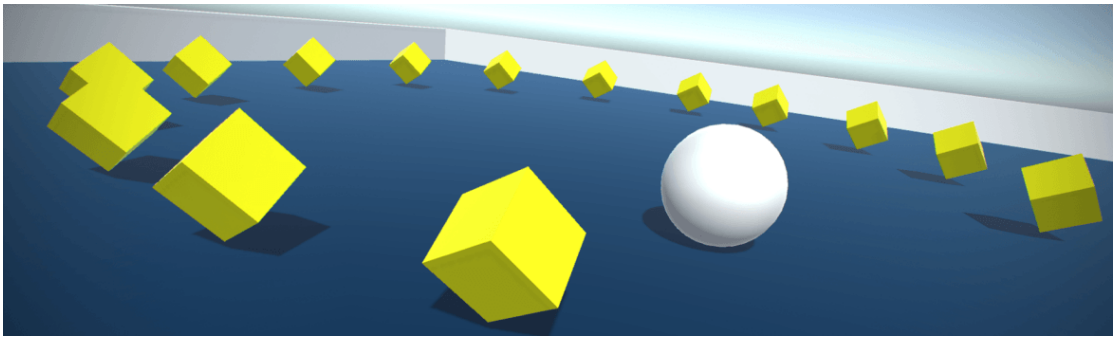
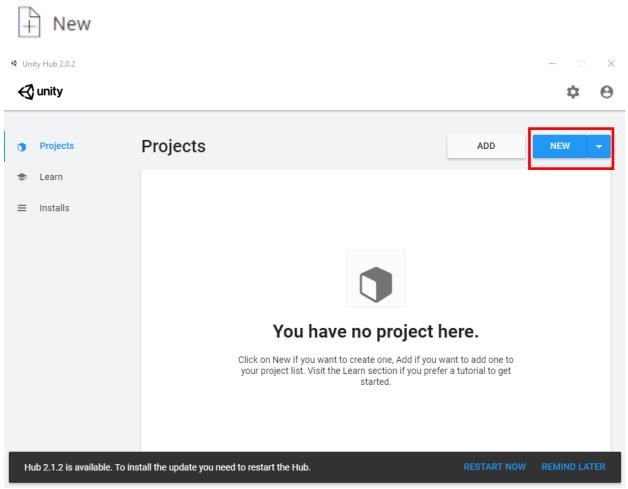
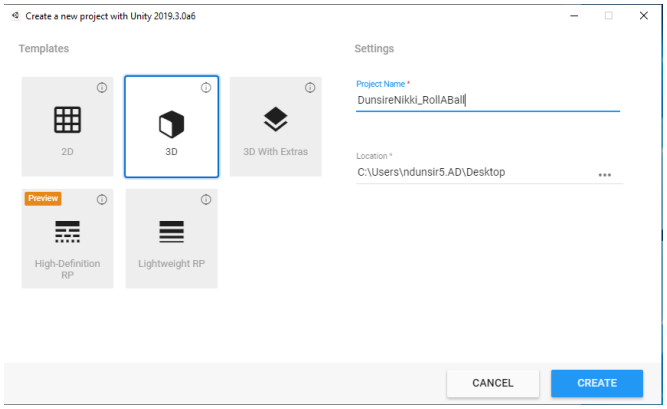
Introduction to Unity: Roll-A-Ball Basic Interactivity



1) Starting with the Unity Launcher:
   a. Creating a new project: New →

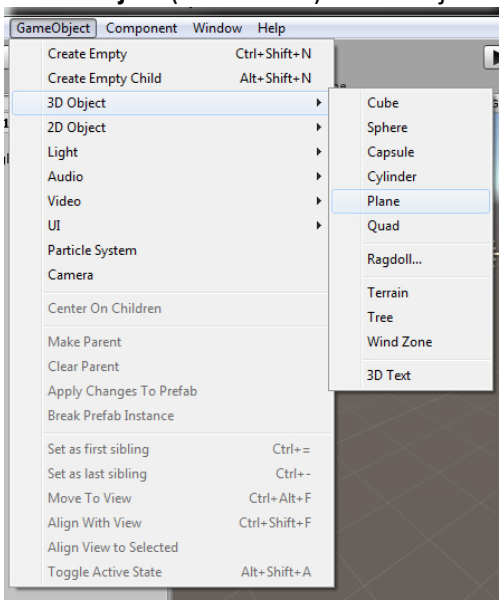


   i. **Project name**:
      - Note – *Name it LastNameFirstName_RollABall*
   ii. **Location**: click on the dots to navigate where you want your project to live
   iii. **3D vs. 2D**
      - Pick 3D for now
   iv. Hit **Create Project**



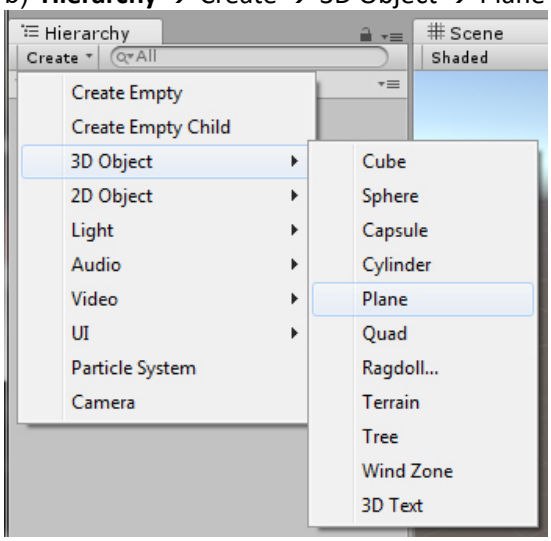
2) Once Unity is open:
   b) File → Save scene as →
      i) Double click on the folder called it 'Scenes' to open it up
      ii) Type a name for your scene file: "01_RollABall"
      iii) Hit Save
   c) Delete the "SampleScene"

3) Create a primitive plane. We can do this in <u>two ways</u>:
   b) **Game Object** (main menu) → 3D Object → Plane

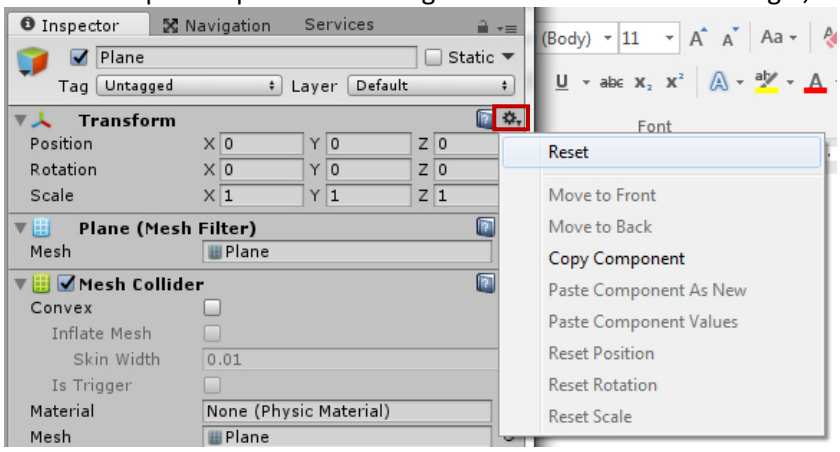


OR

   b) **Hierarchy** → Create → 3D Object → Plane



   c) This should place a plane at the Origin for us. If it's not at the origin, reset its value in the Inspector



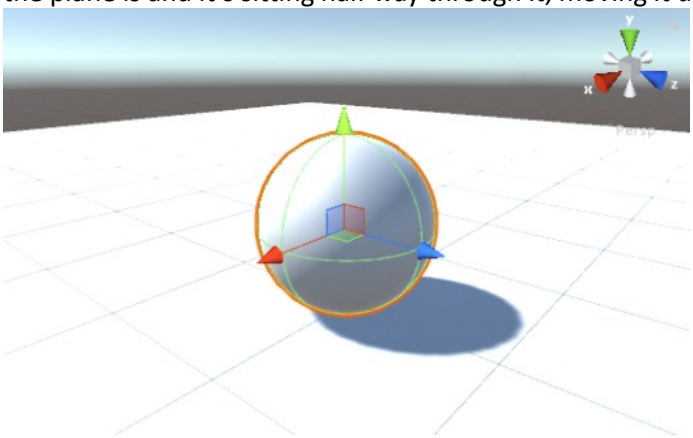
   d) Rename the plane to '**Ground**'

   e) Scale the plane up to 2, 1, 2

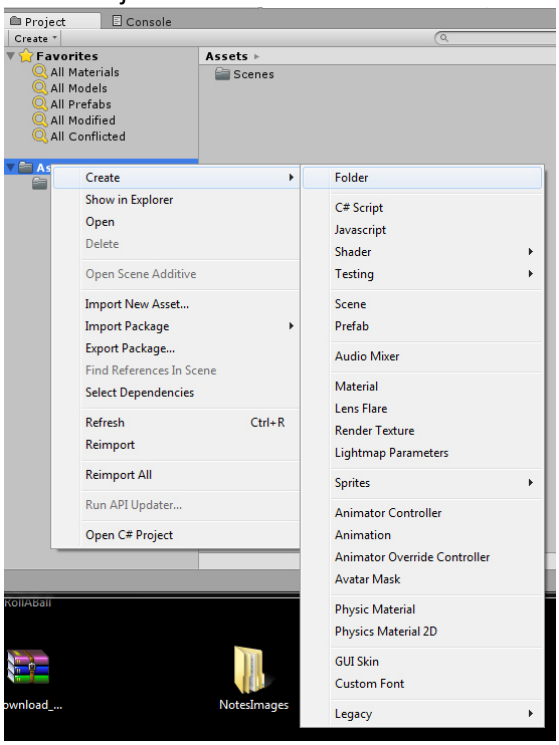
      i) <u>Note</u> – you can't scale up in Y b/c this object is single sided. However, if you gave it a negative number, it would flip the plane the other direction and you wouldn't see it unless you rotated your camera underneath it. Making the non-renderable side transparent is called Backface Culling.

4) Create a Sphere (in the same why you created the plane)
   b) Name it 'Player'
   c) Make sure it's at the Origin
   d) Move it up in Y: 0.5

   i) Note – all objects in Unity take have default size of 1, 1, 1 or 1,2,1. If the Sphere is sitting at the Origin like the plane is and it's sitting half way through it, moving it up by half will rest it perfectly on top.

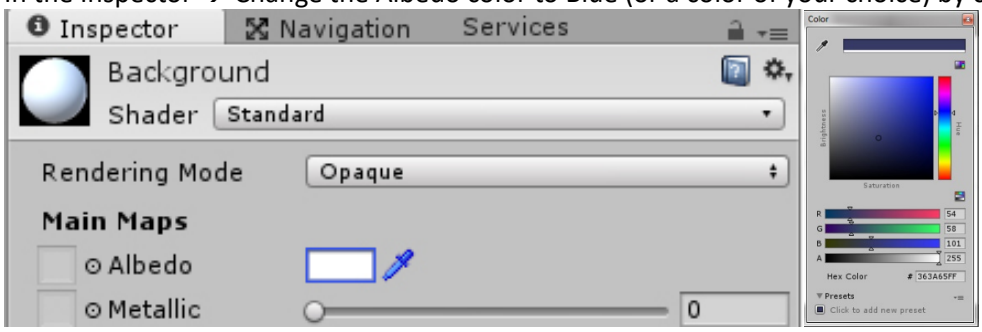


5) Let's add a material to the Ground
   b) In the Project Tab → Rt mouse click over the Assets folder → Create → Folder

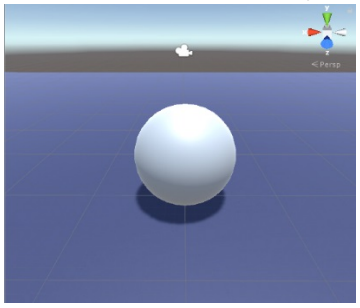


   c) Rename this folder 'Materials'
   d) Highlight the Materials folder → Rt mouse click → Create → Material
      i) Name the new Material: 'Ground_mat'
      ii) Note – we should now see a new Material node in the Materials folder

e) In the Inspector → Change the Albedo color to Blue (or a color of your choice) by clicking on the swatch



f) From the Materials folder, drag the Ground material to the plane in the scene.

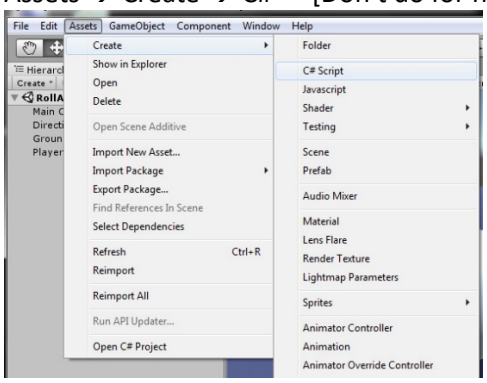


Player Control

1) Let's now setup the player behavior:
   a) Setting up **Contact**:
      i) Select the Sphere
      ii) In the Inspector → Add Component
         ⇒ Physics → Rigidbody
         ⇒ Note – this will allow the Sphere to behavior properly when it bumps into things…i.e. it won't move through other objects.
      iii) Note – Because we created the primitive sphere in Unity, notice that it has a Sphere Collider already applied to it. This matters when we're working with objects that will interact with other objects…which we'll discuss more later. For now, make a mental note of it.

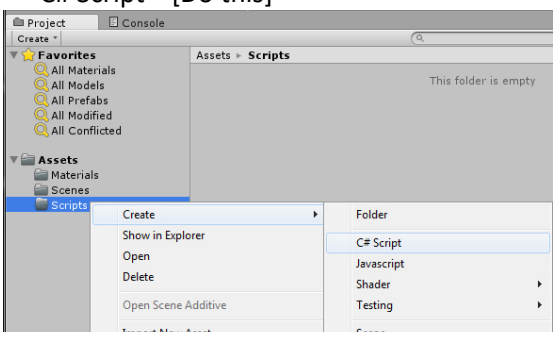
   b) Setting up **Movement**:
      i) Note – to setup movement via our keyboard that drives the sphere we'll need to use a script.

      ii) Create a new folder within Assets called 'Scripts'
         ⇒ To create a new script, we can do this in one of three ways:
            - Assets → Create → C# [Don't do for now]



OR

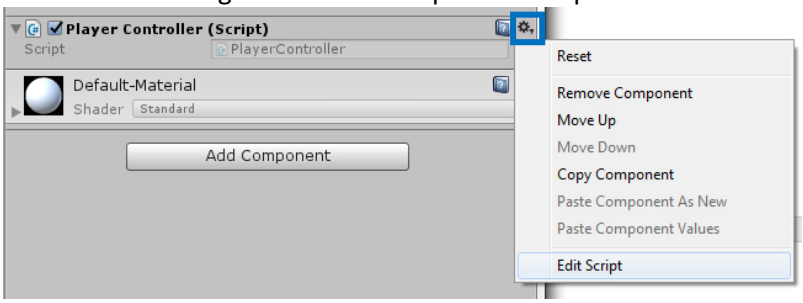
- Project Tab → Create (Over the Scripts Folder) → C# Script [Do this]



- Rename the Script to 'PlayerController'
- Drag the script onto the Sphere in the Scene
- Open up the Script. We can do this in a couple of different ways:

(i) Double click on it in the Scripts folder **OR**

(ii) Click on the Settings Gear of the script in the Inspector once it's added to the sphere



(iii) Note – this should open up Visual Studio which is where we're going to edit the script. If you're on a Mac, it might open MonoDevelop. Either is fine. Visual Studio won't compile on a Mac OS, as its PC based.

1. For now, we are not going into the ins and outs of the code, we simply want to explore how this process works and how it's applied.

(iv) In the Code Editor:

1. Remove the text that's in the document and replace it with:

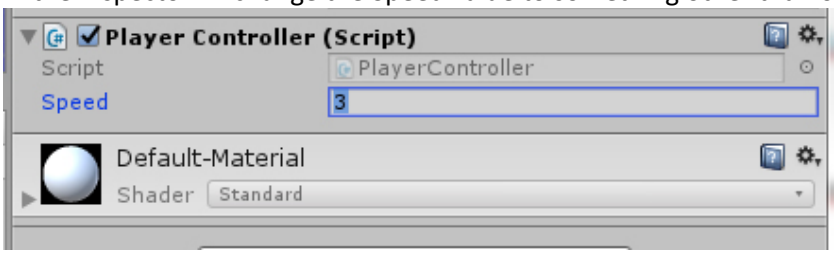
```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class PlayerController : MonoBehaviour
{


    public float speed;

    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void FixedUpdate()
    {
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

        rb.AddForce(movement * speed);
    }

}
```

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class PlayerController : MonoBehaviour
{

    public float speed;

    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void FixedUpdate()
    {
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

        rb.AddForce(movement * speed);
    }

}
```

2. Save the code file

- Play test!

(i) In the Inspector → change the Speed value to something other than 0

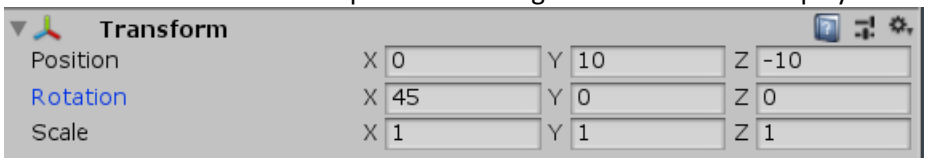


(ii) Hit Play on the player controls

(iii) Use W,A,S,D to control the movement of the Sphere

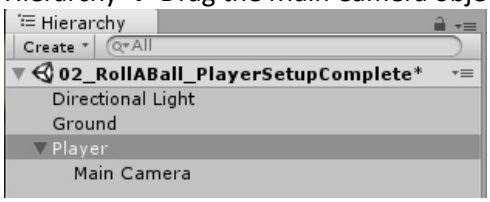
Camera Setup

1) Position the Camera so that it's up above and angled down toward the player



Note – how we setup the camera in relationship to the player will matter. We want them pointing forward down the same axis. So, if Z is the forward axis of the camera, we want Z to be forward on the sphere as well. Otherwise, when we apply our script, our controls may end up backwards.

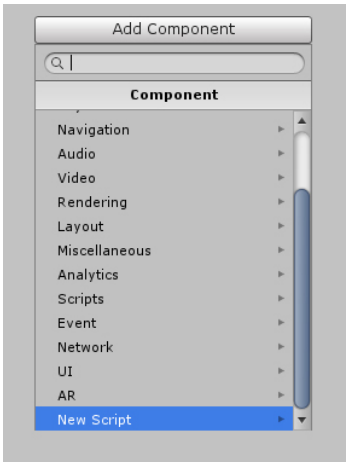
2) We want the camera to move relative to the Player. One thing we could do (but won't) is:
   a) Hierarchy → Drag the Main Camera object over the Player Object to Parent them.

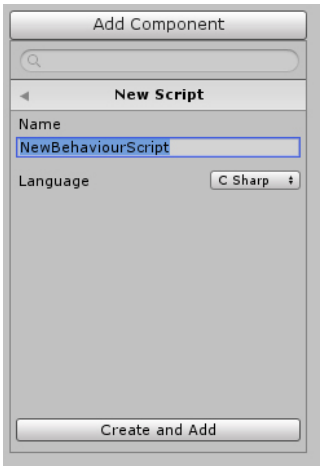


   i) Note – So right now the camera will follow the object as it moves. This is a typical 3rd person setup. The only problem with this however, is that as soon as the player starts rotating (which it's going to do on all three axis) the camera will do the same. So, we can't do it this way, we'll need to use a script.

   ⇒ Drag the Main Camera back out of the Player in the Hierarchy to unparent them.

   ii) Select the camera
   ⇒ Inspector →
   - Add Component → **New Script** →





   - Rename the script 'CameraController'
   - Hit **CreateAndAdd**

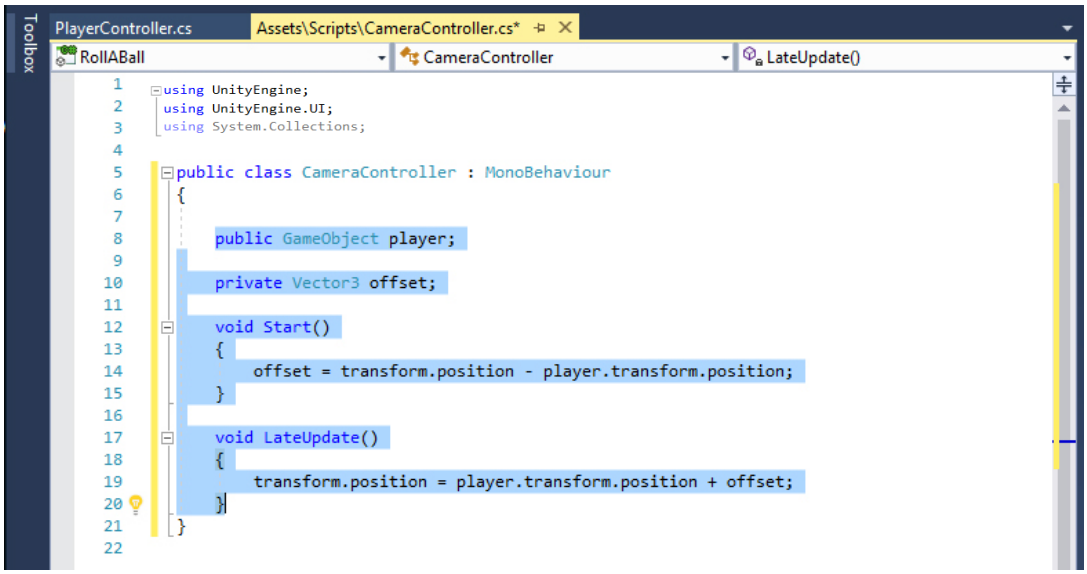
   ⇒ Find the script in the Assets folder and move it into the Scripts folder

⇒ Open the Script Up in Visual Studio

- Remove the text that's in:
- Replace it with:

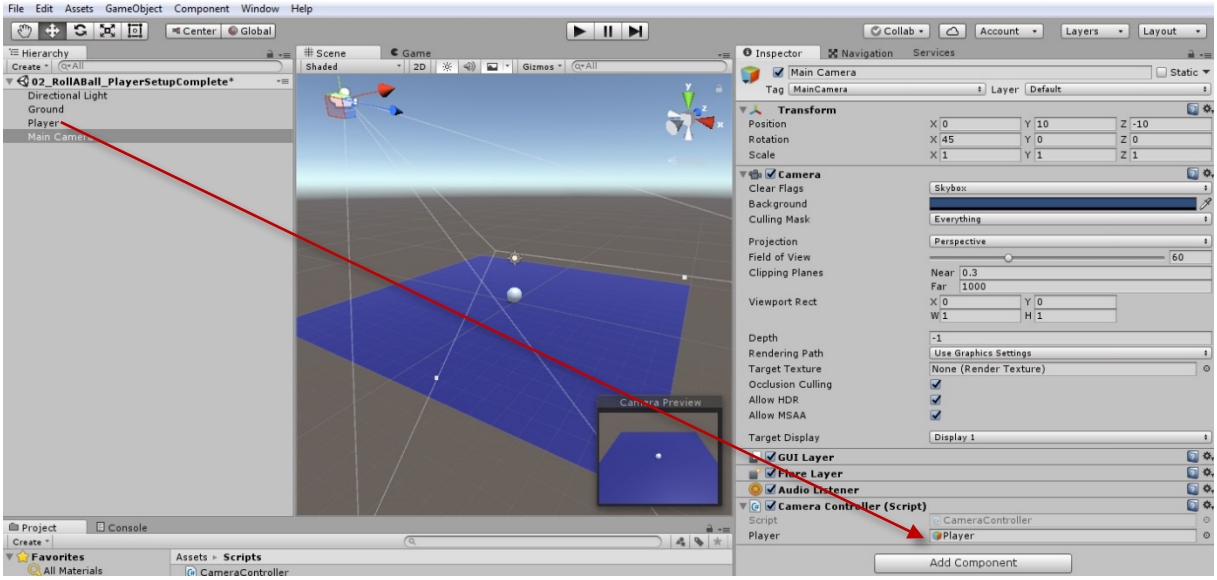
```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class CameraController : MonoBehaviour
{
    public GameObject player;
    private Vector3 offset;
    void Start()
    {
        offset = transform.position - player.transform.position;
    }
    void LateUpdate()
    {
        transform.position = player.transform.position + offset;
    }

}
```

It will look like this:



- Save the script and go back to Unity

iii) Now we need to connect the Player to the Camera so that our Script will work:
⇒ Select the Main Camera
⇒ Go to the Inspector
⇒ In the Camera Controller (Script) drop down → Drag the Player object from the Hierarchy to the Player slot of the Script to make the connection.

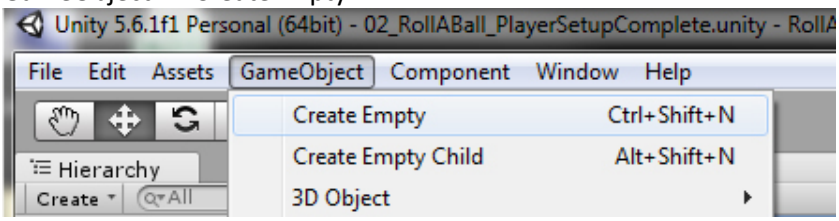


⇒ Play test!

iv) Note – for added functionality, try using the camera scripts that we used in our third person setup of the basic unity scene, instead of the one above!

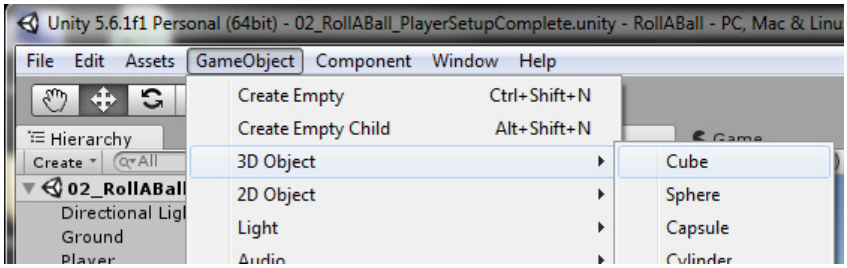
Play Area Setup

1) Create an Empty Game Object
   a) GameObject → CreateEmpty



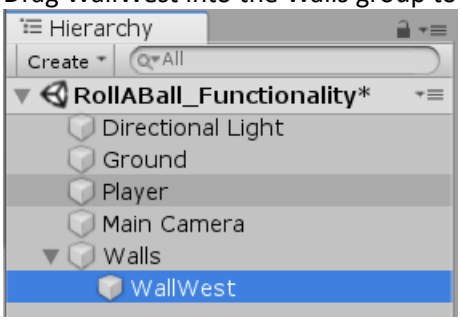
   b) Note – we can use Empty Game Objects to organize and group things in the Hierarchy.
   c) Name this GameObject 'Walls'
   d) Verify that is sitting at the Origin. If it isn't, reset it so that it is.

   e) Create a Cube

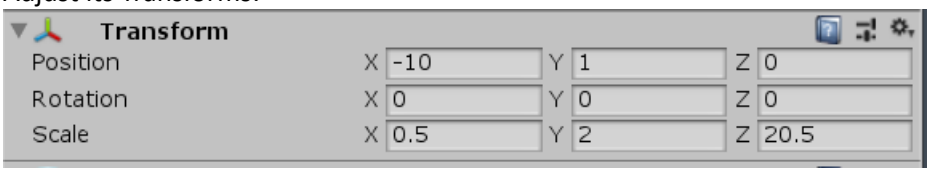


   f) Rename it 'WallWest'
   g) Verify that it is also at the Origin. If it isn't, reset it so that it is.

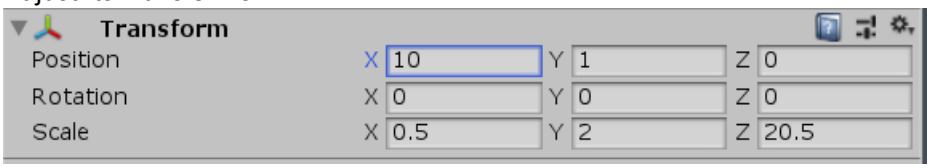
h) In the Hierarchy:
   i) Drag WallWest into the Walls group to parent the cube to the empty game object we created.



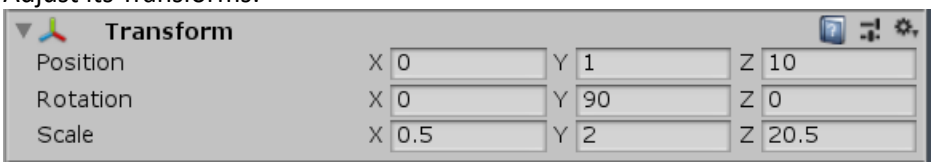
   ii) Adjust its Transforms:



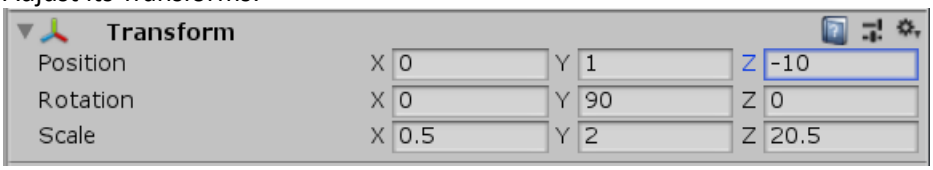
i) Duplicate the wall by hitting **Cntrl d**
   i) Rename it to WallEast
   ii) Adjust its Transforms:



j) Duplicate the wall
   i) Rename it to WallNorth
   ii) Adjust its Transforms:



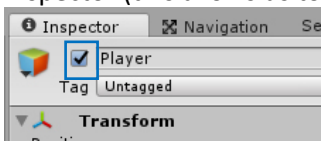
k) Duplicate the wall
   i) Rename it WallSouth
   ii) Adjust its Transforms:



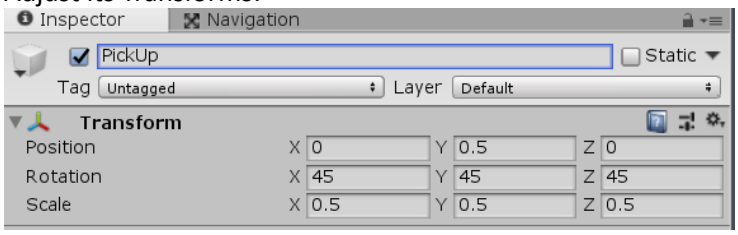
l) Play Test!

Collectables Setup

1) Create a new cube
   a) Reset it to the Origin if it isn't already there.
   b) Note – we can hide the Player for now if we want by clicking on checkbox next to the name of the object in the Inspector (this allows us to make an object active or not).



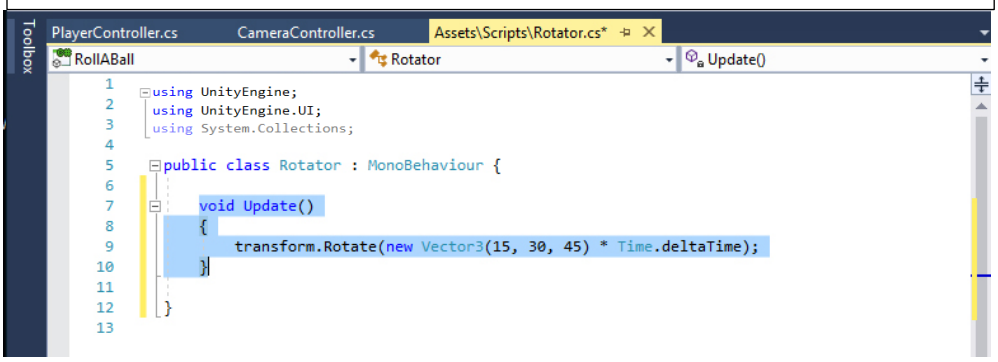
   c) Rename the Cube to '**PickUp**'
   d) Adjust its Transforms:



2) Let's have the PickUp rotate as it sits there. To do this we'll need another script:
   a) Select the PickUp
   b) Inspector → Add Component → **New Script**
      i) Name this Script 'Rotator'
      ii) Click CreateAndAdd

   c) Place the new script in the Scripts folder
   d) Open the new Script in VS
      i) Remove the default code as we did before
      ii) Replace it with the following and then save the file:

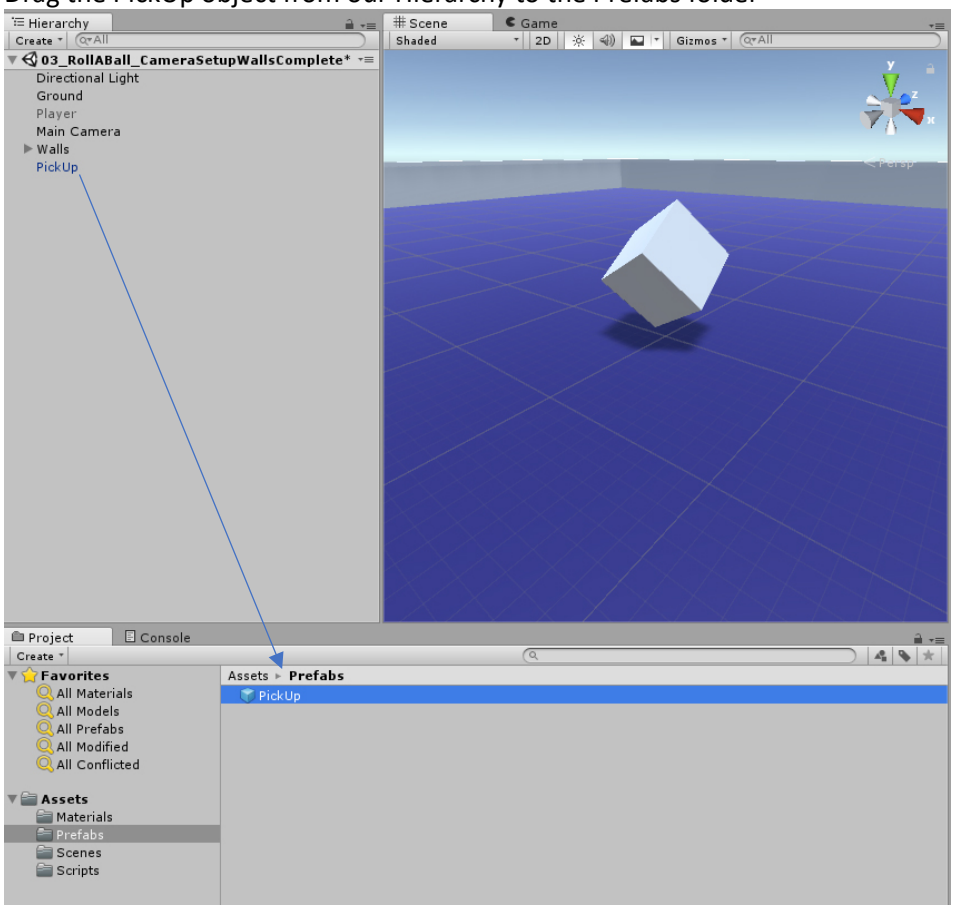
```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class Rotator : MonoBehaviour
{
    void Update()
    {
        transform.Rotate(new Vector3(15, 30, 45) * Time.deltaTime);
    }
}
```



3) Play Test!

4) Now we'd like to place duplicates of this cube around the play area as our pickup objects!
    a. Note – before we can do this however, we need to create what's called a **Prefrab**. A Prefab is like a template that contains a blueprint for a game object or game object family. In other words, it's kind of like creating an instanced object with set parameters. When we create a prefab, we can drag and drop the prefab into the scene. When we make changes to a prefab object, it will update all of the other duplicates.

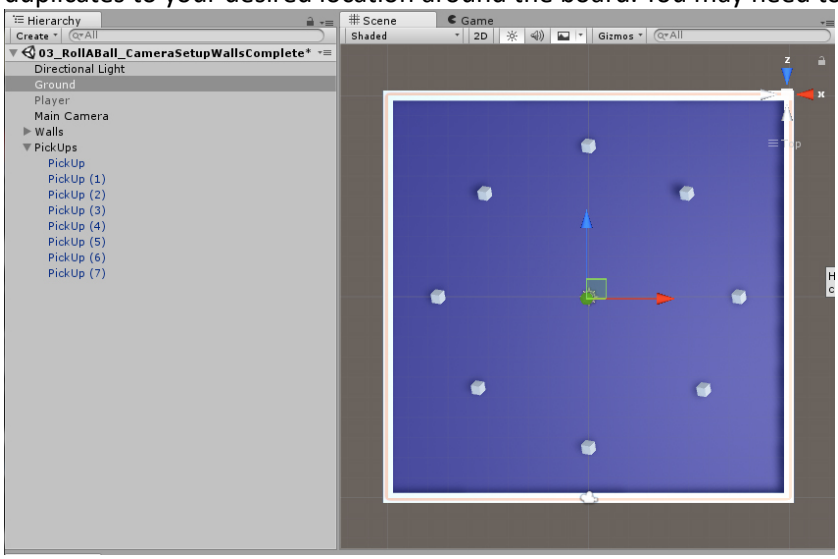
a) Create a Prefab folder in our Project structure
b) Drag the PickUp object from our Hierarchy to the Prefabs folder



c) Note – anytime we drag something from the Hierarchy to the Project structure like this we create a new Prefab asset containing a blueprint of our game object.

d) Let's organize our Hierarchy prior to creating our duplicates!
    i) Create a new Empty Game Object and name it 'PickUps'
    ii) Reset it to the Origin if it's not currently located there.
    iii) In the Hierarchy → Drag the first PickUp Object into the new PickUps group.

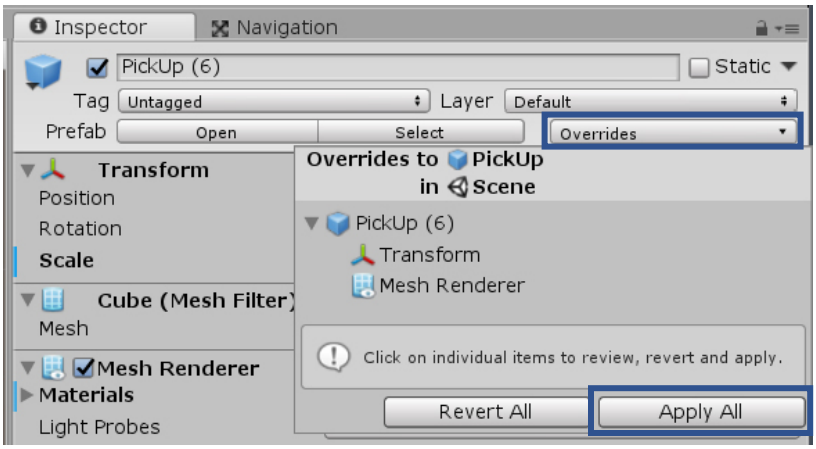
e) In the Top down view, move the first PickUp where you want it on the board and then duplicate it. Move the duplicates to your desired location around the board. You may need to adjust your pivot from Local to Global.



f) Play Test!

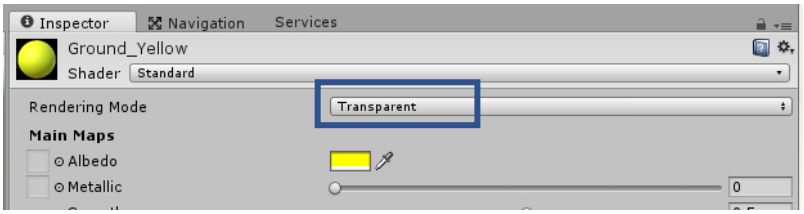
5) Now let's change the color of the cubes (we will need a new material for this):
   a) Select our current material in the Materials folder
   b) Duplicate it (**Cntrl d**)
   c) Rename it 'PickUp_Yellow'
   d) In the Inspector → Change the Albedo color to Yellow (or any other color that you want)

   e) Let's apply this to our Prefab.
      i) Drag the Material to one of the PickUp cubes
      ii) Click on the cube
      iii) In the Inspector →
         ⇒ Notice there's now a Prefab line item
         ⇒ Click on the Overrides drop down
         ⇒ Apply All



      iv) Note – this will apply the material to all of the cubes

   f) Let's make these a little more fun!
      i) Click on the PickUp_Yellow Material
      ii) In the Inspector → Rendering Mode → change to Transparent

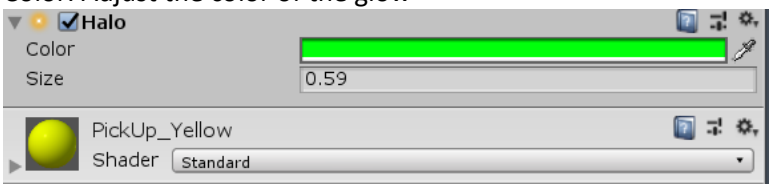


iii) Click on one of the PickUp cubes

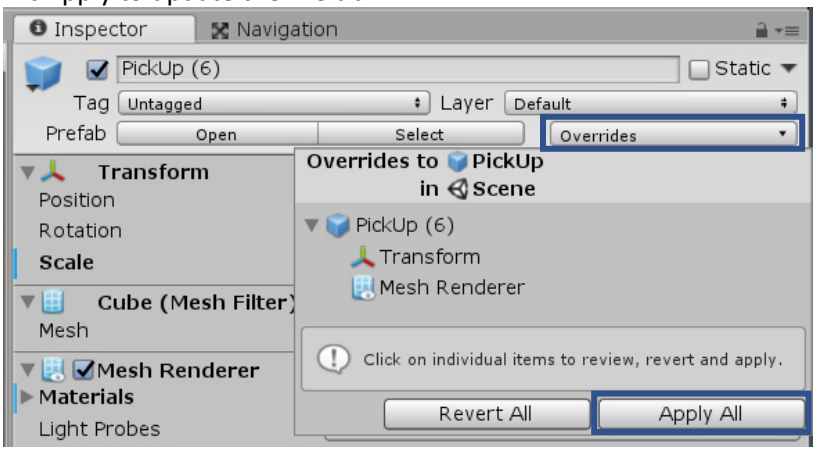
iv) In the Inspector → Add Component → Effects → **Halo**

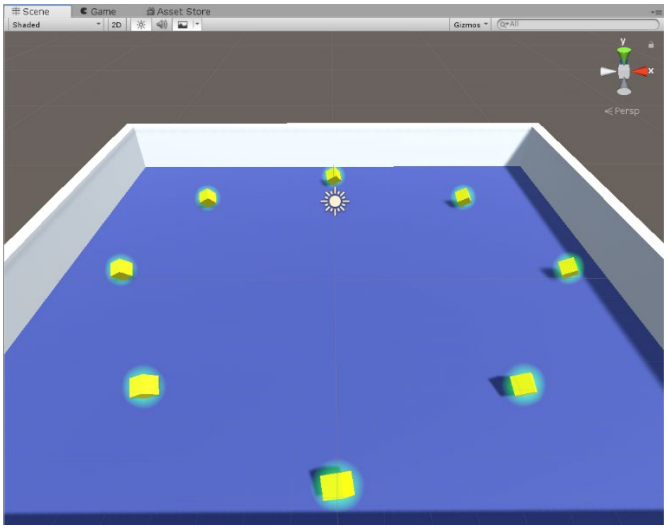
⇒ Size: Adjust the size of the glow you want

⇒ Color: Adjust the color of the glow



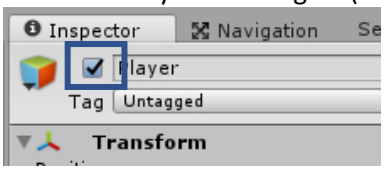
⇒ Hit Apply to update the Prefab





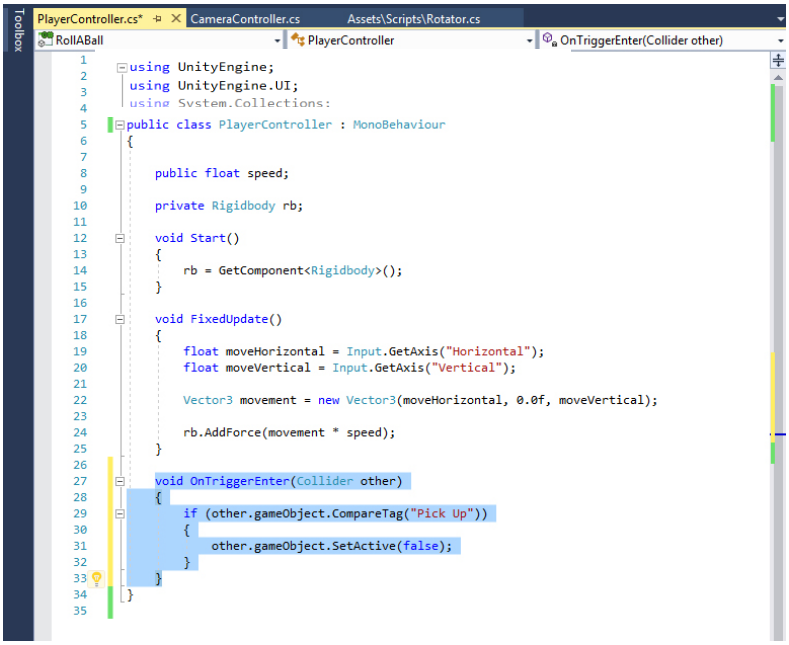
Collectables Picked Up

1) Now we need to tell the Player what to collide with and what happens when the collision happens.
2) Make the Player active again (if you hid it before)

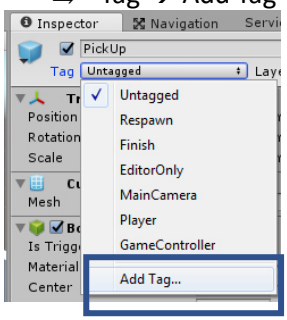


3) Open up the PlayerController Script in VS
   a) Add the following code right before the very last } and save:

```
void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Pick Up"))
        {
            other.gameObject.SetActive(false);
        }
    }
```



   b) In the Project Structure → Prefab folder → select the Prefab PickUp object
      i) Inspector → Open Prefab
         ⇒ Tag → Add Tag

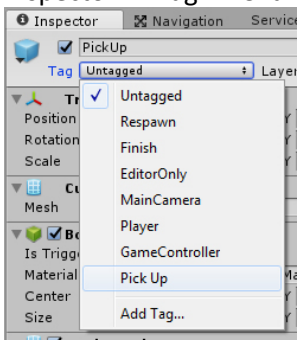


Note – in the code we're referencing a tag that we need to create. This tag will only be associated with our PickUps. This is how our Player will know to only pick up the PickUp objects and not things like the floor and walls. The naming convention that we used in the script needs to be consistent with what we create here. "Pick Up". It is space and case sensitive!!

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag("Pick Up"))
    {
        other.gameObject.SetActive(false);
    }
}
```

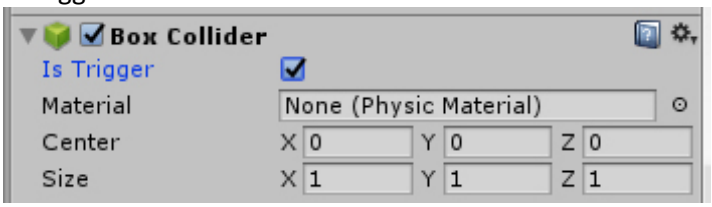
c) Click on the + button to create the new tag
d) Name it "Pick Up"
e) Hit Save

f) Click back on the PickUp Prefab
g) Inspector → Tag → Change to Pick Up



4) Play Test!

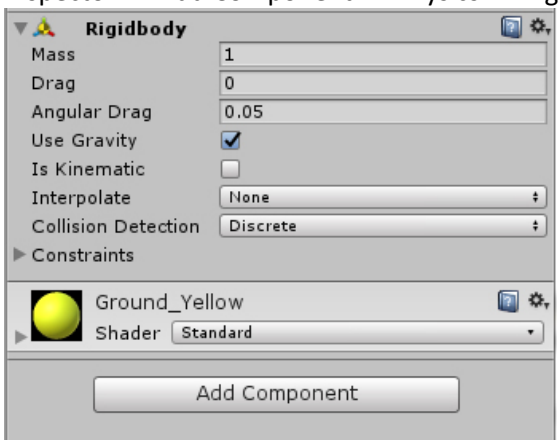
5) Note - You probably noticed that our sphere is still bouncing off our cubes. This is because our cubes are still using colliders and we need them to be Triggers so that our Sphere can occupy the same space with each cube it runs into them and essentially absorb them (or make them inactive upon contact) as per our code. To do this:
   a) Select the PickUp Prefab
   b) In the Inspector → Box Collider
      i) Is Trigger: Check ON



      ii) Play Test!

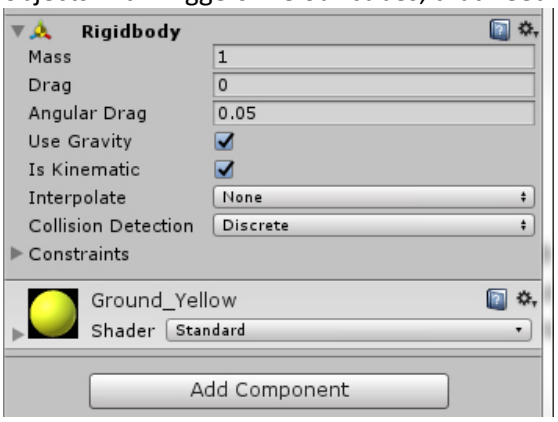
6) Note – right now everything appears to be working, but there is one issue and it has to do with performance and resource depletion based on how Unity optimizes its Physics. As a performance optimization, Unity calculates all the volumes of all the static colliders in the scene and holds this information in cache. This makes sense as static collider shouldn't move and this saves recalculating this information every frame. The problem with our scene is that our cubes are rotating. Anytime we move, rotate or scale a static collider, Unity will recalculate all the static colliders again and update the static collider cache. In doing this we're tapping our system's resources. As an alternative we can move, scale and rotate Dynamic colliders as often as we want and Unity won't re-cache any collider volumes. Unity expects us to move colliders, but we need to indicate which ones are dynamic. We do this by using the RigidBody component. Any game object with a collider and a Rigidbody is considered dynamic. Any game object with just a collider and NO Rigidbody is considered Static. Right now, our cubes don't have a Rigidbody so they're considered Static. So, Unity it is recalculating our Static collider cache every frame. So, we need to add a Rigidbody to the PickUp object Prefrab.

a) Select the Prefab
b) Inspector → Add Component → Physics → Rigidbody



c) Note – Now our Cubes are dynamic. However, if we hit play they'll just fall through the floor because 'Use Gravity' is checked on and b/c we changed the cubes to Triggers, they no longer collide with the floor. We could just uncheck 'Use Gravity', but although they would no longer respond to gravity, they would respond to physics forces. So rather than checking off Gravity:

i) **Is Kinematic**: Check ON
⇒ Now our Cubes will not react to physics forces and they can be animated and moved via their transforms. This is great for things that have colliders like elevators and moving platforms as well as objects with Triggers like our cubes, that need to animate or move by their transforms.

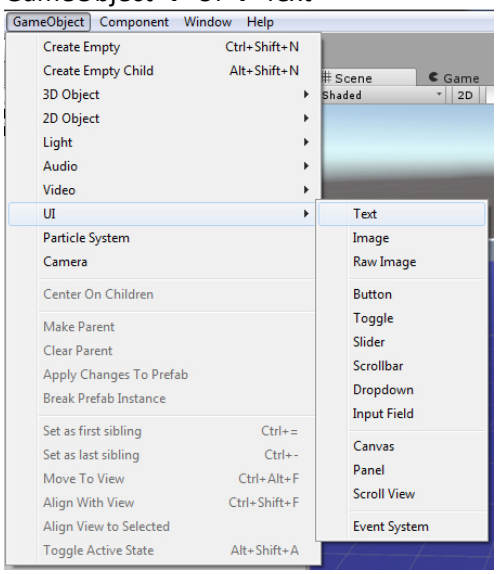


⇒ Play Test!
- Now things are working and performing the way they should

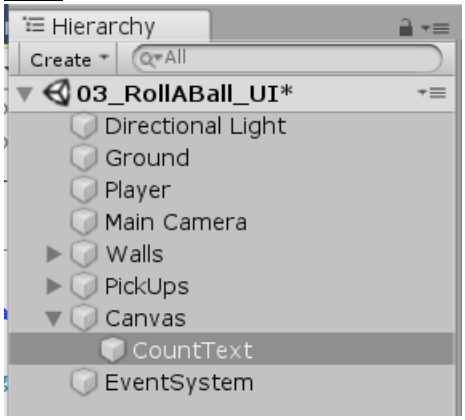
d) To Recap:
   i) Static Colliders shouldn't move (walls, floors, etc.)
   ii) Dynamic Colliders can move and should have a RigidBody component along with their collider
   iii) Standard RigidBodies are moved using Physics forces
   iv) Kinematic RigidBodies are moved using their transforms

Keeping Score and UI

1) Note - We need something to store the counted PickUps and then add to that stored number. We can do this by modifying the Player script. We also need something to display the count and let us know when we've 'won'.
2) Create a new UI text element (this will allow us to display text in our scene)
   a) GameObject → UI → Text



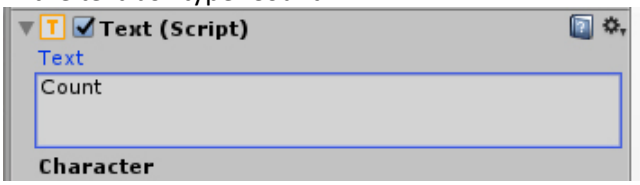
Note – in order for text to behave the way it's supposed to, it needs to be a child of a Canvas.



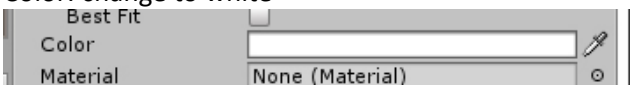
   b) Rename the text object 'CountText'

c) Inspector →

i) In the text box type: Count

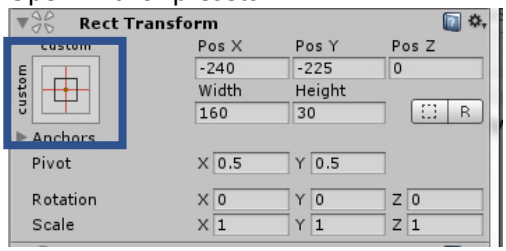


ii) Color: change to white

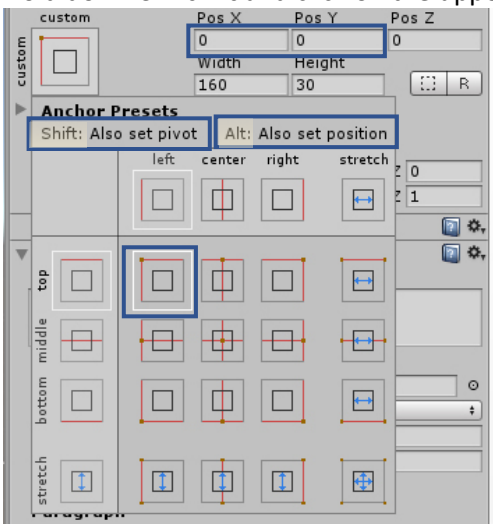


iii) We want the text to appear in the upper left corner of our screen in play mode:

⇒ Open Anchor presets



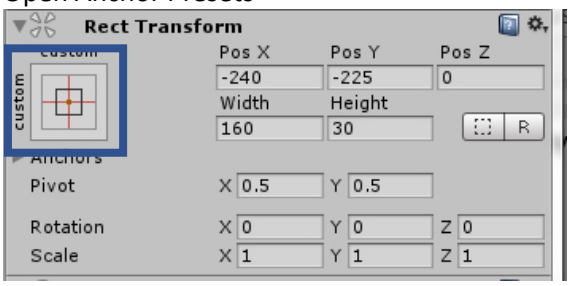
⇒ Hold down **Shift Alt** and click on the upper left corner option



⇒ To give it a little more room:

- Pos X: 10
- Pox Y: -10

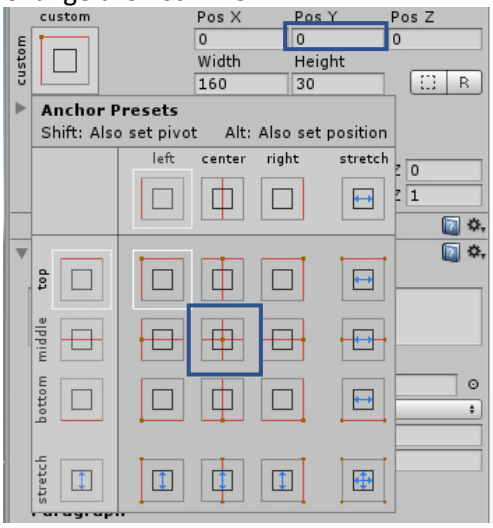
3) Now let's create the Win text
   a) GameObject → UI → Text
   b) Note – this will place the new text in our current canvas
   c) Rename it 'WinText'
   d) Inspector →
      i) Color: white
      ii) Font size: 24
      iii) Text: You Win!!
      iv) Open Anchor Presets



⇒ Hold down Shift and Alt

⇒ Select the center option

⇒ Change the Pos Y: 75

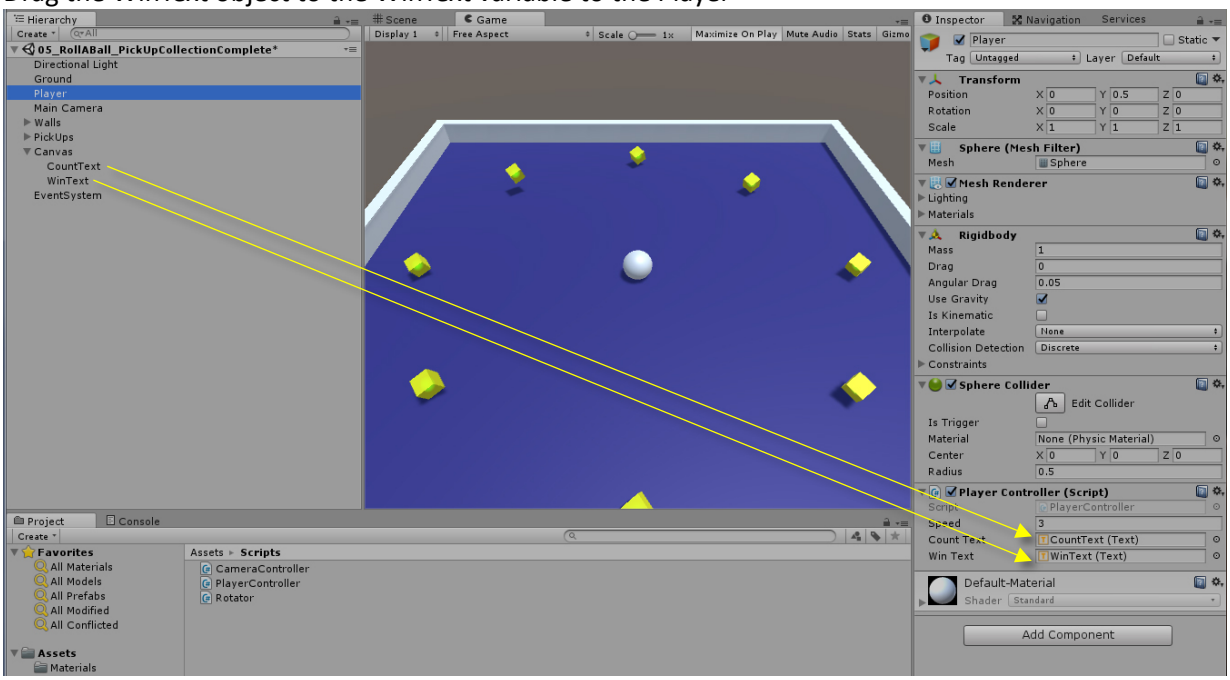


4) Now let's adjust our PlayerController script to keep count of our objects and display them via our text:
   a) Open up the PlayerController script in VS
   b) Add the flowing lines of code (highlighted in yellow):

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class PlayerController : MonoBehaviour
{

    public float speed;
    public Text countText;
    public Text winText;

    private Rigidbody rb;
    private int count;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        count = 0;
        SetCountText();
        winText.text = "";
    }

    void FixedUpdate()
    {
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

        rb.AddForce(movement * speed);
    }

    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Pick Up"))
        {
            other.gameObject.SetActive(false);
            count = count + 1;
            SetCountText();
        }
    }

    void SetCountText()
    {
        countText.text = "Count: " + count.ToString();
        if (count >= 12)
        {
            winText.text = "You Win!";
        }
    }
}
```

   c) Save the script.
   d) Note - if (count >= 8): you will need to change this value depending on how many pickup objects you have in your scene in order for the win screen to pop up!

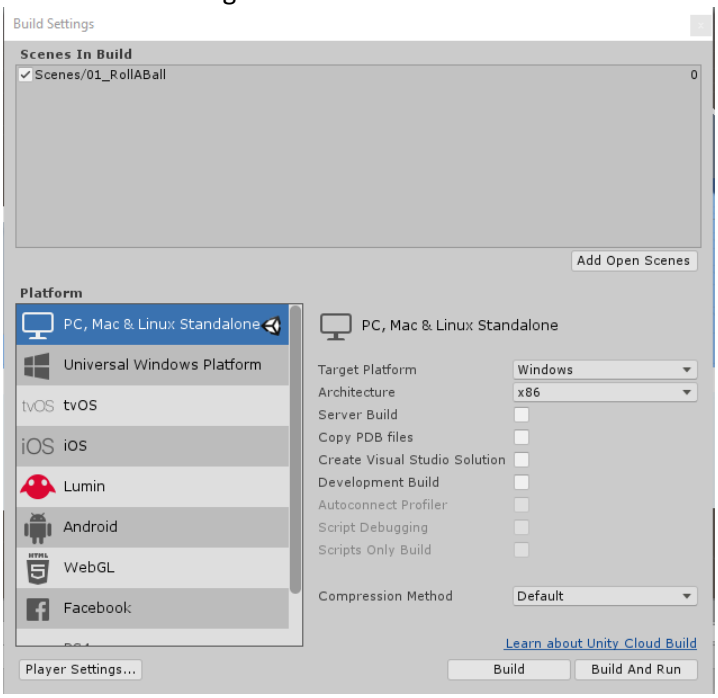
5) Associate the text to Player (sphere)
   a) Click on Player in Hierarchy
   b) Inspector →
      i) Drag the CountText object to the CountText variable of the Player
      ii) Drag the WinText object to the WinText variable to the Player



      iii) Play test!

Building The Game

1) File → Build Settings



a) Hit: Add Open Scene
b) PC, Mac & Linux Standalone
c) Target Platform: Windows OR Mac OS X
d) Architecture: x86_64
e) Click Build
    i) Create a new folder within your project structure called ‘Builds’
    ii) Open this folder
    iii) Name your .exe: “RollABall”
    iv) Hit Save
    v) Note – To play your game open the .exe
    vi) Note – Do not delete the folders that gets created along with the .exe. The .exe needs that info to run your game.