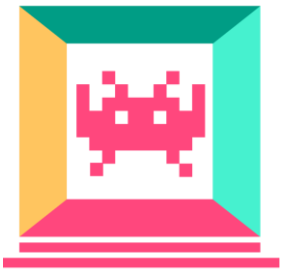


Learn Unity by Creating a 3D Multi-Level Platformer Game

By Pablo Farias Navarro

Certified Unity Developer and Founder of Zenva

Before diving into this eBook, why not check out some resources that will supercharge your coding skills:

Table of Contents

Introduction

Interested in making games with Unity? In this guide you'll learn to create a simple a 3D, multi-level platformer game with Unity.

We'll start from the very basics and I've done my best to leave no stone unturned!

Tutorial requirements and project files

No prior Unity or C# experience is required to follow along, although you should have familiarity with basic programming concepts such as variables, conditional statements and objects.

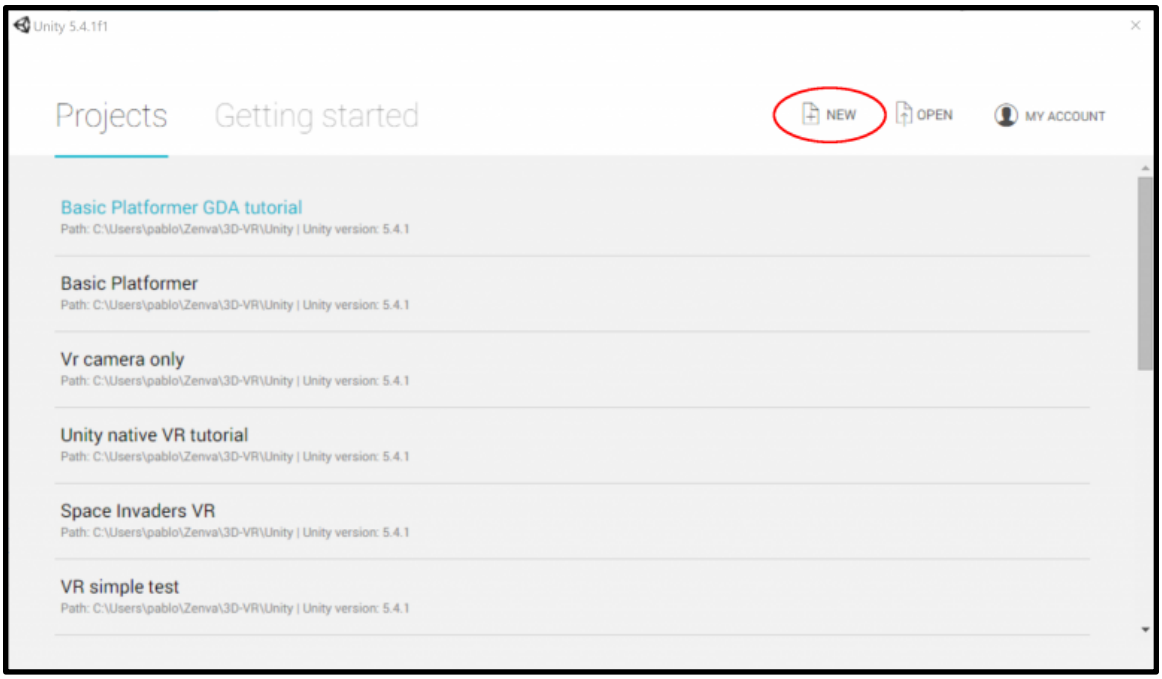
Project files can be downloaded here. This zip file contains all the files included in the Assets folder. You'll still need to create a new project as covered in the tutorial.

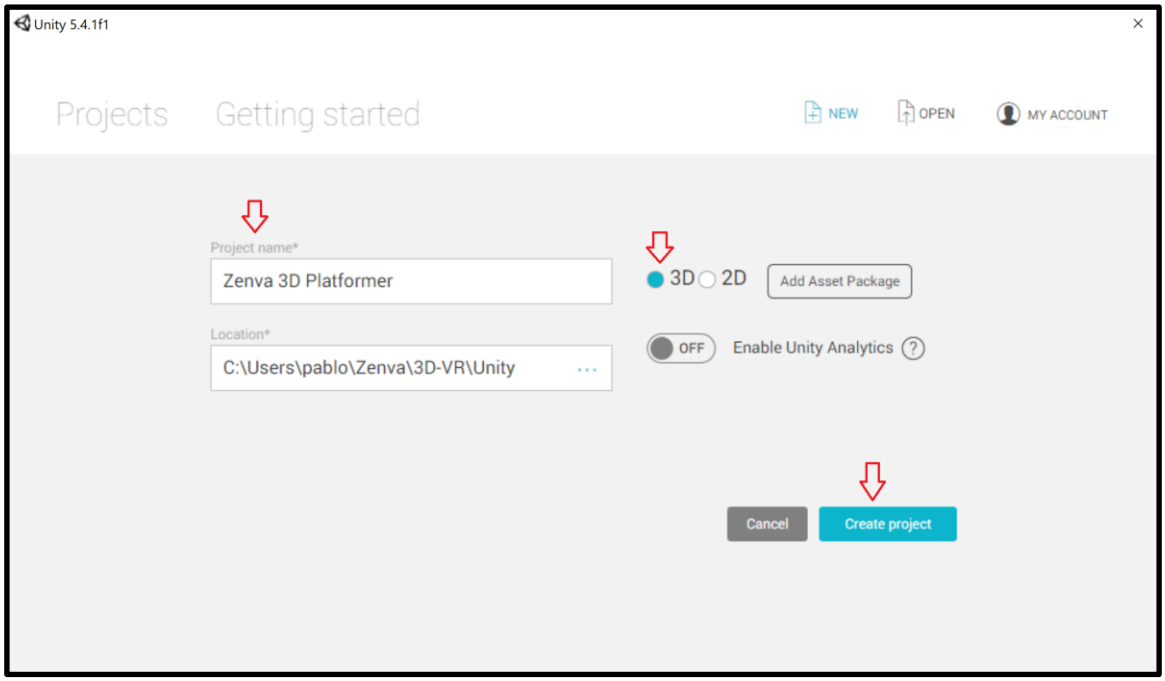
Some of the topics we'll cover include:

- Basics of the Unity Editor, scenes, game objects and components
- Understanding game object core methods in C# scripts
- Working with object transforms both from the Inspector and from scripts
- Accessing user input from scripts
- Collision detection and rigid bodies
- Implementing multiple scenes to create a multi-level game and passing objects along
- Basic UI and importing fonts
- Building your game

Scene basics

Start by opening Unity. Click *New*, enter a name for the project ("Zenva 3D Platformer"), make sure *3D* is selected, then click on *Create project*.

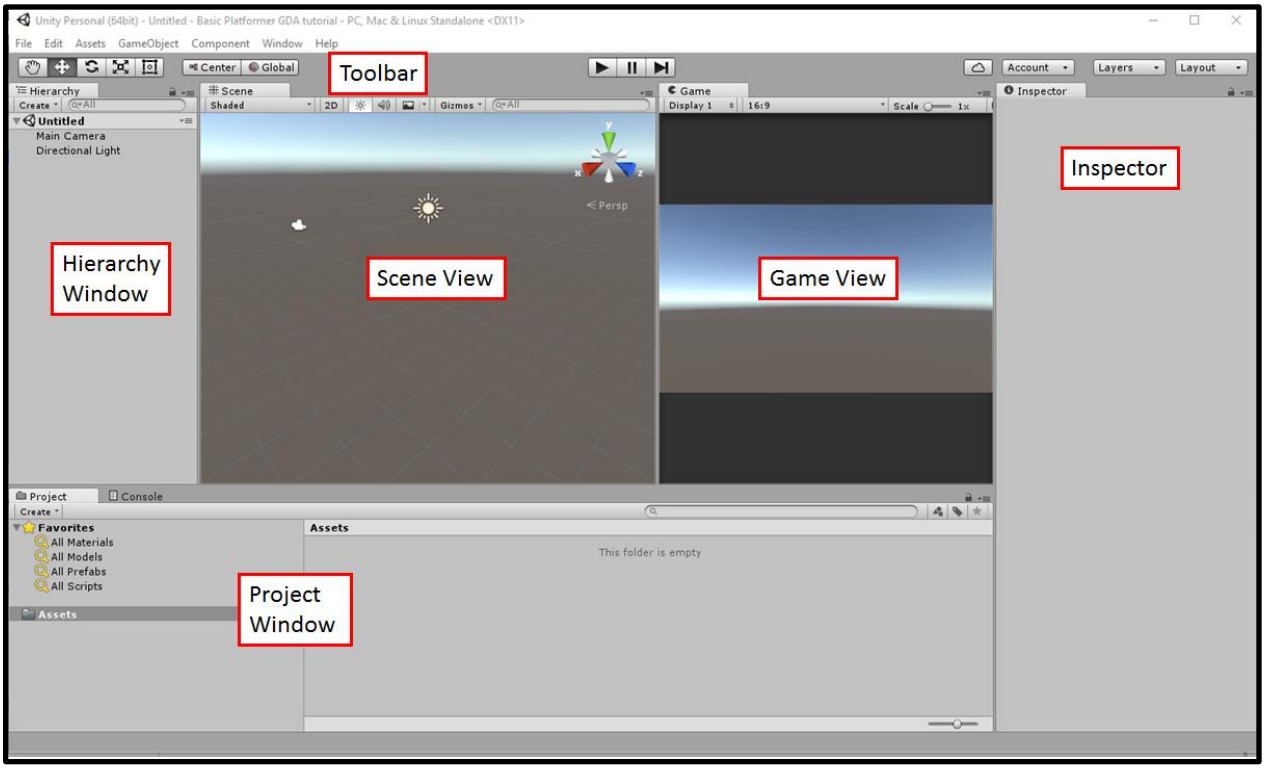




This will bring us to an empty Unity scene. I will now describe some of the main panels and elements present in the Unity Editor. If you are already familiar with the basics you can skip straight to the next section.

What is a **scene**? The word “scene” comes from the Greek *skene*, and was used back in ancient world for the area in the theater that faces the public, where all the action takes place. In Unity the definition is not too distant: a scene in your game is an object that contains all game objects such as players, enemies, cameras, lights, etc. Every game object within a scene has a position which is described in coordinates X, Y and Z.

The following image shows the main elements we find in the Unity Editor:

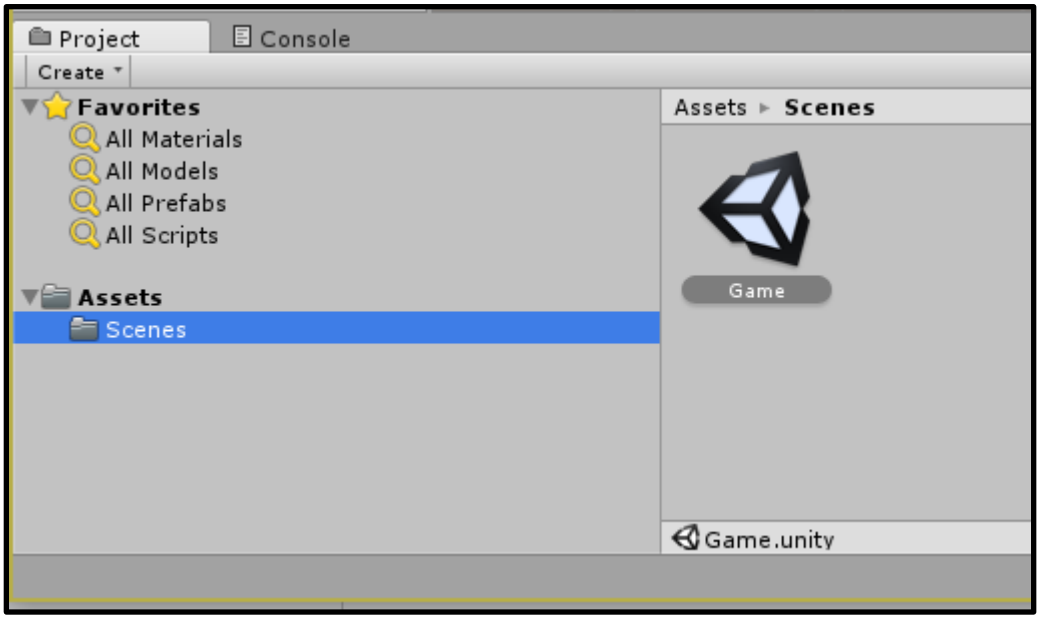


- **Project Window:** this area shows the files and folders of our project. The only folder we’ll see in our new scene is the *Assets* folder, which is created automatically and it’s where we’ll place all the game assets (3D models, scripts, audio files, images, etc).
- **Hierarchy Window:** shows all the game objects that are present in our scene. By default, Unity creates a camera and a directional light.
- **Scene View:** shows the 3D view of your game scene. All the objects that you create in the *Hierarchy Window* will appear here in their corresponding X, Y, Z positions, and by using different “gizmos” or tools you can move, rotate and scale these objects around.
- **Game View:** this view shows what the game actually looks like. In this case, it shows whatever the camera (which was created by default) is looking at.

- **Inspector:** whenever you select a game object, the Inspector will show different options and properties that are available for that game object.
- **Toolbar:** this area contains different tools we can use to modify our game objects, move around the scene, and modify how the Editor works.

When creating a new scene the first thing you'll want to do is to save it. Let's go to *File – Save Scene* and give it a name ("*Game*").

As a Unity project grows, it becomes paramount to keep your files organized. In the *Project Window*, right click in Assets and create a new folder called *Scenes*. Drag our newly created *Game* scene in there.

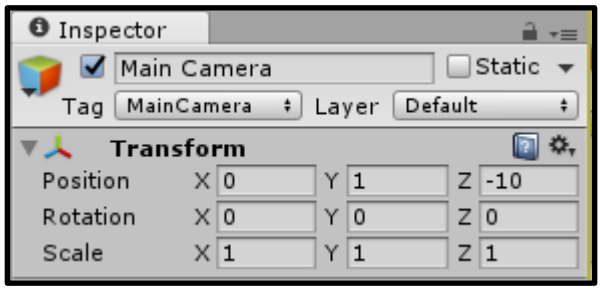


Transform Component

All game objects in a scene have a **Component** named **Transform**. What is a component? Think of components as reusable "Lego pieces" that can be used in different objects. A component provides a game object with certain behaviors and properties.

As we mentioned before, all game objects in a scene have a position described in coordinates X,Y,Z. That in Unity is called the **Transform component**. This is the only component that is present in all game objects. There can't be a game object without a Transform component!

On the *Hierarchy Window*, click on both the default camera and directional light, and observe how the *Transform* component appears in the Inspector, indicating the **position** of the object, plus values for **rotation** and **scale**.

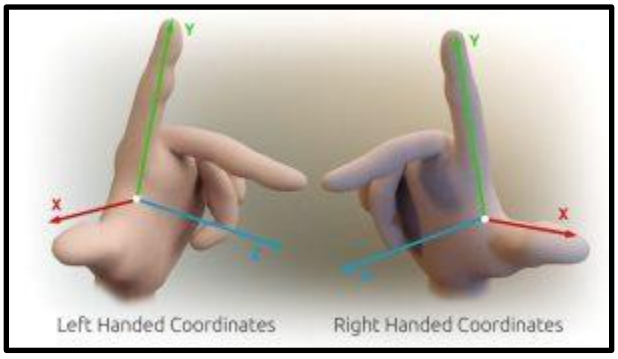


Lets create a Cube to experiment with transforms. In the *Hierarchy Window*, right click and select *3D Object – Cube*.

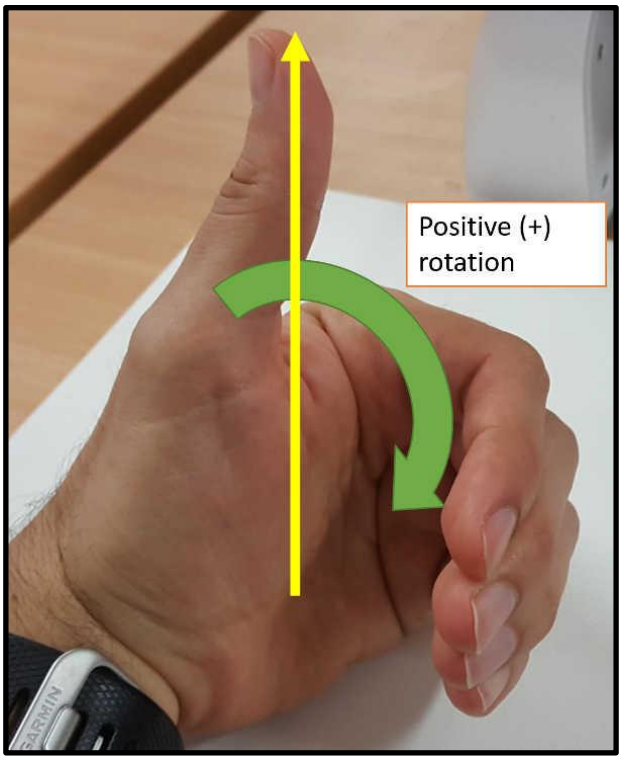
Click on the cube and change the position values in it's *Transform component* to see how it's position in the scene changes as well. Experiment changing the *scale* and *rotation* as well.

A scale of 1 means no change in size. If you set it to 2, it will twice the size in that axis. If you want it halved, set the scale to 0.5.

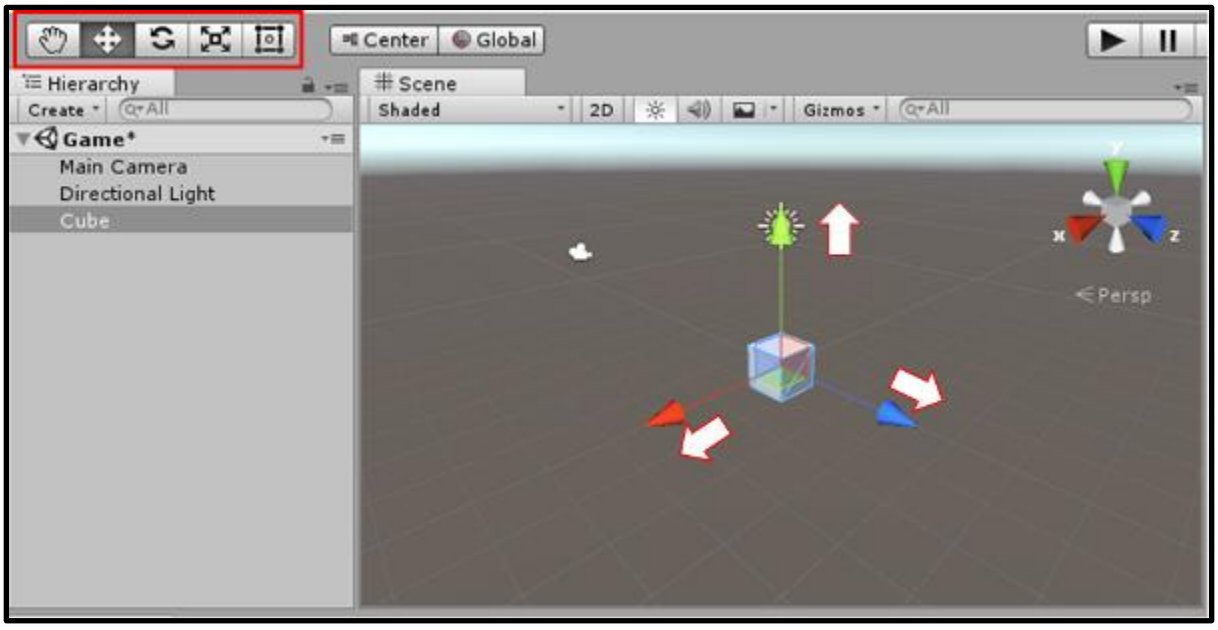
Notice that Unity uses what's called a left-handed coordinate system:



Rotation values are entered in degrees. If you enter, for instance 45 in X, that means that the game object will rotate 45° around the X axis. To determine to which side it will rotate, use the *left-hand rule* as shown in the image below. If the rotation value is negative, use your right hand.



Besides changing the Transform properties of a game object from the Inspector, you can do so by using the *Toolbar* buttons and the “gizmos” in the *Scene View*:

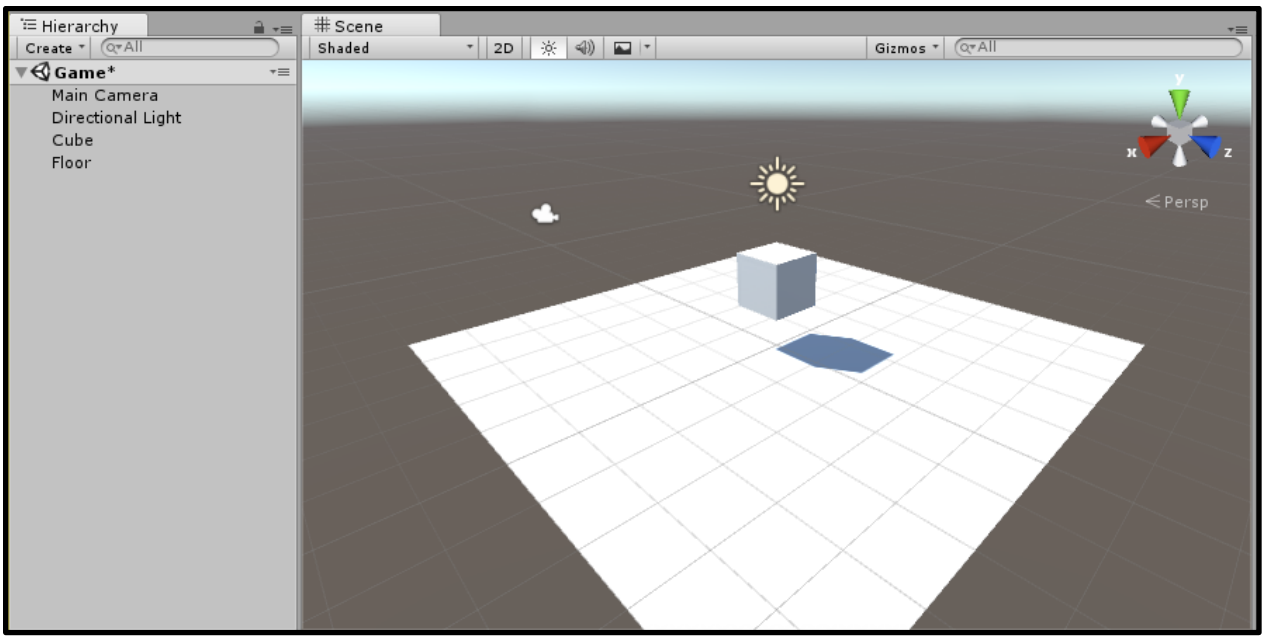


Unity units

You may wonder, what's the unit of measure in a Unity game? if we move an object from X = 1 to X = 2, how much would that represent in the real world? Unity uses *Unity units*. By convention (and this even includes some official Unity tutorials), **1 Unity unit is 1 meter**.

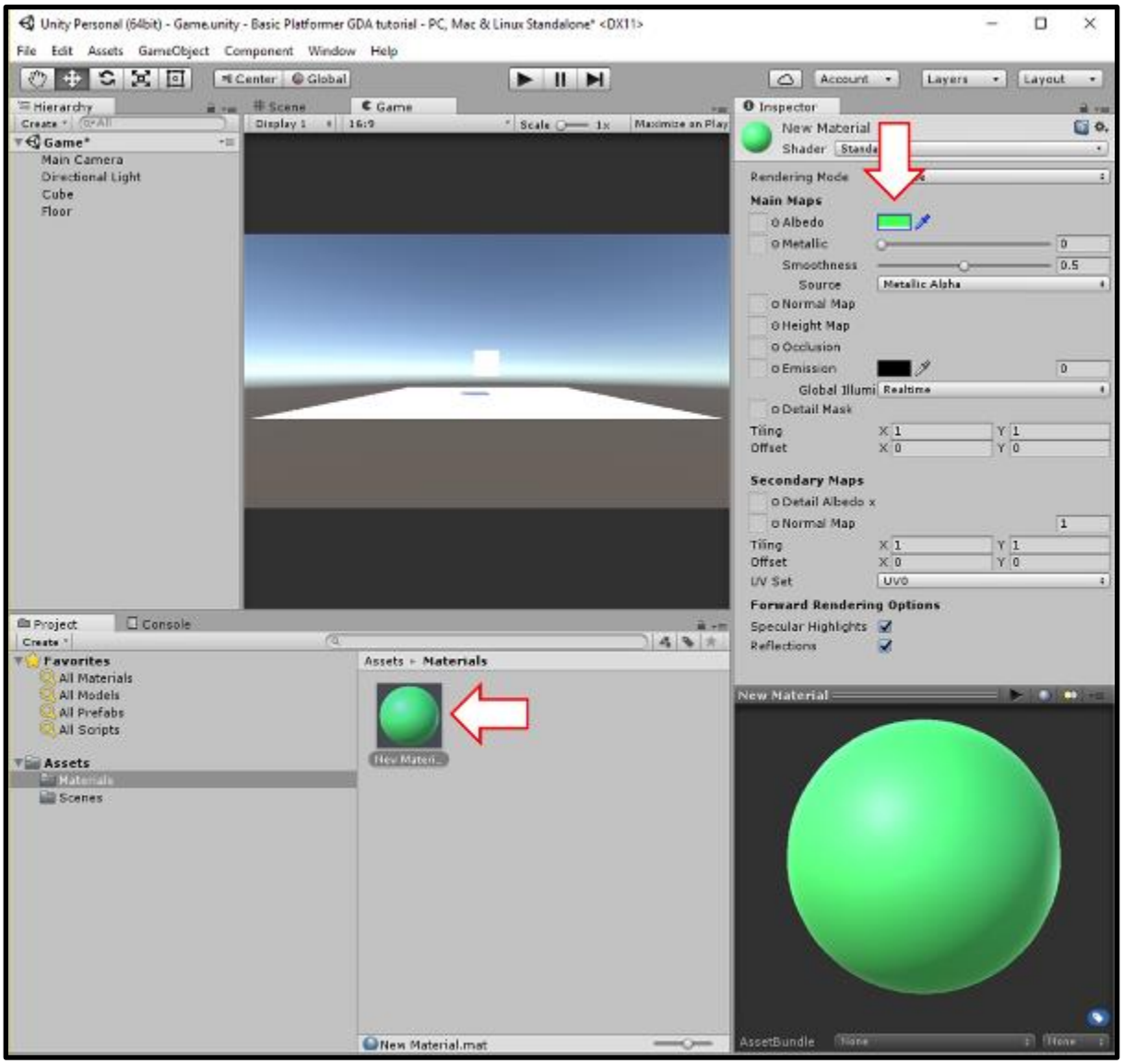
The Floor

Let's go ahead and create the floor of the game. For that, we will use a plane. Right click on your scene in the Hierarchy Window and select *3D Object – Plane*. This will bring up a plane into our scene. From the *Hierarchy Window*, we can rename this object by right clicking – *Rename*, by selecting it and pressing F2, or by single-clicking on it after we've selected it. Call it "Floor".

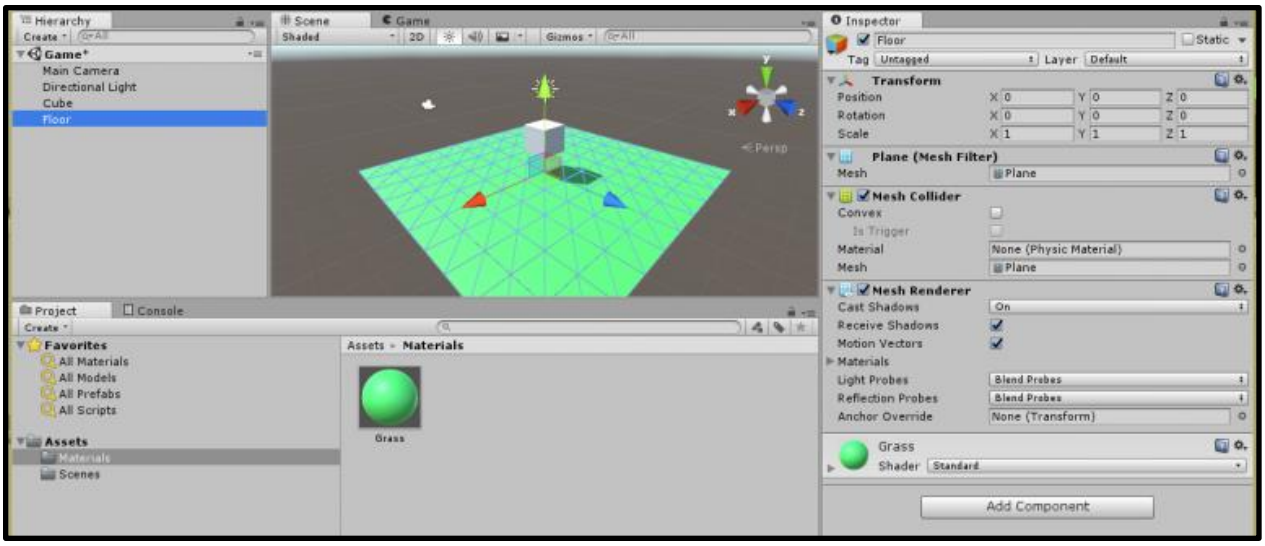


We will now create a new **material** so that it can look green. Un Unity, a *material* is an asset that controls the visual appearance of a game object. We can easily create materials from the *Project Window* and assign them to objects in our scene.

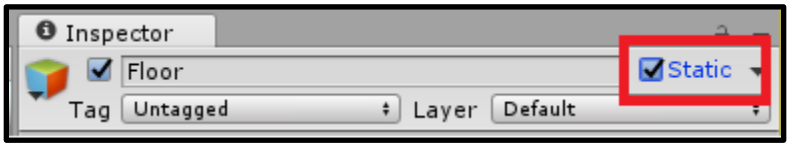
Create a new folder inside of *Assets* called *Materials*. Inside of this new folder, right click and select *Create – Material*. This will create a new material file. Rename it to "Grass". If you click on this file, the Inspector will show the properties of the newly created material. Click on the color white in *Albedo* and pick a color for the grass in the game:



Now drag the material file to the floor in the *Scene View*. The plane will look green, and if you click on it, you'll see the material in the *Mesh Renderer* component in the *Inspector*.



Lastly, since the floor won't be moving around in the game, check the *Static* checkbox located on the top-right of the *Inspector*.



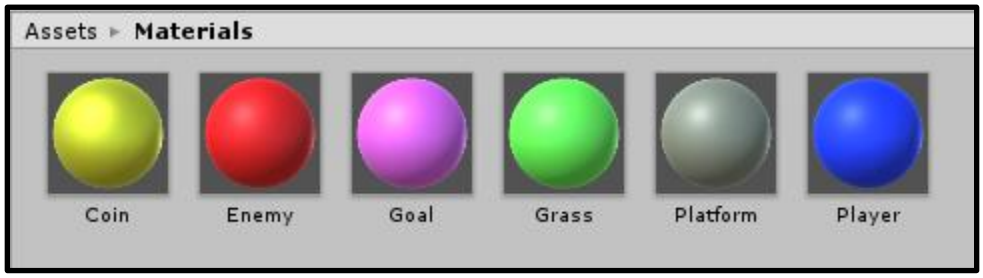
By making a game object *static*, we are informing Unity that this object won't be moving in our game. This allows Unity to perform behind the scenes optimizations when running the game.

Adding more game elements

Let's add the remaining elements of our game. Start by creating the new materials. Pick whichever color you want for each one:

- Platform
- Coin
- Enemy
- Goal
- Player

This is what mine look like:

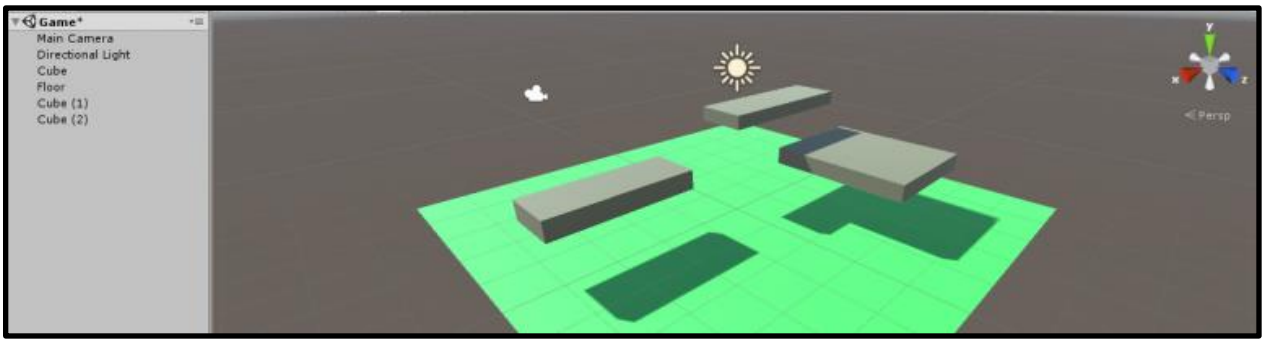


What we'll do next is add all the remaining elements to create our level, so that we can get a clear idea of what it will look like. We haven't implemented our player, the behavior of the enemies or coins yet. Moreover, we haven't written a single line of code. However, it's good practice to design a game as early as possible.

Moving around blocks in Unity like we'll do now is very easy and anyone can do it. This process can give you a very clear idea of what your game will look like, and allow you to save time further down the road, and to show other people what the game will look like. This process is called **prototyping**.

Let's thus begin this process by adding some cubes to be used as platforms. Use the position *gizmos* or the *Inspector* to position them in different places. Set their scale in Y to a smaller value to make them thinner, and scale them up in X and Y so make them wider. Make sure to drag the *Platform* material we created to give them the color you want.

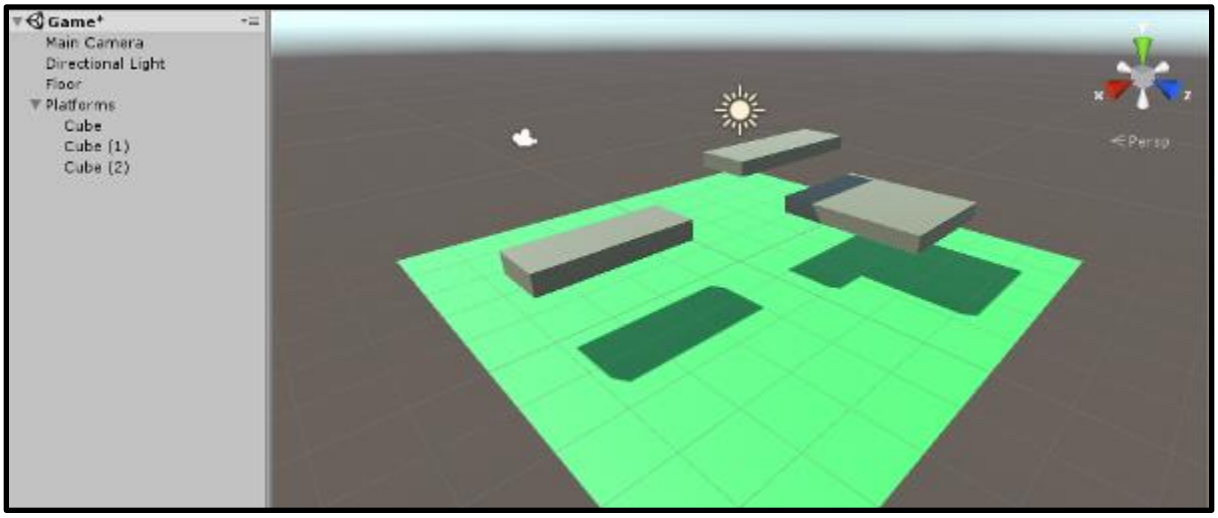
Since platforms won't be moving make sure to set them as "static" (unless you want to create moving platforms of course!).



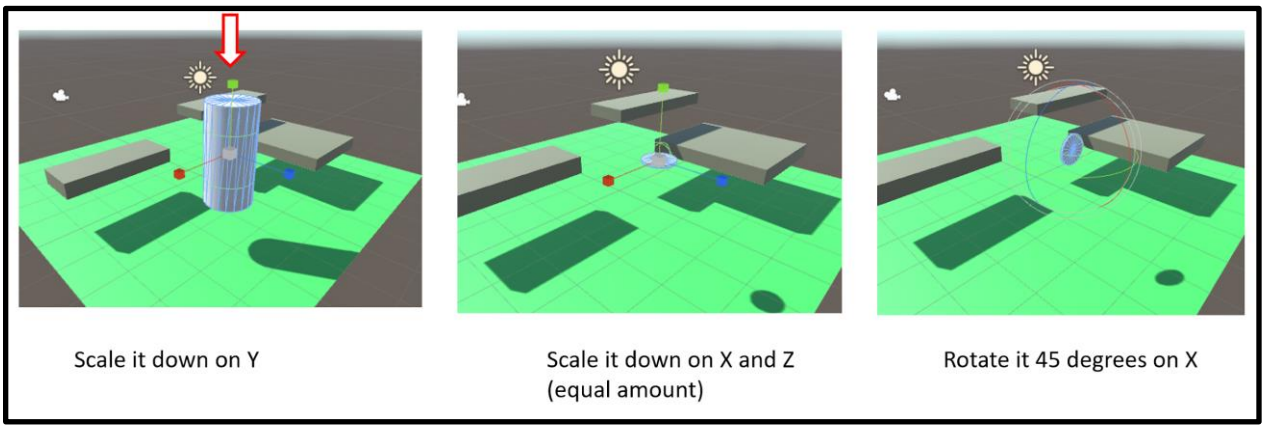
As we create more platforms, the *Hierarchy Window* can start to get crowded of elements. Game objects in Unity can be *children* of other objects. This means that their position is relative

to that of the parent. In this case, grouping all the platforms inside of a single parent object can help us keep this window more clear – we won't be moving this parent object.

In the Hierarchy Window right click and select *Create Empty*. Rename this new object to "Platforms". Drag and drop all the platforms you created into this object. Notice that even though Platforms is empty (it doesn't render anything on the screen), it still has a Transform component. As we said before, this component is always present in Unity game objects.



For the coins we'll start by creating a cylinder in the *Hierarchy Window* (*3D Object – Cylinder*). Shrink it (this means, *scale* it down) on Y so that it looks more like a coin. Also, scale it down on X and Z to make it smaller (I've never seen a coin with a 1-meter diameter!). Lastly, rotate it 90 degrees in X or Z.



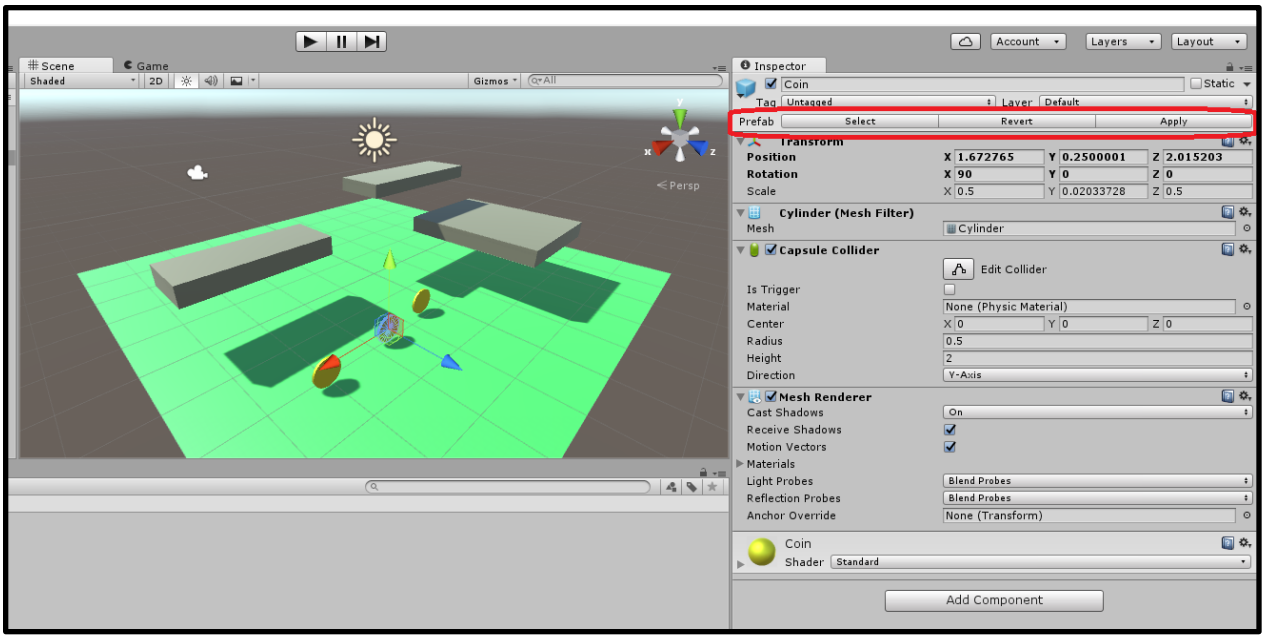
Rename the cylinder to “Coin”. Drag the Coin material into your coin and you’ll have your coin ready! Once we get into scripting, coins will have a C# script associated to them which will determine how they behave. Since we’ll have many coins, having to re-create them each time is not the best approach. Imagine we want to change how all coins behave at once? We need to create what’s called a **prefab**.

Unity *prefabs* are templates of game objects that can be reused (even used in different projects), and that allow us to generate many game objects that share properties and behaviors. Changes made to a prefab are reflected in all of it’s instances.

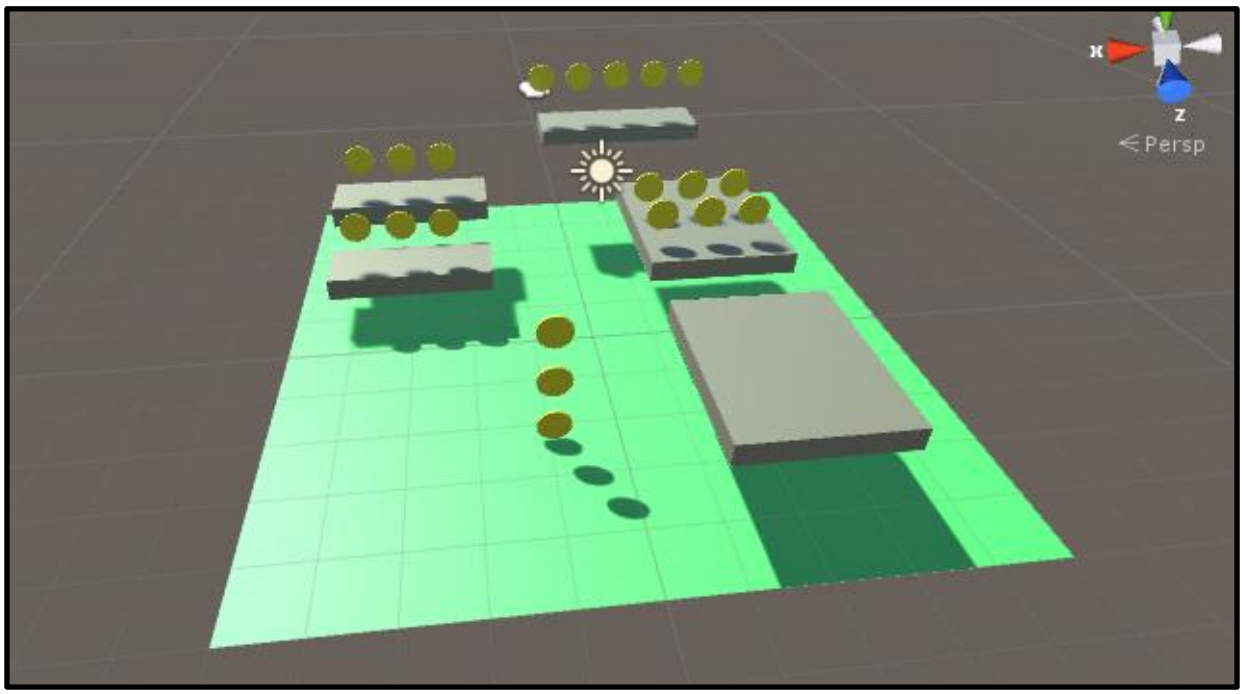
Create a new folder in *Assets* called *Prefabs* to keep our code organized. To create a prefab, simply drag and drop the coin you created (from the *Hierarchy Window*) into this new folder in the *Project Window*.

Now that we have our prefab ready, you could safely delete the coin from the *Hierarchy Window*, but you don’t really have to. To create more *instances* of our prefab, simply drag and drop the coin Prefab into the *Scene View*. Do it many times (you can also do it once and duplicate the object with *Control + D*).

If you select any of these instance you’ll see a *Prefab* area in the *Inspector*. If you click on *Select* in there, see how the Coin prefab is selected. If you make changes to this particular instance (for example, change the scale, rotation or material) and press *Apply*, the changes made will be applied to the prefab itself, changing thus all other instances!



Place coins across your level. This will help you get familiar with moving around the scene, duplicating or copying and pasting objects, and using the position gizmo. When you are done we'll finish off with the design of our level. Make sure to group all the coins in an empty object, just like we did with the platforms before.



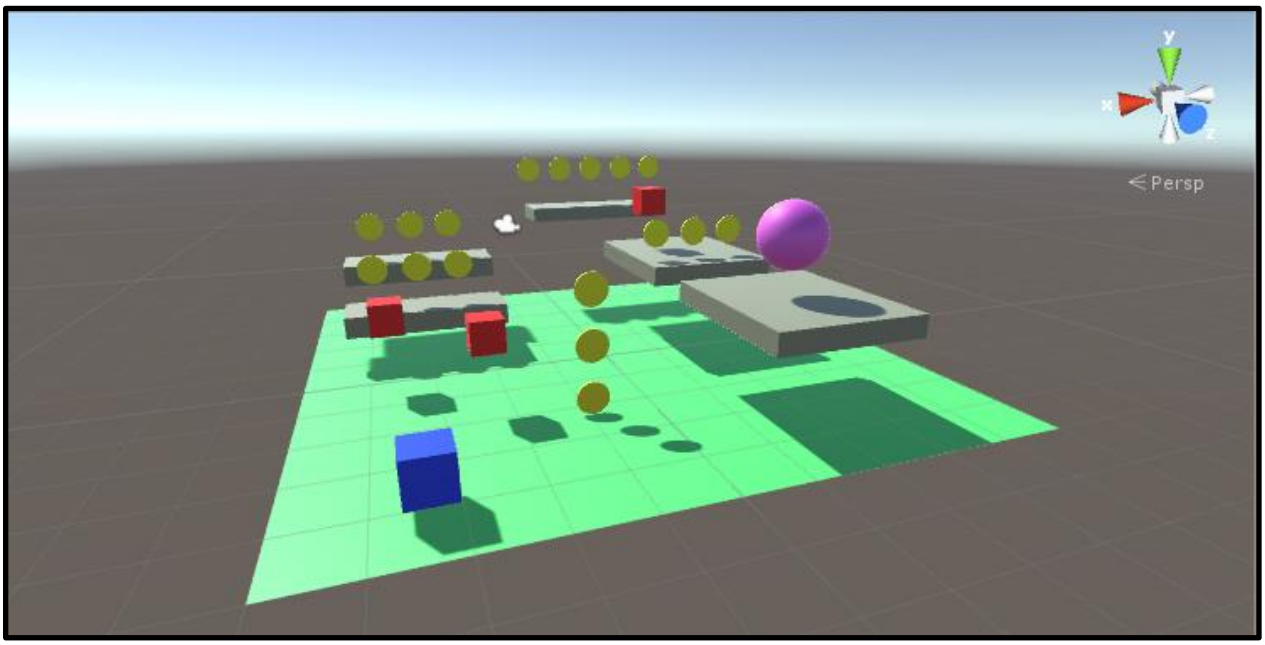
We'll now follow a similar process for the player, enemies and the level goal. For the player and enemies do the following:

- Create a cube
- Scale it to 0.6 in all axes
- Assign the corresponding material
- Drag to the Prefabs folder
- Drag the newly created prefab on to the Scene View to create an instance (for the enemy, create more than one, and group them in an empty object)

We can, of course, change of all this later, so don't much in too much thought or try to be perfectionist at this stage.

For the goal, the only difference is that instead of a cube, we'll use a sphere (*3D Object – Sphere*). The process described about is the same.

This is what my level looks like:



Pro tip: when moving a game object with the position *gizmo*, if you keep the Control key pressed (CMD in Mac) the object will move in fixed increments (which you can change in *Edit – Snap Settings*). If you grab the object from it's center, and press both Control and Shift, the

object will snap to the surface of any near object. This is useful to place the player on the ground with precision. Read the documentation under “Snapping” for more details.

Coin rotation script

Coins will always be rotating around the Y axis. This is really a cosmetic aspect of the game. As you might have guessed, I like to create a rough version of the full game before diving into this kinds of details (which I tend to leave for the very end). However, I think coin rotation can give us a very simple example to cover the basics of *Unity scripting*, so this will be our first approach to scripting, in preparation for the more complex implementation of the player controller.

Unity scripts are a necessary part of any game. Scripts can be written in C#, UnityScript (aka “JavaScript for Unity”) and a language called Boo. C# is the most popular option these days, so that’s the only language we’ll be using.

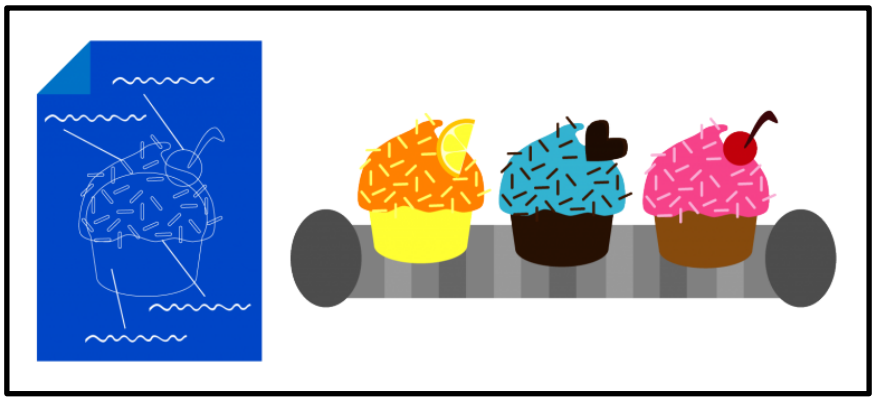
Let’s begin by creating a new folder inside of *Assets* called *Scripts*. In this new folder, right click and select *Create – C# Script*, name it *CoinController*. Voilá! You have created your first Unity script.

Double click on this new file and Visual Studio will open. The default contents are as follows:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class CoinController : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9
10     }
11
12     // Update is called once per frame
13     void Update () {
14
15     }
16 }
```

What we have here is:

- A new class has been created for us. This class is called `CoinController`, and inherits from another class called `MonoBehaviour`. Think of a class as a *blueprint* ("a recipe to make a cupcake") that can be used to create objects with certain characteristics and behaviors ("an actual cupcake"). Objects created from a class are called **instances** of that class.



- Our new class has two methods: `Start` and `Update`. In Unity, there are some reserved method names used in classes that inherit from `MonoBehaviour`. We'll now explain what these two methods do. You can find the full list of `MonoBehaviour` methods here.
    - `Start` is called on the first frame of the game.
    - `Update` is called on every frame, multiple times per second!

What we want to do with our coin is rotate it all the time at a certain speed. We don't need to perform any action on the first frame of a coin, so we'll delete the `Start` method code.

On the other hand, we do want to rotate it slightly on each frame. One way to do that is access the *transform* of the coin, which as we've seen contains it's position, scaling and rotation values. We'll start by defining a rotation speed, as a **public variable**, and give it a default value (more on this later).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CoinController : MonoBehaviour {

    public float rotationSpeed = 100f;

    // Update is called once per frame
    void Update () {

    }
}
```

A **public variable** is a property of the class that can be modified from outside the class. As we'll see shortly, by assigning this property as public we'll be able to modify it directly from the *Unity Editor.*

On Unity scripts we have access to an object `Time`, which has a property called `deltaTime` and provides the time in seconds since the last frame.

Basic physics stats that speed is equal to distance divided by time. This means, distance is equal to speed times time. We'll use that formula to determine how much the coin needs to rotate on each frame:

```
//distance (in angles) to rotate on each frame. distance = speed * time
float angle = rotationSpeed * Time.deltaTime;
```

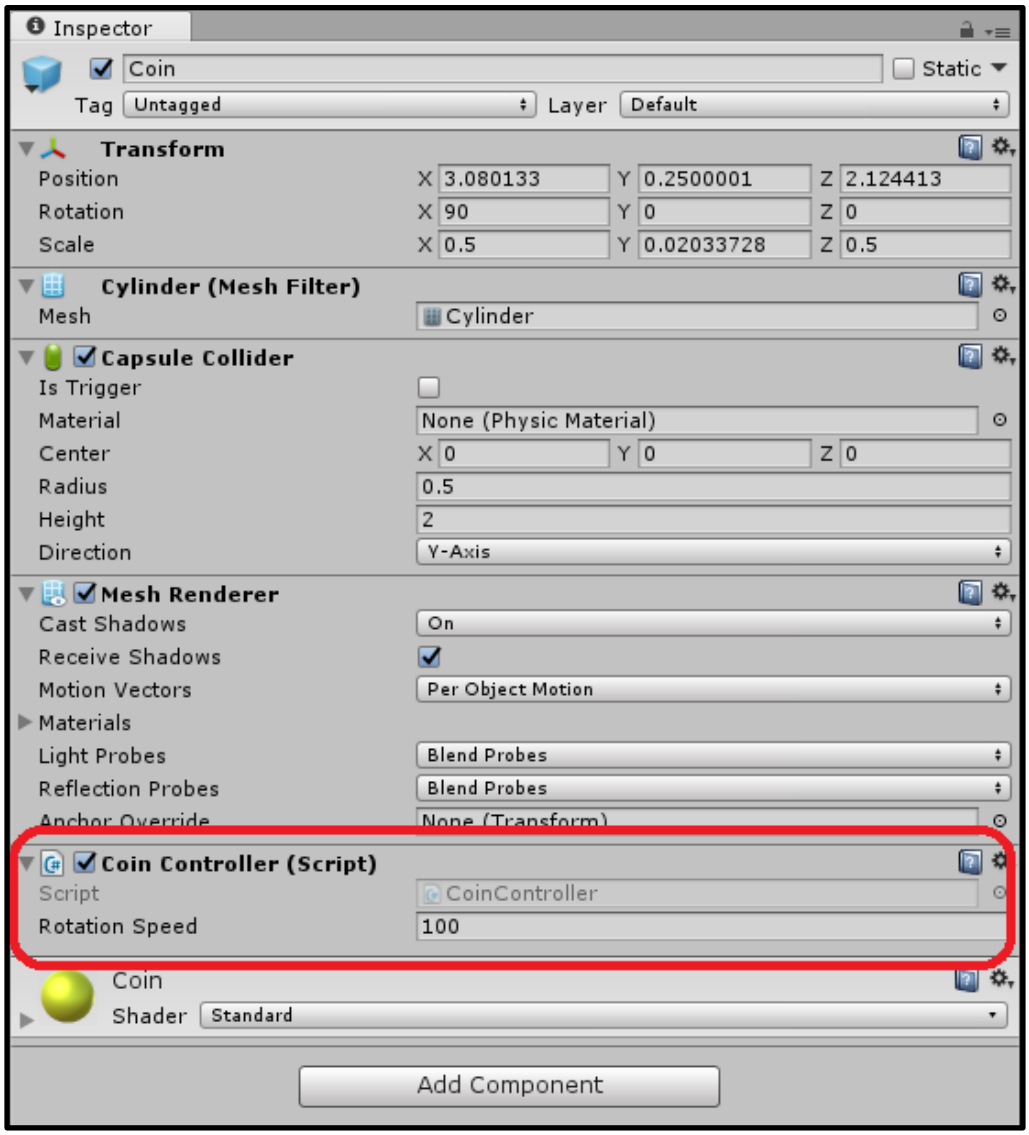
The rotation needs to be about the vertical axis (Y). We can access the object's *transform* and make it rotate as shown below:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CoinController : MonoBehaviour {

    public float rotationSpeed = 100f;

    // Update is called once per frame
    void Update()
    {
        //distance (in angles) to rotate on each frame. distance = speed * time
        float angle = rotationSpeed * Time.deltaTime;

        //rotate on Y
        transform.Rotate(Vector3.up * angle, Space.World);
    }
}
```

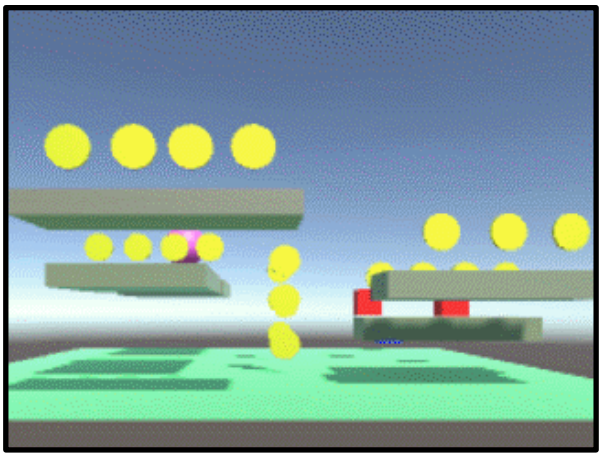
- `transform.Rotate` allows us to rotate the transform. For the full documentation see here.
- `Vector3.up` gives us the vertical axis. An alternative would be to create a new `Vector3` object like so: `new Vector3(0, 1, 0);`
- The last parameter `Space.World` needs to be specified in this case, and it means that the rotation needs to be to the "up" direction in the world, not relative to the object. Remember how we created our coins: we inserted a cylinder, reduced it in size then rotated it so that it would look like a coin. If you don't specify this parameter, the rotation will be applied on local coordinates, in which the Y axis is tilted.

Last but not least, we need to attach this script to our *coin prefab*. So far, Unity has no way of knowing that this script is to be used on coins!

Select your coin prefab (from the *Prefabs* folder). In the *Inspector* click on *Add Component,* select *Scripts – Coin Controller*. This action will attach the script to all the coins you have created. Notice how we can also change the rotation speed from here – all public properties show in the Inspector and can be edited directly from the *Unity Editor*. This is quite useful in case you are working with people who don't do coding but need to make changes to the game.



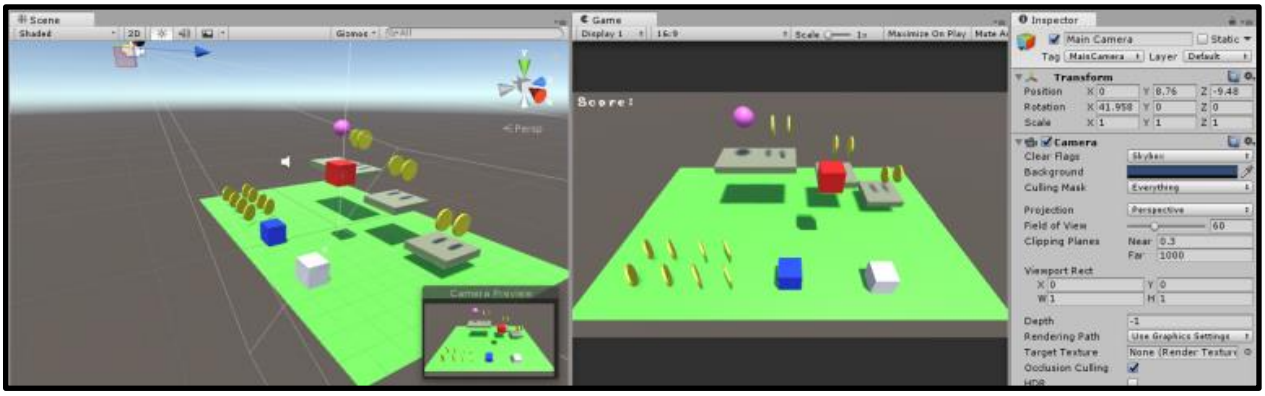
See it in action! Press the “play” icon on the *Toolbar* and see how the coins rotate in either the *Scene View* or the *Game View.*



Where did the "100" for rotation speed come from? The answer is: from trial and error. When making games you'll quite often need to set arbitrary values such as speeds, jumping distances, etc. What I usually do is throw in a number and adjust until it looks/feels good.

Player movement

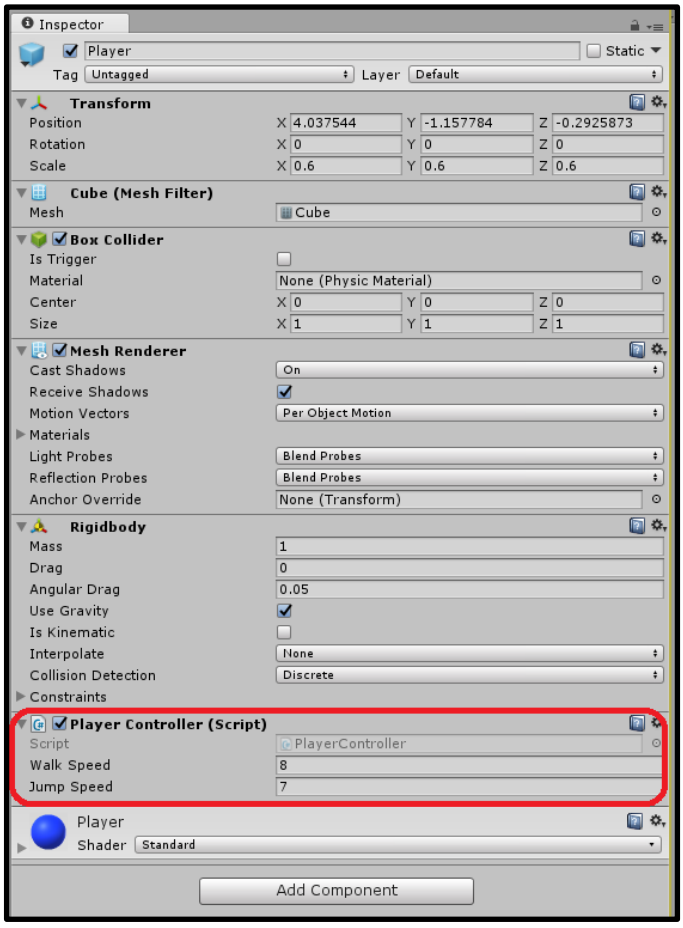
In this section we'll implement player movement using the arrow keys. Let's first place the camera in a position where the whole level can be seen easily. See below where I've placed mine (you can copy the values in Transform if you want). Feel free to manually adjust it.



Our player will be subject to physics laws: gravity, momentum, etc. For that, Unity provides a component named *Rigid Body*, which we need to assign to our Player prefab. In the Inspector click *Add Component*, select *Physics – Rigidbody*.

Create a new script in the *Scripts* folder and call it *PlayerController*. We'll start by adding some public properties for the walking and jumping speed of our player. Attach the script to the Player prefab as described in the previous section.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour {

    public float walkSpeed = 8f;
    public float jumpSpeed = 7f;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```



We'll begin our implementation of basic cursor keys movement by reading user input from the `Update` method. When it comes to player controllers, it's good practice in Unity to read *input axes* instead of specific keys. For example, when you press the "up" key, you activate the *vertical axis*. This axis would also be activated if you pressed the "up" key on a gamepad, or some other device. Moreover, people with certain disabilities can map their gamepads or joysticks in different ways. So it's always good practice to read "axes" instead of specific keys (unless, or course, you really need to read a certain key).

If you are curious, you can see the axes available if you go to the menu *Edit – Project Settings – Input*. We'll be using Horizontal and Vertical.

To be sure that we are reading the arrow keys correctly, add the following code to `Update`, which will show in the the Editor's Console (tab next to *Projects*) a message with the values we get:

```
1     // Update is called once per frame
2     void Update ()
3     {
4         // Input on x ("Horizontal")
5         float hAxis = Input.GetAxis("Horizontal");
6
7         print("Horizontal axis");
8         print(hAxis);
9
10        // Input on z ("Vertical")
11        float vAxis = Input.GetAxis("Vertical");
12
13        print("Vertical axis");
14        print(vAxis);
15    }
```

If you play the game and press the arrow keys you'll notice that `hAxis` shows values from -1 (left) to 1 (right). `vAxis` shows values from -1 (down) to 1 (up). The value of 0 is shown when no key is pressed.

Each time the input is read, we'll move a certain distance which can be calculated as speed * time. (remember: speed = distance / time, which means distance = speed * time).

We'll create a vector that points out where we are moving, based on our current position:

```
    // Update is called once per frame
    void Update ()
    {
        // Distance ( speed = distance / time --> distance = speed * time)
        float distance = walkSpeed * Time.deltaTime;

        // Input on x ("Horizontal")
        float hAxis = Input.GetAxis("Horizontal");

        // Input on z ("Vertical")
        float vAxis = Input.GetAxis("Vertical");

        // Movement vector
        Vector3 movement = new Vector3(hAxis * distance, 0f, vAxis * distance);

        // Current position
        Vector3 currPosition = transform.position;

        // New position
        Vector3 newPosition = currPosition + movement;
    }
```

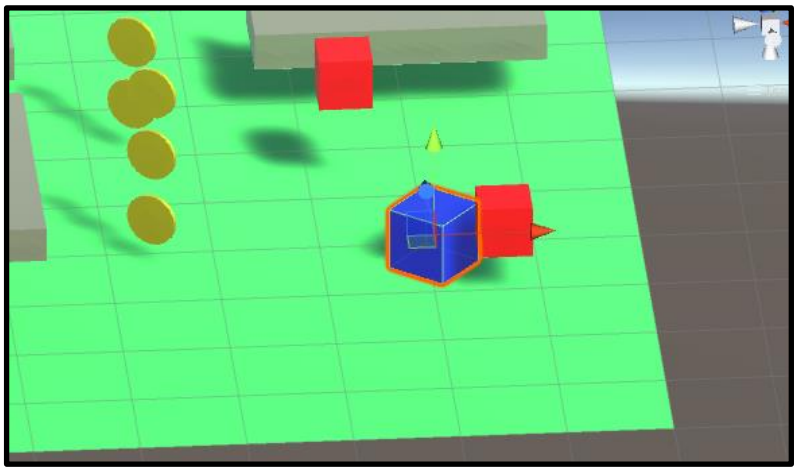
Now, how do we actually move our player? We need to access the player's *rigid body* in order to do that (yes, we could just move the *transform* like we did with the coins, but in this case we want to have an accurate physics simulation with velocity, gravity, etc, so we need to use the rigid body instead).

We need to create a variable to store our rigid body and put the Rigid Body component in it (at the start of our script, for which we can use the `Start` method), so that we can then access it for movement, jumping, and whatever else we need.

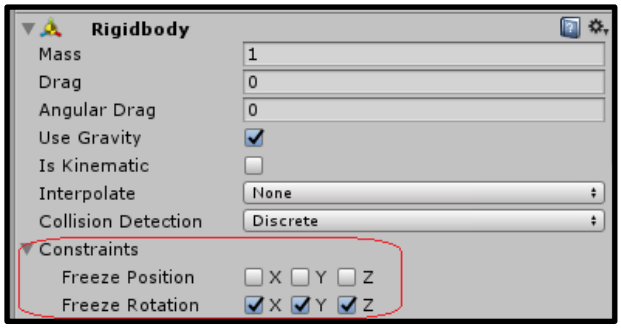
To make our code cleaner we'll put all the movement code in it's own function, named `WalkHandler`.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour {

    public float walkSpeed = 8f;
    public float jumpSpeed = 7f;

    //to keep our rigid body
    Rigidbody rb;

    // Use this for initialization
    void Start () {
        //get the rigid body component for later use
        rb = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    void Update ()
    {
        // Handle player walking
        WalkHandler();
    }

    // Make the player walk according to user input
    void WalkHandler()
    {
        // Set x and z velocities to zero
        rb.velocity = new Vector3(0, rb.velocity.y, 0);

        // Distance ( speed = distance / time --> distance = speed * time)
        float distance = walkSpeed * Time.deltaTime;

        // Input on x ("Horizontal")
        float hAxis = Input.GetAxis("Horizontal");

        // Input on z ("Vertical")
        float vAxis = Input.GetAxis("Vertical");

        // Movement vector
        Vector3 movement = new Vector3(hAxis * distance, 0f, vAxis * distance);

        // Current position
        Vector3 currPosition = transform.position;

        // New position
        Vector3 newPosition = currPosition + movement;

        // Move the rigid body
        rb.MovePosition(newPosition);
    }
}
```

You can now move around! But as you crash against an enemy or a platform you'll notice that the player rotates after the collision:



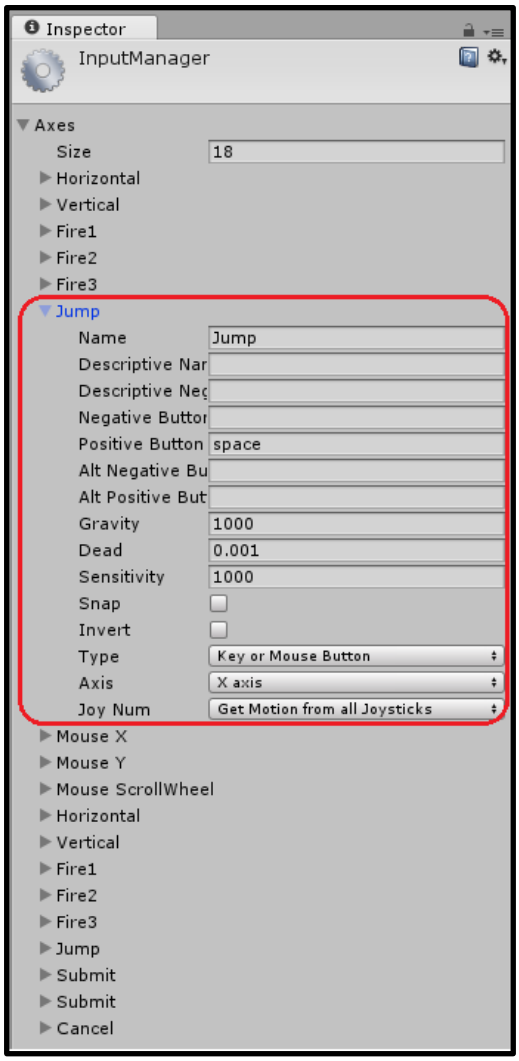
Our player is represented by a rigid body which simulates real physics. If you put a dice on a surface and hit it on a side, it will of course rotate! What we need to do is disable our player from rotating, which can be easily done from the *Inspector*. Find the *Rigid Body* component of the player prefab, under Constraints go and check *Freeze Rotation* for all axes.



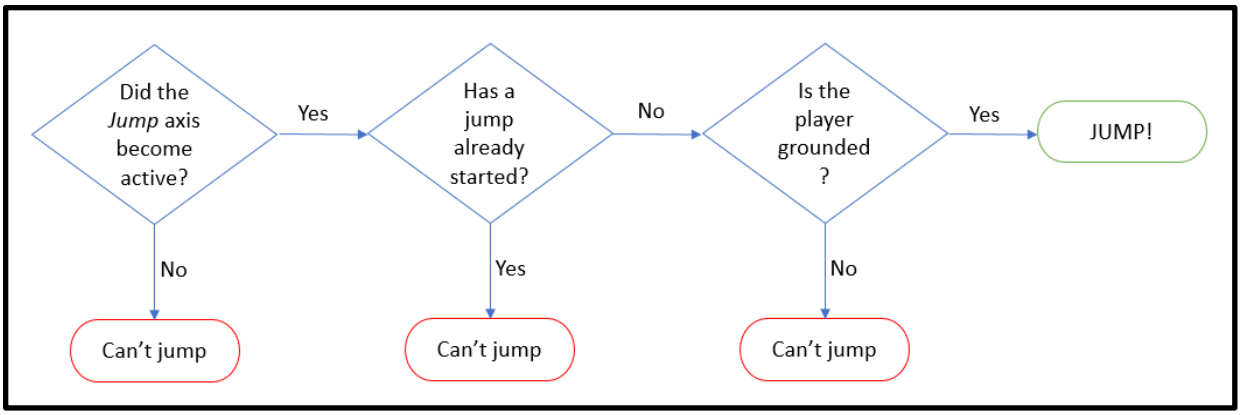
Player jumping

We can now move our player around the game with the arrow keys, and it's time to give it the ability to jump. Implementing proper jumping logic will take some thought, but we'll go step by step covering all that it entails.

Unity comes with an Input Axis called "Jump", which activates with the spacebar by default. Go to *Edit – Project Settings – Input* to double check it's there on your end.



When should the player be allowed to jump? Should it be allowed to jump while it's already in the air? You start to realize that there are some rules around when the player should be allowed to jump. I've put these together in the following diagram:



Let's begin by creating a function to take care of all the jumping logic. We'll call that on `Update`.

```
1     // Update is called once per frame
2     void Update ()
3     {
4         // Handle player walking
5         WalkHandler();
6
7         //Handle player jumping
8         JumpHandler();
9     }
```

If we just check that the *Jump* axis has been pressed, we can make our player jump like so:

```
1     // Check whether the player can jump and make it jump
2     void JumpHandler()
3     {
4         // Jump axis
5         float jAxis = Input.GetAxis("Jump");
6
7         if (jAxis > 0f)
8         {
9             // Jumping vector
10            Vector3 jumpVector = new Vector3(0f, jumpSpeed, 0f);
11
12            // Make the player jump by adding velocity
13            rb.velocity = rb.velocity + jumpVector;
14        }
15    }
```

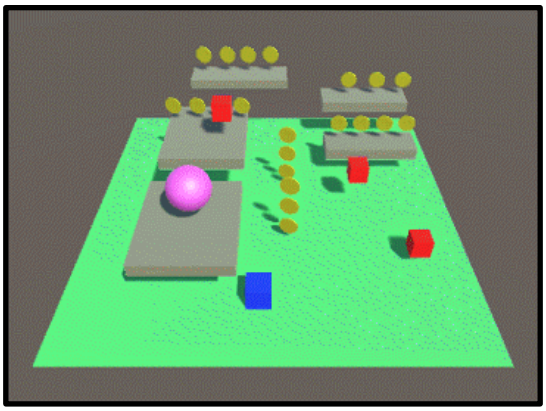
Of course, this is an incomplete implementation, as we are not checking whether the player is grounded or not. Also, if you keep the spacebar pressed, the player will fly away!

We'll now make sure that we can't jump more than once on the same key press. We can do that by using a boolean variable which we'll use as a "flag" to indicate whether we've jumped on the current keypress or not.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour {

    public float walkSpeed = 8f;
    public float jumpSpeed = 7f;

    //to keep our rigid body
    Rigidbody rb;

    //to keep the collider object
    Collider coll;

    //flag to keep track of whether a jump started
    bool pressedJump = false;

    // Use this for initialization
    void Start () {
        //get the rigid body component for later use
        rb = GetComponent<Rigidbody>();

        //get the player collider
        coll = GetComponent<Collider>();
    }

    // Update is called once per frame
    void Update ()
    {
        // Handle player walking
        WalkHandler();

        //Handle player jumping
        JumpHandler();
    }

    // Make the player walk according to user input
    void WalkHandler()
    {
        // Set x and z velocities to zero
        rb.velocity = new Vector3(0, rb.velocity.y, 0);

        // Distance ( speed = distance / time --> distance = speed * time)
        float distance = walkSpeed * Time.deltaTime;

        // Input on x ("Horizontal")
        float hAxis = Input.GetAxis("Horizontal");

        // Input on z ("Vertical")
        float vAxis = Input.GetAxis("Vertical");

        // Movement vector
        Vector3 movement = new Vector3(hAxis * distance, 0f, vAxis * distance);

        // Current position
        Vector3 currPosition = transform.position;

        // New position
        Vector3 newPosition = currPosition + movement;

        // Move the rigid body
        rb.MovePosition(newPosition);
    }
```

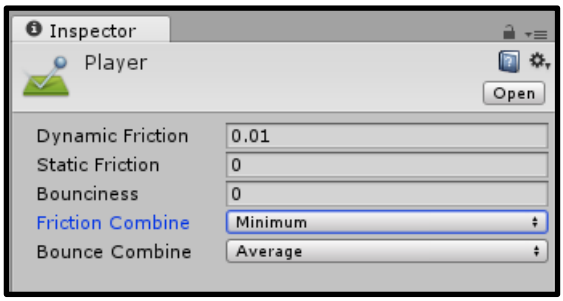
```
65
66      // Check whether the player can jump and make it jump
67      void JumpHandler()
68      {
69          // Jump axis
70          float jAxis = Input.GetAxis("Jump");
71
72          // Is grounded
73          bool isGrounded = CheckGrounded();
74
75          // Check if the player is pressing the jump key
76          if (jAxis > 0f)
77          {
78              // Make sure we've not already jumped on this key press
79              if(!pressedJump && isGrounded)
80              {
81                  // We are jumping on the current key press
82                  pressedJump = true;
83
84                  // Jumping vector
85                  Vector3 jumpVector = new Vector3(0f, jumpSpeed, 0f);
86
87                  // Make the player jump by adding velocity
88                  rb.velocity = rb.velocity + jumpVector;
89              }
90          }
91          else
92          {
93              // Update flag so it can jump again if we press the jump key
94              pressedJump = false;
95          }
96      }
97
98      // Check if the object is grounded
99      bool CheckGrounded()
100     {
101         // Object size in x
102         float sizeX = coll.bounds.size.x;
103         float sizeZ = coll.bounds.size.z;
104         float sizeY = coll.bounds.size.y;
105
106         // Position of the 4 bottom corners of the game object
107         // We add 0.01 in Y so that there is some distance between the point and the floor
108         Vector3 corner1 = transform.position + new Vector3(sizeX/2, -sizeY / 2 + 0.01f, sizeZ / 2);
109         Vector3 corner2 = transform.position + new Vector3(-sizeX / 2, -sizeY / 2 + 0.01f, sizeZ / 2);
110         Vector3 corner3 = transform.position + new Vector3(sizeX / 2, -sizeY / 2 + 0.01f, -sizeZ / 2);
111         Vector3 corner4 = transform.position + new Vector3(-sizeX / 2, -sizeY / 2 + 0.01f, -sizeZ / 2);
112
113         // Send a short ray down the cube on all 4 corners to detect ground
114         bool grounded1 = Physics.Raycast(corner1, new Vector3(0, -1, 0), 0.01f);
115         bool grounded2 = Physics.Raycast(corner2, new Vector3(0, -1, 0), 0.01f);
116         bool grounded3 = Physics.Raycast(corner3, new Vector3(0, -1, 0), 0.01f);
117         bool grounded4 = Physics.Raycast(corner4, new Vector3(0, -1, 0), 0.01f);
118
119         // If any corner is grounded, the object is grounded
120         return (grounded1 || grounded2 || grounded3 || grounded4);
121     }
122 }
```

Is it really over? We'll, there is just one last thing. See what happens if you jump against a wall and keep on pushing towards that direction. Yep, you'll get stuck on the wall.

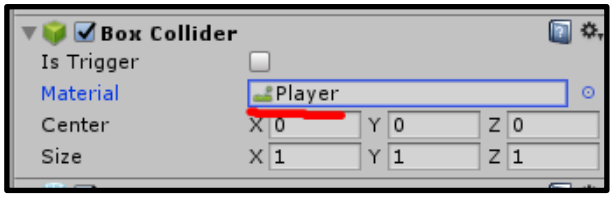


This is caused by friction. This can be disabled so that our player doesn't rotate by creating a **physics material**. Physics materials allow us to give our game object custom physical properties such as friction, bounciness, and how it will behave when colliding with other objects. In your Assets folder, create a subfolder called *Physics Materials*, and inside of that folder right click and select *Create – Physics Material*, name it *Player*.

Give this new material properties as shown below, so that we don't have any friction. Make sure to select *Minimum* in *Friction Combine*. This means that no matter what the friction is of the colliding object, the minimum value will be used for physics calculations.



Select your Player prefab, in the Inspector look for the *Box Collider* component and click in *Material*. Select the physics material you just created in order to assign it to the prefab.



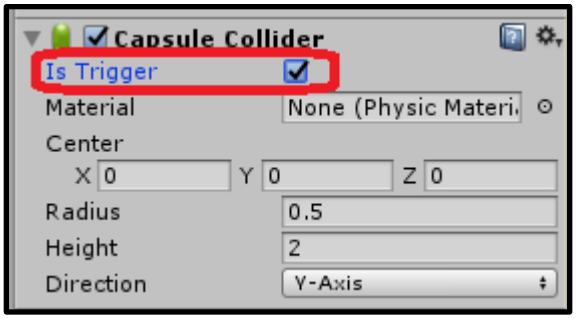
Now we are finally done with player jumping

Collecting coins

In this section we'll implement coin collection. When the player collects a coin, a sound will be played. In the next section, we'll keep track of the score of the player in an object we'll call *GameManager*, which will keep track of game-level information (score, high score, level).

Currently, if you touch a coin you'll notice that they actually stop the player, they are "solid". This is not the behavior we want. We want to be able to detect a collision between the player an a coin, however, we don't want coins to affect the velocity, or any physics property, of our player.

We need to make our coins a *trigger* collider. When another object collides with a coins, a *trigger event* will be fired, and the physics properties of the object won't be affected, just as if you were going through thin air! Select the coin prefab, find the component *Capsure Collider* in the *Inspector*, and check *Is Trigger*.



Do the same for the *enemy prefab* and *goal prefab*. We can now jump through coins. We need to know "collect" these coins on collision, play a sound and destroy the coins after collected. In our player controller script, we can use a method named `OnTriggerEnter` (learn more about it here). This method will be called each time the player runs into a *trigger collider*. The following code outlines the steps of what we'll do next. In this section, we'll only take care of playing the sound and destroying the coin:

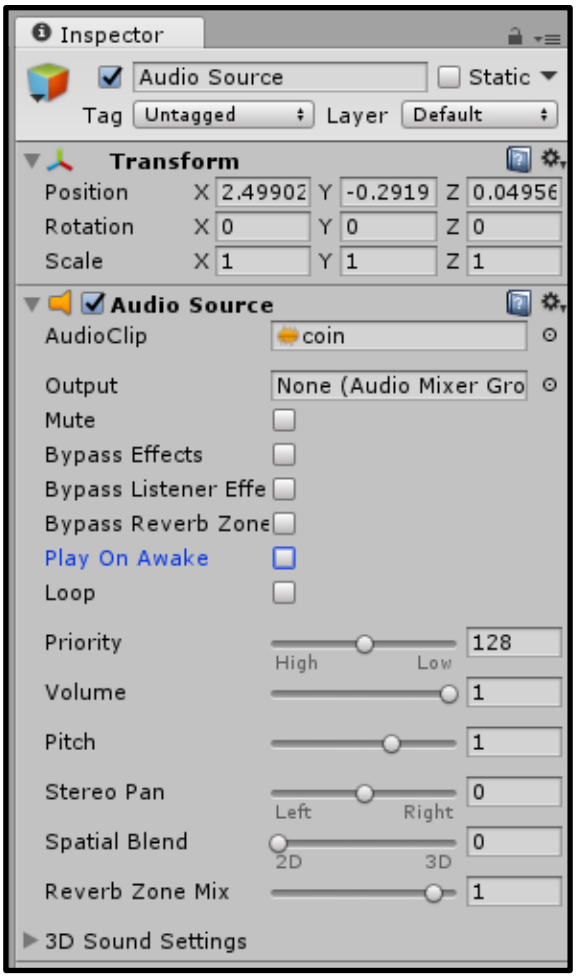
```
1 void OnTriggerEnter(Collider collider)
2 {
3     //detect that we collided with a coin, if that is the case:
4     // - play a coin collecting sound
5     // - update the score
6     // - destroy the coin
7 }
```

To detect whether the trigger object the player ran into is a coin we can give our coin prefab a *tag* that identifies it. Select the coin prefab and in the *Inspector*, in *Tag go* and select *Add Tag*. Create a "Coin" tag. Also, create "Goal" and "Enemy" as we'll use those later. After creating the tags, make sure the coin prefab has the *Coin* tag in *Tag* on the *Inspector* (pick it from the list). Do the same for the enemy and goal prefabs.

We can check the tag of the object we've collided with, and destroy the coin we've collected by doing:

```
1     void OnTriggerEnter(Collider collider)
2     {
3         // Check if we ran into a coin
4         if (collider.gameObject.tag == "Coin")
5         {
6             print("Grabbing coin..");
7
8             // Destroy coin
9             Destroy(collider.gameObject);
10        }
11    }
```

Let's now take care of the coin sound. Create a folder in *Assets* called *Audio*, copy the coin.ogg file provided with the tutorial source code (download at the start of the tutorial or here). In the *Hierarchy Window*, right click and select *Audio – Audio Source*, rename it to "Coin Audio Source". Select the newly created object. In the *Inspector*, drag the coin.ogg file to *AudioClip*. Uncheck the box *Play On Awake* so that the sound doesn't play each time the game starts.



We need to know be able to pass this audio source to our player, so that it can be played each time a coin is collected.

For that, we can create a public variable in our player controller, of type `AudioSource`, and pass our audio source object to it via the *Inspector*. Add the following at the start of your player controller:

```
1 public AudioSource coinAudioSource;
```

Now drag and drop the coin audio source object into the *Coin Audio Source* field in the player prefab's *Inspector*, Script component.

In your player controller file, we can now play the file in `OnTriggerEnter` :

```
1   void OnTriggerEnter(Collider collider)
2   {
3       // Check if we ran into a coin
4       if (collider.gameObject.tag == "Coin")
5       {
6           print("Grabbing coin..");
7
8           // Play coin collection sound
9           coinAudioSource.Play();
10
11          // Destroy coin
12          Destroy(collider.gameObject);
13      }
14  }
```

Game Manager

Most games have some sort of high-level parameters and operations, such as resetting the game, managing game over, number of lives, current level, options, etc. In our game for instance, we want to keep track of the player score, the high score, and the level we are in.

To keep track of these things, we can create an object that keeps track of these things and helps manage it it all. We'll call this object the *Game Manager*.

Something we know for sure about this *Game Manager* object is that we'll only want a single instance of this object to exist at the same time. In computer science terminology, we want this object to be a singleton.

Create a script and call it *GameManager*. The following code will help us keep track of the score.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour {

    // Player score
    public int score = 0;

    // Increase score
    public void IncreaseScore(int amount)
    {
        // Increase the score by the given amount
        score += amount;
    }
}
```

To actually have this script in our game, create an empty game object in the *Hierarchy Window*, name it “Game Manager”, and assign our script to it.

We now need to access this object and its method `IncreaseScore` in our player controller in order to increase the score every time we collect a coin. One way to do this is to repeat the steps we took when attaching the coin audio source to our player (creating a “gameManager” public property and dragging the “Game Manager” object to the player’s *Inspector*). That works and there is nothing wrong in doing that.

Since we are talking about the *game manager* here, it’s highly likely that we’ll want to access it from many game objects, not just the player. Having to drag and drop it each time can become time consuming. A different approach is to use a static variable that can be accessed from anywhere in the code, without having to declare public variables each time and drag and drop elements. This will also help us enforce the fact that there will only be a single instance of the Game Manager in our game (the *singleton* pattern).

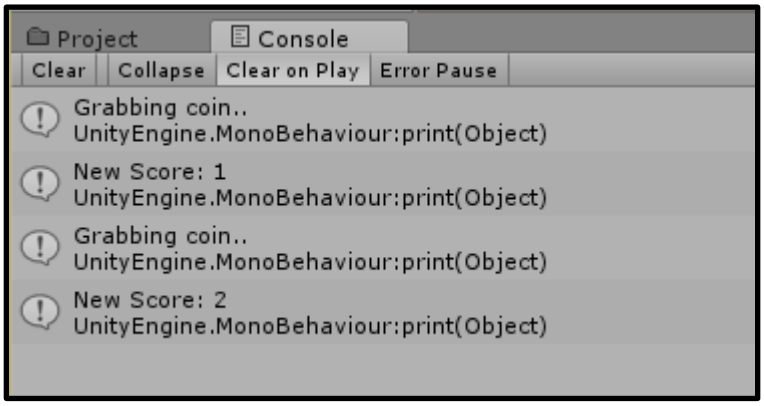
The following implementation will provide all of the above. Plus, when we load different scenes (game over scene, or other levels) the Game Manager is not destroyed, which would happen by default. For more explanations see this official tutorial and this tutorial on game manager.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour {

    // Static instance of the Game Manager,
    // can be access from anywhere
    public static GameManager instance = null;

    // Player score
    public int score = 0;

    // Called when the object is initialized
    void Awake()
    {
        // if it doesn't exist
        if(instance == null)
        {
            // Set the instance to the current object (this)
            instance = this;
        }

        // There can only be a single instance of the game manager
        else if(instance != this)
        {
            // Destroy the current object, so there is just one manager
            Destroy(gameObject);
        }

        // Don't destroy this object when loading scenes
        DontDestroyOnLoad(gameObject);
    }

    // Increase score
    public void IncreaseScore(int amount)
    {
        // Increase the score by the given amount
        score += amount;

        // Show the new score in the console
        print("New Score: " + score.ToString());
    }
}
```

In our player, we can increase the score like so:

```
    void OnTriggerEnter(Collider collider)
    {
        // Check if we ran into a coin
        if (collider.gameObject.tag == "Coin")
        {
            print("Grabbing coin..");

            // Increase score
            GameManager.instance.IncreaseScore(1);

            // Play coin collection sound
            coinAudioSource.Play();

            // Destroy coin
            Destroy(collider.gameObject);
        }
    }
```

We can now collect coins and see the increased score in the *Console*!



What about keeping high score? That will be be quite straightforward and can be done directly in *GameManager*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour {

    // Static instance of the Game Manager,
    // can be access from anywhere
    public static GameManager instance = null;

    // Player score
    public int score = 0;

    // High score
    public int highScore = 0;

    // Called when the object is initialized
    void Awake()
    {
        // if it doesn't exist
        if(instance == null)
        {
            // Set the instance to the current object (this)
            instance = this;
        }

        // There can only be a single instance of the game manager
        else if(instance != this)
        {
            // Destroy the current object, so there is just one manager
            Destroy(gameObject);
        }

        // Don't destroy this object when loading scenes
        DontDestroyOnLoad(gameObject);
    }

    // Increase score
    public void IncreaseScore(int amount)
    {
        // Increase the score by the given amount
        score += amount;

        // Show the new score in the console
        print("New Score: " + score.ToString());

        if (score > highScore)
        {
            highScore = score;
            print("New high score: " + highScore);
        }
    }
}
```

Of course, for now our high score updates just like our score. This will be fully finished once we implement *game over*.

Enemy movement

Enemies will have an up-and-down movement within a range. We'll make it so that you can easily change the speed, initial direction and movement range. In the tutorial we'll only implement movement in Y, but you can easily modify this code to have enemies moving in other directions too!

Start by creating a new script which we'll call "EnemyController". Attach it to the enemy prefab. We'll add some public variables, and also keep the initial position so that we can move around it.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyController : MonoBehaviour {

    // Range of movement
    public float rangeY = 2f;

    // Speed
    public float speed = 3f;

    // Initial direction
    public float direction = 1f;

    // To keep the initial position
    Vector3 initialPosition;

    // Use this for initialization
    void Start () {

        // Initial location in Y
        initialPosition = transform.position;
    }

    // Update is called once per frame
    void Update () {

    }
}
```

In each frame, we'll do the following:

- Calculate how much we are moving (distance = elapsed time * speed * direction).
- Calculate the new Y position (new position = old position + distance)
- If we've passed the movement range, change direction

The code of `Update` will then be:

```
    // Update is called once per frame
    void Update()
    {
        // How much we are moving
        float movementY = direction * speed * Time.deltaTime;

        // New position
        float newY = transform.position.y + movementY;

        // Check whether the limit would be passed
        if (Mathf.Abs(newY - initialPosition.y) > rangeY)
        {
            // Move the other way
            direction *= -1;
        }

        // If it can move further, move
        else
        {
            // Move the object
            transform.Translate(new Vector3(0, movementY, 0));
        }
    }
```

Feel free to adjust the range, speed and direction of each one of your individual enemies if you are not happy with the default values!

To detect enemy collision (for game over purposes!), add the following to your player controller's `OnTriggerEnter` method, after the *if statement* where we check for the coin tag (this assumes you've checked *Is Trigger* on the enemy prefab):

```
        else if(collider.gameObject.tag == "Enemy")
        {
            // Game over
            print("game over");

            // Soon.. go to the game over scene
        }
```

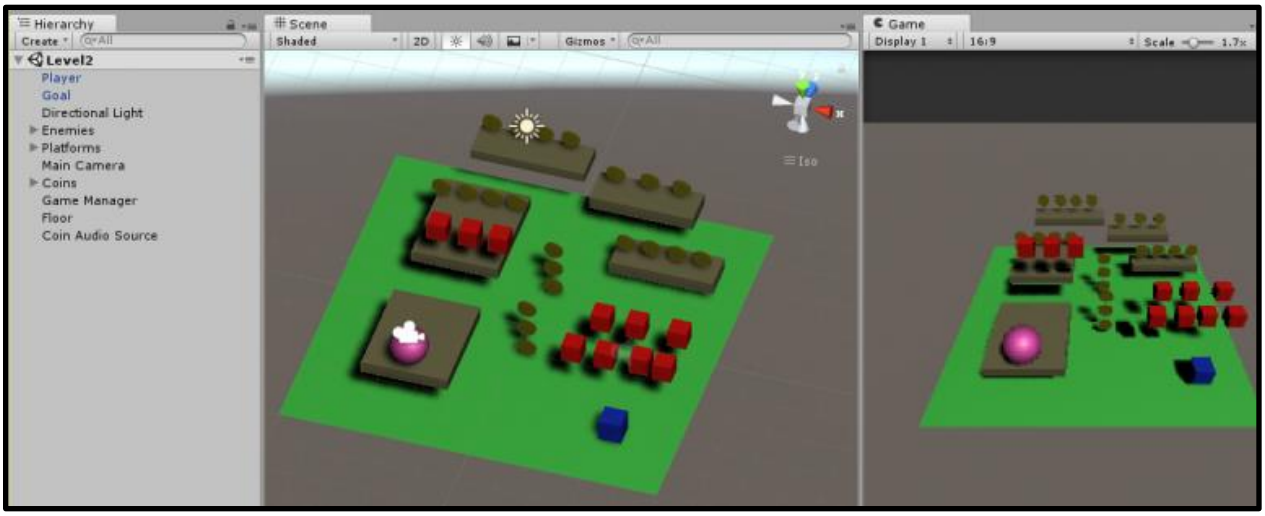
Multi-level game

In this section we'll implement multi-level functionality. Each level will have it's own scene, and we'll use the *Game Manager* to keep track of the current level, the number of levels, and the level loading process.

Go to your *Scenes* folder (some people like to name this folder *_Scenes* so that it shows first on the folder list), rename the scene we've been using to *Level1*. Create a new scene by right clicking and selecting *Create* – *Scene*. Name it *Level2*.

If you double click on the new scene, you will see that you are taken to a blank scene. The easiest way to start a new level is to "copy and paste" the objects from the Hierarchy Window of Level1, into Level2. In Level1, simply select the objects you want, right click and pick "copy". In Level2 right click in and pick "paste".

If you double click on the new scene, you will see that you are taken to a blank scene. The easiest way to start a new level is to "copy and paste" the objects from the *Hierarchy Window* of *Level1*, into *Level2*. In *Level1*, simply select the objects you want, right click and pick "copy". In *Level2* right click in and pick "paste".



Lets modify *GameManager* so that it can take us from one level to the next. I've only implemented two levels for this tutorial, but the code is completely genetic – it can be used for hundreds of levels if you so want!

We'll specify the starting level, and the highest level we have in our game. If we pass that level, we'll send the user back to level 1. Let's also add a method to restart the game (used for game over).

To change Unity scenes, we need to import the `Unity.SceneManagement` package.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour {

    // Static instance of the Game Manager,
    // can be access from anywhere
    public static GameManager instance = null;

    // Player score
    public int score = 0;

    // High score
    public int highScore = 0;

    // Level, starting in level 1
    public int currentLevel = 1;

    // Highest level available in the game
    public int highestLevel = 2;

    // Called when the object is initialized
    void Awake()
    {
        // if it doesn't exist
        if(instance == null)
        {
            // Set the instance to the current object (this)
            instance = this;
        }

        // There can only be a single instance of the game manager
        else if(instance != this)
        {
            // Destroy the current object, so there is just one manager
            Destroy(gameObject);
        }

        // Don't destroy this object when loading scenes
        DontDestroyOnLoad(gameObject);
    }

    // Increase score
    public void IncreaseScore(int amount)
    {
        // Increase the score by the given amount
        score += amount;

        // Show the new score in the console
        print("New Score: " + score.ToString());

        if (score > highScore)
        {
            highScore = score;
            print("New high score: " + highScore);
        }
    }
```

```
60
61      // Restart game. Refresh previous score and send back to level 1
62      public void Reset()
63      {
64          // Reset the score
65          score = 0;
66
67          // Set the current level to 1
68          currentLevel = 1;
69
70          // Load corresponding scene (level 1 or "splash screen" scene)
71          SceneManager.LoadScene("Level" + currentLevel);
72      }
73
74      // Go to the next level
75      public void IncreaseLevel()
76      {
77          if (currentLevel < highestLevel)
78          {
79              currentLevel++;
80          }
81          else
82          {
83              currentLevel = 1;
84          }
85          SceneManager.LoadScene("Level" + currentLevel);
86      }
87  }
```

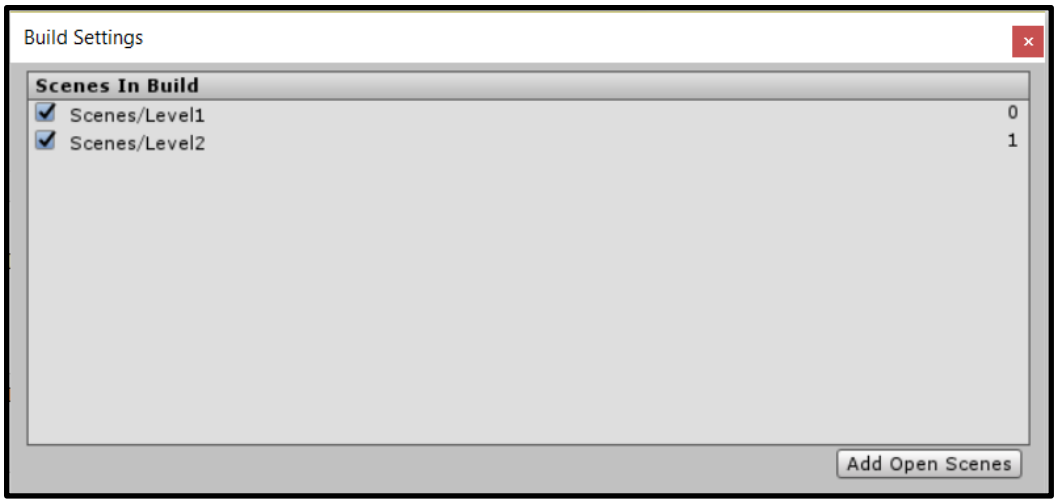
To detect when the level goal is reached. Provided you've checked Is *Trigger* on the goal prefab:

```
1           else if(collider.gameObject.tag == "Goal")
2           {
3               // Next level
4               print("next level");
5           }
```

If you try playing the game now and you reach the level goal, you'll see an error message in the console that will say something in the lines of:

```
1 Scene 'Level1' couldn't be loaded because it has not been added to the build settings or the AssetBundle has not been loaded.
```

To add a scene to the build settings use the menu *File->Build Settings* and add your scenes to the list (you might have to click Add Open Scenes for each scene).

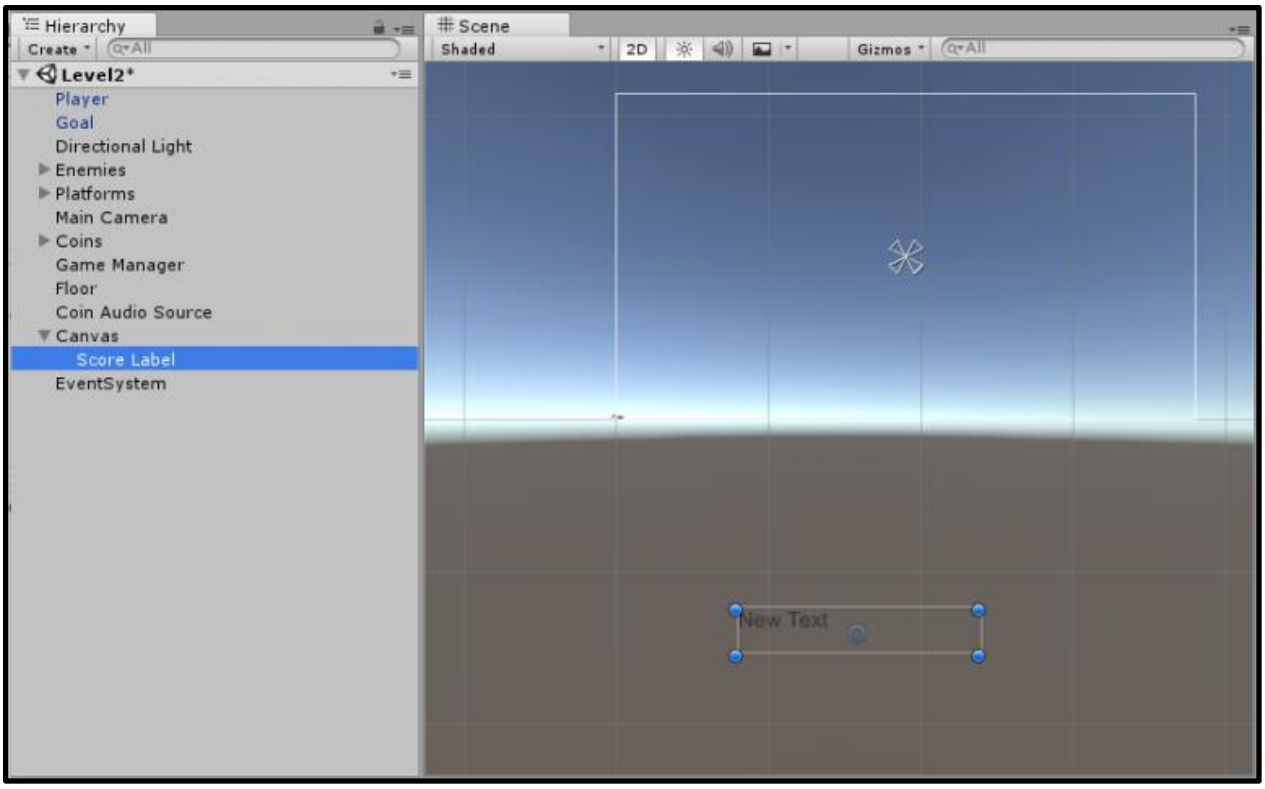


We can now play our game and move from one scene to the next! Notice how the score stays in between scenes. Basically, our *GameManager* object survives between scenes because of the statement `DontDestroyOnLoad(gameObject)` (and of course, the rest of the Game Manager implementation).

Adding the HUD

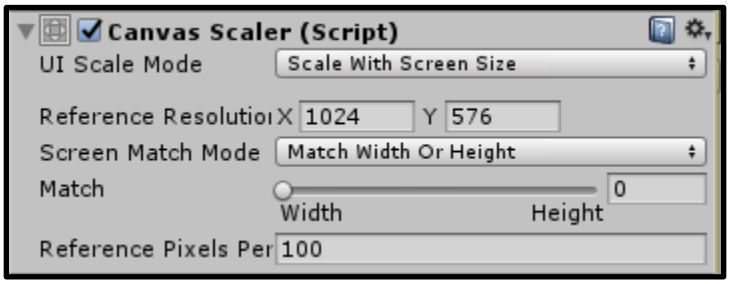
So far we have no idea what our score is. We need to show it to the player. Let's implement a very simple HUD where we show the current score.

Start by adding a text element in the *Hierarchy Window*. Right-click, *UI – Text.* This will create a *Canvas* element, with a *Text* element inside. Rename the text element to "Score Label". In the Scene View, click 2D, select the text and press f. This will show you the canvas location of the text label.



When adding UI elements, a canvas is created automatically. A canvas in Unity is a game object where all UI elements should be located. Some UI elements such as buttons handle user input, for which the canvas provides event system support, for which an *EventSystem* object was automatically created in our *Hierarchy Window*, as you can probably see. You can read more about the canvas in the docs.

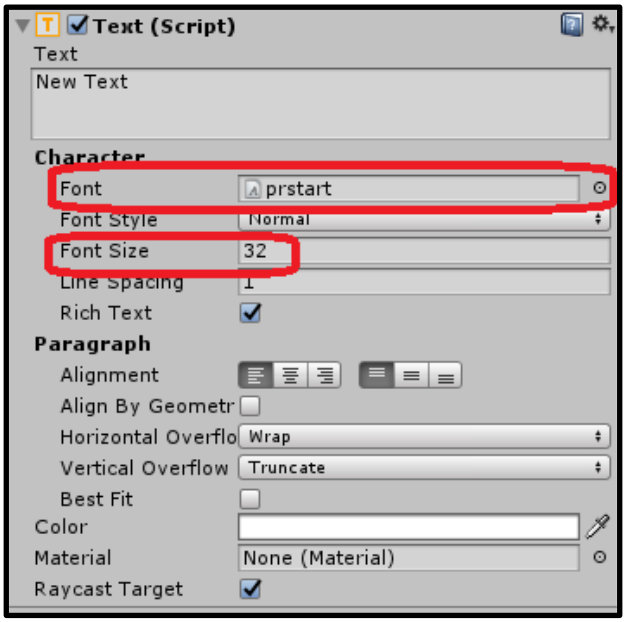
Drag the text label to the upper-left area of the canvas. Select the canvas object and find the *Canvas Scaler* component in the Inspector. *Change UI Scale Mode* to *Scale with Screen Size* so that the whole canvas scales up or down depending on the screen size. In this manner, the text label will always stay in that relative position. Set the *Reference Resolution* to 1024 x 576.



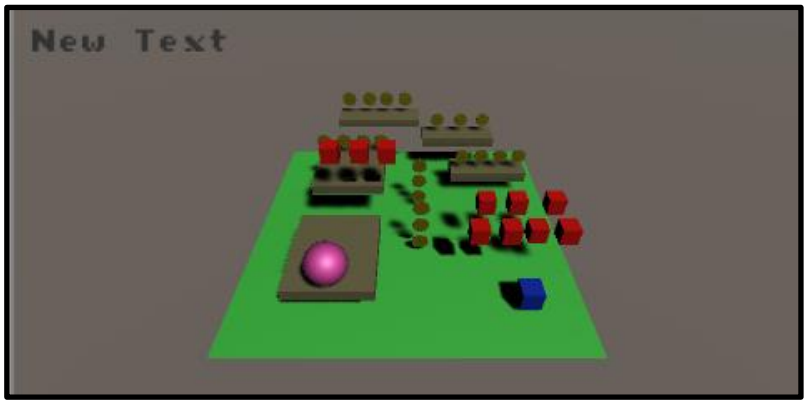
We'll now import a much nicer pixel art font to use called *Press Start.* Create a folder called "Fonts". Assuming you've downloaded the tutorial files, copy the files *prstart.ttf* and *license.txt*

(both located in *Assets/Fonts*) into your *Fonts* folder. Make sure to read the license file which covers how you are allowed to use this font.

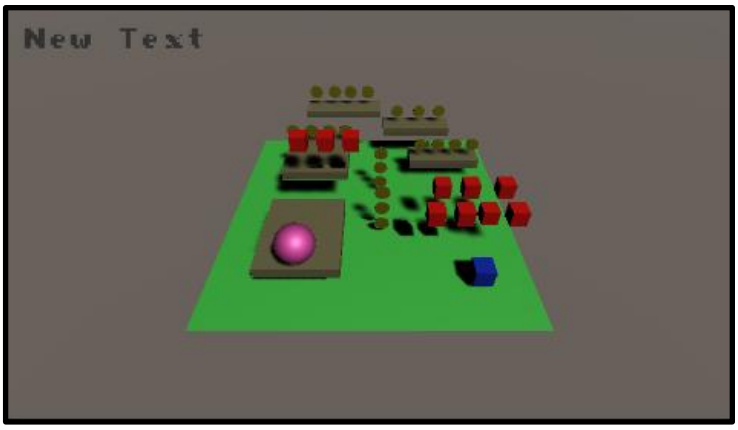
Now, if you select *Score Label*, find the *Font* field and click on the circle next to it, you'll be able to select our *Press Start font*. Set the font size to 32.



We can now see our text in this pixel font! However, it does look a bit blurry:



This can be easily fixed by making a change in the *Import Settings* of the font. Go to your *Fonts* folder, select the font file, and in the Inspector, change *Rendering Mode* to *Hinted Raster*. Press apply and you are pixel crispy good!



Before we move on to the HUD functionality, feel free to change the font color or style (I'm changing it to white).

We'll create a new object that will take care of managing the HUD. Create an empty object called *Hud Manager*. Create a new script and name it *HudManager*. Drag this new script onto the *Hud Manager* object in the *Hierarchy View*.

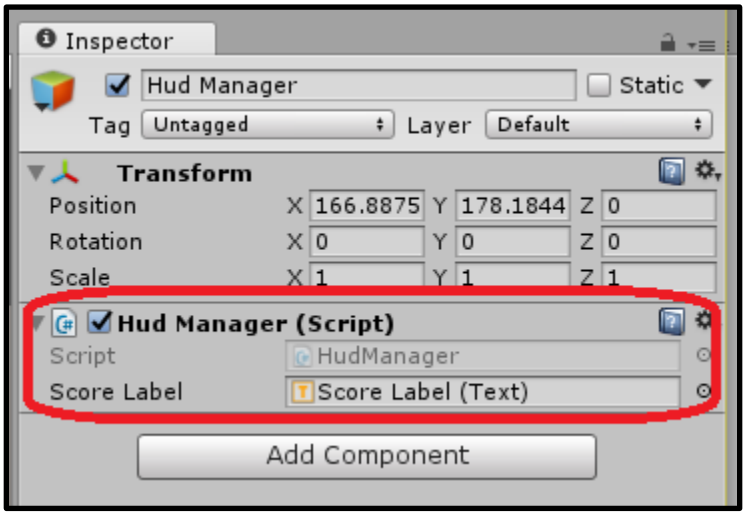
The HudManager script needs to be able to access the text label we created, so that it can change it's contents. This will be added as a public variable. We'll give it a public method called *Refresh*, so that we can trigger a HUD update from anywhere in our code.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  public class HudManager : MonoBehaviour {
7
8      public Text scoreLabel;
9
10     // Use this for initialization
11     void Start()
12     {
13         Refresh();
14     }
15
16     // Show player stats in the HUD
17     public void Refresh()
18     {
19         scoreLabel.text = "Score: " + GameManager.instance.score;
20     }
21 }
```

- We start by importing the *UnityEngine.UI* namespace so that we can easily access UI-related functionality

- Create a public variable of type UnityEngine.UI.Text for our score label.
- We'll refresh the HUD on the first frame of the game, to show the initial score.
- The *Refresh* method modifies the actual text content of the score field. Notice how we can easily access the current score in our game from the *GameManager*.

Make sure to drag the *Score Label* object in your *Hierarchy Window* to the *Hud Manager* object, so that it knows which text element to modify!



The last bit missing is to actually call this *Refresh* method in our game! We'll do this in our *PlayerController*. Let's refresh the HUD at the start of the game (on the first frame), and every time the player gets a coin.

In order to be able to access the *HudManager* from our *PlayerController*, we'll add a public variable:

```
1 // access the HUD
2 public HudManager hud;
```

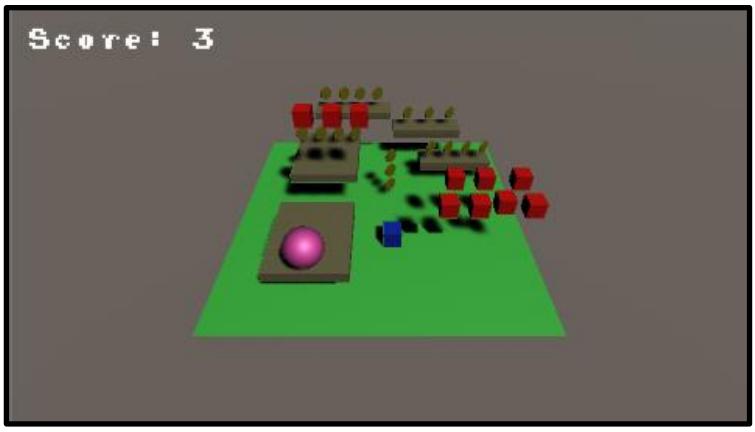
Let's refresh the HUD upon first frame:

```
    // Use this for initialization
    void Start()
    {
        //get the rigid body component for later use
        rb = GetComponent<Rigidbody>();

        //get the player collider
        coll = GetComponent<Collider>();

        //refresh the HUD
        hud.Refresh();
    }
```

And every time we collect a coin:

```
        // Check if we ran into a coin
        if (collider.gameObject.tag == "Coin")
        {
            print("Grabbing coin..");

            // Increase score
            GameManager.instance.IncreaseScore(1);

            //refresh the HUD
            hud.Refresh();

            // Play coin collection sound
            coinAudioSource.Play();

            // Destroy coin
            Destroy(collider.gameObject);
        }
```

Make sure to drag and drop the *Hud Manager* object onto our *Player* object. To make this work multilevel, simply copy these new objects (canvas, HUD manager) into each level, and make the corresponding dragging and dropping so it all wires up.

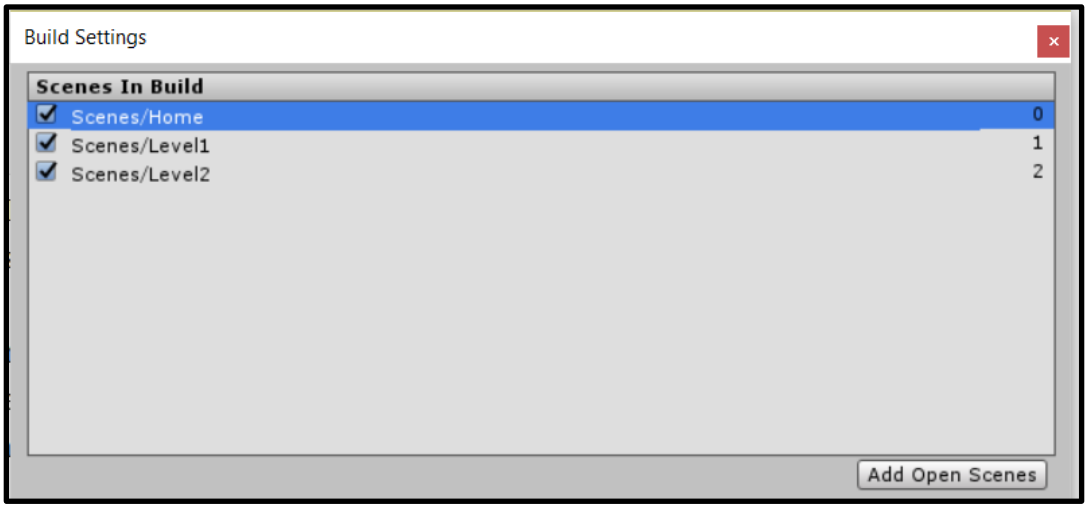


Note: you could also take a different approach and incorporate HUD management into the *Game Manager*, or create similar type of object. There is no single correct way and this seemed to be the simplest approach for our game.

Note 2: if you are an advanced programmer and don't like doing so much "dragging and dropping" I'd recommend reading the Dependency Injection series by Ashley Davis. Keep in mind the approach presented there is definitely more advanced.

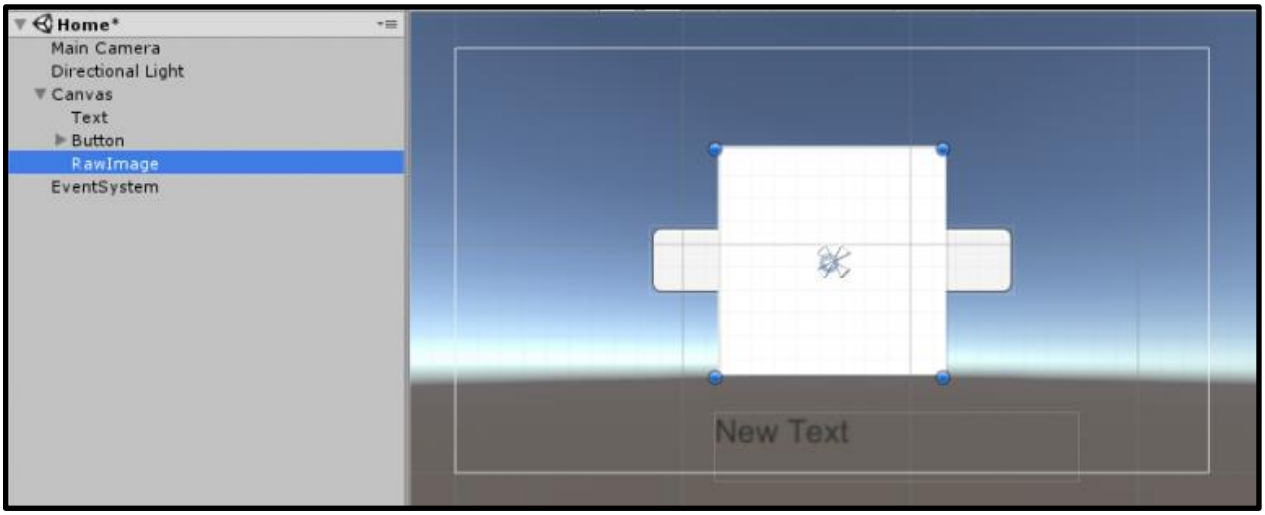
Home screen

In this section we'll implement a simple home screen. Create a new scene in your Scenes folder and call it "Home". Add it to your Build Settings like we did before, make sure it's at the start of the list.

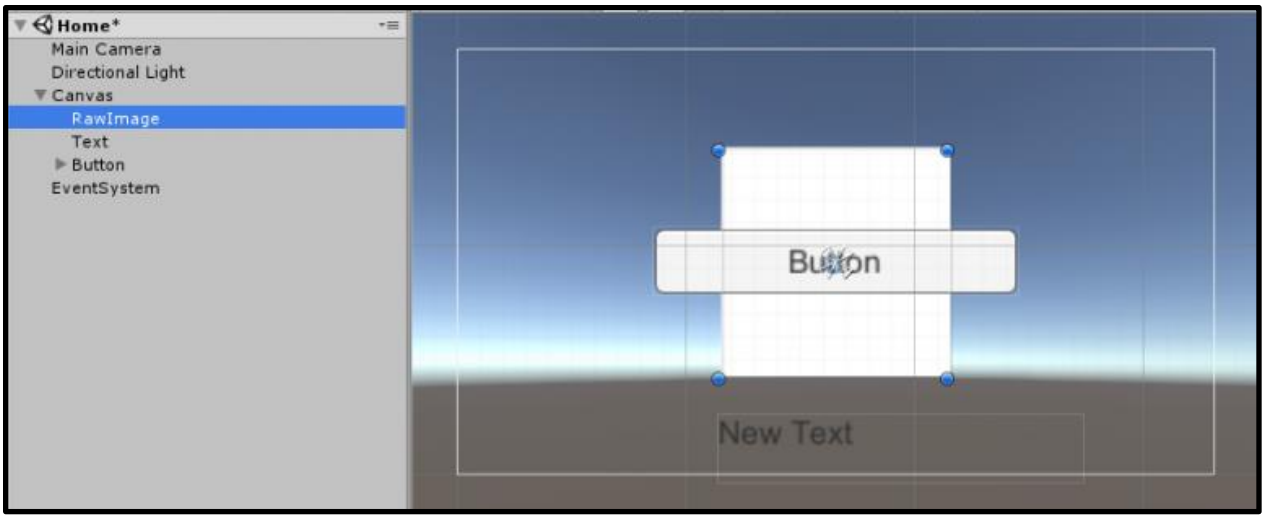


For the background of our home screen (and our game over screen) I'll be using the image that comes in *Assets/Images*. Feel free to use that or any other image. Create an *Images* folder in your project, and put the background image in there. The size of the image is 1024 x 576.

We'll create a text field like before (which will automatically create a canvas). Inside this canvas, also create a new button (*UI – Button*) and a raw image (*UI – Raw Image*).



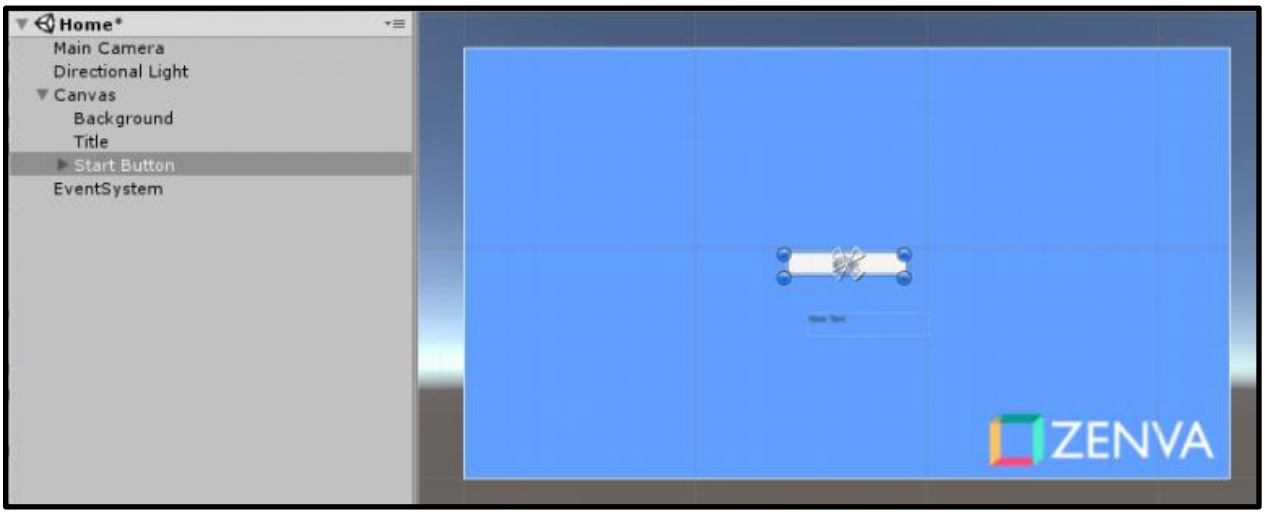
The order in which each UI element is rendered on the screen is that of the elements in the Hierarchy Window. If you want an element to appear on top of everything, move it all the way down. On the contrary, to have an element in the background, move it all the way up. We'll make sure the raw image is in the background.



Select the Canvas, set the *UI Scale Mode* to *Scale With Screen Size,* which will make everything look small. Change the Reference Resolution to 1024 x 576.

Extend the corners of the raw image element to match those of the canvas. In the Inspector, select our background image on the Texture field (or drag the image file onto the RawImage object).

Rename the RawImage object to “Background”, the text to “Title” and the button to “Start Button”.



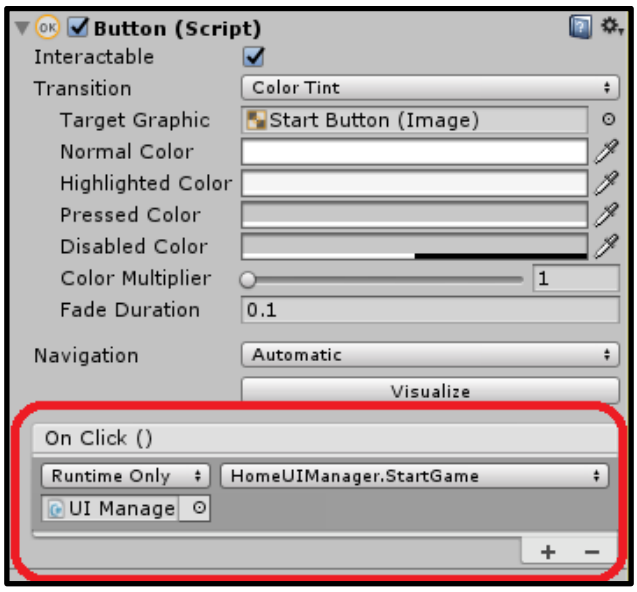
Change the font of Title to our Press Start font. Change the font size to 48, color white. Move it around so that it looks nicer. You can easily style the button too by expending it in the *Hierarchy Window*, and editing it’s child *Text* field just like we did with the title. This is what my home screen looks like:

The last step here is to make our button work and actually make the game start. Create an empty object called *UI Manager*. Create a new script named *HomeUIManager* and drag it on to the newly created object.

This new script will contain a single public method which we’ll call *StartGame* (not to be confused with *Start*):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HomeUIManager : MonoBehaviour
{

    // Start the game
    public void StartGame()
    {
        SceneManager.LoadScene("Level1");
    }
}
```

Select *Start Button* in the Hierarchy Window, find the component *Button (Script)* in the *Inspector*. Where it says *On Click()* click the plus button and find the public method we just created (*StartGame*). After this, your homescreen should work is you press *play*!



Game Over screen

For simplicity, we'll make the Game Over screen will work very similarly to the Home screen. Start by creating a new scene called *Game Over*. As a starting point, I'd recommend copying and pasting all the objects from the *Home* scene. Rename the text to "Game Over". Add the scene to the Build Settings as we've done with the previous scenes.

We'll add 4 additional text labels. Give them style and text values as shown below (enter any number for the score and high score):

The following functionality needs to be implemented:

- Restart the game when Play is pressed
- Show the score and high score

For that, we'll take a similar approach to what we did for the home screen. We already have a *UI Manager* object in our *Hierarchy Window* (provided you copied and pasted everything from the Home screen). We'll use this object, but with a different script, so remove the *HomeUIManager* script component in the *Inspector*.

This new script will make use of the `GameManager` object, which as we've seen keeps track of the score and high score, and already has a method to restart the game. Create a new empty object, name it *Game Manager*, and drag the *GameManager* script to it.

Create a new script called *GameOverUIManager*. This new class will have public variables so that we can pass on the text elements where we want to show the score and high score. Also, it will have a public method to restart the game, which we can use in our Play button.

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

public class GameOverUIManager : MonoBehaviour {

    // Where the score value will be shown
    public Text score;

    // Where the high score value will be shown
    public Text highScore;

    // Run on the first frame
    void Start()
    {
        // Show the score and high score
        score.text = GameManager.instance.score.ToString();
        highScore.text = GameManager.instance.highScore.ToString();
    }

    public void RestartGame()
    {
        // Reset the game
        GameManager.instance.Reset();
    }
}
```

Go to your button, rename it to “Restart Button”, remove the previous method in the On Click section, and find the newly created `RestartGame` public method.

You should be seeing zero values on the score and high score fields. Also, if you click on Play that should take you to Level 1.

The last piece of the puzzle is to actually send the player to Game Over, which we’ll do in `PlayerController`, upon collision with an enemy. Start by including the `UnityEngine.SceneManagement` namespace so we can easily switch between scenes:

```
using UnityEngine.SceneManagement;
```

Then send to Game Over:

```
        else if (collider.gameObject.tag == "Enemy")
        {
            // Game over
            print("game over");

            SceneManager.LoadScene("Game Over");
        }
```

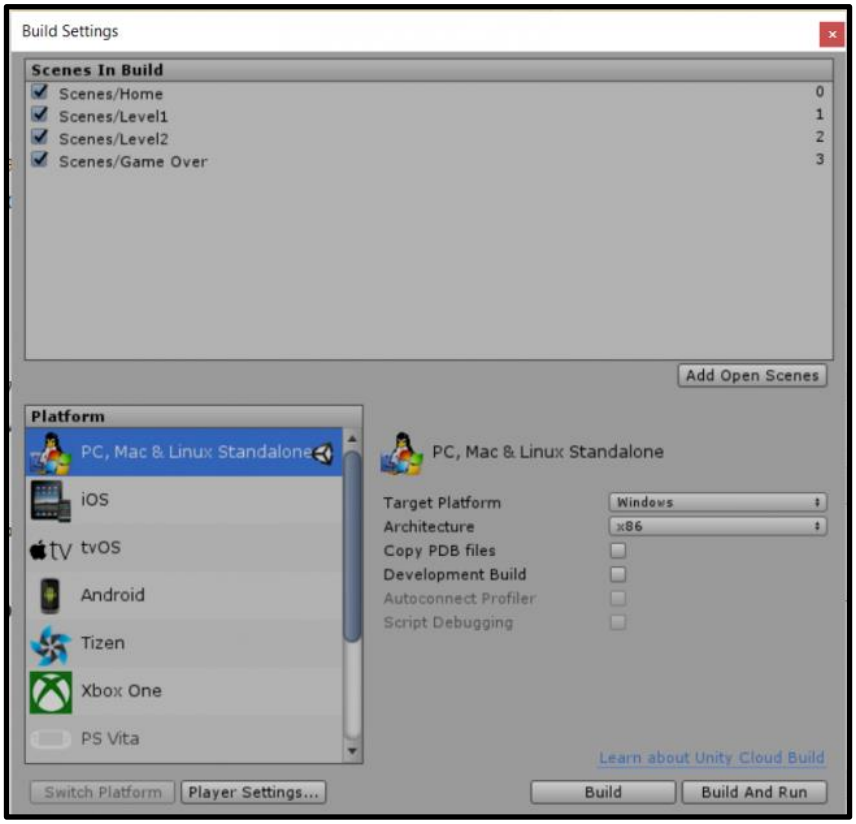
Finishing up

If you've followed along all the way here I have to say well done! We've already covered a lot of ground and all the concepts we've covered here apply to pretty much all games.

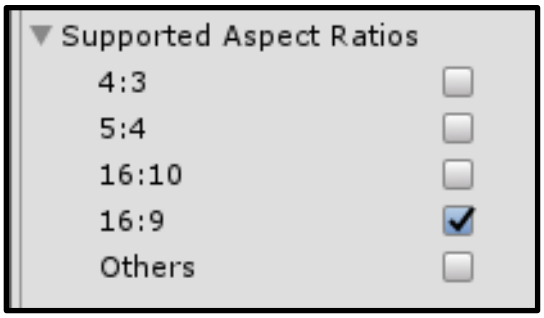
You are definitely on the right path if your goal is to make games with Unity.

The last part is to actually build your game, so that you can run it as a native application on your computer, whether you are on Windows, Mac or Linux (that is the magic of Unity!).

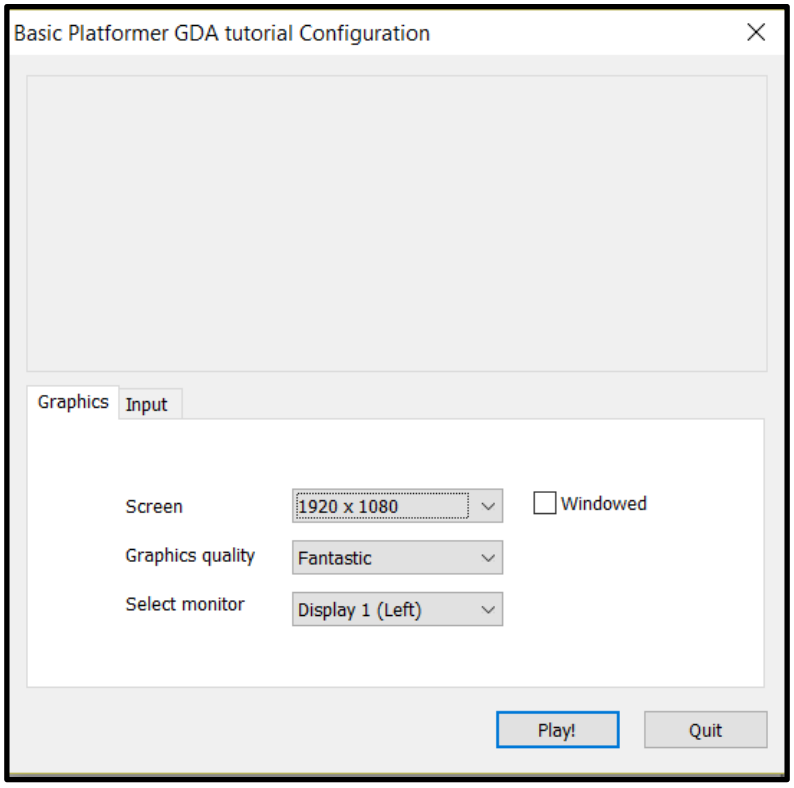
Build the game by going to *Build Settings*, selecting *PC, Mac & Linux Standalone*. I'll select *Windows* as my target platform and x86 as the system architecture, then click *Build and Run*. Make sure all the scenes are showing under *Scenes to Build*.



We've built our game assuming a screen ratio of 16:9 (that is the screen ratio of the background image we are using), so we want to make sure only this ratio is supported. Otherwise, in some screens people might see the area outside of our background. Go to *Player Settings*, find *Resolution and Presentation – Supported Aspect Ratios* – uncheck all but 16:9.



Choose a destination and a name for the executable file (I've called mine *game.exe*). Choose a resolution and whether you want your game to be windowed or not. You can disable this window in *Player Settings – Resolution and Presentation – Display Resolution Dialog*.



And we are done! We can now play our newly created game. Good job making it all the way here!