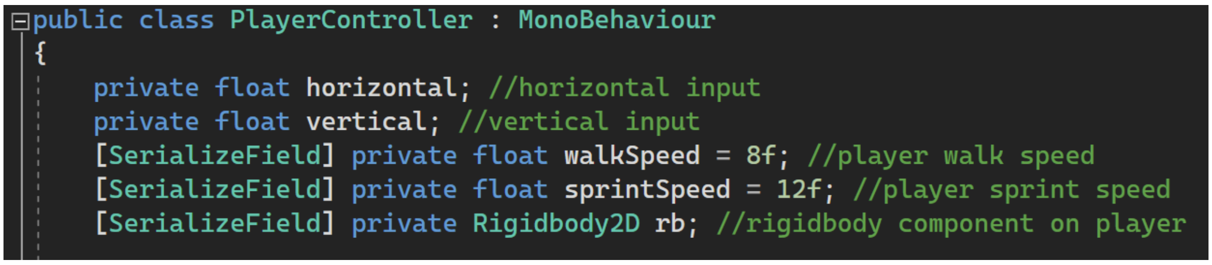
4 direction movement with sprinting

Create a square sprite as a placeholder for the player
Place the square into the scene and rename it to ‘Player’
Assign rigidbody2d component to the square. Freeze the z rotation and set the gravity scale to 0.
Assign a box collider 2d component to the square.
Create a script called PlayerController and assign it to the Player. Open the PlayerController script.

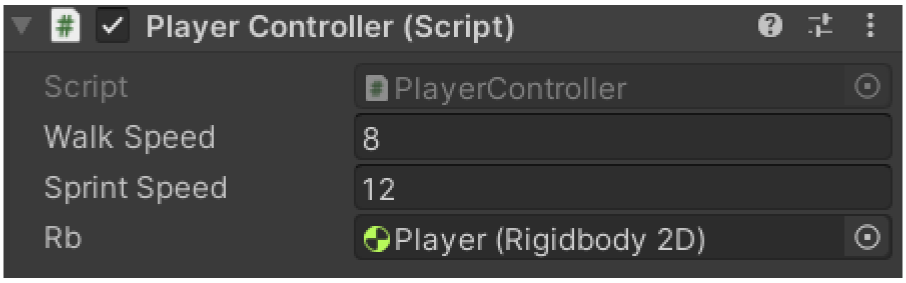
Delete the start function as it is unneeded. Type the following.



```
public class PlayerController : MonoBehaviour
{
    private float horizontal; //horizontal input
    private float vertical; //vertical input
    [SerializeField] private float walkSpeed = 8f; //player walk speed
    [SerializeField] private float sprintSpeed = 12f; //player sprint speed
    [SerializeField] private Rigidbody2D rb; //rigidbody component on player
```

The walkSpeed and sprintSpeed variables will determine how fast the player moves when walking/sprinting. The values assigned to them here are just examples, once our code is finished I recommend you adjust them to your liking. Save the script then return to the unity editor.

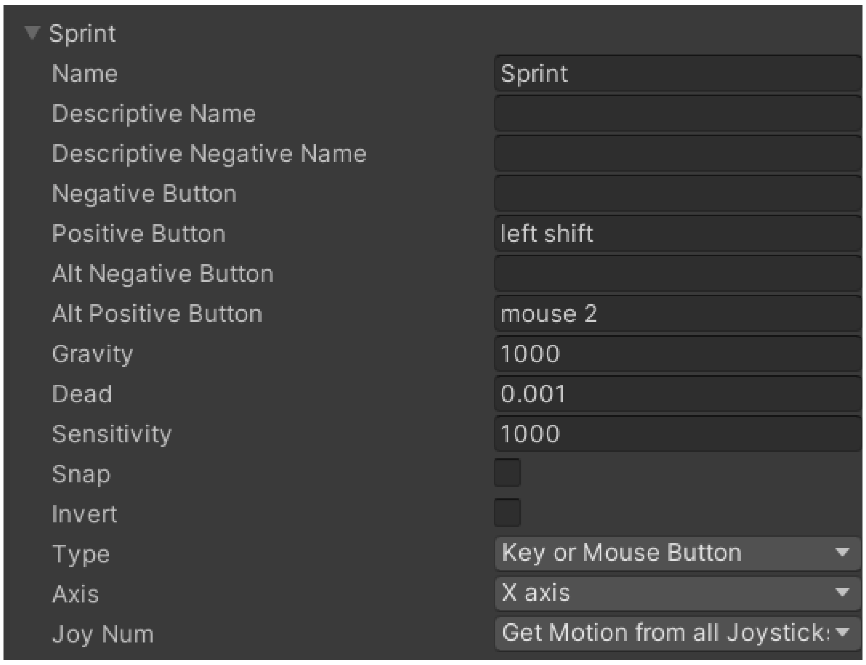
Under the PlayerController component on our Player object, assign the Player’s rigidbody component to the Rb field.



Player Controller (Script)

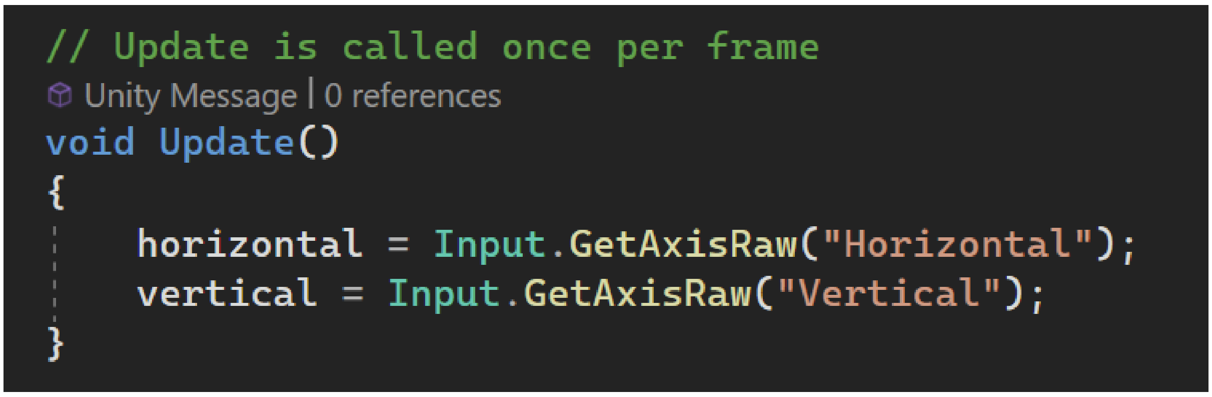
Script	PlayerController
Walk Speed	8
Sprint Speed	12
Rb	Player (Rigidbody 2D)

Go to Edit > Project settings > Input manager. These are inputs the player can use. Rename one of the preexisting axes to ‘Sprint’, though avoid doing this to the Horizontal and Vertical axes as we’ll need those for later. I’ve personally chosen to rename the Fire3 input as that already has left shift assigned as the positive button. If you have chosen to rename a different input, make sure it looks like the following.



```
▼ Sprint
Name                          Sprint
Descriptive Name
Descriptive Negative Name
Negative Button
Positive Button               left shift
Alt Negative Button
Alt Positive Button           mouse 2
Gravity                       1000
Dead                          0.001
Sensitivity                   1000
Snap                          ☐
Invert                        ☐
Type                          Key or Mouse Button
Axis                          X axis
Joy Num                       Get Motion from all Joystick:
```

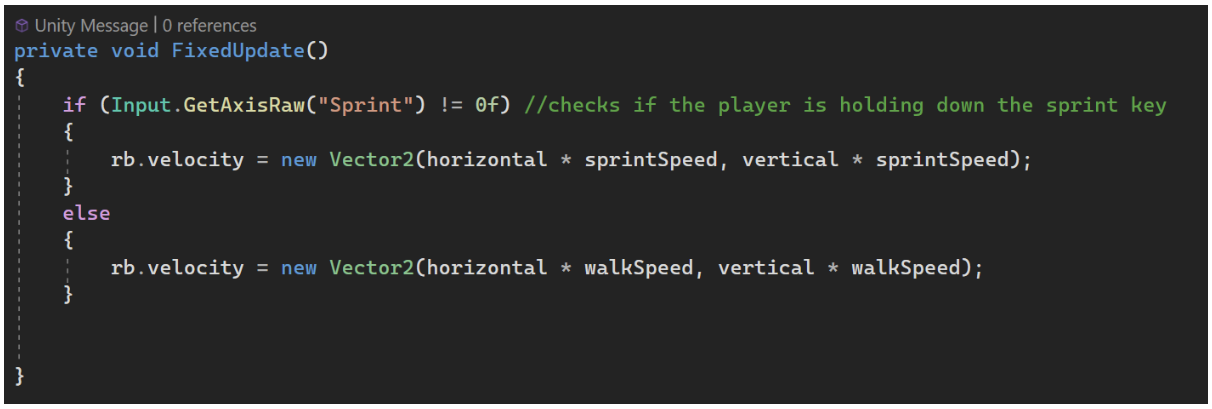
Close the input manager and return to our PlayerController script. Type the following inside the update function.



```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    horizontal = Input.GetAxisRaw("Horizontal");
    vertical = Input.GetAxisRaw("Vertical");
}
```

The GetAxis function return a positive or negative amount on the movement of a virtual axis. Assigning the horizontal and vertical variables to the output of these axes will provide them with a value of 1 or -1 depending on what button is being pressed.

Create a fixedupdate and type the following inside.



```
Unity Message | 0 references
private void FixedUpdate()
{
    if (Input.GetAxisRaw("Sprint") != 0f) //checks if the player is holding down the sprint key
    {
        rb.velocity = new Vector2(horizontal * sprintSpeed, vertical * sprintSpeed);
    }
    else
    {
        rb.velocity = new Vector2(horizontal * walkSpeed, vertical * walkSpeed);
    }


}
```

What this code does is check to see if the player is holding down the sprint key we assigned previously using the GetAxisRaw function. It’s checking to make sure that the input is not equal to 0, as when holding down the key its input will be equal to 1.

The rb.velocity function is whats going to move our player character around. What this does is essentially apply a force to whatever its assigned to using variables we dictate. That’s what the Vector2 here is used for - represents placement on a 2d plane on the x and y axis respectively.

By setting the X value to our horizontal variable times by our walk/sprint speed variable we are making it so that the speed variable is only applied once the keys assigned to the horizontal variable are pressed. As already stated, the axes in unity return a value of 0 when stationary, however when their corresponding buttons are pressed they output a value of either 1 or -1.

Save the PlayerController script and return to the unity editor. Hit the play button in order to test that what we’ve just written is working correctly. The player should move when w, a, s or d are pressed, and they should move at an increased speed when left shift is held down.

Collision tile mapping

This tutorial assumes that you do not currently have a tile palette in your project, and walks you through the steps of creating a basic placeholder. If your project already has a tilemap, skip past the segment highlighted in grey.

In your current unity project, create a folder named ‘Tilemaps’. We’ll need this for later.

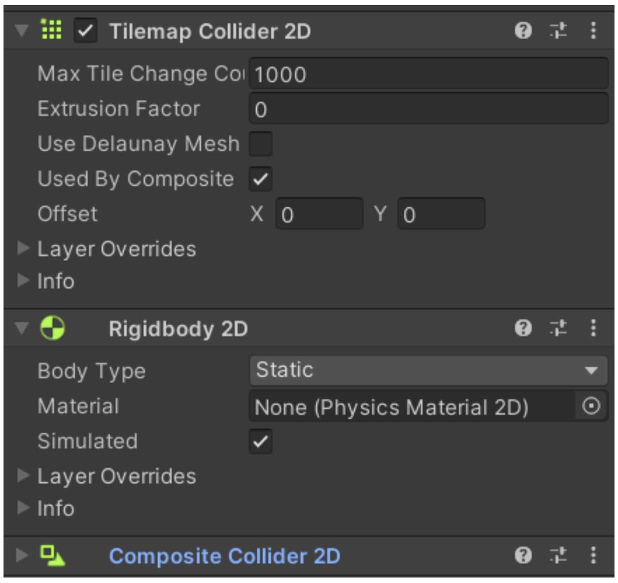
The default size for tiles is 100px by 100px, so we’re going to be working with these dimensions for now. In an external art program, create a tile that is 100px by 100px - I used one that is just a solid colour. Move this tile into the sprites folder of your unity project.

In unity, go to window > 2D > Tile palette. Create a new tile palette. Make sure that the grid is set to rectangular and save the palette. Now you should be able to select this palette in the tile palette window. Open the sprites folder in your unity project and drag the tile we made earlier onto the tilemap.

Create a new tilemap object in the hierarchy. (right click > 2D > Tilemap > Rectangular tilemap). Set its Z position to -1 to ensure that it appears over any background elements that might obscure it.

Select ‘add component’ and add a tilemap collider 2D. Select ‘add component’ again and add a composite collider 2D. This should automatically add a Rigidbody2D component to our grid. The composite collider exists to ensure that our tilemap forms one large collider around its edges, instead of each individual tile having its own collider.

Set the Rigidbody2D component’s body type to static. In the settings for the tilemap collider 2D, set ‘Used by Composite’ to true. The components should look like this.


Tilemap Collider 2D
Max Tile Change Coı 1000
Extrusion Factor 0
Use Delaunay Mesh
Used By Composite ✓
Offset X 0 Y 0
Layer Overrides
Info
Rigidbody 2D
Body Type Static
Material None (Physics Material 2D)
Simulated ✓
Layer Overrides
Info
Composite Collider 2D

Press ‘play’. Now, whenever our player character goes to run into one of the tiles we place down, they should be stopped by the collision.

Camera movement

[Made using unity’s create with code program]

Keep in mind that this tutorial is for a 2D game. This code will still work in 3d, however the offset variable will need to be adjusted.

Create a new C# script called ‘FollowPlayer’ and open it. Delete the start and update functions then type the following code

```
public GameObject player;
private Vector3 offset = new Vector3 (0, 0, -10);
```

The player GameObject will be assigned to your player character. Return to the unity editor and assign your player character GameObject to the ‘player’ variable.

The offset vector will be used to offset the position of the camera. If we didn’t apply the offset, the camera would be moved to our player’s exact position on all axis, and this makes it impossible to see. The -10 offset makes it so that our camera is floating just above the player instead of being right inside it.

Open our script back up and type the following

```
Void LateUpdate()
{
        transform.position = player.transform.position + offset;
}
```

The LateUpdate function is called every frame, however it’s only called after every other function on that same update has already been called. This should prevent our camera movement from looking jittery.

The transform.position command moves the position of our camera to the position of our player. We add the offset onto this so that our camera isn’t directly inside our player character.

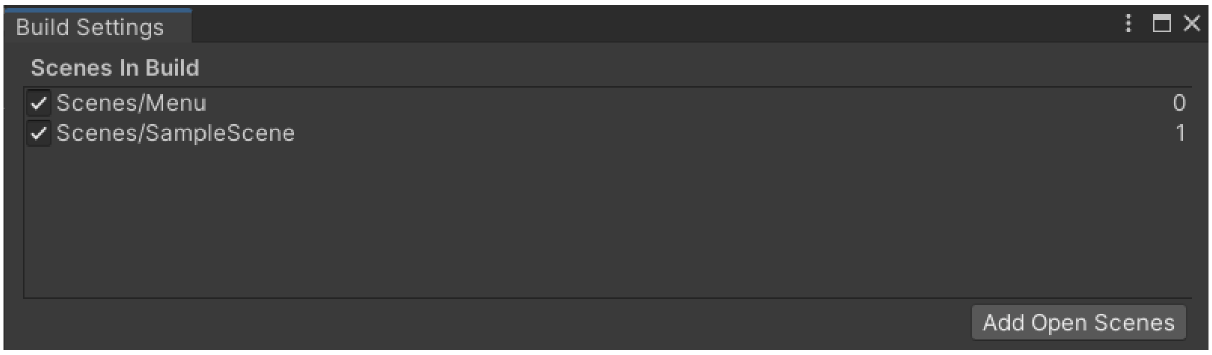
Save this and return to the unity editor.

Add the FollowPlayer script as a component to the main camera.

Main menu

Create a new scene in your unity project. Name it ‘Menu’ and open it.

Go to file > build settings. In the scene index make sure that the menu is at index number 0 and that the first scene in your game is at index number 1, like this:

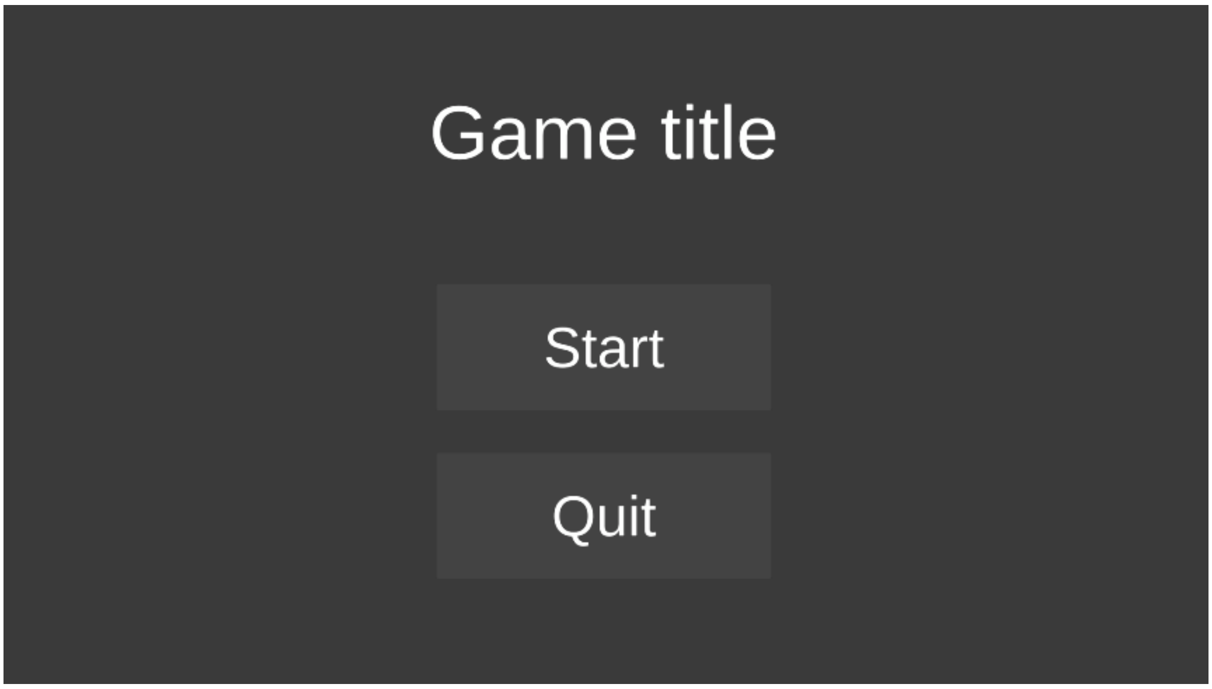


✓ Scenes/Menu	0
✓ Scenes/SampleScene	1

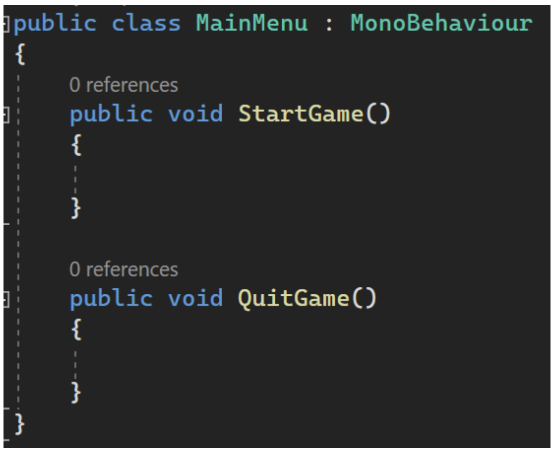
Create a new canvas object in the hierarchy. Set its UI scale mode to ‘scale with screen size’ and set the reference resolution. For this tutorial i’ll be using 1920 x 1080, however you can adjust this value to suit your target resolution.

Inside the canvas, create a raw image object.You can assign a texture to this if you’d like, but for the sake of this tutorial i’ll be assigning it a solid colour. Resize the object to 1920 x 1080 (or your target resolution) and align it with the canvas object.

Within the canvas object, create an empty game object and rename it to ‘MainMenu’. Within this you should create a TMP text object. Use this to input the name of your game (or a placeholder if need be). Also inside of the empty gameobject, create a TMP button object. This will serve as our start button. Edit the text within the button to say ‘start’. Resize the button and the text to your liking, as well as altering the colours to what you see fit. Once you’re done, duplicate it to create the quit button. You can arrange your menu however you like, but here’s what i’ve made as an example if you want to copy it.



In your project files, create a new folder called ‘scripts’ if you do not yet have one. In this folder, create a script called MainMenu and add it as a component to the MainMenu gameobject. In this script we’re going to create two new public functions. Type the following code:



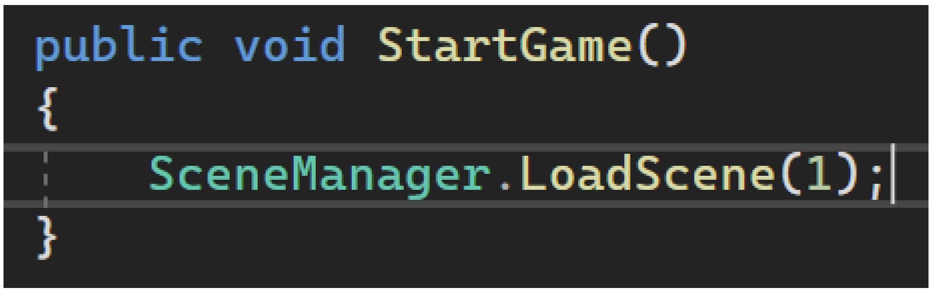
```
public class MainMenu : MonoBehaviour
{
    0 references
    public void StartGame()
    {

    }

    0 references
    public void QuitGame()
    {

    }
}
```

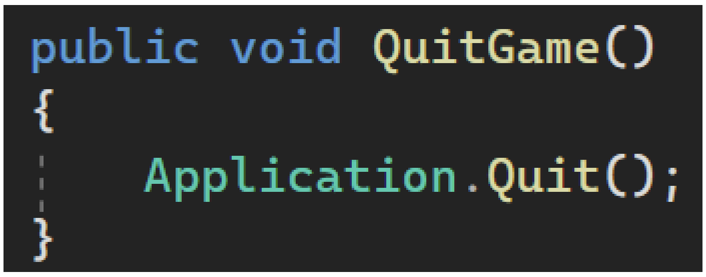
In the StartGame() function, type the following:



```
public void StartGame()
{
    SceneManager.LoadScene(1);
}
```

This is why we needed the first scene of our game to be at index number one, so that we could load it using the scene manager. You could also open the scene by typing its name as a string in place of its index number, but I’m using this method for simplicity's sake.

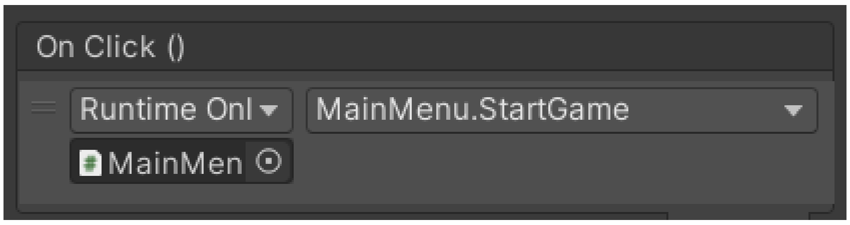
In the QuitGame() function, type the following:



```
public void QuitGame()
{
    Application.Quit();
}
```

This command closes the game window, however in the unity editor it doesn't appear to do anything. Your code isn't broken, this section just can't run in the editor.

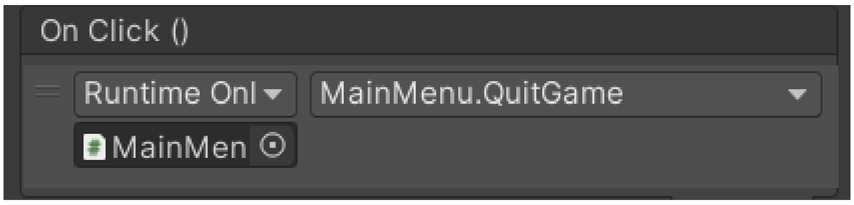
Save this script and return to the unity editor. Select both your start and quit buttons in the hierarchy, and in the onclick() section add the MainMenu gameobject as a component. For the start button, set the function to MainMenu > StartGame(). For the quit button, set it to MainMenu > QuitGame()



On Click ()

Runtime Onl | MainMenu.StartGame

MainMen



On Click ()

Runtime Onl | MainMenu.QuitGame

MainMen

Save the scene then run your game. Upon pressing the start button, you should be loaded in to your first scene.