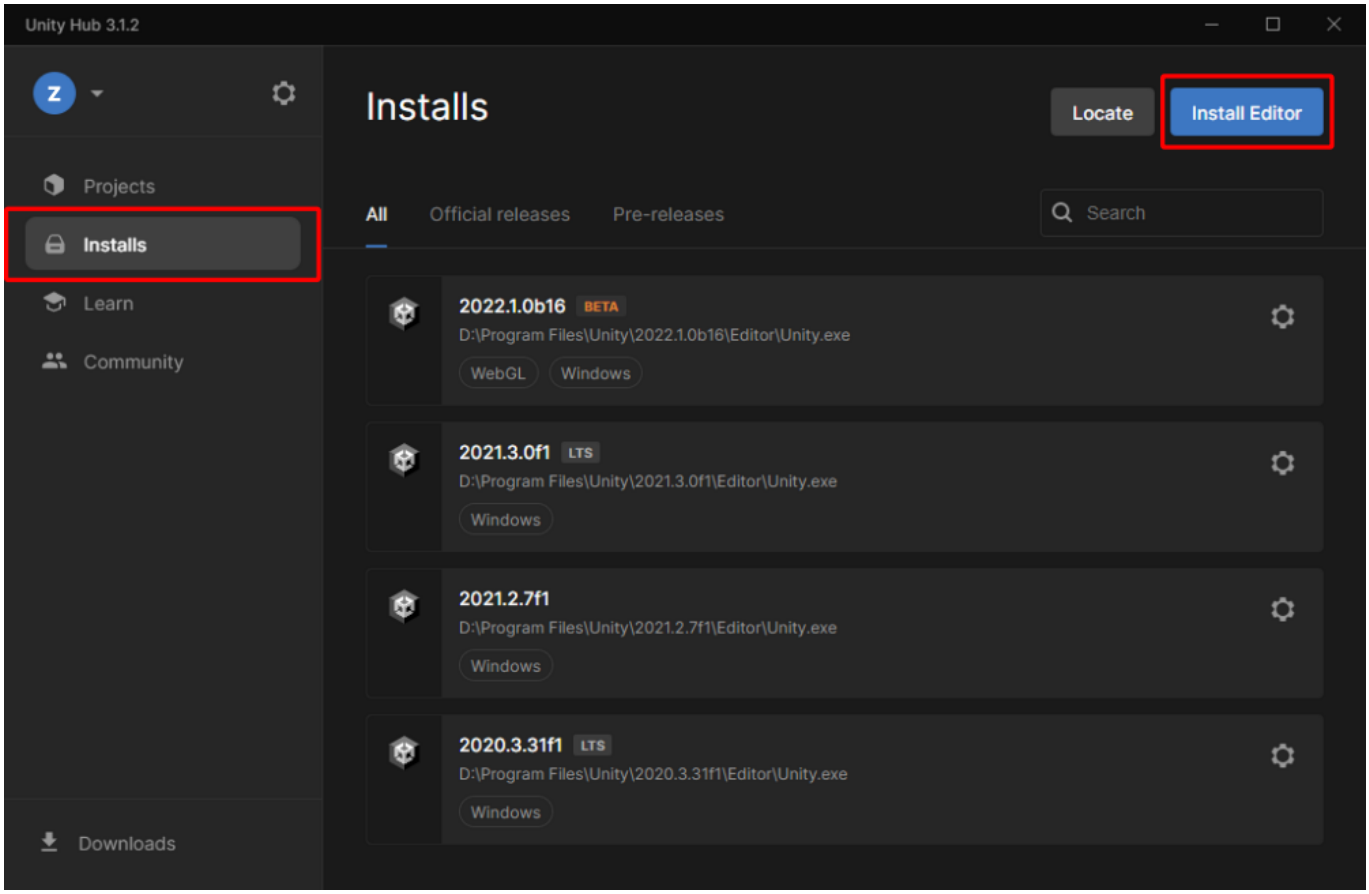
Course Updated to Unity 2021 LTS

We've updated the project files to Unity version **2021.3 (LTS)** for this course – this is a long-term support version, which Unity recommends. LTS versions get released each year and are a stable foundation for you to build your projects.

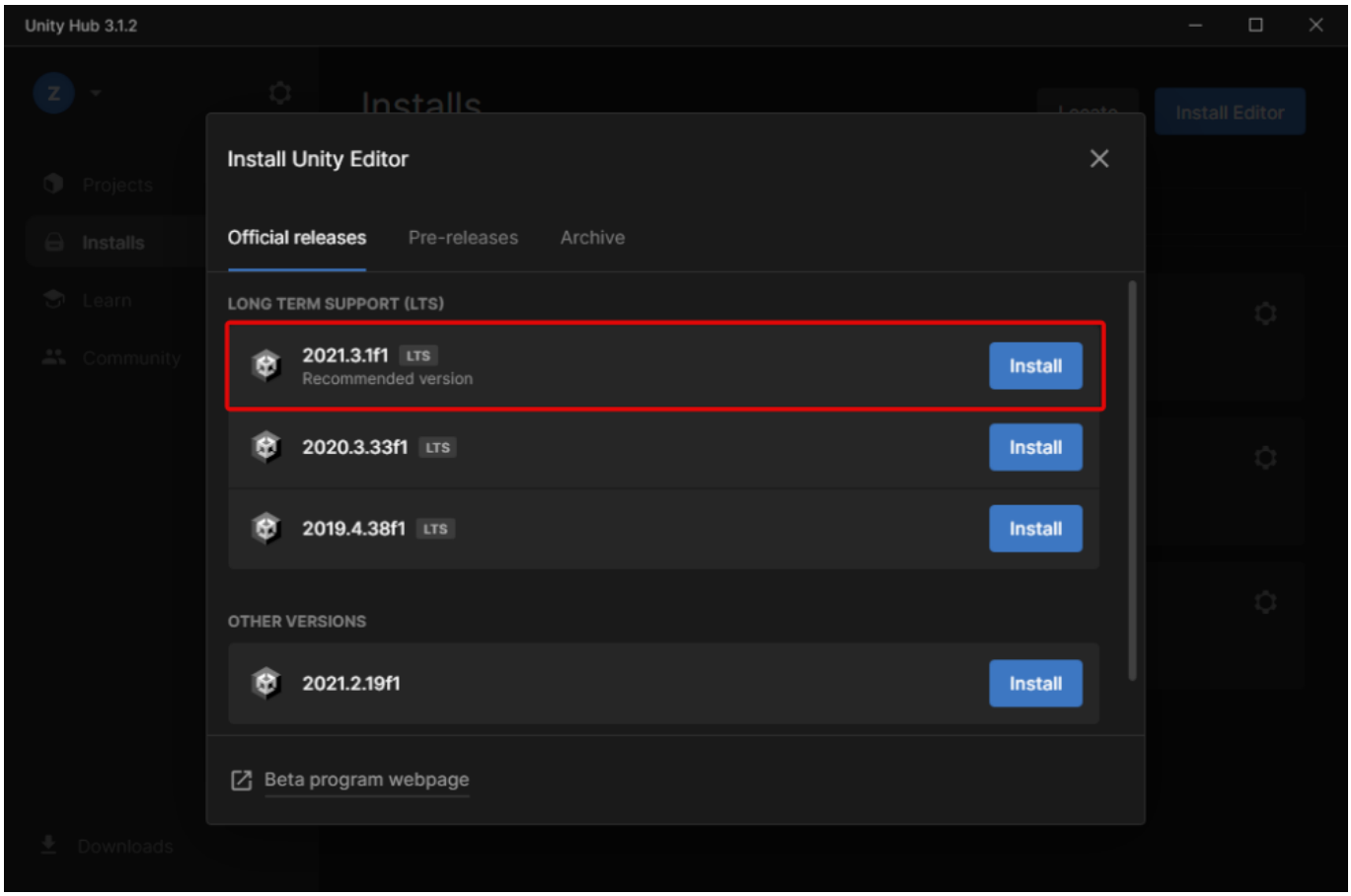
For students, this means consistent project files across all courses – no matter when created. So no more downloading different Unity versions or opening old projects filled with errors!

How to Install 2021 LTS

First, open up your **Unity Hub** and navigate to the **Installs** screen. Then, click on the blue **Install Editor** button.



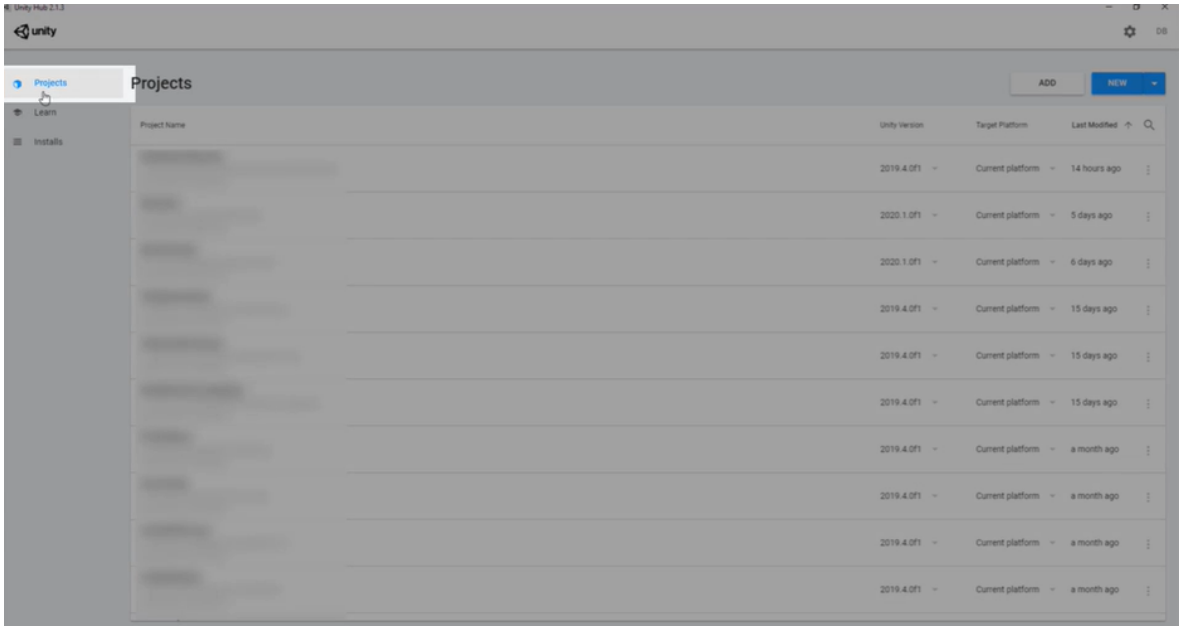
A smaller window with a list of Unity versions will open. At the top of the **Official releases** list, we want to select **Unity 2021.3 (LTS)**. From here, click **Install** and follow the install process.



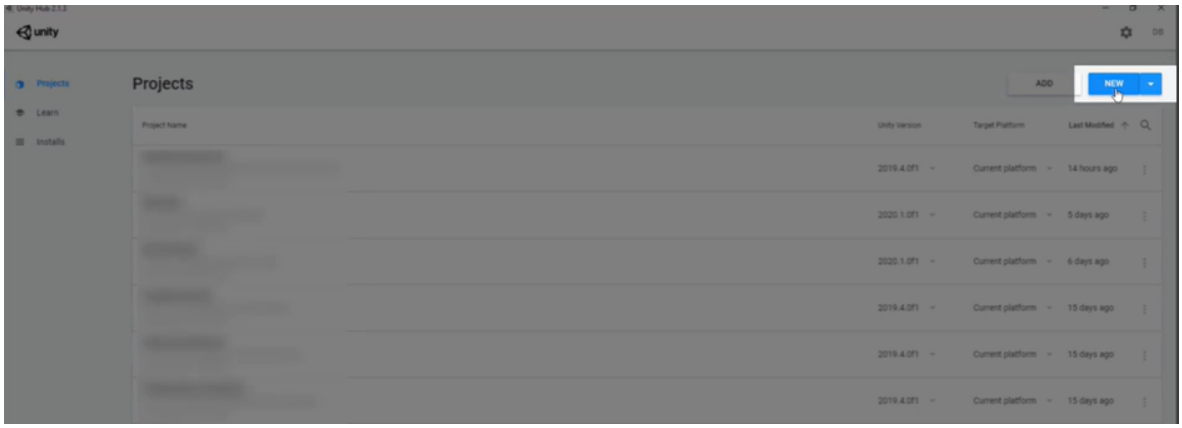
In this lesson, we'll begin setting up our project.

You need to have the latest version of **Unity Hub** installed if you haven't already. Download Unity Hub: https://unity3d.com/get-unity/download

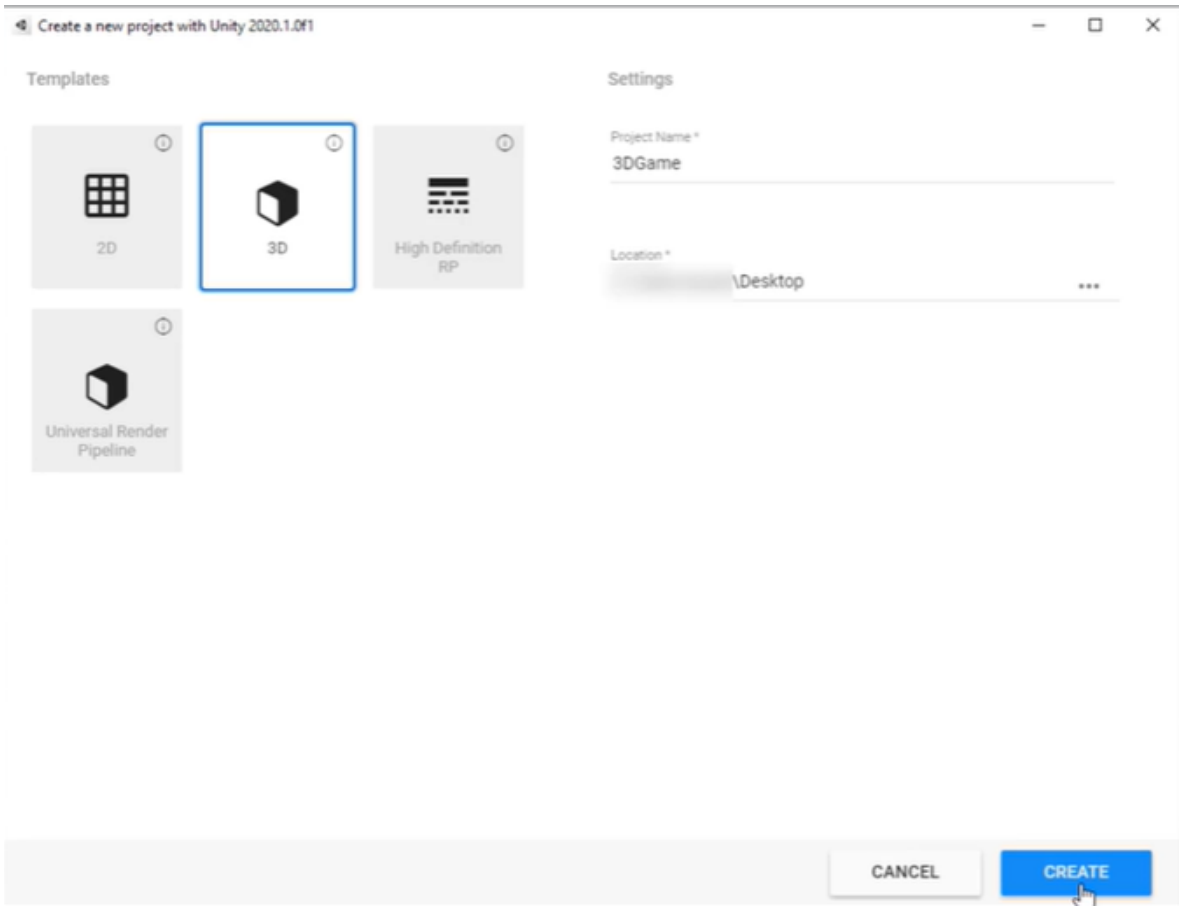
Open up Unity Hub, and click on the **Projects** tab:



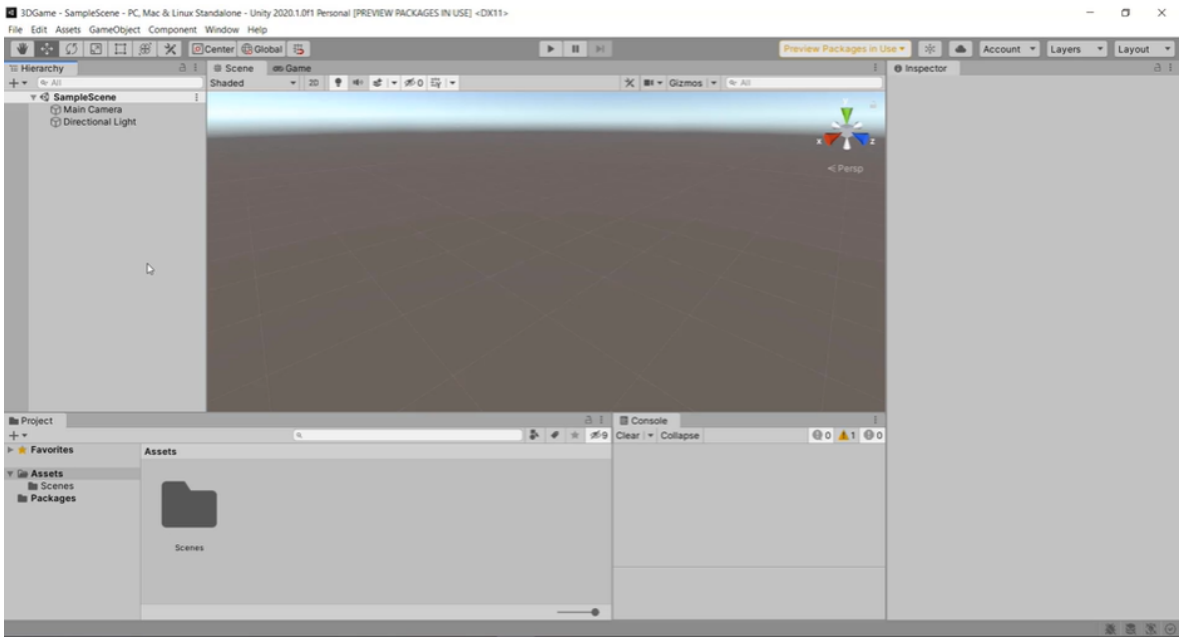
... and click on the **New** button**:**



Select the **3D template**, and set the project's name and location. Then click on the '**Create**' button.

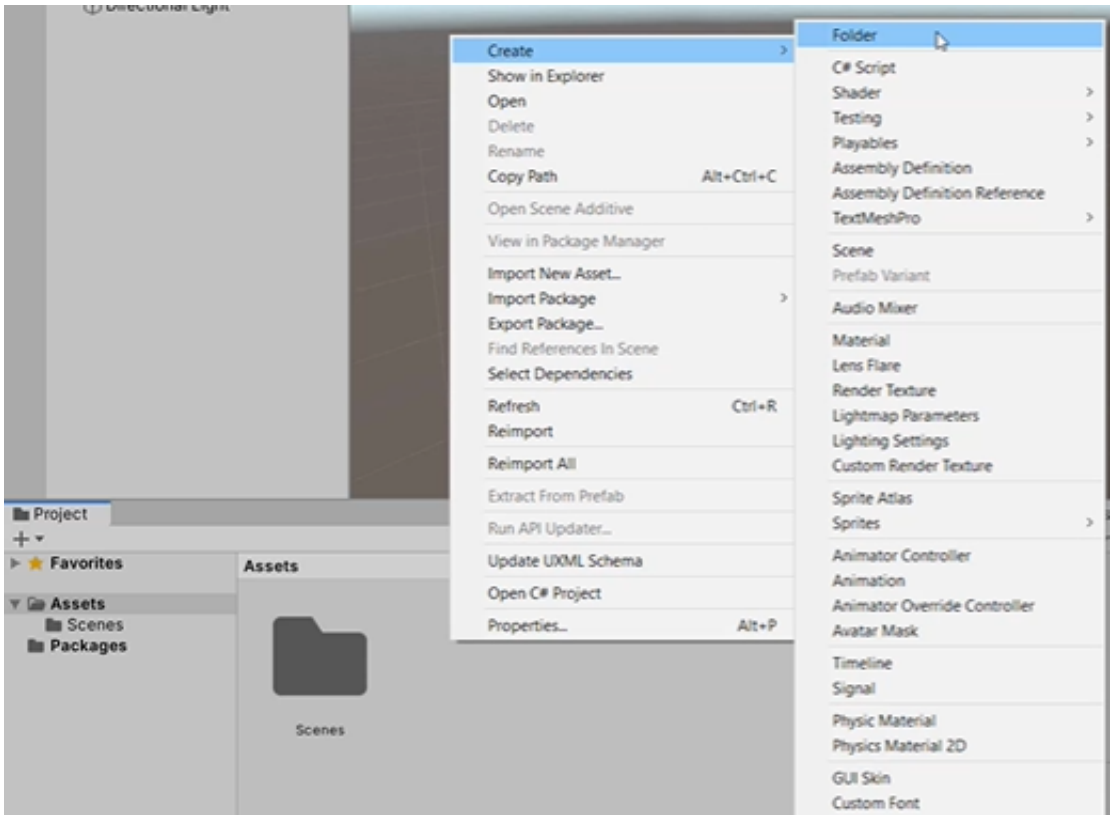


We now have the Unity Editor open inside of our brand new project.

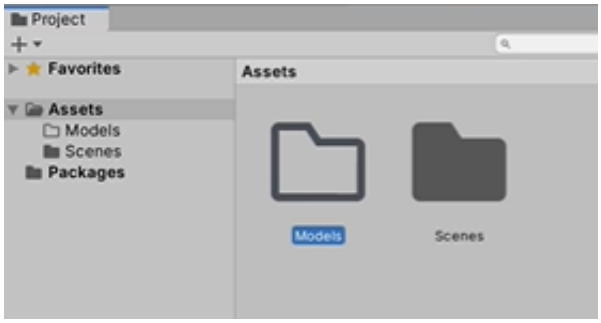


Now we're going to **create a new folder** to store our 3D models.

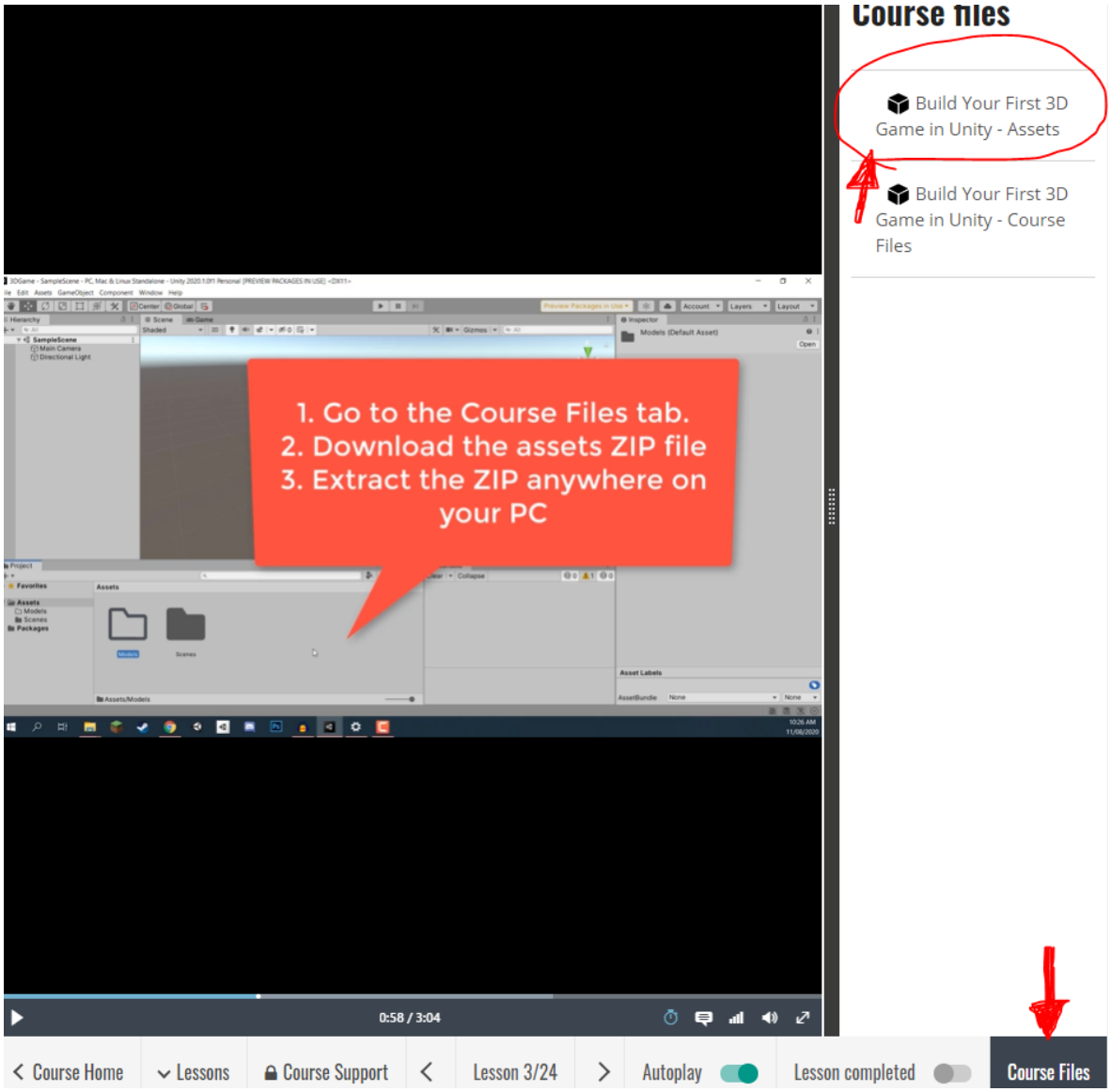
Right-click on the project panel, and go to **Create > Folder**.



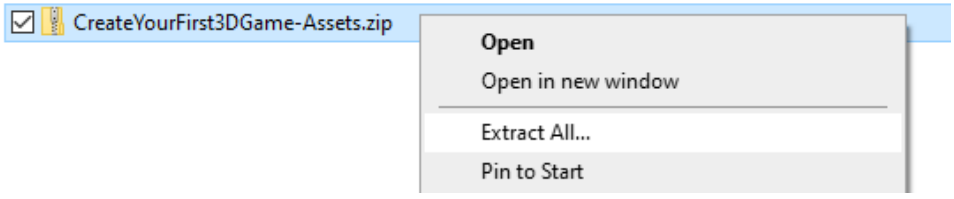
Then we're going to **name** the folder "Models".

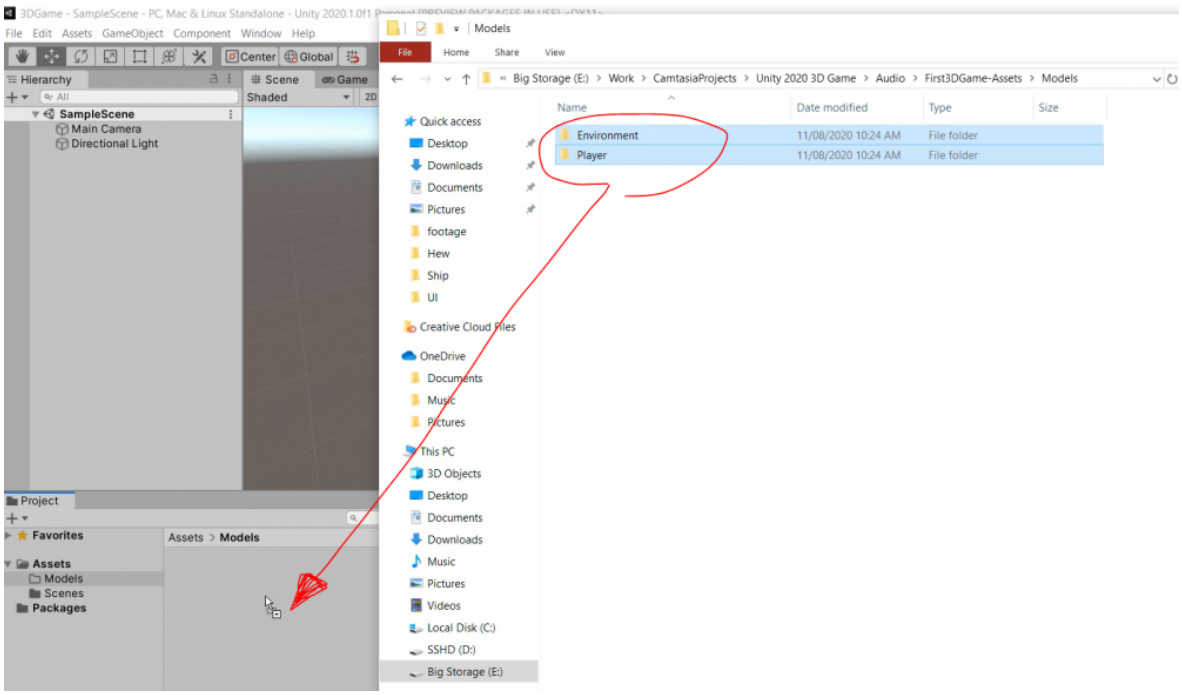


Next, we're going to download the **Assets.zip** file from the **Course Files** tab.



Select all the files inside the extracted folder, and drag them into our project browser.

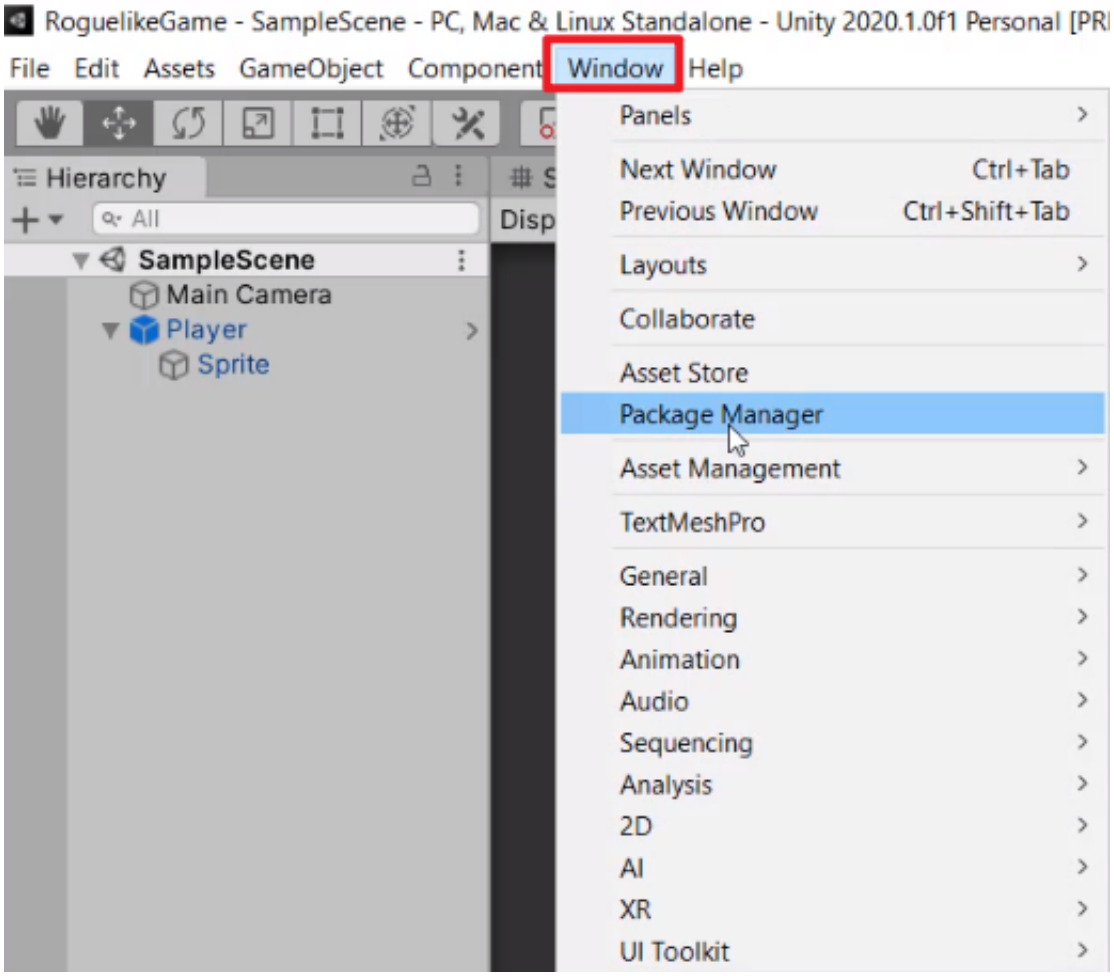




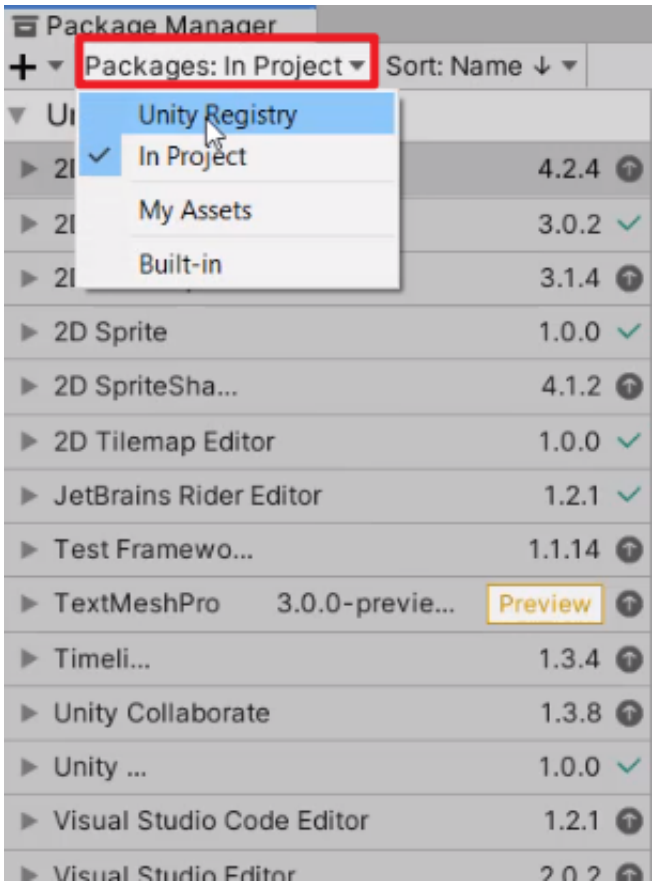
Now, we're going to be setting up our **inputs** using Unity's **Input System** package.

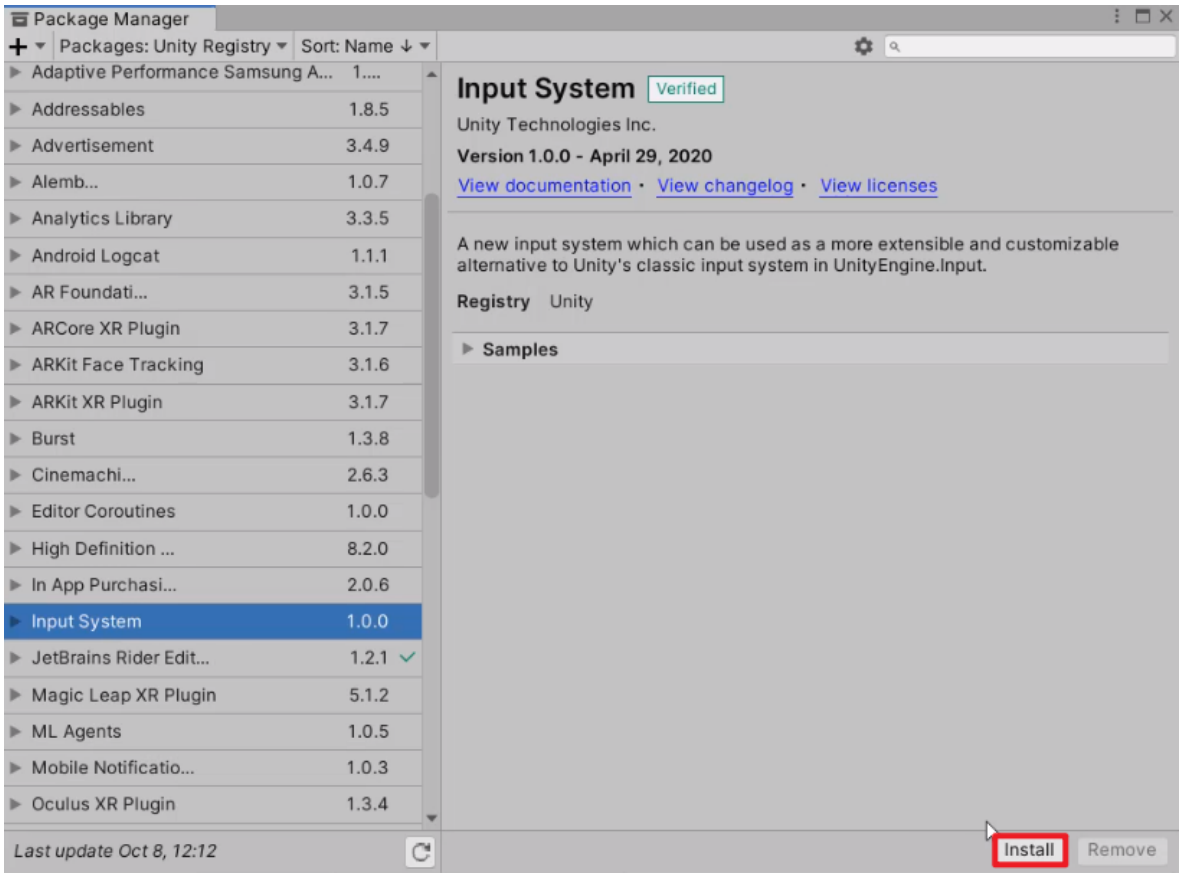
Installing Input System Package

Let's open up the **Package Manager** window (**Window > Package Manager**).



Select **Packages > Unity Registry** to access all the Unity packages, and install the **Input System**.

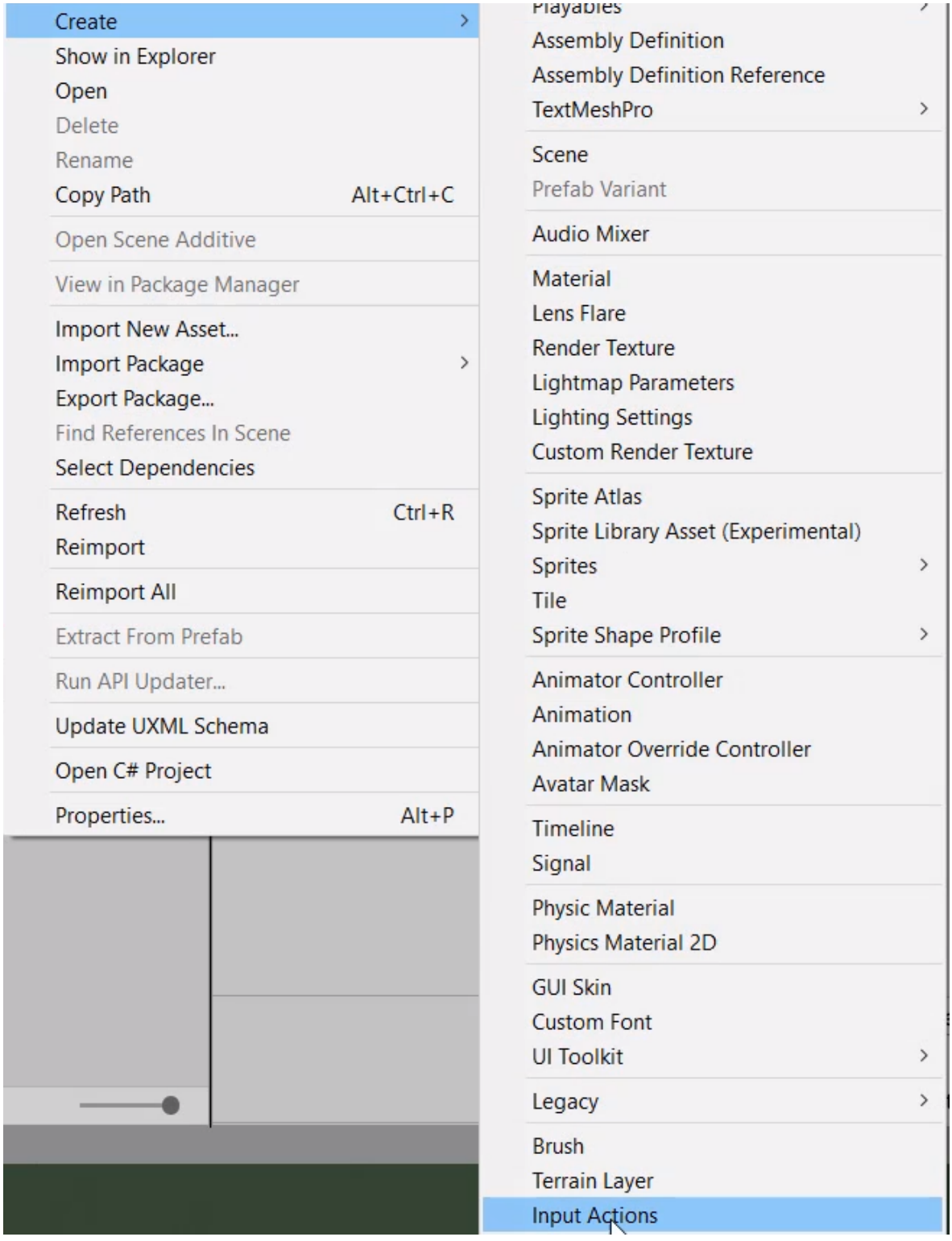




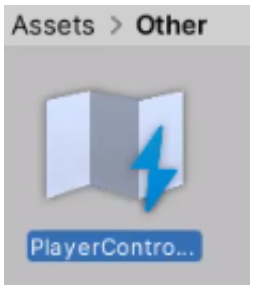
In this lesson, we're going to be setting up **inputs** for our game.

Creating Input Actions

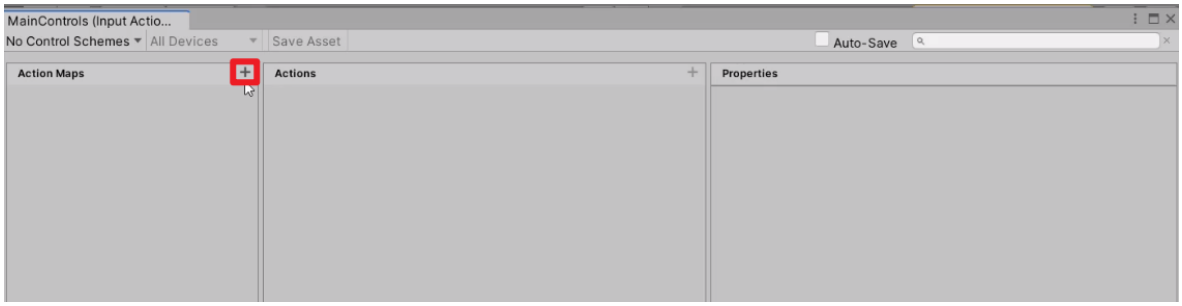
Once the **Input System Package** is installed, we can **right-click** on the Project window and click on **Create > Input Actions**.



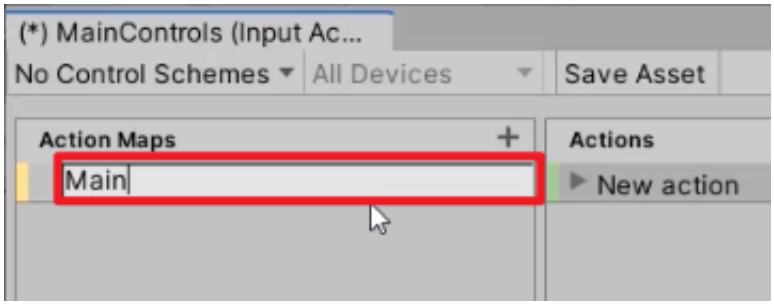
We're going to call this "**PlayerControls**" and **double-click** to open it up.



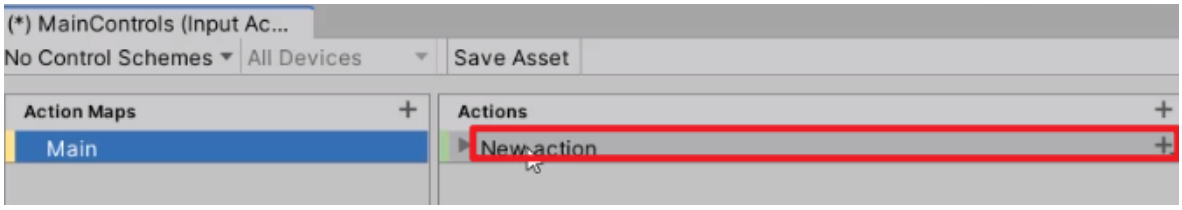
This is going to open up the **Input Actions** window. On the left panel, we can create an **Action Map** by clicking on the **+** button. This is basically a category for all your different inputs.



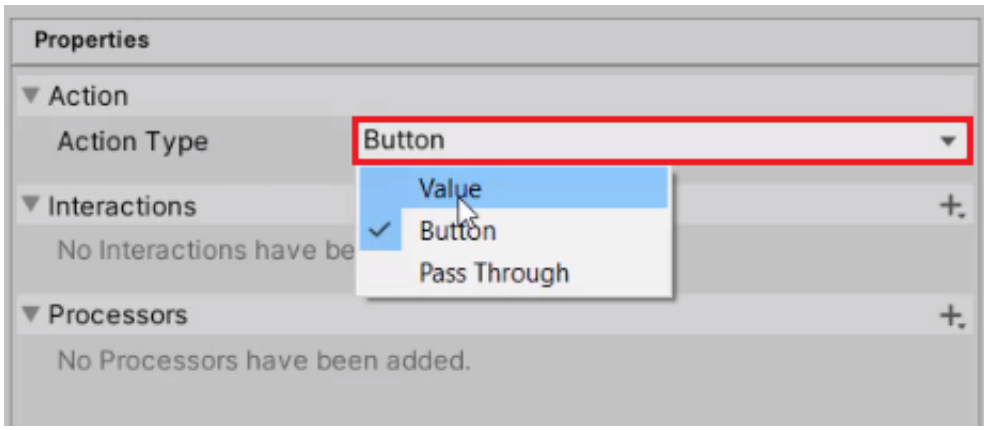
We'll call the first action map "**Main**", and create a new **Action** connected to the "Main" action map. Actions define what we want to do when pressing certain keys.



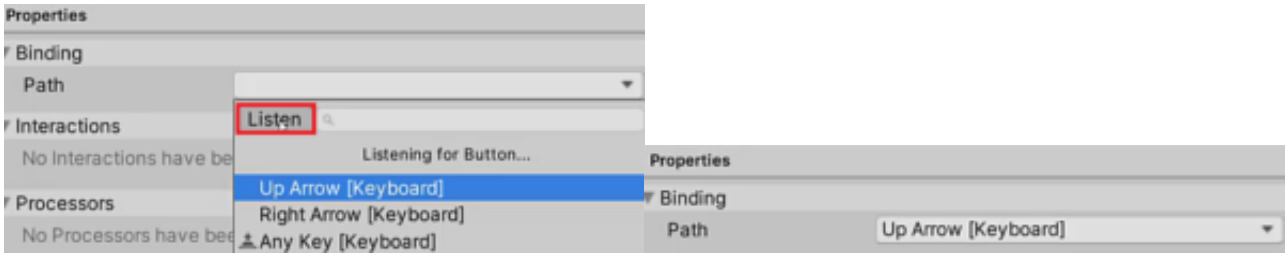
So first of all, we're going to create an action called "**Move**".



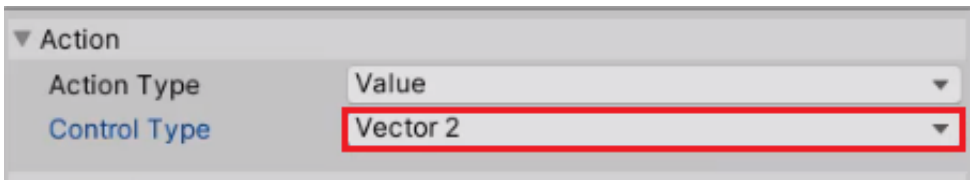
Then we want to go over to the **Properties** panel and set the **Action type** to **Button**.



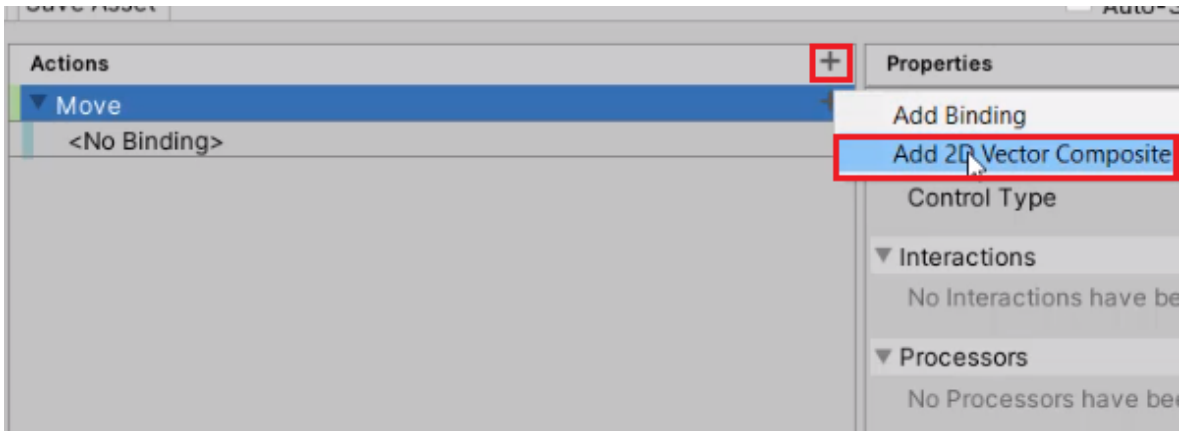
Now we can click on **Binding > Path > Listen** and then press any button on the keyboard to register and bind it with our **Input Action**.

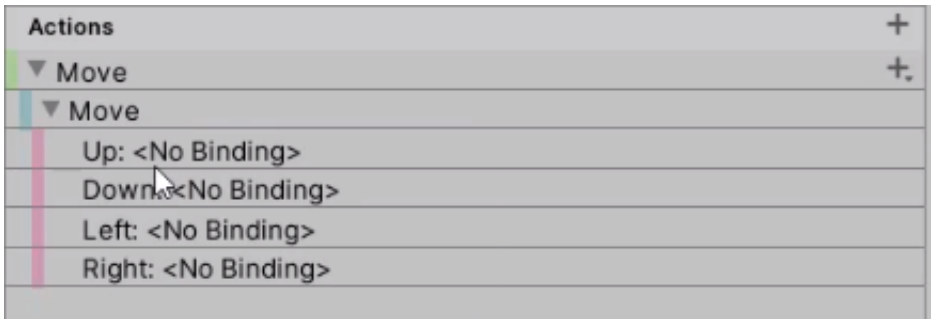


(Alternatively, if we want our movement to be represented with a **Vector2** value between -1 and 1, we can change the **Action Type** to **Value** and set the **Control Type** to **Vector2**.)

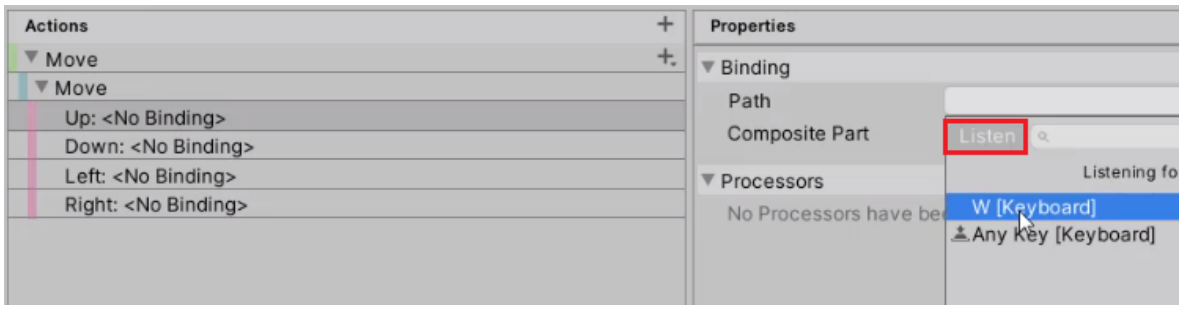


(In that case, we can determine which direction the player needs to move in by adding **2D Vector Composite** to the Move action. This will create a new binding with four children bindings called Up, Down, Left, and Right.)

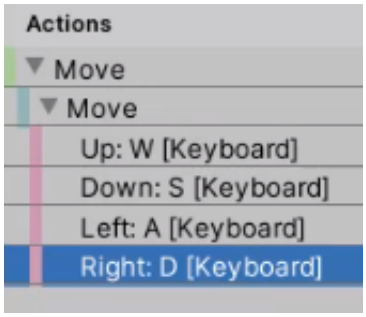




To bind a key to an input action, click on **Path > Listen**, and hit the corresponding keyboard button (W, A, S, and D).



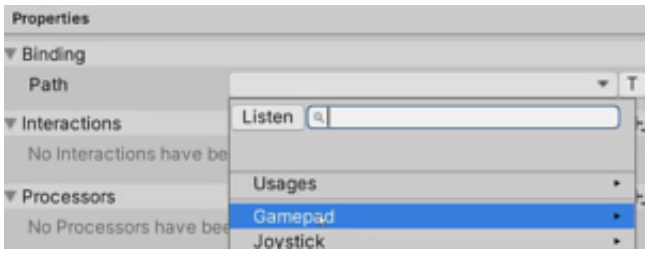
Make sure that all input actions are assigned a key.



Additional Inputs

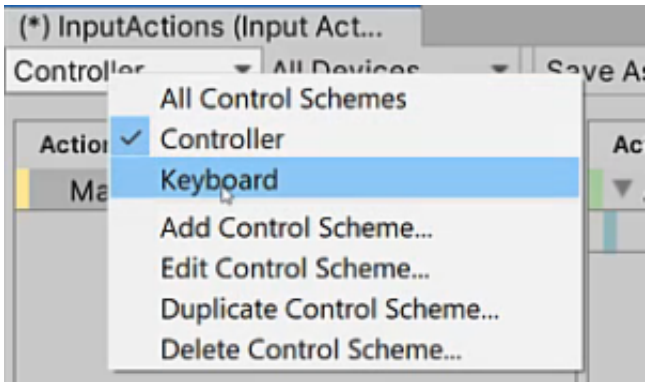
What if we want to plug in a controller for our second player? In that case, we can click on the **+ (Add) button** next to the **Actions** heading, and then click on **Add Binding**.

If you have Xbox or PlayStation controllers, you can choose the **Gamepad** option instead of the Keyboard for binding the keys.



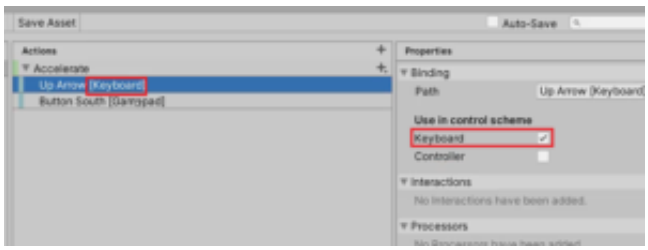
For specific gamepad controls' information, refer to the documentation:
https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Gamepad.html

You can also separate the window for each type of controller by adding a new **Control Scheme**.



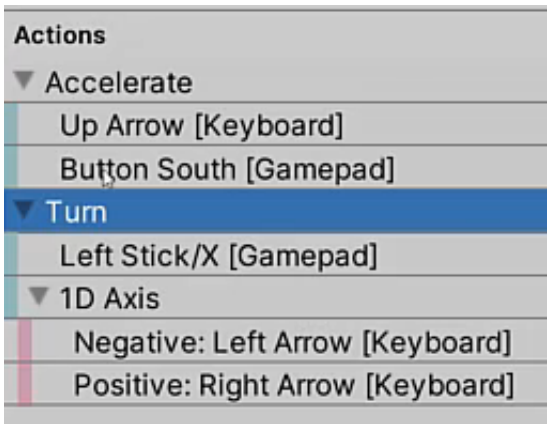
By clicking on the **Control Scheme** dropdown at the top-left corner, you can switch over to different control schemes (Keyboard/Controller/Joystick/etc).

If you have multiple control schemes, you need to make sure that each key binding is used in the appropriate control scheme. This can be done by enabling/disabling the checkboxes under '**Use in Control Scheme**'.

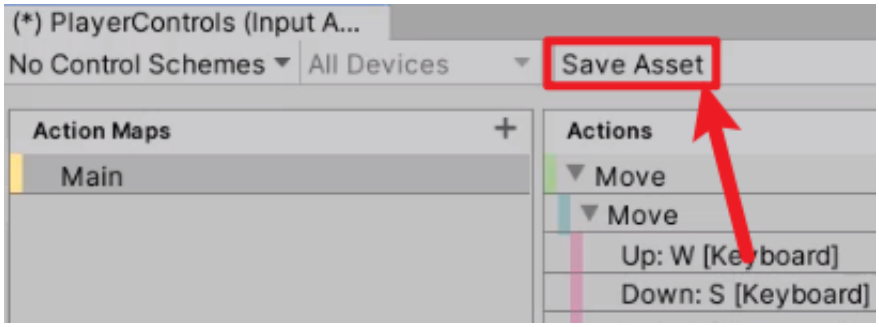


Refer to the table below to complete all the control schemes for our Arcade Karting game:

Accelerate	Up Arrow [Keyboard]
Accelerate	Button South [Gamepad]
Turn	Left Stick/X [Gamepad]
Turn	Left/Right Arrow (-/+) [Keyboard]

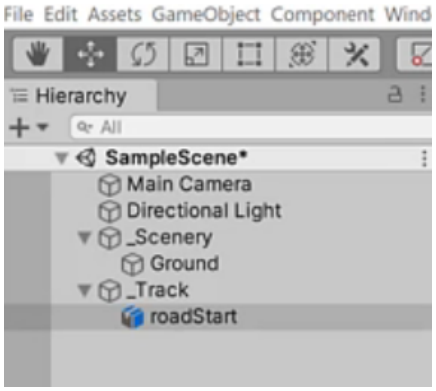


Make sure to click **Save Asset** before closing the window.

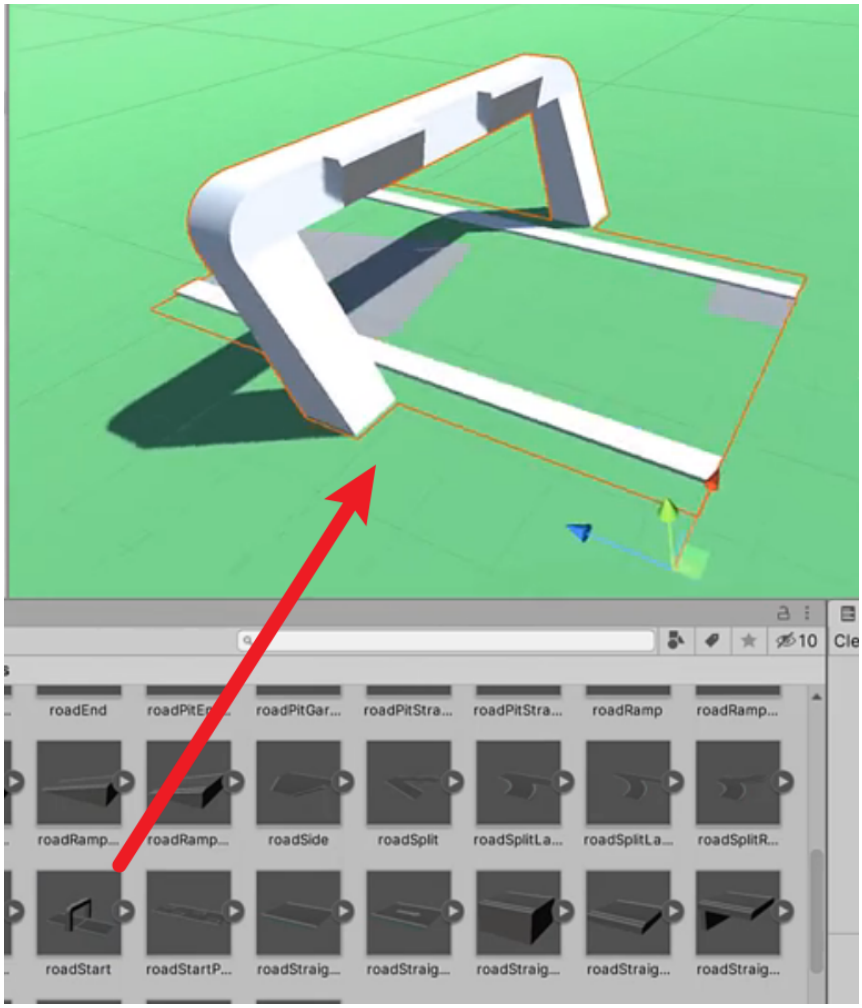


In this lesson, we're going to be **setting up the track** for our game. The **Track** is going to act as the grass where the players' speed gets slower when they land on it.

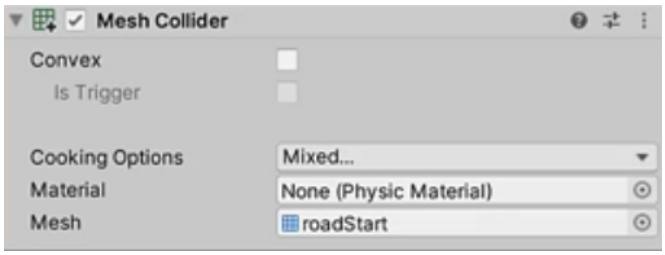
To begin, we need to set up a **Ground** for the track to lay on top of. So first of all, we're going to create two new **empty** game objects called "**_Scenery**" and "**_Track**". These are going to act as containers that hold all of the relevant objects as children. (e.g. Ground = Scenery)

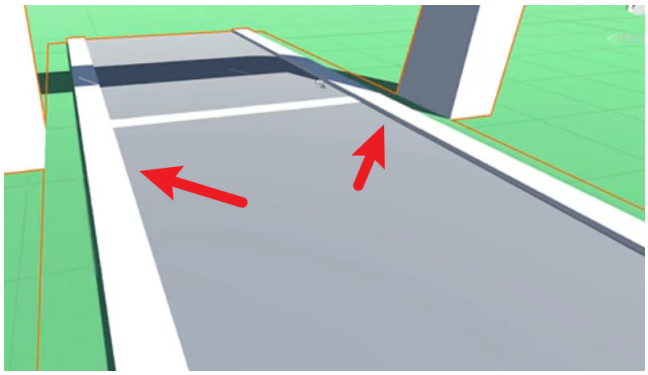


Then we'll drag the "**roadStart"** model from the **Assets > Models > Roads** folder into the scene, ensuring that it is a child of the "**_Track**" object.

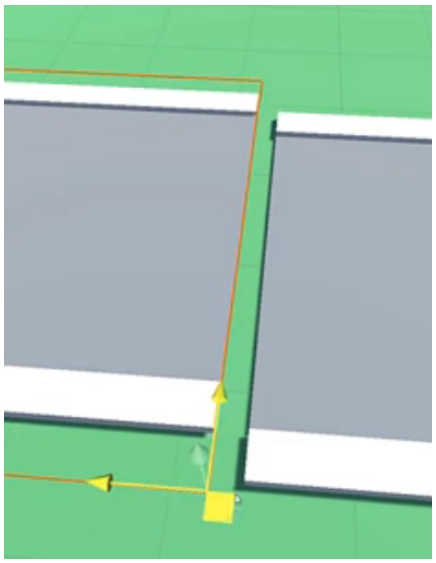


Make sure to add a **Mesh Collider** component to the **roadStart** object so that the players can bounce off the side edges.

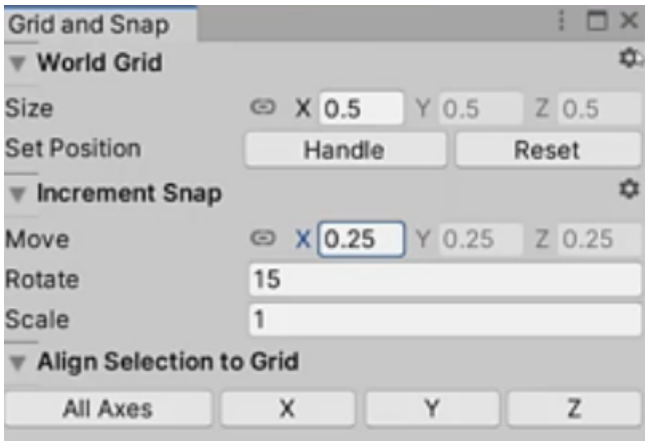




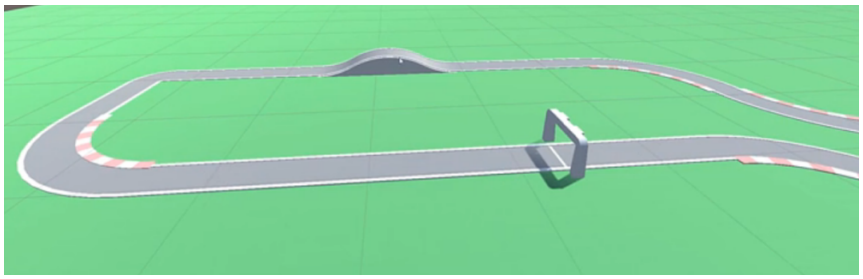
When moving additional road parts, you can hold down the **Ctrl** key to **Snap** it to the existing track.

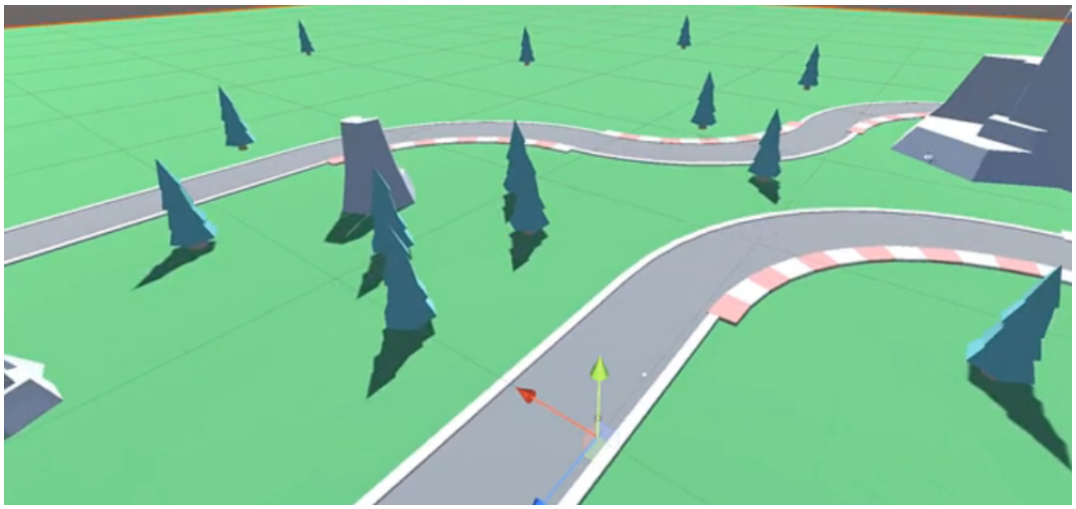


If it doesn't get perfectly aligned, you can go to **Edit > Grid and Snap Settings…** and lower the value of the **Increment Snap** property (e.g. from 1 to 0.25). This will allow moving and snapping in terms of smaller units.



With that in mind, let’s go ahead and add more to the scene. Feel free to change the track and the scenery to however you want.

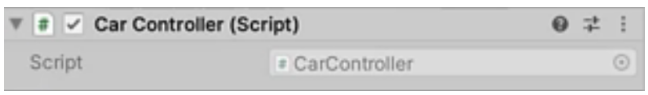




In this lesson, we're going to set up a new GameObject for our car.

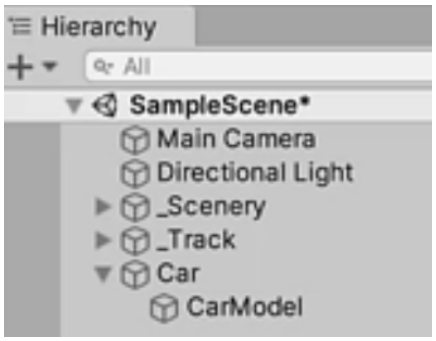
To begin, let's create a new **empty** GameObject called "Car". Then we want to add a **Rigidbody** component for the physics, a **Sphere Collider** for the collision, and a **Player Input** for detecting inputs.

We also need to attach a new **C# script**, which will detect inputs and enable us to drive the car around. Let's call this script "CarController".

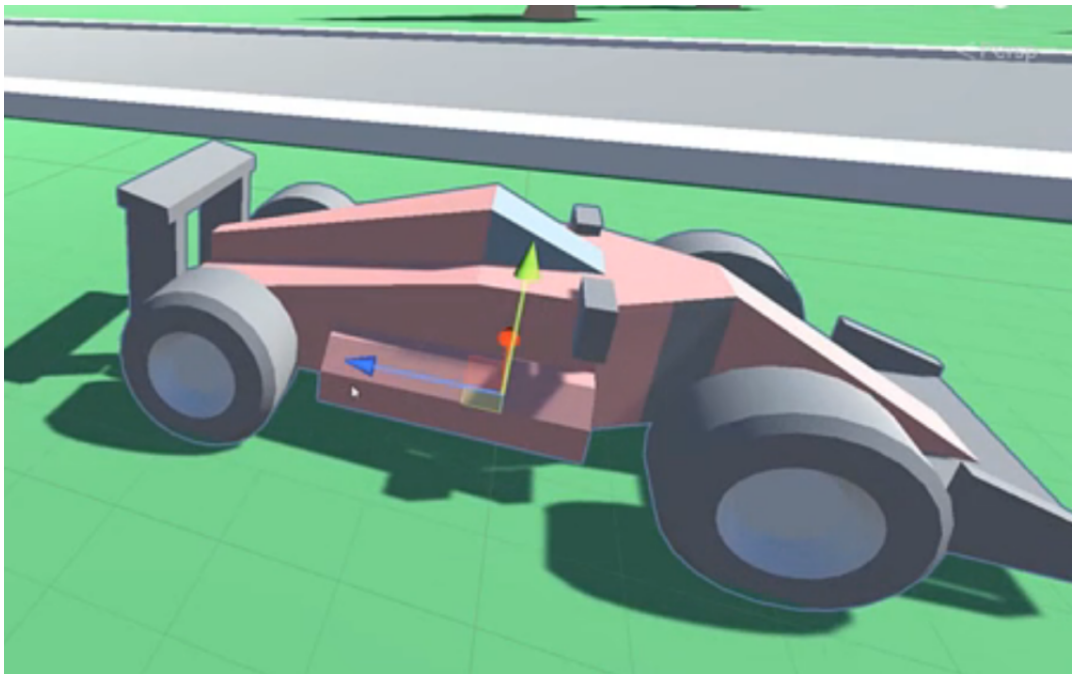


Next, we're going to **right-click** on our Car object and click on **Create Empty** to create a new empty child object. This is going to allow our car model to maintain upright rotation (instead of

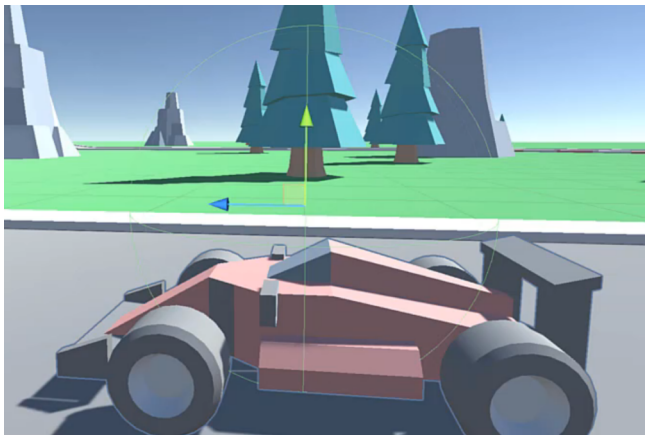
spinning around with the parent Car object).



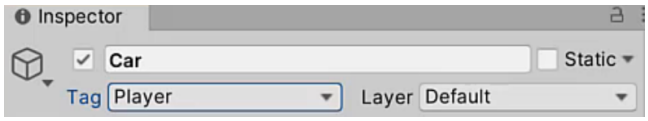
The car models can be found in the **Course Files**, or inside the Unity project if you have already imported them into the **Assets > Models > Cars** folder. Let's drag it into our CarModel object and ensure that the model's **Y-rotation** is set correctly and facing forward.



Adjust the **Sphere Collider** to match the model's size. The model should be positioned at the bottom of the sphere.



Make sure that the car object has the “Player” tag assigned.

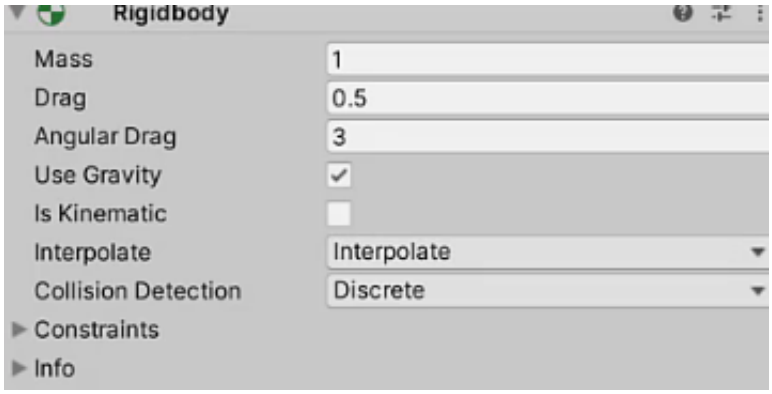


Rigidbody Settings

The Rigidbody component controls the car’s position through physics simulation. By default, it is set to be pulled downward by gravity and to react to collisions with other objects.

To simulate a realistic car’s behavior, we’re going to adjust these properties:

- **Drag**: 0 → 0.5 (This acts as wind resistance/ground friction in forwarding momentum)
- **Angular Drag**: 0.05 → 3 (Same as Drag, but for rotation)
- **Interpolate**: None → Interpolate (Smoothes out the effect of running physics at a fixed frame rate)

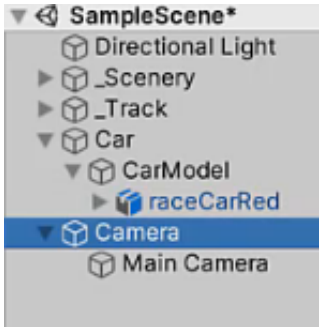


Feel free to tweak the values to how you like.

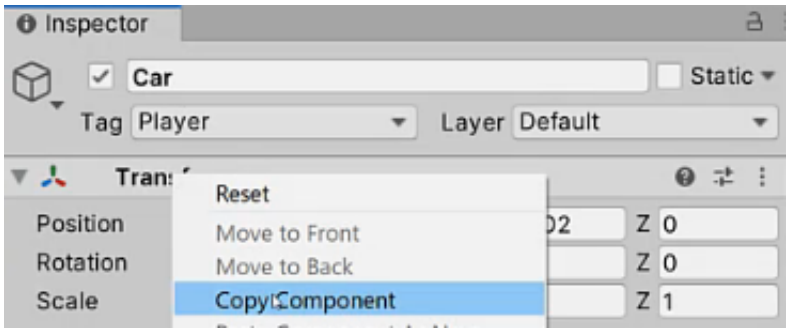
Camera Settings

The Main Camera in the scene will be moving based on the car’s position and rotation, keeping a certain distance away from the car. We’re going to call this ‘**Camera Offset**‘.

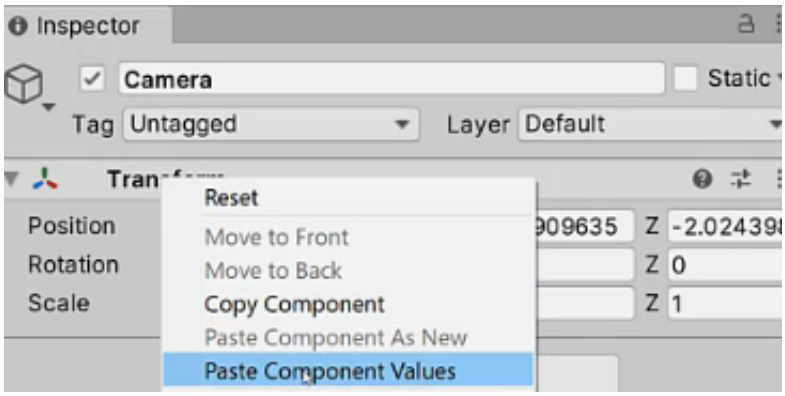
First, we need to create an empty object that will contain our camera as a child object.



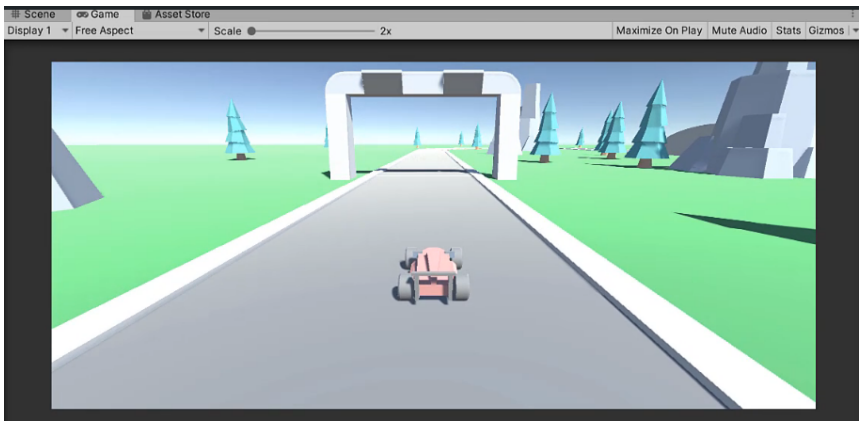
For the Main Camera to orbit around the car, we need to have the parent container object located at the exact same position as the car. We can copy the position values of the car by **Right-click** on **Transform** > **Copy Component**,



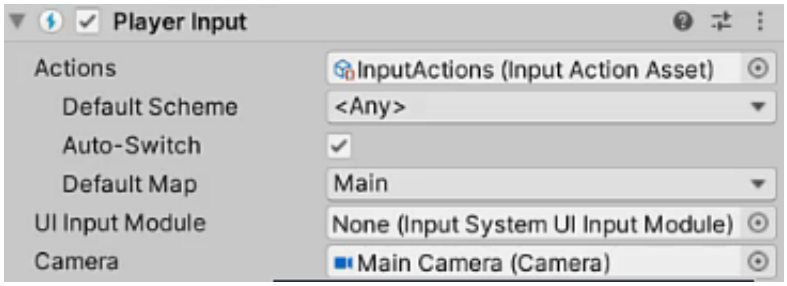
Then, we can go to the Main Camera object and right-click on **Transform > Paste Component Values**.



Now you can tweak the position and rotation values of the Main Camera to set the default camera angle.

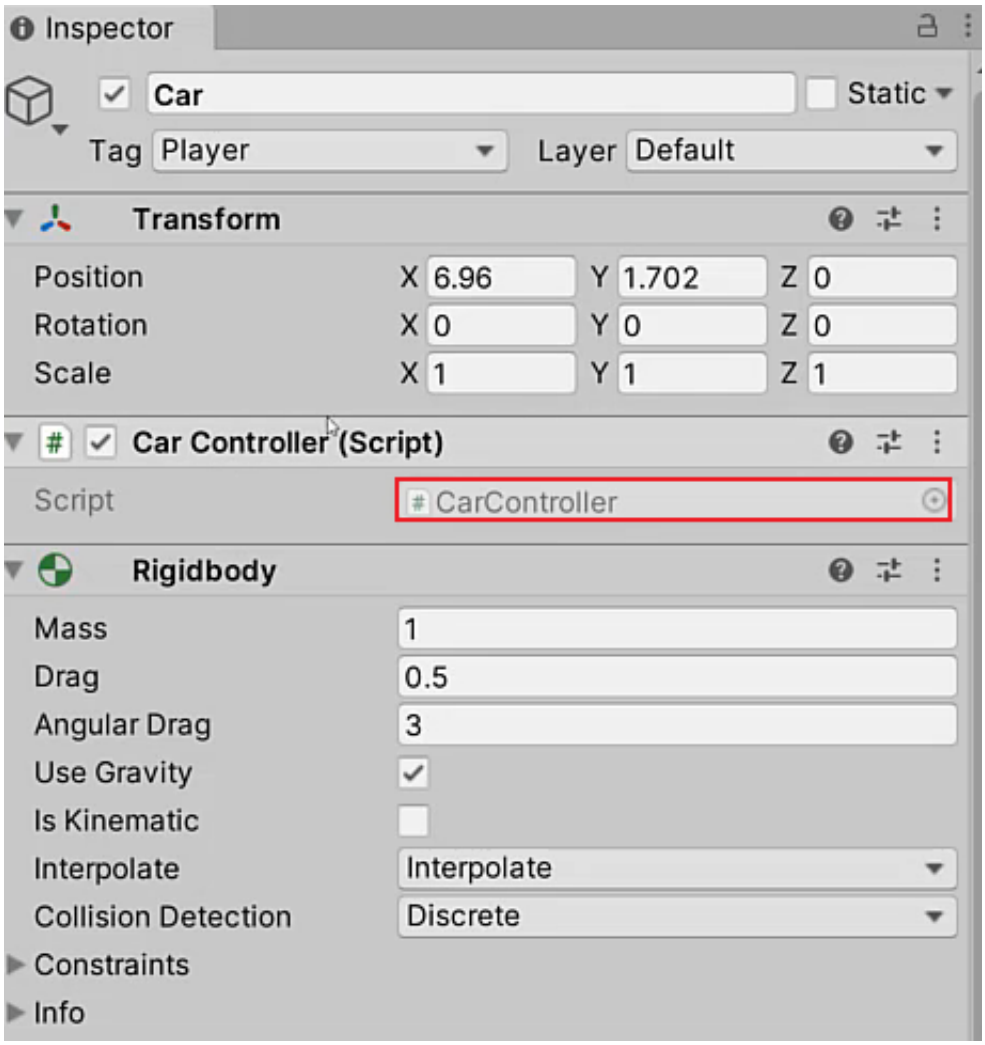


Make sure that you have assigned the **Input Action Asset** and the **Main Camera** to the **Player Input** component.



In this lesson, we're going to begin scripting our **CarController**.

Make sure that the script is attached to our **Car** GameObject, and let's start editing the script.



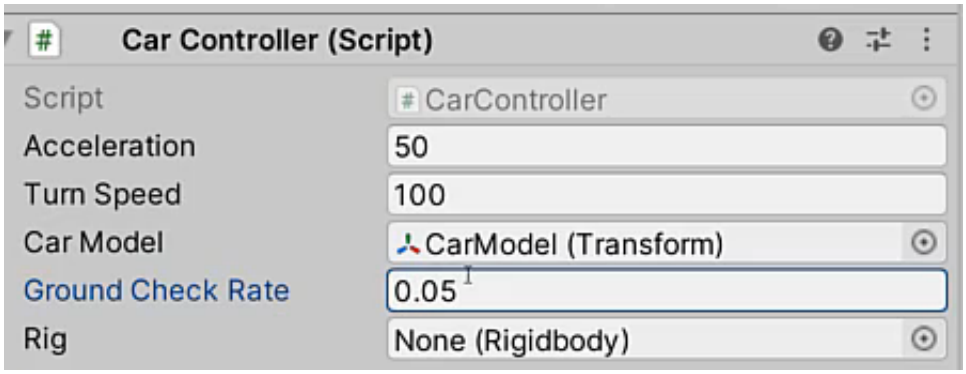
Creating Variables

To control our car, we need to define some variables. Take a look at the list below for a snapshot of each variable:

- **Acceleration (float)** – the rate at which the car accelerates forward
- **Turn speed (float)** – the rate at which the car rotates left and right when steering
- **Car model (Transform)** – the reference to the car's model, which we want to keep stationary in terms of rotation.
- **Start Model Offset (Vector3)** – The distance offset from the car model to its parent object to update the model's position on every frame.
- **Ground Check Rate (float)** – The rate of raycasting in order to determine if the car is on the ground.
- **Last Ground Check Time (float)** – The time the previous ground check was performed.
- **Current Y Rotation (float)** – The current Y rotation of the car model.
- **Accelerate Input (bool)** – Input state for acceleration
- **Turn Input (float)** – Input state for turning
- **Rigidbody**

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

public class CarController : MonoBehaviour
{
    public float acceleration;
    public float turnSpeed;

    public Transform carModel;
    private Vector3 startModelOffset;

    public float groundCheckRate;
    private float lastGroundCheckTime;

    private float curYRot;

    public bool canControl;

    private bool accelerateInput;
    private float turnInput;

    public TrackZone curTrackZone;
    public int zonesPassed;
    public int racePosition;
    public int curLap;

    public Rigidbody rig;
}
```

Once all the variables are declared, we can save the script, go back to the editor, and set up the public variables' values in the Inspector.



Detecting Inputs

Since we're using **UnityEngine.InputSystem**, we can create some functions that we can connect to the **Player Input** component attached to the **Car** object.

The name of the function can be whatever you want, but it should have a parameter of **InputAction.CallbackContext**.

For example, let's start working on the function to be called when the **Accelerate** input is detected:

```
// called when we press down the accelerate input
public void OnAccelerateInput (InputAction.CallbackContext context)
{
}
```

Here, the "context" sends over all the information regarding the input, such as if the button has just been pressed down, the duration of holding down the key, etc.

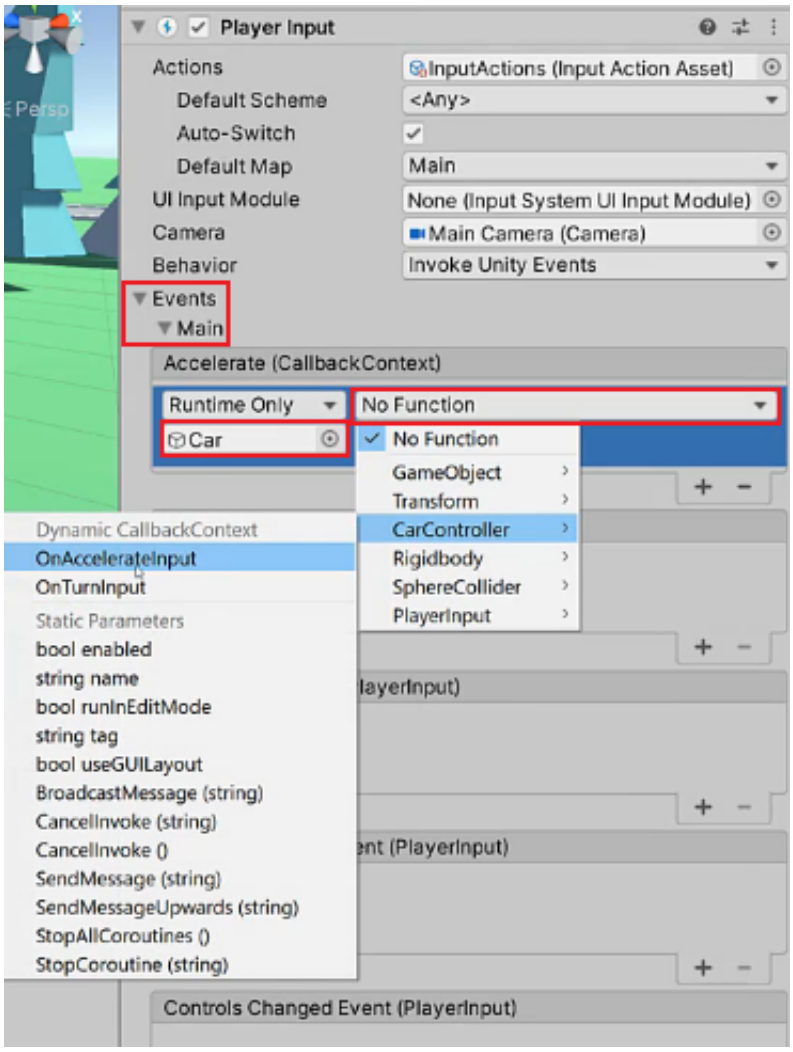
By checking if its **phase** is **InputActionPhase.Performed**, we can ensure that the key is being pressed down.

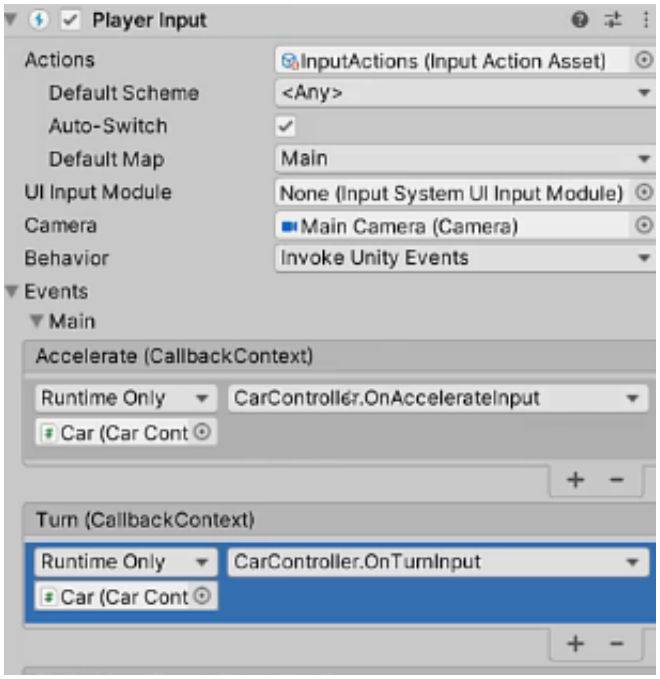
```
// called when we press down the accelerate input
public void OnAccelerateInput (InputAction.CallbackContext context)
{
    if(context.phase == InputActionPhase.Performed)
        accelerateInput = true;
    else
        accelerateInput = false;
}
```

Similarly, we can read the turn input as a float value (negative = left, positive = right).

```
// called when we modify the turn input
public void OnTurnInput (InputAction.CallbackContext context)
{
    turnInput = context.ReadValue<float>();
}
```

Let's save the script, and bind the input functions with specific keys. Go to the **Player Input** component, assign the **Car** object to the input events (under **Events > Main**), and select both **OnAccelerateInput** and **OnTurnInput**.





Now, these event functions are connected to the keys which we have mapped in our **Input Action Asset**.

In this lesson, we're going to continue working with the **CarController** script.

Defining Offset

Right now, we have the car's model attached under its parent **Car** GameObject. In order to **fix** the model's position at the current offset from the parent, we're going to store the initial **localPosition** in a **Vector3** variable, called **startModelOffset**.

So let's define the **Start()** function to set the **startModelOffset**'s value:

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

public class CarController : MonoBehaviour
{
    ...
    private Vector3 startModelOffset;
    void Start ()
    {
        startModelOffset = carModel.transform.localPosition;
    }
}
```

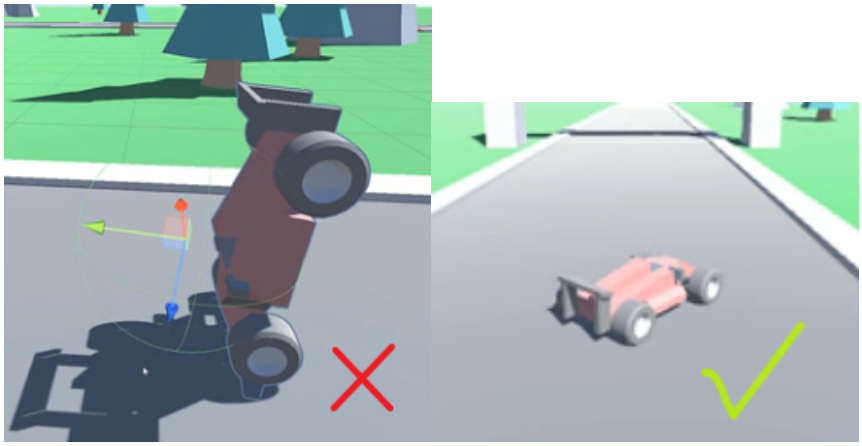
Then we can define the **FixedUpdate** function to **add force** to our car. Remember, **FixedUpdate** runs 60 times per second consistently (whereas **Update** runs every single frame), and hence it is useful for physics calculations.

Here, we're only adding force into the **forward** direction if we have both '**canControl**' and '**accelerateInput**' as true.

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

public class CarController : MonoBehaviour
{
    ...
    public float acceleration;
    public bool canControl;
    private bool accelerateInput;
    void FixedUpdate ()
    {
        // don't accelerate if we don't have control
        if(!canControl)
            return;

        if(accelerateInput == true)
        {
            rig.AddForce(carModel.forward * acceleration, ForceMode.Acceleration);
        }
```

```
    }
}
```

Our car model should only rotate around the **y-axis** based on our **turn input**. We can directly replace the **y-rotation** to reflect our **turnInput**, **turnSpeed,** and **turnRate**, which is changing over **Time.deltaTime**.



So let's manually update the rotation inside **Update**:

```
public class CarController : MonoBehaviour
{
    public float acceleration;
    public float turnSpeed;

    public Transform carModel;
    private Vector3 startModelOffset;

    public float groundCheckRate;
    private float lastGroundCheckTime;

    private float curYRot;

    public bool canControl;

    private bool accelerateInput;
    private float turnInput;

    public TrackZone curTrackZone;
    public int zonesPassed;
    public int racePosition;
    public int curLap;

    public Rigidbody rig;

    void Start ()
    {
        startModelOffset = carModel.transform.localPosition;
    }
```

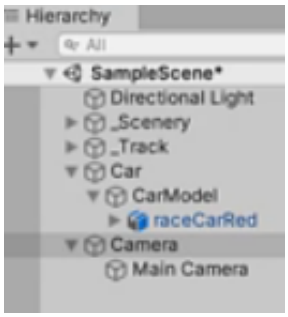


```
    void Update ()
    {
        curYRot += turnInput * turnSpeed  * turnRate * Time.deltaTime;

        carModel.position = transform.position + startModelOffset;
        carModel.eulerAngles = new Vector3(0, curYRot, 0);
    }

    void FixedUpdate ()
    {
        // don't accelerate if we don't have control
        if(!canControl)
            return;

        if(accelerateInput == true)
        {
            rig.AddForce(carModel.forward * acceleration, ForceMode.Acceleration);
        }
    }
}
```

In this lesson, we're going to be setting up our **CameraController**.

Let's begin by creating a new **C# script** called **CameraController**, and attaching it to an empty GameObject.

We'll call this empty object "**Camera**", and make it contain our **Main Camera** as a child.



Declaring Variables

First, we need to define a **target (Transform)** to follow.

Then we can move and rotate our camera based on **followSpeed** and **rotateSpeed**.

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

public class CameraController : MonoBehaviour
{
    public Transform target;

    public float followSpeed;
    public float rotateSpeed;
}
```

LERP (Linear Interpolation)

When updating our camera's position and rotation, we don't want to snap it to follow the target immediately. Instead, we gradually **ease in/out** at the designated speed.

One way to achieve this effect is by using **Lerp** (short for **L**inear Int**erp**olation). Lerp is a useful mathematic function built in Unity, which takes the three parameters:

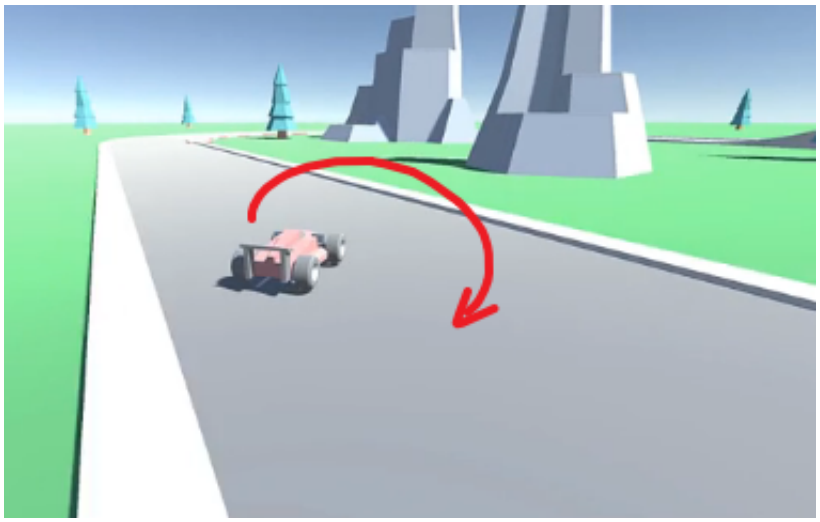
Lerp(a, b, t) where **a** = **start value, b** = **end value**, and **t** = **interpolant**, meaning it is a value used to interpolate between **a** and **b.**

This function can work on a **float**, **Vector3**, **Color**, or even **Quaternion**. So for our camera's position to follow the target, we can use **Vector3.Lerp**. For our camera's rotation, we can use **Quaternion.Lerp**, to smoothly ease in/out when following the target.

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

public class CameraController : MonoBehaviour
{
    public Transform target;

    public float followSpeed;
    public float rotateSpeed;

    void Start ()
    {
        transform.parent = null;
    }

    void Update ()
    {
        transform.position = Vector3.Lerp(transform.position, target.position, follow
Speed * Time.deltaTime);
        transform.rotation = Quaternion.Lerp(transform.rotation, target.rotation, rot
ateSpeed * Time.deltaTime);
    }
}
```


In this lesson, we're going to be improving upon our steering.

Right now, when we turn our car keeps rotating at exact same rate, skidding out of control.



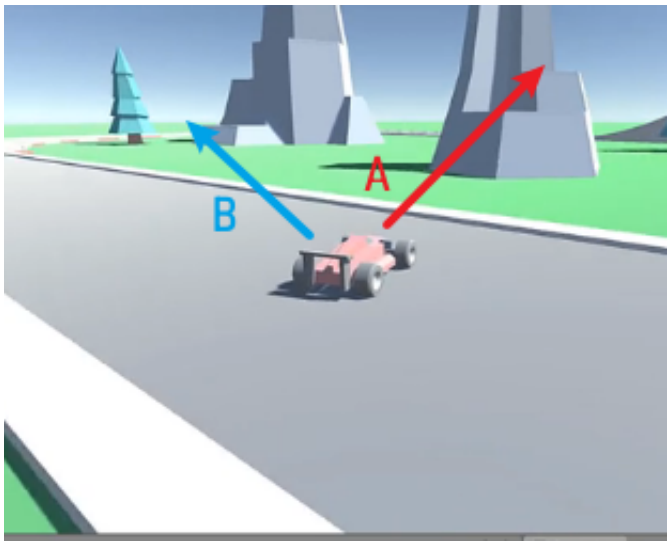
So to fix this, we're going to restrict the player's ability to steer. Let's go back to the **Update** function of our **CarController** script.

Our speed should get slower the further away we're facing from the direction we're moving in. But how can we possibly measure this?

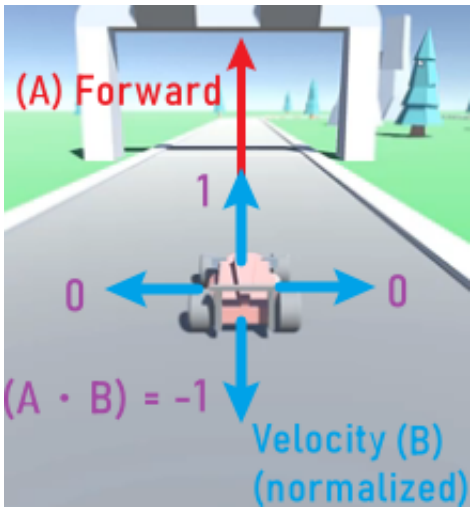
Dot Product

The direction we're currently facing can be expressed by: **(A) carModel.forward**.

The direction we're currently moving in can be expressed by: **(B) rig.velocity.normalized**. (A normalized vector has a magnitude of 1; this is useful when we want to regard only the direction of the vector.)



The **Dot Product** between the two vectors **(A · B)** will return a number between -1 and 1, based on the similarity of the two vectors' direction.



- If the two vectors are pointing in the **same** direction: the dot product is **1**.
- If they are pointing in the **opposite** direction: the dot product is **-1**.
- If they are **perpendicular** to each other: the dot product is **0**.
- (Note: At least one of the two vectors must be normalized for the dot product to be in the range of -1 to 1).

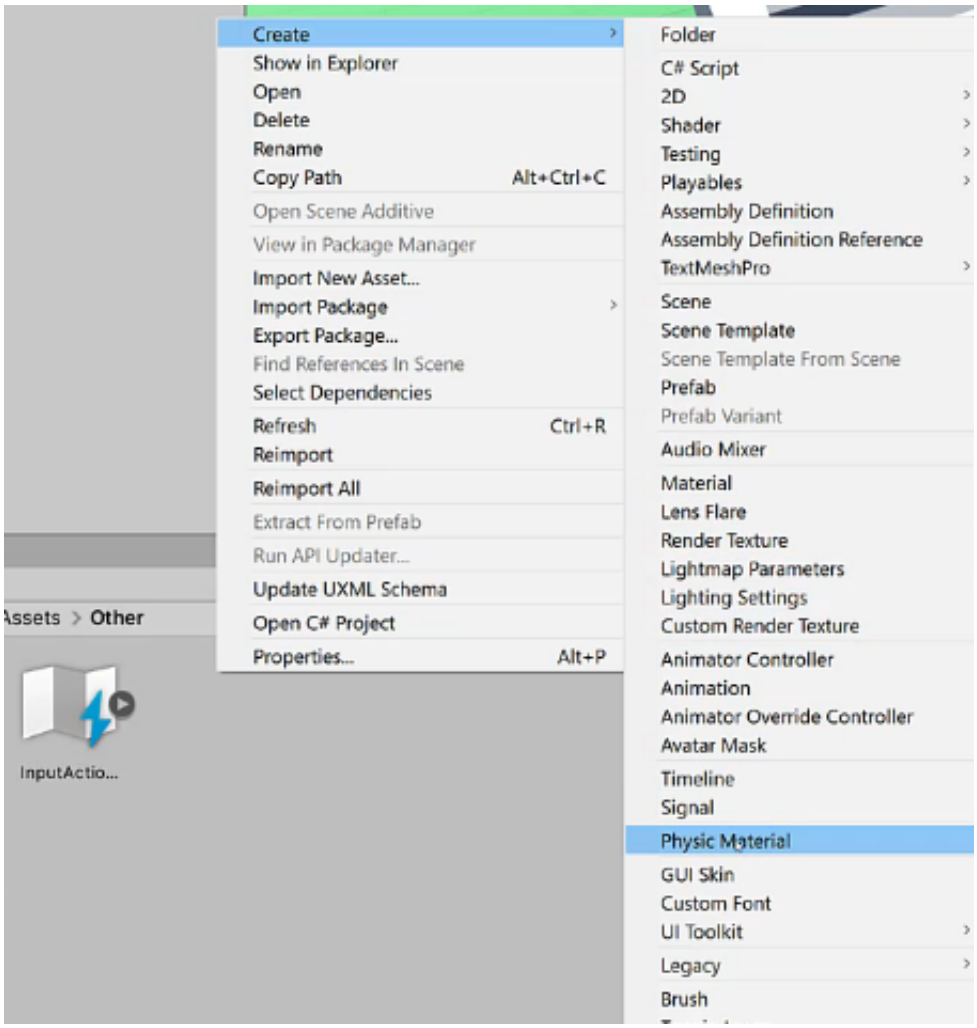
We can calculate the dot product between these two vectors using **Vector3.Dot(Vector3 a, Vector3 b)**. Then we can ensure that the value is always positive by using **Mathf.Abs(float value)**.

Having this multiplied by our **turnInput** will now allow us to limit the player's ability to turn, based on the direction they are moving in.

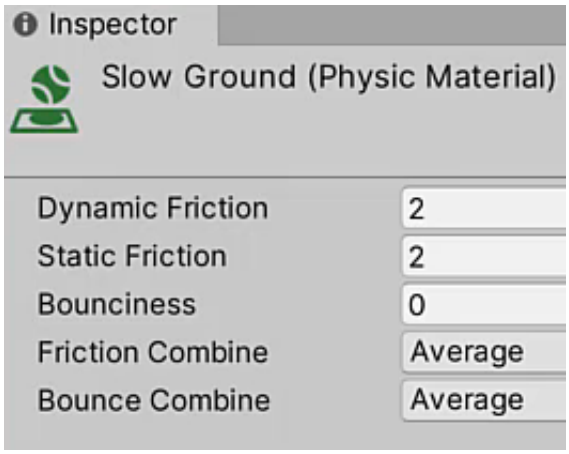
```
void Update ()
{
    ...
    // calculate the amount we can turn based on the dot product between our velocity
 and facing direction
    float turnRate = Vector3.Dot(rig.velocity.normalized, carModel.forward);
    turnRate = Mathf.Abs(turnRate);

    curYRot += turnInput * turnSpeed  * turnRate * Time.deltaTime;

    carModel.position = transform.position + startModelOffset;
    //carModel.eulerAngles = new Vector3(0, curYRot, 0);

    CheckGround();
}
```

Physics Material – Friction

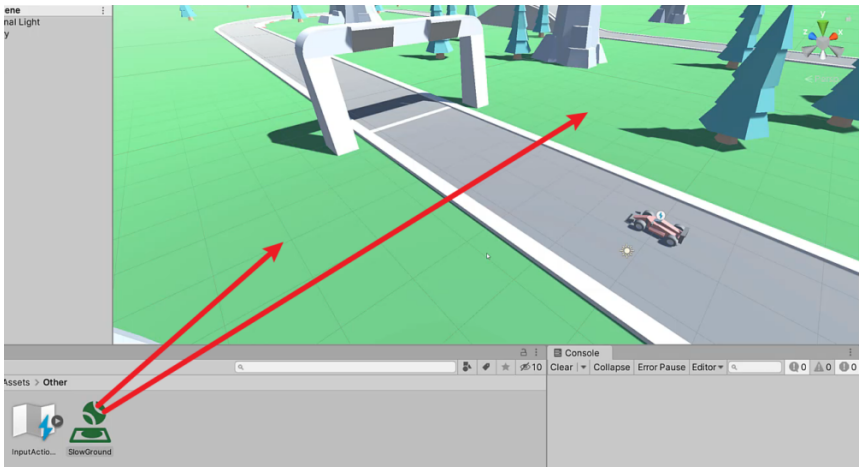
To add some more friction to the grass area, we can create a new **Physics Material**. (Right-click > Create > Physics Material)



In the Inspector we will increase the **Dynamic Friction** and **Static Friction** up to **2**.

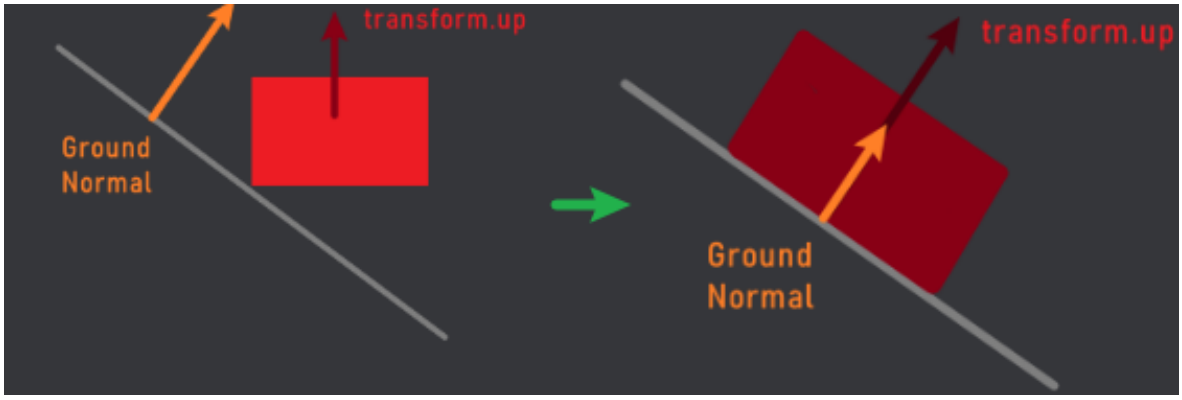


To apply it to the grass, simply drag it and drop it onto the grass.



In this lesson, we're going to be working on having our car stick to the track, instead of staying in a forward direction.

This is because our car needs to rotate based on the **normal** direction of the ground.

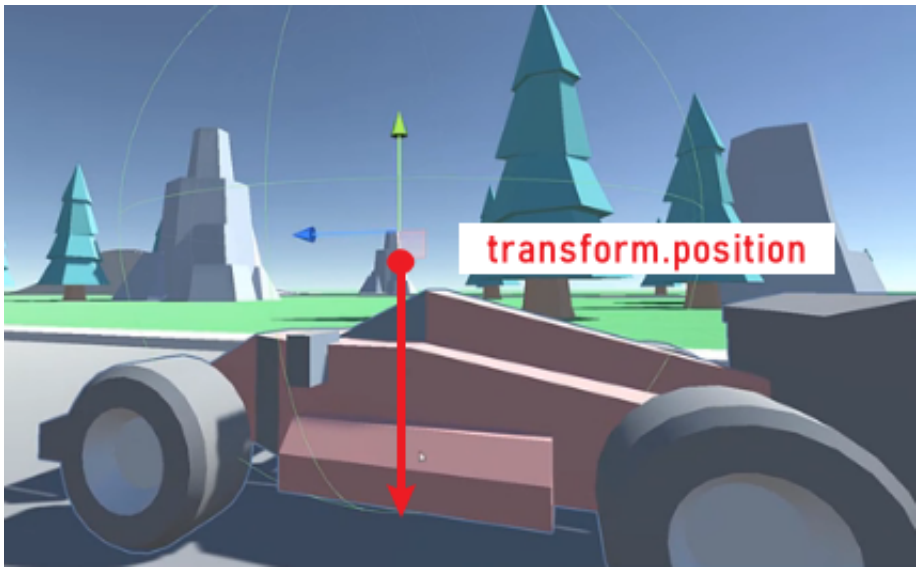


Checking Ground

First of all, we need to shoot a **Raycast** downwards from our car. Let's create a new function called **CheckGround**, and create a new **Ray**:

```
// rotate with the surface below us
void CheckGround ()
{
    Ray ray = new Ray(transform.position, Vector3.down);
}
```

Since our car's origin is in the center of the collider, it is likely that our ray is going to hit the **SphereCollider** attached to our car itself.



To avoid hitting our car's collider with the ray, we can add a **Y-offset** of the radius of the collider:

```
// rotate with the surface below us
void CheckGround ()
{
    Ray ray = new Ray(transform.position + new Vector3(0, -0.75f, 0), Vector3.down);
}
```

Then we can declare a **RaycastHit** variable, and see if the ray (of the given length) has hit anything.

If it hit the ground, then we're going to set our car's **up direction** to be aligned with the normal of the ground.

If it didn't hit anything, then we'll just set it to be pointing up in the world direction.

```
// rotate with the surface below us
void CheckGround ()
{
    Ray ray = new Ray(transform.position + new Vector3(0, -0.75f, 0), Vector3.down);
    RaycastHit hit;

    if(Physics.Raycast(ray, out hit, 1.0f))
    {
        carModel.up = hit.normal;
    }
    else
    {
        carModel.up = Vector3.up;
    }
}
```

Calling this at the end of the Update function will now override all our transform modifications, including the Y-rotation of the car. This means we can't steer or change direction anymore.

So we need to combine all of the above with our **current Y rotation**. We can do this by rotating our **carModel** by **curYRot** around the **Local Y-axis**.

The **Transform.Rotate** function allows us to specify whether to rotate it around the **local axis (Space.Self)** or the **global axis (Space.World)**.

```
// rotate with the surface below us
void CheckGround ()
{
    Ray ray = new Ray(transform.position + new Vector3(0, -0.75f, 0), Vector3.down);
    RaycastHit hit;

    if(Physics.Raycast(ray, out hit, 1.0f))
    {
        carModel.up = hit.normal;
    }
    else
    {
        carModel.up = Vector3.up;
    }

    carModel.Rotate(new Vector3(0, curYRot, 0), Space.Self);
}
```

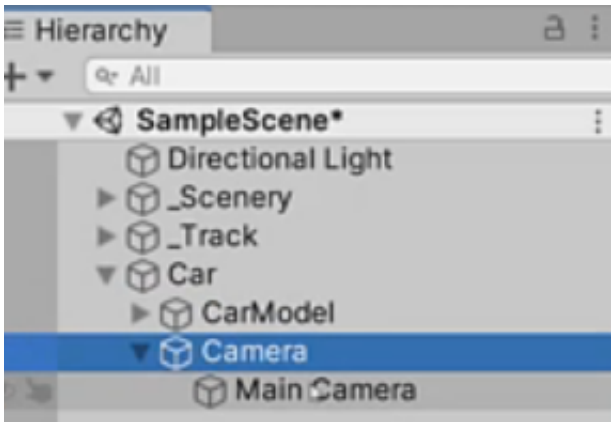
Now, calling the **CheckGround** function at the end of the Update function will allow our car to rotate with the track.

In this lesson, we're going to get multiplayer implemented into the game.

Setting Up Car Prefab

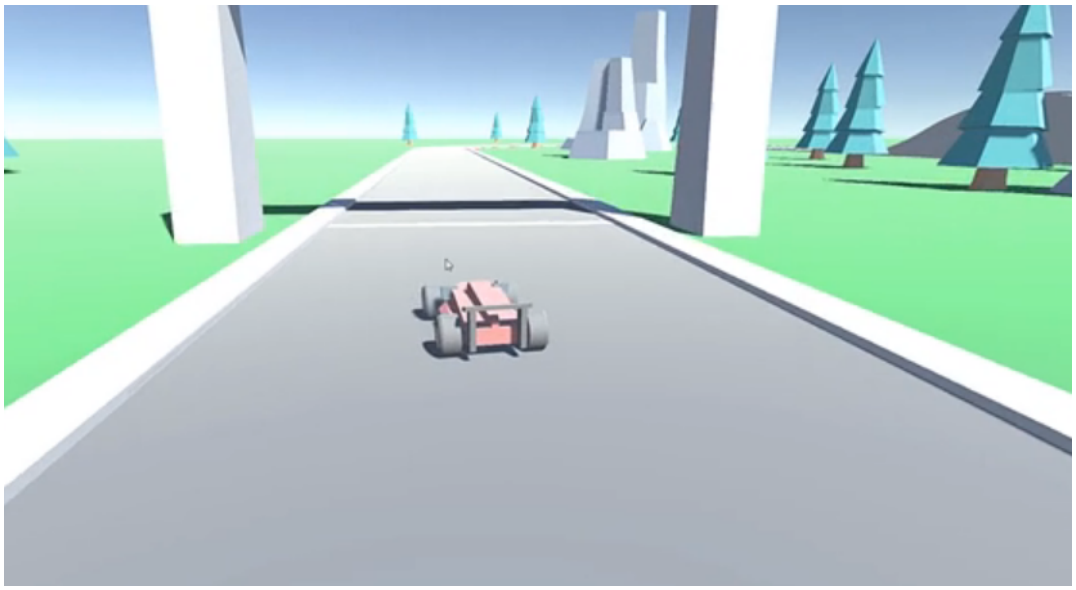
When a new controller is connected, we're going to spawn in a new **Car** object; this means we need to package the car as a **Prefab** so that we can instantiate it.

One thing to note is that we want to set the camera position so that it always puts the car at the center of the screen. So we'll have the **Camera** attached to our Car as a **child** object:

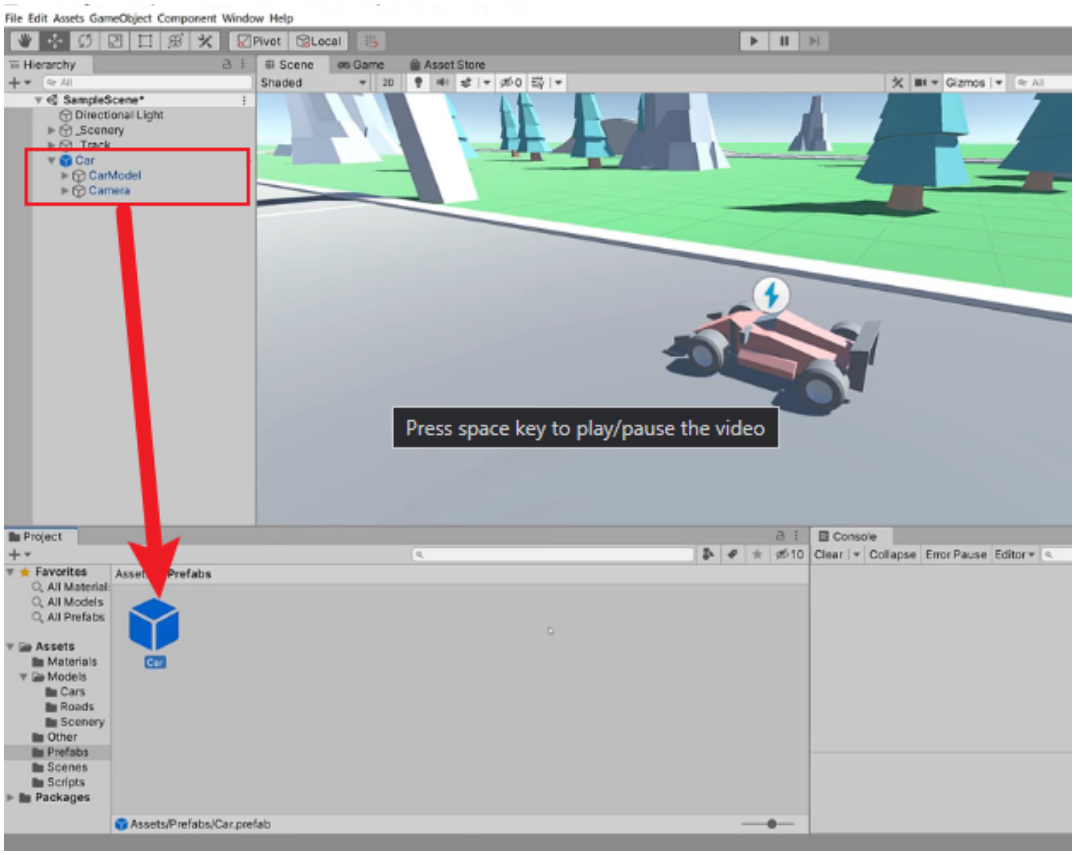


This allows us to set the camera position so that it always puts the car at the center of the screen. However, we need to detach the camera from the car at the start of the game. We can do this by opening up the **CameraController.cs**, and setting the **transform.parent** to 'null' inside the **Start** function.

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

public class CameraController : MonoBehaviour
{
    public Transform target;

    public float followSpeed;
    public float rotateSpeed;

    void Start ()
    {
        transform.parent = null;
    }

    void Update ()
    {
        transform.position = Vector3.Lerp(transform.position, target.position, follow
Speed * Time.deltaTime);
        transform.rotation = Quaternion.Lerp(transform.rotation, target.rotation, rot
ateSpeed * Time.deltaTime);
    }
}
```

We can then drag the **Car** (grab the parent object) into the **Project** window and convert it into a prefab. Once turned into a prefab, you can disable the object as we're going to spawn in a new car at runtime.



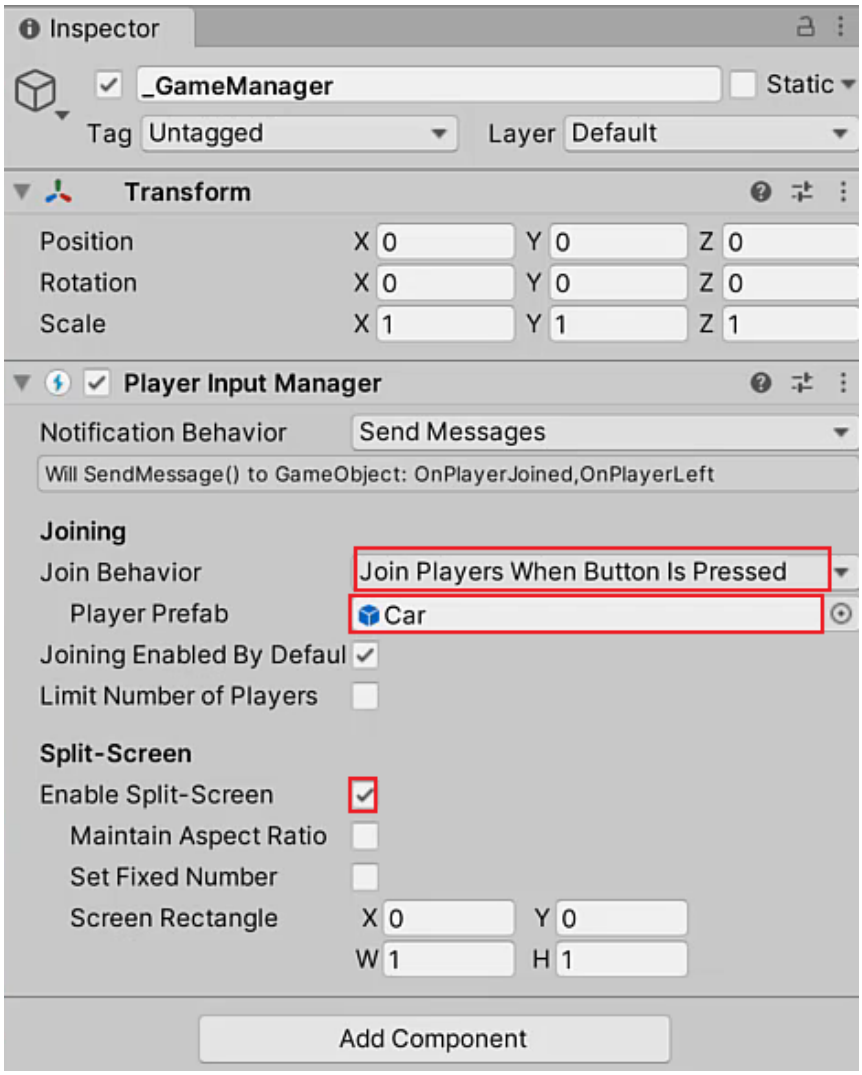
Splitting Screen

Let's create a new empty game object called "**_GameManager**", and add the **Player Input Manager** component.

The **PlayerInputManager** component can detect multiple different input devices (e.g. controllers,

keyboards, mice, etc.). It can also spawn in a prefab when a new controller has been detected.

Make sure that our **Car** prefab is assigned, and that **Enable Split-Screen** is ticked.



GameManager

Let's create a new C# script called "**GameManager**" and attach it to the **_GameManager** object.

Here, we'll set up a **Singleton** so that **GameManager** class can be accessed by other scripts as well.

We'll also create a new **List** of **CarController** so we can keep track of the controllers that are connected.

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

public class GameManager : MonoBehaviour
{
    public List<CarController> cars = new List<CarController>();
    public static GameManager instance;
```



```
    void Awake ()
    {
        instance = this;
    }
}
```

Note that **Awake** function is called before **Start** function, and hence it helps us ensure that the singleton instance is assigned before it gets called by other scripts.

In this case, we're going to access the GameManager instance from **CarController.cs,** so that it adds the controller class itself to the list.

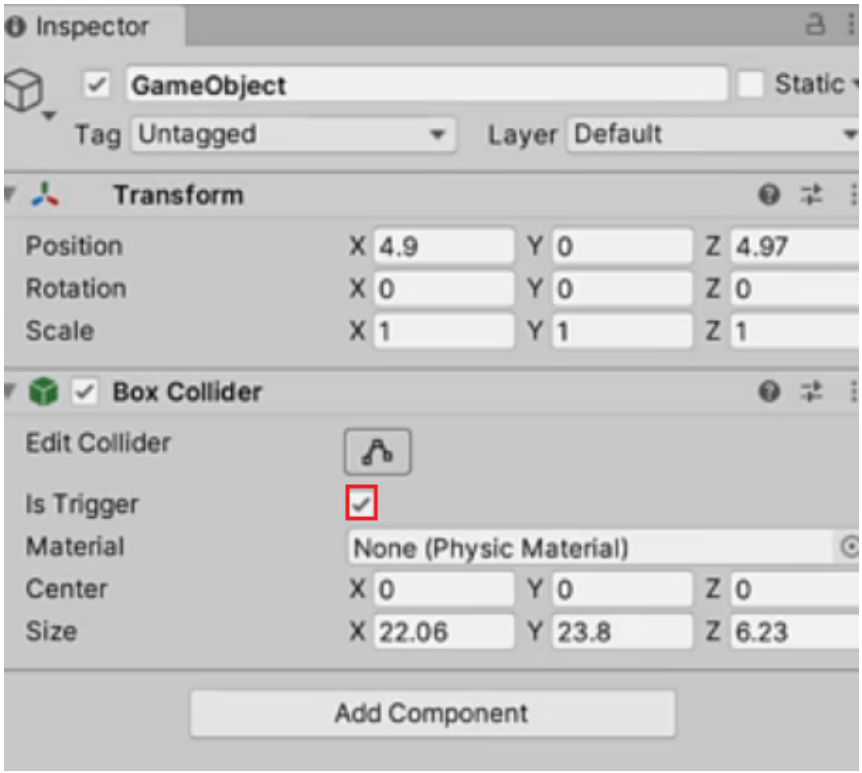
```
void Start ()
{
    startModelOffset = carModel.transform.localPosition;
    GameManager.instance.cars.Add(this);
    transform.position = GameManager.instance.spawnPoints[GameManager.instance.cars.C
ount - 1].position;
}
```

Every time a new **CarController** instance is created, it will add itself to the list.

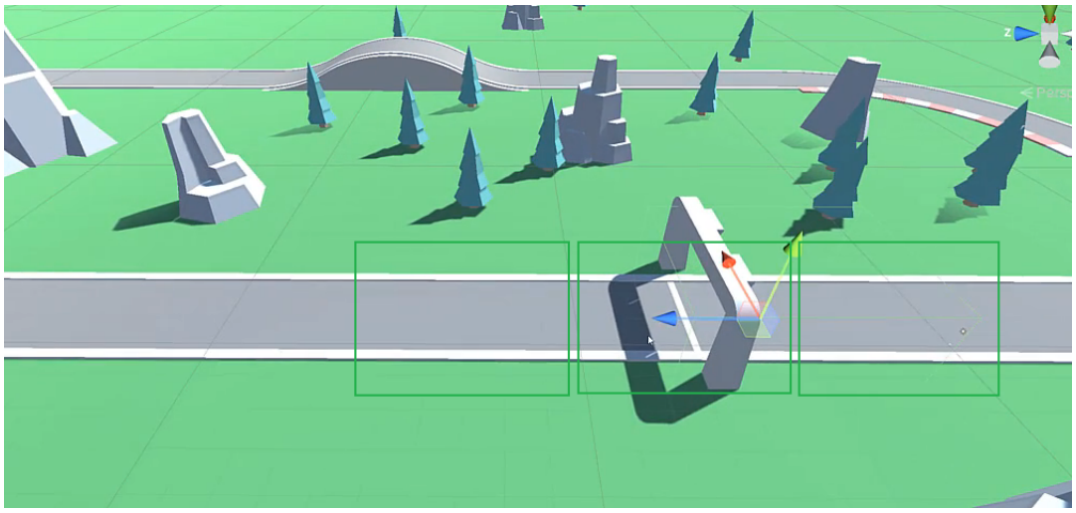
In this lesson, we're going to set up the abilities for the players to keep track of where they are along the track.

For this, we'll divide some sections of the track that the players need to pass through, and label them as different **Track Zones**.

Let's create a new empty object and add a **Box Collider** component. We should then increase the size to match the size of the track and enable **IsTrigger** so that players don't collide with the box.



Now we can **duplicate** this to cover up the other areas of the track:



Inside the **CarController** script, we will declare a TrackZone variable to see which zone the player is currently in.

We also need three more **int** variables to store:

- **zonesPassed**: how many zones has the player passed?
- **racePosition**: which position is the player currently in? (1st, 2nd, 3rd..)
- **currentLap**: how many laps has the player passed?

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

public class CarController : MonoBehaviour
{
    public float acceleration;
    public float turnSpeed;

    public Transform carModel;
    private Vector3 startModelOffset;

    public float groundCheckRate;
    private float lastGroundCheckTime;

    private float curYRot;

    public bool canControl;

    private bool accelerateInput;
    private float turnInput;

    public TrackZone curTrackZone;
    public int zonesPassed;
    public int racePosition;
    public int curLap;

    public Rigidbody rig;

    ...
}
```

We will then increment these values from the **TrackZone** script inside **OnTriggerEnter**. Whenever a player enters the zone, we want to set the **curTrackZone** to be **this**, and also increment **zonePassed** by one.

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

public class TrackZone : MonoBehaviour
{
    public bool isGate;

    private void OnTriggerEnter (Collider other)
    {
        if(other.CompareTag("Player"))
        {
```



```
            CarController car = other.GetComponent<CarController>();
            car.curTrackZone = this;
            car.zonesPassed++;
        }
    }
}
```


In this lesson, we're going to set it up so that we can start determining which player comes first in the race.

Let's open up our **GameManager** script, and create two new variables:

- public float **positionUpdateRate**: the minimum time between updates of the players' positions.
- private float **lastPositionUpdateTime**

Then we're going to update our car race positions every **positionUpdateRate**.

We can assign **Time.time** (the current time since the game has started) to our **lastPositionUpdateTime** to repeat this.

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

public class GameManager : MonoBehaviour
{
    public List<CarController> cars = new List<CarController>();

    public float positionUpdateRate = 0.05f;
    private float lastPositionUpdateTime;

    public static GameManager instance;

    void Awake ()
    {
        instance = this;
    }

    void Update ()
    {
        // update the car race positions
        if(Time.time - lastPositionUpdateTime > positionUpdateRate)
        {
            lastPositionUpdateTime = Time.time;
            UpdateCarRacePositions();
        }
    }
}
```

We then need to define the **UpdateCarRacePositions** function so that it **sorts** the **cars** list based on their position on the track. By default, the list gets ordered based on who entered the game first.

We want to sort the list based on **SortPosition**, which is basically an **integer** that is either **1 or -1**. It will be **1** if CarController '**a**' is ahead of the CarController '**b**', and **-1** if '**b**' is ahead of '**a**'.

```
// updates which car is coming first, second, etc
void UpdateCarRacePositions ()
{
    cars.Sort(SortPosition);
```

```
}

int SortPosition (CarController a, CarController b)
{
    if(a.zonesPassed > b.zonesPassed)
        return 1;
    else if(b.zonesPassed > a.zonesPassed)
        return -1;

    float aDist = Vector3.Distance(a.transform.position, a.curTrackZone.transform.pos
ition);
    float bDist = Vector3.Distance(b.transform.position, b.curTrackZone.transform.pos
ition);

    return aDist > bDist ? 1 : -1;
}
```

The **Sort** function is going to automatically sort all the elements inside the **cars** list based on the outcomes of this **SortPosition** function.

Once we've done that, we can loop through the list and assign the positions to the cars' **racePositions**.

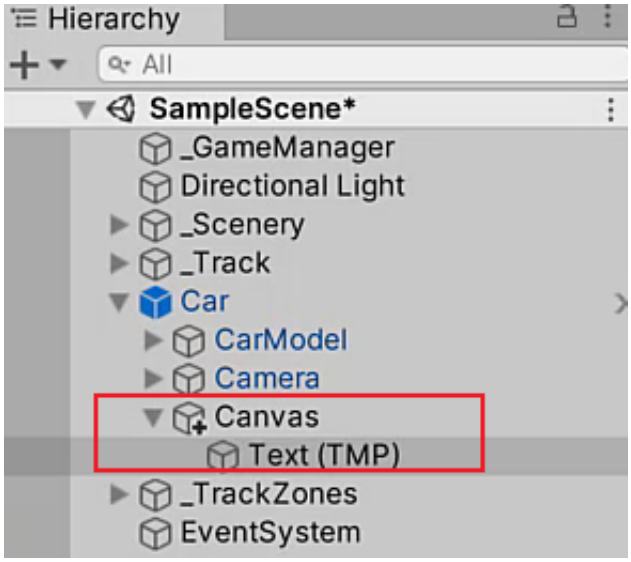
The first place (1) will be equal to **cars.Count**, and the second place (2) will be equal to **cars.Count – 1**, and the nth place will be equal to **cars.Count – n**, and so on.

```
// updates which car is coming first, second, etc
void UpdateCarRacePositions ()
{
    cars.Sort(SortPosition);

    for(int x = 0; x < cars.Count; x++)
    {
        cars[x].racePosition = cars.Count - x;
    }
}

int SortPosition (CarController a, CarController b)
{
    if(a.zonesPassed > b.zonesPassed)
        return 1;
    else if(b.zonesPassed > a.zonesPassed)
        return -1;

    float aDist = Vector3.Distance(a.transform.position, a.curTrackZone.transform.pos
ition);
    float bDist = Vector3.Distance(b.transform.position, b.curTrackZone.transform.pos
ition);

    return aDist > bDist ? 1 : -1;
}
```

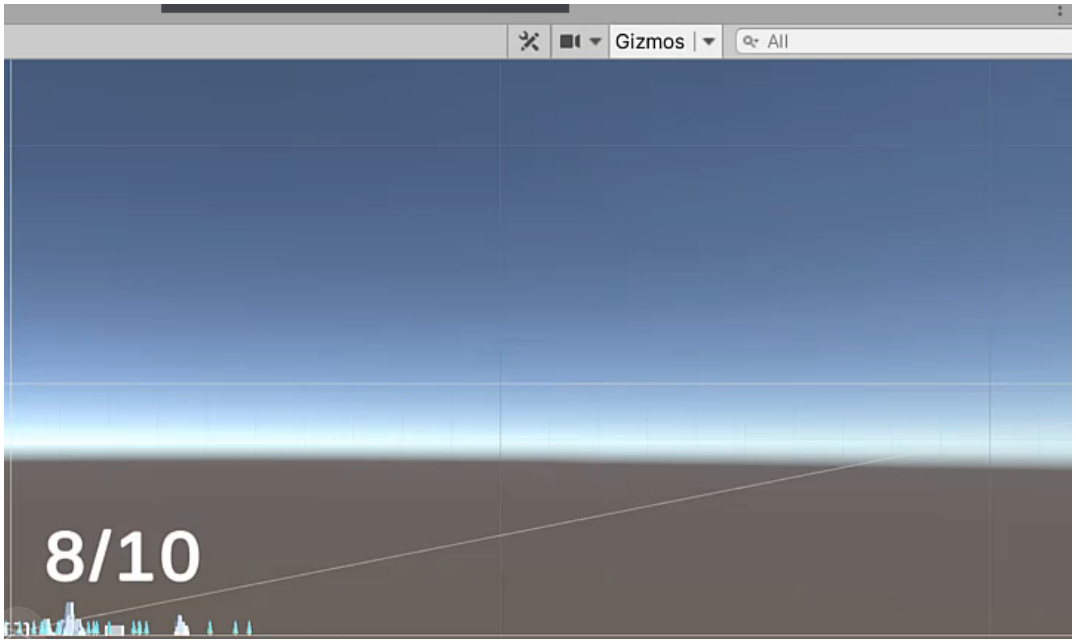
In this lesson, we're going to be setting up the **UI** for our game.

Race Position UI

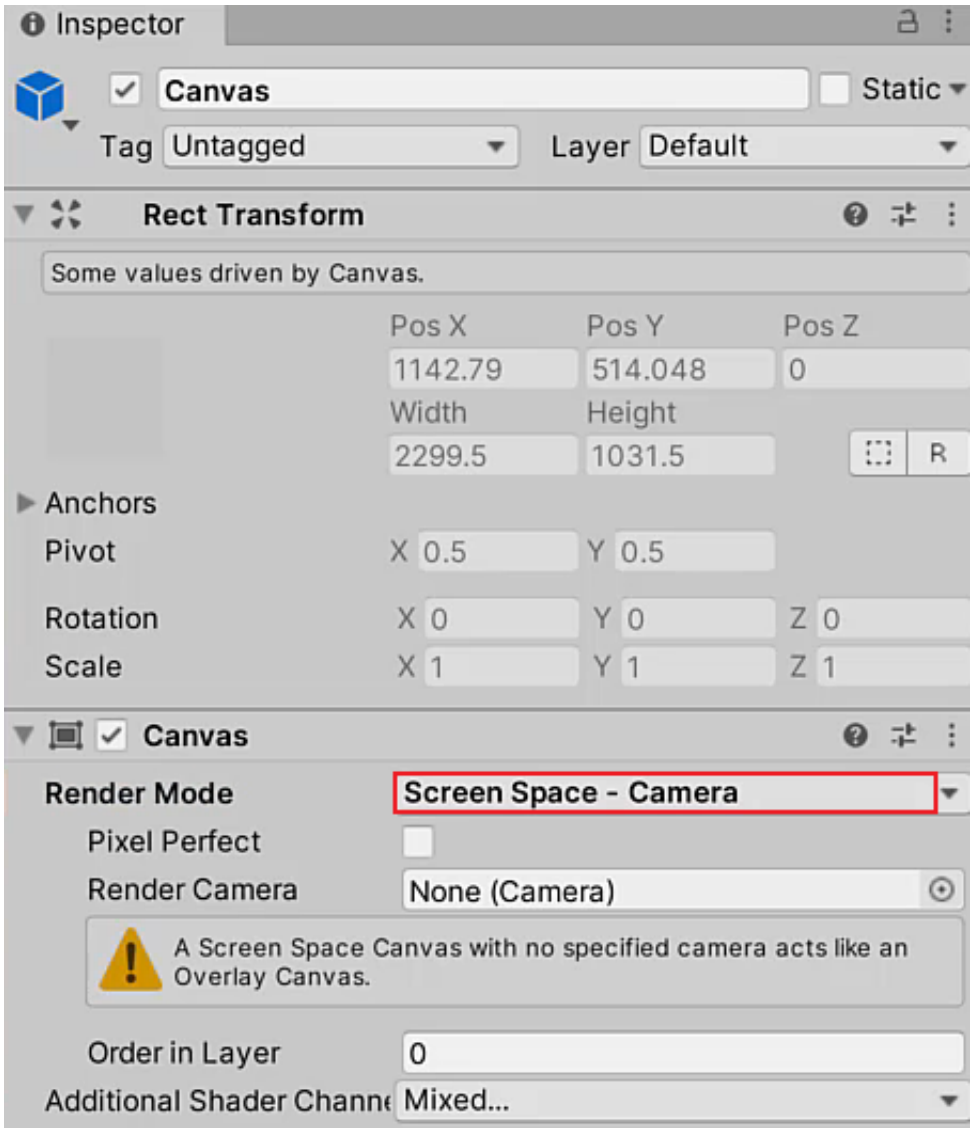
First, let's add a **Canvas** to the scene and add a **Text (TextMeshPro)** as a child.

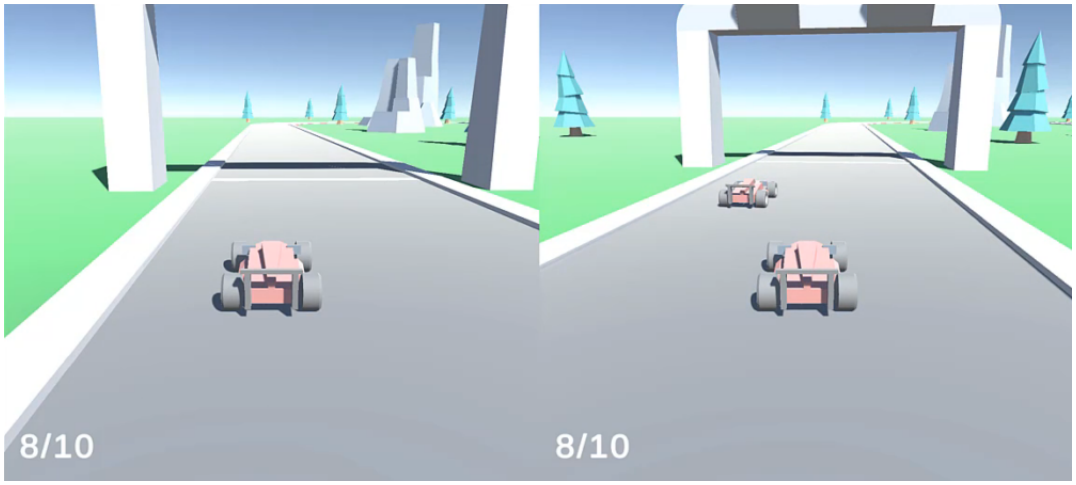


This text is going to display our current **position** in the race; e.g. 8 / 10. Let's anchor it to the bottom left corner of the screen.

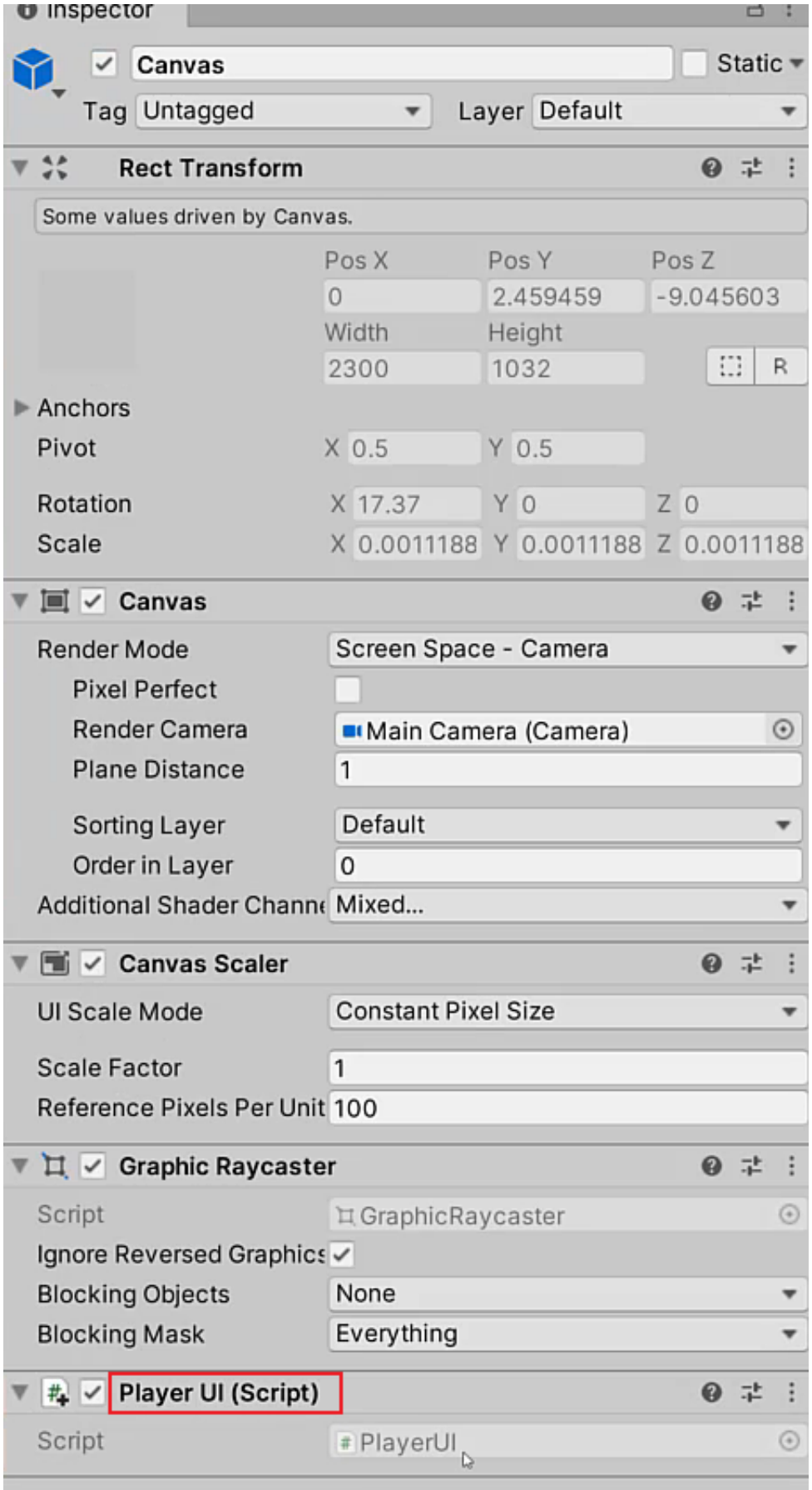


Since we have multiple screens, we need to set the **Render Mode** to **Screen Space – Camera** so that the UI is displayed on each screen.





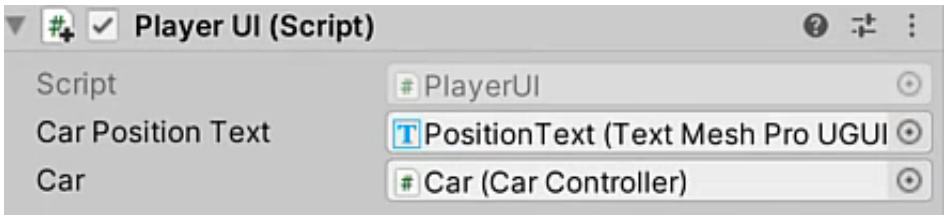
To actually update the UI text, we'll create a new C# script called "**PlayerUI**" and attach it to the Canvas.

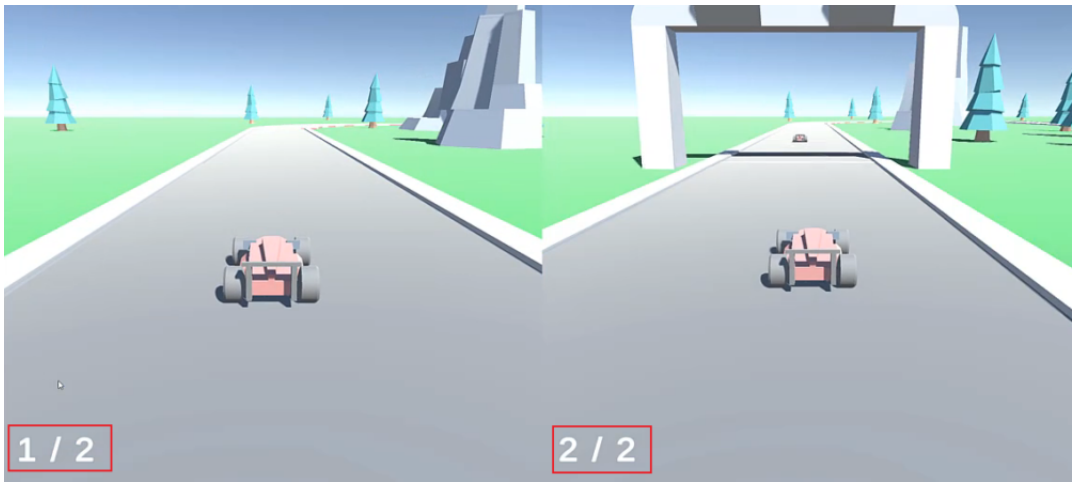


This script will be updating the TextMeshProUGUI.text to be equal to “(current race position) / (total number of cars) “.

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

public class PlayerUI : MonoBehaviour
{
    public TextMeshProUGUI carPositionText;
    public CarController car;

    void Update ()
    {
        carPositionText.text = car.racePosition.ToString() + " / " + GameManager.inst
ance.cars.Count.ToString();
    }
}
```

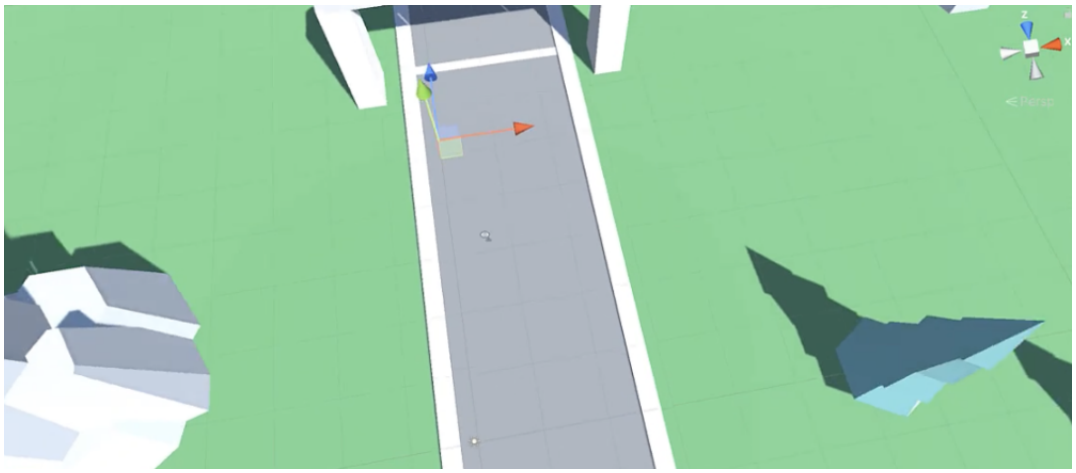
Make sure the public variables are correctly assigned to the Inspector.





In this lesson, we're going to be setting it up so that we have starting race positions.

First, we need to create an empty GameObject and place it at the starting line.



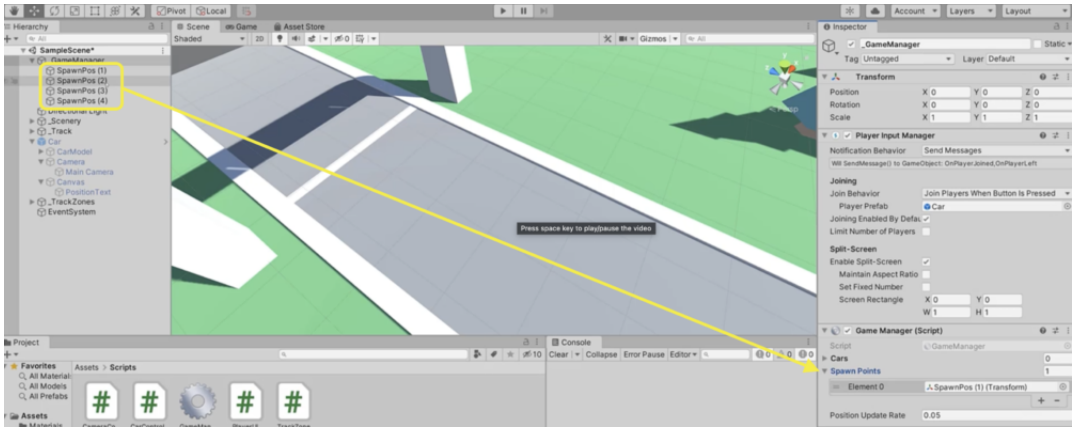
We'll then duplicate it and spread them out along the starting line.

Let's go over to the **CarController** script, and create a new public **transform array** called "spawnPoints".

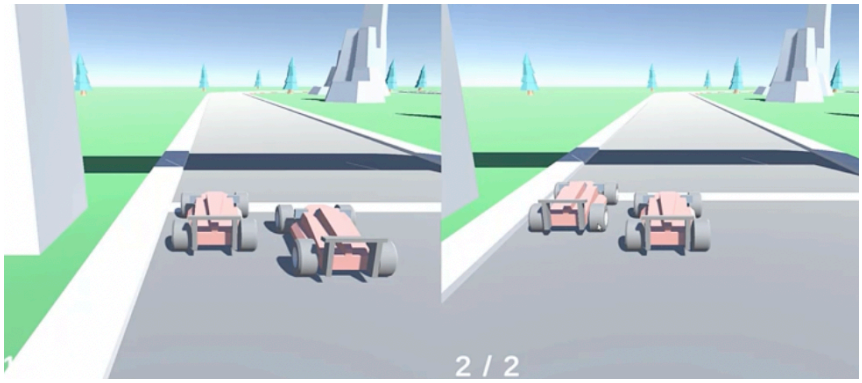
When a car is added to the cars list inside Start function, we're also going to set our position to the respective **spawnPoint**. The index of the transform array should be **cars.Count – 1**, since the first car will be at index of 0.

```
void Start ()
{
    startModelOffset = carModel.transform.localPosition;
    GameManager.instance.cars.Add(this);
    transform.position = GameManager.instance.spawnPoints[GameManager.instance.cars.C
ount - 1].position;
}
```

Once that's saved, we can star dragging the empty game objects (**spawnPos**) in the Hierarchy into the public **SpawnPoints** array in the Inspector.



Freeze At Start



Now the cars should spawn in the designated location, but you'll notice that they move around a bit.

To prevent this, we're going to add the two lines of code, which returns if **canControl** is false, at the start of both **Update** and **FixedUpdate** function:

```
void Update ()
{
    // disable the ability to turn if we cannot control the car
    if(!canControl)
        turnInput = 0.0f;

    // calculate the amount we can turn based on the dot product between our velocity
 and facing direction
    float turnRate = Vector3.Dot(rig.velocity.normalized, carModel.forward);
    turnRate = Mathf.Abs(turnRate);

    curYRot += turnInput * turnSpeed  * turnRate * Time.deltaTime;

    carModel.position = transform.position + startModelOffset;
    //carModel.eulerAngles = new Vector3(0, curYRot, 0);

    CheckGround();
}

void FixedUpdate ()
{
    // don't accelerate if we don't have control
    if(!canControl)
        return;

    if(accelerateInput == true)
    {
        rig.AddForce(carModel.forward * acceleration, ForceMode.Acceleration);
    }
}
```

Countdown To Start

Now let's set up a **Countdown** that goes "Three, two, one, go!" inside our **GameManager** script.

First of all, we need to check if all cars are ready before we start the countdown. Let's do this inside the Update function below where we update the car race positions:

```
void Update ()
{
    // update the car race positions
    if(Time.time - lastPositionUpdateTime > positionUpdateRate)
    {
        lastPositionUpdateTime = Time.time;
        UpdateCarRacePositions();
    }

    // start the countdown when all cars are ready
    if(!gameStarted && cars.Count == playersToBegin)
    {
        gameStarted = true;
        StartCountdown();
    }
}
```

We can then define **StartCountdown** so that it displays the count to each players' UI, and then invoke **BeginGame** at the end of the count. (We'll define what **StartCountdownDisplay** does later inside **PlayerUI** script.)

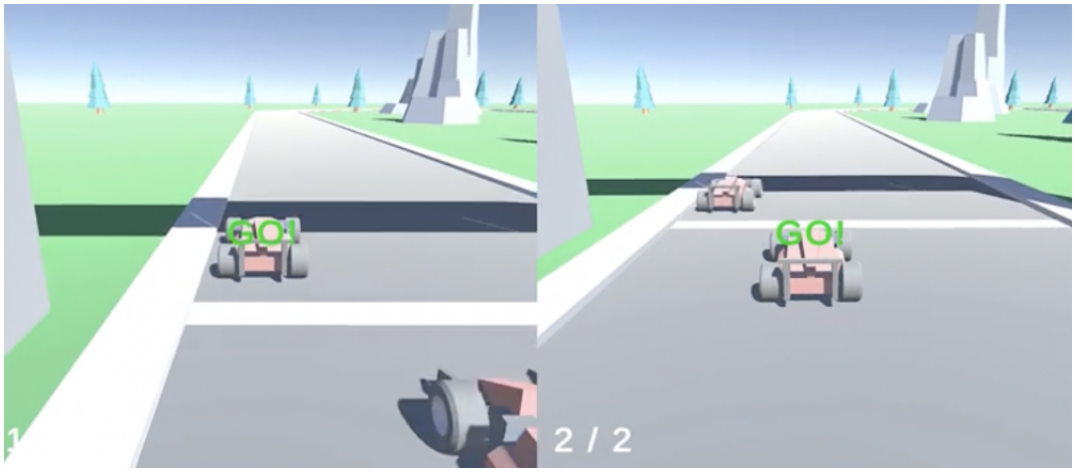
When **BeginGame** function is called, we should set each car's **canControl** to true.

```
// called when all players in in-game and ready to begin
void StartCountdown ()
{
    PlayerUI[] uis = FindObjectsOfType<PlayerUI>();

    for(int x = 0; x < uis.Length; ++x)
        uis[x].StartCountdownDisplay();

    Invoke("BeginGame", 3.0f);
}

// called after the countdown has ended and players can now race
void BeginGame ()
{
    for(int x = 0; x < cars.Count; ++x)
    {
        cars[x].canControl = true;
    }
}
```

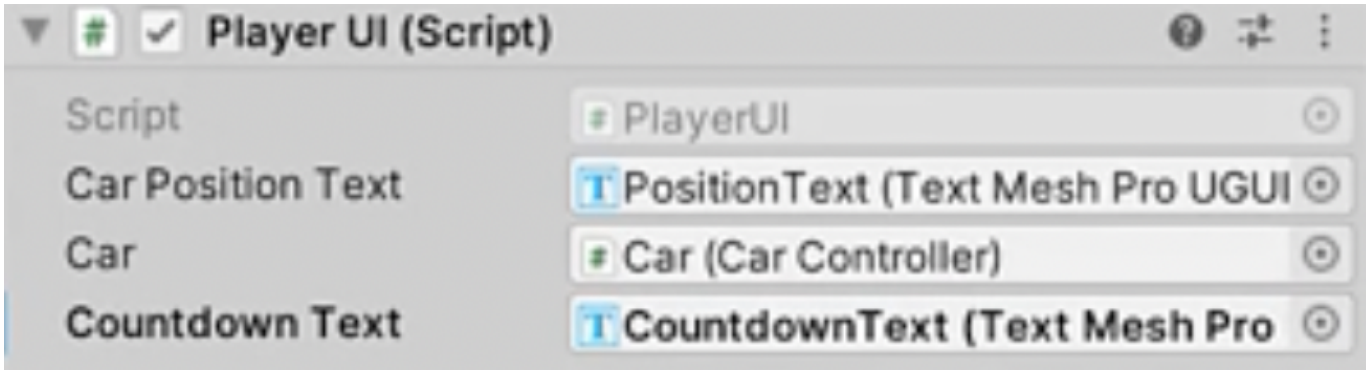
Then we can open up **PlayerUI.cs**, and define **StartCountdownDisplay** so that it enables the **countdownText.** Then it should change the text from "3", "2", "1", to "GO!", pausing 1 second in between each line with **WaitForSeconds**. Once that's all done, we must disable the countdown text.

```
public void StartCountdownDisplay ()
{
    StartCoroutine(Countdown());

    IEnumerator Countdown ()
    {
        countdownText.gameObject.SetActive(true);
        countdownText.text = "3";
        yield return new WaitForSeconds(1.0f);
        countdownText.text = "2";
        yield return new WaitForSeconds(1.0f);
        countdownText.text = "1";
        yield return new WaitForSeconds(1.0f);
        countdownText.text = "GO!";
        yield return new WaitForSeconds(1.0f);
        countdownText.gameObject.SetActive(false);
    }
}
```

Let's save the script and create a new **TextMeshPro Text** for the **Countdown.** You may want to anchor it in the centre of the screen.



Then drag the text component into the **PlayerUI**'s **CountdownText**:



Now you should be able to see that the countdown begins when there are two players in the scene.

In this final lesson of the **Arcade Kart Racing** game course, we're going to set up the **winner** trigger.

First of all, we need to keep track of our players' **lap** counts.

Let's open up **TrackZone.cs**. When a player completes a lap and enters the next trigger zone, their lap count should increment.

If the previous lap was the final lap, then it means the player has won the race. So right after where we increment the lap count, we're going to access our **GameManager's** singleton instance and call a function that checks if the current player is the **winner**:

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

public class TrackZone : MonoBehaviour
{
    public bool isGate;

    private void OnTriggerEnter (Collider other)
    {
        if(other.CompareTag("Player"))
        {
            CarController car = other.GetComponent<CarController>();
            car.curTrackZone = this;
            car.zonesPassed++;

            if(isGate)
            {
                car.curLap++;
                GameManager.instance.CheckIsWinner(car);
            }
        }
    }
}
```

We can then go over to **GameManager.cs** and define the number of **laps to win**.

Then when **CheckIsWinner** is called, we just need to check if the current car's lap is bigger than **lapsToWin**.

```
// called when a car has crossed the finish line
public void CheckIsWinner (CarController car)
{
    if(car.curLap == lapsToWin + 1)
    {
    }
}
```

If the car did cross the finish line, we're going to **disable** the ability for players to **control** their car.

Then we want to display a text that the game is over on each player's screen.

```
// called when a car has crossed the finish line
public void CheckIsWinner (CarController car)
{
    if(car.curLap == lapsToWin + 1)
    {
        for(int x = 0; x < cars.Count; ++x)
        {
            cars[x].canControl = false;
        }

        PlayerUI[] uis = FindObjectsOfType<PlayerUI>();
}
```

For this, we need to create a new function called "GameOver" inside **PlayerUI** script. The function takes in a boolean to decide whether the text will display "You win" in green, or "You lost" in red.

```
public void GameOver (bool winner)
{
    gameOverText.gameObject.SetActive(true);
    gameOverText.color = winner == true ? Color.green : Color.red;
    gameOverText.text = winner == true ? "You Win" : "You Lost";
}
```

We can then call the function inside **CheckIsWinner** to display it on each player's screen:

```
// called when a car has crossed the finish line
public void CheckIsWinner (CarController car)
{
    if(car.curLap == lapsToWin + 1)
    {
        for(int x = 0; x < cars.Count; ++x)
        {
            cars[x].canControl = false;
        }

        PlayerUI[] uis = FindObjectsOfType<PlayerUI>();

        for(int x = 0; x < uis.Length; ++x)
            uis[x].GameOver(uis[x].car == car);
    }
}
```