ROLLER GAME TUTORIAL

INTRODUCTION

- In this beginner assignment we are going
- to make a very simple but playable game to use many of the basic concepts from the beginner tutorial modules.
- We will be making a roll-a-ball game where we will collect special game objects.
- We will see how to create new game objects, add components to these game objects, set the values on their properties and position these game objects in the scene to create a game.
- In our game the player will control a ball rolling around the game board.
- Introduce the new input system
- Introduce UI
- We will move the ball using physics and forces.

ENVIRONMENT AND PLAYER

Setting up the Game

1. create a new project from File - New Project.
2. call the new project Roller Game
3. set the destination, or path to our new project.
4. use default settings and choose 3D
5. Before creating anything in the new scene, we need to save our scene.
6. Save Scene by going to the File in new folder called _scenes in the Assets directory.
7. Name the scene as Level_1
8. Create the game board, or play field by using stock unity plane and name it Ground
9. Reset the transform component using the context sensitive gear menu in the upper right
10. Change the scale of the ground
11. Create player object using unity sphere, and move it up until rests on the plane
12. We can see that the player game object is lit and it cast a shadow on the plane. This is because all the new unity scene come with the default sky box and a directional light
13. Add color to the ground. We need to use a material.
14. Create a folder called Materials to hold new materials and create a materials called Ground
15. Change the color using the Albido property RGB (0, 32,64).
16. Rotate the main directional light as this will help us later in this project, select the directional light and in the transform, component change the Transform Rotation on the X and Y axis to 50. This will give better shape to our plyer sphere.

MOVING THE PLAYER

17. We want to have the sphere roll around on the game area, bump in to walls, stay on the ground and not fly off in to space. We want to be able to collide with our collectible game objects and pick them up when we do. **This requires physics**.
18. To use physics, the game object needs a Rigidbody component attached.
19. select the game object we want to attach the component to, in this case we will select our player game object. Then we can either choose the Component menu and select Physics - Rigid Body Or use the Add Component button in the Inspector choosing Physics - Rigid Body.
20. Need to activate the NEW input system by going to Window→Package Manager→Input System→Install. This is install the new input system. Answer No to the dialog and then Go to Edit→Project Setting→Player→Other Setting→ Input Handling→ set to Both.
21. Now we need to **get the player object moving under our control**. To do this we need to get input from our player through the keyboard and we need to apply that input to the player game object as forces to move the sphere in the scene. We will do this by using a script that we attach to the player game object.
22. Add Player Input component to the Player and then create a new folder "Input" and click on create Action in the Player Input component and save it to as "InputAction" to Input folder.
23. create a folder in our project view to hold our script assets. Rename this folder Scripts.
24. Next let's create a new C# script. To create a new script, we have some choices.
25. We can choose Assets - Create to create our new C# script.
26. Or we can use the Create menu in the project view. But what might be more efficient in this case would be to select the player game object and use the Add Component button in the Inspector. **This allows us to both create and attach** a script in one step.
27. First let's name this script PlayerController.
28. Let's open the script. We can do this a number of ways. First let's remove the sample code provided in the base script.
29. Next let's think, what do we want to do with this script? We want to check every frame for player input and then we want to apply that input to the player game object every frame as movement.
30. Where will we check for and apply this input? We have two choices. **Update and Fixed Update**. Update is called before rendering a frame and this is where most of our game code will go. Fixed Update on the other hand is called just before performing any physics calculations and this is where our physics code will go. We will be moving our ball by applying forces to the rigidbody, this is physics. So we will put our code in Fixed Update.
31. What code to we need to write? Select the item you want to research, in our case input, and **hold down the command or control key and type the single quote** button. We use this class to read the axis setup in the input manager and to access multitouch touch and accelerometer data on mobile devices.
32. In our code we will be using Input.GetAxis.
33. Let's return to our script and write our code.
34. float moveHorizontal = Input.GetAxis ("horizontal")

35. float moveVertical = Input.GetAxis ("vertical")
36. This **grabs the input from our player through the keyboard**.
37. We will use this input to add forces to the rigidbody and move the player game object in the scene.
38. To know more about how to apply forces to the rigidbody let's check the documentation. Type Rigidbody in to our script   and hold down the command key on the mac or the control key on the pc and type single quote.
39. let's choose AddForce. This adds a force to the rigidbody as a result the rigidbody will start moving.
40. This signature tells us we need a vector3 and an optional ForceMode to add force to our rigidbody. which requires a value for force on each of the X, Y and Z axis.
41. The next concept that we need to cover is how to get a hold of, or how to reference different components on our game object. We are writing a script called PlayerController. Which is attached as a script component to our Player game object. From this script we need to AddForce using the rigidbody component. We want to access that component from this script. **We will create a variable to hold this reference in our script** and we will set this reference in the Start function.
42. public Rigidbody rb creates a public variable In Start the reference is set by using the code GetComponent This will find and return a reference to the attached rigidbody, `rb = GetComponent<Rigidbody>();`
43. Finally in FixedUpdate the attached rigidbody component is accessed through the variable named rb with rb.AddForce.
44. The X, Y, Z values will determine the direction of the force we will add to our ball.
45. What is our X value? That would be moveHorizontal.
46. What is our Y? 0. We don't want to move up at all.
47. What is our Z value? That would be moveVertical.
48. Now we use Movement, a vector3 value, in rb.AddForce as rb.AddForce(movement).
49. **Hit Play**, and by using the keys setup on the input manager
50. the ball moves in the scene. But it's very slow.
51. Let's return to our code and create a tool that will give us control over the speed of the ball. We need to multiply our movement by some value.
52. The solution is to create a public variable in our script. Let's create a public float called Speed. By creating a public variable in our script this variable will show up in the Inspector as an editable property. We now have control over our movement value

```
//Old Input
using UnityEngine;
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);
    }
}
//NEW Input System
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerController : MonoBehaviour
{
    public float speed = 0;

    private Rigidbody rb;

    private float movementX;
    private float movementY;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    private void OnMove(InputValue movementValue)
    {
        Vector2 movementVector = movementValue.Get<Vector2>();

        movementX = movementVector.x;
        movementY = movementVector.y;
    }

    private void FixedUpdate()
    {
        Vector3 movement = new Vector3(movementX, 0.0f, movementY);

        rb.AddForce(movement * speed);
    }

}
```

2. CAMERA AND PLAY AREA

MOVING THE CAMERA

1. We need to tie the camera to the player game object.
2. First let's set the position of the camera. Let's lift it up by 10 units and tilt it down by about 45 degrees.
3. Next let's make the camera a child of the player game object.
4. Unlike a normal third-person game our player game object is rotating on all 3 axis not just 1. In a typical third-person setup the camera as a child of the player game

object will always be in a position relative to it's immediate parent, and this position will be the parent's position in the game, modified or offset by any values in the child's transform.

5. We can't have the camera as a child of the player, so let's detach it.
6. Our offset value will be the difference between the player game object and the camera.
7. Now we need to associate the camera with the player game object, not as a child but with a script. Using the Add Component button choose New Script. and name the script CameraController
8. We need 2 variables here. A public GameObject reference to the player and a private vector3 to hold our offset value.
9. For our offset value we will take the current transform position and subtract the transform position of the player to find the difference between the two. So in start we can make offset equal to our transform position minus the player's transform position. And then every frame we set our transform position to our player's transform position plus the offset.
10. However, update is not the best place for this code. is true that update runs every frame and in update each frame we can track the position of the player's game object and set the position of the camera. However, for follow cameras, procedural animation, and gathering last known states it's best to use LateUpdate.
11. **LateUpdate** runs every frame, just like update. But it is **guaranteed to run after all items have been processed in update**.
12. First we need to create a reference to the player game object by dragging the player game object in to the Player slot in the camera controller's component.

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

public class CameraController : MonoBehaviour
{
    public GameObject player;
    private Vector3 offest;

    // Start is called before the first frame update
    void Start()
    {
        offest = transform.position - player.transform.position;
    }

    // Update is called once per frame
    void LateUpdate()
    {
        transform.position = player.transform.position + offest;
    }
}
```

SETTING UP THE PLAY AREA

1. So let's set up our play field. We will place walls around the edges to keep our player game object from falling off and we will create and place a set of collectable objects for our player to pick up.

2. First let's create a new empty game object and rename it Walls. This will be an organising parent game object our Wall objects.
3. Reset this game object to origin.
4. Now we will build our walls by creating a new cube.   Rename this West Wall.
5. Reset this game object to origin. We need to change the size of the cube to fit one side of our play area.
6. Change the cube's transform scale of X, Y and Z to 0.5 for thin, 2 for tall and 20.5 for long. Now we can simply push the wall in to place using the Translate tool or we could enter a value in to the transform component. In this case we can set the transform's position X value to -10.
7. To create the next wall we could start with another new cube, So let's duplicate the West Wall game object. Let's rename it East Wall. To place the wall simply remove the negative sign
8. Now let's duplicate the east wall and call it North Wall. Reset the X position so the north wall is in the center of the play area. We can rotate the wall by 90 degrees, or, as this is a cuboid, we can rescale the wall to 20.5 in the X and 0.5 in the Z. We can drag the wall in to place, or we can simply use the value of 10
9. Duplicate North Wall and call it South Wall, and -10 in the Z axis pops it in to place.
10. Enter play mode and test.

3. COLLECTING, SCORING AND BUILDING THE GAME

CREATING COLLECTABLE OBJECTS

Next let's create our collectable objects.

1. Create a new cube and rename it PickUp. Reset the Pickup's transform to origin.
2. The cube is also a regular shape, 1 by 1 by 1. So let's lift it up by half a unit and we know it's resting on the plane. This cube will be our Pickup object. To be effective it must attract the attention of the Player.
3. So let's make the cube more attractive. We will start by making it smaller by half. This will also give it the effect of floating in above the play area. Both will help identify this object as special. Not enough however.
4. Let's tilt it over. 45, 45, 45 in each of the axis of the transform's rotation.
5. One thing I feel certainly attracts the attention of a player, and that is movement. **So let's rotate the cube**. To do this we need a script. With the Pickup object selected use the Add Component button in the Inspector. Create a new **script called Rotator**. Click Create and Add to confirm.
6. **We want the cube to spin**, and we want to do it with this script.
   We will not be using forces, so we can use update rather than fixed update.
7. We have already set the transform rotation in the transform component to 45, 45, 45 for the X, Y and Z axis. But these values don't change by themselves. We want to change these values every frame. To make the cube spin we don't want to set the transform rotation,
8. Again, this brings up the page with a search term Transform. Select Transform.
   This brings up the Transform page in the documentation.
9. There are two main ways to effect the transform. These are Translate and Rotate.

10. We will use Rotate.
11. write **Rotate (new Vector3 (15, 30, 45)** Now this action also needs to be smooth and frame rate independent. So we need to multiply the vector3 value by Time.deltaTime. Save this script and return to Unity. Test by entering play mode, and our Pickup object rotates.
12. Let's exit play mode. Okay, we have the start of a working Pickup object.
13. Next we want to place these around the game area. But before we do this we need to do one important step. We need to make our Pickup object in to a prefab. So remember, a prefab is an asset that contains a template, or blueprint of a game object or game object family. We create a prefab from an existing game object or game object family, and once created, we can use this prefab in any scene in our current project. With a prefab of our Pickup object we will be able to make changes to a single instance in our scene, or to the prefab asset itself, and all of the Pickup objects in our game will be updated with those changes.
14. So first let's create a folder to hold our prefabs. We want this folder on our root, or top level of our project. So select the project view _and make sure that no other item or directory is highlighted. And then choose Create – Folder Rename this folder Prefabs. Now drag the Pickup game object from our hierarchy and place it in to our Prefabs folder. When we drag an item from our hierarchy in to our project view we create a new prefab asset containing a template, or blueprint of our game object or game object family.
15. Before we spread our collectables around the game area we should create a new game object to hold our Pickups and to help organize our hierarchy. Let's create a new game object and call it Pickups. Check to make sure this parent game object is at origin and drag our Pickup game object in to it.
16. Now we want to spread a number of these Pickup objects around the play area.
17. First, make sure we have the Pickup game object selected, not the parent. Now let's move in to a top-down view by clicking on the gizmo in to upper-right of our scene view. Let's back out a little so we can see the entire game area. Grab the Pickup game object, and it doesn't move in the scene the way we want it to. _Note how the cube is moving in relation to
18. Now see how the orientation of the gizmo changes?
19. And now we can drag our game object around relative to the world's global axis. So let's lay down a few of these in the game area. Take our first Pickup object and place it in to the game area, some place convenient. I'm going to place mine at the top. With the game object selected, duplicate it. This can be done either by selecting Edit - Duplicate or by using the hot key combination, this is command-D on the Mac or control-D on the PC.
20. Now we place the second instance of the prefab. Using the hot keys we will create a few more, placing them around the play area. Okay, that's 12.
21. Let's hit play and test. Excellent.
22. Let's change their color/ To do this we need another material. To make things easy we can simply select our existing material and duplicate it. Let's rename this new material Pickup. With the material selected in the project view let's change the

albido color property to yellow. Now we can change the color of the prefab by changing the prefab's material. We can do this in two ways. We can change the material on just one instantiated prefab_If we do this, we must remember to use the Apply button to apply those changes to the prefab asset. Otherwise we will only change the material on this single instance. The other way, which is perhaps more simple is to simply change the material on the prefab asset directly.

23. _Let's hit play and test.
24. There, that looks better.

```
using UnityEngine;
using System.Collections;

public class Rotator : MonoBehaviour {

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

Collecting the Pick Up

Collecting the pick-up objects; discussing physics, collisions and triggers

1. We want to be able to pick up our collectable game objects when our player game object collides with them. To do this we need to detect our collisions between the player game object and the PickUp game objects.
2. We will need to have these collisions trigger a new behavior and we will need to test these collisions to make sure we are picking up the correct objects.
3. First, we don't need our player to remain inactive. so let's tick the Active checkbox and bring back our player.
4. Next let's select the PlayerController script and open it for editing.
5. what code are we going to write? We could write collider and then search the documentation using the hot key combination.
6. What we are interested in here is the sphere collider component.
7. In the header of each component on the left is the component's turndown arrow, the icon, the Enable checkbox if it's available, and the type of the component.

On the right is the context sensitive gear gizmo and an icon of a little book with a question mark. Now this is what we need. This is the quick link to the component reference.

8. If we select this icon we are taken not to the scripting reference but to the component reference.
9. We, however, want to find out how to script to this component's class. To do this we simply switch to scripting and we are taken to the scripting reference for the sphere collider.
10. We want to detect and test our collisions. For this project we are going to use OnTriggerEnter.

11. This code will give us the ability to detect contact between our player game object and our PickUp game objects without actually creating a physical collision.
12. First, let's copy the code, and then let's return to our scripting application.
13. We are using the function OnTriggerEnter. OnTriggerEnter will be called by Unity when our player game object first touches a trigger collider.
14. Now we have the page on GameObject. There are two important items here that we want. They are tag, tag allows us to identify the game object by comparing the tag value to a string.
15. And SetActive. This is how we activate or deactivate a game object through code.
16. The last item we need to know about is Compare Tag.
17. We must declare our tags in the Tags and Layers Panel before using them.
18. Let's save this script and return to Unity and check for errors.
19. The first thing we need to do it set up the tag value for the PickUp objects.
20. Select the prefab asset for the PickUp object. When we look at the tag list
21. Select Add Tag. To create a new custom tag select the + button to add a new row to the tags list. In the new empty element, in our case tag 0, type PickUp, and this is case sensitive and needs to be exactly the same string that we have in our script.
22. Save the scene and enter play mode.
23. Hmm, okay, our tag is set to PickUp but we are still bouncing off the PickUp cubes just like we are bouncing off the walls.
24. we need to have a brief discussion about Unity's physics system.
25. This leaves us with the two green outlines of the collider volumes for these two objects.
26. How do collisions work in Unity's physics engine? The physics engine does not allow two collider volumes to overlap.
27. One of the major factors in this calculations whether the colliders are static? or dynamic. Static colliders are usually non-moving parts of your scene, like the walls, the floor, Dynamic colliders are things that move like the player's sphere or a car. When calculating a collision, the static geometry will not be affected by the collision. But the dynamic objects will be. In our case the player's sphere is dynamic, or moving geometry, and it is bouncing off the static geometry of the cubes. Just as it bounces off the static geometry of the walls.
28. The physics engine can however allow the penetration or overlap of collider volumes. We do this by making our colliders in to triggers, or trigger colliders.
29. When we make our colliders in to a trigger, or trigger collider, we can detect the contact with that trigger through the _OnTrigger event messages.
30. When a collider is a trigger you can do clever things like place a trigger in the middle of a doorway in, say, an adventure game, and when the player enters the trigger the mini-map updates and a message plays 'you have discovered this room'.
31. We are using OnTriggerEnter in our code rather than OnCollisionEnter. So we need to change our collider volumes in to trigger volumes.

32. To do this we must be out of play mode. select the prefab asset and look at the box collider component. Here we select Is Trigger
33. Let's save our scene, enter play mode and test.
34. We only have one issue. We have made one small mistake,
35. and this is related to how Unity optimizes it’s physics. As a performance optimization Unity calculates all the volumes of all the static colliders in a scene and holds this information in a cache. This makes sense as static colliders shouldn't move, and this saves recalculating this information every frame.
36. Where we have made our mistake is by rotating our cubes. Any time we move, rotate, or scale a static collider Unity will recalculate all the static colliders again and update the static collider cache. To recalculate the cache takes resources.
37. We simply need to indicate to Unity which colliders are dynamic before we move them. We do this by using the rigid body component. **Any game object with a collider and a rigid body is considered dynamic. Any game object with a collider attached but no physics rigid body is expected to be static**.
38. Currently our PickUp game objects have a box collider but no rigid body. So Unity is recalculating our static collider cache every frame. The solution is to add a rigid body to the PickUp objects. This will move the cubes from being static colliders to being dynamic colliders.
39. Let's save and play. And our cubes fall through the floor. Gravity pulls them down, and as they are triggers they don't collide with the floor.
40. Let's exit play mode. If we look at the rigid body component we could simply disable Use Gravity, which would prevent the cubes from being pulled downwards. This is only a partial solution however. If we did this, even though our cubes would not respond to gravity they would still respond to physics forces There is a better solution. And that is to select Is Kinematic. When we do this we **set this rigid body component to be a kinematic rigid body**.
41. **A kinematic rigid body will not react to physics forces and it can be animated and moved by it's transform**. This is great for everything from objects with colliders like elevators and moving platforms, to objects with triggers, like our collectables that need to be animated or moved by their transform.
42. So, static colliders shouldn't move, like walls and floors.
43. Dynamic colliders can move, and have a rigid body attached.
44. Standard rigid bodies are moved using physics forces.
45. Kinematic rigid bodies are moved using their transform.

```
using UnityEngine;
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);
    }

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

NEW Input System

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

public class PlayerController : MonoBehaviour
{
    public float speed = 0;

    private Rigidbody rb;

    private float movementX;
    private float movementY;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    private void OnMove(InputValue movementValue)
    {
        Vector2 movementVector = movementValue.Get<Vector2>();

        movementX = movementVector.x;
        movementY = movementVector.y;
    }

    private void FixedUpdate()
    {
        Vector3 movement = new Vector3(movementX, 0.0f, movementY);

        rb.AddForce(movement * speed);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("PickUp"))
```

```
            {
                other.gameObject.SetActive(false);
            }

        }
}
```

Displaying the Score and Text

1. Counting, displaying text and ending the game.
2. We need a tool to store the value of our counted collectables.
3. Let's add this tool to our PlayerController script.
4. Select the Player game object and open the   PlayerController script for editing.
5. Let's add a private variable to hold our count. This will be an int, as our count will be a whole number, we won't be collecting partial objects, and let's call it Count.
6. First we need to set our count value to 0. As this variable is private we don't have any access to it in the Inspector. This variable is only available for use within this script There are several ways we can set the starting value of Count, but in this assignment we will do it in the Start function.
7. In Start set our Count to be equal to 0.
8. Next we need to add to Count when we pick up our collectable game objects.
9. We will pick up our objects in OnTriggerEnter if the Other game object has the tag Pickup. So this is where we add our counting code.
10. After setting the other game objects active state to False we add our new value of Count is equal to our old value of Count plus 1.
11. Save this script and return to Unity.
12. Now we can store and increment our count but we have no way of displaying it. It would also be good to display a message when the game is over.
13. To display text in this assignment we will be using Unity's UI Toolset to display our text and values.
14. First let's create a new UI text-textMeshPro element from the hierarchy's Create menu.
15. Rename the text element CountText. So let's customize this element a bit.
16. The default text is a bit dark. Let's make the text color white,
17. The size and alignment are good.
18. And let's add some placeholder text Count Text.
19. We want our text to display in the upper left of the screen when the game is playing.
20. One of the easiest ways to move the   count text element in to the upper left is to anchor it to the upper left corner of the canvas, rather than to it's center.
21. So let's wire up the UI text element _to display our count value.
22. Start by opening the PlayerController script for editing.
23. Before we can code anything related to any UI elements **we need to tell our script more about them. The details about the UI toolset are held in what's called a**

namespace. **We need to use this namespace just as we are using UnityEngine and System.Collections**. So to do this, at the top of our script write **using TMPro**.

24. First create a **new public text variable called countText** to hold a reference to the UI text component on our UI text game object.
25.
26. We need to set the starting value of the UI text's Text property. We can do this in Start as well.
27. Write countText.Text = "Count: " _+ count.ToString, and we need the parenthesis.
28. Now this line of code must be written after the line setting our count value.
29. Count must have some value for us to set a text with.
30. Now we also need to update this text property every time we pick up a new collectable so in OnTriggerEnter after we increment our count value let's write again _countText.Text = 'Count: ' + count.ToString();
31. One way to make this a little more elegant is to create a function that does the work in one place and we simply call this function every time we need it.
32. Let's create a new function with void SetCountText
33. Finally let's replace the other line with the function call as well.
34. Now we see our PlayerController script has a new text property. We can associate a reference to our Count text simply by dragging and dropping the _CountText game object on to the slot.
35. Let's exit play mode.
36. We need to display a message when we have collected all of the cubes. To do this we will need another UI text object.
37. Again, using the hierarchy's Create menu _make a new UI text game object. Rename it Win Text.
38. Again, as before, let's customize the values on the component. Let's color the text white so it is easier to see. Let's make the text a little larger, _let's try about 24. Lastly, let's adjust the alignment to center and middle. And again let's add placeholder text Win Text.
39. Save the scene and swap back to our scripting editor.
40. We need to add a reference for this UI text element.
41. Create a new public text variable and call it winText.
42. Now let's set the starting value for the UI text's text property.
43. This is set to an empty string or two double quote marks with no content.
44. This text property will start empty. Then in the SetCountText function let's write
45. _if Count is greater than or equal to 12, which is the total number of objects we have in the game to collect, then our winText.Text equals You Win.
46. Save this script and return to Unity.
47. Again on our player, our PlayerController has a new UI text property. We can associate the component again by dragging the WinText game object in to the slot.
48. Save the scene and play.

49. So we're picking up our game objects,
50. we're counting our collectables,
51. and we win!

```
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!";
        }
    }
```

NEW Input System

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

public class PlayerController : MonoBehaviour
{
    public float speed = 0;
    public TextMeshProUGUI countText;
    public TextMeshProUGUI winText;

    private Rigidbody rb;
    private int count;

    private float movementX;
    private float movementY;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();
        count = 0;
        setCountText();
        winText.text = " ";
    }

    private void OnMove(InputValue movementValue)
    {
        Vector2 movementVector = movementValue.Get<Vector2>();

        movementX = movementVector.x;
        movementY = movementVector.y;
    }

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

    private void FixedUpdate()
    {
        Vector3 movement = new Vector3(movementX, 0.0f, movementY);

        rb.AddForce(movement * speed);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("PickUp"))
        {
            other.gameObject.SetActive(false);
            count += 1;

            setCountText();
        }

    }
}
```

Building the Game