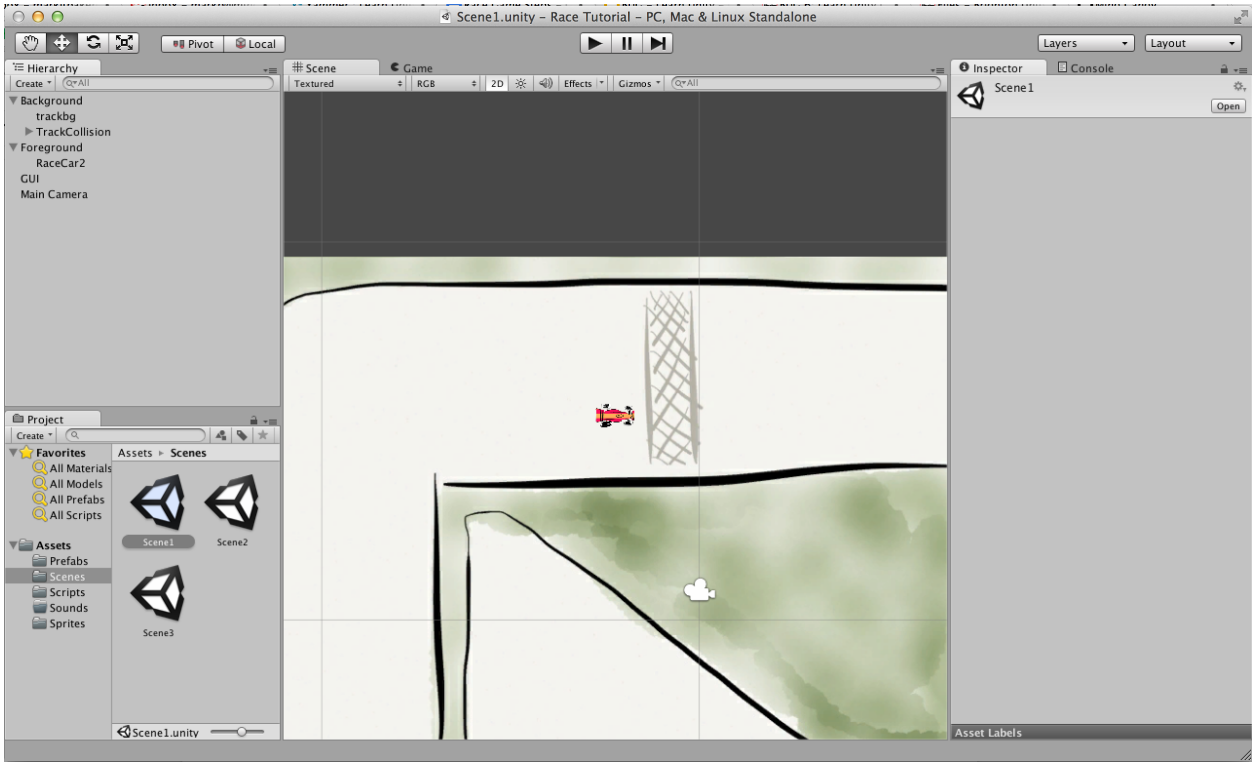
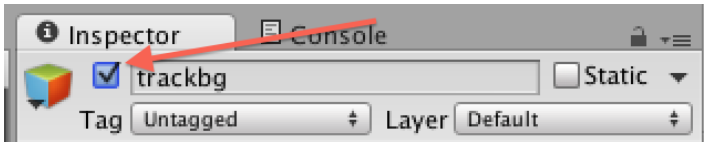
This tutorial will show you how to make a simple top-down 2d racing game, including AI drivers.
You will need Unity 4.3 or later which can be downloaded free from http://unity3d.com and the tutorial files can be downloaded from https://github.com/mindcandy/lu-racer

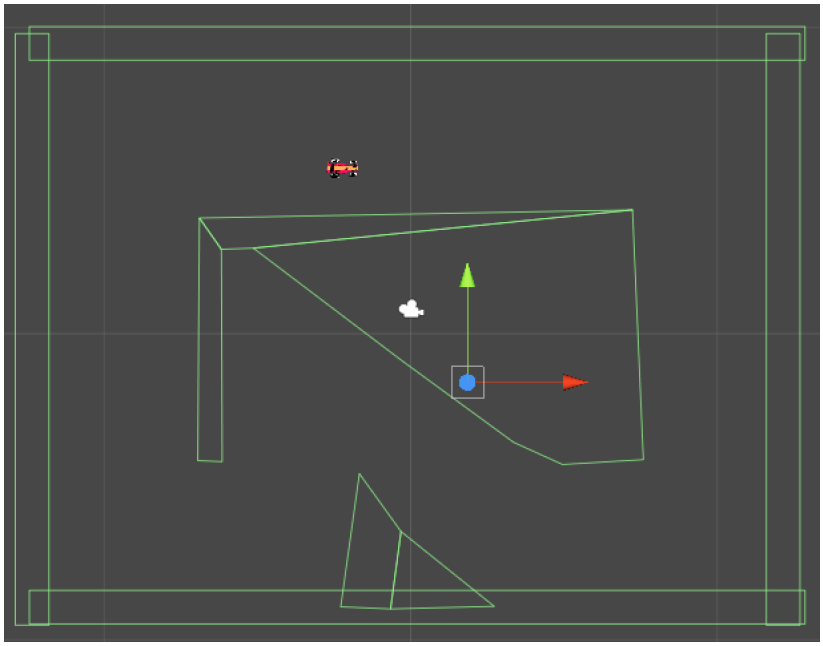
Open Unity and open the downloaded project (in File -> Open Project)
Find Scene 1 in the Assets/Scenes folder and load it by double clicking. You should see

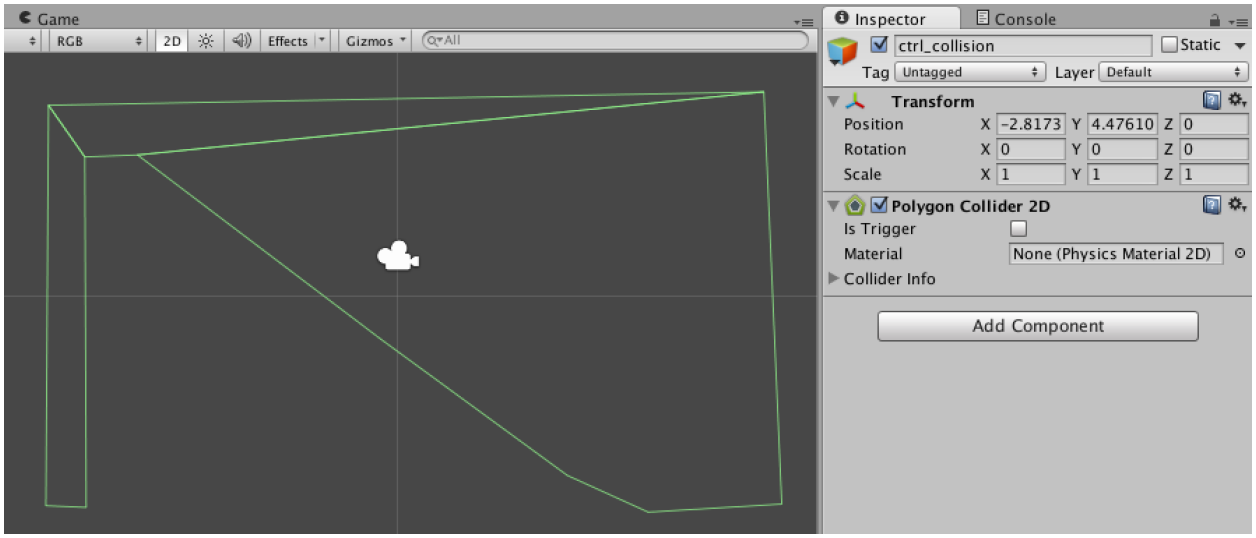


The ‘trackbg’ object is a simple sprite - we can turn it off temporarily by unchecking the box between the cube and its name in the property editor:



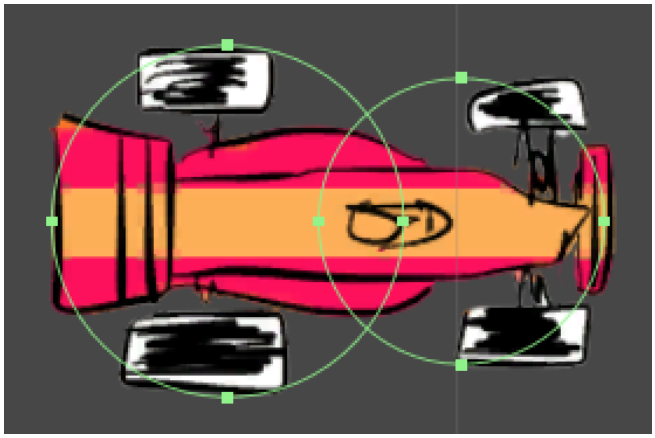
If you select TrackCollision you can see it is made up - of various BoxCollider2Ds with the central part being a PolygonCollider2D. To manipulate the Polygon Collider you hold down shift and then can drag around individual points, or click in the middle of a line to split it in half and create a new point.





So now we've got a visual representation of the track and some collision to stop cars and other game objectives going in the wrong place.

Look at RaceCar2. Like many of our previous game objects it has a Sprite Renderer and RigidBody2D. Instead of a BoxCollider2D, we use a pair of CircleCollider2Ds - this gives us nicer collision if we hit a wall, a Box Collider would want to make the car rotate around when hitting the wall at an angle.



RaceCar2 also has a pair of scripts - Player Inputs and Car Movement. We'll look at those in a second.

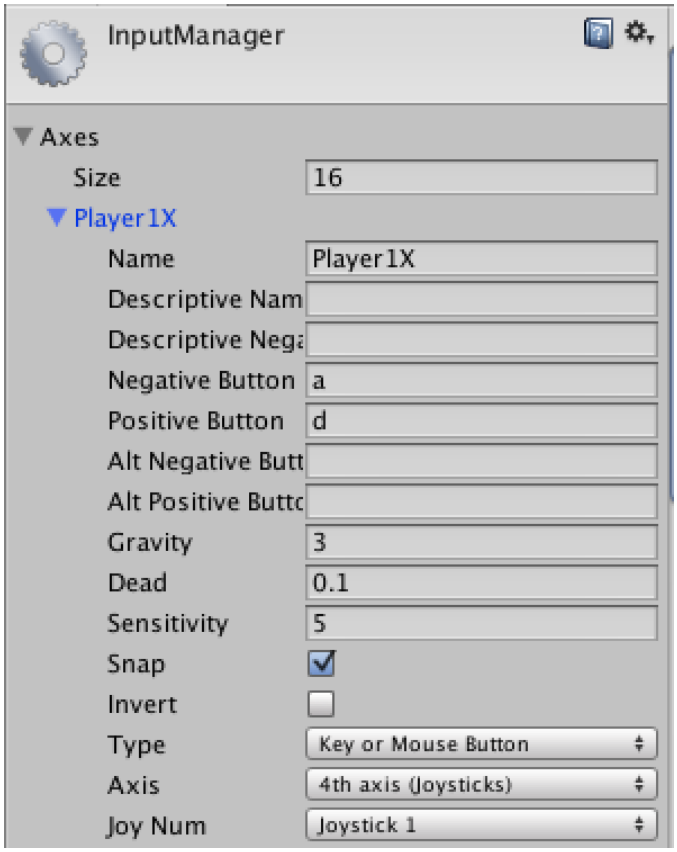
Show track again by checking box in trackbg

Now Play the game and use the cursor keys to try and drive the car. You'll notice it acts more like the spaceship in an asteroids game - it accelerates forward when you press the Up cursor key, and rotates when you press the Left/Right keys, but it doesn't "go around corners" as expected.

Let's look at the Player Inputs script for a moment. (If you click on the 'cog' next to Player Inputs and choose Edit Script it will open in MonoDevelop). Because we want this to be a 4 player game, we abstract out the player inputs to a separate component, so we can re-use that easily and simply change the player number, or even change this dynamically during the game. We use C# property syntax to define X and Y properties:

```
// get X axis from -1 to +1
public float x {
    get {
        return Input.GetAxis(string.Format("Player{0}X", playerNumber));
    }
}
// get Y axis from -1 to +1
public float y {
    get {
        return Input.GetAxis(string.Format("Player{0}Y", playerNumber));
    }
}
```

These read from the Input Axis defined by unity - you can see that player 1 will read from the Player1X axis, player 2 from Player2X axis, and so on. These are set up in the unity Input Manager which you can find in the menu under Edit -> Project Settings -> Input.



Look at Car Movement script now. We have some tweakable properties for acceleration, braking and steering speeds. We get a reference to the PlayerInputs component so we can read X/Y values from this. The script applies the inputs to the physics of the car in the FixedUpdate() function:

```
void FixedUpdate() {
    // steering
    rigidbody2D.AddTorque(_inputs.x * steering * -0.2f);
    // apply car movement
    rigidbody2D.AddForce(transform.right * _inputs.y * acceleration * 50.0f);
}
```

We're trying to 'twist' the car by adding torque (a twisting force) and driving the car forward by adding a force in the direction we want to go. That's not quite what we want, hence the slightly rubbish controls.

comment out the 'AddTorque' line and uncomment the 2 line below it so FixedUpdate is now:

```
void FixedUpdate() {
    // steering
    float rot = transform.localEulerAngles.z - _inputs.x * steering;
    transform.localEulerAngles = new Vector3(0.0f, 0.0f, rot);

    // apply car movement
    rigidbody2D.AddForce(transform.right * _inputs.y * acceleration * 50.0f);
```

```
}
```

If you Play the game now you'll notice the steering is a little better, though if you bounce off a wall it won't stop spinning. Remove the comment from the lin rigidbody2D.angularVelocity = 0.0f; to stop that spinning!

Now, if you comment out AddForce and uncomment the acceleration/braking lines so FixedUpdate() looks like:

```
void FixedUpdate() {
    // steering
    float rot = transform.localEulerAngles.z - _inputs.x * steering;
    transform.localEulerAngles = new Vector3(0.0f, 0.0f, rot);

    // acceleration/braking
    float velocity = rigidbody2D.velocity.magnitude;
    float y = _inputs.y;
    if (y > 0.01f) {
        velocity += y * acceleration;
    } else if (y < -0.01f) {
        velocity += y * braking;
    }

    // apply car movement
    rigidbody2D.velocity = transform.right * velocity;
    rigidbody2D.angularVelocity = 0.0f;
}
```

Now play the game it the car actually drives sensibly. If you steer, then the car's velocity changes with its direction - just like a 'real' car! What did we do? Simply, we take the current velocity of the car and turn it into a single number - usually it is a 2d number (x,y) that is the velocity in the x and y axis. We turn this into a single number giving the total velocity without a specific direction. We then increase or decrease that speed based on the acceleration / braking inputs. Finally we set turn the car velocity back into a 2d velocity by multiplying it by the transform.right vector - this is the current direction that the car is facing in.

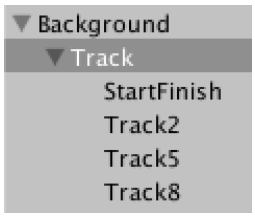
So you can see if the velocity speed stays the same, it will still change direction as the car steers.

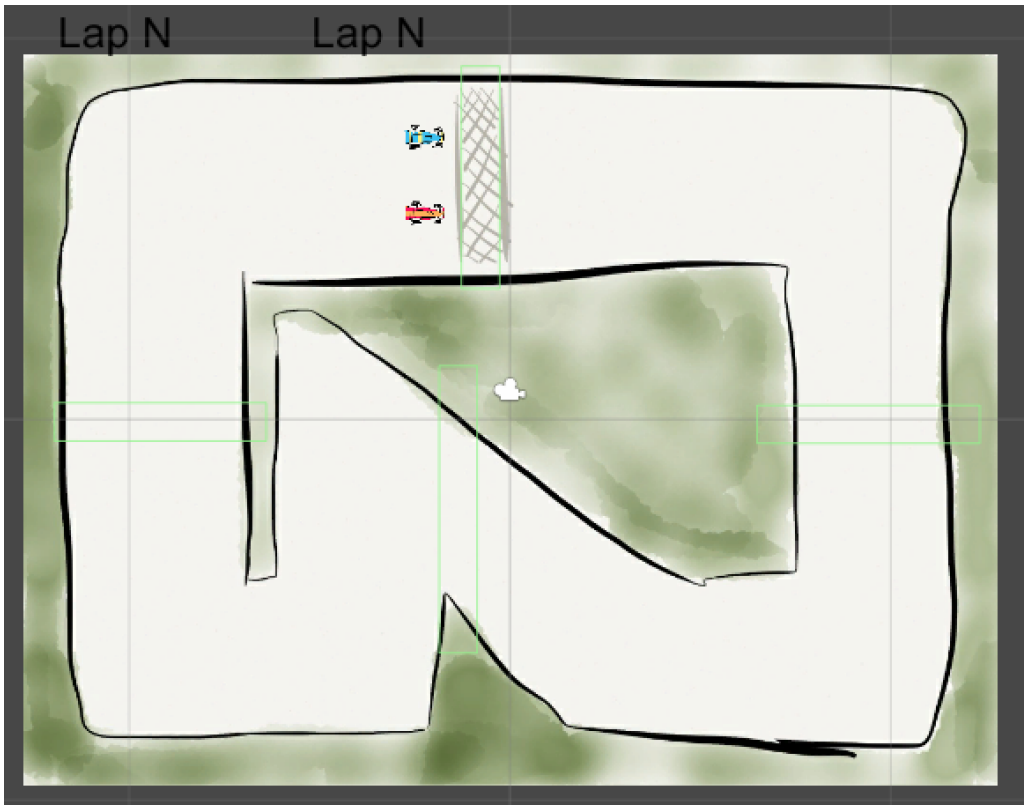
Now let's move on to **Scene2** - find it in Assets/Scenes and double click to load it.

If you click Play you can see we now have two cars. The second car uses the keys WASD to steer it. If you look at the RaceCar1 object you can see the PlayerInputs script has "Player Number" set to 1. The Spite used is also different. To allow us to easily have multiple players, I've created this as a Prefab - you can see this in Prefabs/RaceCar. (There is also an AIRaceCar prefab, more on that later!)

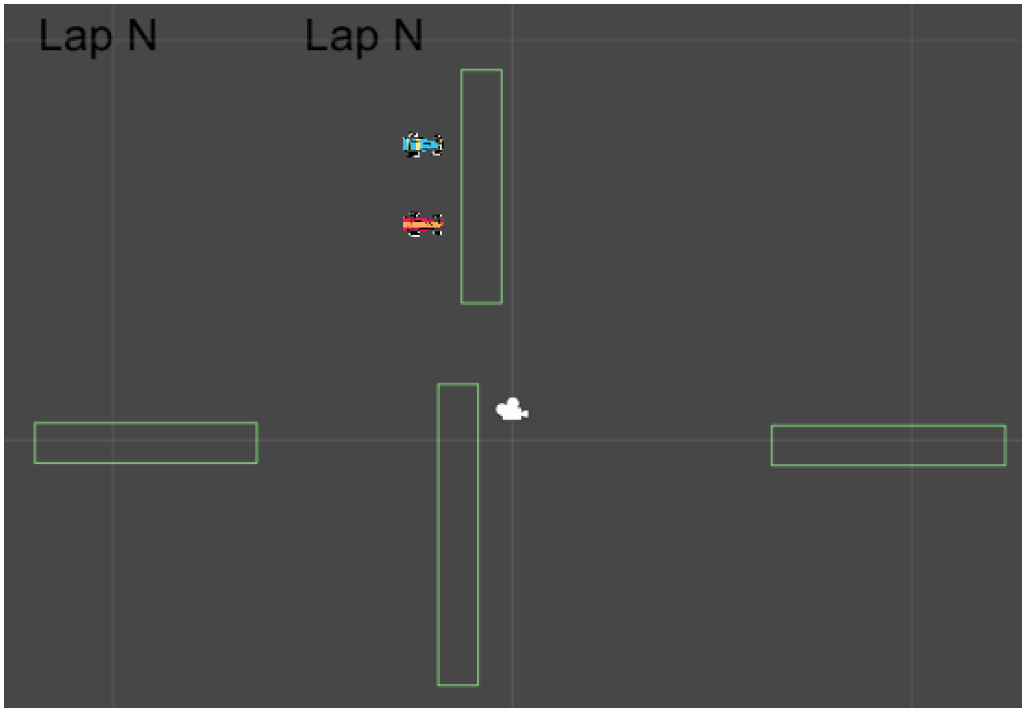
We added some lap counters! If you select LapCount1 you can see its a simple Text Mesh that was created with Game Object -> Create Other -> 3D Text (as we've done in previous tutorials).

But how do we know when we've completed a lap? If you look at the car (either prefab or an actual car) you can see we have a CarLapCounter component. That takes a 'text mesh' which it uses to output the lap counter to -- that is hooked up to our visible lap counters. There's also a start/finish line of component 'TrackLapTrigger'. If you look in the heirarchy you can see we have a 'Track' parent object which has various trigger boxes set up on it:

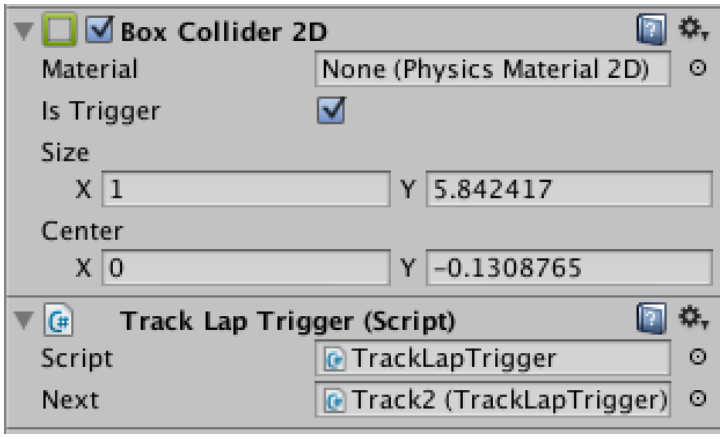




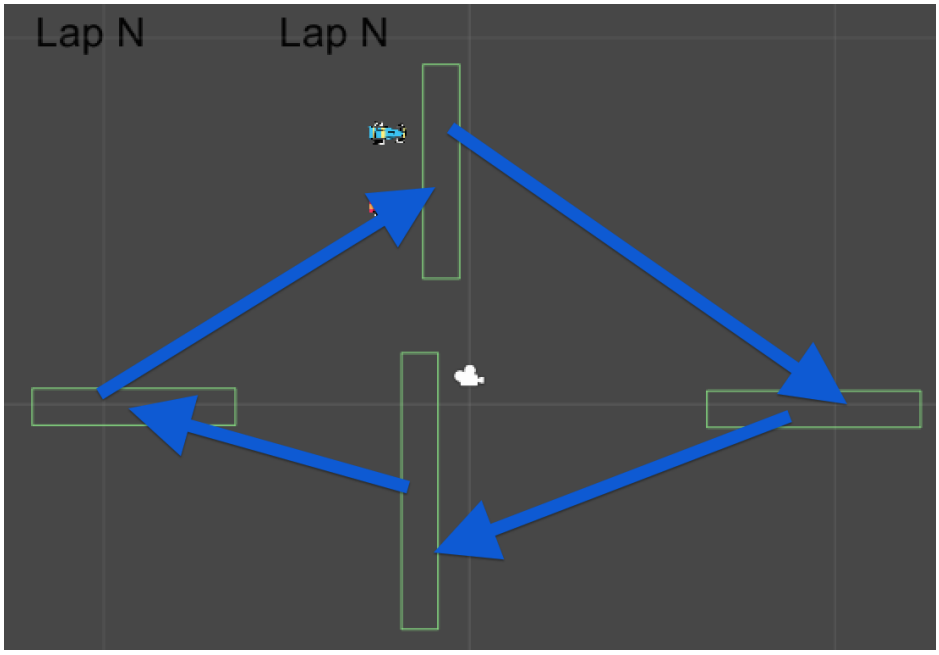
It's kind of hard to see this, you might want to hide the trackbg again:



If you select one of these (e.g. StartFinish) you will see it has a BoxCollider2D set to be a trigger, and a Track Lap Trigger component attached, which simply has a Next reference to the next trigger:



By linking all the triggers together we define where the track is. You can only go around that in one direction, so it's not possible to just go back and forth across the start/finish line to record a lap!

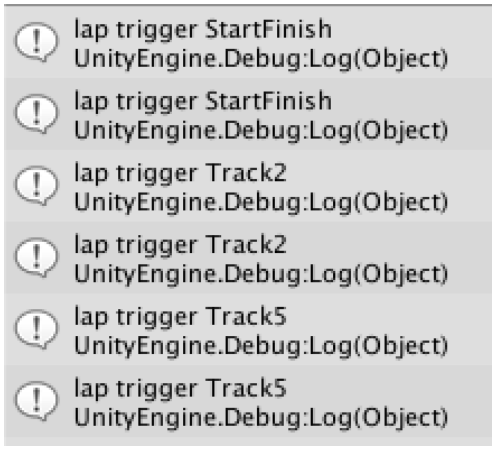


Now let's look at the code for TrackLapTrigger. It just has an OnTriggerEnter2D event:

```
void OnTriggerEnter2D(Collider2D other) {
    CarLapCounter carLapCounter = other.gameObject.GetComponent<CarLapCounter>();
    if (carLapCounter) {
        Debug.Log("lap trigger " + gameObject.name);
        carLapCounter.OnLapTrigger(this);
    }
}
```

This is called whenever a game object enters the trigger and provides information about the object. We use GetComponent() to find if a CarLapCounter just entered the trigger. If it did, we call the OnLapTrigger() function on that. (and print out a debug log so we can easily see what triggers the car is passing through as we drive around)

Now press Play and look at the Console output as you drive around.



You might wonder why there are TWO sets of messages - remember we have two CircleCollider2Ds for our cars, that's why! Fortunately it doesn't matter.

Now look at the source of CarLapCounter. If we look at OnLapTrigger first:

```
// when lap trigger is entered
public void OnLapTrigger(TrackLapTrigger trigger) {
    if (trigger == next) {
        if (first == next) {
            _lap++;
            UpdateText();
        }
        SetNextTrigger(next);
    }
}
```

The logic here is that we always have a 'next' trigger to look out for. We ignore all other triggers! (This is why the double messages in the log don't matter - we ignore the second trigger). If we've reached the start/finish line, then we increment our lap counter and call UpdateText() to update the visible lap counter. We then find the next trigger:

```
void SetNextTrigger(TrackLapTrigger trigger) {
    next = trigger.next;
    SendMessage("OnNextTrigger", next, SendMessageOptions.DontRequireReceiver);
}
```

We simply find the .next trigger set in the property inspector. But we also send a message to show we've picked a next trigger -- this will be used later by the AI!

Looking quickly at the rest of the CarLapCounter:

```
// Use this for initialization
void Start () {
    _lap = 0;
    SetNextTrigger(first);
    UpdateText();
}
```

We set the lap counter to 0 initially and find the trigger *after* the first one.

```
// update lap counter text
void UpdateText() {
    if (textMesh) {
        textMesh.text = string.Format("Lap {0}", _lap);
    }
}
```

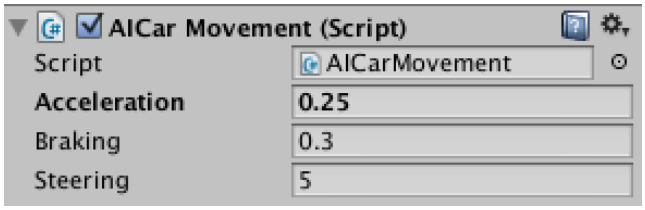
To set the lap counter text we just use handy C# string.Format() function!

Go back to unity, hit Play and drive a few laps, noticing the lap counter increments each time.

Now it's time to add AI cars! Open up Scene3 and hit play to see the AI cars drive around!

Looking at the AIRaceCar in Foreground you can see we're using the AIRaceCar prefab. These are a duplicate of the Player race cars but with a slightly different set of components. One of the great things about Unity is that you can share components between AI and Players really easily! Here we're sharing the same CarLapCounter, sprite renderer, rigidbody and circle colliders. But instead of PlayerInputs & CarMovement we have the AICarMovement component.

We can see that by changing the Acceleration and Steering values we can make the cars faster/slower - the two AI cars only differ in the sprite they use and the values of acceleration and steering.



Look at the source for the AICarMovement script. As with the player CarMovement the FixedUpdate() is where the physics magic happens:

```
void FixedUpdate() {

    SteerTowardsTarget();

    // always accelerate
    float velocity = rigidbody2D.velocity.magnitude;
    velocity += acceleration;

    // apply car movement
    rigidbody2D.velocity = transform.right * velocity;
    rigidbody2D.angularVelocity = 0.0f;
}
```

The way the AI cars work is that they pick a target to drive towards, then each turn steer a little towards it and then always accelerate. You can see the similarity between the player physics and AI physics - we could have made this completely shared by using another CarMovement behaviour and then exposed out steering and velocity inputs to it, but I didn't want to overcomplicate things.

```
void SteerTowardsTarget ()
{
    Vector2 towardNextTrigger = target - transform.position;
    float targetRot = Vector2.Angle (Vector2.right, towardNextTrigger);
    if (towardNextTrigger.y < 0.0f) {
        targetRot = -targetRot;
    }
    float rot = Mathf.MoveTowardsAngle (transform.localEulerAngles.z, targetRot, steering);
    transform.eulerAngles = new Vector3 (0.0f, 0.0f, rot);
}
```

How does the AI steer 'towards' the target? Well we do some 2d maths - this is actually the same maths we used back in the Angry Scrawls tutorial for aiming, to get the angle to rotate at a position. So we get the angle that we want to rotate to so that we're pointing directly at the target.

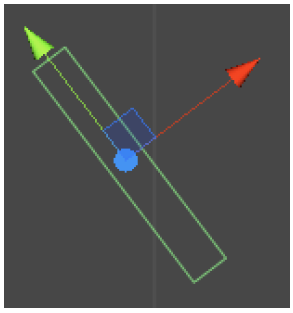
There is a handy function in unity Mathf.MoveTowardsAngle() which takes our current angle,the desired angle, and a maximum amount that we can turn. It then produces the actual angle. This is aware that there are 360 degrees and so will always turn the 'shortest way' to get to the desired angle.

Finally for the AI car we need to pick a destination point to aim at:

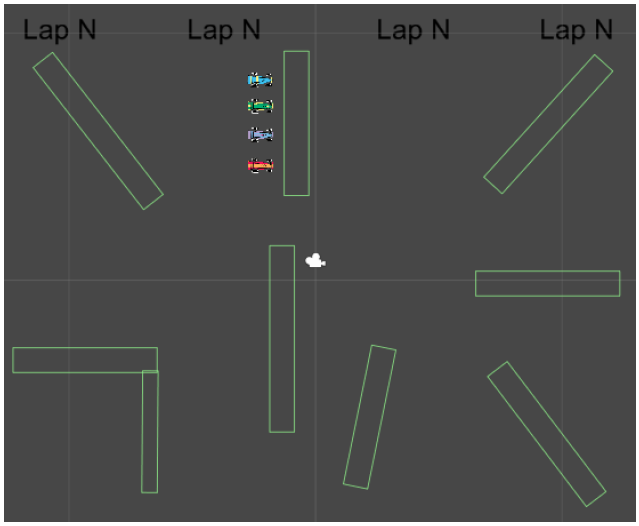
```
public void OnNextTrigger(TrackLapTrigger next) {

    // choose a target to drive towards
    target = Vector3.Lerp(next.transform.position - next.transform.right,
                          next.transform.position + next.transform.right,
                          Random.value);
}
```

If you remember in the CarLapCounter we sent the OnNextTrigger message when we SetNextTrigger(). The AI car receives this message and then chooses a random location inside the trigger box. It does that by taking the Random.value (always between 0 and 1) and then feeding it to Vector3.Lerp() which interpolates between two positions. We use the trigger's transform.position and the .right vector which gives us an idea of its size/direction. If you want to see the 'right' vector it is the red arrow in the transform. Looking at that I realise we should probably have been using the green arrow, so go ahead and change next.transform.right to be next.transform.up and that will be more correct!



This gives us a very simple way for the AI car to navigate around! But because the car is really dumb, to make it work we had to add more TrackLapTriggers. If you select the Track you can see them:



Without these, it's hard for the car to work out how to navigate around corners. But it's surprisingly effective for a simple racing game!