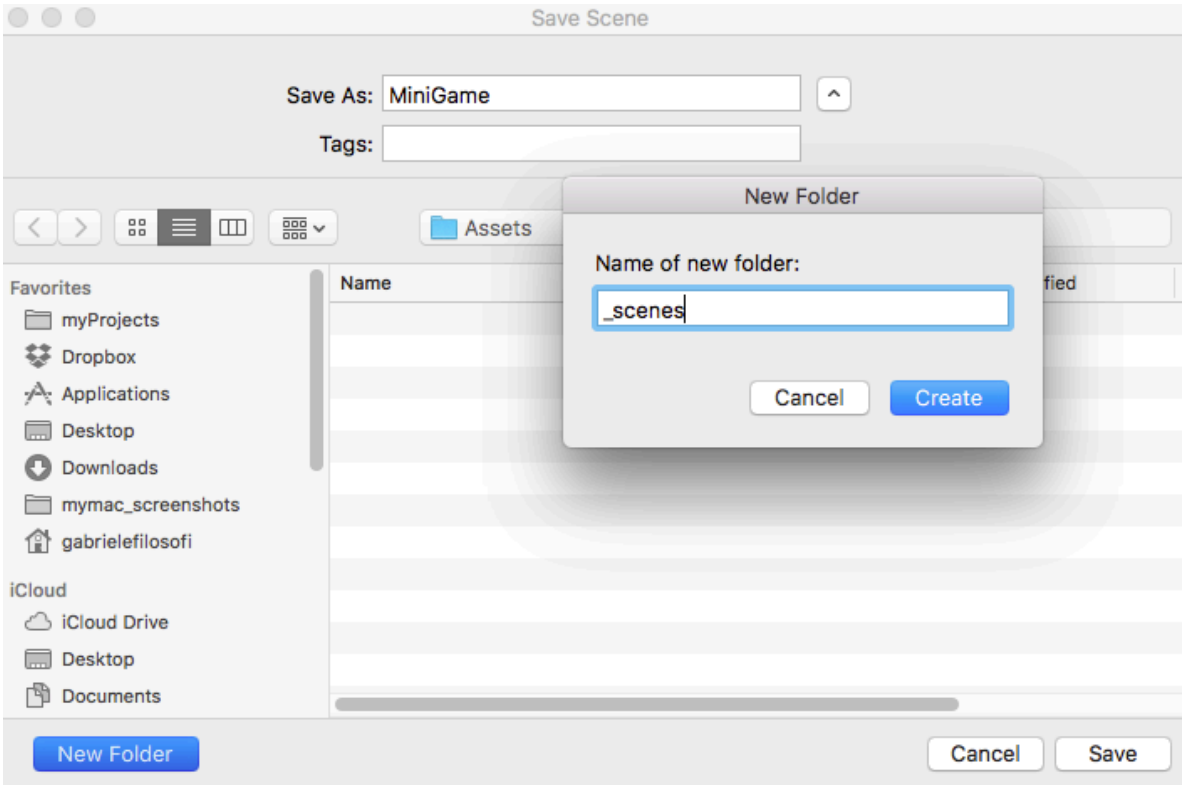
Unity3D - RollerBall 1.0

A rolling sphere hitting objects over the table

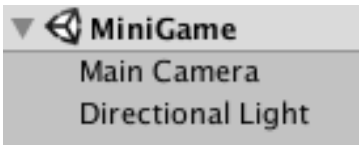
Get Started

- Under Unity 5.x create a 3D project “RollerBall”
- Click *File > Save Scenes*

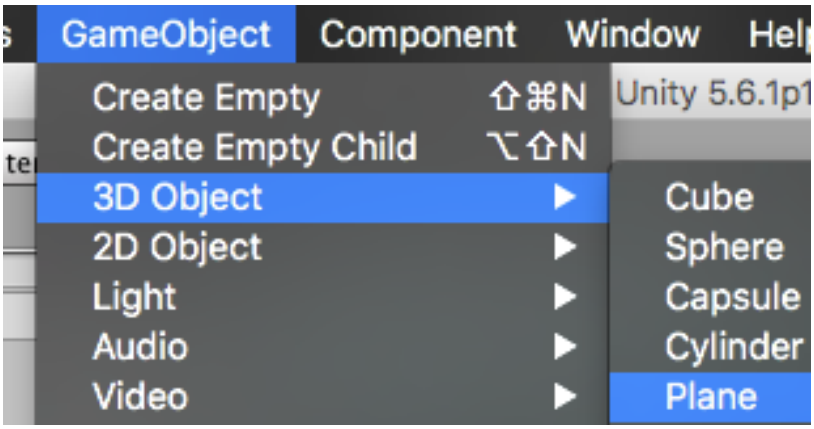
with name “MiniGame” in a new subfolder of *Assets* called *_Scenes*



Eventually you have

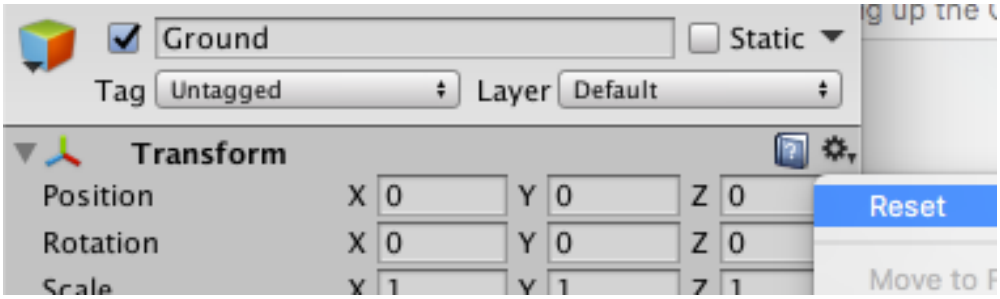


- Let’s create a plane GameObject

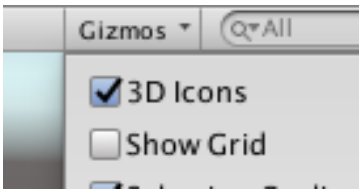


Rename it as “Ground”

- Reset the Ground’s transform to origin. This is standard practice when you get started with a new GameObject



- In order to hide the grid lines of the Scene reference plane, uncheck “Show Grid” under *Gizmos* menu



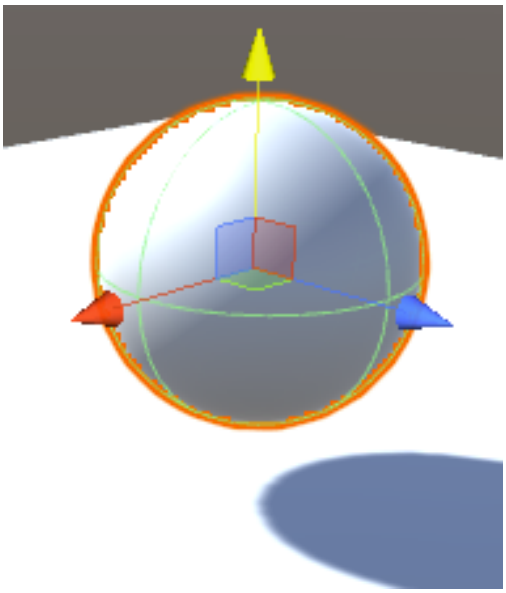
- You can rescale the Ground GameObject in several different ways

For example, select the scale button in the Toolbar

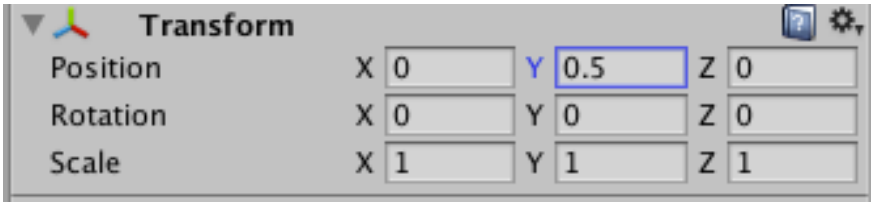
and stretch the object in the Scene view.

- Create a sphere GameObject, call it “Player”, reset its transform, then center Scene view on it with F key, or by clicking *Edit > Frame Selected*
- To make the Player looks like a solid ball on top of the Ground you need to adjust its quote. You can select the Player object, select the translation button in the Toolbar

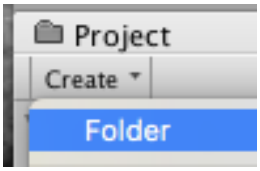
and then move the Player up



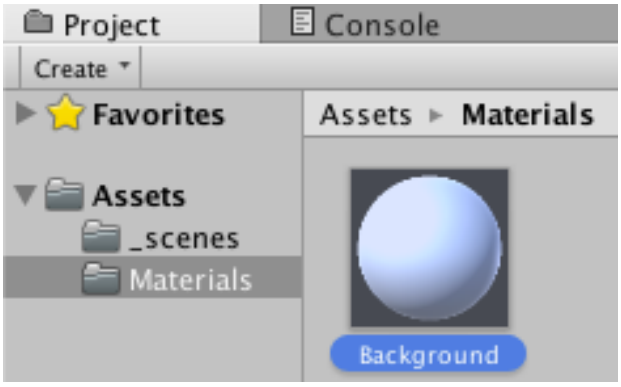
Or you can set the Player's transform position setting this way



- Now we want to add a color to the Ground. To do that we first have to apply a *material* to the object. Let's create a new folder for materials in the Project window



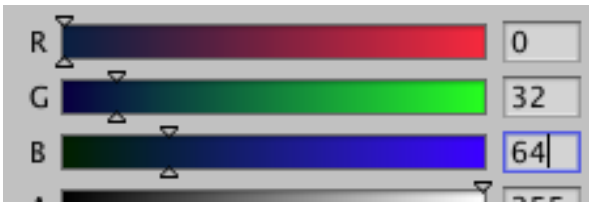
and call it "Materials"
With this folder selected, create a new Material and call it "Background"



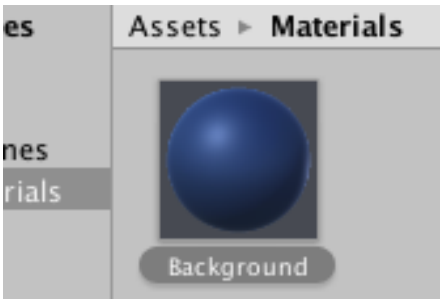
Select the Background material and click the *Albedo* slot under the *Main Maps*

section.

Choose the color

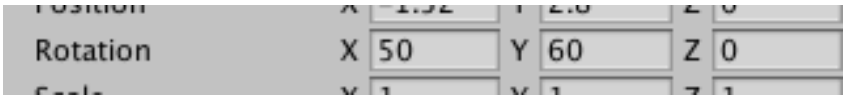


and see the effect.
Then drag the Background material

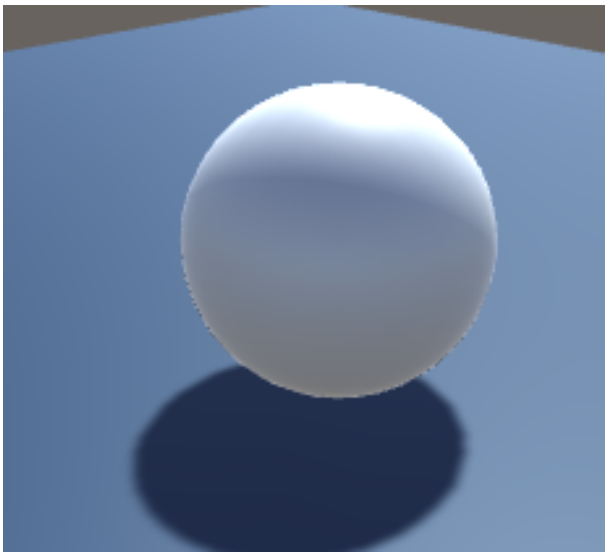


onto the *Ground* in the scene.

- In order to improve the lighting, select the *Directional Light* object and set this transform rotation values



The result is much better

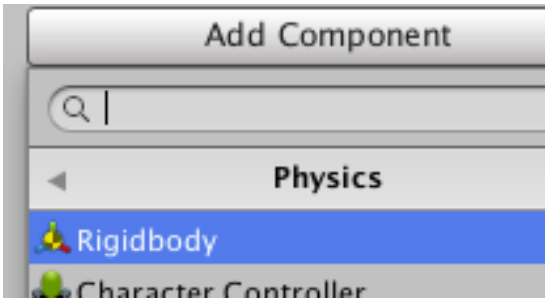


- Click *File > Save Scenes*

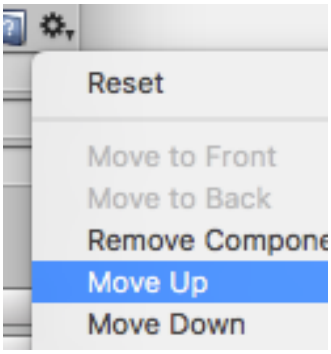
Moving the Player

We want the Player sphere object to be a rigid body rolling on top of the Ground plane object. To do that we need physics.

- Select the Player, and add it a *Rigidbody* component

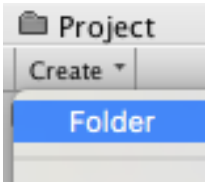


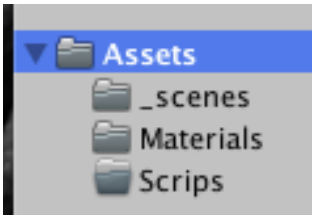
- Note that you can always change the order of the components of a given object, for a better visual order, using *Move Up/Down*



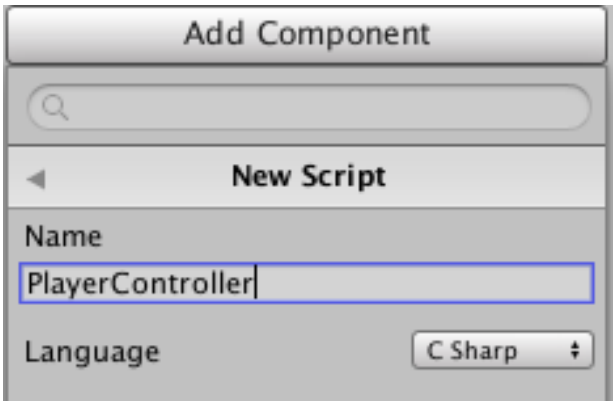
- To apply a force to the Player object we need to add a script component.

First of all, let's create a folder for scripts called "Scripts"

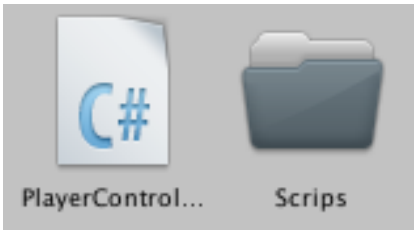




- Select the Player object and add it a *New Script* component. Call it "PlayerController"



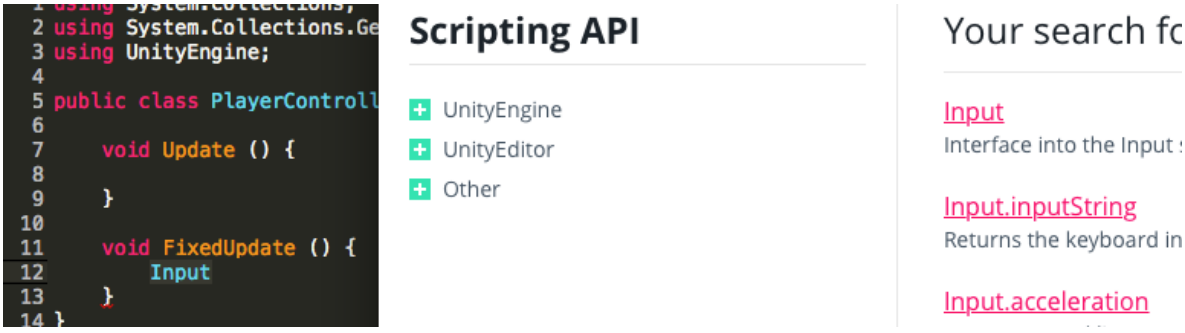
In the Project window drag the script into the Scripts folder



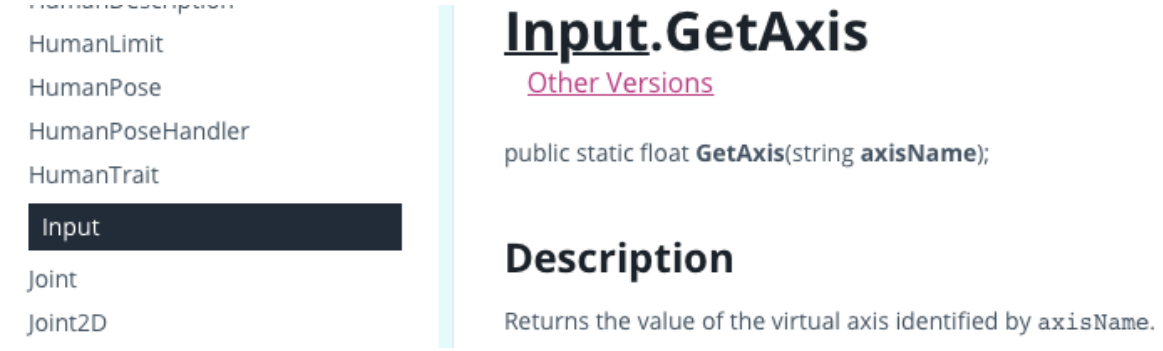
- Double click the C# script just created to edit it in the *MonoDevelop-Unity* editor (you can do that in several different ways)
- Delete the main class content and add two callback functions,
  - *Update*() - it will be called once a new frame is going to be rendered
  - *FixedUpdate*() - it will be called right before any new physics calculation

```
 5 public class PlayerController : MonoBehaviour {
 6
 7     void Update () {
 8
 9     }
10
11     void FixedUpdate () {
12
13     }
```

Let's edit the *FixedUpdate* function.
Type the keyword *Input*, then select it and click *Cmd+'* to open the scripting API reference in Safari



Navigate through the Input method called *GetAxis*



This is method to allow the user to input a force

```
11      void FixedUpdate () {
12          float horizontal = Input.GetAxis ("Horizontal");
13          float vertical = Input.GetAxis ("Vertical");
14      }
```

Type the keyword *Rigidbody*, then select it and click *Cmd+'* . You soon realise this time *AddForce* is the method to be used.
But first create a reference to the Player's *Rigidbody* component using the Start() function, which is a sort of class initialiser

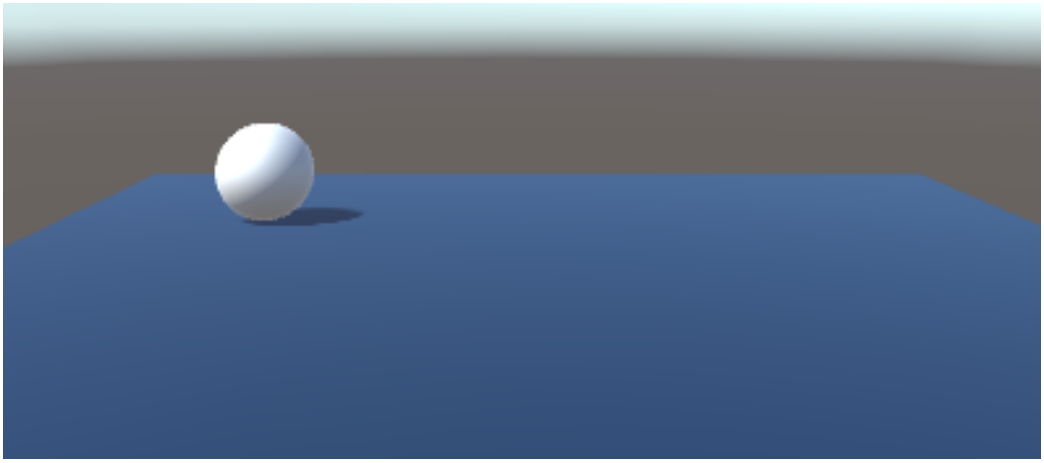
```
 7      private Rigidbody rb;
 8
 9      void Start () {
10          rb = GetComponent<Rigidbody>();
11      }
12
13      void Update () {
14
15      }
16
17      void FixedUpdate () {
18          float horizontal = Input.GetAxis ("Horizontal");
19          float vertical = Input.GetAxis ("Vertical");
20          Vector3 aforce = new Vector3 (horizontal, 0.0f, vertical);
21          rb.AddForce (aforce, ForceMode.Impulse);
22      }
```

Furthermore, let's remove the Update function and add a class property called gain in order to adjust the force easily from the Inspector window

```
 5 public class PlayerController : MonoBehaviour {
 6
 7     public float gain;
 8     private Rigidbody rb;
 9
10     void Start () {
11         rb = GetComponent<Rigidbody>();
12     }
13
14     void FixedUpdate () {
15         float horizontal = Input.GetAxis ("Horizontal");
16         float vertical = Input.GetAxis ("Vertical");
17         Vector3 aforce = new Vector3 (horizontal, 0.0f, vertical);
18         rb.AddForce (aforce * gain, ForceMode.Impulse);
19     }
20 }
```

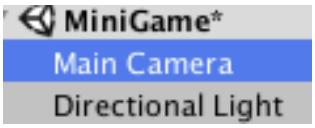
Save and go back to the Editor. Change the Gain

Press the play button and use the arrow keys to apply impulsive forces to the Player sphere !

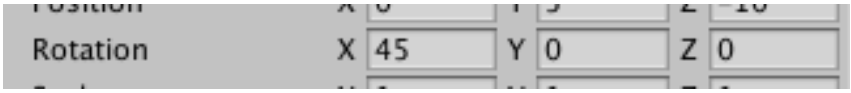


Moving the Camera

Select the Camera object

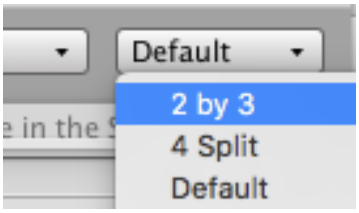


Set the Transform Rotation in the typical way



Drag the Main Camera on top of the Player object, to make it a child

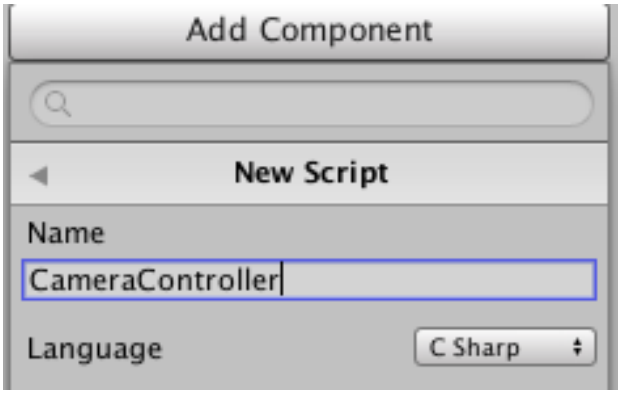
If you run the game you will notice the view rotating with the Player.
In order to see better this concept, select the “2 by 3” perspective of the editor



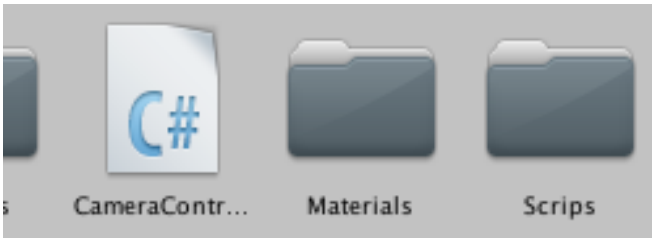
This behaviour is not acceptable. The camera can’t be a child of the Player GameObject. Then detach the Camera from the Player.

The correct way to associate the Camera with the Player is not a parent-child relationship, but a script. Therefore

- Add a C# script component to the Main Camera. Call it “CameraController”



From the Project window drag the script into the Scripts folder



Double click the C# script just created to edit it in the MonoDevelop-Unity editor

- First of all, we need to add a public reference to the Player GameObject and

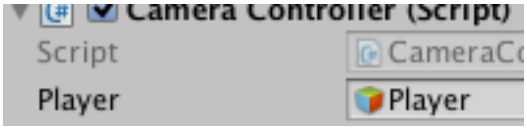
a private vector which represents the position of the Player relative to the Main Camera

```
5 public class CameraController : MonoBehaviour {
6
7     public GameObject player;
8     private Vector3 offset;
```

As the game gets started, we get the offset, the, at each new frame, we adjust the Camera position at that offset relative to the Player. This operation can take place in the Update() callback or, more conveniently in the LateUpdate() callback, just to make sure everything is in place when the camera moves.

```
10    // Use this for initialization
11    void Start () {
12        offset = transform.position - player.transform.position;
13    }
14
15    // Update is called once per frame. LateUpdate is called right after Update
16    void LateUpdate () {
17        transform.position = player.transform.position + offset;
18    }
```

- Save and go back to the Editor.
- Select the Main Camera object and drag the Player object into the empty "Player" slot of the CameraController, just to create a reference

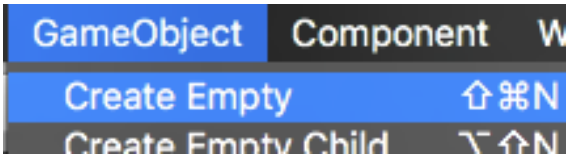


- Press the play button and use the arrow keys to apply impulsive forces to the Player sphere. Now the camera follows the sphere by a constant offset.

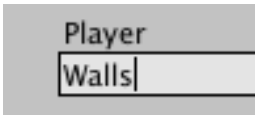
Setting up the Play Area

We want to add walls around the play area to prevent the Player sphere to fall outside.

- Let's create an new empty GameObject



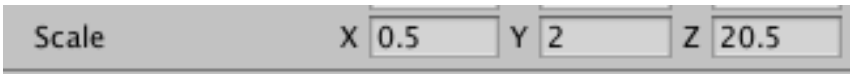
and call it "Walls"



We can you a GameObject as a folder for other GameObjects, if this helps to make hierarchy organisation more clear.

- Reset the Walls transform to origin
- Create a new Cube object

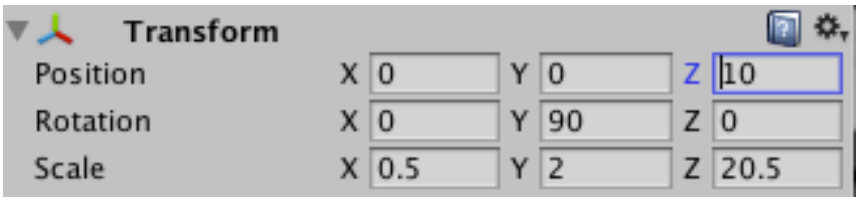
- rename it as “West Wall”, drag it on top of Walls and reset transform
- Press F in order to focus the scene view camera on this object
- Set the transform scale this way



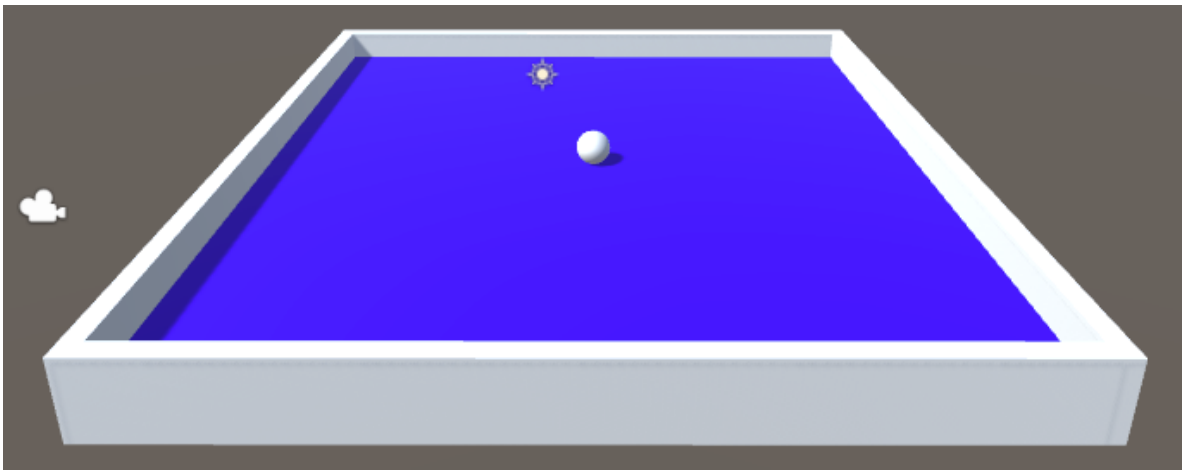
- Set the x position to -10

- Duplicate the West Wall to create the East, North and South Walls.

The North Wall transforms look like this

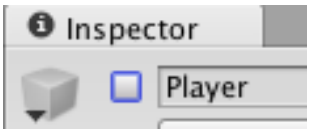


Finally you get something like this

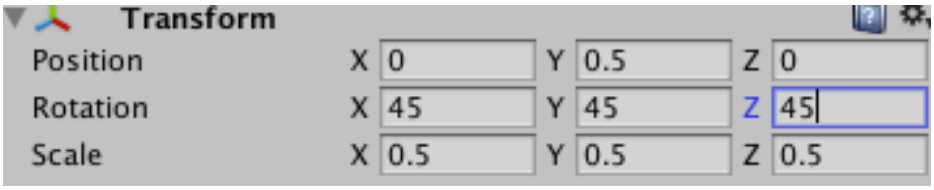


Creating Collectibles

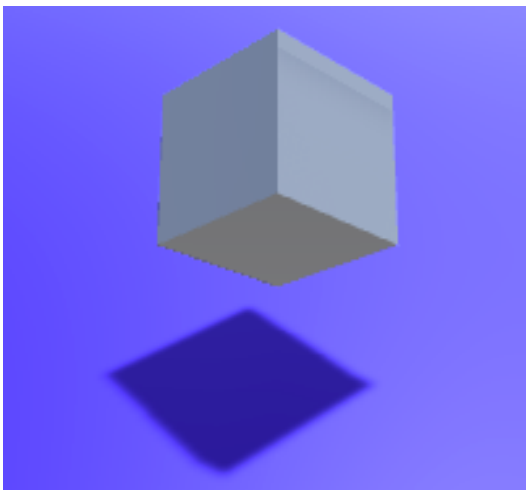
- Create a new Cube object. Rename it “Pick Up” and reset transform to origin
- Press F in order to focus the scene view camera on this object
- Select the Player GameObject and deselect it under Inspector window



Now we can improve the Pick Up object shape.
Set this transform



Now it looks like this



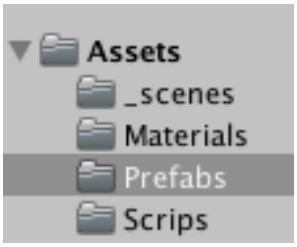
Now we want make the cube spinning.

- Add a C# script component to the Pick Up. Call it “Rotator”

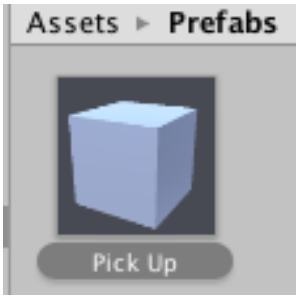
Move the script into the Scripts folder, then open it up in the editor
Write transform, select it and type CMD+’ to open the scripting API reference in Safari. Navigate through the Input method called *Rotate*

```
 7    // Update is called once per frame
 8    void Update () {
 9        transform.Rotate (new Vector3 (15, 30, 45) * Time.deltaTime);
10    }
```

The multiplication by Time.deltaTime allows rotation smoothness and makes rotation speed independent of frame rate.
Save and test it.
Now we want to create multiple instances of the Pick Up object. In order to apply subsequent change to a Pick Up object automatically to all instances, let’s create a Prefabs folder in the Project window

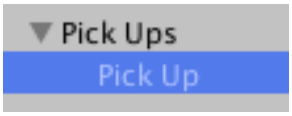


and drag the Pick Up object into this folder

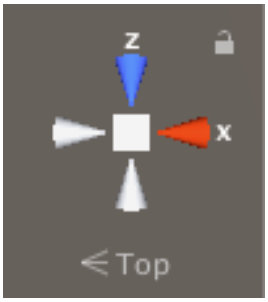


Then create an empty GameObject, all it “Pick Ups”

and drag the Pick Up into it

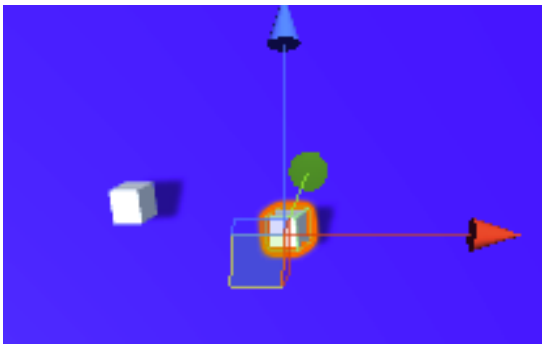


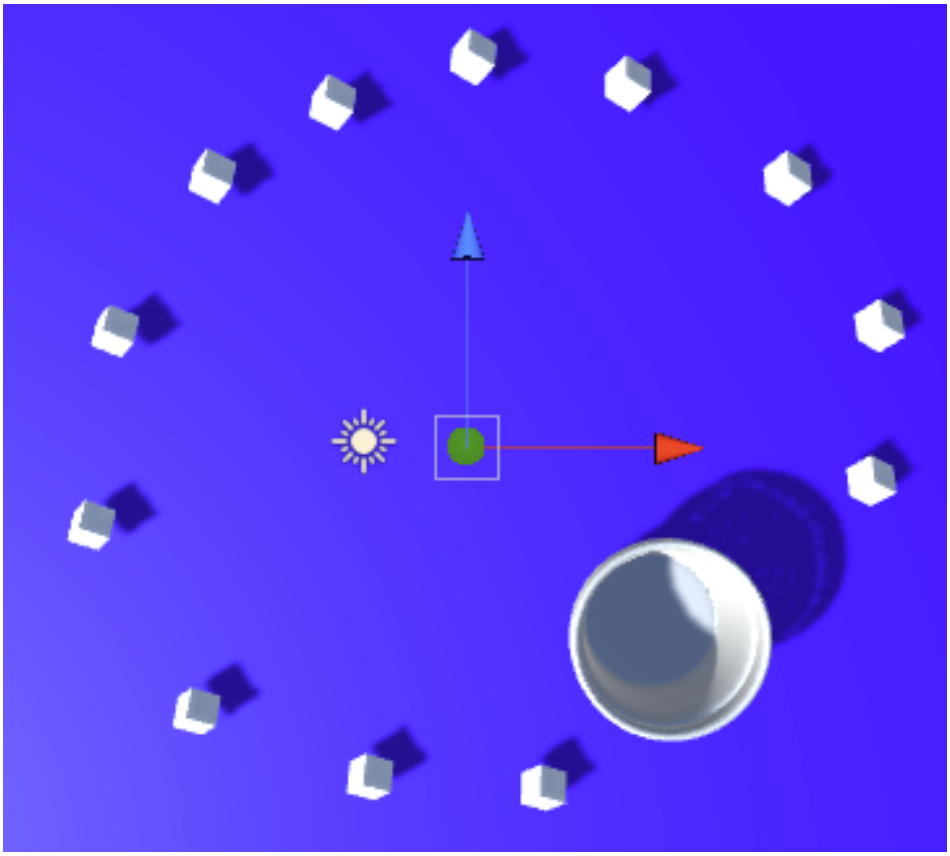
Now we have to create a lot of duplicates (collectibles) of the Pick Up.
Flip the ..



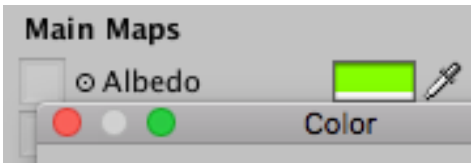
in such a way we have a top view of the scene.
Set to Global reference in order to keep the Pick Up object over the Ground

Type CMD+D to create copies



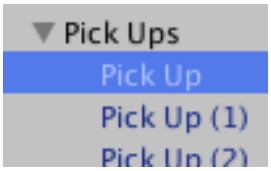


Test and save the scene.
Now we want apply a color to all Pick Up collectibles.
Let’s create a new material in the Materials folder. Call it “Pick Up” and set its Albedo property

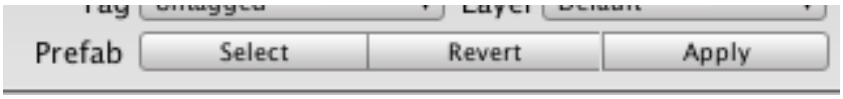


Drag the material on top of any one of the Pick Up objects

Select the object under Hierarchy window



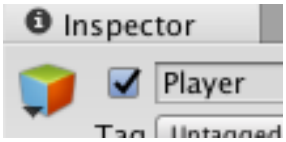
and click Apply button in the Prefab section of the Inspector window



The last operation will propagate the same material to all collectibles

Trigger object collisions

We want to deactivate a Pick Up GameObject when hit by the Player.
First of all, bring back the Player into the scene.



Click the question mark in the Player's collider component

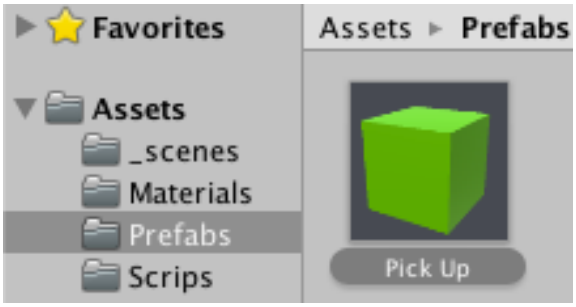
A component's reference opens up. Click the Switch To Scripting tab

The useful callback is OnTriggerEnter which occurs on the FixedUpdate after the Player object experiences a collision by a trigger Collider which is passed as "other" parameter.
other.GameObject is the colliding object. By looking at the GameObject API reference we find out that the property tag can help to identify the game, and setActive can be used to deactivate the game object.

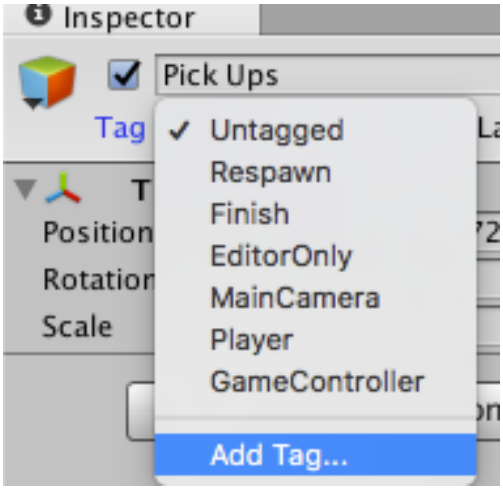
When comparing the tag, use *CompareTag* method which is more efficient than a simple equal test.

```
21      void OnTriggerEnter(Collider other) {
22          if (other.gameObject.CompareTag("Pick Up")) {
23              other.gameObject.SetActive(false);
24          }
25      }
```

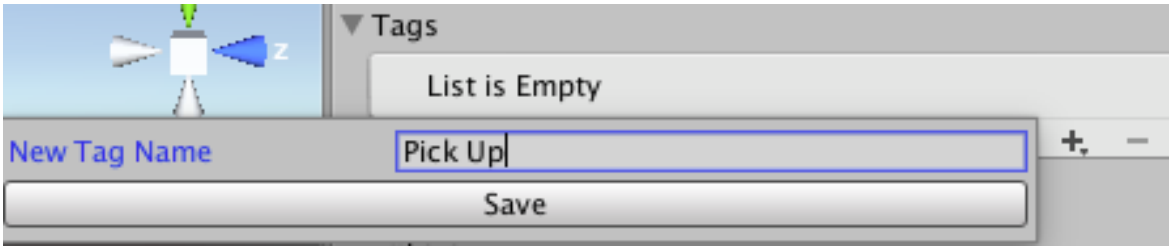
Save.
Now we have to assign the tag “Pick Up” to all Pick UP object.
To do that, select the Pick Ups Prefab



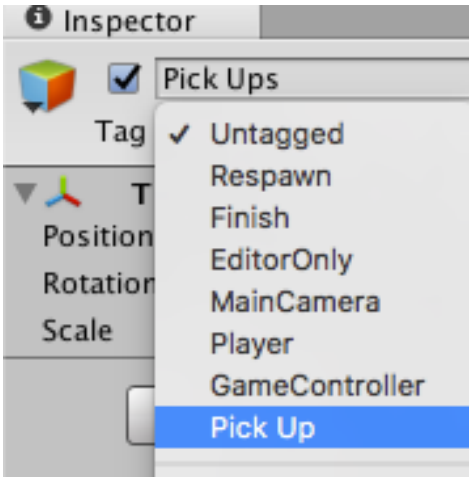
and create a new tag



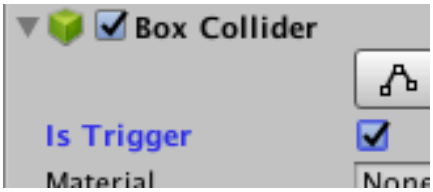
Take care of naming this tag exactly the same of the tag you used in the *CompareTag* function. Then Save it.



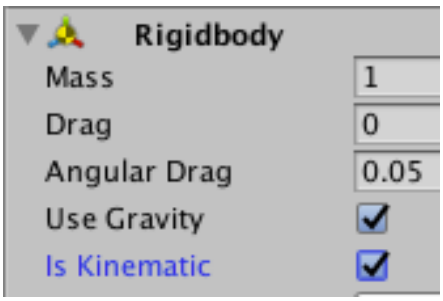
Now you can apply the tag



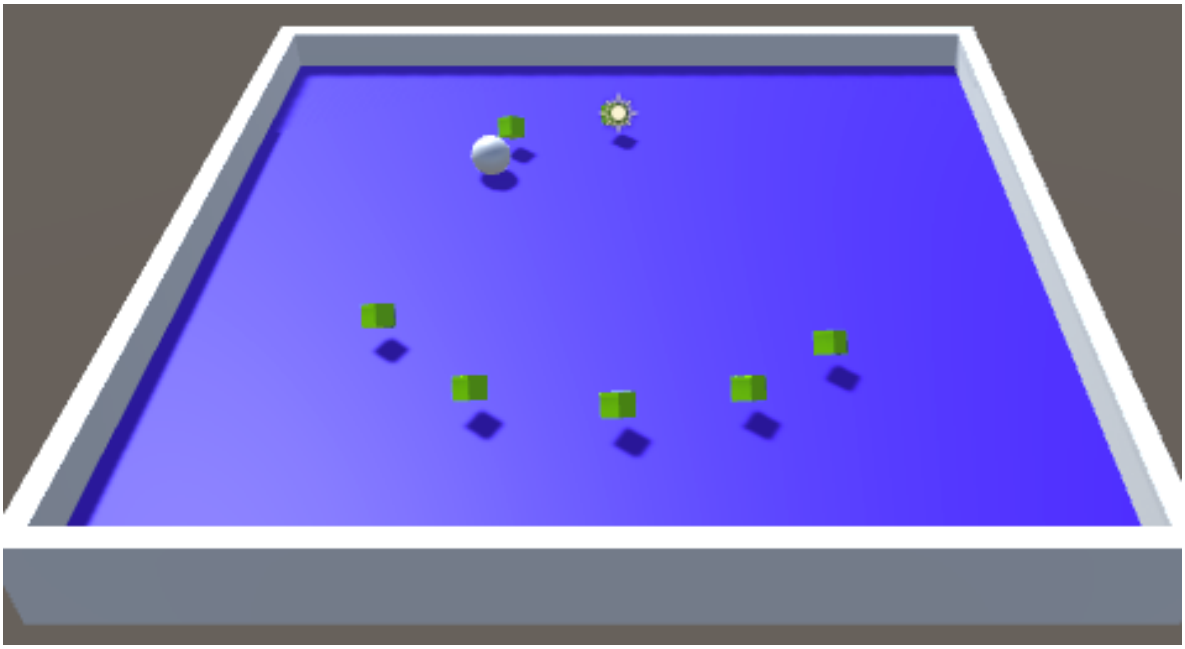
Now we need to enable the Trigger of the Box Collider



Furthermore we need to acc a Rigidbody component to the Pick Up Prefab. Check “Is Kinematic”



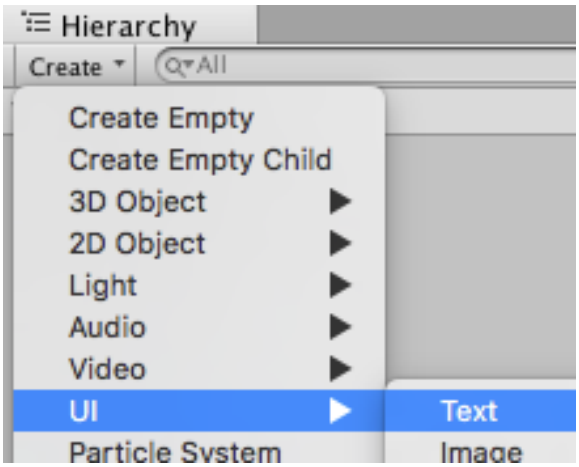
Test and save the scene.



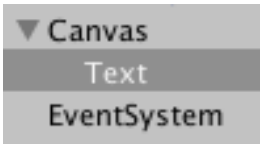
Add UI elements

we want to count in real time the Pick Up objects hit by the Player, and visualise it on the screen.

- Open the PlayerController script, add a private int “count”, reset to 0 it in the Start() function and increment it every time a new collision to a Pick Up object is triggered.
- In the Hierarchy window create a UI Text component



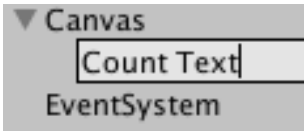
This operation created also a parent object Canvas and another object EventSystem.



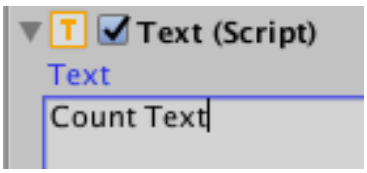
Note that all UI elements must be attached to a Canvas object.

- Let’s customise the Text element.

Call it *Count Text*



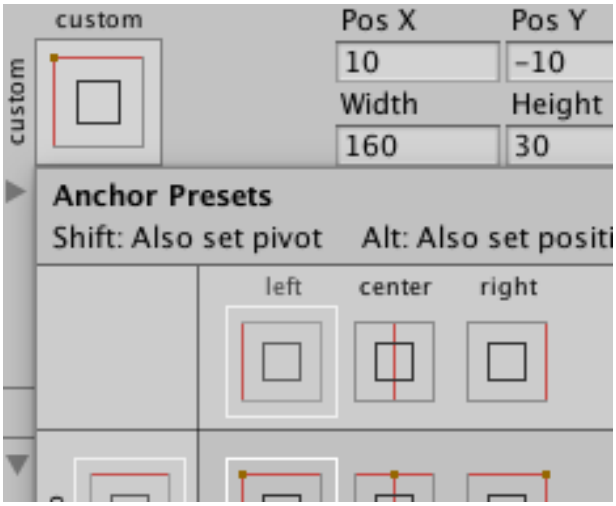
Set also the text value to “Count Text”



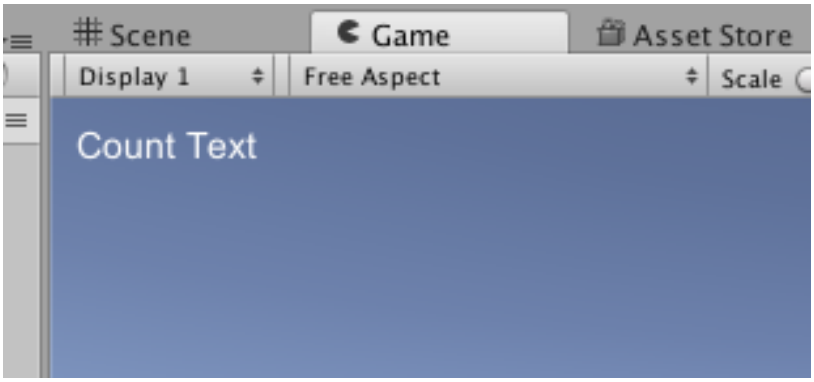
Set the color white

Anchor it to the top-left corner of the screen, with some X and Y safe margins

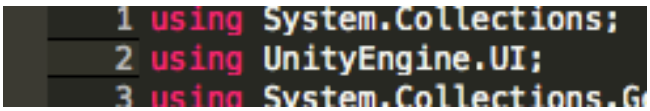
When you do it please hold Option and Shift keys



In the Game window the Text element will look like this



Now bring back the PlayerController script, where we need to do something to display the counter information in the UI.
The details of the UI toolset is enclosed in the *UnityEngine.UI* namespace. Let's add it



and add some code as follows

```
 6 public class PlayerController : MonoBehaviour {
 7
 8     public float gain;
 9     public Text countText;
10
11     private Rigidbody rb;
12     private int count;
13
14     void Start () {
15         rb = GetComponent<Rigidbody>();
16         count = 0;
17         SetCountText ();
18     }
19
20     void FixedUpdate () {
21         float horizontal = Input.GetAxis ("Horizontal");
22         float vertical = Input.GetAxis ("Vertical");
23         Vector3 aforce = new Vector3 (horizontal, 0.0f, vertical);
24         rb.AddForce (aforce * gain, ForceMode.Impulse);
25     }
26
27     void OnTriggerEnter (Collider other)    {
28         if (other.gameObject.CompareTag("Pick Up")) {
29             other.gameObject.SetActive(false);
30             count++;
31             SetCountText ();
32         }
33     }
34
35     void SetCountText () {
36         countText.text = "Count: " + count.ToString();
37     }
38 }
```

Save and go back to the Unity editor.
Select the Player GameObject and drag the Count Text object on top of its Count Text empty slot

Deployment

From *File > Build Settings* switch the project to the desired platform, the click *Build*. The first time you perform this operation, create a new folder called Builds where to store the current and all future builds.

iOS target

In mobile platforms like iOS we can take advantages of the embedded accelerometer.
If you hold the device upright (with the home button at the bottom) in front of you, the X axis is positive along the right, the Y axis is positive directly up, and the Z axis is positive pointing toward you.
Taking into account also the device orientation, the PlayerController script will be changed this way

```
22      void FixedUpdate () {
23          float horizontal;
24          float vertical;
25
26          //for mobile platforms
27          if (Input.deviceOrientation == DeviceOrientation.LandscapeLeft) {
28              horizontal = -Input.acceleration.x;
29              vertical = -Input.acceleration.y;
30          } else {
31              horizontal = Input.acceleration.x;
32              vertical = Input.acceleration.y;
33          }
34
35          Vector3 aforce = new Vector3 (horizontal, 0.0f, vertical);
36          rb.AddForce (aforce * gain, ForceMode.Impulse);
37      }
```