Making a marble-rolling game in Unity3D

Introduction

If you've never used Unity before, it can be a little daunting to produce a project from scratch. Instead of trying to do that then, we're going to work from an incomplete project and get it working as a basic game.

The template is of a marble-rolling game along the lines of Marble Madness or Monkey Ball. All you'll need to get started is a Mac or a Windows laptop, the Unity editor (linked below) and the template project. Hopefully I'll have both of these files on a USB stick. Call me over if you're having trouble with slow downloads.

Unity Editor download link: http://unity3d.com/unity/download

Project template download link: https://bitbucket.org/CardTrick/marble-tutorial/downloads

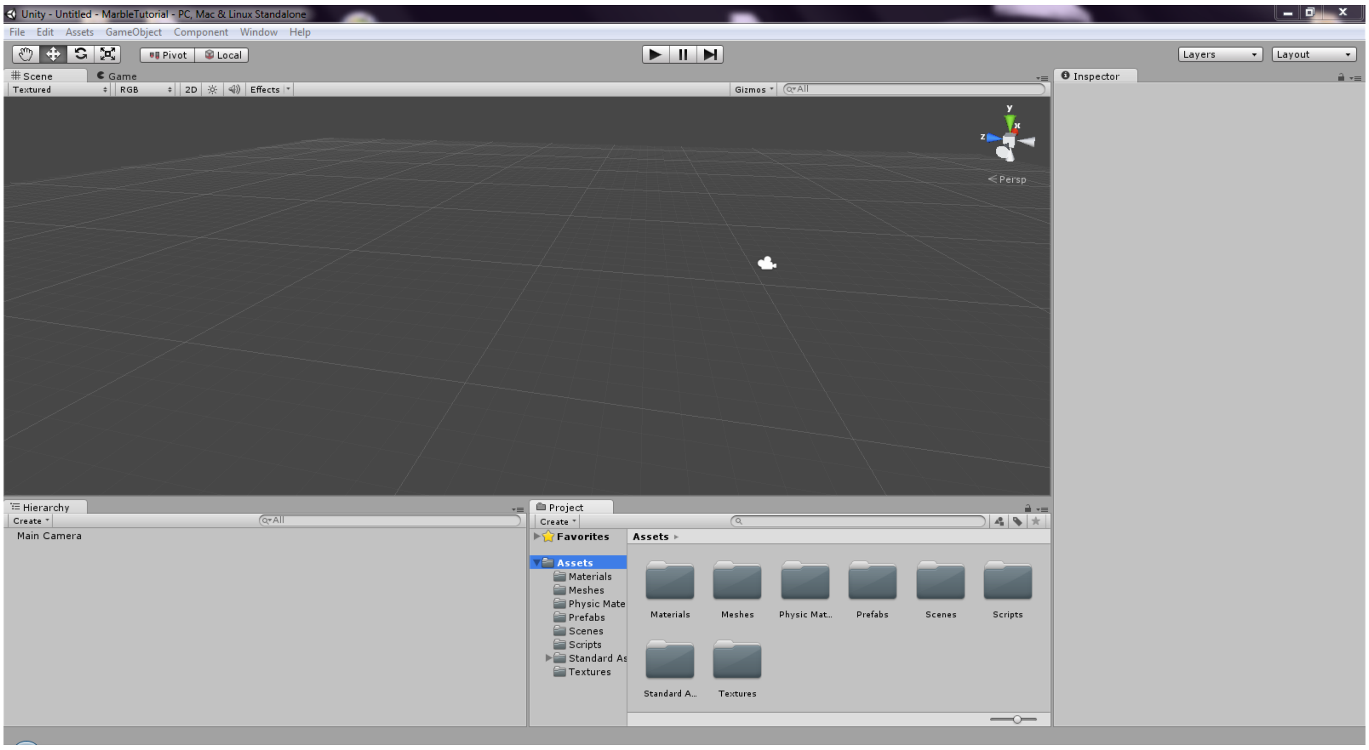
The programming language we'll use is **C#**. It's similar to Java, so if you have experience there, this tutorial will hopefully be a breeze. If you have no programming experience at all, it may be more intimidating, but you shouldn't get stuck if you follow the tutorial.

One final note: it's very likely that I've missed a few thing in this tutorial. If you're at all confused and need help, don't hesitate to ask.

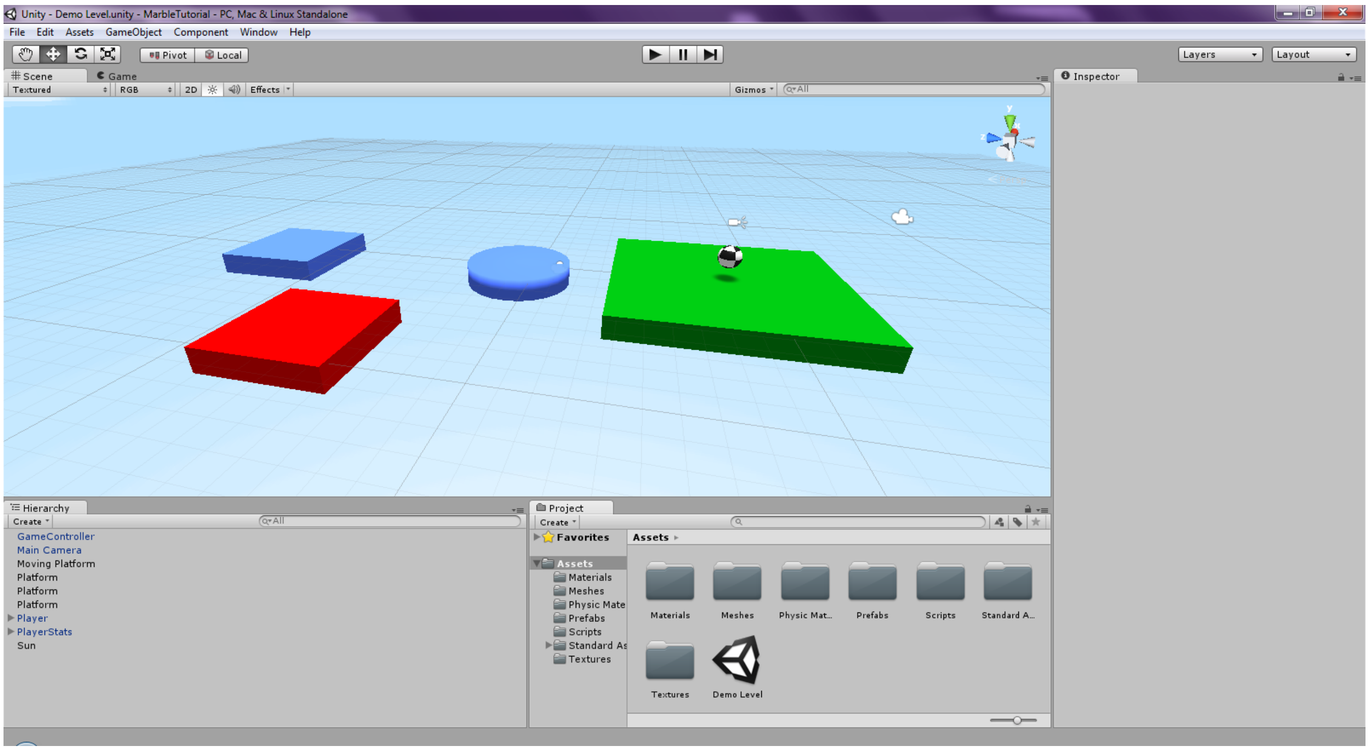
Setting Up

Once you have Unity installed, start it up and you should get a small window asking you to create or open a project. In the **Open Project** tab, click the "**Open Other...**" button at the bottom and choose the template project. It should be a folder named "**MarbleTutorial**".

The editor should open showing a blank scene. To make sure what you're seeing matches my screenshots more closely, go to the menu bar and select **Window → Layouts → Wide**. You should then see something like this.



Next, choose **File → Open Scene** and select "**Scenes/Demo Level.unity**". Now you should be seeing this:



A few important parts of the editor window: the largest pane is the **Scene view** which shows all objects in the scene. The bottom-left panel is the **Hierarchy** which lists all objects in the scene. Clicking on an object in either of these panes will show you details of that object in the **Inspector** – the panel on the right. Finally, the lower-right panel is the file manager which shows all the asset files of the project.

Feel free to mess around with everything in the scene until your curiosity is satisfied. Don't worry about making changes, you can always just reload the scene (or at worst, re-download the project.) Once you're ready, read on to the tutorial.

Tutorial

This tutorial is split into a few different sections:

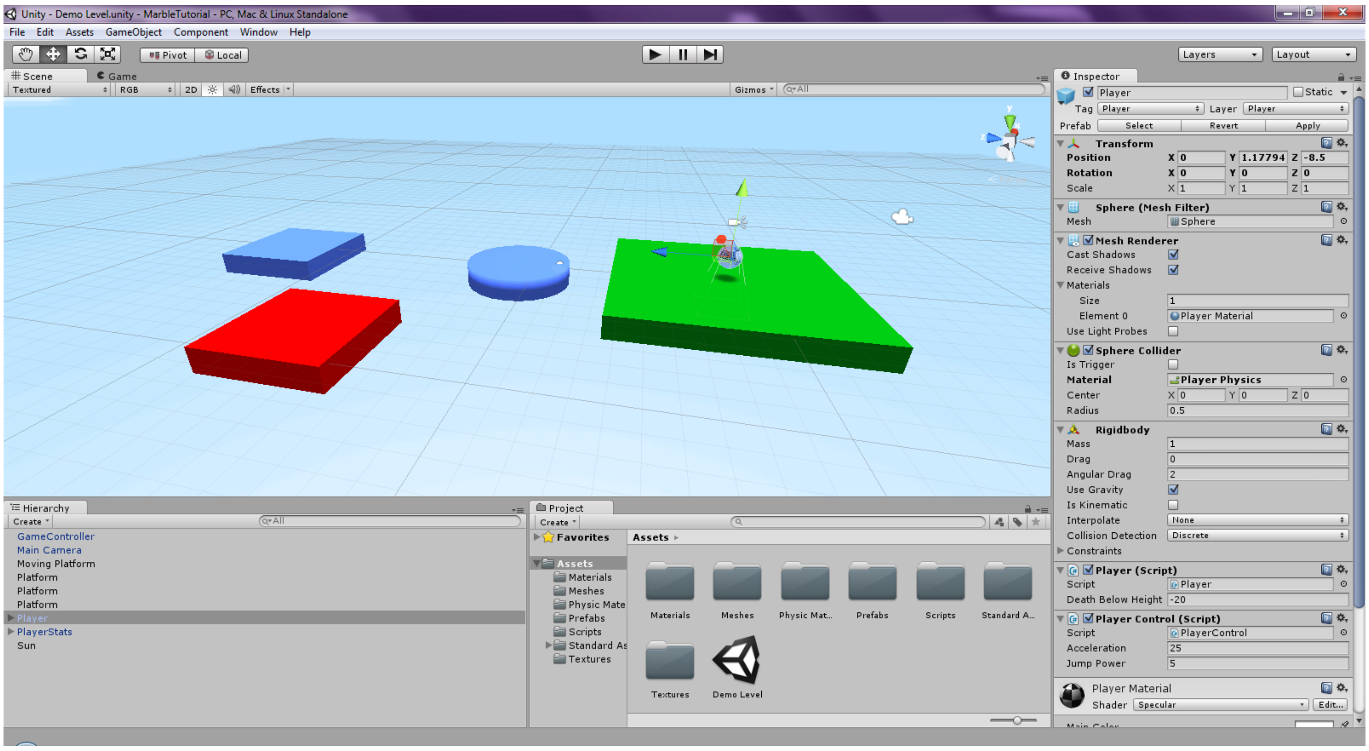
1.	Unity Basics	
2.	Player Movement	(Axis input and Rigidbody torque)
3.	Player Jumping	(Button input and Physics.CheckSphere)
4.	Pickups	(Making a prefab and Triggers)
5.	Moving Platforms	(Kinematic objects, Physics.movePosition and Lerping)
6.	The Goalposts	(Raycasting and Transform.Find)

Each of these six things will introduce you to an important part of how Unity works. You can do them in any order since they don't really depend on each other. It's probably a good idea to start with the first and second ones though so you can actually move around the level and look at things.

Unity Basics

In Unity, every object in your game is a **GameObject** and all of the properties of that object are given by its **Components**. For example, the **Rigidbody** component allows a GameObject to be affected by physics.

That little black and white ball you see in the level is a GameObject. The player in fact. If you click on it, you'll be able to see all of its Components in the inspector at the side.



```
Player                         Static
Tag Player          Layer Player
Prefab  Select  Revert  Apply
Transform
Position        X 0    Y 1.17794   Z -8.5
Rotation        X 0    Y 0         Z 0
Scale           X 1    Y 1         Z 1
Sphere (Mesh Filter)
Mesh            Sphere
Mesh Renderer
Cast Shadows    ✓
Receive Shadows ✓
Materials
  Size          1
  Element 0     Player Material
Use Light Probes ☐
Sphere Collider
Is Trigger      ☐
Material        Player Physics
Center          X 0    Y 0    Z 0
Radius          0.5
Rigidbody
Mass            1
Drag            0
Angular Drag    2
Use Gravity     ✓
Is Kinematic    ☐
Interpolate     None
Collision Detection Discrete
Constraints
Player (Script)
Script          Player
Death Below Height -20
Player Control (Script)
Script          PlayerControl
Acceleration    25
Jump Power      5
Player Material
Shader Specular   Edit...
```

You'll see the first Component is a **Transform**. This is the location and orientation of the GameObject in 3D space. It also has a **Mesh Renderer** which is what causes the object to appear on screen, and a Rigidbody.

Moving around the scene

To make any changes, you'll need to be able to look around. Use the **Right click** to rotate the camera and the **Middle click** to pan around the scene. You're probably on a laptop though and that may not be easy, so here is a good alternative: hold the Right Mouse Button and use the WASD keys to fly around as you would in a first-person game. You can also use Q and E to raise and lower your point of view.

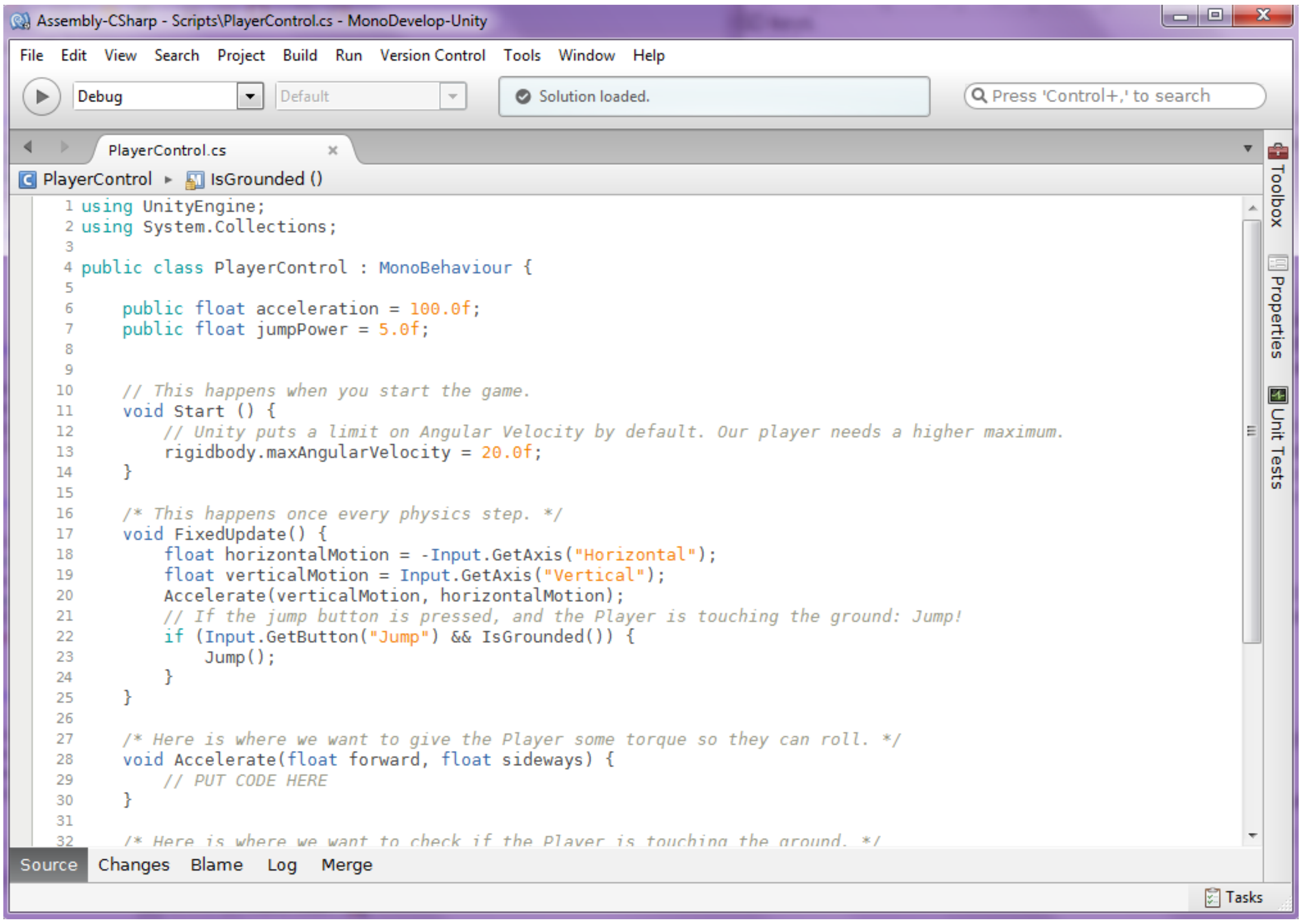
Moving objects in the scene

To reposition objects, you can edit the values in the **Transform** component in the Inspector panel on the right. However, it's usually easier to move them by dragging the coloured arrows that are centred on the object in the scene view when you've selected it. The buttons at the top left allow you to change modes between Moving, Rotating and Scaling.

Player Movement

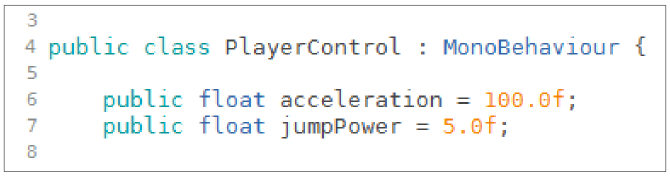
The player also has two components: **Player** and **Player Control** which are **Scripts**. These are small programs that can control the GameObject however you like. At the moment, the Player Control script doesn't do much, so we're going to edit it.

In the little file manager at the bottom of the editor, go to the "**Scripts**" folder and double click the "**PlayerControl**" file to edit it.


Assembly-CSharp - Scripts\PlayerControl.cs - MonoDevelop-Unity
File Edit View Search Project Build Run Version Control Tools Window Help
Debug
Default
Solution loaded.
Press 'Control+,' to search
PlayerControl.cs
PlayerControl ▸ IsGrounded ()
Toolbox
Properties
Unit Tests
Source Changes Blame Log Merge
Tasks

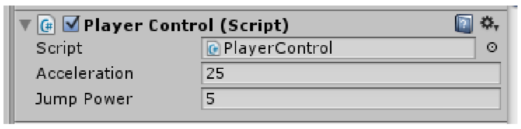
```
using UnityEngine;
using System.Collections;

public class PlayerControl : MonoBehaviour {

    public float acceleration = 100.0f;
    public float jumpPower = 5.0f;


    // This happens when you start the game.
    void Start () {
        // Unity puts a limit on Angular Velocity by default. Our player needs a higher maximum.
        rigidbody.maxAngularVelocity = 20.0f;
    }

    /* This happens once every physics step. */
    void FixedUpdate() {
        float horizontalMotion = -Input.GetAxis("Horizontal");
        float verticalMotion = Input.GetAxis("Vertical");
        Accelerate(verticalMotion, horizontalMotion);
        // If the jump button is pressed, and the Player is touching the ground: Jump!
        if (Input.GetButton("Jump") && IsGrounded()) {
            Jump();
        }
    }

    /* Here is where we want to give the Player some torque so they can roll. */
    void Accelerate(float forward, float sideways) {
        // PUT CODE HERE
    }

    /* Here is where we want to check if the Player is touching the ground. */
```

We're only going to add a little bit of code here, but first a few details about scripts. Note the two lines near the top, each defining a value.



```

public class PlayerControl : MonoBehaviour {

    public float acceleration = 100.0f;
    public float jumpPower = 5.0f;

```

The **public** part means that any other script can "see" this value, and also that this value can be changed in the inspector. The numbers in the Script file are just the default values. I've already changed them in the Inspector, as you can see on the right.



Player Control (Script)

Script	PlayerControl
Acceleration	25
Jump Power	5

The **float** part is the type of value that it is. Float means floating-point, e.g. a number with a decimal point.

Under this are a few blocks of code. Each one is a **function** (or **method**). Let's look at the **FixedUpdate** function. You can see a comment just above it, explaining when it happens.

```

    /* This happens once every physics step. */
    void FixedUpdate() {
        float horizontalMotion = -Input.GetAxis("Horizontal");
        float verticalMotion = Input.GetAxis("Vertical");
        Accelerate(verticalMotion, horizontalMotion);
        // If the jump button is pressed, and the Player is touching the ground: Jump!
        if (Input.GetButton("Jump") && IsGrounded()) {
            Jump();
        }
    }
```

The first two lines here use the **Input.GetAxis** function to get a number for the player controls. An axis is a virtual joystick that can have a value between -1 and 1. The Horizontal and Vertical axes are built into Unity by default, and are mapped to both the arrow keys and WASD keys.

So for example, if you press the A key, **Input.GetAxis("Horizontal")** will give you the value -1. If you press D, it will give you the value 1. Similar rules apply to the "Vertical" axis.

So now we have two values: **horizontalMotion** and **verticalMotion** which represents the directions that the player is trying to move in. On the third line, we can see that the Accelerate function is being **called** (e.g. executed) and it is being **passed** these motion values. Now we need to put some code in Accelerate to make it actually do something.

The Acceleration Function

```
    /* Here is where we want to give the Player some torque so they can roll. */
    void Accelerate(float forward, float sideways) {
        // PUT CODE HERE
    }
```

In physics, rotation force is called **torque**. We need to give the player the correct torque to make them roll the correct amount forward and sideways, based on the values from the virtual joysticks.

This first line we'll add creates a torque vector which is the axis which we will rotate the player around.

```
    /* Here is where we want to give the Player some torque so they can roll. */
    void Accelerate(float forward, float sideways) {
        Vector3 torqueVector = new Vector3(forward, 0, sideways);

    }
```

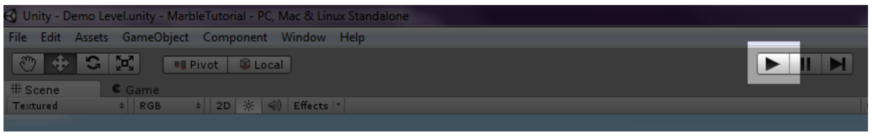
A vector is essentially a direction. It has X, Y and Z components. X is how far to the left it goes, Y is how far upwards, and Z is how far forwards.

For our ball to roll forwards, we want to spin it *around* the X axis. For it to roll sideways, we want to spin it around the Z axis. We leave the Y component as zero.

In our second line, we apply this torque to the ball. The variable "rigidbody" refers to the Rigidbody component of the object that the script is attached to.

```
    /* Here is where we want to give the Player some torque so they can roll. */
    void Accelerate(float forward, float sideways) {
        Vector3 torqueVector = new Vector3(forward, 0, sideways);
        rigidbody.AddTorque(torqueVector * acceleration, ForceMode.Acceleration);
    }
```

We're then calling the AddTorque function that belongs to that Rigidbody, and passing it a torque vector. We've multiplied (the * symbol) that torque vector by the 'acceleration' value so that it will roll faster when we increase this value. The other part, **ForceMode.Acceleration**, just tells Unity that we want to affect the acceleration directly and ignore the mass of the object. It's not too important for now.


Unity - Demo Level.unity - MarbleTutorial - PC, Mac & Linux Standalone
File Edit Assets GameObject Component Window Help
Pivot
Local
Scene
Game
Textured
RGB
2D
Effects

So if you added that new code just right, you'll be able to hit the play button at the top of the Unity window and roll about using the keyboard! If you still don't understand exactly what you did, don't worry about it, it will make sense over time. Just enjoy rolling about for a while and then continue with the tutorial.

Player Jumping

To get across gaps, we'll need our player to be able to jump. We'll put this code in the same script as we used for the movement. Look to the last three lines of the **FixedUpdate** function in **PlayerControl**.

```

    /* This happens once every physics step. */
    void FixedUpdate() {
        float horizontalMotion = -Input.GetAxis("Horizontal");
        float verticalMotion = Input.GetAxis("Vertical");
        Accelerate(verticalMotion, horizontalMotion);
        // If the jump button is pressed, and the Player is touching the ground: Jump!
        if (Input.GetButton("Jump") && IsGrounded()) {
            Jump();
        }
    }
```

These lines essentially say: “if the jump button is pressed AND the player is on the ground: make the player jump.”

The **Input.GetButton** function is a bit like GetAxis, but instead of giving us a number between -1 and 1, it just returns **true** if the button is pressed and **false** if it isn't. Values that can be true or false are called booleans, or **bool**s in C#.

Let's look at the Jump function before we get to IsGrounded.

The Jump Function

```
    /* Here is where we want to give the player some vertical velocity so they jump. */
    void Jump() {
        // PUT CODE HERE
    }
```

To make our player jump, all we want to do is give them some upward velocity. So in Unity, that means giving them some velocity in the Y axis.

The first thing we need to do is make a vector for this new velocity. We set it equal to “rigidbody.velocity” which is the ball's current velocity.

```
    /* Here is where we want to give the player some vertical velocity so they jump. */
    void Jump() {
        Vector3 newVelocity = rigidbody.velocity;
    }
```

On the next line, we simply set the Y value of this new velocity to our “jumpPower”.

```
    /* Here is where we want to give the player some vertical velocity so they jump. */
    void Jump() {
        Vector3 newVelocity = rigidbody.velocity;
        newVelocity.y = jumpPower;
    }
```

Finally, we set our Rigidbody's velocity to our new velocity.

```
    /* Here is where we want to give the player some vertical velocity so they jump. */
    void Jump() {
        Vector3 newVelocity = rigidbody.velocity;
        newVelocity.y = jumpPower;
        rigidbody.velocity = newVelocity;
    }
```

That was reasonably simple. Make sure the code is right and then hit the play button to test it out. Use the **spacebar** or **Z key** to jump.

You may have noticed that you can jump even while you're in the air. Remember back in FixedUpdate when we used the **IsGrounded** function? At the moment that always returns **true** even if the player is *not* grounded. Let's change that.

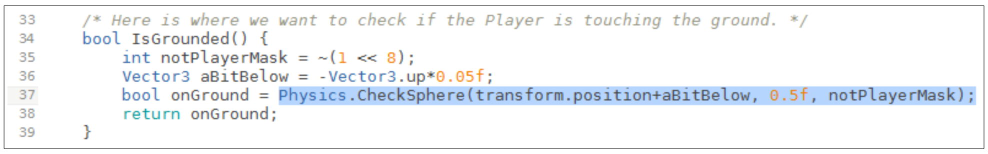
The IsGrounded Function

```
    /* Here is where we want to check if the Player is touching the ground. */
    bool IsGrounded() {
        int notPlayerMask = ~(1 << 8);
        Vector3 aBitBelow = -Vector3.up*0.05f;
        bool onGround = true; // CHANGE THIS LINE
        return onGround;
    }
```

The first two lines give us a couple of values we need. The first one allows us to filter out the player when we're looking for ground objects. The second gives us a vector that is slightly below the player's centre. Don't think about either of them too much at the moment.

The third line at the moment just says that onGround is true. E.g. we are always “on ground”. The fourth line then returns this value to where the program asked for it. In this case, it was FixedUpdate. Notice that it says “bool” before the name of the function, where it said “void” for all the others. This is because IsGrounded returns a “bool” type: e.g. true or false, whereas the others returned nothing at all: void.

We need to change the third line so that it is only true when we are actually on the ground. To do this, we are going to look at a spherical area just below the player to see if there are any objects intersecting it. Unity's function for doing this is **Physics.CheckSphere**. It returns true if there is anything in the sphere and false otherwise.



```
/* Here is where we want to check if the Player is touching the ground. */
bool IsGrounded() {
    int notPlayerMask = ~(1 << 8);
    Vector3 aBitBelow = -Vector3.up*0.05f;
    bool onGround = Physics.CheckSphere(transform.position+aBitBelow, 0.5f, notPlayerMask);
    return onGround;
}
```

Three values are passed to CheckSphere. The first is a vector which points to the position of the sphere. Our value is “transform.position+aBitBelow”. This is the position of our ball, plus a little bit below. So just underneath. The second value is the radius of the sphere. We've set it to 0.5 (the 'f' just means its a float) which is the radius of the player. The last is the value we're using to filter collisions. Again, don't think too much about that for now.

This is the first complex Unity function we've used. Nobody ever remembers exactly what to pass into this function. Thankfully, there's documentation! Here is the documentation page for Physics.CheckSphere if you're interested: http://docs.unity3d.com/Documentation/ScriptReference/Physics.CheckSphere.html

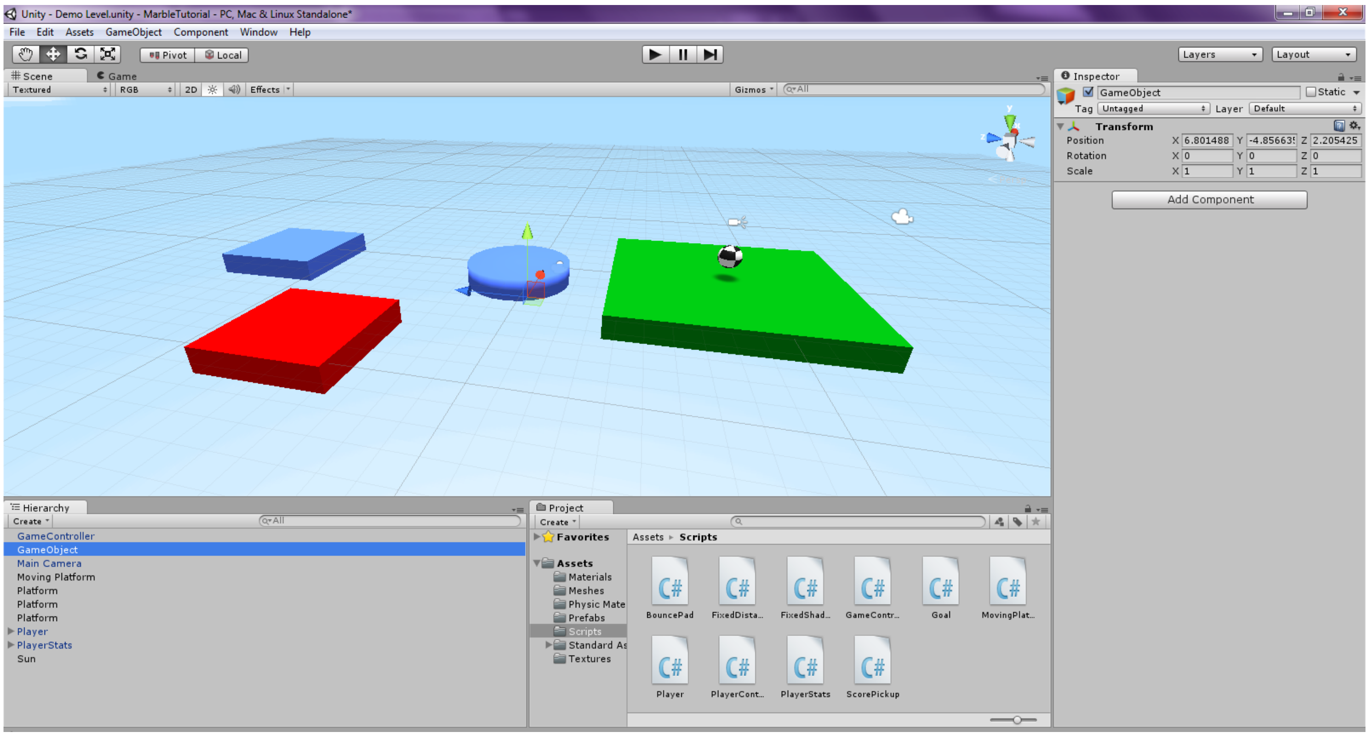
Testing the game again should show that we can now only jump when we're on the ground. (Or technically, when we're very close to the ground.)

Pickups

I'm sure you've noticed that score counter at the top of the screen that always says zero. Here we're going to make an object that you can roll into and pick up, increasing your score!

This is going to be a little less about programming and a little more about using Unity's interface.

When you want to make a new *kind* of object, the first thing you need to do is make an example of it. So the first thing we want to do is make a new empty object. At the top menu bar, choose **GameObject → CreateEmpty**. You should end up with this:

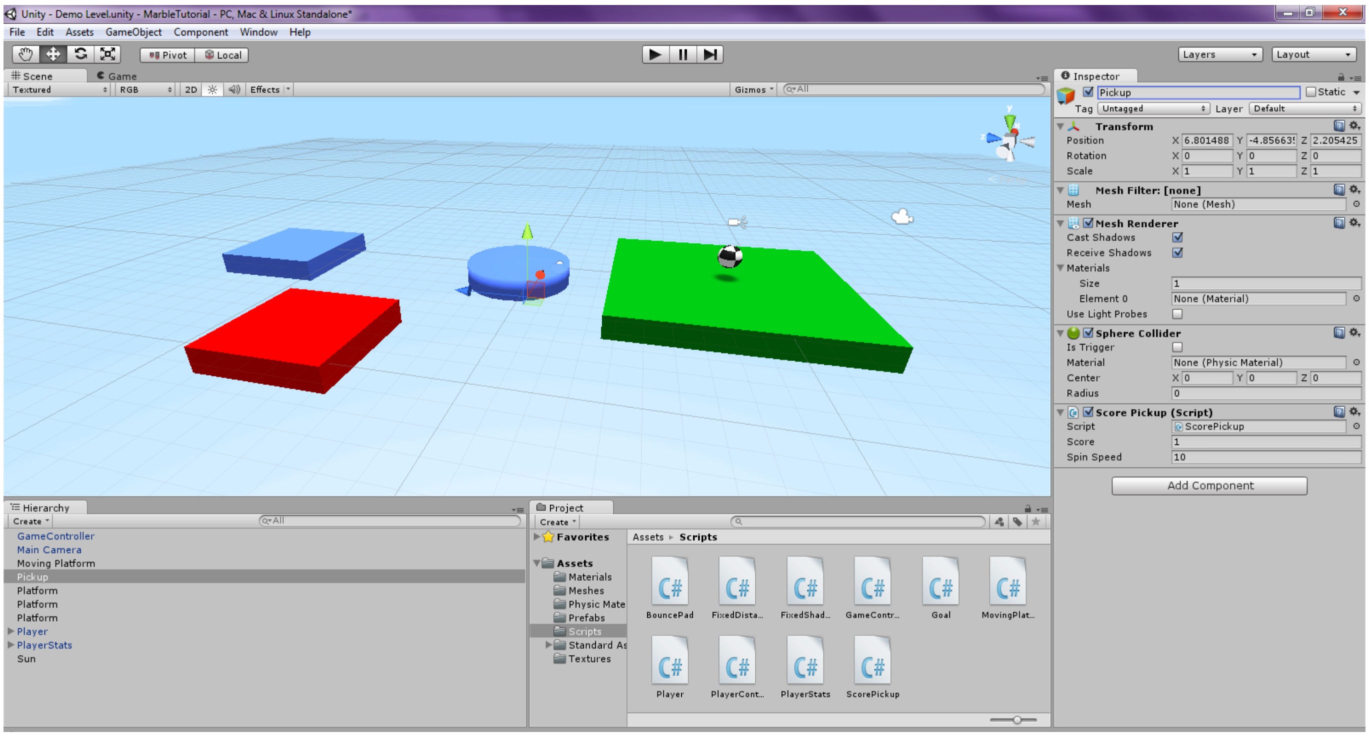


GameObject ☐ Static
Tag Untagged Layer Default
Transform
Position X 6.801488 Y -4.85663 Z 2.205425
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

To edit this new GameObject, you'll need to use the Inspector panel at the side. Change it's name from GameObject to something meaningful using the text field at the top. Next, add the following components with the Add Component button:

1. **Mesh → Mesh Filter** (so that it has a mesh)
2. **Mesh → Mesh Renderer** (so that we can see it)
3. **Physics → Sphere Collider** (so that the player can touch it)
4. **Scripts → Score Pickup** (the script we'll use to make it pickup-able)

So now you should have something like this:



GameController
Main Camera
Moving Platform
Pickup
Platform
Platform
Platform
Player
PlayerStats
Sun

Assets: Materials, Meshes, Physic Mate, Prefabs, Scripts, Standard As, Textures

BouncePad, FixedDista..., FixedShad..., GameContr..., Goal, MovingPlat..., Player, PlayerCont..., PlayerStats, ScorePickup

Pickup ☐ Static
Tag Untagged Layer Default

Transform
Position X 6.801488 Y -4.85663 Z 2.205425
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Mesh Filter: [none]
Mesh None (Mesh)

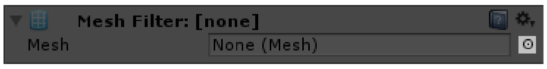
Mesh Renderer
Cast Shadows ✓
Receive Shadows ✓
Materials
Size 1
Element 0 None (Material)
Use Light Probes ☐

Sphere Collider
Is Trigger ☐
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0

Score Pickup (Script)
Script ScorePickup
Score 1
Spin Speed 10

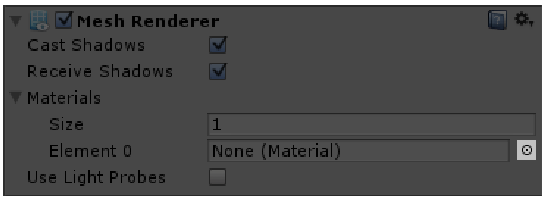
You'll notice that we still can't see it. Well it has all the right components, but now we need to configure them a little.

First we need to give it a mesh by clicking the small circle next to the Mesh field of the MeshFilter:


Mesh Filter: [none]
Mesh None (Mesh)

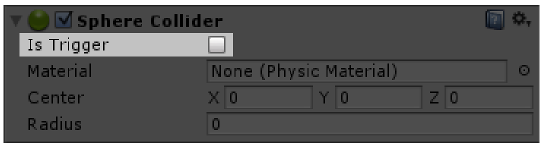
You can choose any mesh you like; I made a gem-shaped mesh for just such an occasion.

Next, we need to give it a material:


Mesh Renderer
Cast Shadows ✓
Receive Shadows ✓
Materials
Size 1
Element 0 None (Material)
Use Light Probes ☐

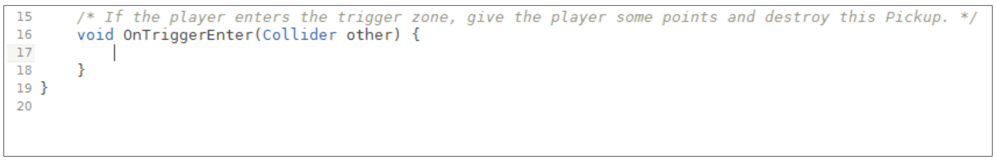
This determines the colour, texture, reflectiveness and so on. You can have a look at the materials that are in the project already and play around with them, or you can just use an existing one. To give an object a material, you can either use the little circle, or you can just drag the material itself onto the object in the Scene.

Aha, now we can see it, but there's still one more important thing we need to do. Everything physical in Unity will have Colliders. This is how they bounce off each other and such. However, we don't really want the player to hit into these pickups, we just want to detect when the player is overlapping with it. So check the checkbox on the SphereCollider that says "Is Trigger".


Sphere Collider
Is Trigger ☐
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0

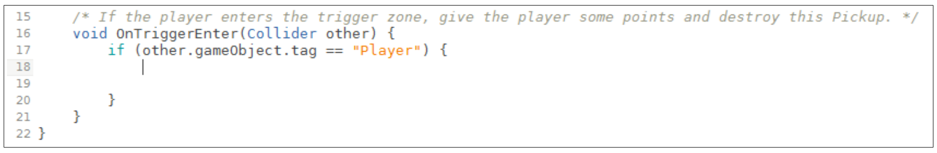
Triggers are a bit like colliders, but instead of being affected by physics, they just let us know when something is being touched. This will make more sense when we look at the script, so go ahead and open the **ScorePickup** script for editing.

The OnTriggerEnter Function


```
	/* If the player enters the trigger zone, give the player some points and destroy this Pickup. */
	void OnTriggerEnter(Collider other) {
		|
	}
}
```

Any script that is attached to an object which has a collider can have a function called **OnCollisionEnter** which allows you to write code that will happen whenever an object is collided with. There is an equivalent for triggers called **OnTriggerEnter** which behaves similarly, and it's what we're going to use for our Pickups.

The first thing we have to do in this script is make sure that our trigger only activates when it is the *player* that enters it. If you have a look at the player, you can see in the Inspector that it has a tag: "Player". This is what we will use.

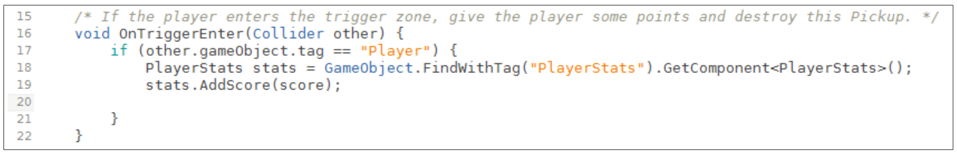


```
    /* If the player enters the trigger zone, give the player some points and destroy this Pickup. */
    void OnTriggerEnter(Collider other) {
        if (other.gameObject.tag == "Player") {


        }
    }
}
```

The collider “other” is the collider of whatever has entered the trigger zone. This collider must belong to a GameObject, and so we can get that GameObject with “other.gameObject”. We then check if the tag of that GameObject is “Player”.

(In most programming languages, a double equals sign means “is equal to”, while a single equals sign means “assign this value to”. C# is the same.)

The code inside this “if” block will only execute if the condition is met, so we write the code for increasing our score here.



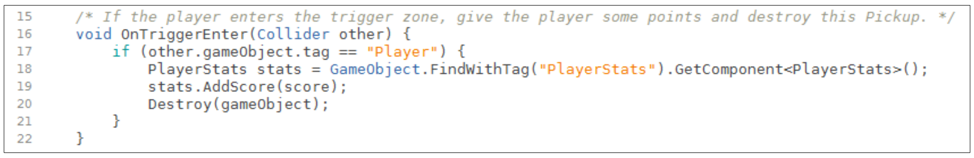
```
    /* If the player enters the trigger zone, give the player some points and destroy this Pickup. */
    void OnTriggerEnter(Collider other) {
        if (other.gameObject.tag == "Player") {
            PlayerStats stats = GameObject.FindWithTag("PlayerStats").GetComponent<PlayerStats>();
            stats.AddScore(score);

        }
    }
```

The first of these lines is a little hard to explain. The GameObject with the tag “PlayerStats” holds the score text as well as the amount of score the player has. **GameObject.FindWithTag(“PlayerStats”)** will get this object for us.

From there, we wish to get the Script called PlayerStats from this object, which is what **GetComponent** does.

So now that we have the PlayerStats script in a variable called “stats”, we want to call the **AddScore** function of it to increase our score. If you're curious, you can look at the PlayerStats script to see how this function works, but it's quite simple.

Now that we have increased our score, we need to destroy the pickup so that we can't collect it twice. This is very easy:



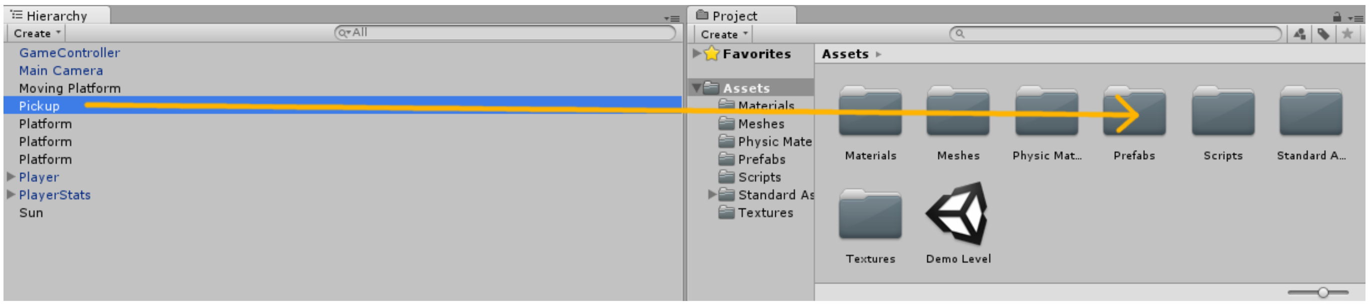
```
    /* If the player enters the trigger zone, give the player some points and destroy this Pickup. */
    void OnTriggerEnter(Collider other) {
        if (other.gameObject.tag == "Player") {
            PlayerStats stats = GameObject.FindWithTag("PlayerStats").GetComponent<PlayerStats>();
            stats.AddScore(score);
            Destroy(gameObject);
        }
    }
```

We pass the object that we want to destroy to the **Destroy** function. In this case, we want to use the GameObject that this script is attached to. The variable “gameObject” is this. This is similar to how we have used “transform” and “rigidbody” in earlier parts of the tutorial.

You can test if your pickup works just now if you like. Once you're satisfied, you'll want to save this object as a **Prefab**. A prefab is essentially a template. Once you've done this, you can place as many copies of the object as you'd like. You can also change them in bulk by changing the prefab itself.

Saving a prefab

Saving the prefab is as simple as dragging the object from the Hierarchy panel into the File Manager panel:



And we're done!

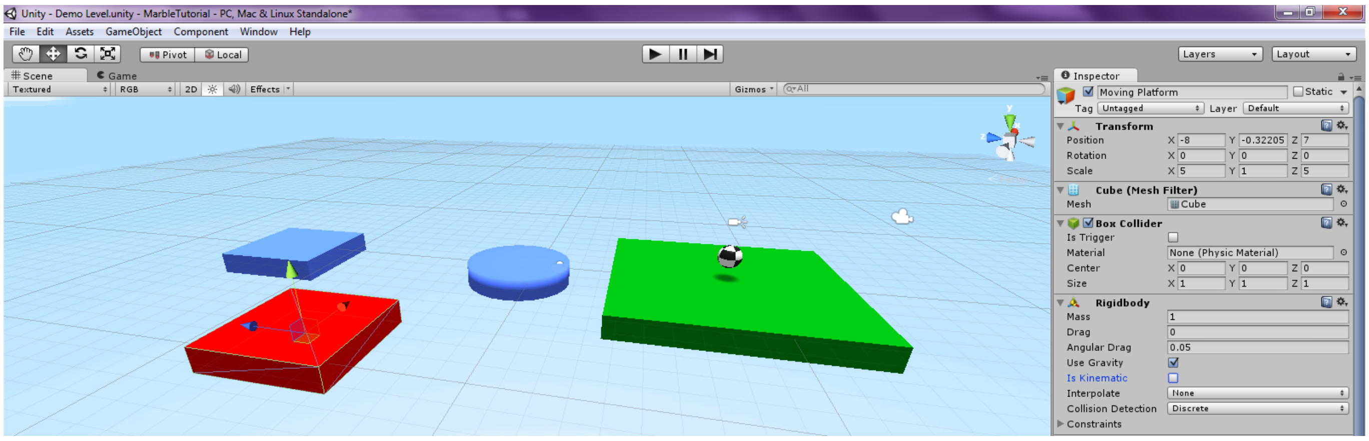
You can drag prefabs from the file manager onto the scene itself to place them in the level. Place a few more pickups just to be sure they're working.

Moving Platforms

Perhaps you've noticed the red platform in the level that just falls down whenever you start. That's because it's going to be our moving platform. The reason it falls and other things don't is that it has a **Rigidbody** component, just like the Player.

However, we can't easily move this platform around with physics. We'd be constantly balancing forces so that it didn't drop out of the sky as soon as the player landed on it. So we need it to respond to the game physics, but only in the ways that we want it to. For this, we use something called a **kinematic rigidbody**.

Click on the red platform in the Scene editor and look over at its Rigidbody component.


Unity - Demo Level.unity - MarbleTutorial - PC, Mac & Linux Standalone*
File Edit Assets GameObject Component Window Help
Pivot Local
Layers Layout
Scene Game
Textured RGB 2D Effects Gizmos
Inspector

```
Moving Platform                       Static
Tag Untagged          Layer Default
Transform
Position   X -8   Y -0.32205  Z 7
Rotation   X 0    Y 0         Z 0
Scale      X 5    Y 1         Z 5
Cube (Mesh Filter)
Mesh       Cube
Box Collider
Is Trigger
Material   None (Physic Material)
Center     X 0    Y 0         Z 0
Size       X 1    Y 1         Z 1
Rigidbody
Mass                1
Drag                0
Angular Drag        0.05
Use Gravity         ✓
Is Kinematic
Interpolate         None
Collision Detection Discrete
Constraints
```

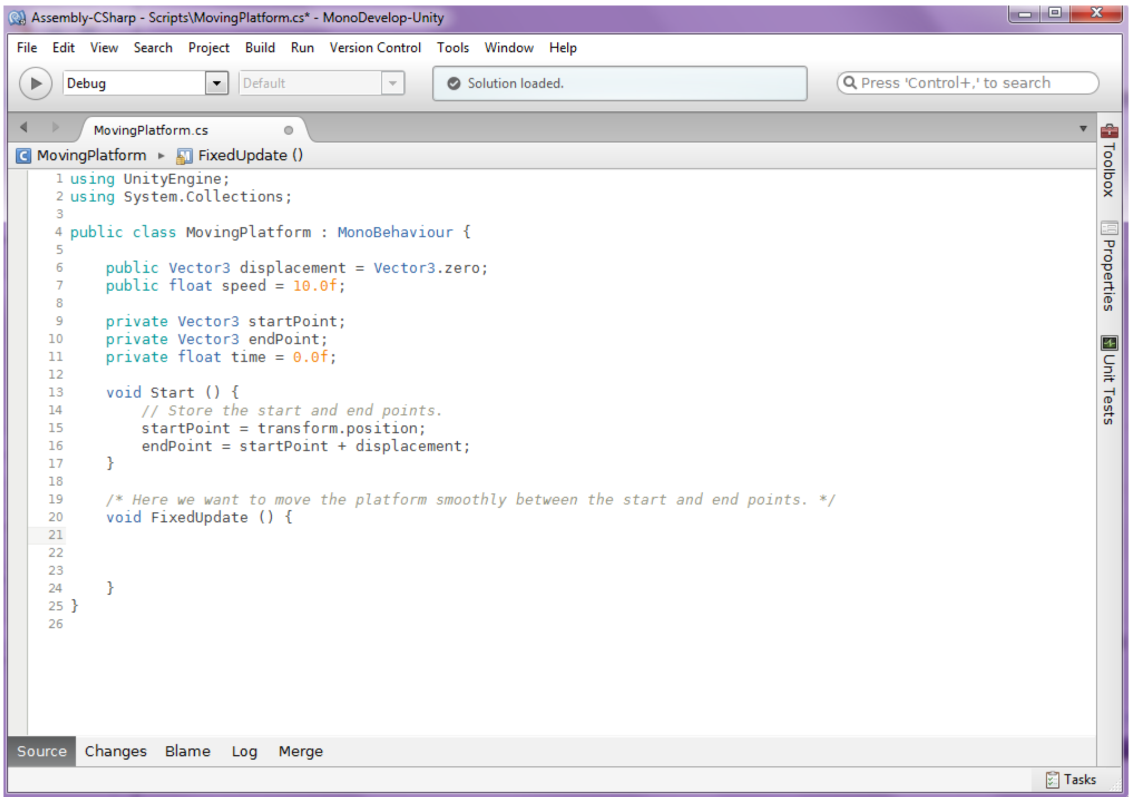
We don't want it to be affected by gravity, so untick the **Use Gravity** checkbox. We also don't want it to be affected by normal forces: we only want it to be affected by our own scripts, so tick the **Is Kinematic** box.

Now we have to do the scripting.

If you look at the **MovingPlatform** component in the sidebar, you'll notice it has two public variables: Displacement and Speed. The displacement is supposed to be a vector for where the platform will move to before returning to its original position. At the moment, it's supposed to move 15 units along X, and then come back to where it started. We want this to be a smooth motion as well.

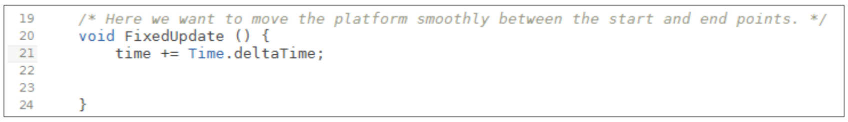
Open up the **MovingPlatform** script and we'll try to make this happen.

The MovingPlatform Script


Assembly-CSharp - Scripts\MovingPlatform.cs* - MonoDevelop-Unity
File Edit View Search Project Build Run Version Control Tools Window Help
Debug
Default
Solution loaded.
Press 'Control+,' to search
MovingPlatform.cs
MovingPlatform ▸ FixedUpdate ()
Toolbox
Properties
Unit Tests
Source Changes Blame Log Merge
Tasks

```csharp
using UnityEngine;
using System.Collections;

public class MovingPlatform : MonoBehaviour {

    public Vector3 displacement = Vector3.zero;
    public float speed = 10.0f;

    private Vector3 startPoint;
    private Vector3 endPoint;
    private float time = 0.0f;

    void Start () {
        // Store the start and end points.
        startPoint = transform.position;
        endPoint = startPoint + displacement;
    }

    /* Here we want to move the platform smoothly between the start and end points. */
    void FixedUpdate () {



    }
}

```

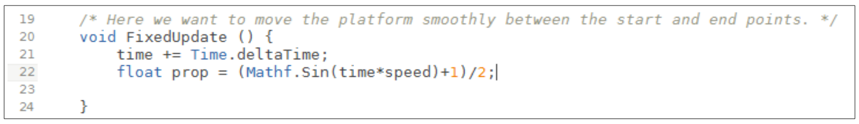
As you can see, the Start function gives our script a “startPoint” and “endPoint”, so all we have to do is move between them over time. The first thing we need to do then, is keep track of time.



```
    /* Here we want to move the platform smoothly between the start and end points. */
    void FixedUpdate () {
        time += Time.deltaTime;


    }
```

The value **Time.deltaTime** is Unity's built-in measure of time. It is the number of seconds since the last frame. If we keep track of the total time, and we have a function for position over time, we can give the platform a position.

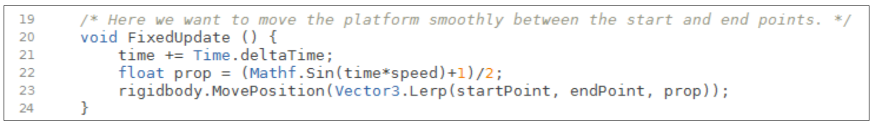
We want a function that will go between a start and end. The easiest way to do this is to get a function that goes from 0 to 1, and multiply it up. To smooth it off, we want something like a sine wave.



```
    /* Here we want to move the platform smoothly between the start and end points. */
    void FixedUpdate () {
        time += Time.deltaTime;
        float prop = (Mathf.Sin(time*speed)+1)/2;

    }
```

This may not be obvious at first, but “prop” above will oscillate smoothly between 0 and 1. Breaking up that line, we can see that *Sin(time*speed)* will oscillate between -1 and 1, because that's what Sin does. Adding 1 to that gives us a range from 0 to 2. All we need do from there is divide by 2.

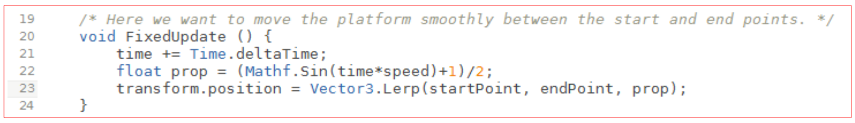
So now we need to use this number to move our platform. Luckily, Unity has a function – **Vector.Lerp** – which will linearly interpolate between two vectors. E.g. give us a vector between two others based on a proportion. At t=0, it will give us the startPoint, at t=1 it will give us the endPoint and any number in between will give us a point in between. Here's how to use it to set the position:



```
    /* Here we want to move the platform smoothly between the start and end points. */
    void FixedUpdate () {
        time += Time.deltaTime;
        float prop = (Mathf.Sin(time*speed)+1)/2;
        rigidbody.MovePosition(Vector3.Lerp(startPoint, endPoint, prop));
    }
```

And so this should now work perfectly!

As a side note, you may be tempted to do something like this instead:



```
    /* Here we want to move the platform smoothly between the start and end points. */
    void FixedUpdate () {
        time += Time.deltaTime;
        float prop = (Mathf.Sin(time*speed)+1)/2;
        transform.position = Vector3.Lerp(startPoint, endPoint, prop);
    }
```

However, this won't work quite how you'd hope. Just setting the position won't move objects that are resting on top of the platform. The function **Rigidbody.MovePosition** moves the object to a position as if it had been done with normal physics, ensuring that anything on top comes with it. You can try out both versions to see the difference when the Player is resting on the platform.

At any rate, once you have a working Moving Platform, save it as a Prefab in case you want to use it later. (You can see how to do this in the previous tutorial section about Pickups).

The Goalposts

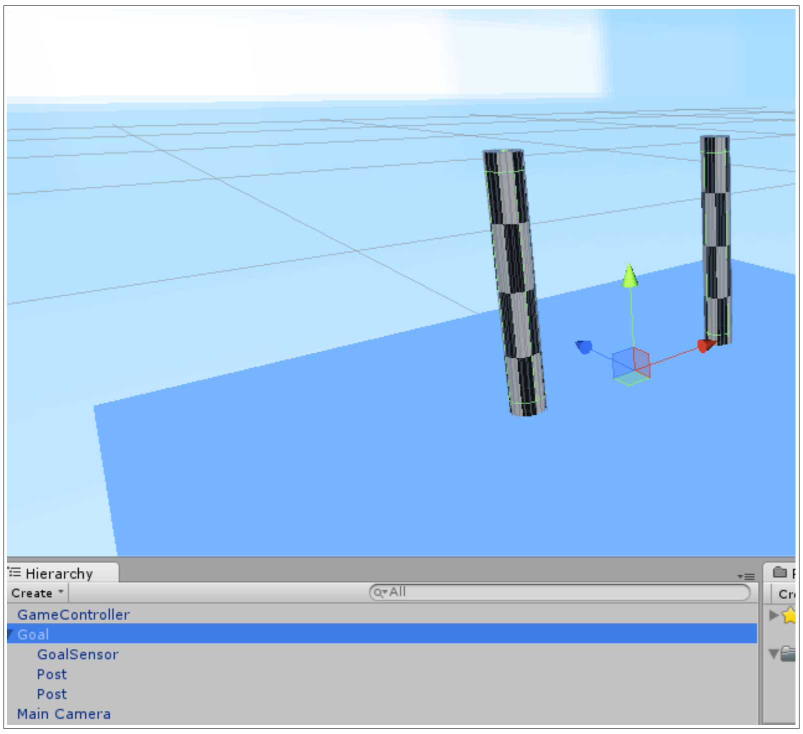
The main thing we're missing from our game now is a victory condition. You'll see those goal posts over at one side of the level, but they don't actually do anything yet. What we're going to do here is add some code to restart the level when the Player passes between them.

Not exactly glamorous, but that can come later.

So how *do* we check if the player has gone between the posts? We could use the CheckSphere method from our Jumping section, or maybe a trigger from the Pickups section. But you don't always want to check for collisions within a certain area, sometimes you want to check if an object has crossed a line.

We can do this with something called Raycasting. It's essentially like firing a beam from one position to another and checking if any object is in the path of that beam. What we'll do here is Raycast from one post to the other and see if the Player is in between.

First things first, look at the Goalposts in the Hierarchy (click the arrow to expand them if you need to).

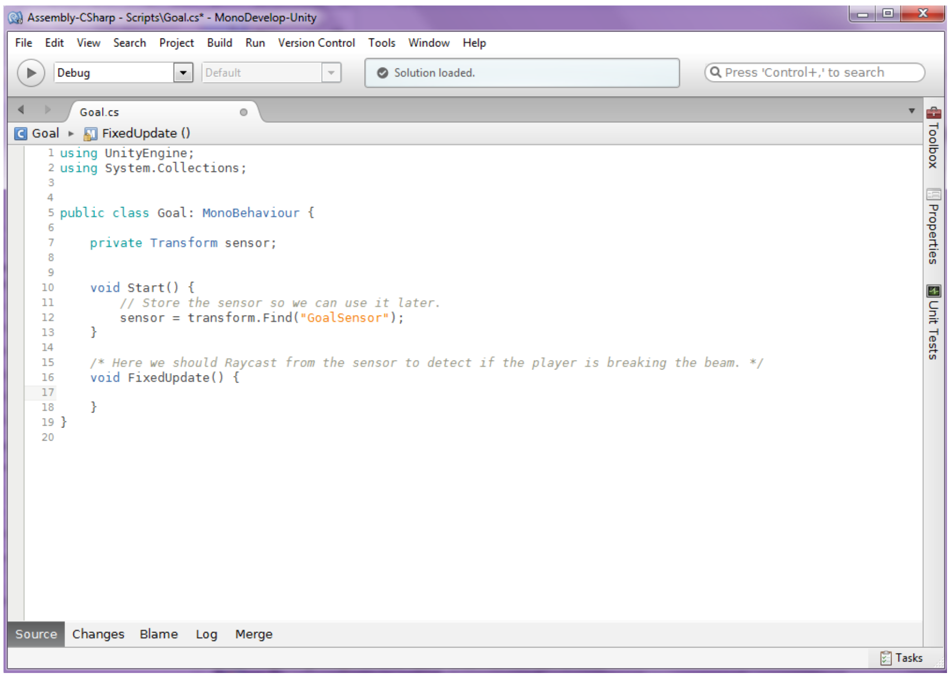


Now we finally see why it's called a hierarchy. This GameObject actually contains three other GameObjects. The Goal contains two Posts and a GoalSensor.

The latter is just an empty GameObject with no components. It's only purpose is to give us a convenient spot for our Raycast to originate from.

Open up the **Goal** script and we'll see what we can do with it.

The Goal Script

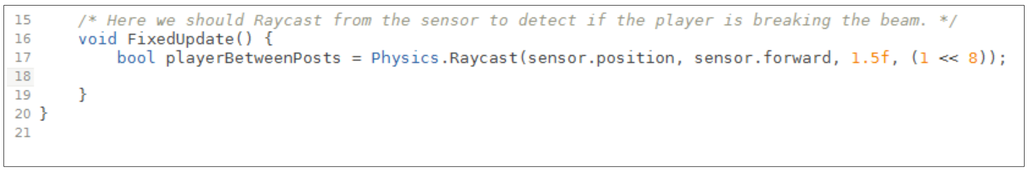

Assembly-CSharp - Scripts\Goal.cs* - MonoDevelop-Unity
File Edit View Search Project Build Run Version Control Tools Window Help
Debug
Default
Solution loaded.
Press 'Control+,' to search
Goal.cs
Goal ▸ FixedUpdate ()
Toolbox
Properties
Unit Tests
Source Changes Blame Log Merge
Tasks

```
using UnityEngine;
using System.Collections;


public class Goal: MonoBehaviour {

    private Transform sensor;

    void Start() {
        // Store the sensor so we can use it later.
        sensor = transform.Find("GoalSensor");
    }

    /* Here we should Raycast from the sensor to detect if the player is breaking the beam. */
    void FixedUpdate() {

    }
}

```

One thing to note about the code already here is the content of the Start function. You'll see we're setting the sensor variable by calling **transform.Find**. The “transform” variable as usual refers to the Transform component of this GameObject. The Transform is what holds the position of the GameObject, but also holds all child GameObjects that this one contains.

The **Find** function will try to get the GameObject with the specified name from inside this Transform. In this example, we're getting the GoalSensor object from inside the Goal object.

Now we need to do a Raycast from the position of the GoalSensor.



```
    /* Here we should Raycast from the sensor to detect if the player is breaking the beam. */
    void FixedUpdate() {
        bool playerBetweenPosts = Physics.Raycast(sensor.position, sensor.forward, 1.5f, (1 << 8));

    }
}

```

The **Physics.Raycast** function is another one you'll be frequently looking up in the documentation. The arguments that we have to pass in to it are as follows:

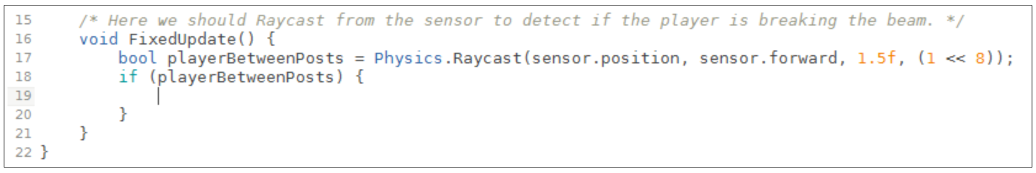
1. The start position of the ray
2. The direction of the ray
3. The distance of the ray
4. The layermask, which filters out some collisions.

The position is just the position of the sensor. The direction is “sensor.forward”, which is exactly what it sounds like: the vector going forwards from the sensor. The distance is 1.5, which is the distance between the posts.

Finally, the layermask is similar to what we did in the **IsGrounded** function from the Jumping section, but inverted. If you've heard about bitmasks, then it may make sense to say each layer has a bit, and the layer mask is a bitmask of those layers.

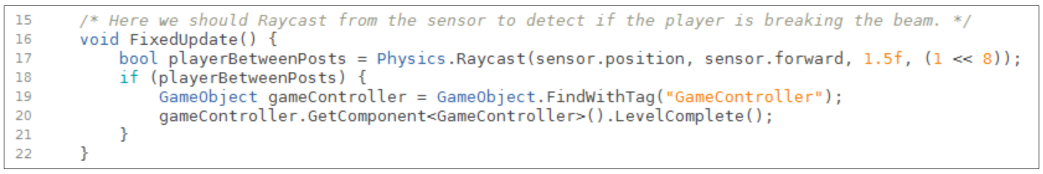
If you don't know what a bitmask is, then just know that Layer 8 is set up to be the Player layer, and the mask (1<<8) is filtering out all objects that are not the Player. This is another way to make sure that we're only checking for the Player.

So this Raycast function returns a **bool**: true if the player was hit by the ray, false if it wasn't. So next, we need to do something if the player was hit:



```
    /* Here we should Raycast from the sensor to detect if the player is breaking the beam. */
    void FixedUpdate() {
        bool playerBetweenPosts = Physics.Raycast(sensor.position, sensor.forward, 1.5f, (1 << 8));
        if (playerBetweenPosts) {

        }
    }
}
```

This should be simple enough. Next, we have to put some code in that “if” block to restart the level.



```
    /* Here we should Raycast from the sensor to detect if the player is breaking the beam. */
    void FixedUpdate() {
        bool playerBetweenPosts = Physics.Raycast(sensor.position, sensor.forward, 1.5f, (1 << 8));
        if (playerBetweenPosts) {
            GameObject gameController = GameObject.FindWithTag("GameController");
            gameController.GetComponent<GameController>().LevelComplete();
        }
    }
```

This may look a bit more complex, but it's not too bad.

The “GameController” is the object which has the function to restart the level. To get that GameObject we use **GameObject.FindWithTag** to find the GameObject that is tagged “GameController”.

Now that we have it, we use **GetComponent** to get the GameController script, and then call the **LevelComplete** function of the GameController script. You can look at that script if you want to see how the function works. It's exactly the same as the GameOver function that is being called when you fall off the edge of a platform.

So, test out those goal posts and see if it all works the way it should.

What to do next?

Hopefully now you have a good grasp of a few basic parts of Unity, but there's really no substitute for experimentation. You've made a few objects, and there are a few Prefabs already in there to work with.

Try expanding the Demo level with more platforms – both static and moving.

Have a look at some of the scripts that weren't covered in this tutorial and see if they make sense to you. Maybe try adding some new ones, or changing how these ones work.

If you are just done with scripting altogether, make a few materials. Paint some textures or make some 3D models and see if you can change how the game looks (I'll be honest, it's not the prettiest the way I left it).

Ultimately, just see what you can do. If you need any help with anything, I'll be there to ask.

If you're reading this at the tutorial itself, I hope to see you at the GameJam.

If you're reading this in the future – I hope you enjoyed the GameJam.

Now go and build some beautiful games.