

Seb Hall

Registration number 100390459

2025

Creating a Tutorial for a 3D Racing Game in Unity

Supervised by Dr Yingliang Ma



University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The main objective of this project was to create a tutorial for a 3D racing game in Unity. The style of racing game was chosen to be arcade-style, with mechanics taken from the Mario Kart franchise of games. The tutorial used step-by-step instructions, attempting to explain each concept in the most detail possible. The outcome is a long but comprehensive tutorial covering many aspects of not only creating a car controller, but a full game around it as well. Overall, this project is a valuable resource for game developers wishing to create arcade-style racing games in Unity, filling a gap in the current market of available tutorials.

Acknowledgements

Thank you to my family and friends for supporting me. Thank you as well to Dr. Ma for giving the opportunity to complete this project.

Contents

1. Introduction	7
1.1. Current Problem	7
1.2. Aim	7
1.3. Objectives	7
1.4. Scope of the Project	8
1.5. Structure of report	9
2. Background	10
2.1. Why is Unity suitable for this project	10
2.2. Mario Kart Preference over regular racing games	10
2.3. Raycast based physics	10
2.4. Tutorial structure	11
2.5. Literature review	11
2.5.1. Suspension physics using Raycasts	11
2.5.2. Vehicle Movement: Acceleration, Deceleration, Turning, and Sideways Drag	12
2.5.3. Drifting Mechanics	12
2.5.4. Particle Effects	13
2.5.5. Instructional design or technical writing for tutorials	13
2.6. Comparison to other racing tutorials	14
3. Preparation	14
3.1. Methodology	14
3.1.1. Tutorial creation process	14
3.1.2. Sourcing Assets	15
3.2. Design of the tutorial structure	16
3.3. Car controller – suspension core design goals	17
3.4. Drifting Mechanics core design goals	18
3.5. Camera system core design goals	19
3.6. Particle System core design goals	19
3.7. Target Audience	20
3.8. Self-evaluation	20
3.9. Success criteria	20

4. Implementation and Evaluation	21
4.1. Development of tutorial document and the tools used	21
4.2. Process of writing and formatting	21
4.3. Challenges in developing the tutorial	21
4.4. Implementation of Key systems	22
4.4.1. Raycast suspension	22
4.4.2. Drifting Logic	23
4.4.3. Checkpoints	24
4.4.4. Particles	26
4.5. Evaluation	27
4.5.1. Strengths of the tutorial	27
4.5.2. Weaknesses of the tutorial	28
4.6. How does it meet needs of target audience	28
5. Discussion/Evaluation	28
5.1. Reflecting on the challenge	28
5.2. Unity experience gained by this tutorial	29
5.3. Learned about explaining complex topics	29
5.4. What would be done differently?	30
5.4.1. Video tutorial	30
5.4.2. More concise	30
5.4.3. More advanced topics	30
5.5. Value of community resources	30
6. Conclusion and Future Work	31
6.1. Summarize project	31
6.2. Future work	31
References	33
A. Examples of work from the tutorial document	34
B. Permission message from MrFluffy to use his track in the game.	53
C. Assets provided for the tutorial user	53

List of Figures

1.	Text explanation of suspension from tutorial page 4-5.	22
2.	Suspension() code snippet from tutorial page 5.	23
3.	Drifting explanation and code snippet from tutorial page 11.	24
4.	Checkpoint explanation and Checkpoint.cs script from tutorial page 17.	25
5.	Drift particles explanation and code snippets from tutorial page 15.	27
6.	Hierarchy view of Ray Points GameObjects.	35
7.	Aligning the BR Ray Point with the wheel.	35
8.	Transform coordinates for FL, FR, BL, and BR Ray Points.	35
9.	Acceleration() code snippet.	36
10.	Deceleration() code snippet.	37
11.	CameraFollow2 script Inspector values.	37
12.	CameraFollow.cs code snippet.	39
13.	Drift related variables in carScript.	40
14.	Initial input check in Drift() method.	40
15.	lockedDriftDirection assignment logic.	41
16.	Logic for ending a drift and applying boost.	42
17.	Calculating driftCharge while drifting.	42
18.	Steering logic calculation while drifting.	43
19.	Full drifting logic with added forward force and relative torque.	44
20.	Drift() method showing particle activation and deactivation logic.	45
21.	Particle related GameObject and Color array variables in carScript.	45
22.	AdjustParticleColour() method logic.	46
23.	foreach loop for setting particle color in AdjustParticleColour().	46
24.	Particle enabling/disabling logic on drift end and in DriftSpeedBoost coroutine.	47
25.	Enabling DriftChargeParticles in Drift() method if charge > 0.8.	47
26.	DriftSpeedBoost() Coroutine logic.	47
27.	Checkpoint explanation and Checkpoint.cs script from tutorial page 17.	48
28.	GameManager script variables from tutorial page 18.	49
29.	GameManager Start() method from tutorial page 18.	49
30.	PlayerHitCheckpoint() method logic from tutorial page 19.	50
31.	HandlePlayerOutOfBounds() method from tutorial page 19.	51

32.	StartTimerAfterDelay() coroutine from tutorial page 19.	51
33.	GameManager Update() method for timer display from tutorial page 20.	52
34.	Permission message from MrFluffy for track usage from tutorial page 17.	53

1. Introduction

1.1. Current Problem

The current problem is that there is a lack of detailed and high level Unity tutorials teaching faster and more arcade-style car movement. There are basic and low-level tutorials about applying movement to a sphere object, or without proper drifting mechanics, but none that try to accurately mimic the physics and movement of games from the Mario Kart franchise.

Without a tutorial like this, it would be especially daunting to try and learn how to code arcade-style car movement. Having to learn how to use Unity at the same time as learning the physics required to implement this movement means the majority of people would find it too confusing to learn all at once, and decide to give up.

A tutorial like this would be particularly helpful to people who want to learn Unity through applied programming, diving straight into a project that they care about and want to make rather than slogging away making games they don't feel inspired by. This tutorial aims to bridge that gap for them – helping people understand the basics of Unity as well as the Physics behind programming the car's movement, and especially adding fun extras such as high level drifting mechanics.

1.2. Aim

The aim for this project is: To create a step-by-step tutorial teaching users how to create a 3D arcade-style racing game in Unity, with a focus on raycast-based car control, drifting, and creating relevant game systems around those.

1.3. Objectives

- Research and pick crucial game mechanics from the Mario Kart franchise of games to implement in this project.
- Learn and understand Unity to a high enough level to create a tutorial teaching others.
- Develop a prototype that implements the core Mario Kart car functionality, including drifting, turning and accelerating.

- To write a tutorial document covering the basic functionality of the car movement and describing in detail the steps and processes required to create them.
- To write in the tutorial document not only the basic functionality, but additional extra complementary features commonly found in Mario-Kart-style racing games.
- To structure the tutorial to build on each component – starting out with simple elements and creating more difficult details that build on top of core functionality learned earlier.
- Provide necessary assets to simplify obtuse elements of the tutorial.

1.4. Scope of the Project

The scope of this project is wide, as the tutorial aims to leave people with a full and playable game experience, therefore requiring not only detailed explanations of core mechanics but also more subtle systems that make a better game in the end.

The tutorial will teach people these mechanics:

- Basics of how to use Unity for first time users
- Applying a pre-made model to a GameObject
- Raycast-based physics for suspension
- Handling user input in accordance with the Mario Kart games
- Acceleration and Deceleration
- Game turning that feels satisfying
- Creating a camera that follows behind the player
- Drifting mechanics – sliding around corners and charging & releasing a drift
- Implementing common and visually appealing particle effects
- Adding a checkpoint/Lap system to prevent players from cheating
- Adding speed boosts and offroad layers to speed up/slow down the player

- Making a mini-map of a top-down view of the player's location
- Implementing basic sounds and controlling pitch
- Adding Post Processing effects

The tutorial will not cover:

- Advanced CPU Implementation
- Online Multiplayer
- Complex 3D modelling
- Scene Management

The project also uses a custom Mario Kart track, created for use in Mario Kart Wii by a fan, who has granted permission to use this custom track for this project. More information on this will be provided later in this report.

1.5. Structure of report

This report will be split into these sections:

1. Focus on the background context of why a Mario Kart-style game was chosen as well as any literature used to help in the understanding and creation of this project.
2. It will talk about the methodology and design of both the Car controller, the tutorial, and all of the extra content.
3. The process of implementing the tutorial and creating the game alongside it, as well as critically evaluating how the tutorial and game turned out.
4. Reflecting on the challenge of creating the tutorial, the knowledge gained and what to do differently next time.
5. Concluding by summarising the achievements project and talking about the tutorial would implement next.

2. Background

2.1. Why is Unity suitable for this project

Unity was the game engine required for the project, and it was a good choice as it is perfectly adequate for creating 3D games such as this. There are a wide range of tutorials teaching basics to car physics, and a lot of it is simple and easy to understand. It doesn't have the same complex rendering techniques that Unreal Engine games do, but those aren't necessary for a project focusing a car controller and building out a game around it.

2.2. Mario Kart Preference over regular racing games

The Mario Kart franchise was the main muse for this project. This is because of it's fast and frantic gameplay focuses more on fun rather than accuracy which is preferential over more realistic racing games such as the Forza or Gran Turismo series.

A Mario Kart style game was also chosen due to a lack of existing tutorials describing the process of creating physics and especially drifting mechanics in that style. Before this project was even an idea, an attempt was made to create a multiplayer racing game in the style of Mario Kart, but didn't make any progress due to lack of depth in available resources. This shows a clear gap in the market for these kinds of tutorials.

2.3. Raycast based physics

The choice for Raycast-based physics was one that was heavily deliberated over during development. Initially, the guideline for this project was to use Unity's built-in Wheel Colliders. However, these come with a lot of limitations. They are useful for calculating suspension, but for an arcade-style racing game, they are much too restricting. In the previous progress report, a method was discussed of using Wheel Colliders for Suspension, and then ignoring them for acceleration, deceleration and turning, instead simply applying forces to the Rigidbody. This method however brought up even more issues than just using Raycasts for suspension. An example of these issues is the car trying to climb up ramps. A wheel collider is designed to be as realistic as possible, so it would struggle and slow down trying to get up a ramp and get no air-time as it came back down. Since there are no wheels in a Raycast implementation, they just recalculate where the

“wheels” should be, as well as using their spring-like suspension to boost themselves up when they come in contact with a ramp. They also keep the current momentum, leading to the car flying off the ramp, which is the intended effect.

In addition, attempting drifting mechanics using wheel colliders would be a lot more difficult than using Raycasts. Implementing drag and turning manually means that it’s easy to edit those functions once drifting is started and need to change how they operate. It also teaches the user a better understanding of how these functions fundamentally work, which is a main objective of the tutorial.

2.4. Tutorial structure

The structure of the tutorial used several key elements. These included:

- Step by step instructions, to help break up complex tasks into smaller more manageable steps.
- Clarity, giving the tutorial an informal tone to help convey concepts easier than using specific jargon and confusing users.
- Visual aids such as screenshots of code, which help effectively communicate what is being described to the user.
- A logical flow, where each topic builds upon the last.

2.5. Literature review

This review will look at each aspect of the game and share a resource that helped in the development of it.

2.5.1. Suspension physics using Raycasts

The biggest resource that helped in the development of the suspension physics for this project was the YouTube Video “Making Custom Car Physics in Unity” by “Toyful Games”. The video explains how it’s better to rely on coding a core game mechanic like Wheels rather than relying on Unity’s built in WheelColliders, citing that it helped them maintain “full customization and control” as well as making debugging easier. That same amount of control was useful in this project as it gives users the freedom to

change the physics how they like, as well as making the implementation of drifting a lot easier.

In addition, the video does an excellent job of explaining how suspension force is calculated on a “rest distance” and a spring’s strength, as well as how a damping force is added to prevent the spring from bouncing. Both of these concepts are used in this project’s suspension implementation.

2.5.2. Vehicle Movement: Acceleration, Deceleration, Turning, and Sideways Drag

Once again, the biggest resource that helped craft these core concepts would be “Making Custom Car Physics in Unity” by “Toyful Games”. In the same video, they describe implementing acceleration, braking and sideways drag. Specifically, it mentions applying a sideways drag method, which applies force in the opposite direction of the car’s turning velocity, giving the car enough friction for it to only turn in the correct direction without slipping on the other axis. This was the main inspiration for the same SidewaysDrag method found in this tutorial.

In addition, the use of an Acceleration curve is not one found in Mario Kart, as acceleration is designed to be fast to get the player quickly back into action. However, this idea worked well for a turning curve that is implemented into the project, as the Mario Kart games do have tighter or wider turning based on the player’s speed.

2.5.3. Drifting Mechanics

The drifting mechanics are what make this tutorial unique, and as such were sourced from the Mario Kart games themselves. “Mario Kart Wii” was the main inspiration for the physics, with the intention that the project’s drifting felt as close to that game’s as possible. Unfortunately, there is no access to the game’s code, so there is no common consensus on how to replicate that kind of drift. The way drifting was analysed for this project was to watch footage of drifting in “Mario Kart Wii” and try to mimic what that drifting looked like. This resulted in a drift being an alternative type of turn where there is more of a definitive arc being applied, as well as having less sideways drift to give more of that slipping effect. Once the base idea was in place, values were tweaked and the gameplay between this project and “Mario Kart Wii” were compared back and forth until the drifting reached a satisfactory level of accuracy.

2.5.4. Particle Effects

The primary inspiration for the particle effects came from the YouTube video "Recreating Mario Kart Drifting in Unity". Ironically, the drifting mechanics shown in this video felt too tight and stiff, so weren't a factor in the creation of this project's drift mechanic. In addition, the use of a sphere instead of a proper car and raycast system made this video too simplistic. However, the particle effects made for the drift charge and the boost were perfect. They looked extremely similar to the particles found in the original Mario Kart games. The boost did look too realistic though, so changes were made to make this project's boost look more cartoon-like. The same applies with the charge particles, which were too big and bright in the video's version. Tweaks and changes were made to add similar designs into this project while still being unique.

2.5.5. Instructional design or technical writing for tutorials

When researching how to plan a tutorial, plenty of websites offering guides on video tutorials only gave extremely basic instructions and more described the literal process of setting up a video tutorial rather than how to structure a tutorial and how to keep people engaged.

The most helpful guides came from guides to creating software documentation, which is similar to a tutorial in its goals. Both "Software Documentation guide" by "Write the Docs" and "Writing Effective Documentation" by "Indiana University" were used as references.

The first piece of advice from "Writing Effective Documentation" is "be brief", which in this case does not apply. Developers writing documentation typically want a small explanation and an example in code. When writing a tutorial though, it's important to explain topic thoroughly so that the majority of people reading understand.

The rest of the advice from these two references are applicable to this project though. The advice to "be explicit and speak in plain language where possible" is one clearly echoed in the tutorial, which takes on an informal tone as to not alienate people reading. Often tutorials try to be too formal or use too much jargon and it ends up confusing readers, which is not the goal of this tutorial.

"Write the Docs" says to make sure a documentation is Skimmable, which isn't entirely possible due to the length of each explanation, but a balance is achieved using frequent bold lettering. This helps to make the key information stand out, so a user just

skimming the tutorial still picks up on what's going on.

Both references say to provide examples to follow your work, and this is achieved in the code snippets alongside each explanation, providing a helpful visual for understanding the topics covered.

2.6. Comparison to other racing tutorials

A lot of other video tutorials tend to focus on physics being as realistic as possible and as simple as possible as they fight for the shortest tutorial on the internet. Some are only 5 or 6 minutes.

This tutorial differs. It has a clear vision of what it wants to be, and it being in written form means there's no pressure to fit under a certain minute mark. In addition, the clear traffic light system allows users to skip the parts they're not interested in, which is uncommon for other tutorials.

The biggest difference this tutorial makes is its implementation of Mario Kart-style physics and mechanics. Only one video tutorial makes a proper attempt at teaching Drifting like the Mario Kart games, and it's Mix and Jam's "Recreating Mario Kart's Drifting", which was critiqued in the Literature review for its simplicity. No written tutorials were found for this type of project either. Therefore, this tutorial stands as one of two tutorials on recreating the drifting from the Mario Kart games, making this very unique.

3. Preparation

3.1. Methodology

3.1.1. Tutorial creation process

The process of creating a tutorial was broken down with one idea in mind: implement the component that the player can test out next. For example, one could implement acceleration and turning first, but it won't work until suspension is added. Therefore, the best place to start building the car mechanics is with suspension, since everything builds on top of that. The same applies with drifting: It won't work without turning controls in place.

Before that though, the choice was made to start with a basic overview of creating a Unity project and an explanation of all of Unity's windows, to make sure every possible person following the tutorial is on the same page. Some might be using Unity for the first time, and this has been accounted for.

The latter half of the tutorial chooses to focus on features outside of the car controller, such as checkpoints, sound effects, a minimap and post processing. These help to make the game more complete, even if they shy away from the initial brief. These are seen more as optional objectives, so that if people are only following the tutorial for the car physics, they can stop there.

One of the main focuses was on an extreme level of detail for the tutorial. This project began with only a basic familiarity with Unity or Car physics; therefore, the tutorial was designed with that in mind: What somebody who knew nothing about Unity or Car physics would want to know in the creation of this. Explaining why specific lines of code are written rather than just telling the user to write them. This way, the player learns fundamentally why everything about the project works and therefore can replicate or build off them as they work on new projects.

As for the level of detail of code, the focus was on going a step above the already available resources. There are two parts of this: Width of the code and Depth of the code.

The width of the code is very wide, meaning it covers a lot of bases. The tutorial wouldn't have much value if the player ended up with a normal-feeling car controller like all the others just driving around an empty canvas. If the player follows all of the steps of this tutorial, they end with a car that can drift on a fully playable course with proper lap detection and Out of Bounds detection. The code also covers sound and respawn locations.

As for the depth of the code, having such a wide range of topics means that not all the code can be deep and detailed. Therefore, the focus was made on making sure the core mechanics of the car were as detailed as they needed to be, and the extra topics were functional and displayed an example of how they worked.

3.1.2. Sourcing Assets

The priority for this project was on the code, and the tutorial for the car model. As a result, the majority of the main assets were sourced from other locations.

The Car model used in the project is sourced from a model creator named Kenney, specifically their toy-car kit. This gave a wide variety of cars to choose from. The cars are low poly with simple textures, which may seem out of place compared to the quite detailed course design. This was an intentional design decision though, as it helps the car stand out from the rest of the course. The player can always pick the car out of a scene even when moving quickly, which is a crucial visual design requirement.

As mentioned in the Introduction, this tutorial uses a Custom Mario Kart track called Quaking Mad Cliffs, created by a modder named MrFluffy. This track was selected because a lot of conventional racetracks don't seem fitting for this project's more arcade-style physics. They often have tight turns and narrow tracks with no interesting changes in elevation. The ideal scenario would be to be able to use official tracks Mario Kart franchise, as they are polished, tested, and designed for these kinds of physics. Unfortunately, the use of these is prohibited, and there is no feasible way to get permission from Nintendo to use them.

One of the best things about Mario Kart Wii though is that it has one of the biggest modding communities in the world, with thousands of custom courses being made by fans for the game. A lot of these are on par or even better than real tracks in the game. These are not officially licenced by Nintendo, and therefore the fans who create them are the licence holders. These tracks are also free, since selling work created for Mario Kart Wii that is sold violates Nintendo's End User License Agreement. Therefore, if permission is granted to use one of these tracks for this project, it is legally permissible to use it in this project.

The track of choice is *Quaking Mad Cliffs*, by MrFluffy. It has a unique colour palette, interesting elevation and shortcuts, all of which match up well with this tutorial's design and physics. Permission to use the track was granted by the creator following direct communication.

3.2. Design of the tutorial structure

The tutorial was broken down into chunks, each chunk being an important part of the game. Some examples of these chunks are Suspension, Acceleration, Drifting and the GameManager.

One design method implemented in the tutorial was using emojis to signal whether a project was important or not. These emojis were as follows:

- — CRUCIAL: Without implementing these steps the project fundamentally will not work.
 - Examples: Suspension, Turning
- — IMPORTANT: Not absolutely necessary, but these steps are what make the tutorial unique.
 - Examples, Drifting, Particles, Course, Checkpoints
- — OPTIONAL: Extra bonus features to make the game feel more complete
 - Examples: Unity Basics, Mini-map, Post Processing

Since this tutorial is long, at over 9000 words, not every user is going to follow every word. This way they can pick and choose what they want to implement, without missing a vital piece of code that breaks the whole system.

One challenge when designing a tutorial is figuring out the placement of code blocks compared to text. One point of reference used when designing these is the tutorials found for lab sessions at the University of East Anglia, which always describe the code above a code block, before cutting off a whole section of the page in order to display it. No other words appear on either side of the image. From having completed many of these before, the change was made in this tutorial to have the code on the same Y axis as the text. This is because a lot of the text in this tutorial is used to explain to the user why code works the way it does. If the user has to scroll to find the code being described, they are less likely to read and understand the text as they then have to flick back and forth. By having the code right next to the text, this issue is eliminated.

3.3. Car controller – suspension core design goals

These are the design goals of what the user following the tutorial is meant to learn and understand about the car controller's suspension, and how they are broken down and explained for the user:

Goal 1: For the user to understand why raycast suspension is better than WheelColliders for this arcade-style controller

Before Raycasts are implemented the choice is justified over WheelColliders, with the explanation that it allows more freedom in movement and

physics design. The purpose of specifically Raycasts in this project is also taught, explaining that they calculate the distance between a point and the ground.

Goal 2: For the user to understand the core physics concept (Hooke's Law: $F=kx$) and how it applies to creating spring-like suspension with raycasts.

The variables used in the equation are all introduced before implementing the main logic, explaining how each of them relates to the code they are about to write. (springCompression for X, springStiffness for K, resulting in springForce in F).

Goal 3: For the user to be able to implement and replicate a basic suspension function that makes the car rest realistically on "wheels" using raycasts to apply upward forces once it detects ground.

Step-by-step calculations are shown, from raycasting to find the ground, to the currentSpringLength and springCompression, the springVelocity and the dampeningForce, and then applying the net of those forces to the car. Detailed explanations of these help the user to gain a fundamental understanding of these concepts so they can recreate in their own projects as well as this one.

3.4. Drifting Mechanics core design goals

Goal 1: For the user to learn how a drift locks in one direction once the player starts drifting, allowing them to change the tightness of the drift.

The use of lockedDriftDirection explains to the user something that they might have known but not thought about, which is that you can't change direction when drifting in Mario Kart-style games. This will help them take a closer look and re-evaluate similar game design concepts for the future.

Goal 2: For the user to learn about the charging up and releasing of a drift, including the case variables alongside that.

This is a core mechanic in the Mario Kart games, and the user learns about building up drift force overtime, and applying certain thresholds that the

charge needs to meet E.g. if it's under a certain amount the player gets no boost, and what colour the spark particles should be when a player hits a certain amount of drift. The player also learns about the speedboost applied based on that drift charge, and the boost particles appearing alongside them, which they can replicate and apply in their own projects.

3.5. Camera system core design goals

Goal 1: For the user to understand that setting up a following Camera is easy.

A lot of camera tutorials and resources write a lot of code for the camera module, with lots of algorithms and edge cases. This camera tutorial sets to teach the user that a simple and understandable Camera script at a high standard is easy and less daunting than the user may think.

Goal 2: For the user to learn about implementing a smooth following camera system

This camera system is also designed to smooth it's position from one place to another with Vector3.Lerp, which helps add a degree of professionalism and class to the camera. The user will see the impact it has on the camera and learn that it's a useful tool to use in future projects as well as this one.

3.6. Particle System core design goals

Goal 1: For the user to understand why particle effects are not only visually appealing but also practical

The tutorial mentions particles such as the charge sparks (for when you charge up a drift to a certain amount) appearing letting the player know how much their drift is charged up. This is a useful visual indicator for the player, which will teach the user to use more of these in future projects.

Goal 2: For the user to learn how to create an effective particle controller in a project

The user will learn how to enable and disable particle effects as well as how to group particle effects under a GameObject, to be triggered or disabled on command. This saves time compared to enabling and disabling each particle effect individually.

3.7. Target Audience

The target audience for this tutorial is very open: The tutorial catches learners up to speed on the basics of Unity, which means anyone with basic programming knowledge can follow this tutorial. The tutorial doesn't cover any programming fundamentals, and doesn't explain how methods or variables work, so some coding experience is required here. The tutorial is mainly for learners looking for an extremely detailed comprehensive guide to creating a fun arcade racing game, as well as all of the extra details required to build the game out into a full experience.

3.8. Self-evaluation

At every step in the process, this tutorial was reviewed on two main factors:

1. How clear and easy is it for someone with no knowledge on the topic to understand what's written?
2. Has any steps been missed in writing and applying?

Some of the tutorial was written with future steps already implemented in the code, which occasionally overwrites other code. Testing and remaking the game helped to make sure every step was detailed in the tutorial.

3.9. Success criteria

In order for the tutorial to be a success, the user has to be able to successfully complete these criteria:

- Able to follow along with and complete all CRUCIAL — sections and have a car that can accelerate and turn.
- Able to understand why code is written rather than just writing it.
- Be comfortable with tweaking or changing values and expanding on foundational mechanics shown.
- Follow the tutorial with minimal frustration.

4. Implementation and Evaluation

4.1. Development of tutorial document and the tools used

In order to write the tutorial, Microsoft Word was used. It's a very reliable word processor that makes it easy to format a document and move images around while still looking professional.

For the code snippets, Visual Studio Code was used to edit and format the code, before using Windows' built in Snipping Tool to take screenshots of the code to paste into the word documents.

4.2. Process of writing and formatting

The workflow for writing the document was as such. Firstly, research would be conducted on common ways of executing this step in other tutorials. A combination of these versions would then be decided on. This version would be the fastest to implement, easiest to explain, all the while producing the desired fun-feeling physics was chosen. This version would then be programmed in Unity and Visual Studio Code, tested to see if it was functional, and the values would be tweaked until the gameplay felt right. Once all of this is confirmed, then finally screenshots can be taken of the code, and writing up the tutorial can begin.

4.3. Challenges in developing the tutorial

One of the main challenges encountered in the development of the tutorial was the code side getting ahead of the tutorial. Many times in development the Unity code would be worked on a step ahead of the tutorial code. This is because implementing the idea in Unity is easier than having to write a detailed guide on why each individual part works.

The issue with this was there were several times during code development where previously written code gets overwritten to add new variables or clauses. For example, when adding speed boosts and offroad support, the Acceleration() method needs to change to check if either speed boost's or off road's Boolean values are true. If they are, it means the player is on those layers, and the maxSpeed typically set in Acceleration now gets increased or decreased respectively. Writing about this in the tutorial would be especially tricky if the Acceleration method hadn't already been implemented, as the

new code would need to be removed so the user could write the original version without creating unnecessary variables for features they haven't implemented or may not want to implement.

4.4. Implementation of Key systems

4.4.1. Raycast suspension

Raycast suspension is one of the trickiest parts of the tutorial for the player to understand. Therefore, it is one of the most detailed explanations in the tutorial.

Suspension is found on pages 4 and 5. The screenshot shows the code snippet provided for the suspension method. Since this method is so large, the code is far below the rest of the explanation. The text is a brief part of the tutorial breaking down each line of the suspension method and going into detail on why the method works at calculating and applying the upward force. The tutorial talks about using Hooke's law, $F = k \cdot x$ in order to calculate what the force of the spring used in our suspension calculations. This shows the user how common physics laws tie into the creation of our car controller.

First, it loops through each of our Ray Points. Then it calculates the maximum potential length of the wheel. Next, it does a raycast from our point down to the ground, and checks if it's landing on ground marked as drivable. If it is, then the if statement works, and suspension physics can be calculated.

This equation is calculated using Hooke's law: $F = kx$. F is the Force from the spring, k is the spring stiffness, and x is the spring compression. We don't have any of these variables except spring stiffness, so we need to calculate them now.

First, it marks this wheel as touching the ground in our `wheelsGrounded` array. Then it calculates the current length of the spring by subtracting the length of the raycast from the radius of the wheel. Based on the spring's current length and how long it is supposed to be at, we can then calculate how the **spring compression**, the x in our equation. It's calculated as a fraction: the current length of the spring over the maximum length of the spring (`springTravel`).

Next, we calculate the F of our equation, the **Force** of the spring. Since we enter the Spring Stiffness (k) as a variable beforehand, we now have all we need to calculate F , simply by following the equation and multiplying k and x . This force pushes the car upwards when compressed, stopping the bottom of the car from hitting the floor. If the suspension goes beyond its length too, the car is pulled downwards, stopping it from unnecessarily leaving the ground.

Figure 1: Text explanation of suspension from tutorial page 4-5.

```
1 reference
private void Suspension()
{
    for (int i = 0; i < rayPoints.Length; i++)
    {
        RaycastHit hit;
        float maxLength = restLength + springTravel;

        if (Physics.Raycast(rayPoints[i].position, -rayPoints[i].up, out hit, maxLength + wheelRadius, drivable))
        {
            wheelIsGrounded[i] = 1;

            float currentSpringLength = hit.distance - wheelRadius;
            float springCompression = (restLength - currentSpringLength) / springTravel;

            float springVelocity = Vector3.Dot(carRB.GetPointVelocity(rayPoints[i].position), rayPoints[i].up);
            float dampForce = damperStiffness * springVelocity;

            float springForce = springStiffness * springCompression;

            float netForce = springForce - dampForce;

            carRB.AddForceAtPosition(netForce * rayPoints[i].up, rayPoints[i].position);

            Debug.DrawLine(rayPoints[i].position, hit.point, Color.red);
        }
        else
        {
            wheelIsGrounded[i] = 0;
            Debug.DrawLine(rayPoints[i].position, rayPoints[i].position + (wheelRadius + maxLength) * -rayPoints[i].up, Color.green);
        }
    }
}
```

Figure 2: Suspension() code snippet from tutorial page 5.

4.4.2. Drifting Logic

Drifting is one of the most important components in this car controller, as it's what sets this tutorial apart from the rest. It's found on pages 10-14 of the tutorial. The explanation of drifting here is different from the suspension calculations though. Instead of showing the full method and talking through the physics application of it, we instead break each part of the drifting method down into chunks.

The first chunk is adding all of the necessary variables. There's a lot of extra variables since a lot of time was spent tweaking values to find the correct balance for the game, and that process is simplified when variables can be changed from the Unity Inspector rather than just in the code.

The next chunk describes the process of starting a drift. This includes the initial idea of locking the drift to a certain direction, changing the steering controls to instead control how tight or wide the drift should be, and setting the drifting Boolean to true.

The chunk after that handles the stopping of the drift, with cutoff values if the player hasn't charged enough boost, setting the drifting Boolean value to false, applying a velocity change if the player has enough boost, and starting a coroutine to increase and

slowly bring back down the maximum speed.

The final chunk of the drifting method handles what the script should do while the player is drifting. This includes charging up the drift, calculating the current amount of steering the player has, adding extra force so that the car doesn't slow down when sliding on the incorrect axis, and adding an arc-like turn to mimic the drifting from the Mario Kart games.

Start by first reading in if the **spacebar is being pressed**. Unity already has a button for space called **Jump**, so we just read in if that's **pressed down**. We also so check if we're **not drifting** and we are **on the ground**. You can't start drifting if you're in the air.

```
1 reference
private void Drift()
{
    if (Input.GetButtonDown("Jump") && !drifting && isGrounded)
    {
```

If all three of those variables are true, then we set **drifting** to true and we start setting up for drifting. We want to **lock** which direction the car is currently turning for the drift. E.g. when the player presses **space** and **left**, the car should drift to the **left**, and the controls are **now** used for **how much** left turning the player wants. So, we set our new **lockedDriftDirection** variable to a **slight** amount depending on what direction is **currently** being **pressed**. If **no** direction is pressed, I've chosen to set it to drift **right** (notice how it's **steer input >= 0** rather than **> 0**) just to handle the **edge case**.

```
drifting = true;
if (steerInput >= 0)
{
    lockedDriftDirection = 0.4f;
}
else
{
    lockedDriftDirection = -0.4f;
}
```

Figure 3: Drifting explanation and code snippet from tutorial page 11.

After the main Drifting method is established, we then go and change our Turn() and SidewaysDrag() methods to have clauses for when the player is drifting.

Breaking down the drifting method into smaller and more digestible chunks helps the user to gain a greater understanding of why we are writing the code we write, to avoid them just copying out an obtuse method and just moving on.

4.4.3. Checkpoints

Implementing a checkpoint system in a car controller tutorial sounds like a feature that's too far out of the scope. However they are very easy to implement and help a lot in making the tutorial more of a complete game rather than just a car controller. Setting up, sizing, and applying the respawn anchors would have been a frustrating process to explain in a tutorial, so the decision was made to include the core gameObjects in the course prefab, and then let the players write the code to make them function.

This section is also broken up into parts, similar to the drift. However, the checkpoint system requires many small scripts to be added to different objects, which seemed like

a natural way to break the tutorial up. A brief explanation of what a checkpoint is as well is provided since not everyone may be familiar with the term and why they are so important.

Checkpoints 🟡

Firstly, what are checkpoints for?

Games such as **Mario Kart Wii** use a **checkpoint** system, providing **barriers** around the map that it **checks** if the player has hit. If the player has hit **all of** the checkpoints, then it counts as a **lap**. If the player has **missed** a checkpoint, then the game knows not to count a **lap**. This helps prevent **cheating**, as well as just driving **back** over the finish line and counting another **lap**.

Let's start by creating a **Checkpoint.cs** script. This is **really simple**. It stores two values in variables: the **CheckpointID**, and the **RespawnPoint**. This is useful when the car falls off, as it can go back to the **last** checkpoint.

```
using System.Collections.Generic;
using UnityEngine;

4 references
public class Checkpoint : MonoBehaviour
{
    1 reference
    public int checkpointID;
    2 references
    public Transform respawnPoint;
    //yup this is really everything
}
```

Each checkpoint (including the **FinishLine**) has a **child** **GameObject** called "**RespawnAnchor**". Add the **Checkpoint.cs** script to **each** of the checkpoints, and drag the **corresponding RespawnAnchor** into each "**Respawn Point**" slot.

Figure 4: Checkpoint explanation and Checkpoint.cs script from tutorial page 17.

The tutorial first focuses on a base checkpoint script, which is only used to give an ID to each of the checkpoints and to get the location of where the player should respawn if they fall off and this is the last checkpoint they hit.

The next part is creating a Game Manager, which is very common in Unity games. In here we add a timer, a lap counter, checkpoint collision handling, and out of bounds collision handling. It's one of the few times in the tutorial where methods are made before they are needed, since the checkpoint collision method has no way of being called yet. But since it reads in any necessary variables, or has them defined in the gameManager, no errors are thrown when making these methods early.

The final part of the checkpoint system is the PlayerLapTracker, which is attached to the car component. It has one main method, which is a collision handler for the car, and all it does is tell the game manager to trigger the methods we made earlier.

4.4.4. Particles

Implementing the particles was another strategic choice. This is because while the creation of particles is enjoyable, there are so many variables to change that it would have resulted in a large amount of work for the user, to follow all of the steps. The focus of the tutorial is always on the car controller, and the particles are a helpful visual indicator and a bit of flair, but never the main focus of the project. Therefore the decision was made to include them as prefabs to be added by the user, and the code for enabling them and disabling them to be written out.

There are 3 types of particles used here: Drift particles, for when the player starts drifting and smoke comes out the back out their car, Charge Particles for when the player is charging a drift, and Boost Particles, for when the player releases the drift and gets a speed boost.

These drift particles are added to the drifting script we already made. Small code excerpts that the player should recognize from just implementing the drift function are shown with changes to that existing function. On occasion, arrows are drawn to signify the differences between the new code and the code we already wrote.

Drift Particles 🟡

In the particles folder you'll find **DriftParticles**, **DriftChargeParticles**, and **DriftBoostParticles**. We want **DriftParticles** for now. Drag the prefab **underneath** the Car Prefab, so that it's a **child** of the Car GameObject. For this tutorial, I have already created all the particles required to save time, but feel free to have a go at making your own particles if you don't like mine.

Anyway, this is easy to add into our **carScript**. First, add the **driftParticles** GameObject as a **variable** at the top of **carScript**. Then in the **Start()** function, make sure it's set to **inactive** using **DriftParticles.SetActive(false);**

Now we alter our **Drift()** method with two easy changes. If the drift particles **aren't** active when we press jump, set them to **active**. When we finish pressing jump, set them to **inactive**. The particles inside this GameObject **play on awake**, so they start the second we set it to active.

Make sure to assign the **DriftParticles** GameObject in the **carScript Inspector**, and you now have drift particles!



```
if (DriftParticles.activeSelf == false)
{
    DriftParticles.SetActive(true);
}
else if (Input.GetButtonUp("Jump") && drifting)
{
    drifting = false;
    if (driftCharge <= 0.8f)
    {
        driftCharge = 0f;
    }
    if (DriftParticles.activeSelf)
    {
        DriftParticles.SetActive(false);
    }
}
```

Figure 5: Drift particles explanation and code snippets from tutorial page 15.

4.5. Evaluation

4.5.1. Strengths of the tutorial

The main strength of this tutorial as an instructional tutorial is its length. The tutorial is over 9000 words long, which should give an idea of both the width of topics covered, and the necessary depth of most important topics. Other tutorials are more niche and specific, and either try to go too far in depth (losing the focus of the user), or too shallow (providing only entertainment rather than making sure the user learns and understands the topics.) This tutorial strikes the perfect balance between the two. It teaches its user's important topics, but never dwells on them too long, and continually keeps things interesting.

Another lesser but still important strength is the tutorial's clarity. Having such a long tutorial means methods have to be used to keep the user's attention. One small but impactful difference is adding emboldened words to each sentence. The most important words are written in bold, meaning if a user is skimming the text they still understand what's going on and have a lower chance of missing a key detail to implement.

4.5.2. Weaknesses of the tutorial

The length of the tutorial could also be seen as a potential weakness. Having followed tutorials previously, a deterring factor in them is typically their length.

An hour-long video tutorial is a lot less appealing than a 10 or 20 minute video tutorial, since completing the task doesn't just take the length of the video. When the user needs to copy the code and understand the concepts, it ends up taking a lot longer. For a written tutorial like this, this logic still applies.

Users will be scared off by seeing the high word count, and it's one of the reasons that the traffic light emoji system exists. To tell users that not all of what's written here is mandatory, and that they can pick and choose what to implement. Despite this though, a lot of people will unfortunately not have time to follow a tutorial like this, meaning some users are alienated.

4.6. How does it meet needs of target audience

This tutorial does effectively meet the needs of the target audience, as it is a detailed, step-by-step guide from the basics of Unity to a playable arcade facing game with complex mechanics. This appeals to the target audience of users wanting a comprehensive project experience.

5. Discussion/Evaluation

5.1. Reflecting on the challenge

This tutorial was a difficult challenge to create, having to write such a long tutorial all the while keeping it interesting and engaging. One of the biggest challenges faced in this tutorial was trying to find the correct amount of depth. The approach taken of explaining each section in as much detail as possible lead to plenty of questions about whether the scope of this project was too large for what it was. Tenacity was required to make sure everything was completed all the while keeping a satisfactory level of depth.

In addition, there was difficulty finding sufficient resources on the topics. This project was started out with not none but minimal Unity experience, especially not in the creation of 3D Unity games. Having such a barebones starting point and then not being met with a large number of online resources was tough. Research found lots of simple

and basic Car controller tutorials using WheelColliders, or incredibly complex papers requiring a level of physics knowledge not owned. Especially for finding and creating drifting mechanics there were little to no resources.

These factors lead to a difficult but satisfying challenge.

5.2. Unity experience gained by this tutorial

From this tutorial, the key topics learned about Unity game development were:

- Raycasts
- Car physics (Suspension, Acceleration, Turning)
- Implementing Drifting mechanics
- Collision Triggers
- Game Managers
- Audio Sources and controlling pitch
- Particle Systems
- Importing Assets

5.3. Learned about explaining complex topics

From the creation of this tutorial, the main aspects about teaching and explaining topics learned were:

- Breaking components down to explain them
- Giving clear examples
- Focusing on explaining why things work
- Be as clear and brief as possible (If the project calls for it)
- Make sure the tutorial is Skimmable (understandable with a glance)

5.4. What would be done differently?

5.4.1. Video tutorial

One of the aspects of this tutorial that would be changed if repeated next time would be creating a Video tutorial instead of a written tutorial. Video Tutorials are the most common type of tutorial for Unity and other game development teachings. Therefore, this tutorial is likely to find a larger audience if it was a video tutorial. In addition, the screen is being recorded the whole time, meaning that code snippets and screenshots are not required, and speaking about the steps to take is much easier than having to write it out. These factors lead video tutorials to being the best type of tutorials.

5.4.2. More concise

Another aspect of this tutorial that would be changed if repeated would be the length of it. In this version, the goal was to make sure the user had a full understanding of why they were writing the code they were writing. This made the tutorial too long for most people to realistically follow though, so a more concise tutorial is likely to be more applicable. It also is easier to create.

5.4.3. More advanced topics

Once the car controller is created, drifting is implemented and the course is imported, a lot of the tutorial's content are aspects found at a higher level in other tutorials. Rather than including lots of these little details, it would be more valuable to include another large but uncovered topic, such as multiplayer. There are plenty of multiplayer tutorials, but finding ones for specific frameworks as well as tutorials that use all free components is difficult. The creation of a different project led to this discovery, and it's a lacking factor that should be covered, and would work well in a racing game like this, since the Mario Kart games are known for their online support.

5.5. Value of community resources

Community resources are hugely valuable in work like this. Not every aspect of this game is able to be taught or created specifically for this tutorial, and therefore having pre-made assets such as Kenney's toy car model as well as MrFluffy's custom Mario

Kart track help the product to be finished and polished looking, as any other race track wouldn't have worked to anywhere near the same effect.

6. Conclusion and Future Work

6.1. Summarize project

This project achieved a comprehensive tutorial on implementing not only a complex car controller using Raycasts, but drifting similar to the Mario Kart style of games, as well as instructions for building a proper game around the car controller created.

The most important takeaways from creating this tutorial are:

1. There is a severe lack of tutorials explaining how to implement Mario Kart-style drifting, which hopefully this project helps with.
2. It's a tough balance to make a project with depth while still resulting in a complete end product.
3. Explaining knowledge which you know to be common sense having learned it already is difficult.
4. The structure and pacing of a tutorial are unbelievably important.

6.2. Future work

The tutorial is designed to also be just a starting point for a user to branch into a full racing game. It includes a "What's next" section at the end, with ideas for users to create next in the game. Here are a few of them:

- Creating a Title Scene to swap to the game scene from when the user hits "Play".
- Adding in another course and creating a screen to swap between them.
- Adding in Items like in the Mario Kart games, such as a Speed Mushroom or a Banana Peel.
- Add in AI racers to race against the player.

- Add in Online Multiplayer to create a lobby of other people before racing against them.

These ideas serve as building blocks to place on the foundation they've already made with this tutorial to create a fully fledged game they can be proud of.

One way the tutorial itself could be made better is with a Troubleshooting or FAQ section. Such a long tutorial means there's a chance players could make a mistake or copy something incorrectly. Testing this tutorial with plenty of people would give a good idea of where common pitfalls are in the tutorial, and those can be covered or answered in that section. For example, "have you set this variable in the inspector?".

References

- [1] Author: "Toyful Games"
Title of Work/Video: "Making Custom Car Physics in Unity"
<https://www.youtube.com/watch?v=CdPYlj5uZel>

- [2] Author: "Mix and Jam"
Title of Work/Video: "Recreating Mario Kart's Drifting"
<https://www.youtube.com/watch?v=Ki-tWT50cEQ>

- [3] Author: "Nintendo"
Title of Work/Video: "Mario Kart Wii"
https://en.wikipedia.org/wiki/Mario_Kart_Wii

- [4] Author: "Indiana University"
Title of Work/Video: "Writing effective documentation"
<https://ux.iu.edu/articles/writing-effective-documentation/>

- [5] Author: "Write the Docs"
Title of Work/Video: "Software Documentation Principles"
"Software Documentation Principles" by "Write the Docs"
<https://www.writethedocs.org/guide/writing/docs-principles/>

- [6] Author: "MrFluffy"
Title of Work/Video: "Quaking Mad Cliffs"
https://wiki.tockdom.com/wiki/Quaking_Mad_Cliffs

- [7] Author: "Kenney"
Title of Work/Video: "Toy Car Kit"
<https://kenney.nl/assets/toy-car-kit>

- [8] Author: "Unity Documentation"
Title of Work/Video: "Render Texture" (Used for the minimap)
<https://docs.unity3d.com/6000.1/Documentation/ScriptReference/RenderTexture.html>

A. Examples of work from the tutorial document

These are some segments from the tutorial to give an idea of how the tutorial is structured and works out.

This is a tutorial to create a 3D racing game in Unity. The racing game I have chosen to make is more arcade-y, and similar to Mario Kart. This is because of how easy those games are to play, and it's also what I grew up playing.

This is a LONG tutorial, so each section is marked with an emoji symbolizing how important the section is to the objective of “create a racing game in unity”

- 🚫 CRUCIAL: The game will not function without this! Listen up!
- 🟡 IMPORTANT: Usually what makes this tutorial unique. You should follow these.
- 🟢 OPTIONAL: Not that important but a nice touch

Implementing and explaining raycast system: 🚫

Let's talk about wheels. Usually, a project like this would use **Wheel Colliders**, which are Unity's built-in wheel physics for cars. However, these come with a lot of limitations. They are useful for calculating suspension, but for an **arcade-style** racing game, they are much **too restricting**. Especially the Mario-Kart drifting mechanic we plan to implement. Therefore, we need to calculate our own suspension in order to give ourselves the freedom to create this arcade-style movement.

We will do this by using **Ray Casts**, which calculate the distance between a point (in this case our wheels) and the ground. By making this calculation, we can then code the rest of the physics necessary to make the car work.

Lets start by making these Ray Casts then. Create an empty GameObject called Ray Points, and then add four child GameObjects of Ray Points: FL, FR, BL and BR. These are going to be the positions of each Wheel.

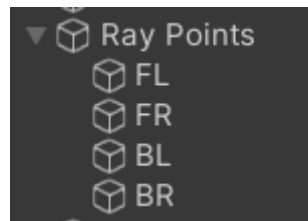


Figure 6: Hierarchy view of Ray Points GameObjects.

Now comes a process of aligning the **Ray Points**. For each of the child GameObjects we just created, align them with the top middle of the corresponding wheel. The one shown in the picture is BR, or Back-Right. Use the transform arrows to line this up with each axis. This is a slow process, but absolutely crucial for making sure the car's physics are accurate. Make sure the Y coordinates are the same for all the points, so that they have the same height. Also make sure that the Z coordinates for the front two wheels are the same, as well as the back two wheels.

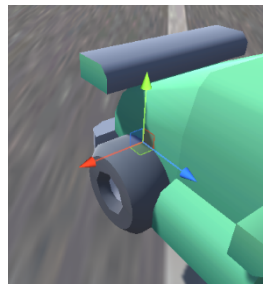


Figure 7: Aligning the BR Ray Point with the wheel.

If you're using the model included in the tutorial, then here are my Ray Point Coordinates, which should also work with yours.

FL	FR	BL	BR
Position X: -0.712 Y: 0.726 Z: 0.75	Position X: 0.5625 Y: 0.726 Z: 0.75	Position X: -0.712 Y: 0.726 Z: -0.75	Position X: 0.5625 Y: 0.726 Z: -0.75
Rotation X: 0 Y: 0 Z: 0	Rotation X: 0 Y: 0 Z: 0	Rotation X: 0 Y: 0 Z: 0	Rotation X: 0 Y: 0 Z: 0
Scale X: 1 Y: 1 Z: 1	Scale X: 1 Y: 1 Z: 1	Scale X: 1 Y: 1 Z: 1	Scale X: 1 Y: 1 Z: 1

Figure 8: Transform coordinates for FL, FR, BL, and BR Ray Points.

Finally, we need to add an **acceleration point**. This should just be in the middle of the car so that when we apply forces to the car, they are applied to the centre. Add an empty GameObject as a child of Car, and set it to 0.5 on the Y axis.

Movement —

It's finally time to add movement. Create a **Movement()** Script and add it to **FixedUpdate()** underneath **GroundCheck()**. In this movement script we first check if the car is grounded with an **if statement**, before calling an **Acceleration()** function and a **Deceleration()** function.

In our **Acceleration** function, we start by calculating the **velocity** we want the car to move at. We do this by adding the **current velocity** of the car to a combination of **adjustable variables** that allow us to **accelerate** the car. The first is a manual **acceleration** value, which we've set at **25** earlier. Next is our **moveInput**, which we know will be 1 if the button is pressed. Next is **transform.forward**, which makes sure the car's velocity is in the direction that the car is facing. Finally it's multiplied by **Time.fixedDeltaTime**, which ensures the acceleration is applied consistently.

Once we've calculated our **Acceleration** and stored it in a **Vector3**, we then use the **ClampMagnitude** function to make sure it doesn't go above our **maxSpeed**, which is **40**. We will increase the **maxSpeed** when we get a **drift boost**. Finally, we apply this new velocity to our car's **RigidBody**.

```
1 reference
private void Acceleration()
{
    Vector3 newVelocity = carRB.velocity + (acceleration * moveInput * transform.forward * Time.fixedDeltaTime);
    newVelocity = Vector3.ClampMagnitude(newVelocity, maxSpeed);
    carRB.velocity = newVelocity;
}
```

Figure 9: Acceleration() code snippet.

Now it's only right that we add the **deceleration** code right after, since it's very similar to the **acceleration** code. In our **Deceleration()** function, we use **carRB.addForceAtPosition**, since we don't need to **clamp** the **magnitude** of backwards movement. The deceleration is **too slow** to justify it. In this code, we're already adding on to the **existing force**, so we start with our pre-set **deceleration** value, then **moveInput**, which will be **-1**, then **-transform.forward**, which will be the **opposite** of the **direction** the car is facing, therefore **backwards**. We multiply it by **Time.fixedDeltaTime** again as well. We also need to specify here **where** we want to apply the force, which will of course be at our **accelerationPoint**, and then we specify what kind of force it is, which is **acceleration**.

```
1 reference
private void Deceleration()
{
    carRB.AddForceAtPosition(deceleration * moveInput * -transform.forward * Time.fixedDeltaTime, accelerationPoint.position, ForceMode.Acceleration);
}
```

Figure 10: Deceleration() code snippet.

Make sure **both** of these are in your **Movement()** function, and that the **Movement()** function is in your **FixedUpdate()** method.

Camera —

We've now got some movement and turning in our game, but they're no use if the **camera** doesn't follow the player around.

Start by creating an empty gameObject called '**Camera**', and then drag the **Main Camera** that the scene comes with underneath this '**Camera**' gameObject. This lets the **Main Camera** be a **child** of that object, allowing us to **indirectly** move it's location and add forces **with ease**.

Add a script to our '**Camera**' gameObject called '**CameraFollow**'. This will tell the camera to **follow** the player around.

Thankfully this Camera system isn't too tedious to setup, so once we define our **Target**, **Offset** and a public variable that lets us change **how smooth** the camera is, we can write the rest of the code in a **FixedUpdate()** Loop.

The car is moving **all the time**, so every frame we need to update the **position** of the camera to match this. We start by first getting where we want the camera to **end up at**, which in this case is the **exact position** of the car **minus** the offset. The offset is **how far away** from the car we want the camera to be, on the **x, y** and **z** values. We want to be **behind** and **above** the car, so we adjust those accordingly. You can see the values I've used on the right, but **feel free** to tweak it to your preferences.

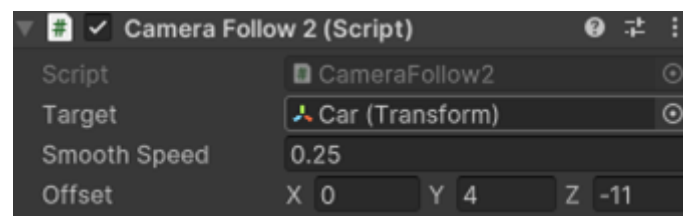


Figure 11: CameraFollow2 script Inspector values.

Next we calculate a **middle ground** between where the camera **currently** is and where it **needs to be**. We do this using **Linear Interpolation**, referred to here as **Vector3.Lerp**. We enter into this the position of the camera, the **desiredPosition** we calculated, and the **smoothSpeed**, which we define in the **Inspector**. The higher the **smoothSpeed**, the **faster** the camera will smooth between where it is and needs to be.

We then apply this **smoothedPosition** to the **current camera position**.

The camera now follows the player, but doesn't **rotate** with the player. Let's fix that. We only want rotation on the **Y axis** of the camera though, as **none** of the other axis are useful for camera control. We start again by getting the current **rotation** of the car, and then we get it's **eulerAngles**. This allows us to individually cancel out the axis we don't want, which is the **x and z axis**.

Now that we've cancelled out the car rotations we don't want on our camera, we can then **convert** the rotation back into a **Quaternion** which allows us to **apply** it. Similar to the position, we also apply **Linear Interpolation** to the rotation, making it **smoother** before applying it to the **rotation vector** of the Camera's **transform**.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
1 reference
public class CameraFollow : MonoBehaviour
{
    5 references
    public Transform target;

    2 references
    public float smoothSpeed = 0.125f;
    1 reference
    public Vector3 offset;

    0 references
    void FixedUpdate()
    {
        Vector3 desiredPosition = target.TransformPoint(offset);
        Vector3 smoothedPosition = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed);
        transform.position = smoothedPosition;
        //transform.rotation = target.rotation;

        Quaternion targetRotation = target.rotation;
        Vector3 eulerAngles = targetRotation.eulerAngles;
        eulerAngles.x = 0;
        eulerAngles.z = 0;
        Quaternion desiredRotation = Quaternion.Euler(eulerAngles);
        Quaternion smoothedRotation = Quaternion.Lerp(transform.rotation, desiredRotation, smoothSpeed);
        transform.rotation = smoothedRotation;
    }
}
```

Figure 12: CameraFollow.cs code snippet.

Make sure that you **set the transform** of the Car GameObject as the target with the **Inspector** before running it!

Try driving around on the terrain and see what you think. Feel free to adjust any values to your liking.

Drifting —

Now it's time for the most important part, implementing a **drift mechanic**! This will mean when the player hits the space bar, the car will go into a **drift turn** depending on the direction they **last pressed**. The **longer** the drift turn is, the **bigger** the **speed boost** when the player releases space.

```
10 references
private bool drifting = false;
11 references
private float driftCharge = 0f;
1 reference
public float maxDriftCharge = 30f;
0 references
public float driftBoostForce = 100f;
3 references
public float driftTurnMultiplier = 2f;
1 reference
public float driftTorqueMultiplier = 3f;
1 reference
public float driftSidewaysDragMultiplier = 0.1f;
12 references
private float lockedDriftDirection = 0f;
3 references
private Coroutine driftSpeedBoostCoroutine;
```

Figure 13: Drift related variables in carScript.

Keep in mind that this isn't the only way to implement drifting, or even the **best** way to do it. This is however a way that is simple and easy to understand, for the sake of the tutorial.

First, let's go back to our **carScript** and add some more drift related variables. See the image on the **right** hand side. We will use these later.

Now let's make a new method and call it **Drift()**. We handle the majority of the drift code from this method INCLUDING the **input** (spacebar), so we call this from the **Update()** method.

Start by first reading in if the **spacebar is being pressed**. Unity already a button for space called **Jump**, so we just read in if that's **pressed down**. We also so check if we're **not drifting** and we are **on the ground**. You can't start drifting if you're in the **air**.

```
1 reference
private void Drift()
{
    if (Input.GetButtonDown("Jump") && !drifting && isGrounded)
    {
```

Figure 14: Initial input check in Drift() method.

If all three of those variables are true, then we set **drifting** to true and we start setting up for drifting. We want to **lock** which direction the car is currently turning for the drift. E.g. when the player presses **space** and **left**, the car should drift to the **left**, and the controls are now used for **how much** left turning the player wants. So, we set our

new **lockedDriftDirection** variable to a **slight** amount depending on what direction is **currently being pressed**. If **no direction** is pressed, I've chosen to set it to drift **right** (notice how it's steer input ≥ 0 rather than > 0) just to handle the **edge case**.

A screenshot of a code editor with a dark background and light-colored text. The code is as follows:

```
drifting = true;
if (steerInput >= 0)
{
    lockedDriftDirection = 0.4f;
}
else
{
    lockedDriftDirection = -0.4f;
}
```

Figure 15: lockedDriftDirection assignment logic.

Add an **else if** statement on to our if statement that tracks when the button is **released** and we're **already drifting**. No ground check this time as you should be able to **finish** drifting in the air.

Obviously, we want to set our **Boolean** drifting variable to **false**. We will write the code of what the car does **while** drifting afterwards, so a lot of these values won't make sense at the moment.

Next, we set the amount of **charge** our drift has to **0** if it's under a certain amount. This is for the case where you **don't** drift for **long enough** to get a **speed boost**.

Now we add that **force** from the **drift charge**. If it's **0** then the force is **never** applied, but otherwise the the current amount of **driftCharge** is applied to the car's **rigidbody** by a **VelocityChange** instead of **Acceleration** this time, as it's not a **gradual** increase but an **instant** boost.

Next we have a **Coroutine** used to **damper** the speed the car goes when **boosting**. We want the player to get that boost and then **slowly** come down to normal speed. The **if** statement that **stops** that coroutine makes sure that it **doesn't** run **two consecutive coroutines** for slowing down at the same time. We then **reset** our variables and the cycle **resets**.

```
else if (Input.GetButtonUp("Jump") && drifting)
{
    drifting = false;
    if (driftCharge <= 0.8f)
    {
        driftCharge = 0f;
    }
    if (isGrounded)
    {
        carRB.AddForce(transform.forward * Mathf.Clamp(driftCharge, 0, 3) * 10, ForceMode.VelocityChange);
    }
    if (driftSpeedBoostCoroutine != null)
    {
        StopCoroutine(driftSpeedBoostCoroutine);
    }
    driftSpeedBoostCoroutine = StartCoroutine(DriftSpeedBoost(driftCharge));
    driftCharge = 0f;
    lockedDriftDirection = 0f;
}
```

Figure 16: Logic for ending a drift and applying boost.

For now that's the end of that if statement though. Let's make a **new** if statement underneath that handles what the car does **while it's actually drifting**.

We will come to change the **sideways drag** of the car while drifting, but other than that there's still lots of **friction** being applied to the car, since it's turning in an **unusual way**. Therefore, we need to add an **extra force** to **counteract** this friction while we're drifting.

We start by calculating the **current charge** of the drift. This is how much **power** it's built up that we wrote to **release** when the spacebar is released in the last if statement. We just add the **change in time** to it's **current value**, so that it increases **consistently**. We use **Mathf.min** to make sure it doesn't go over our **maxDriftCharge** value.

```
if (drifting)
{
    driftCharge = Mathf.Min(driftCharge + Time.deltaTime, maxDriftCharge);
}
```

Figure 17: Calculating driftCharge while drifting.

Next we calculate the **amount of steering** that the drift should have. We've already **locked** the direction that we're drifting in, so we need to calculate how **tight or wide** this drift will be, which the player is able to change based on which **direction** they're pressing.

The first part of the if statement **below** calculates if the player **isn't** pressing the same direction as they're drifting. This means we want the **widest** turn possible. We calculate this inside **lockedDriftDirection** as we'll use it instead of **steerInput** in our turn method later. We use **Mathf.Lerp** like we did when creating the camera script, as we want to make the drift turn as **smooth-looking** as possible. In this first part we're making the drift direction go to **plus or minus 0.1**, depending on whether **lockedDriftDirection** is **positive or negative**. This makes the turn as **wide** as we can go while still **executing** the turn. We then apply a **Time.deltaTime** call and multiply it by the **driftTurnMultiplier**, which allows us to **tune** the drift to apply that turn **slower or faster**.

If we do try and turn the **same direction** as **lockedDriftDirection** though, we call the **else** of this if else statement. The function is very similar except that we are trying to make **lockedDriftDirection** equal to the current **steerInput**, which will be **1 or -1** depending on the direction has been pressed.

```
if (drifting)
{
    driftCharge = Mathf.Min(driftCharge + Time.deltaTime, maxDriftCharge);
    if ((lockedDriftDirection > 0 && steerInput < 0) || (lockedDriftDirection < 0 && steerInput > 0))
    {
        lockedDriftDirection = Mathf.Lerp(lockedDriftDirection, 0.15f * Mathf.Sign(lockedDriftDirection), Time.deltaTime * driftTurnMultiplier);
    }
    else
    {
        lockedDriftDirection = Mathf.Lerp(lockedDriftDirection, steerInput, Time.deltaTime * driftTurnMultiplier);
    }
}
```

Figure 18: Steering logic calculation while drifting.

When we **do** start drifting, the car slides on angles that would **typically** slow it down. We **will** adjust the **sideways drag** to help with this, but to make sure that we **don't slow down** when drifting we add **extra forward force** to our car.

Then finally we do **carRB.addRelativeTorque** in order to make the drift more of an **arc like** motion. When the car drifts the **entire turn** is an **arc shape**, drifting out **wide** before finally moving the car in **close**. This torque ensures that the car **follows** that. We create a **driftTorqueMultiplier** so that we can **finetune** this arc to the right size.

```
if (drifting)
{
    driftCharge = Mathf.Min(driftCharge + Time.deltaTime, maxDriftCharge);
    if ((lockedDriftDirection > 0 && steerInput < 0) || (lockedDriftDirection < 0 && steerInput > 0))
    {
        lockedDriftDirection = Mathf.Lerp(lockedDriftDirection, 0.15f * Mathf.Sign(lockedDriftDirection), Time.deltaTime * driftTurnMultiplier);
    }
    else
    {
        lockedDriftDirection = Mathf.Lerp(lockedDriftDirection, steerInput, Time.deltaTime * driftTurnMultiplier);
    }
    carRB.AddForce(transform.forward * acceleration * 0.1f * moveInput, ForceMode.Acceleration);
    carRB.AddRelativeTorque(lockedDriftDirection * driftTorqueMultiplier * transform.up, ForceMode.Acceleration);
}
```

Figure 19: Full drifting logic with added forward force and relative torque.

The drifting tutorial continues on afterwards, this is just an example of what the tutorial looks like.

Drift Particles —

In the particles folder you'll find **DriftParticles**, **DriftChargeParticles**, and **DriftBoostParticles**. We want **DriftParticles** for now. Drag the prefab **underneath** the Car Prefab, so that it's a **child** of the Car GameObject. For this tutorial, I have already created all the particles required to save time, but feel free to have a go at making your own particles if you don't like mine.

Anyway, this is easy to add into our carScript. First, add the **driftParticles** GameObject as a **variable** at the top of **carScript**. Then in the **Start()** function, make sure it's set to **inactive** using **DriftParticles.SetActive(false)**;

Now we alter our **Drift()** method with two easy changes. If the drift particles **aren't** active when we press jump, set them to **active**. When we finish pressing jump, set them to **inactive**. The particles inside this GameObject **play on awake**, so they start the second we set it to active.

```
    if (DriftParticles.activeSelf == false)
    {
        DriftParticles.SetActive(true);
    }
}
else if (Input.GetButtonUp("Jump") && drifting)
{
    drifting = false;
    if (driftCharge <= 0.8f)
    {
        driftCharge = 0f;
    }
    if (DriftParticles.activeSelf)
    {
        DriftParticles.SetActive(false);
    }
}
```

Figure 20: Drift() method showing particle activation and deactivation logic.

Make sure to assign the **DriftParticles** GameObject in the **carScript Inspector**, and you now have drift particles!

You may notice by playing that it's difficult to understand when your drift boost is **charged** up without a **visual indicator**. If you've played the Mario Kart games before you know it's typically a set of sparks that build up next to your wheels as you drift. The **colour** of the sparks determines how much your drift has charged up. Let's implement those, as well as the boost particles to simulate a speed boost.

```
6 references
public GameObject DriftParticles;
7 references
public GameObject DriftChargeParticles;
7 references
public GameObject DriftBoostParticles;
3 references
public Color[] boostColours;
```

Figure 21: Particle related GameObject and Color array variables in carScript.

The process is very similar to implementing the drift particles. Add all of these variables at the top of our carScript (you should already have the **DriftParticles** GameObject), and set **DriftChargeParticles** and **DriftBoostParticles** to be **false** in the **Start()** method, just like we did with **DriftParticles**.

Now we make an **AdjustParticleColour()** Method. We will put this in the **FixedUpdate()** loop. We start by creating a **temporary colour** we will apply to the drift charge. We want to change the colour based on **how long** the player has been **drifting** for, from

blue to **orange** to **purple**. We do this by checking for **purple** first, then **orange**, then **blue**. If somehow **none** of them trigger we set the colour to **white** as a **backup** case. We do it in this order as if the colour **should be purple**, the if statement for **orange** and **blue** will also be **true**, so we need to do **purple** first so the others don't trigger. We also set the **alpha** value to **1** so the colour isn't **transparent**.

```
1 reference
void AdjustParticleColour()
{
    Color colour;
    if (driftCharge > 2.9f)
    {
        colour = boostColours[2];
    }
    else if (driftCharge > 2.0f)
    {
        colour = boostColours[1];
    }
    else if (driftCharge > 0.6f)
    {
        colour = boostColours[0];
    }
    else
    {
        colour = Color.white;
    }
    colour.a = 1f;
}
```

Figure 22: AdjustParticleColour() method logic.

Now we make a **for** loop for each of the particles in the **driftChargeParticles** GameOb-
ject. We get the **main module** of the ParticleSystem and set the colour to whatever is
the **current colour**.

```
colour.a = 1f;
foreach (ParticleSystem chargeParticle in DriftChargeParticles.GetComponentsInChildren<ParticleSystem>())
{
    ParticleSystem.MainModule main = chargeParticle.main;
    main.startColor = new ParticleSystem.MinMaxGradient(colour);
}
}
```

Figure 23: foreach loop for setting particle color in AdjustParticleColour().

Back to our **Drift()** Method, we want to do two things here: 1. Enable the **Drift-BoostParticles** when the **driftCharge** is over **0.8** and we release jump. 2. Disable the charge particles, since the drift has ended.

In our **if (drifting)** statement further down, we want to enable the **driftChargeParti-
cles** if **driftCharge > 0.8** as well.

```
else if (Input.GetButtonUp("Jump") && drifting)
{
    drifting = false;
    if (driftCharge <= 0.8f)
    {
        driftCharge = 0f;
    }
    else
    {
        if (DriftBoostParticles.activeSelf == false)
        {
            DriftBoostParticles.SetActive(true);
        }
    }
    if (DriftParticles.activeSelf)
    {
        DriftParticles.SetActive(false);
    }
    if (DriftChargeParticles.activeSelf)
    {
        DriftChargeParticles.SetActive(false);
    }
}
```

Figure 24: Particle enabling/disabling logic on drift end and in DriftSpeedBoost coroutine.

```
if (driftCharge > 0.8f)
{
    if (DriftChargeParticles.activeSelf == false)
    {
        DriftChargeParticles.SetActive(true);
    }
}
```

Figure 25: Enabling DriftChargeParticles in Drift() method if charge > 0.8.

```
1 reference
private IEnumerator DriftSpeedBoost(float charge)
{
    maxSpeed = baseMaxSpeed + 10;
    float duration = Mathf.Clamp(charge, 0, 3) * 0.6f;
    yield return new WaitForSeconds(duration);
    while (maxSpeed > baseMaxSpeed)
    {
        maxSpeed -= Time.deltaTime * 9;
        if (maxSpeed < baseMaxSpeed + 5)
        {
            DriftBoostParticles.SetActive(false);
        }
        yield return null;
    }
    maxSpeed = baseMaxSpeed;
}
```

Figure 26: DriftSpeedBoost() Coroutine logic.

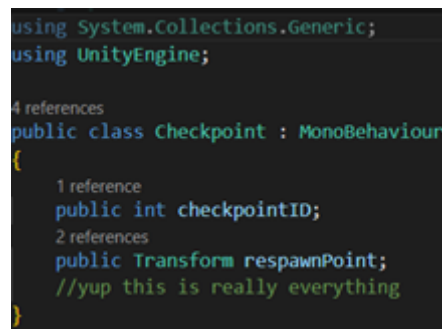
All we need to add now is the **disabling** of the **DriftBoostParticles**. This will be done in our **DriftSpeedBoost** method.

When the **maxSpeed** gets below the threshold we set, we disable the **driftBoostParticles**. This way the thrusters **stop** as the car slows down.

Don't forget to assign the **DriftChargeParticles** and **DriftBoostParticles** in the **carScript Inspector**, as well as our **Colours**! It doesn't matter about what specific colours you choose are, but the standard Mario Kart charge colours are **Blue -> Orange -> Purple**. So create 3 colours and assign them as you wish.

Checkpoints —

Firstly, what are checkpoints for?

A screenshot of a code editor showing the C# code for a Checkpoint script. The code includes using statements for System.Collections.Generic and UnityEngine, followed by a public class Checkpoint inheriting from MonoBehaviour. It contains two public variables: checkpointID (int) and respawnPoint (Transform), with a comment indicating that respawnPoint is used for everything.

```
using System.Collections.Generic;
using UnityEngine;

4 references
public class Checkpoint : MonoBehaviour
{
    1 reference
    public int checkpointID;
    2 references
    public Transform respawnPoint;
    //yup this is really everything
}
```

Figure 27: Checkpoint explanation and Checkpoint.cs script from tutorial page 17.

Games such as **Mario Kart Wii** use a **checkpoint** system, providing **barriers** around the map that it **checks** if the player has hit. If the player has hit **all** of the checkpoints, then it counts as a **lap**. If the player has **missed** a checkpoint, then the game knows not to count a **lap**. This helps prevent **cheating**, as well as just driving **back** over the finish line and counting another **lap**.

Let's start by creating a **Checkpoint.cs** script. This is really simple. It stores two values in variables: the **CheckpointID**, and the **RespawnPoint**. This is useful for when the car falls off, as it can go back to the **last** checkpoint.

Each checkpoint (**including the FinishLine**) has a **child** GameObject called **"RespawnAnchor"**. Add the **Checkpoint.cs** script to **each** of the checkpoints, and drag the **corresponding RespawnAnchor** into each **"Respawn Point"** slot.

Game Manager —

In order to keep track of the **checkpoints**, number of **laps**, and a **timer** for our race, we'll need a **GameManager**. Create a new GameObject called **GameManager** in the Hierarchy, and then add a script called **GameManager.cs** and start adding in the variables from the image on the right.

```
1 reference
public class GameManager : MonoBehaviour
{
    4 references
    public int totalLaps = 3;
    3 references
    public Checkpoint[] checkpoints;
    8 references
    private int currentLap = 0;
    5 references
    public int playerNextCheckpointIndex = 0;
    5 references
    public float timeElapsed = 0;
    3 references
    private bool raceFinished = false;
    1 reference
    public TMP_Text timeText;
    4 references
    public Transform car;
    5 references
    public Transform lastCheckpointRespawnPoint;
    // Start is called before the first frame update
```

Figure 28: GameManager script variables from tutorial page 18.

In our **Checkpoint[]** array, we enter all of our checkpoint GameObjects **IN ORDER** (**FinishLine**, **Checkpoint1**, **Checkpoint2**, **Checkpoint3**, **Checkpoint4**)

Our **Car** Transform is of course a reference to our car GameObject. Our **lastCheckpointRespawnPoint** Transform is in reference to the **Transform** of whatever checkpoint we last crossed, so we **don't** set this value in the **Inspector** like the rest.

Our start function is simple, we call a **Coroutine** called **StartTimerAfterDelay**, which is self-explanatory. We also set the **lastCheckpointRespawnPoint** as the starting line, so that somehow if the player drives off before then they still have a place to respawn.

```
0 references
void Start()
{
    StartCoroutine(StartTimerAfterDelay(1f));
    lastCheckpointRespawnPoint = checkpoints[0].respawnPoint;
}
```

Figure 29: GameManager Start() method from tutorial page 18.

Now we're going to make two functions that we call from a script we will make later. The first one is called **PlayerHitCheckpoint**, and it takes in the **playerID** (in case we ever implement **multiplayer**) and the **Checkpoint** we hit as arguments.

Initially we want to **return** and not run the method if we've already finished the race. Next, we get the **ID** of the checkpoint, so that we can check if the checkpoint we just hit is the one that is supposed to come next around the track. If it is, then we have a **log** statement just for checking that's the case, but mainly we set our **lastCheckpointRespawnPoint** to the **respawnPoint** of the checkpoint we hit. We also increment the **playerNextCheckpointIndex** so that it now checks for the next checkpoint.

```
1 reference
public void PlayerHitCheckpoint(int playerID, Checkpoint checkpointHit)
{
    if (raceFinished) return;
    int checkpointID = checkpointHit.checkpointID;
    if (checkpointID == playerNextCheckpointIndex)
    {
        Debug.Log("Player " + playerID + " hit checkpoint " + checkpointID + " correctly.");
        lastCheckpointRespawnPoint = checkpointHit.respawnPoint;
        playerNextCheckpointIndex++;
        if (checkpointID == 0 && currentLap >= 0)
        {
            currentLap++;
            if (currentLap >= totalLaps)
            {
                raceFinished = true;
                //if we had race end logic, we would call it here
            }
        }
        else if (checkpointID == 0 && currentLap == 0)
        {
            currentLap = 1;
        }
        if (playerNextCheckpointIndex >= checkpoints.length)
        {
            playerNextCheckpointIndex = 0;
        }
    }
}
```

Figure 30: PlayerHitCheckpoint() method logic from tutorial page 19.

If the **checkpointID** is **0** (The finish line) and this isn't the start of the race (where you start behind the line), then we **increment** the lap counter. If the lap counter is now the same as our **totalLaps** variable based on that, then we class the race as **completed**.

We also provide a specific check for when the player starts **behind** the finish line, and making sure to **reset** the checkpoint counter to **0** when we get **back** to the finish line.

Our next method is called **HandlePlayerOutOfBounds**, and we read in the **playerID** again as well. We quickly check that **no errors** will be thrown by this before moving the

car's **position** and **rotation** to match the **position** and **rotation** of the **lastCheckpoint's RespawnPoint**.

```
2 references
public void HandlePlayerOutOfBounds(int playerID)
{
    if (car != null && lastCheckpointRespawnPoint != null)
    {
        car.position = lastCheckpointRespawnPoint.position;
        car.rotation = lastCheckpointRespawnPoint.rotation;
        Rigidbody rb = car.GetComponent<Rigidbody>();
        if (rb != null)
        {
            rb.velocity = Vector3.zero;
            rb.angularVelocity = Vector3.zero;
        }
    }
}
```

Figure 31: HandlePlayerOutOfBounds() method from tutorial page 19.

We also reset the car's **velocity** and **angularVelocity** so that it doesn't carry any **momentum** from when it fell off.

We also have a small **IEnumerator** method that simply **waits** for the delay that we give it in the **Start()** Method, before **resetting** the clock and then increasing the **timeElapsed** variable in accordance with **Time.deltaTime**, which stores the time in **seconds** and **milliseconds**.

```
1 reference
private IEnumerator StartTimerAfterDelay(float delay)
{
    yield return new WaitForSeconds(delay);
    timeElapsed = 0;
    while (true)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }
}
```

Figure 32: StartTimerAfterDelay() coroutine from tutorial page 19.

Finally, we need an **update** loop. This will **dynamically** update the text we will make next.

We start by creating an **if** statement that only runs the code that changes the time IF the player **hasn't completed 3 laps** yet. This way it stops when the player **crosses** the finish line for the third time.

Most of this code is just converting the milliseconds in **timeElapsed** into **seconds** and **minutes**, to display it **properly**. Once we have that we display it in one long string with a **lap counter** underneath, displaying **how many laps** out of 3 we have done.

A screenshot of a code editor showing the C# code for the GameManager.Update() method. The code is as follows:

```
0 references
void Update()
{
    if (currentLap <= 3)
    {
        int minutes = (int)(timeElapsed / 60);
        int seconds = (int)(timeElapsed % 60);
        int milliseconds = (int)(timeElapsed * 1000 % 1000);
        string text = string.Format("{0:00}:{1:00}.{2:000}", minutes, seconds, milliseconds);
        timeText.text = text + "\n" + $"Lap: {Mathf.Min(currentLap, totalLaps)}/{totalLaps}";
    }
}
```

Figure 33: GameManager Update() method for timer display from tutorial page 20.

Let's quickly make that text we'll use to display this. In the Hierarchy, Right Click and create a new **UI Text** using **TextMeshPro** called "**TimerText**". Accept if it asks you to import **TMPPro** essentials. This should create a **Canvas** and an **EventSystem**. In our Canvas settings, set "**UI Scale Mode**" to "**Scale with Screen Size**". This means that our text is the same size no matter the resolution.

In **TimerText's Rect Transform**, click on the **square** in the top left to open the **Anchor window**, and then click on the "**top**" "**right**" anchor preset (**Not the stretched one!**). Set **PosX** and **PosY** both to **-20**, to give the text some **clearance** from the corner. Now edit the text settings however you like. Change the **font**, change the **size**, change the **outline** etc.

Once you are happy with it, drag **TimerText** into our **GameManager** script on our **GameManager** object. Don't forget to drag the **Car transform** in as well.

B. Permission message from MrFluffy to use his track in the game.

For reference, @green-man is my online username.

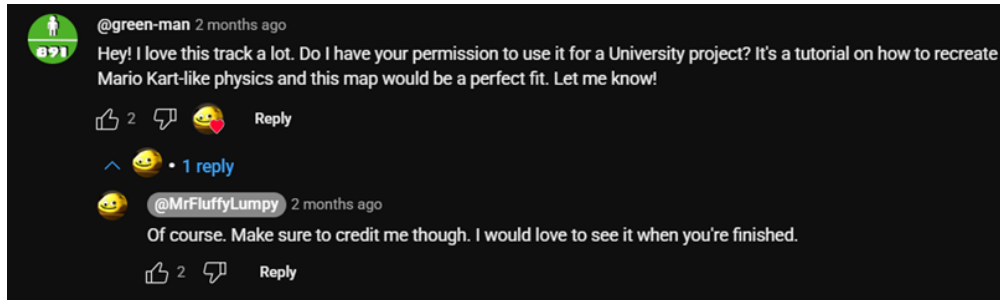


Figure 34: Permission message from MrFluffy for track usage from tutorial page 17.

C. Assets provided for the tutorial user

- A package containing the course Quaking Mad Cliffs, as well as checkpoints around the map with RespawnAnchors. It also contains an Out of Bounds Collider that resets the player when they go out of bounds.
- A package containing the Drift Particles, Drift Charge Particles and Drift Boost Particles.
- The Toy Car Model created by Kenney
- 3 Sound files (dirtDrift, driveSound, skidSound)
- A Cloudy Day Skybox Material