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"

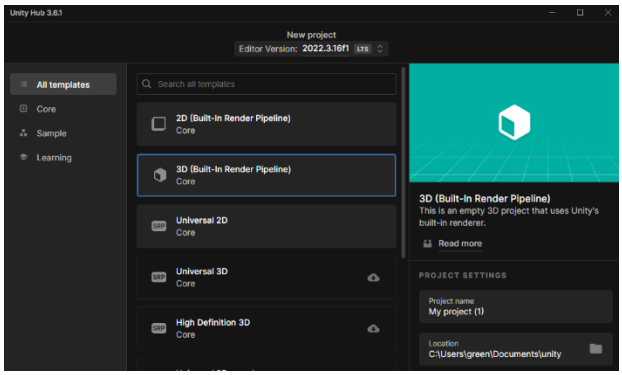
🔴 – CRUCIAL: The game will not function without this! Listen up!

🟡 – IMPORTANT: Usually what makes this tutorial unique. You should follow these.

🟢 – OPTIONAL: Not that important but a nice touch

## Unity Basics 🟢 (If you already understand them)

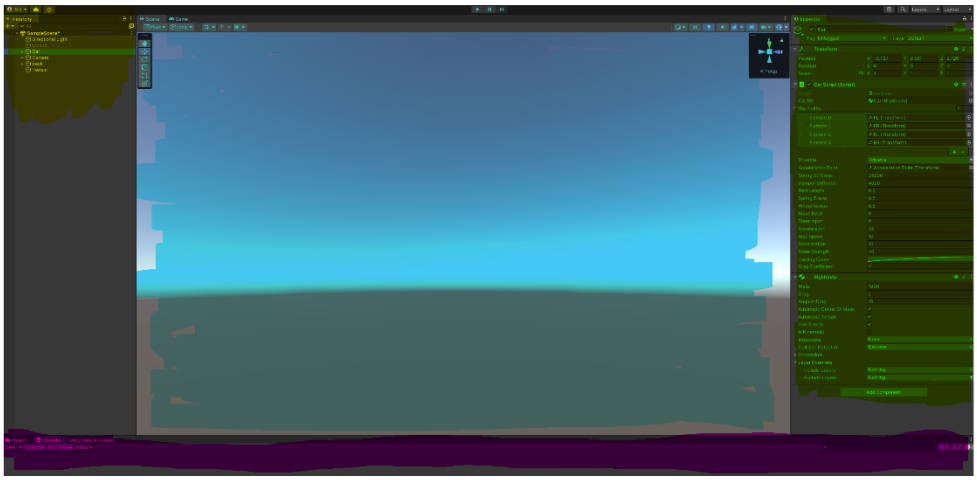
Let's start by making a project in Unity. Open the Unity Hub, and select "New Project", then 3D (Built-in Render Pipeline). This means we use Unity's pre-made process to render our 3D game into a 2D image on our screen, reducing the need to develop that ourselves.




Pressing create project will take us to the Unity Editor, where we can start to build the game.

I've highlighted a screenshot of the editor in order to explain each section of it:

- On the left (Highlighted in Yellow) we have the Hierarchy, where you can create and organize GameObjects, which are the building blocks of the game. Our car will be a GameObject, as will each wheel.
- On the right (Highlighted in Green) we have the Inspector, where you can edit properties of GameObjects. This includes adding Components, which allow us to define what we want from GameObjects. It could be Collision through the use of RigidBodys, or Wheel physics through the use of Wheel Colliders. In those components we can adjust individual values, as you can see in the screenshot.



- In the centre (Highlighted in Blue) we have the Scene Window, where you can place our created GameObjects into our game. The Scene Window also contains the Game Window, where by pressing play at the top, we can run our Game and view it running.

- Down at the bottom (Highlighted in Purple) we have the Project and Console bar, which Contains all the external files you may add into your game, such as Assets and Scripts. It also contains the console, which shows errors and is able to be printed to in code.

Adding a Model 🔴

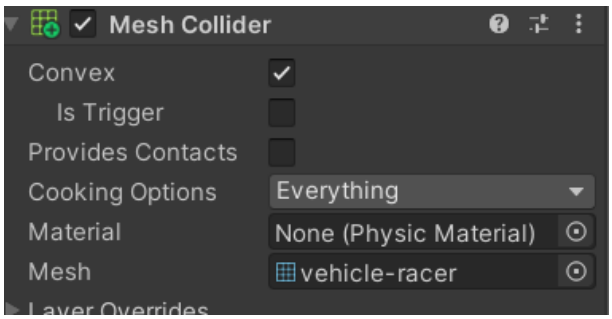
Let's start by adding a model to our Unity project. If you would like, you can add a different car model and as long as the wheels are in the correct places, the car should function exactly the same as ours. Download the model from wherever it's stored and drag it into the **Assets** folder of our Unity project. You can also just drag it into the Project Bar at the bottom of the Unity window.

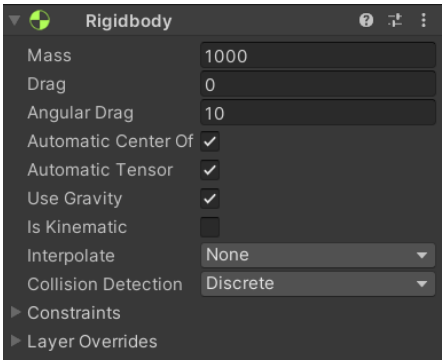
Our car model is a little small though, so let's increase the scale of it now before it messes up our physics later. Click on the model itself in the Car Model folder, and in the Inspector, set the Scale number from **1** to **3**, and click Apply down the bottom.

Making our Car GameObject. 🔴

First, for testing purposes, add a new Terrain by right-clicking on the Hierarchy, then pressing 3D Object, then **Terrain**. Change the Transform of the Terrain in the Inspector so that it's position is -500 on the X axis and -500 on the Z axis, so that 0,0 is in the middle of it.

Next, make an empty GameObject by right-clicking on the Hierarchy and pressing **Create Empty**. Name this GameObject Car and then drag our Car model on top of the Car GameObject. This will add the model into the world as a **Child** of the Car GameObject. It's best practice to not directly add physics to the Car model, so instead we add them to a **Parent** GameObject (Our *Car* GameObject). One change we will make to the Car model though is adding a **Mesh Collider**. This adds **collision** for our car body that's specific to our model, rather than just a box or a sphere. Click on the Car model, go to the Inspector and press Add Component. Start typing "Mesh" and it will appear for you to add. Once it's added, make sure **Convex** is checked, and leave everything as seen in the picture.




Now let's add a **Rigidbody** component. These are responsible for adding physics to an object such as ours. Add it as a Component of the **Car** GameObject, rather than of the Car model. Once you've created it, set the **Mass** to 1000 (Since this is a Car after all), and the **Angular Drag** to 10, and don't change anything else.

Our car now has basic physics, and if you run the game (press play at the top of the screen), our car should fall to the ground.

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

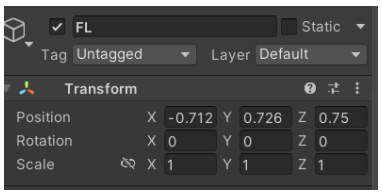
need to calculate our own suspension in order to give ourselves the freedom to create this arcade-style movement.

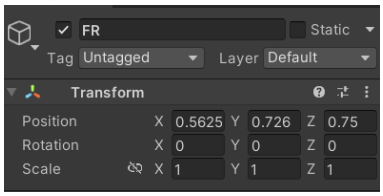
We will do this by using **Ray Casts**, which calculate the distance between a point (in this case our wheels) and the ground. By making this calculation, we can then code the rest of the physics necessary to make the car work.

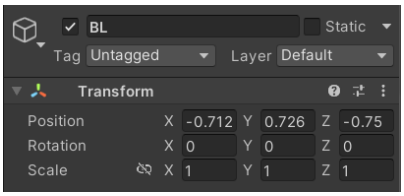

Lets start by making these Ray Casts then. Create an empty GameObject called Ray Points, and then add four child GameObjects of Ray Points: FL, FR, BL and BR. These are going to be the positions of each Wheel.

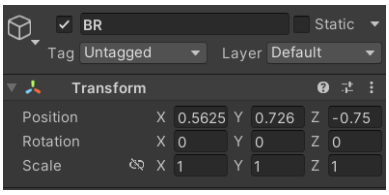
Now comes a process of aligning the **Ray Points**. For each of the child GameObjects we just created, align them with the top middle of the corresponding wheel. The one shown in the picture is BR, or Back-Right. Use the transform arrows to line this up with each axis. This is a slow process, but absolutely 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.

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




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.

## carScript

It's time to start coding! Right click in the Assets folder and create a **C# Script** called carScript. Open it up, and you'll be greeted by the default code layout.

First, lets add some variables. Underneath the public class carScript bracket, start making the variables in the photo. The **Rigidbody**, **Transform[]** and **Transform** variables are all to get GameObjects that we made earlier. The LayerMask is to define which surfaces are able to be driven on and which aren't.

```
16 references
public Rigidbody carRB;
11 references
public Transform[] rayPoints;
1 reference
public LayerMask drivable;
2 references
public Transform accelerationPoint;

1 reference
public float springStiffness;
1 reference
public float damperStiffness;
2 references
public float restLength;
2 references
public float springTravel;
3 references
public float wheelRadius;
```

The other variables here are typically calculated by the Wheel Colliders, so we need to add those manually now.

The **Update()** loop is a method that gets called every frame, and while this is useful for calculating

movement, calculating physics every frame doesn't work, since the game will run at different framerates for different people depending on their computer. Instead, we use the **FixedUpdate()** method, since it runs at a fixed time interval and therefore removes this issue.

Suspension 🔴

The code below is the method for calculating Suspension. Call this method by adding **Suspension();** into the FixedUpdate() Method. Let's talk about how this works.

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: $\boldsymbol{F = kx}$. **F** is the Force from the spring, **k** is the spring stiffness, and $\boldsymbol{x}$ 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 **wheelIsGrounded** 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**).

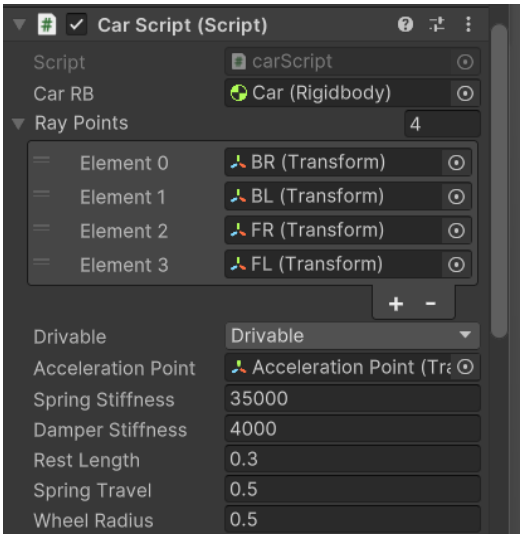
Next, we calculate the **F** of our equation, the **Force** of the spring. Since we enter the Spring Stiffness (**k**) as a variable beforehand, we now have all we need to calculate F, simply by following the equation and multiplying **k** and **x**. This force pushes the car upwards when compressed, 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.

Now we calculate the damping on this suspension. We do this by first calculating the **Spring Velocity**, which is how fast the suspension is moving **along** it's axis. We use the **GetPointVelocity()** as we can use it to get the velocity of only up or down movement for this wheel. Now we can multiply the **springVelocity** by the **damperStiffness** in order to get the **dampened force**, which stops the car from bouncing if the wheels move up and down too fast.

All that's left to do is add this force to the car. We calculate the **Net Force** by subtracting the **Spring Force** and **Damp Force**. Obviously since we're in a for loop, this is only for one of the wheels, so we add the **Net Force** upward at the position of this wheel, using **AddForceAtPosition**. It can also help to draw a Line to show the raycasts using **Debug.DrawLine**, just make sure to remove this code before the final version.

If the initial Ray Cast doesn't hit, then we mark this wheel as **not grounded** in our array, and we draw a green line for now, just to show that the Ray Casts aren't finding anything.

```
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);
        }
    }
}
```




Back into the Unity, and it's time to add in our values for the variables we made earlier. Drag the script into the **Car GameObject's** Inspector in order to add it as a component like the **Rigidbody**. Now, proceed to fill out it's variables as seen in the image on the right. Pressing the circle by a variable like **Rigidbody** lets you select the component you want, which is useful when dealing with a lot of components.

As for the **Drivable** layer, we need to make it in Unity before being able to select it. At the top of the Inspector, underneath the name of the GameObject, there is an option to change the **layer** the Car is on. Click on it, and press "**Add Layer**". Here, add a "**Drivable**" layer into User Layer 6, and then apply it to both the **Car Script** component and the Terrain **(Don't apply it to the Car GameObject though).**

Now if you run the game, the car should hopefully be resting on its "**wheels**"! If not, check over this document to see if you missed a step. Make sure to drag the camera to a place where you can see the car as well.S

Input 🔴

It's time to make the car drive. Go back into our code, and add a **GetPlayerInput()** Function to our **Update** Loop (**Not** Fixed Update!) Add the variables seen in the Image on the side of the screen to the top of our class before creating this **GetPlayerInput** function. We will get the **Inputs** from the player here. We do this by calling **Input.GetAxis("Horizontal")**, which gets the left and right axis of movement. Calling **Input.GetAxis("Vertical")** though takes the forward and backwards input as axis, depending on how far the stick is forward, or how long the key has been pressed, when in the "**Mario Kart**" games it's a button that is either **on or off**. Therefore, we will set up the movement to be either **1** (forwards), **-1** (backwards) or **0**. In Unity's **Input Manager**, set "**Fire1**" and "**Fire2**" to the W and S keys for now. They already correspond to the correct buttons on a gamepad.

```
8 references
public float moveInput = 0;
6 references
public float steerInput = 0;

2 references
public float acceleration = 25f;
9 references
public float maxSpeed = 40f;
4 references
private float baseMaxSpeed = 40f;
1 reference
public float deceleration = 10f;
```

```
//moveInput = Input.GetAxis("Vertical");
steerInput = Input.GetAxis("Horizontal");

if (Input.GetButton("Fire1"))
{
    moveInput = 1;
}
else if (Input.GetButton("Fire2"))
{
    moveInput = -1;
}
else {
    moveInput = 0;
}

if (drifting && moveInput < 0)
{
    moveInput = 0;
}
```

Now, we need to get an active check as to if the car is on the **ground**. We don't want to run our movement code-to-be if the car is in the air, or it'll just fly away. This **GroundCheck** script checks our wheel raycasts for if more than one of them is touching the ground. If they are, then we set **isGrounded** to **true**. Otherwise we set it to **False**. Call **GroundCheck()** in your **FixedUpdate()** loop, underneath **Suspension().**

## 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.

```
1 reference
private void GroundCheck()
{
    int tempGroundedWheels = 0;

    for (int i = 0; i < wheelIsGrounded.Length; i++)
    {
        tempGroundedWheels += wheelIsGrounded[i];
    }

    if (tempGroundedWheels > 1)
    {
        isGrounded = true;
    }
    else
    {
        isGrounded = false;
    }
}
```

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 out **maxSpeed**, which is **40**. We

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

will increase the **maxSpeed** when we get a **drift boost**. Finally, we apply this new velocity to our car's **RigidBody**.

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);
}
```

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

## Turning 🔴

Now you should have acceleration and deceleration working! Let's figure out how to **turn** left and right next.

Let's of course start by adding some turning-specific variables.

```
1 reference
public float steerStrength = 15f;
1 reference
public AnimationCurve turningCurve;
1 reference
public float dragCoefficient = 1f;

3 references
private Vector3 currentCarLocalVelocity = Vector3.zero;
3 references
private float carVelocityRatio = 0;
```

Then we need to calculate the **current velocity** of the car, as we need to use it when we calculate our turning. Create a **CalculateCarVelocity()** method and add it in **FixedUpdate()**, above **Movement()** as we need to calculate it **before** we turn. This is how this code works:

Firstly, we start by taking the car's **velocity** in the **world space**, and use **transform.InverseTransformDirection()** to change it to the **local velocity** of the car. Getting the local velocity of the car makes physics calculations we use later in our turning code a lot **more accurate**. We then calculate the car's **velocity ratio** by dividing the car's **forward velocity** (the z axis) by the **top speed**, which **normalizes** it and shows us how far the car is driving **relative** to how fast it can drive. This will be used for the amount of turn given, as turning is **different** at different speeds, giving that more **arcade-y feel**.

```
1 reference
private void CalculateCarVelocity()
{
    currentCarLocalVelocity = transform.InverseTransformDirection(carRB.velocity);
    carVelocityRatio = currentCarLocalVelocity.z / maxSpeed;
}
```

Now we can create our **turn method**.

Our turning will be different depending on whether we're **drifting** or not, so start by creating two float variables we'll change later when we add drifting. These are **appliedSteer** and **multiplier**. We set **appliedSteer** to **steerInput** for now, and **multiplier** to 1f.

```
private void Turn()
{
    float appliedSteer = steerInput;
    float multiplier = 1f;
    carRB.AddRelativeTorque(
        multiplier *
        steerStrength *
        appliedSteer *
        turningCurve.Evaluate(Mathf.Abs(carVelocityRatio)) *
        Mathf.Sign(carVelocityRatio) *
        transform.up,
        ForceMode.Acceleration);
}
```

Now we add **torque** to the car using all the variables we're calculated.

We multiply our **multiplier** (currently just 1) by the **strength** of our steering (pre-set value) by our **steerInput** (called appliedSteer), and then we add a **turning curve**. In order the make the game feel more **arcade-y**, adding a **turningCurve** based on the cars current velocity means that turns will be **sharper** when going faster, and **weaker** when going slower. **Mathf.Sign** and **transform.up** help us to get the position and direction of the car's movement to apply this correctly. Finally we set the type of force as **Acceleration**.

Make sure our **Turn()** method is added to our **Movement()** script, which should now look like the image you see on the right.

```
1 reference
private void Movement()
{
    if (isGrounded)
    {
        Acceleration();
        Deceleration();
        Turn();
        SidewaysDrag();
    }
}
```

Sidewards Drag 🔴

You might have tested out the turning and noticed that the car keeps spinning out. It's like it has no friction. That's because of the raycast system we're using. It means that we have to apply some of this friction ourselves in a script.

This code is simple, it first gets how fast the car is **currently moving sideways** (hence checking the **X** axis), then we calculate the **magnitude** of the drag we want to apply. We want to **push against** the car to slow it down, so we get the **opposite** direction that the car is currently moving in. This way when we multiply **dragMagnitude** by **transform.right**, we get a **Vector3** of force that we can apply **against** our car to slow it down on **sideways axis**.

```
1 reference
private void SidewaysDrag()
{
    float currentSidewaysSpeed = currentCarLocalVelocity.x;
    float dragMagnitude = -currentSidewaysSpeed * dragCoefficient;
    Vector3 dragForce;
    dragForce = transform.right * dragMagnitude;
    carRB.AddForceAtPosition(dragForce, carRB.worldCenterOfMass, ForceMode.Acceleration);
}
```

Make sure to call this in our **Movement()** Function.

Camera 🔴

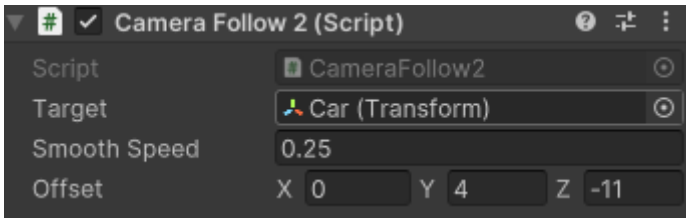
We've now got some movement and turning in our game, but they're no use if the **camera** doesn't follow the player around.

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 to 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.




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;
    }
}
```

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.

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.

```
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;
```

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)
    {
```

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;
}
```

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;
}
```

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 **sidewards 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);
```

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 l**ockedDriftDirection** 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);
    }
```

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);
```

Now that we have our main drifting method finished, let's work on that coroutine we created earlier to control the **speed boost** gained from **charging** a drift.

We want to reward the player for drifting by giving them a **boost**, but it doesn't do anything if the max speed is **too low**. So first we **increase** that by **10**. All we control in this script is **how long** it's increased for, rather than boosting their velocity. We first calculate a **duration** of how long to boost the player for. Our **charge** variable doesn't have a **cap**, so we set one here as we set the duration of the boost. We then use a **yield return** statement to make the method **wait** for the duration we calculated.

```
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;
        yield return null;
    }
    maxSpeed = baseMaxSpeed;
}
```

We then create a **while** statement to slowly **decrease** the max speed over time, and return **null** so that the **IEnumerator** statement is able to loop. Once the statement is over we **reset maxSpeed**, in case our while loop made it **lower** than **baseMaxSpeed**.

One of the issues currently with our drift system is that the car **isn't able to slide**, and will slow down quickly because of the **friction** we apply in **SidewardsDrag**. Let's fix that.

Here are the changes we make to the script:

1. Change **appliedDragCoefficient** to first check if we're drifting. If we are, then it is calculated as **dragCoefficient * driftSidewaysDragMultiplier**. If we aren't then keep it as **dragCoefficient**.
2. Only **do dragForce = transform.right * dragMagnitude** if we aren't drifting
3. Otherwise do the same but add a **multiplier** based on the **direction**. This helps to **minimize** the amount of drag, which doesn't really need to be based on the direction we're turning, it just **reduces** it down which is what we want.

```
1 reference
private void SidewaysDrag()
{
    float currentSidewaysSpeed = currentCarLocalVelocity.x;
    float appliedDragCoefficient = drifting ? dragCoefficient * driftSidewaysDragMultiplier : dragCoefficient;
    float dragMagnitude = -currentSidewaysSpeed * appliedDragCoefficient;
    Vector3 dragForce;
    if (!drifting)
    {
        dragForce = transform.right * dragMagnitude;
    }
    else
    {
        dragForce = transform.right * dragMagnitude * Mathf.Min(Math.Abs(lockedDriftDirection), 0.5f);
    }

    carRB.AddForceAtPosition(dragForce, carRB.worldCenterOfMass, ForceMode.Acceleration);
}
```

We also need to make a quick change to the Turn() Method as well, so that it obeys **lockedDriftDirection** and makes it so we can't turn the other direction when drifting.

Similar thing here to sideways drag. Just changing it so that if **drifting** is true then we follow **lockedDriftDirection** rather than **steerInput**. Same with the **multiplier**, it doesn't need to exist until we drift, and then the turn needs to get **steeper and steeper** as the turn continues, so we make that adjustment.

```
1 reference
private void Turn()
{
    float appliedSteer = drifting ? lockedDriftDirection : steerInput;
    float multiplier = drifting ? driftTurnMultiplier : 1f;
    carRB.AddRelativeTorque(
        multiplier *
        steerStrength *
        appliedSteer *
        turningCurve.Evaluate(Mathf.Abs(carVelocityRatio)) *
        Mathf.Sign(carVelocityRatio) *
        transform.up,
        ForceMode.Acceleration);
}
```

One final change, we add an if statement to the bottom of **GetPlayerInput()**. It resets our **moveInput** to **0** if the player is drifting. Obviously, our player can't drift **backwards**, so this is in place to prevent that.

```
if (drifting && moveInput < 0)
{
    moveInput = 0;
}
```

That's all of the functionality of drifting implemented! Try it out in game and see what you think! Feel free to play around and tweak values to whatever you like.

Let's work on a little **visual flair** now. When a car drifts in real life, usually clouds of smoke **billow** out from the wheels. We should add something similar so that it's **obvious** to the player that the car is drifting.

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);**

```
        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);
        }
```

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!

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;
```

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**.

```
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;
```

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**.

Now we make a **for** loop for each of the particles in the **driftChargeParticles** GameObject. 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);
    }
}
```

Back to our **Drift()** Method, we want to do two things here:
1. Enable the **DriftBoostParticles** 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 **driftChargeParticles** 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);
    }
```

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

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.

```
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;
}
```

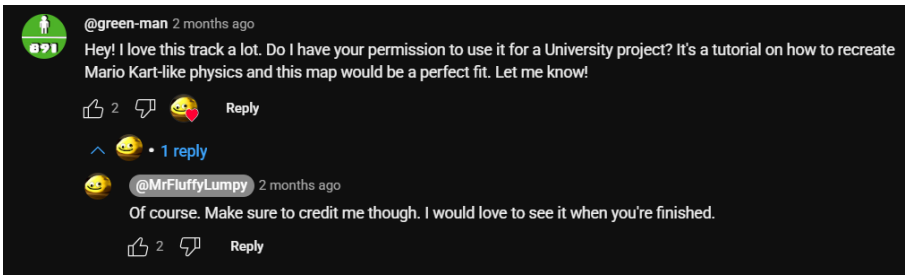
Don't forget to assign the **DriftChargeParticles** and **DriftBoostParticles** in the **carScript Inspector**, as well as our **Colours**! It doesn't matter about what specific colours you choose are, but the standard Mario Kart charge colours are **Blue** -> **Orange** -> **Purple**. So create 3 colours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

Now our car is all set up and ready to race, we need somewhere to race it in. Let's finally set up our **racetrack**.

A lot of conventional racetracks **don't seem fitting** for these more **arcade-y physics**. They often have **tight turns** and **narrow tracks** with no interesting changes in **elevation**. It would be much better if we could use an **official** track from the **Mario Kart** franchise. Those are available to download on websites that store **Sprite Resources**, but I can't tell you to use them since I don't **own the rights** to use them and have **no way of asking**.

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 EULA**. Therefore, we can simply ask for permission to use one of these tracks for our project, and then **legally** have the rights to use it.

My track of choice is **Quaking Mad Cliffs, by MrFluffy**. It has a unique colour palette, **interesting elevation** and shortcuts, and makes for one of **my favourite** custom courses. I reached out to MrFluffy and have his permission to use this track in this tutorial.




## Course

If you are to use a different course, make sure to add a **mesh collider** to every object when you import it. This means that the car won't clip through it.

Drag the course prefab into your **Unity Hierarchy**. I've added a lot extra to it by **rigorously** testing it. It now includes a **ramp** and **speed boosts**, **Out of bounds Collision Boxes**, the **correctly-sized** rickety bridge, a **checkpoint system**, and fixed **terrain issues**. Set the course's transform **position** to **0,0,0** and drag the car GameObject **in front** of the finish line. Make sure to set everything as our **drivable** layer as well.

Run the game and the car should now be able to **drive around**.

All of these Extras though, such as out of bounds detection, checkpoints and laps all need to be programmed in to get the level up to scratch.

## 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 prevents **cheating**, as well as just driving **back** over the finish line and counting another **lap**.

```
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
}
```

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
```

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.

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

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.

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;
        }
    }
}
```

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;
        }
    }
}
```

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;
    }
}
```

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.

```
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}";
    }
}
```

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 **TMPro** 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.

Car Lap Tracker 🟡 (Crucial for Checkpoints to work)

The final piece of the **Checkpoint** puzzle is the **Car Lap Tracker**, which is a script we attach to the car to detect if the car has **hit a checkpoint** or **fallen** off the map. Create the script in Assets and call it **PlayerLapTracker.cs**.

We make a reference to the **gameManager**, we have the **player's ID,** and we have the **outOfBoundsTag**, which we will set to our **OOBColliders** later.

The first Method we make is a **callback** called **OnTriggerEnter**, reading in the **collider** we touched.

```
0 references
public class playerLapTracker : MonoBehaviour
{
    3 references
    public GameManager gameManager;
    3 references
    public int playerID;
    1 reference
    public string outOfBoundsTag = "OutOfBounds";
    0 references
    private void OnTriggerEnter(Collider other)
    {
        Checkpoint checkpoint = other.GetComponent<Checkpoint>();
        if (checkpoint != null)
        {
            gameManager.PlayerHitCheckpoint(playerID, checkpoint);
        }
        else if (other.CompareTag(outOfBoundsTag))
        {
            gameManager.HandlePlayerOutOfBounds(playerID);
        }
    }
```

We get the **Checkpoint** component of the collider we touched. If it **exists**, then we **trigger** the **PlayerHitCheckpoint** method from our **gameManager**. If it doesn't exist, and it has the **Tag** "**OutOfBounds**", then we call the **HandlePlayerOutOfBounds** script from our **GameManager**.

```
private void Update()
{
    if (Input.GetKeyDown(KeyCode.R))
    {
        gameManager.HandlePlayerOutOfBounds(playerID);
    }
}
```

We use the **PlayerLapTracker** update in order to trigger that reset when **R** is pressed, which allows the player to reset when the car flips or gets stuck without having to restart.

Add the **PlayerLapTracker** to the **Car GameObject**, and add the **GameManager** into the slot in the Inspector. Make sure to set the **playerID** to **0** as well.

Select the **OOBColliders GameObject**. Click on "**Tag**" in the **Hierarchy** and select "**Add Tag...**". Press the **+** symbol and call the Tag exactly "**OutOfBounds**". Apply this to the **OOBColliders GameObject**, and select **Yes** to apply it to the **Child GameObjects** as well.

Now checkpoints should be working! Only one way to find out!

## Speed Boosts and Offroad 🟡

Not every imported course will have speed boosts. But every course will have parts of the course that are considered Offroad and therefore need to slow the player down on those parts.

Let's head back to our **carScript**.cs and add **4** new variables: Two **LayerMasks** for the offroad and the SpeedBoost called **Offroad** and **Speedboost**, and two Boolean variables called **isOffroad** and **onSpeedBoost**.

```
1 reference
private void GroundCheck()
{
    int groundedWheels = 0;
    int offroadWheels = 0;
    int speedBoostWheels = 0;
    float raycastDistance = restLength + springTravel + wheelRadius;

    for (int i = 0; i < rayPoints.Length; i++)
    {
        if (Physics.Raycast(rayPoints[i].position, -rayPoints[i].up, out RaycastHit hit, raycastDistance))
        {
            int hitLayerBit = 1 << hit.collider.gameObject.layer;
            if ((hitLayerBit & drivable) != 0) groundedWheels++;
            if ((hitLayerBit & offRoad) != 0) offroadWheels++;
            if ((hitLayerBit & speedBoost) != 0) speedBoostWheels++;
        }
    }
```

We start by **changing GroundCheck()** to be able to check for **isOffroad** and **onSpeedBoost** too. We do this by making **variables** for how many wheels are **currently** in that state. Then, from each of the wheels we fire a **raycast** downwards to hit the floor. We get the **layer** of the floor, and if it's **drivable** then that wheel is **just on the ground**. If it's **offroad** then that wheel is **offroad**, and if it's **speedBoost** then that wheel is **on a speedBoost**.

```
    bool currentlyGrounded = groundedWheels > 1;
    bool currentlyOffRoad = offroadWheels > 2;
    bool currentlyOnSpeedBoost = speedBoostWheels > 0;

    isGrounded = currentlyGrounded || currentlyOffRoad || currentlyOnSpeedBoost;
    isOffRoad = currentlyOffRoad;

    if (currentlyOnSpeedBoost)
    {
        if (!onSpeedBoost)
        {
            DriftBoostParticles.SetActive(true);
            StartCoroutine(DriftSpeedBoost(3));
        }
        onSpeedBoost = true;
    }
    else
    {
        onSpeedBoost = false;
    }
}
```

We then calculate what the **threshold** is for each of these states to be the case. If **2 wheels** are on the **ground** then we are **grounded**. If **3 wheels** are on the **Offroad** layer then we are **Offroad**. If there are **any wheels** on a **SpeedBoost** then we are on a **SpeedBoost**. We don't want to be **stingy** about a **SpeedBoost** or **generous** on an **Offroad** since we want to **maximise** the player having **fun**.

Of course **isGrounded** is **true** if any of them are **true**, so we set that as the case. We set **isOffRoad** to the result of **currentlyOffRoad**. For the **Speed Boost** though, if we're not already on a **Speed Boost**, then we start the **Boost Particles**, and give the Car the **equivalent** of a **maximum drift boost**.

Now we make changes to **Acceleration**, to create a **newMaxSpeed** based on if the Car is **OffRoad** or on a **Speed Boost**. In **Offroad** we want to set the **maxSpeed** to **lower** than usual, and on the **Speed Boost higher** than usual. We also increase the **velocity faster** on a **Speed Boost** so that it feels more like a **Boost** than a gradual increase.

```
1 reference
private void Acceleration()
{

    Vector3 newVelocity;
    newVelocity = carRB.velocity + (acceleration * moveInput * transform.forward * Time.fixedDeltaTime);
    float newMaxSpeed = isOffRoad ? 25 : onSpeedBoost ? 70 : maxSpeed;
    newVelocity = Vector3.ClampMagnitude(newVelocity, newMaxSpeed);
    newVelocity = onSpeedBoost ? newVelocity * 1.05f : newVelocity;
    carRB.velocity = newVelocity;
}
```

Make sure to also change the **Suspension** code so that the Raycast accepts **drivable**, **Offroad** and **Speed Boost**.

```
if (Physics.Raycast(rayPoints[i].position, -rayPoints[i].up, out hit, maxLength + wheelRadius, drivable | offRoad | speedBoost))
```

Now we need to create these **layers** and apply them in the correct place. Firstly, create the layers using the **Layer** bar next to where we made a **Tag** earlier (at the top of the Inspector when selecting a GameObject) and call them **Offroad** and **SpeedBoost**. Apply the Speed Boost layer to **Plane.022Boost**, and **SpeedBoost** (Found **under** the **Ramp** GameObject).

For the **Offroad** layer, it is up to you to look around the course and double click on anything you consider to be off of the standard road. Double clicking it should select it **without** selecting the rest of the course. Make sure it hasn't before applying the layer.

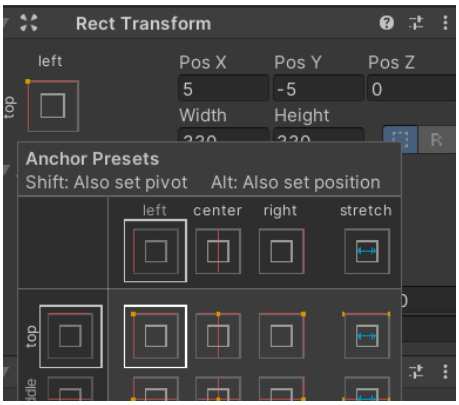
In **carScript**, set **OffRoad** and **SpeedBoost** to be the **correct** layers in the Inspector.

## Mini map 🟢

Let's add a **mini-map** to give the player a helpful **top-down** view of where they're driving.

Start by creating a new **Camera** gameObject in the **hierarchy** (separate from our gameObject called **Camera**) called **MinimapCamera**. Place this **anywhere** you like, and set the **X axis** rotation to **90**, so that it's facing **directly** downward. In the Inspector, set the Camera's settings from **Perspective** to **Orthographic**. Set the **Orthographic** size to **15**.

Next, create a **Render Texture** in the **Assets** folder from the **Project window** at the bottom, and call it **MinimapRenderTexture**. Set the size to **256x256**. Click back on the **MinimapCamera** and apply this **Render Texture** in the "**Target Texture**" slot. This allows us to apply what the **MinimapCamera** sees as a **Texture** to use later.




Now create a new **UI Image** (by going **Create** -> **UI** -> **Image**) and calling it **MinimapOutline**. In "**Rect Transform**", press **shift** + **alt** and click the **top left** anchor. This makes sure that the minimap **stays** in the top left. Change **posX** and **posY** next to the **rect transform** to **5** and **-5** respectively, so that the minimap is positioned **slightly away** from the corner of the screen. In **Width** and **Height**, we set how **big** our minimap should be on the screen. I have chosen **330x330**, since it'll be a **1920x1080** screen and we don't want to take up too much of the screen. In the Image component of our UI element, add the "**whiteCircle**" Image provided in the assets. Set the colour to **black**.

Create another UI image as a **child** of **MinimapOutline**. Call this one **MinimapViewport**. In **Rect** Transform, select the **bottom right** anchor preset, with the arrows going in **all directions**. Set **Left**, **Right**, **Top** and **Bottom** to all be **15**. This allows the black circle we made earlier to be an **outline**. Put the same **whiteCircle** Image into the **Image component**. Add a **Mask component** to this image as well, and uncheck "**Show Mask Graphic**"


Finally, create a **UI Raw Image** as a child of **MinimapViewport**. Call this one **MinimapDisplay**. In **Rect Transform**, set the **anchor preset** to the **same** one as last time, **bottom right**. Add the "**Minimap Render Texture** into the **Image** texture slot.

We're almost done with the Minimap, we just need to add a **script** to tell our camera to **follow** the car. Create a script called **MinimapController.cs** and add a variable for the player's **position** (Transform), how **high** above the player we want the camera to be, how **zoomed in** or **out** our camera should be, and a **reference to the camera** we want to move.

In our start component, all we do here is get our **Camera** component and set the **orthographicSize** to our **camDistance** variable.

```
0 references
public class minimapController : MonoBehaviour
{
    2 references
    public Transform player;
    1 reference
    public float heightOffset = 20f;
    2 references
    public float camDistance = 15f;
    4 references
    private Camera minimapCamera;

    // Start is called before the first frame update
    0 references
    void Start()
    {
        minimapCamera = GetComponent<Camera>();
        if (minimapCamera)
        {
            minimapCamera.orthographicSize = camDistance;
        }
    }
```

Next, we change the default **update** loop into a **LateUpdate** loop, since it's good practice to move the camera **after** the player's moved that frame. We start with the car's **position**, shift it on the **Y axis** up by our **offset**, and then set the camera's position to that **new position** we made.

We want to **mimic** the rotation as well, so we make a new **Vector3** with the **X axis** on **90** (since we're facing downwards),

and then **copying** the car's **y axis** rotation, which is then the car turns **left** or **right**.

```
    void LateUpdate()
    {
        Vector3 newPosition = player.position;
        newPosition.y += heightOffset;
        transform.position = newPosition;
        transform.eulerAngles = new Vector3(90f, player.eulerAngles.y, 0f);
        minimapCamera.orthographicSize = camDistance;
    }
}
```

Drag the car's GameObject into the **minimapCamera's minimapController** script and the minimap should be working!

## Sounds 🟡

Let's implement some **basic** sound effects. The routine for each of the sound effects is pretty similar, so we will implement two **crucial** ones – the **engine sound** and the **drifting sound**.

```
2 references
public AudioSource engineAudioSource;
5 references
public AudioSource driftAudioSource;
3 references
public AudioSource skidAudioSource;
3 references
private carScript carScript;
2 references
private Rigidbody rb;

2 references
public float minPitch = 0.5f;
1 reference
public float maxPitch = 2.5f;
1 reference
public float maxPitchSpeed = 50f;
1 reference
public float pitchChangeSpeed = 2.0f;
7 references
private float currentPitch;
```

Keep in mind that the **driftSound** chosen here is for the **specific course** I've chosen – it's a **dirt track** and therefore the drift sound reflects that. Use a different sound for a different surface.

In the **Car** Game Object, create 3 Child GameObjects: **EngineSound**, **DriftSound** and **SkidSound**. On each of these attach an **Audio Source Component** and drag the corresponding **mp3** file found in the **Sounds folder** into each **AudioClip** slot. Make sure to hit **loop** if it's a looping sound effect. All of these ones **are** so we do enable it.

Now we create a new script called **CarSound.cs** and attach it as a component in our Car GameObject. We add our **AudioSources**, our **carScript**, the car's **rigidbody** for **velocity** readings, the **highest** and **lowest pitches** of our sounds, and **how fast** they get there.

```
void Start()
{
    carScript = GetComponent<carScript>();
    rb = GetComponent<Rigidbody>();
    currentPitch = minPitch;
    engineAudioSource.pitch = currentPitch;
    driftAudioSource.pitch = currentPitch * 0.6f;
    skidAudioSource.pitch = currentPitch * 0.8f;
}
```

In our **Start()** Method, we first get the **necessary** components and just set all the audio sources to a **base pitch**, in case anything errors.

In our Update Loop, we want to make the engine pitch **change with the speed** of the car, so that it sounds **realistic**. We start doing this by getting the **current velocity** of the car using it's **rigidbody**, before **normalizing** it by **clamping** it by our **maxPitchSpeed**.

```
0 references
void Update()
{
    float currentSpeed = rb.velocity.magnitude;
    float normalizedSpeed = Mathf.Clamp01(currentSpeed / maxPitchSpeed);
    float targetPitch = Mathf.Lerp(minPitch, maxPitch, normalizedSpeed);
    currentPitch = Mathf.Lerp(currentPitch, targetPitch, Time.deltaTime * pitchChangeSpeed);
    engineAudioSource.pitch = currentPitch;

    if (carScript.drifting)
    {
        if (!driftAudioSource.isPlaying)
        {
            driftAudioSource.pitch = currentPitch * 0.6f;
            driftAudioSource.Play();
            skidAudioSource.pitch = currentPitch * 0.8f;
            skidAudioSource.Play();
        }
    }
    else if (driftAudioSource.isPlaying)
    {
        driftAudioSource.Stop();
        skidAudioSource.Stop();
    }
}
```

If the car just **jumped** up to a high pitch it would sound **strange**, so instead we use **Linear Interpolation** to bridge the gap between the two pitches in a **fluid** fashion. We do the same with the **actual pitch** we apply to the engine, multiplying it by **Time.deltaTime** so that it sounds **the same** across framerates.

For our **engine pitch**, that's it. It plays all the time since the car is running all the time, so the pitch is just very low when the car is stopped. For the **drifting Audio**, it's a little different.

Our drifting audio really only needs the **driftingAudioSource**, but it's more fun to **play into** the **tire-skidding** sound with a **skidSound** as well.

We check if the car is drifting using **carScript.drifting**, and if the audio sources aren't **already** drifting, then we play them **both**, using a **lower** version of **currentPitch** since it's mostly for the engine. If the car **isn't** drifting and the audio sources is playing, then we **stop** the audio sources.

This is pretty rinse and repeat for the rest of the sounds you may want. Want a **spark** sound for when the player hits the drift sparks? Simply check the **driftCharge** from **carScript** and play the **corresponding** sound. Want to make an **Offroad grass** sound? Check **isOffroad** from **carScript** and play the sound **accordingly**.

Make sure to add the **Audio Sources** into the correct slots on the **CarSound.cs** component in the Car GameObject.

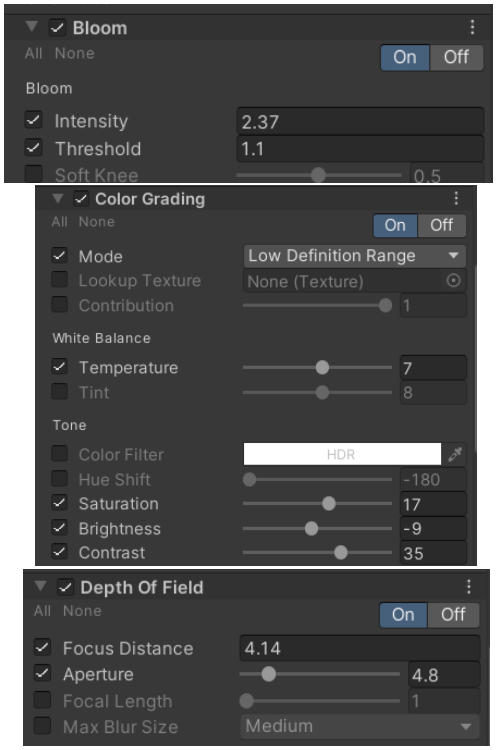
## Post-Processing 🟢

Let's add some **post-processing** and make the game look a little fancier. Currently the colours are completely different to how courses look in the Mario Kart games. They're very **contrast-y** and **colourful**. Let's implement some of that.

In the bar at the very top, click "**Window**", and then "**Package Manager**". In the top left of the Package Manager, Click on "**Packages**:" and select "**Unity Registry**". Scroll until you find the "**Post Processing**" package. Click **Install** and **Import** if necessary.

Once that's installed, Right click on **Assets**, press **Create** and create a **Post Processing Profile**. Create a blank GameObject called **Post Processing** and add a **Post Processing Volume Component**. Add the Profile we created into the "**Profile**" slot.

Now it's time to start changing the game however you feel fit. Mess around with **all** of the values and see what they do and how they change how the game looks. Here's what I've changed **personally**:




- **Bloom** really makes the sparks particles look **flashy** and **vibrant**, highly recommend.
- **Color Grading** allows us to give the game more of a **warm** look, as well as being nice, **bright** and **contrasting**.
- **Depth of Field** lets us **blur** the game in the distance, which helps **mask** over any pop-in the Unity Engine might do, as well as giving more of a hazy **retro** feel which is what we want for a game inspired by **Mario Kart.**

Feel free to add or change any of these values to your liking.

We can also make changes to the lighting tab. In "**Window**", click "**Rendering**", then "**Lighting**". It will open a window which you can put where the **Inspector** is for easy access.

In the **Materials** folder you will also find a nice-looking **skybox** of a **foggy day**. Apply this at the top of the Lighting settings to get rid of the **default skybox** and **match the vibes** a little more.

In "**Environmental Lighting**", Change **Source** to "**Color**" to be able to change the colour of the **environment** to whatever tone you want.

In "**Other Settings**", **Enable Fog**, and set the value to **0.003.** We want the appearance of fog without it **hurting** the **gameplay** and blocking vision too much, and this is a nice balance.

If you want more direct lighting control, play around with the "**Directional Light**" that comes with your **Unity Project** and is found in the **Hierarchy**. This is enough for this tutorial though.

## What Next? 🟢

**The tutorial ends here, but here is a list of other things to implement, going from easiest to most difficult.**

- A **3, 2, 1, Go** countdown before the race begins
- A **Title Screen** for the player
- A **leaderboard** of the best track times (Local or Online)
- **Different Tracks** and **Courses** for the player to choose from
- **Different** Types of **Vehicles** for the player to choose from
- **AI Racer** Opponents to face against
- **Items** like **Speed Boost Mushrooms** or **Banana Peels**

- **Online Support for Multiplayer** – Use **FishNet Networking** & **Steamworks API** since they're both free.