

COMP4002/G54GAM Lab Exercise 07 – Racing and Menus

16/03/20

This exercise involves creating a game setup commonly seen in arcades – split screen racing. You will need to download the Standard Assets from the asset store, if you have not done so already, as we'll make use of the car character controller.



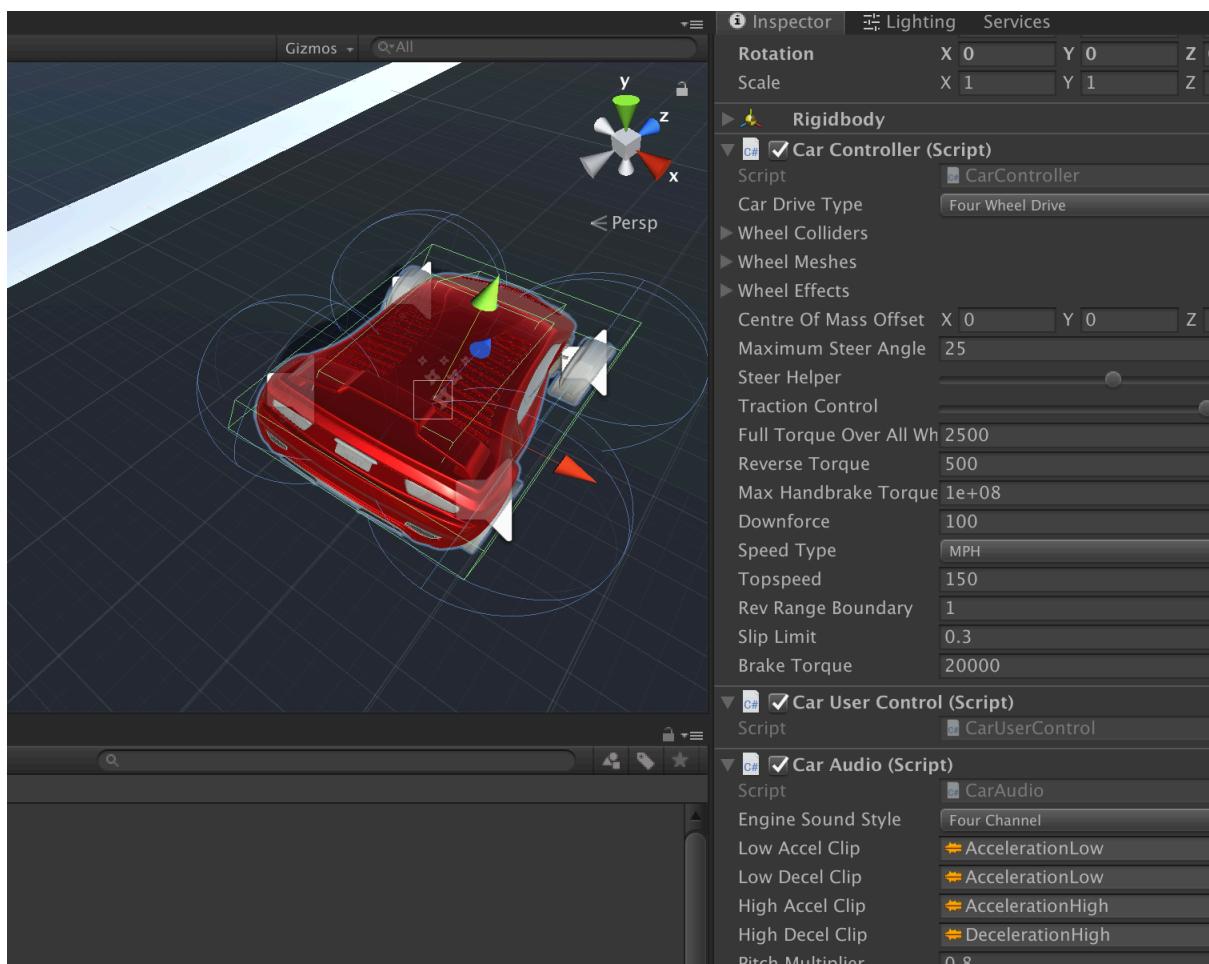
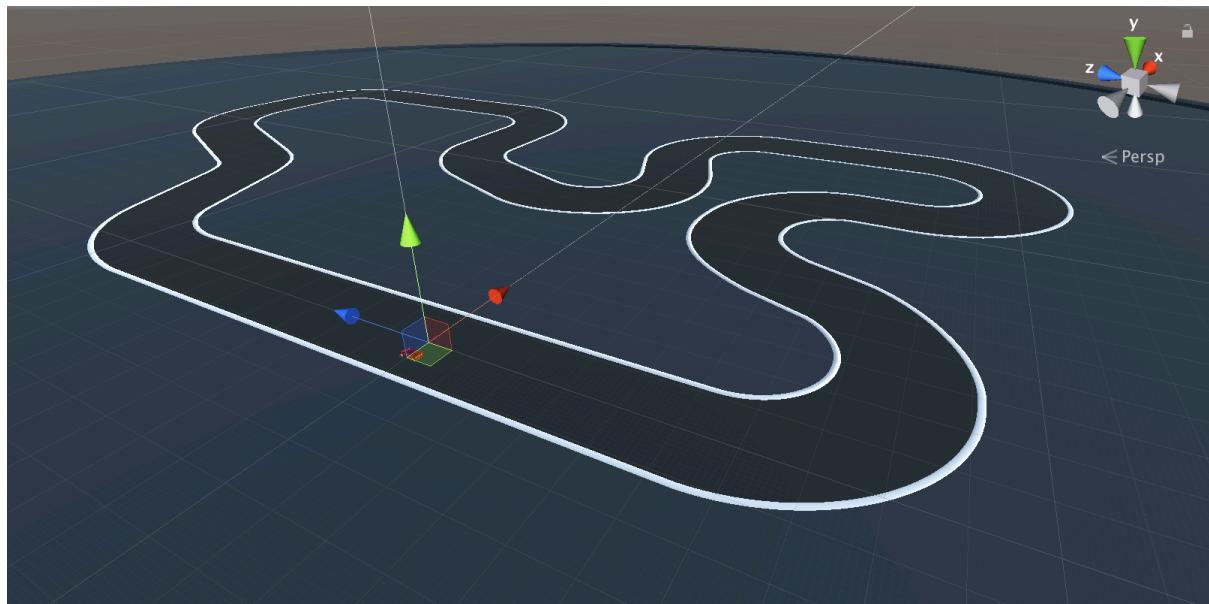
The game will have the following features

- A car that can be driven around a simple terrain or race track
- Split-screen multiplayer
- An AI opponent
- A simple menu system

Driving

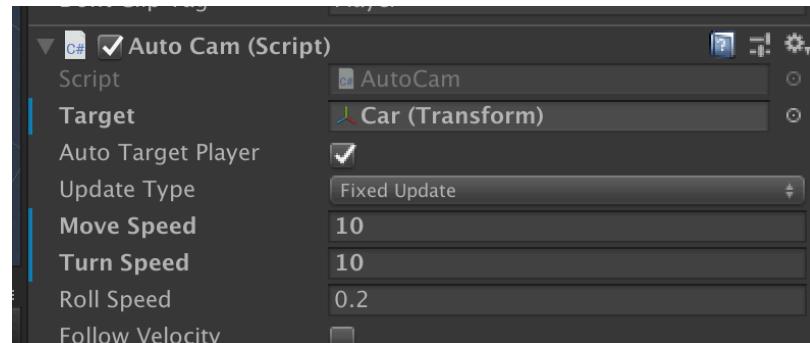
Create a new project and include the *Standard Assets* content from the asset store as in lab 04. This exercise is going to reuse several components from the assets to implement the racing mechanic that will be extended into a more significant game.

From the standard assets add the Assets->SampleScenes->GroundTrack prefab to your scene, or alternatively create whatever terrain you think might be appropriate for in initial race track (perhaps try out the *Terrain* tools having added a GameObject->3D Object->Terrain object) The GroundTrack prefab consists of a large ground mesh, an example race track layout and kerbs, however none of these objects include a mesh collider and so are not solid. Add a *Mesh Collider* component to the three objects to resolve this.



Building a realistic car character controller is a non-trivial task, so for this exercise we will make use of one provided in the standard asset pack. Add the Assets->Standard Assets->Vehicles->Car->Prefabs->Car prefab to the scene. This also provides a reasonable amount of configurability to the car.

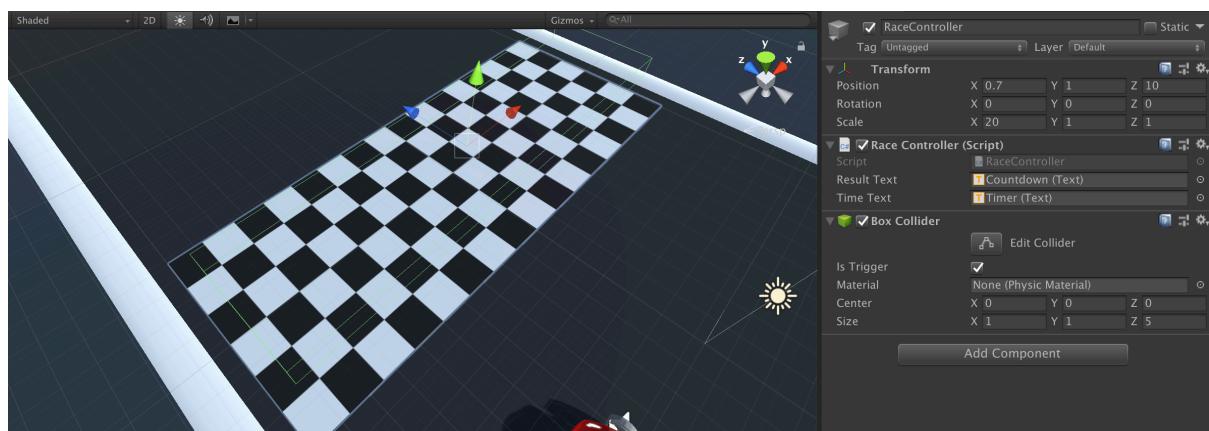
As with the third-person and first-person character controllers from previous exercises we can also attach the *MainCamera* object as a child of the Car object. Here you have several options – is this a first-person perspective, or is the camera set back and above the car? An alternative is to use a more sophisticated camera rig that is less tightly coupled to the car but attempts to follow it.



Add the Assets->Standard Assets->Cameras->Prefabs->MultipurposeCameraRig prefab to the scene, and set the target of its AutoCam script component to be your Car object.

Racing

The game should have a simple start countdown before the race starts, then the player will be timed driving around the track, stopping the clock when they cross the finish line. Add an object to serve as a race controller, with a *Box Collider* trigger to register when the car crosses the line. In the screenshot below a textured quad provides the visual representation of the line.



The code fragment below gives an example of how a countdown might be created. Extend and add this to the race controller. Note that *resultText* and *timeText* are GUI components that should be added to a canvas as in the HUD in lab 03.

```
RaceState raceState;

enum RaceState
{
    START,
    RACING,
    FINISHED
};
```

```

void Start()
{
    StartCoroutine(startCountdown());
    raceState = RaceState.START;
}

IEnumerator startCountdown()
{
    int count = 3;

    while (count > 0)
    {
        resultText.text = "" + count;
        count--;
        yield return new WaitForSeconds(1);
    }
    raceState = RaceState.RACING;
    startTime = Time.time;
    resultText.text = "GO";

    yield return new WaitForSeconds(1);
    resultText.enabled = false;
}

void Update()
{
    if(raceState == RaceState.RACING)
    {
        timeText.text = "" + (Time.time - startTime);
    }
}

```

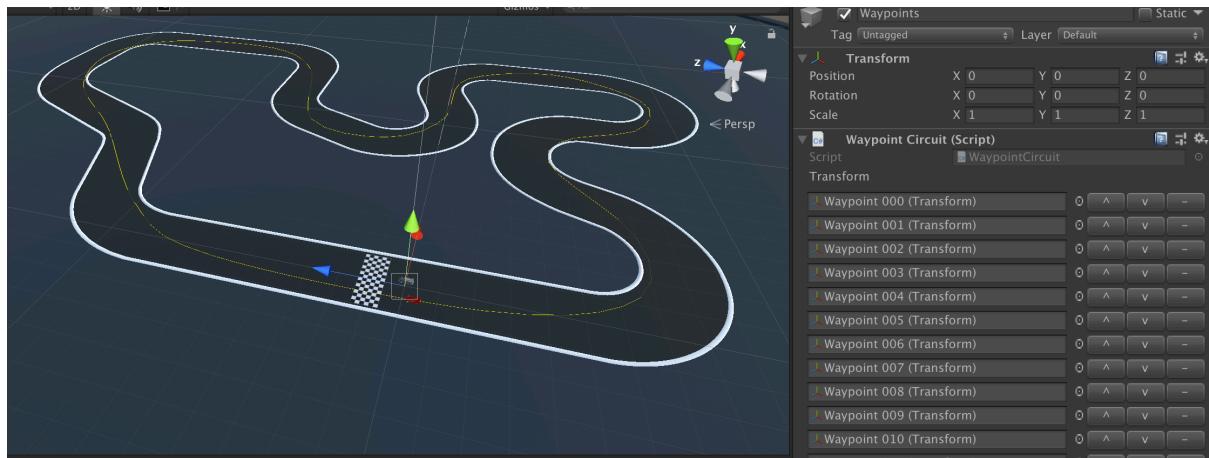
AI

A racing game with only a single car does not present much of a challenge for the player, so the next step is to add other cars as opponents. Here we will add one single AI player however it should be apparent that by modifying the *CarUserControl* script attached to a second controllable car object, a second human player using a different set of keys could be introduced. Either way, the game will have a classic “split-screen” multiplayer view as seen in games such as Mario Kart.

To create the AI car add the Assets->Standard Assets->Vehicles->Car->Prefabs->CarWaypointBased prefab to the scene. This is a variant of the player controlled car but one that attempts to drive between a series of waypoints. This is similar to the construction used in lab 05 for controlling the AI, but unlike the NavMesh agent, a script will calculate a *spline* (a smooth curve) between the waypoints and then the car will attempt to follow it as best it can. However, the realistic physics of the car will mean that the curve will not be followed exactly.

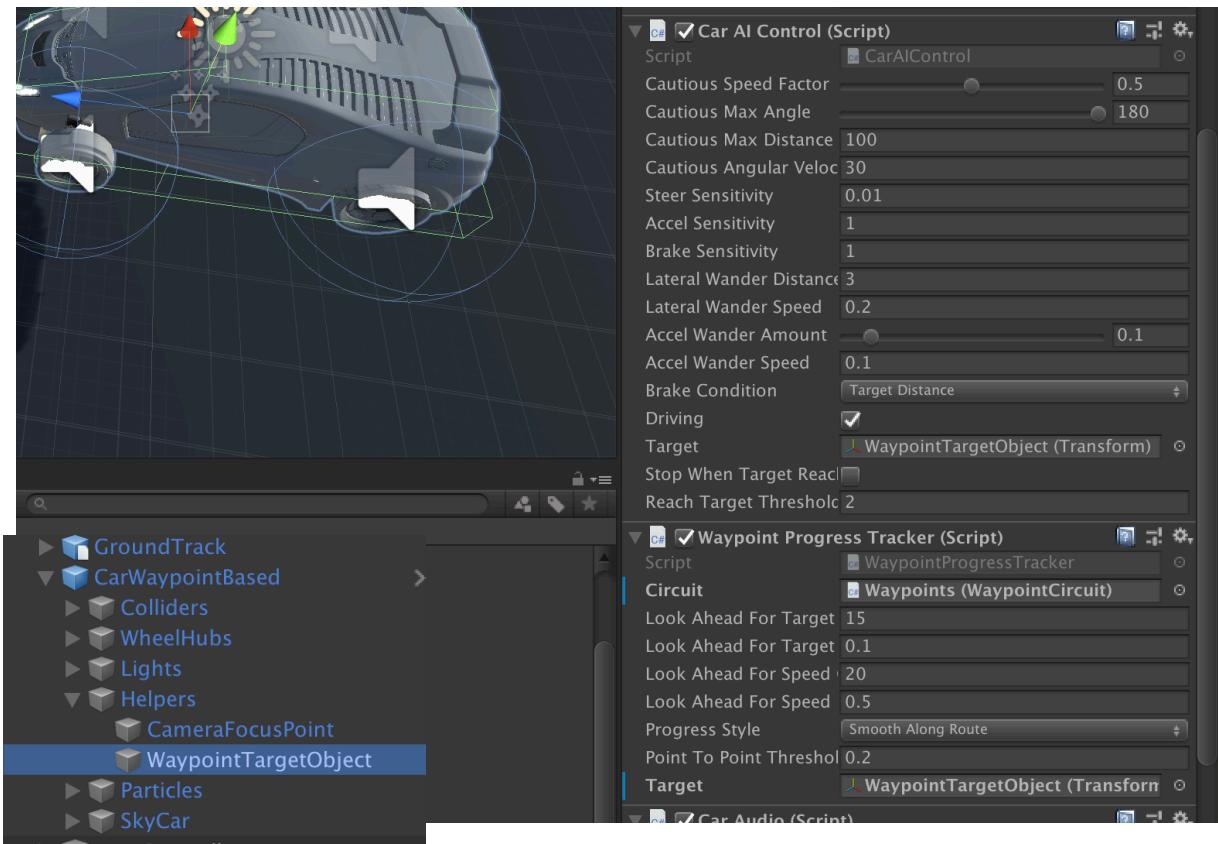
Here you may find it useful to look at the example *CarAIWaypointBased* scene in the SampleScenes folder of the standard assets to see a functioning setup in case of any issues.

Create an empty game object to serve as a parent for the waypoints. Create a series of empty game objects as children of this object around the track. Adding the script *WaypointCircuit* to the parent object and clicking *Assign using all child objects* will construct a smooth route between the waypoints for the AI car to attempt to drive.



Finally, the AI car must be configured to follow this new circuit. In the `WaypointProgressTracker` component of the car, set the *Circuit* property to be your waypoint parent object. Set the *Target* property of both the `CarAIControl` script and the `WaypointProgressTracker` script to reference the `WaypointTargetObject` that is a child of the AI car.

The upshot of all of this setup is that the `WaypointProgressTracker` will identify a point on the circuit some way ahead of the car, and continually update this point, and the car will attempt to drive towards it.

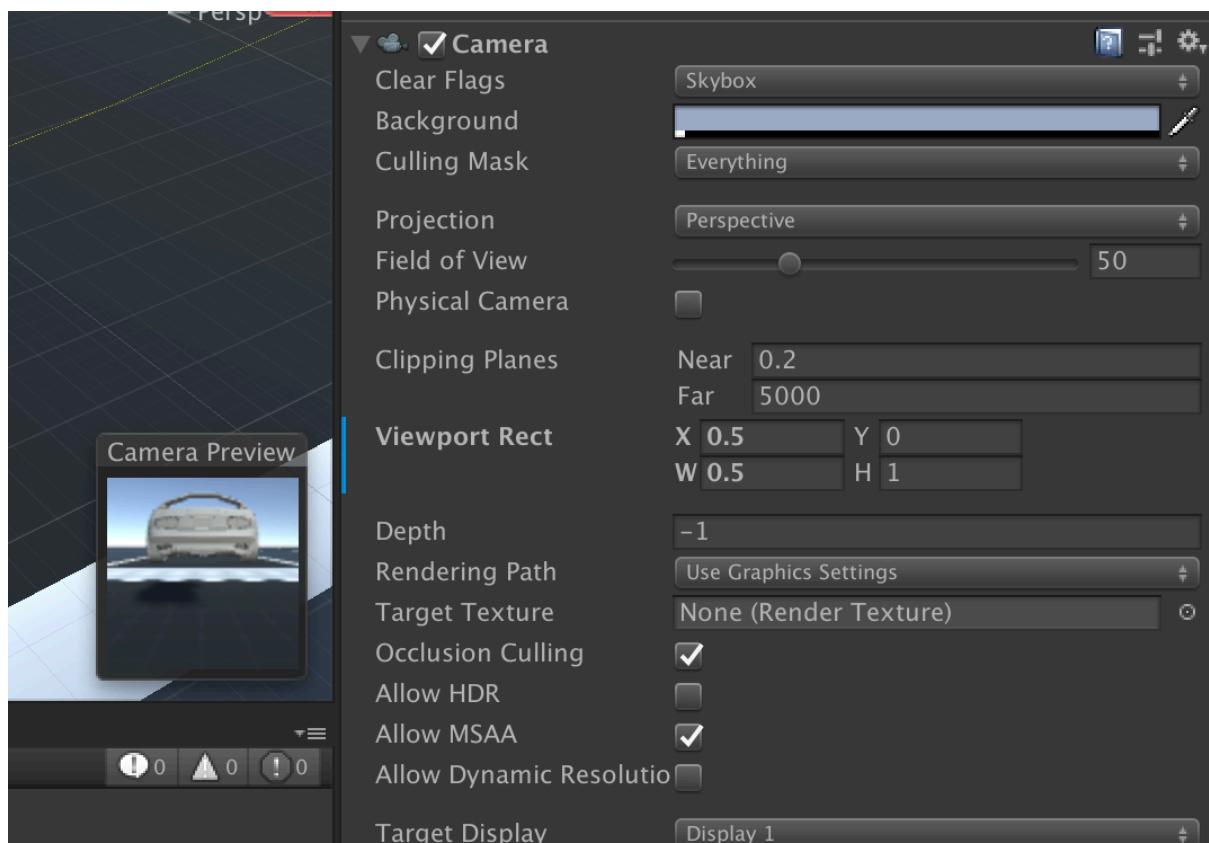


The AI car will set off when the scene is loaded. To control its behaviour, disable the `CarAIControl` script component, but then programmatically enable it via your `RaceController` script, by tagging the car appropriately.

```
using UnityStandardAssets.Vehicles.Car;  
...  
    GameObject[] AICars = GameObject.FindGameObjectsWithTag("AICar");  
  
    foreach(GameObject car in AICars)  
    {  
        car.GetComponent<CarAIControl>().enabled = true;  
    }
```

Split Screen Multiplayer

Giving each “player” their own screen is quite straightforward. Each car will have its own camera, and be allocated part of the screen to display that camera.

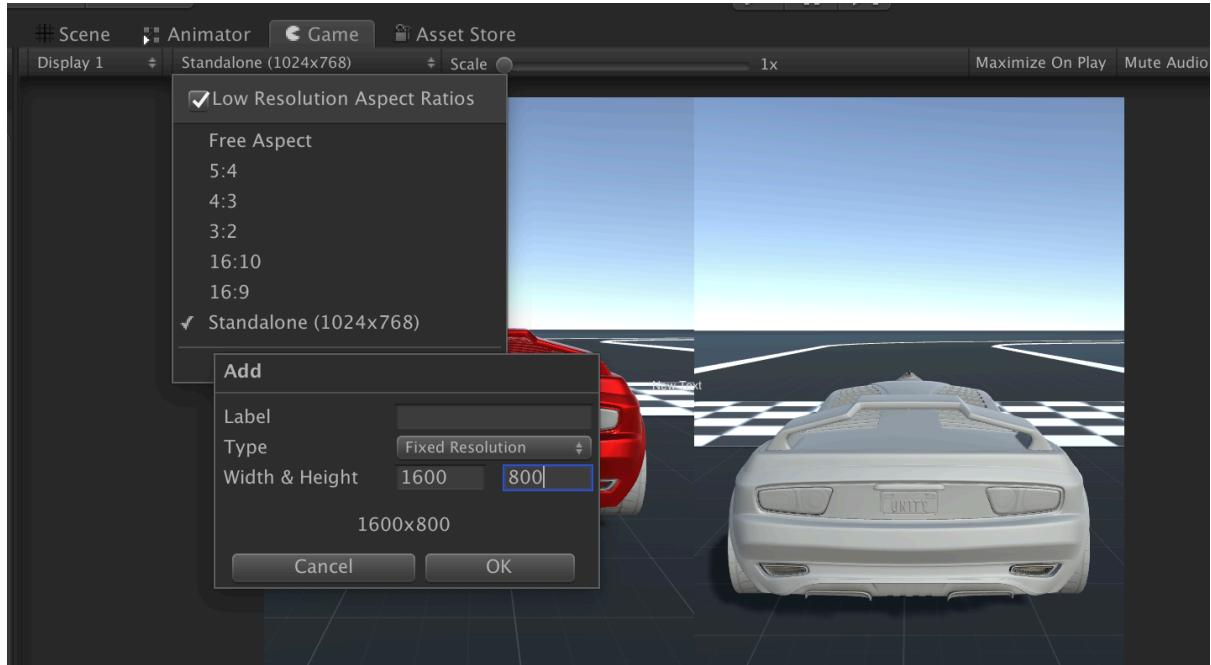


Add a second `MultipurposeCameraRig` to the scene, but have this one target the AI car. In both rigs disable the *Auto Track Player* flag. The grandchild of the camera rig is the `Camera` object itself, and here we will configure the split-screen effect.

Pick one of the car and rig combinations to take the left side of the screen. In the `Camera` set its *Viewport Rect* – this is the space on the screen the view from this camera will occupy – and set its width to 0.5 (i.e. half of the screen) and its X coordinate to 0 (i.e. the left of the screen).

Repeat this process for the other car and rig combination, but this time also set the X coordinate of the *Viewport Rect* to be 0.5 (i.e. specifying that this view will take up the right of the screen).

Finally, in the game view, set the resolution of the screen to be of an aspect ratio that allows the two views to sit side by side, for example a 2:1 ratio such as 1600 by 800.



Menus and Multiple Scenes

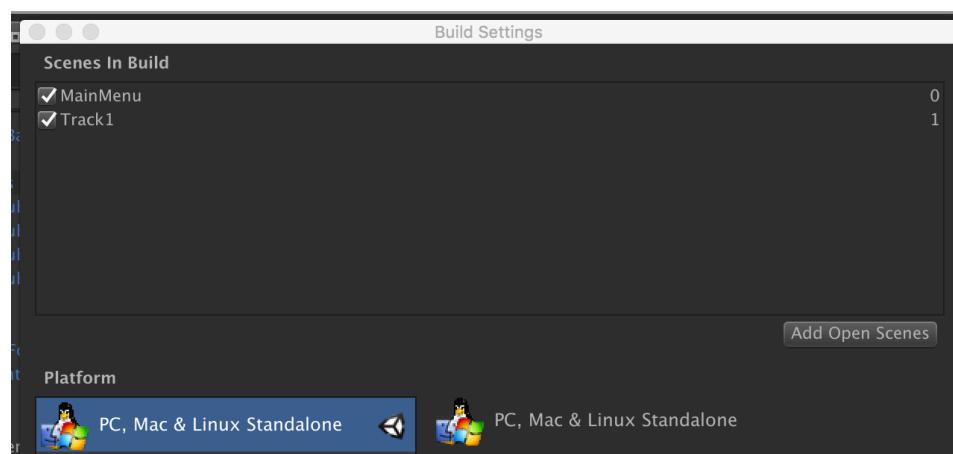
So far, the games that have been made have been demonstrations of various principles and core mechanics that take place in a single *scene*, rather than things that would arguably be complete, albeit very simple, games. Next, we will consider how the game can be further structured to address this.

Dividing the game into multiple levels, with each in an individual scene gives distinct advantages. The game can be segmented into, for example, easy, medium and hard difficulties. More commonly, however, multiple levels allow for the game to be structured into manageable chunks, for example requiring each to be completed before the next is unlocked or to give an indication of progress. A full game is likely to be far too large to be loaded at once, and so only loading the subset of the game that will be immediately interacted with is a significant efficiency step.

In Unity, the *scene* – the structure that is used to encapsulate a level – can be used to also scaffold large elements of interaction with the game, for example to host interfaces to allow the player to start the game, or select a level, to show credits, or to return to on failure rather than just quitting the game or automatically restarting the same level. Here we need to consider how to move from one scene to the next – i.e. from the menu to the level, or from the level to the credits – and also how data can persist between scenes, given that to date all objects have been added to the hierarchy for a particular scene, and are reset when that scene is restarted.

Create a new *Scene* asset. This will be the first screen that the user sees, and they will return to it between races.

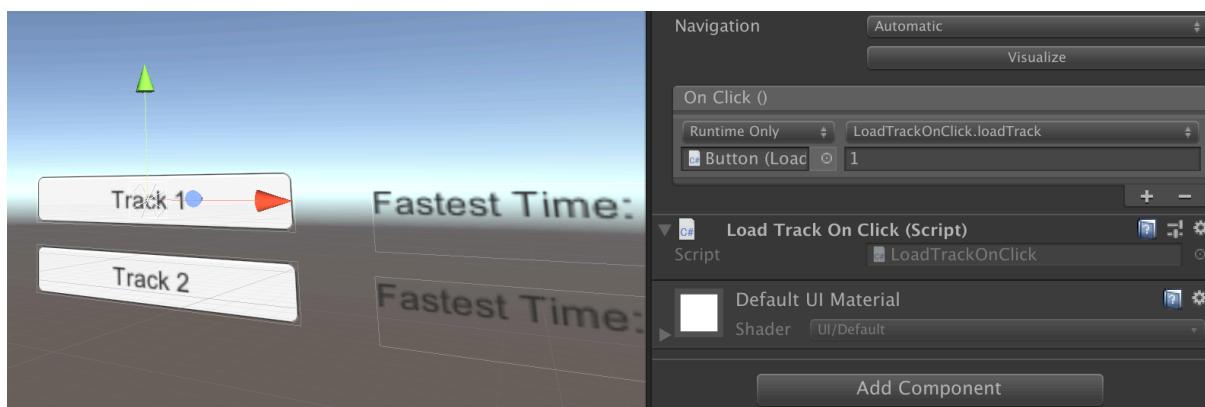
The first task is to decide which scenes should be in the game, and to be able to identify each one appropriately. Open the File->Build Settings menu, which displays the scenes currently in the build (note this menu is also where you can configure and build an executable package for the game that can be distributed). By either dragging scenes into this list or removing superfluous scenes, ensure that your new “menu” scene is at index 0, and your race track scene is at index 1.



Add a *Button* to the scene, that when clicked calls an attached script (as in lab 06) to load the scene at the specified index. Note that when the *loadTrack* method is selected in the drop down the ability to specify the argument for this method appears. This means that this script can be added to multiple buttons, to provide an appropriate button for scenes at different indices.

```
using UnityEngine.SceneManagement;

public class LoadTrackOnClick : MonoBehaviour
{
    public void loadTrack(int trackIndex)
    {
        SceneManager.LoadScene(trackIndex);
    }
}
```



- Modify your RaceController class so that, when the player crosses the finish line, the Menu scene is reloaded – i.e. scene index 0 is loaded as above.

A detailed tutorial on creating a similar menu is available here:

<https://unity3d.com/learn/tutorials/topics/user-interface-ui/creating-main-menu>

Score Manager

The final element of the game is to maintain the “best time” for the race track – displayed in the menu scene, but updated by the race track scene. At first glance it might not be clear how to achieve this, as we would by default add a script component to a game object with which to maintain the score (time). However, game objects are part of the object hierarchy of an individual scene, and would be lost when we move from one scene to the next.

There are two ways to achieve persistent data that exists across all scenes. The conventional, and until recently only approach is to implement a *singleton* pattern that ensures that there will be only one instance of the Score Manager object, and where this instance can be accessed via a *static* method of the class rather than as an object in the hierarchy. A *DontDestroyOnLoad* call ensures that the object itself persists between Scenes. The code for this is given in the appendix.

The better practice is to make use of a *ScriptableObject*. This is a data container that acts as an asset, and a reference to which can be given to other game objects in the scene – this means that the data container can be used as our Score Manager to persist scores between scenes.

<https://docs.unity3d.com/Manual/class-ScriptableObject.html>

```
using UnityEngine;

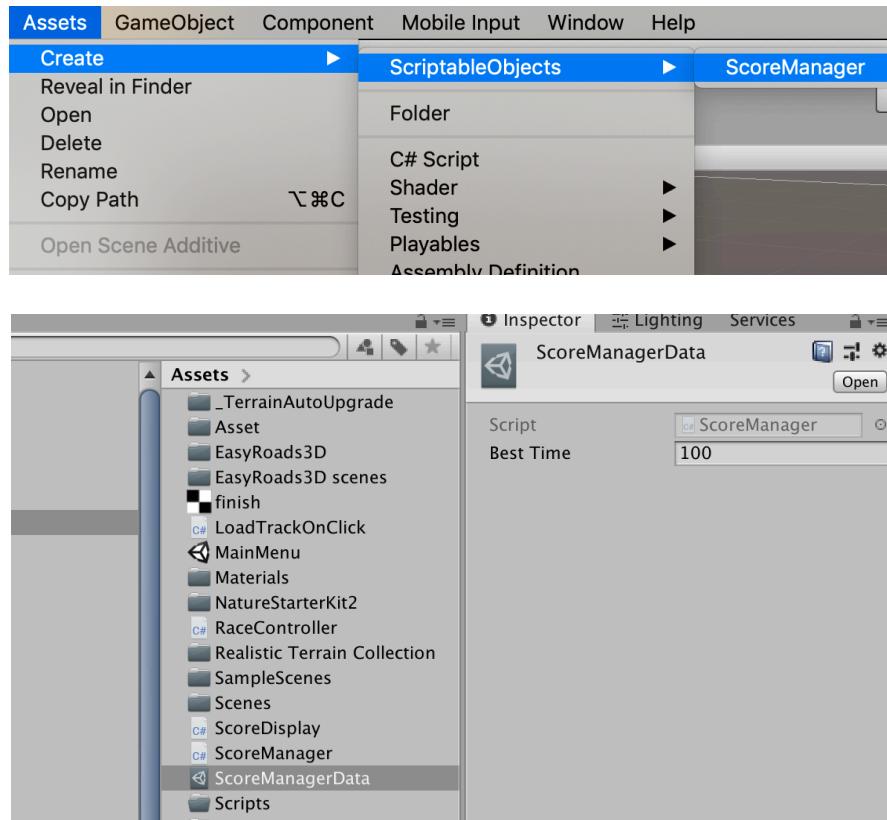
[CreateAssetMenu(fileName = "ScoreManagerData", menuName = "ScriptableObjects/Score
Manager", order = 1)]
public class ScoreManager : ScriptableObject
{
    public float bestTime = 100.0f;

    public void setTime(float newTime, int trackNumber)
    {
        if (bestTime > newTime)
            bestTime = newTime;
    }
}
```

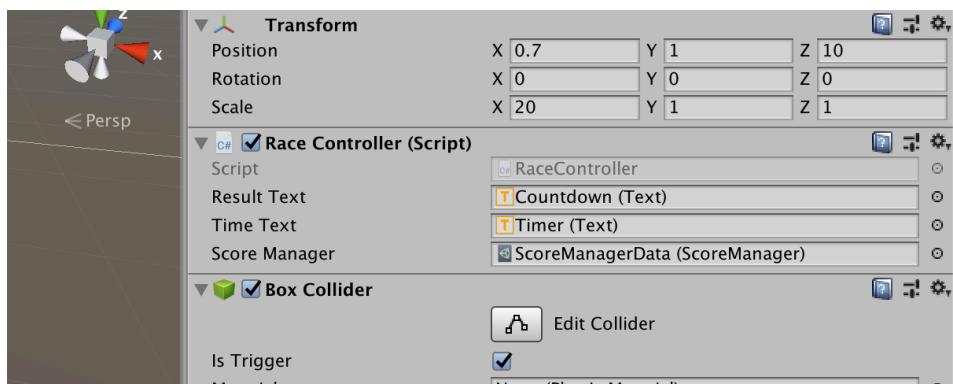
Create a new *Script* asset in your project called ScoreManager. This script does two things; it defines the data and functions for the ScoreManager (here simply maintaining the best time), and via the *CreateAssetMenu* directive it adds a new menu entry to the Unity Editor (scripts can alter the editor as well as the game play).

The menu entry, shown below, allows us to create a new instance of the ScoreManager object, but as it extends *ScriptableObject* rather than the familiar *MonoBehaviour* rather

than being added to the hierarchy the ScoreManager is created as a file asset called ScoreManagerData.



Finally, the various scripts and game objects in the hierarchy that require access to the Score Manager, regardless of the scene that they are in, can expose a reference for the Score Manager that can be set by dragging the ScoreManagerData from the assets into the inspector. From there the object can be used as normal.



When the race ends, the RaceController can save the lap time to this object:

```
public ScoreManager scoreManager;

...
scoreManager.setTime(Time.time - startTime),
SceneManager.GetActiveScene().buildIndex);
```

The best time will not persist between launches of the game as the ScoreManager object only survives between *scenes*. The ScoreManager could make use of the *PlayerPrefs* storage functionality, which provides key-value pair persistence (disk based).

<https://docs.unity3d.com/ScriptReference/PlayerPrefs.html>

<https://unity3d.com/learn/tutorials/topics/scripting/high-score-playerprefs>

Exercises

Create a second track – copy your initial race track scene but add more obstacles to it to make it more complicated. There are ramps and a loop in the Sample Scene assets.

Add a second button to the menu that loads the new scene, but restrict access to this (i.e. disable the button) until the player has achieved a certain target time in the first track, essentially “unlocking” the new track.

Appendix

Singleton approach to score management across scenes:

<https://unity3d.com/learn/tutorials/projects/2d-roguelike-tutorial/writing-game-manager>
https://en.wikipedia.org/wiki/Singleton_pattern

```
public class ScoreManager : MonoBehaviour
{
    static ScoreManager instance = null;

    void Awake()
    {
        //Check if instance already exists
        if (instance == null)
            //if not, set instance to this
            instance = this;

        //If instance already exists and it's not this:
        else if (instance != this)
            // Then destroy this. This enforces the singleton pattern,
            // meaning there can only ever be one instance of a ScoreManager.
            Destroy(gameObject);

        //Sets this to not be destroyed when reloading scene
        DontDestroyOnLoad(gameObject);
    }

    public float bestTime = 100.0f;

    public void setTime(float newTime, int trackNumber)
    {
        if(bestTime > newTime)
            bestTime = newTime;
    }
}
```