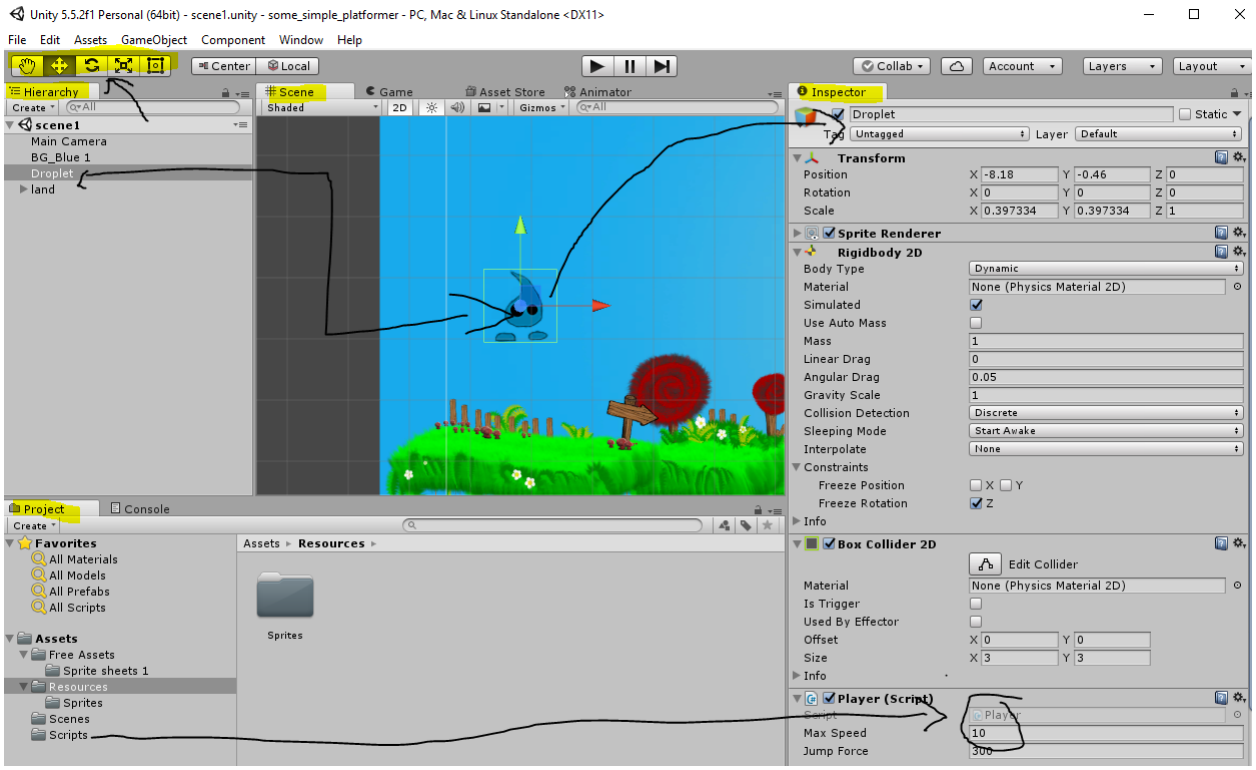
Class notes for Unity 2D Platformer Beginner Course
By Terry Gruenewald
Class resources are here: https://github.com/tgruenewald/unity2dclass.git
Day 1

Editor overview



The main parts are:

1. Mode selector
2. Hierarchy
3. Scene
4. Inspector
5. Project

Adding sprites to the scene as game objects

Last week we placed sprites in the scene. Created an empty game object called “land”, parented the smaller sprites to it and placed a large box collider on it.
Next we took a sprite for our player “droplet” and placed it on the scene, and added a RigidBody2D component to it. It immediately fell so we added a BoxCollider2D to it as well.

Horizontal player movement

We added movement to droplet so he could move left and right, and this was done by creating a script component on Droplet and calling the script Player.cs. The Player.cs file was placed in a newly created “Scripts” folder in the Assets folder in the project space. This code was added for droplet to move:

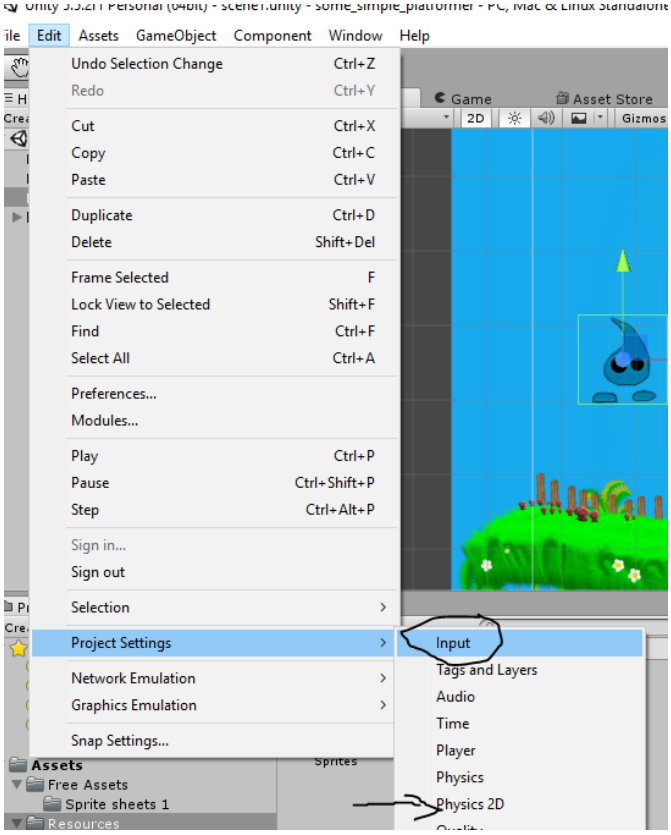
```
// Update is called once per frame
void Update () {
    move = Input.GetAxis ("Horizontal");
}
void FixedUpdate () {
    GetComponent<Rigidbody2D> ().velocity = new Vector2 (move * maxSpeed, GetComponent<Rigidbody2D> ().velocity.y);

}
```

“Update” vs “FixedUpdate”

Next we went over the difference between “Update” and “FixedUpdate” and how Update is called once per frame (usually 60 times a second) and that FixedUpdate occurred at predictable intervals and was synced with the game’s physics.

Where does “Horizontal” come from?

We talked about the “Input.GetAxis(“Horizontal”) and that “Horizontal” was setup in Edit | Project Settings | Input.



GetComponent and RigidBody2D

And that this is where the default settings for the game engine’s physics is located as well. The “GetComponent” was explained as a way to get access to the other components of the same game object that this script is located on. For movement the one we were interested in was “RigidBody2D” and we set the player’s velocity to being a new Vector2 which was based on

the movement captured from Input.GetAxis for the x axis, and then mirrored the movement from the player's y axis.

Jumping

Next we added jumping. This involved calling the Input.GetButtonDown for "Jump" and "Vertical". If either of these buttons were pressed then once again we accessed the RigidBody2D component and added a force with a strictly y vector equal to some amount of force that we fiddled with until we got it right. This was the code afterwards:

```
// Update is called once per frame
void Update () {
    move = Input.GetAxis ("Horizontal");
    jump = Input.GetButtonDown ("Jump") || Input.GetButtonDown ("Vertical");
    //Debug.Log ("move = " + move);
}
void FixedUpdate () {
    GetComponent<Rigidbody2D> ().velocity = new Vector2 (move * maxSpeed, GetComponent<Rigidbody2D> ().velocity.y);
    if (jump) {
        GetComponent<Rigidbody2D> ().AddForce(new Vector2(0f, jumpForce));
    }
}
```

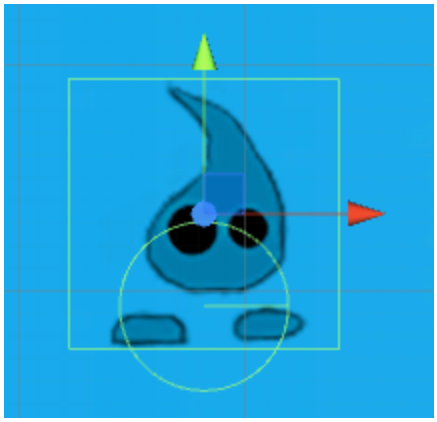
Pitfalls:

- When adding a method to override a MonoBehavior method, spelling and case are important. So it's "FixedUpdate()", not "fixedupdate()".

Day 2

Ground detection

A CircleCollider2D will need to be added to the bottom of droplet.



Add this to Player.cs. Then in the inspector we will specify on LayerMask what is ground.

```
public bool grounded = true;
public LayerMask whatIsGround;
private CircleCollider2D groundCheck;
```

Add this code to see if the collider is touching any of our specified layers. Also add “grounded” boolean to the jump conditional since we only want to be able to jump when we are on the ground.

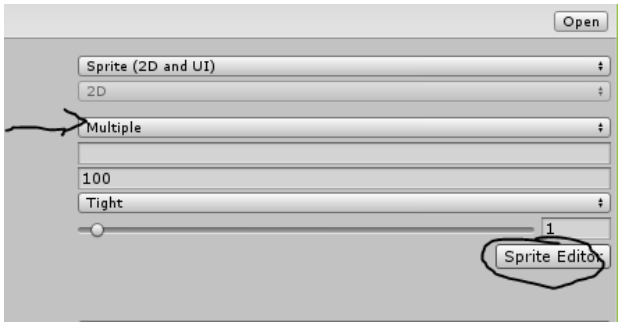
```
void Update () {
    grounded = groundCheck.IsTouchingLayers (whatIsGround);

if (grounded && jump) {
    GetComponent<Rigidbody2D> ().AddForce(new Vector2(0f, jumpForce));
}
```

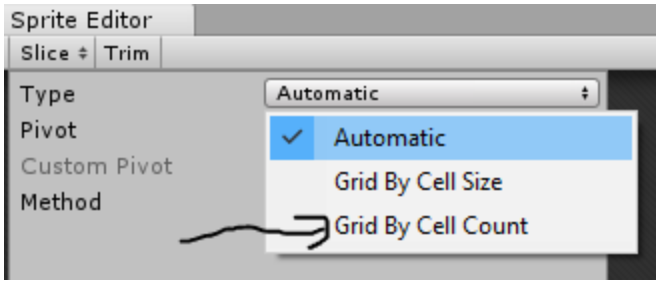
Animating droplet

To create some animations we first need to get some sprite sheets. See “droplet idle animation.png” and “droplet moving.png” from the imported resources.

Slicing sprites

Before we had nicely sliced sprites, but now we need to do this ourselves

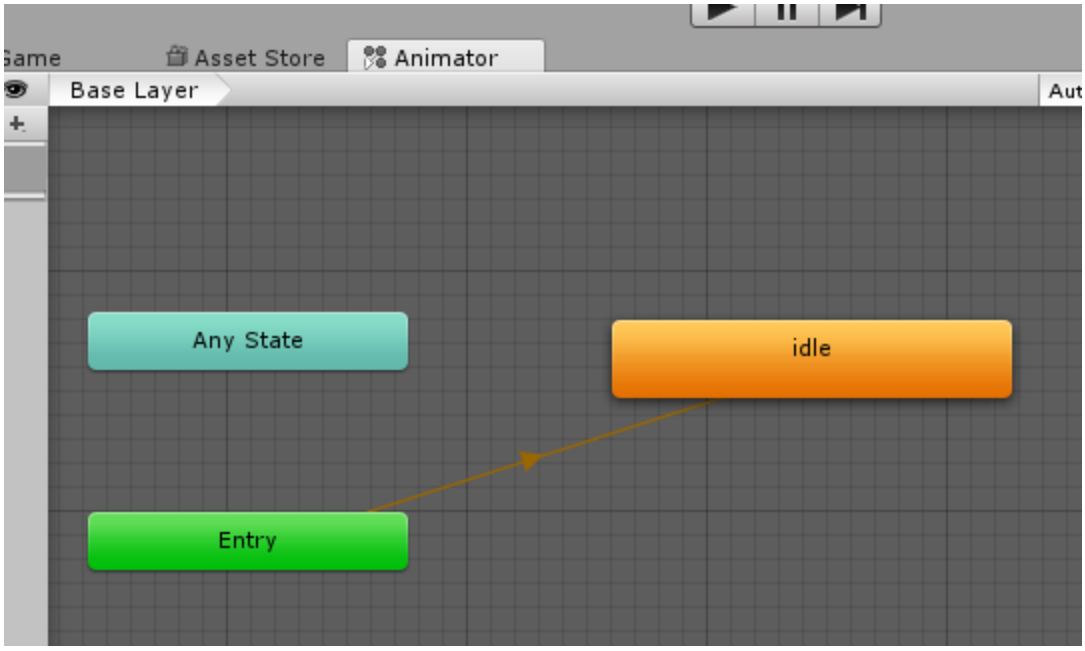




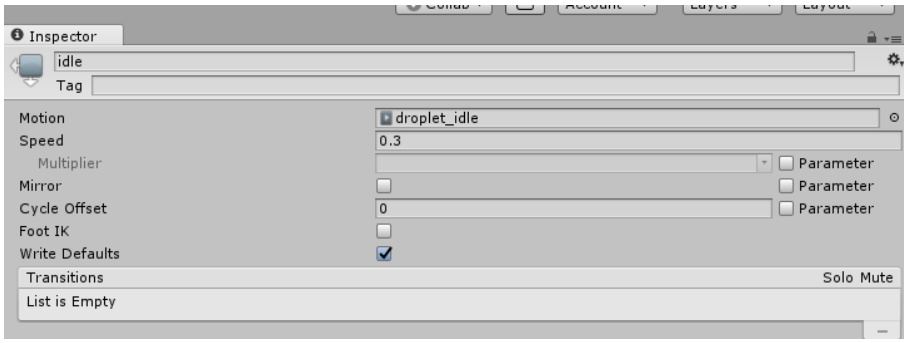
For the walking we will need to slice with an offset since the first cell is missing.

Creating the animations and the animation controller (Animator).

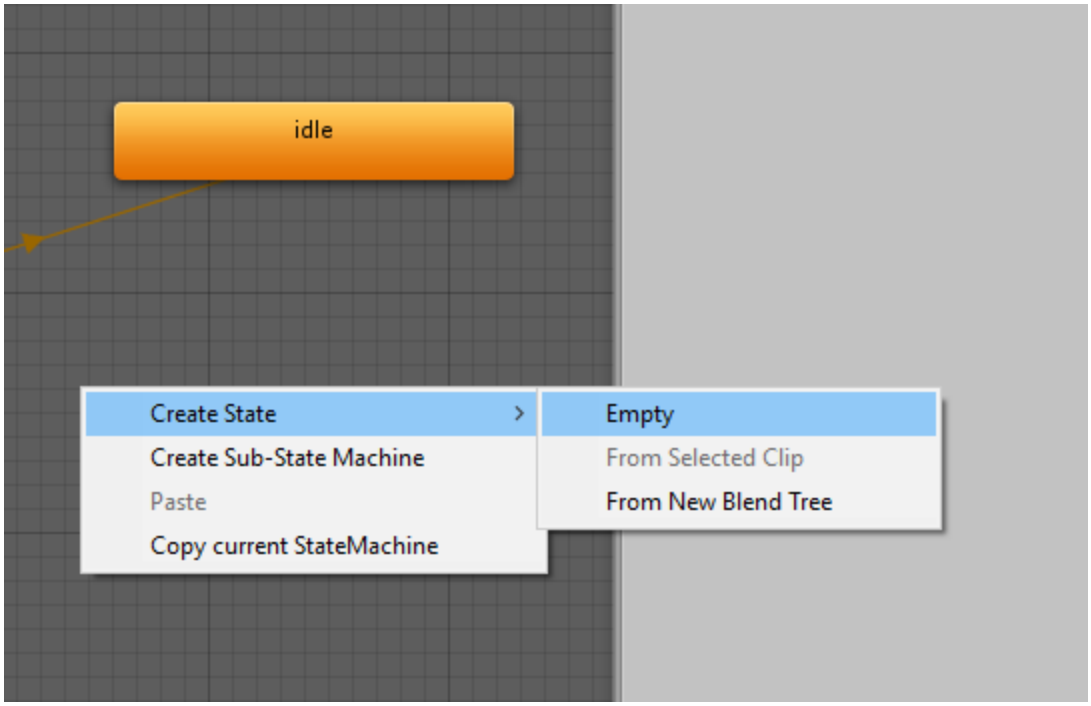
After slicing, drop the newly sliced png onto the scene. This will create an animation and an animation controller.



Setting the animation speed



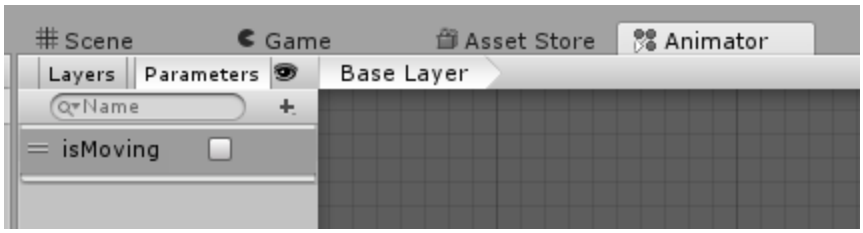
Do the same for the walking png. Notice that it creates another controller. Delete that as we only need one controller. Now edit the controller created during the idle animation creation and manually add the walking state.



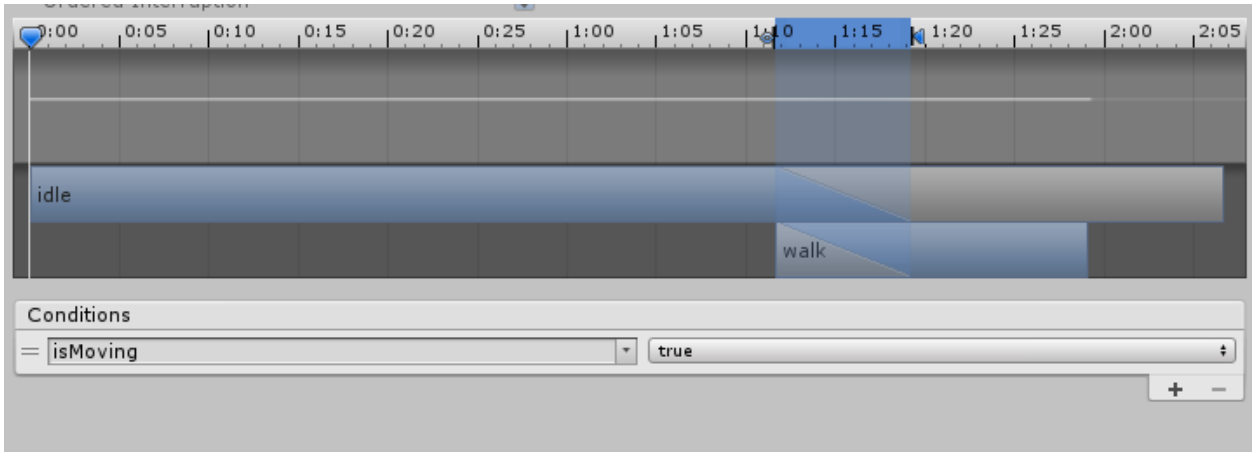
Setting the transitions

Right click the idle state and click Transition. Then drag that transition to the walk state. Do the reverse from walk back to idle state.

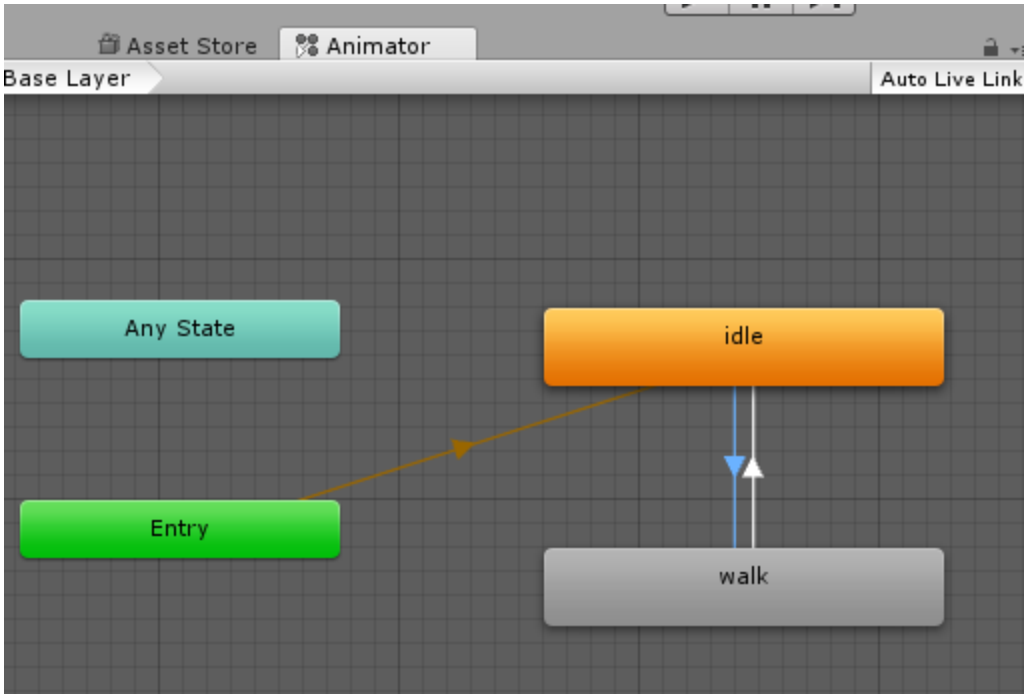
Now we need to create the code that will wire this animation up to our code. Create a new parameter called isMoving. Make it a boolean.



Now for each transition, add the condition:



When you are done, it should look like this:



Attaching the transition parameter to our code in Player.cs

Attach the animation controller to droplet game object. This is called an Animator.

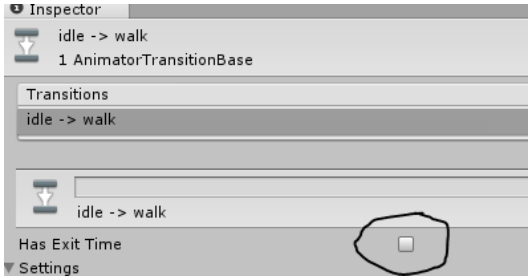
```
    private Animator animator;
    private bool facingRight = true;
    // Use this for initialization
    void Start () {
        groundCheck = GetComponent<CircleCollider2D>();
        animator = GetComponent<Animator>();
    }
```

In the Update method, add this code to set the state conditions:

```
if (move != 0) {
    animator.SetBool("isMoving", true);
}
else {
    animator.SetBool("isMoving", false);
}
```

Interrupting the animation

You will notice that the animation continues to play even after you transition. To interrupt the animation playback, do this:



Flipping the character when he changes direction

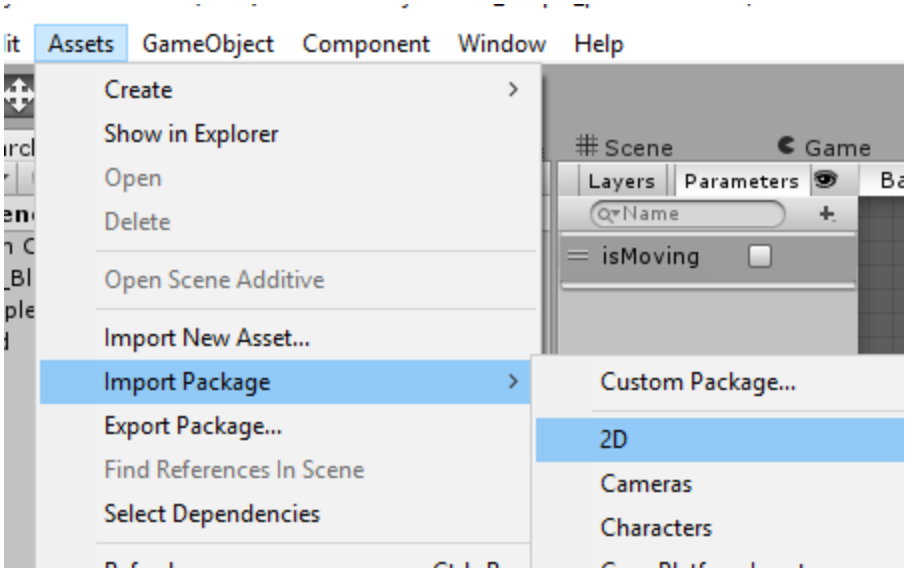
```
void Flip()
{
    //Debug.Log("switching...");
    facingRight = !facingRight;
    Vector3 theScale = transform.localScale;
    theScale.x *= -1;
    transform.localScale = theScale;
}
```

```
if (move > 0 && !facingRight)
{
    Flip();
}
else if (move < 0 && facingRight)
{
    Flip();
}
```

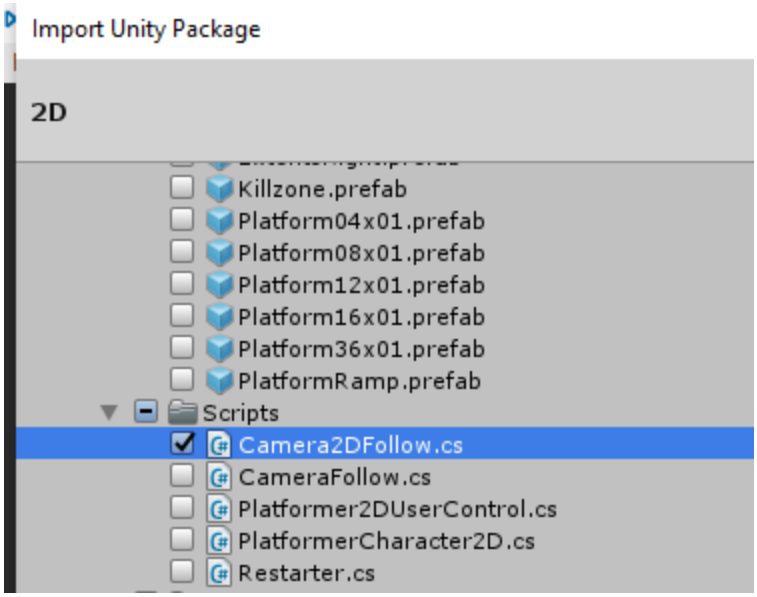
Getting the camera to follow our player

All this time our camera has been aimed at the same spot. Now it is time to have it follow the player. To do this you will want to import a standard 2D asset of some camera follow code that works.

Importing a standard asset



Deslect all, and just select Camera2DFollow.cs



Reloading a scene

Here is the code for loading a scene. You will need a new import using UnityEngine.SceneManagement;

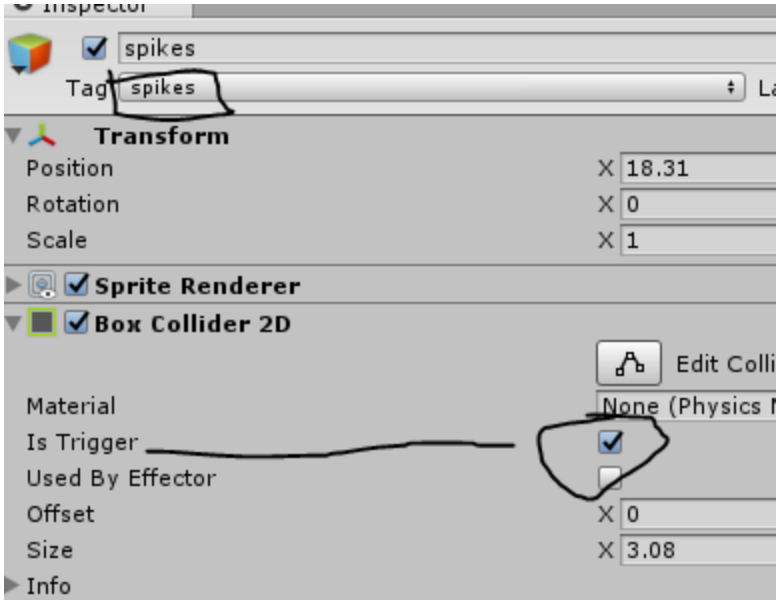
And then call this code to load a scene.

```
SceneManager.LoadScene (SceneManager.GetActiveScene ().name);
```

OnTriggerEnter2D: Setting up the spike trap to kill droplet

Add a trap and set the spike sprite on it.

Create a tag called “spikes” to the newly created game object and add a box collider 2D. This time the box collider will be a trigger.



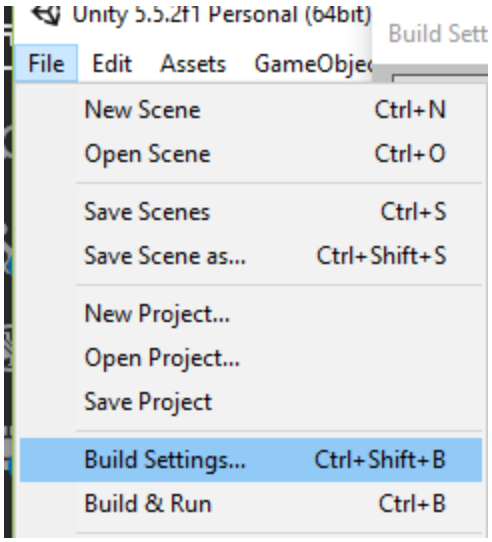
Add this code to Player.cs and now when this collision happens the current scene will reload.

```
}
void OnTriggerEnter2D(Collider2D coll){
    if (coll.gameObject.tag == "spikes" ) {
        SceneManager.LoadScene (SceneManager.GetActiveScene ().name);
    }
```

Loading a new scene

Next we are going to add new scene when the player goes through a log. To do this we are going to save a copy of our current scene and just adjust the background color to indicate night.

How the build works: attaching scenes in order to the build



Now we are going to add the log with a box collider 2D (with a trigger) for moving to the next scene.

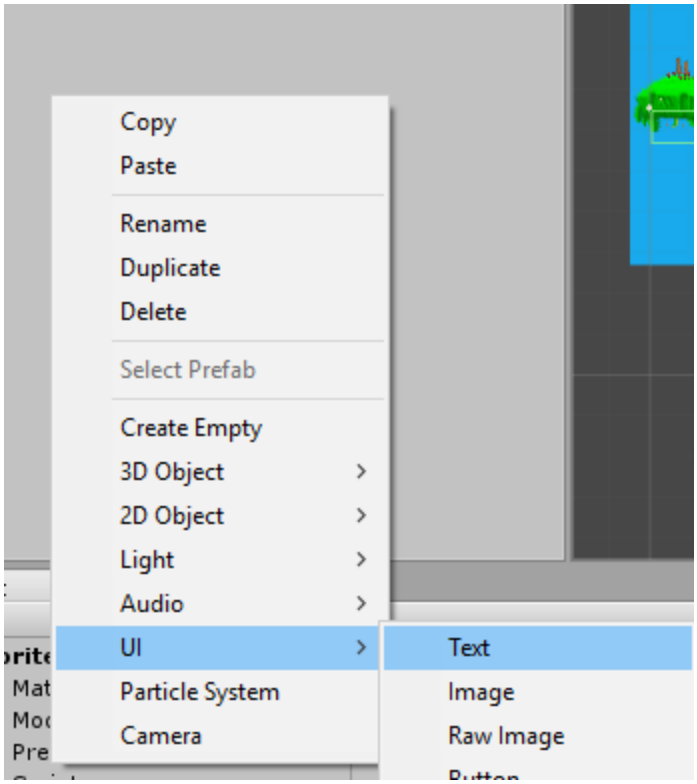
```
void OnTriggerEnter2D(Collider2D coll){
    if (coll.gameObject.tag == "spikes" ) {
        SceneManager.LoadScene (SceneManager.GetActiveScene ().name);
    }

    if (coll.gameObject.tag == "exit" ) {
        SceneManager.LoadScene("scene2");
    }
}
```

Creating a title scene

This will also be a save as from scene1. As before add this scene to the Scenes In Build and drag and drop it so that it is scene 0. Remove droplet from the scene and center the camera on something interesting.

Adding a UI

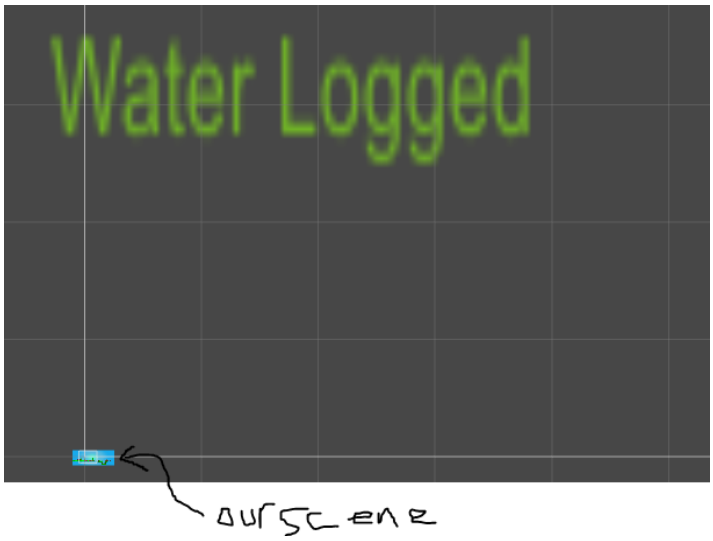
We are going to add a title and a button to start our game.



This will automatically create a Canvas object in the scene with the Text object parented under it.

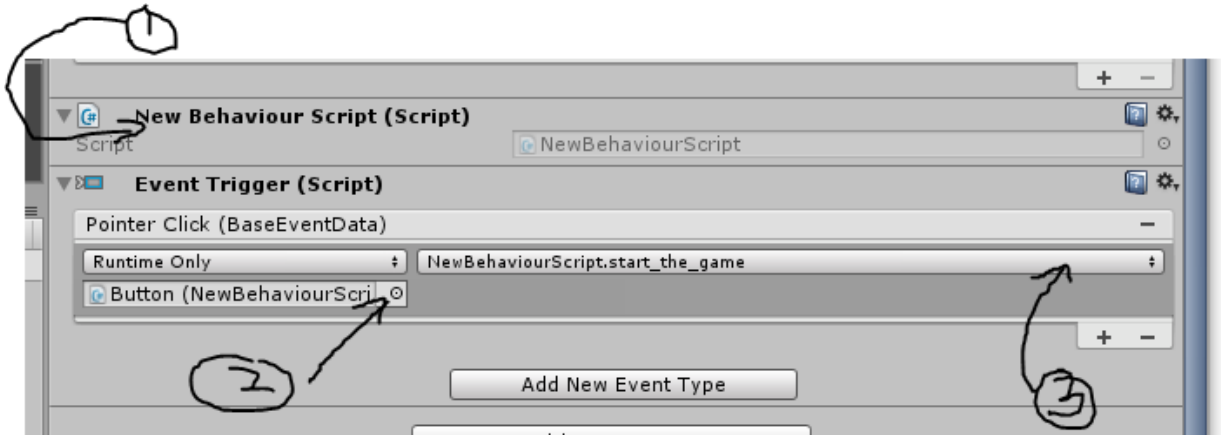
Where is the text object?

This is a little strange in how this was implemented so I'll go through it. It is located to the upper right hand side just out of the scene.



Attaching an action to a button

Under the canvas add a button. Parented under the button is the button's text. Change it to "Start". Now we are going to wire up the pointer click of our button to load a scene.



The order of the steps are important. Add a new script to the button game object. It will contain this code:

```
using UnityEngine.SceneManagement;
public class NewBehaviourScript : MonoBehaviour {

    public void start_the_game() {
        SceneManager.LoadScene("scene1");
    }
}
```

Next, add an Event Trigger component. This will need to be underneath the script component.

Then click “Add New Event Type” and select “PointerClick”. Click on the “+” button. Next (step 2 from above), take the script and drag and drop it onto where “2” points to in the above picture. Finally (3), select the NewBehaviorScript.start_the_game method.