from the road and then used to rotate the vehicle accordingly.

Creating the system begins with the same line trace used in the movement systems. This line trace, when hitting a road returns information about it, which is stored in variables for later use. The information consists of the location where the line trace hit, the normal of the face that was hit, the actual component and finally, face index. Face index is a unique numerical identifier that every face in a polygon has.

After acquiring the information through the line trace, the stored hit component is used to get the road’s render data. This data includes among other things the road’s LOD information. The render data is then used to access the road’s static mesh vertex buffer. The static mesh vertex buffer contains vertex data such as vertex positions and normals. Finally, the road’s index buffer is retrieved, which is an array that defines how vertices are connected to form the road’s faces.

Once the data from the road has been gathered, 2 arrays are created, 1 for vertex positions and 1 for vertex normals. These arrays contain the positions of the 3 vertices of a triangle, as well as the normals of those vertices. Next face index is used to calculate the indices of the vertices inside the road’s vertex buffer.

Following that position of the vertices is retrieved from the vertex buffer using the previously calculated indices and the results are stored in the vertex positions array. Likewise, using the same indices the normals of the vertices are also retrieved from the vertex buffer and stored in the vertex normals array.

This data is then transformed into world space from local space. Local space refers to an objects own coordinate system, where X 0, Y 0 and Z 0 is typically at the center of the object. World space on the other hand is the coordinate system used by the entire scene or game world and is used to position and orient all objects within a scene.

After the transform, using the vertex positions along with the location of the line trace hit, barycentric coordinates are calculated. As previously mentioned, barycentric coordinates are used to calculate a position inside a triangle and in this case the triangle is formed by the vertex positions and the location is the hit location of the line trace. The calculation is done with Unreal Engines built-in function, ComputeBaryCentric2D.

The function returns the calculated barycentric coordinates, which consists of 3 components. Each of these components are then multiplied by 1 of the vertex normals from the normals array and added together in what is called normal interpolation. Normal interpolation in this context means determining an averaged direction of the surface based on the vertex normals.

After the normal interpolation, the shortest rotation from the vehicle’s up vector to the interpolated normal is calculated using Unreal Engines FindBetweenNormals function and the result is stored in a newly created quaternion. Following that the vehicle’s current rotation is acquired as a quaternion, after which the new quaternion is multiplied with the vehicle’s current rotation.

Following this a check is made to see if the vehicle is strafing. If not, the calculated rotation is set as the vehicle’s rotation. If the vehicle is strafing, an adjustment is calculated. The formula is *Desired adjustment degrees * direction of strafe.* After this the target rotation is calculated. First the vehicle’s forward vector is acquired. The following calculations are done using cross product. A cross product is a vector that is perpendicular to the vectors used in its calculation. After acquiring the vehicle’s forward vector, the cross product of the interpolated normal and the vehicle’s forward vector is calculated, resulting in a vector that points to the vehicle’s right side. Then using the newly calculated right vector, the vehicle’s forward vector is recalculated to ensure it remains in the correct angle. This is done by calculating the cross product of the right vector and the interpolated normal.

These vectors are then used to create a rotation matrix. A rotation matrix is a mathematical tool used to perform rotations in 3D space. The rotation matrix is then converted into a rotator, from which the X axis is extracted and adjusted based on the adjustment calculation. This newly created rotation is then applied to the vehicle, resulting in the vehicle rotating an extra *x* degrees in the direction of the strafe on the X axis.

The following figure shows the vehicle, represented by the cube. The vehicle is driving on the top of a cylinder, with its bottom facing the ground.

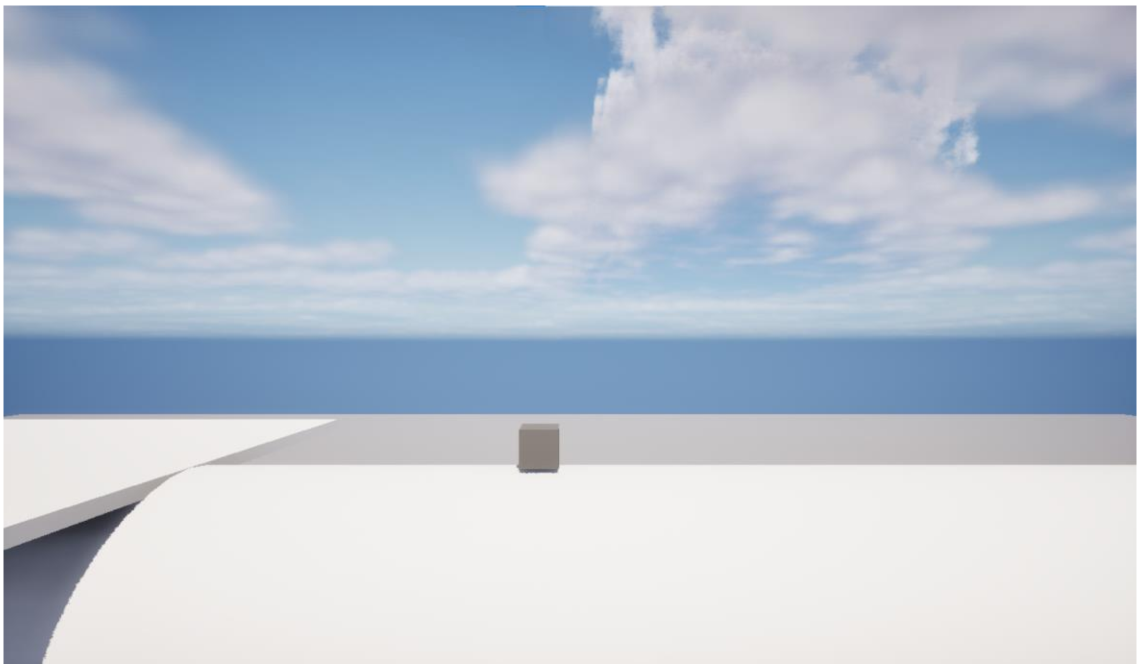


FIGURE 9. Driving on top of a cylinder.

This figure shows the cube clinging to the side of a cylinder. The cubes left side is facing the ground.

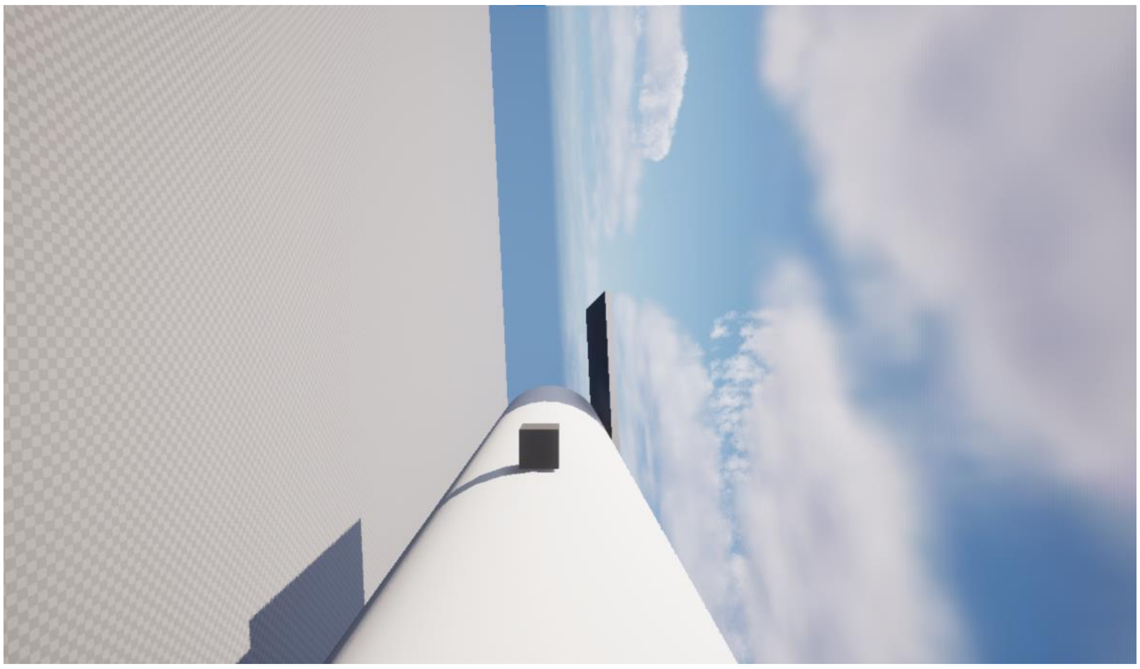


FIGURE 10. Sticking to the side of the cylinder.

This figure shows the cube upside down. It is clinging to the bottom of the cylinder with its top facing the ground.

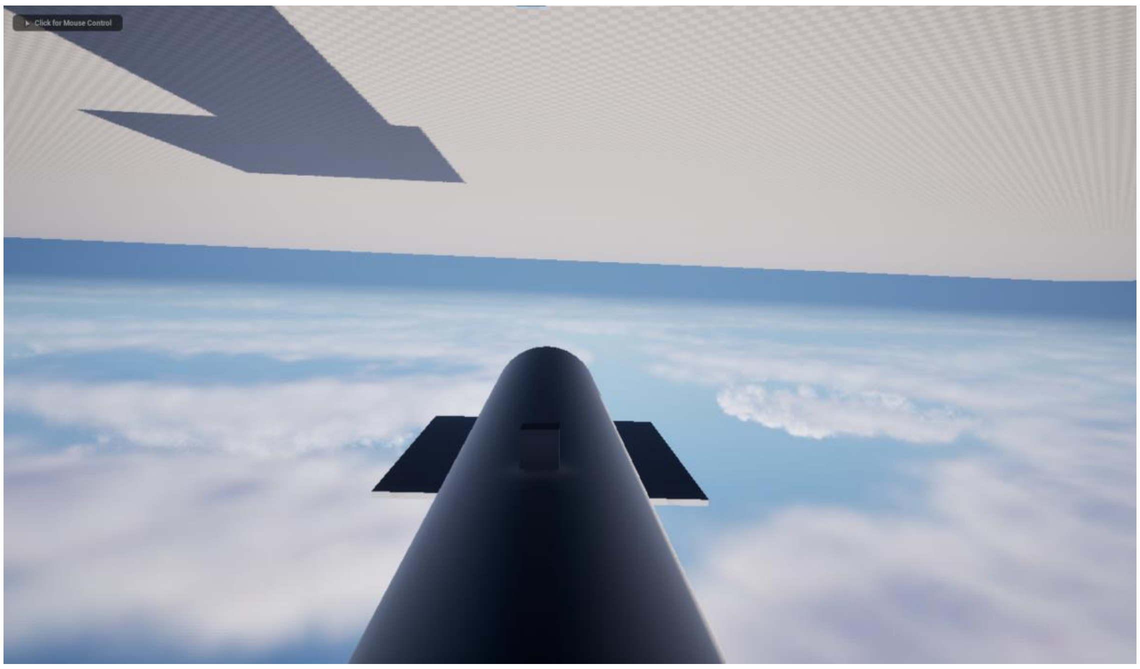


FIGURE 11. Sticking to the bottom of the cylinder.

The air rotation is triggered when the vehicle enters air, meaning the previously used line trace is not hitting anything. The calculation is done similar to the rotation during a strafe. First the vehicle's yaw when entering air is acquired. Following that world up vector is defined and the vehicle's forward vector is calculated. The forward vector is calculated by creating a rotation matrix that only has the previously acquired yaw as a component so the rotation only happens on the X and Y axes. After that the forward vector is extracted from this matrix to ensure the vehicle's forward direction aligns correctly with its yaw orientation. Then similar to strafe rotation, right vector is calculated, and the forward vector is recalculated, both via cross product. Finally, the rotation matrix is converted into a rotator and applied to the vehicle.

4.8 Gravity

The gravity system is done entirely in C++. It consists of 2 components: Behavior when the vehicle is not touching a road, and what happens when the vehicle is lifted into the air in certain track conditions.

First, the lift system. A new line trace is created and shot diagonally downward. It retrieves normal of the hit face before the vehicle has passed through that bit of road. The retrieved normal is then used with a world up vector to calculate the angle of the face that was hit. If the angle is a desired angle for allowing the vehicle to lift off the ground, the Hover system as well as Vehicle Rotation are turned off until the vehicle has landed to the height defined by the Hover system.

The gravity part begins by acquiring the vehicle's speed when the function was triggered. After that it is adjusted by getting the vehicle's pitch (Y) rotation then, depending on if the vehicle is pitched forward or back, the amount of pitch is either added to or decreased from the entry speed. Following that the direction of velocity is calculated by lerping from the vehicle's forward vector towards its down vector at a speed of *vehicle speed factor * DeltaTime.*
Finally, a new vector is created, which has the value *velocity direction * falling speed.* This is then set as the vehicle's new velocity.

4.9 Race Progression

This system is done entirely through blueprint, with the exception of the vehicle components, which are done through C++. It consists of 3 blueprint classes, one for checkpoint, one for start line and one for race manager. A new array in the vehicle class is also created to store the checkpoint the vehicle has hit during each lap.

First the checkpoints are placed along the track in such a way that it is impossible not to drive through them. Each checkpoint is a duplicate of the same blueprint class. They all share the same model, but all have a unique ID. In the blueprint a OnComponentBeginOverlap node is used to check when a vehicle collides with the checkpoint. This triggers a function defined in the race manager class to which it sends the ID of the checkpoint hit, as well as the vehicle that hit it.

The following figure shows checkpoint blocks placed along a track in such a way that the player is forced to drive through them. Normally in game they are invisible.

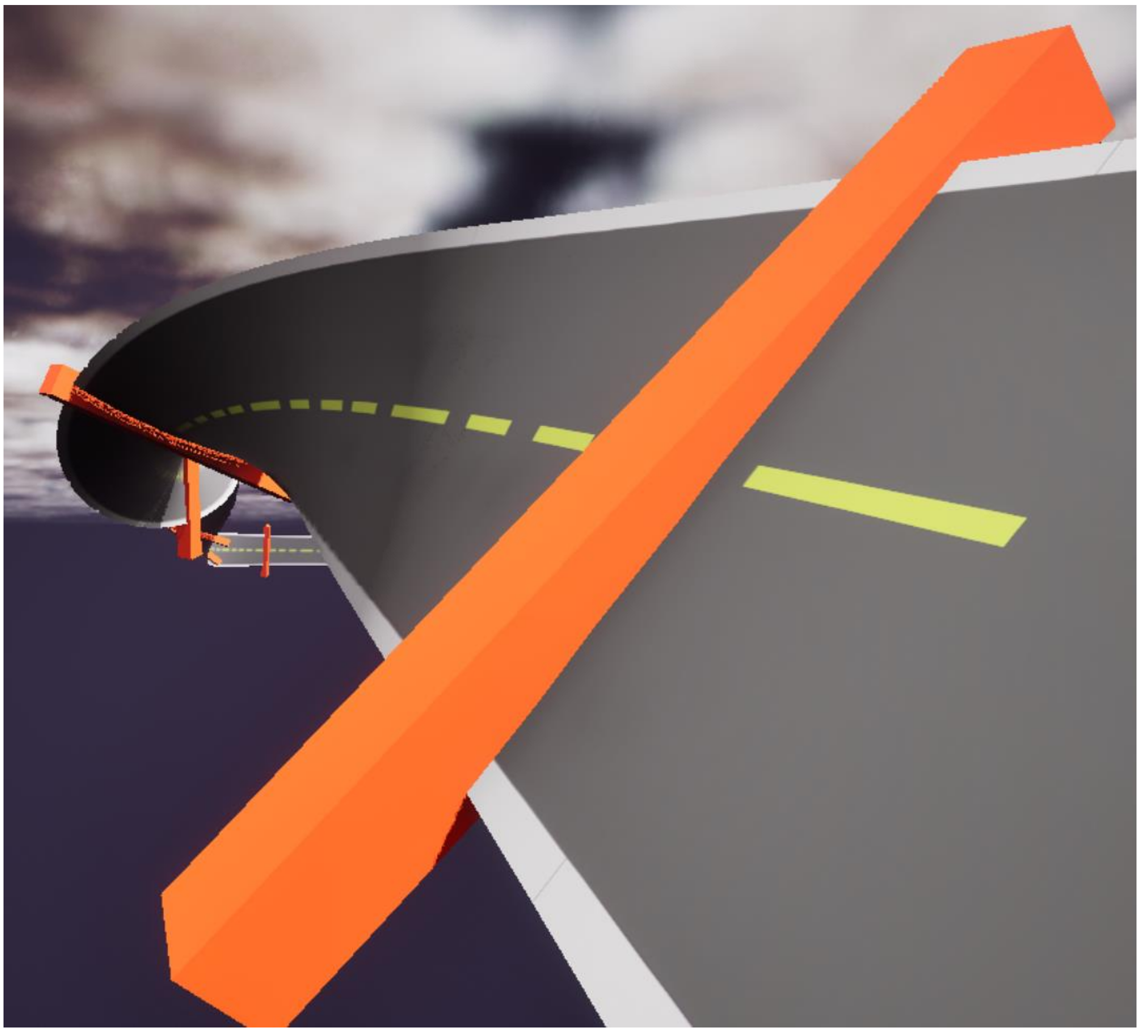


FIGURE 12. Checkpoint blocks.

The race manager contains an array of all the racers in the game as well as an array of all the checkpoints. With a BeginPlay node the race manager retrieves all actors with the checkpoint and vehicle classes and populates their respective arrays.
The function mentioned in the checkpoint part is then defined. It first retrieves the vehicle that hit the checkpoint, then retrieves that vehicle from the array of vehicles. After that it retrieves the vehicle's checkpoint array and adds the ID of the checkpoint that was hit to that array. The ID is added with an Add Unique node, to prevent the same checkpoint being stored multiple times.

Finally, the start line blueprint, similar to checkpoint is triggered through OnComponentBeginOverlap node, which also triggers a function defined in the race manager, which receives the vehicle that hit it. It then retrieves the checkpoint array of the vehicle and compares it to its own array of checkpoints in the game. If the arrays are identical, a lap is added to the vehicle. The laps are done through a lap variable, which is initiated as 1 in the beginning of a race and then incremented as the vehicle completes laps.

5 FURTHER DEVELOPMENT

At this point in development all the games basic systems are working as intended, some refining and balancing needed, but a solid basis for further development. Following this thesis, development will continue to get a demo out and eventually a full version. The first priority when beginning demo development is refining and bug fixing all the systems already developed. Following that, AI opponent and split-screen will be developed for more robust testing. After that work will begin on more advanced movement systems and game content like tracks and characters.

This chapter will go through some of the currently planned systems for the full version of the game. Gameplay impacting systems include an energy meter. This meter will deplete when the vehicle boosts, but it also acts as the health bar, meaning using all energy to boost and crashing will have fatal consequences. Other movement abilities that use energy are also planned. The vehicles will consist of multiple parts, which can be broken through crashing or being crashed into by other racers. Breaking a part will affect the vehicle's stats in some manner.
The game will have other racers, be they AI opponents of varying difficulties or other players. The number of racers per race is going to be 30.

For gamemodes, there will be multiplayer, both online and offline 4-player splitscreen. For online the number of human players in a race is yet to be determined but will not be the full 30. Besides multiplayer, there will be a grand prix mode, where players will play back-to-back through a number of tracks with some sort of reward at the end for winning. Time trial, where the intention is to complete races as fast as possible. Story mode, which will consist of hand-crafted missions. Practice mode, where players can learn the basics of the game and track layouts. Finally a career/arcade mode, which consists of completing races with unique modifiers and objectives.

The content will consist of 30+ unique racers. Undetermined number of tracks and track themes. A gallery, where players can read about the games characters and world. A track editor that players can use to create custom tracks.

6 CONCLUSION

The purpose of this thesis and the prototype developed was to have a working base for the game, from which demo development could be started and to give the reader a good understanding how the various systems of the game work and the design purpose of them.
The prototype side of things I think was a success. I managed to develop all the systems I set out to do in a fairly short amount of time and those systems are doing everything they should be, with only some refining needed.

The amount of work that I had to do in the span of roughly a month was a lot. Prior to starting this thesis, I had mostly just done design, some 3D modeling and had roughly plotted out how to develop some of the systems. The movement systems were fairly easy to develop and didn’t take a lot of time, whereas the game physics, especially the various parts of the vehicle rotation took the bulk of the development time.
The subject of the thesis I thought was very interesting, which makes sense, since the reason for choosing this as the subject was the fact I was already developing this game independently of my studies, for my company.

I learned a lot about game development and especially Unreal Engine throughout the course of this project, a lot of which I can immediately apply when I move on to demo development.

REFERENCES

1. Bryant, Paul 2002. Web archive F-Zero press conference. Search Date: 17.4.2024. https://web.archive.org/web/20070929155224/http://www.gaming-age.com/news/2002/3/28-106

2. Gerstmann, Jeff 2003. Gamespot F-Zero GX review. Search Date: 16.4.2024. https://www.gamespot.com/reviews/f-zero-gx-review/1900-6073623/

3. Epic Games. Unreal Engine documentation page. Search Date: 20.3.2024. https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-programming-and-scripting?application_version=5.0

4. Epic Games. Unreal engine licensing. Search Date: 20.3.2024. https://www.unrealen-gine.com/en-US/faq

5. Savage, Nina 2024. Unity market share. Search Date: 20.3.2024. https://www.gameslearning-society.org/what-percentage-of-indie-games-are-made-with-unity/

6. Epic Games. Unreal engine multiplayer support. Search Date: 20.3.204. https://dev.epicgames.com/documentation/en-us/unreal-engine/networking-and-multiplayer-in-un-real-engine?application_version=5.0

7. The Blender Foundation. Blender.org about page. Search Date: 20.3.2024. https://www.blender.org/about/

8. Epic Games. Unreal engine documentation. Search Date: 4.4.2024. https://dev.epicgames.com/documentation/en-us/unreal-engine/enhanced-input-in-unreal-engine

9. Epic Games. Unreal engine documentation. Search Date: 16.4.2024. https://docs.unrealen-gine.com/4.26/en-US/InteractiveExperiences/Framework/Controller/PlayerController/

10. Epic Games. Unreal engine documentation, Lerp. Search Date: 16.4.2024. https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/Math/Float/Lerp?ap-plication_version=5.3

11. Epic Games. Unreal engine documentation, physics. Search Date: 17.4.2024. https://dev.epicgames.com/documentation/en-us/unreal-engine/physics-in-unreal-engine

12. G2 best physics engines in 2024. Search Date: 17.4.2024. https://www.g2.com/catego-ries/physics-engine

13. Epic Games Unreal engine view modes. Search Date: 17.4.2024. https://docs.unrealen-gine.com/4.26/en-US/BuildingWorlds/LevelEditor/Viewports/ViewModes/

14. Computer-graphics.se, LOD. Search Date: 16.4.2024. https://computer-graphics.se/TSBK07-files/pdf/PDF09/10%20LOD.pdf

15. Hamilton, William Rowan 1847. On Quaternions. Search Date: 16.4.2024.
https://www.maths.tcd.ie/pub/HistMath/People/Hamilton/Quatern2/Quatern2.pdf