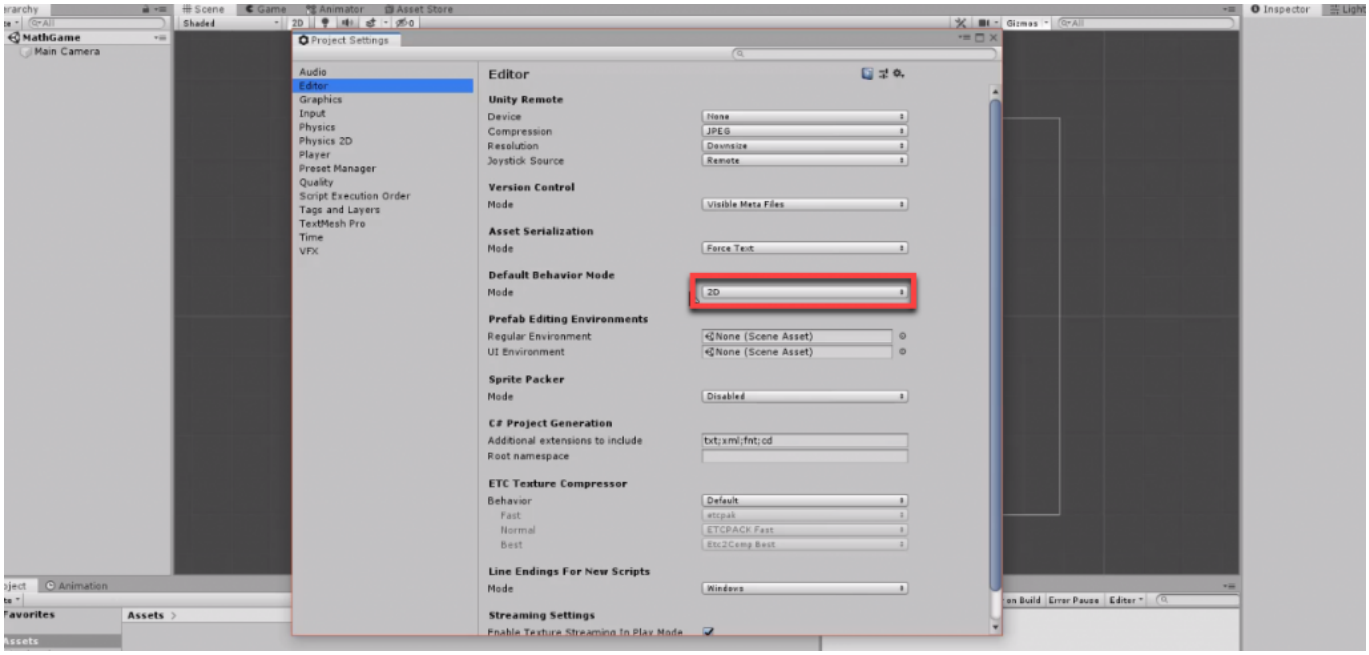
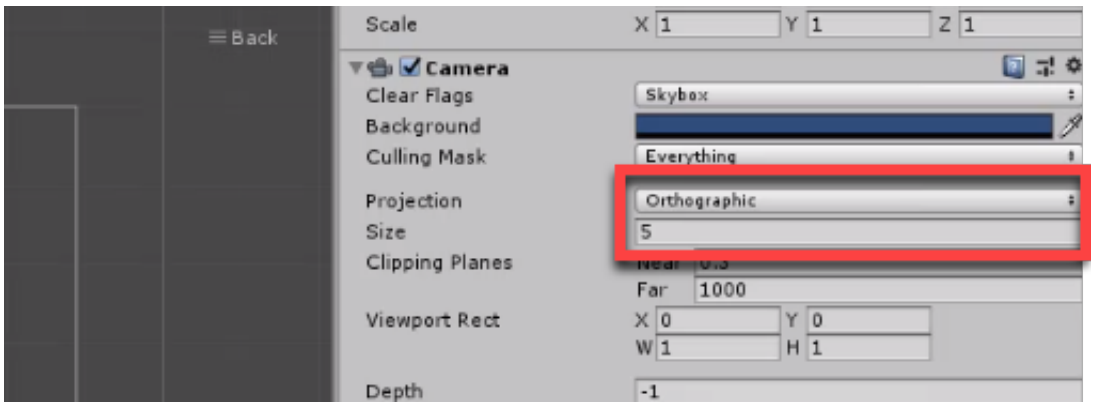
Setting Up the Project

Welcome to the course! We're going to be creating a 2D math game in Unity. First thing to do, is make sure that the Editor is in 2D mode. When creating a project you can set this, or to do that after the fact, we can go to the **Project Settings** window (*Edit > Project Settings*) and in the **Editor** tab – set **Mode** to 2D.



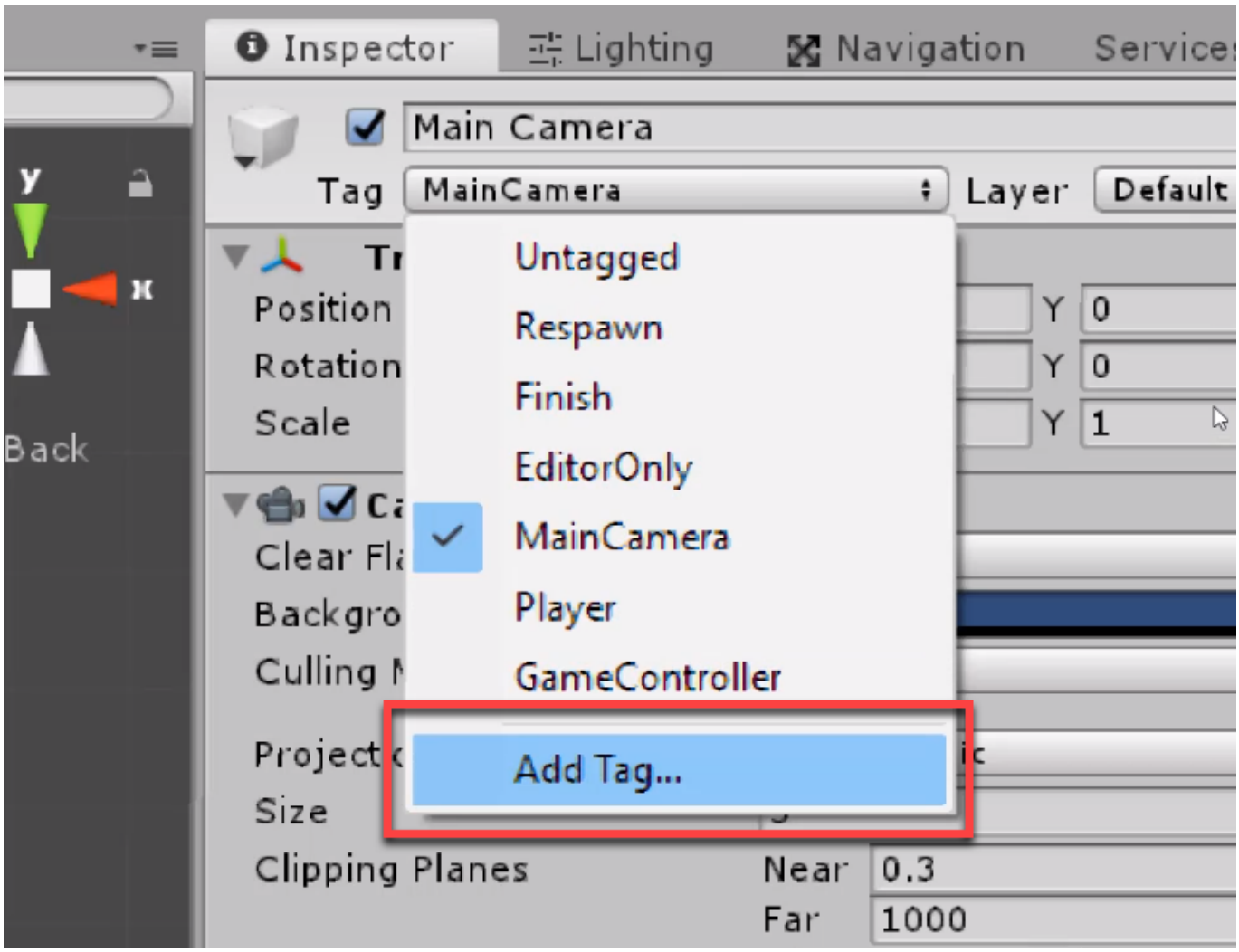
Now we need to make sure our camera is good to go.

- Set the **Projection** to Orthographic
- Set the **Size** to *5*



Tags, Layers and Sorting Layers

Now we need to setup our tags, layers and sorting layers. This is done in the **Tags & Layers** window. We can get here easily by selecting any object and at the top of the **Inspector** go *Tag > Add Tag...*



In the window, make sure the **Tags** drop-down is open and we can begin to enter in tags (click on the '**+**' icon).

- Obstacle
- ProblemTube
- Floor

Next is the **Sorting Layers**. Make sure that these are in the following order. This is because sprites with a lower sorting layer will appear below other sprites with a higher sorting layer.

- Jetpack
- Player
- Obstacle
- Ship
- UI

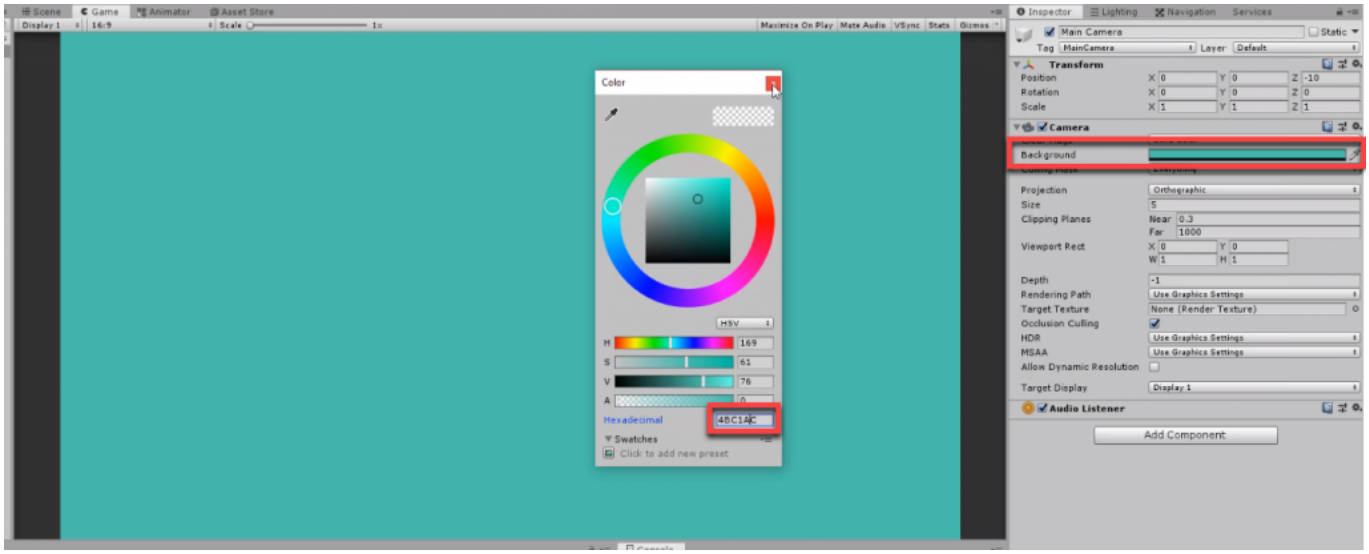
Finally for the **Layers**, we just want to add in...

- Player

Changing the Camera Background Color

The default background color isn't that good and doesn't suit our game. So let's select the camera

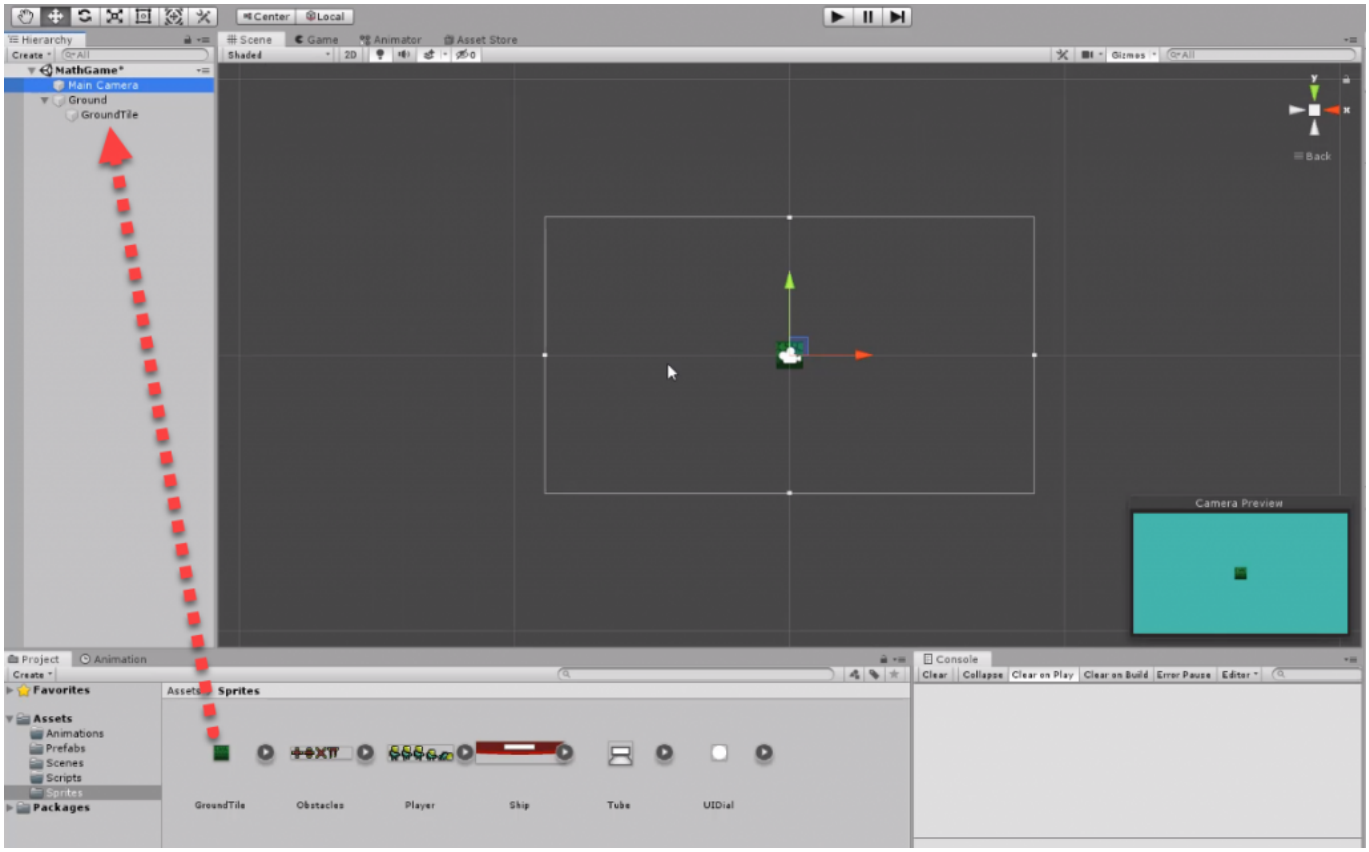
and change the **Background** color property. We're going for a blue/green teal color. The hexadecimal color i'm using is **4BC1AC**.



Ground Tiles

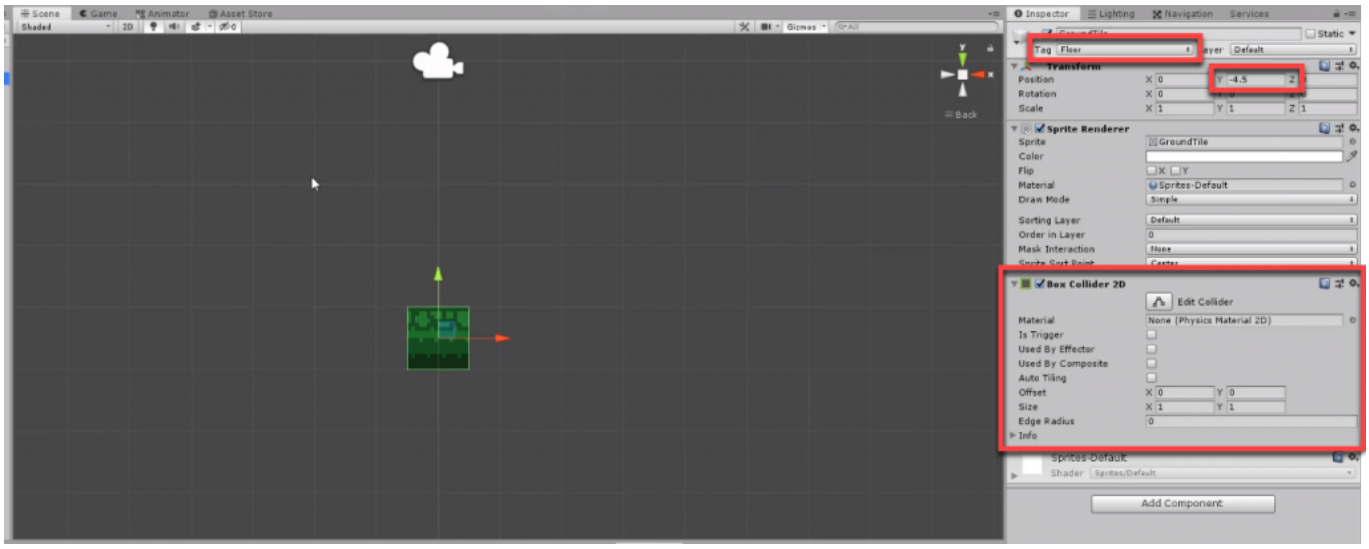
Let's start with creating the ground tiles. Create a new empty GameObject (right click **Hierarchy** window > *Create Empty*) and call it **Ground**.

Then drag the **GroundTile** sprite (> *Sprites*) into the scene as a child of **Ground**.

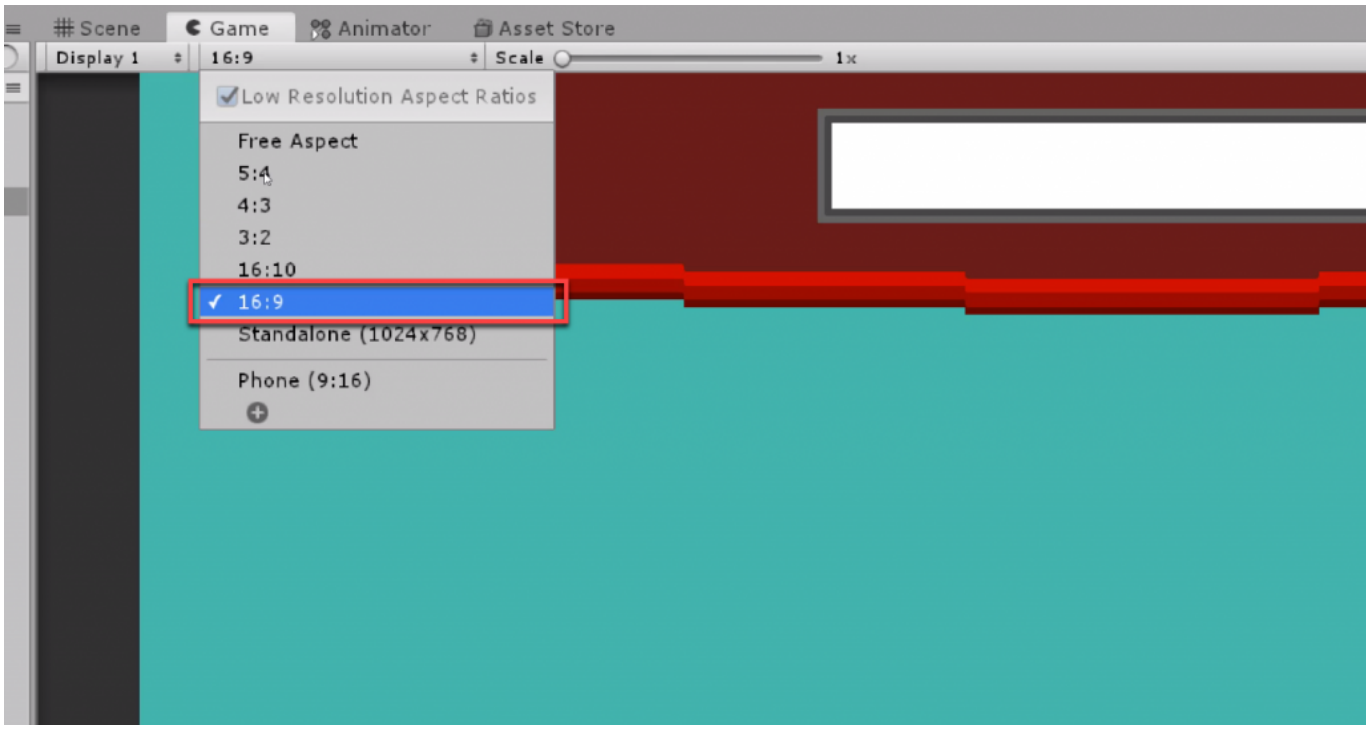


With our new ground tile object, let's set it up.

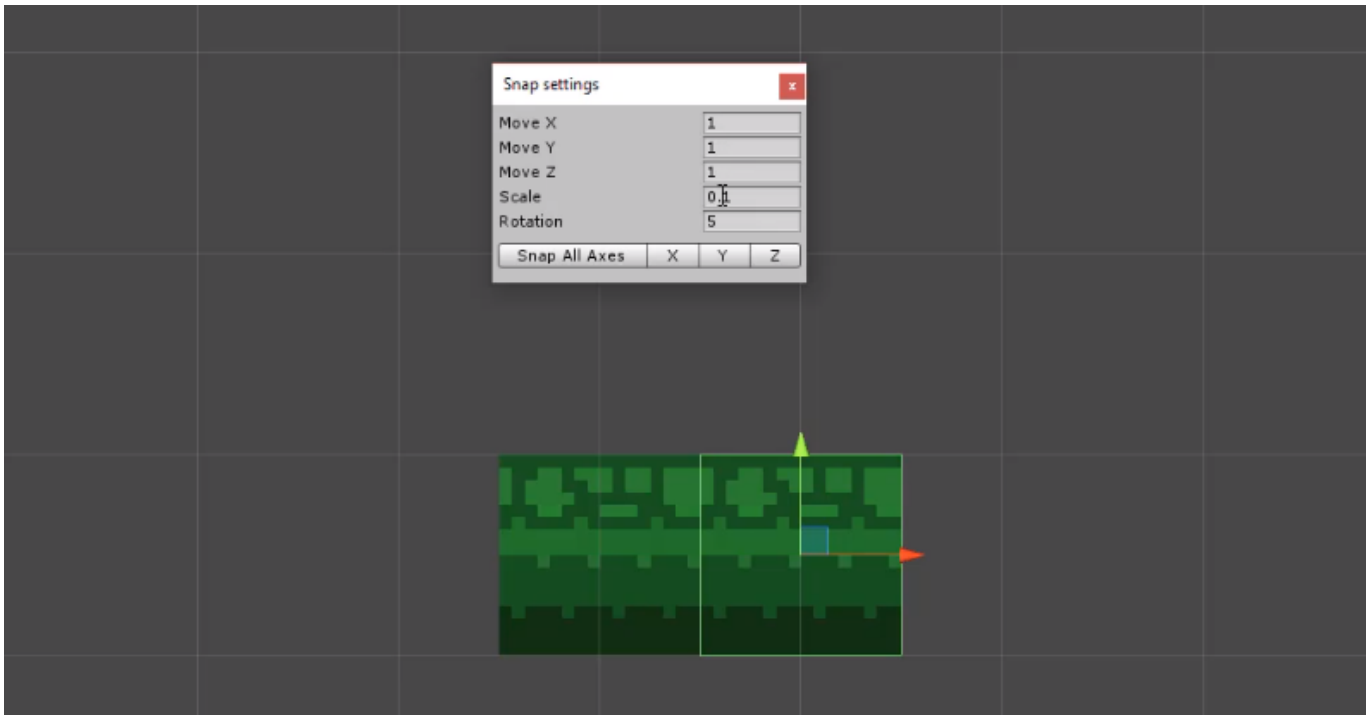
- Set the **Tag** to Floor
- Add a **BoxCollider2D** component
- Set the position to **0, -4.5, 0** so it sticks to the bottom of the screen



Since the game's 2D with a static camera, we're going to need a set aspect ratio. Go to the **Game** window and set the aspect ratio to **16:9**.



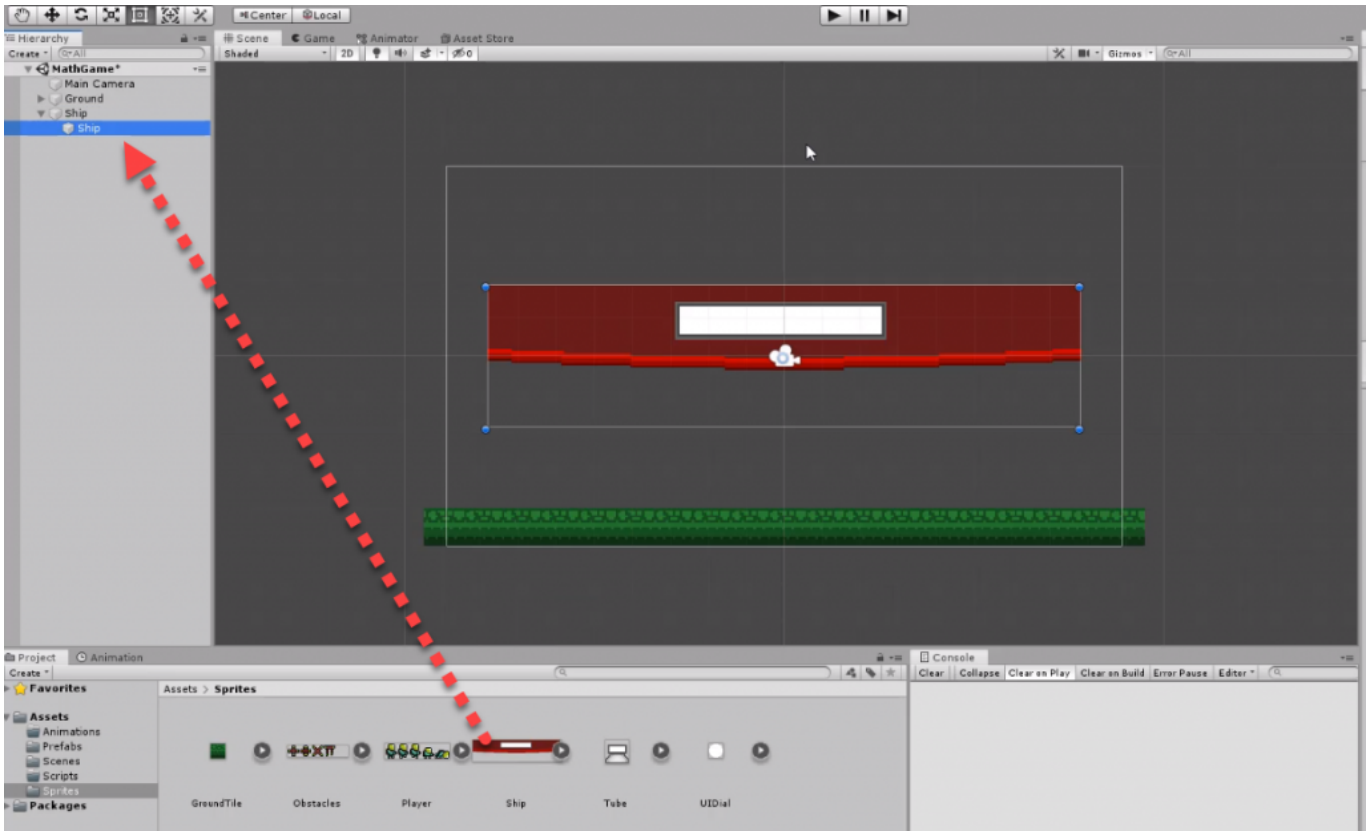
With the tile setup, let's now start duplicating them to create a flat surface along the bottom of the camera view. Selecting the tile, press '**Ctrl**' + '**C**' then '**Ctrl**' + '**V**'. With this new tile we can hold down '**Ctrl**' and use the move gizmo to snap the tile to units of 1. This can be changed in the **Snap Settings** window (*Edit > Snap Settings...*).



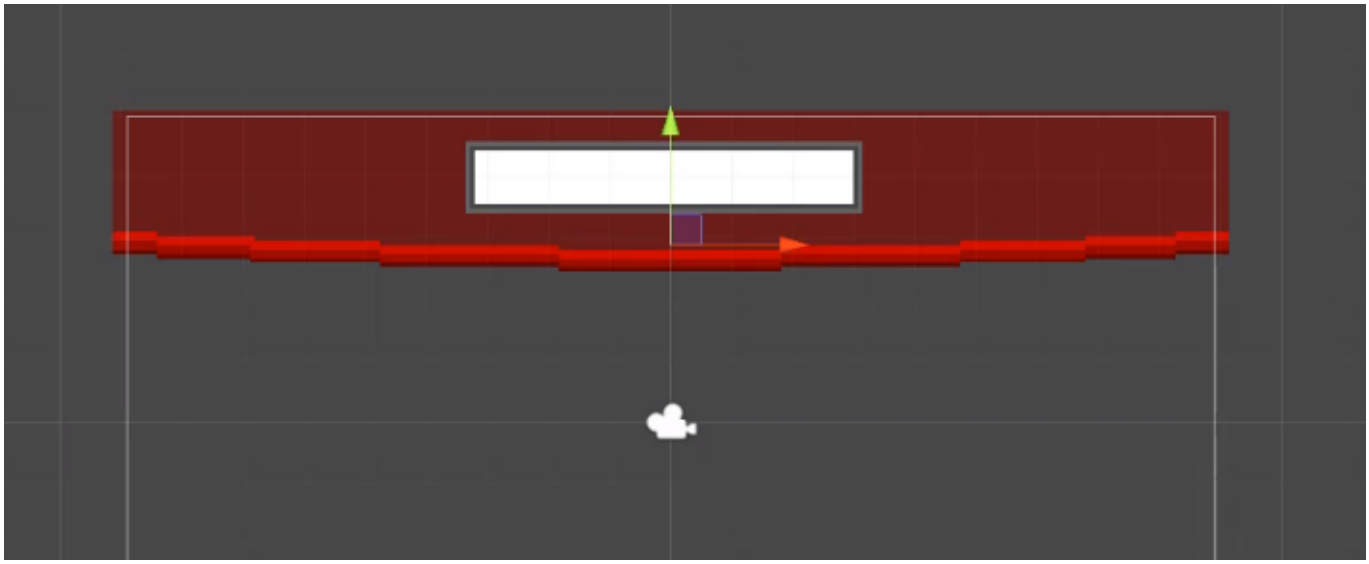
Now keep duplicating the tiles until they cover the bottom of the camera's view like so.

Setting Up the Ship

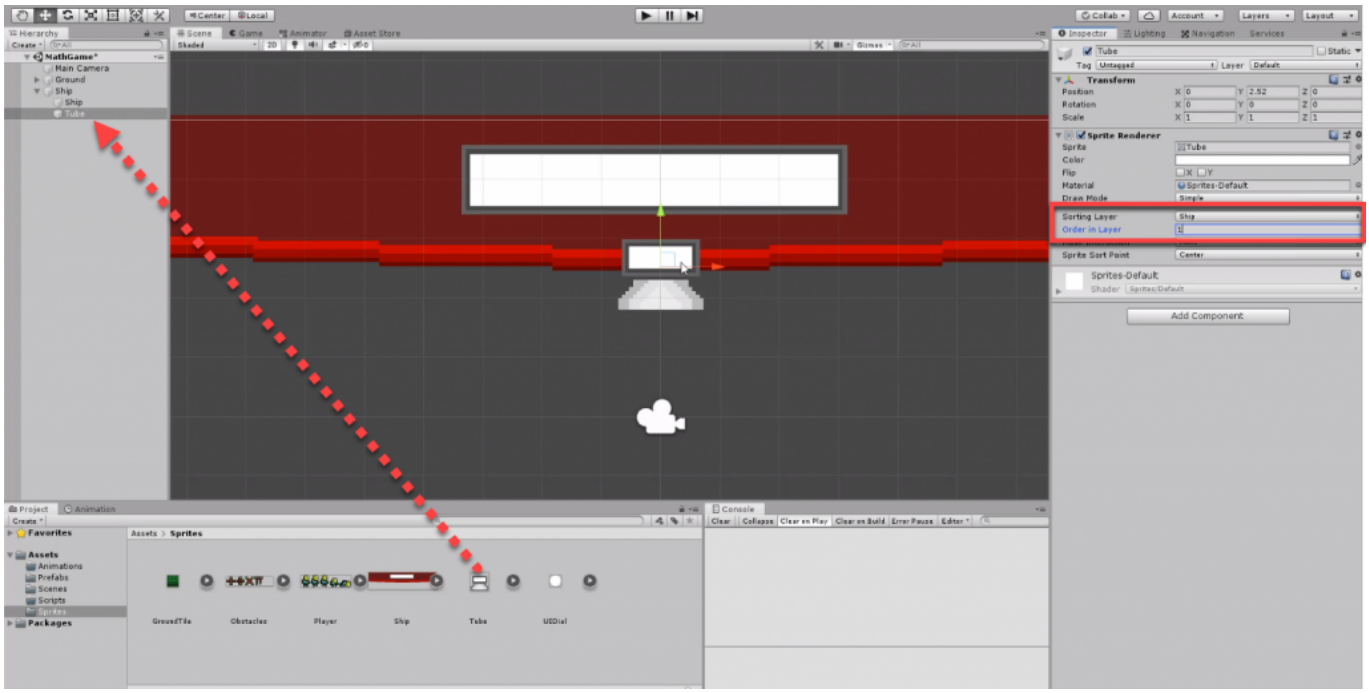
The ship is going to be the object on top of the screen that holds the problem tubes. Create a new empty GameObject and call it **Ship**. Then drag the **Ship** sprite (> *Sprites*) into the scene as a child of the **Ship** object.



It’s not at the right scale to let’s scale it up and place it at the top of the camera’s view.

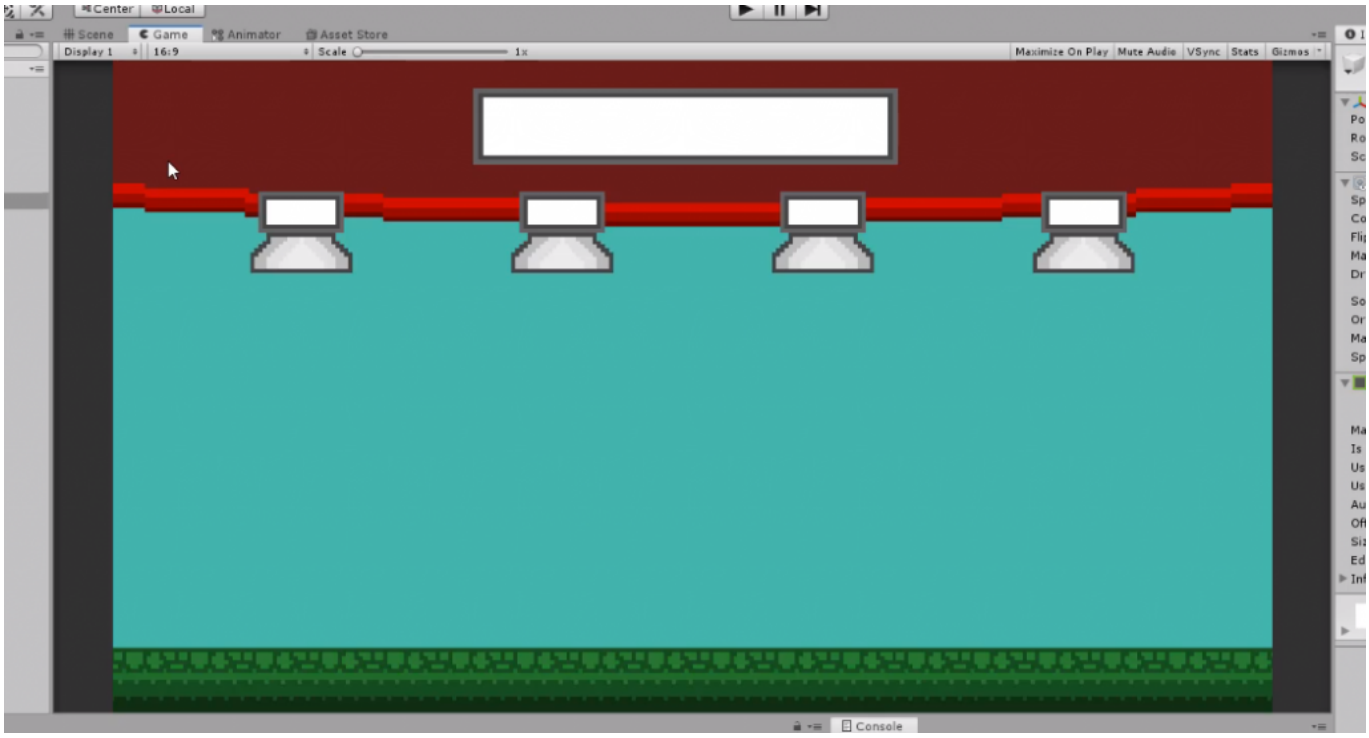


Now let’s add in the problem tubes. Drag the **Tube** sprite into the scene as a child of the **Ship** object. Set the SpriteRenderer’s **Sorting Layer** to Ship and **Order in Layer** to 1. Also make the **Ship’s** SortingLayer to Ship and Order in Layer to 1. This will make the tube be on the same layer as the ship but appear in front of it.

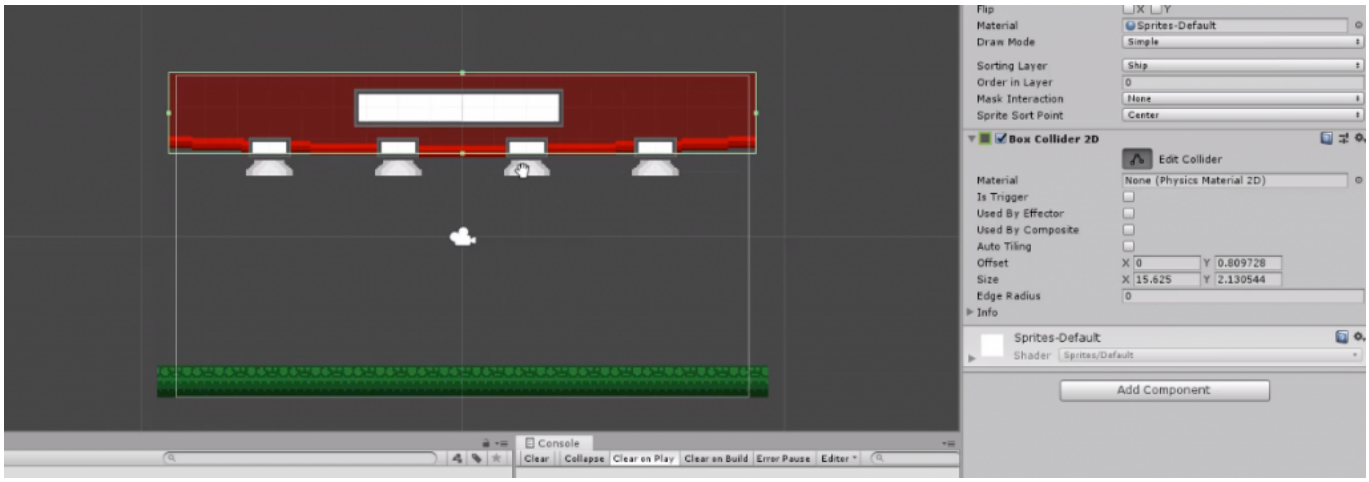


Next, set the **Tag** to ProblemTube and add a **BoxCollider2D** component, resizing it to be around half the size of the tube sprite.

Duplicate the tube and position 4 of them along the top of the screen like so.

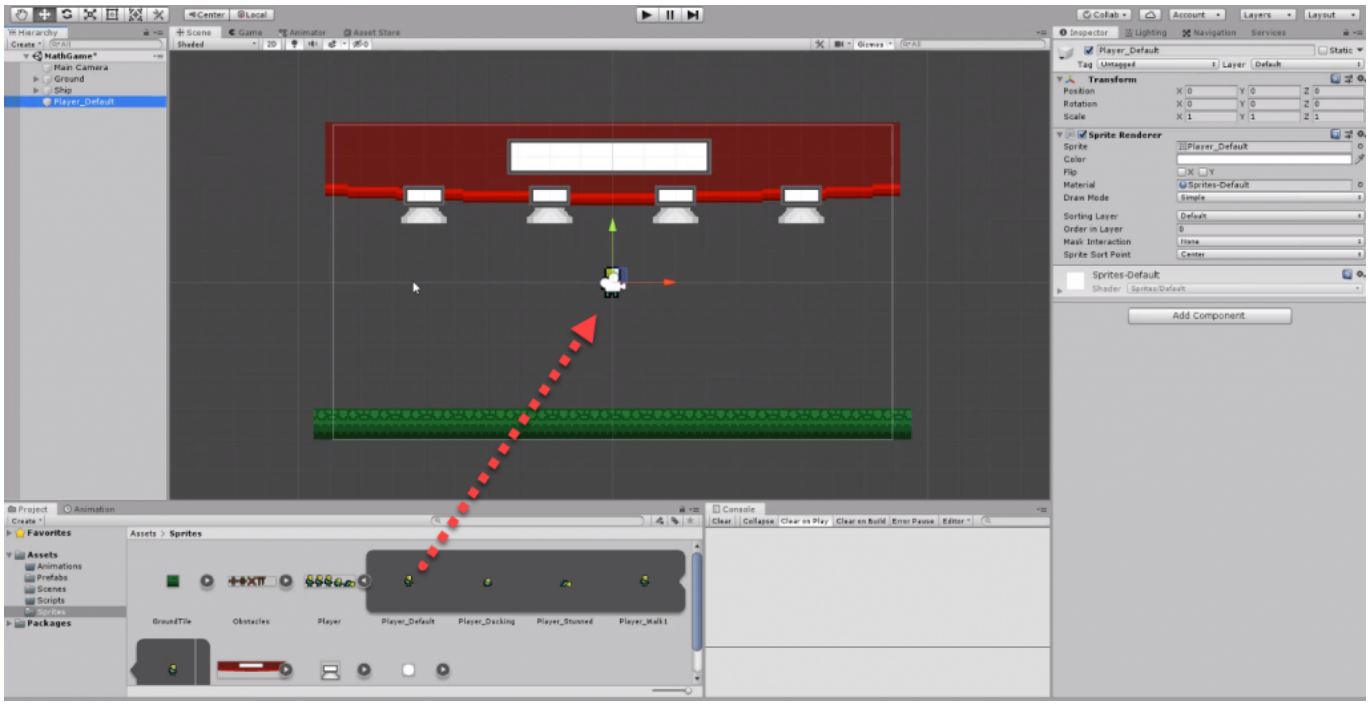


Finally, let's add a **BoxCollider2D** to the ship sprite so the player can't just fly all the way upwards.



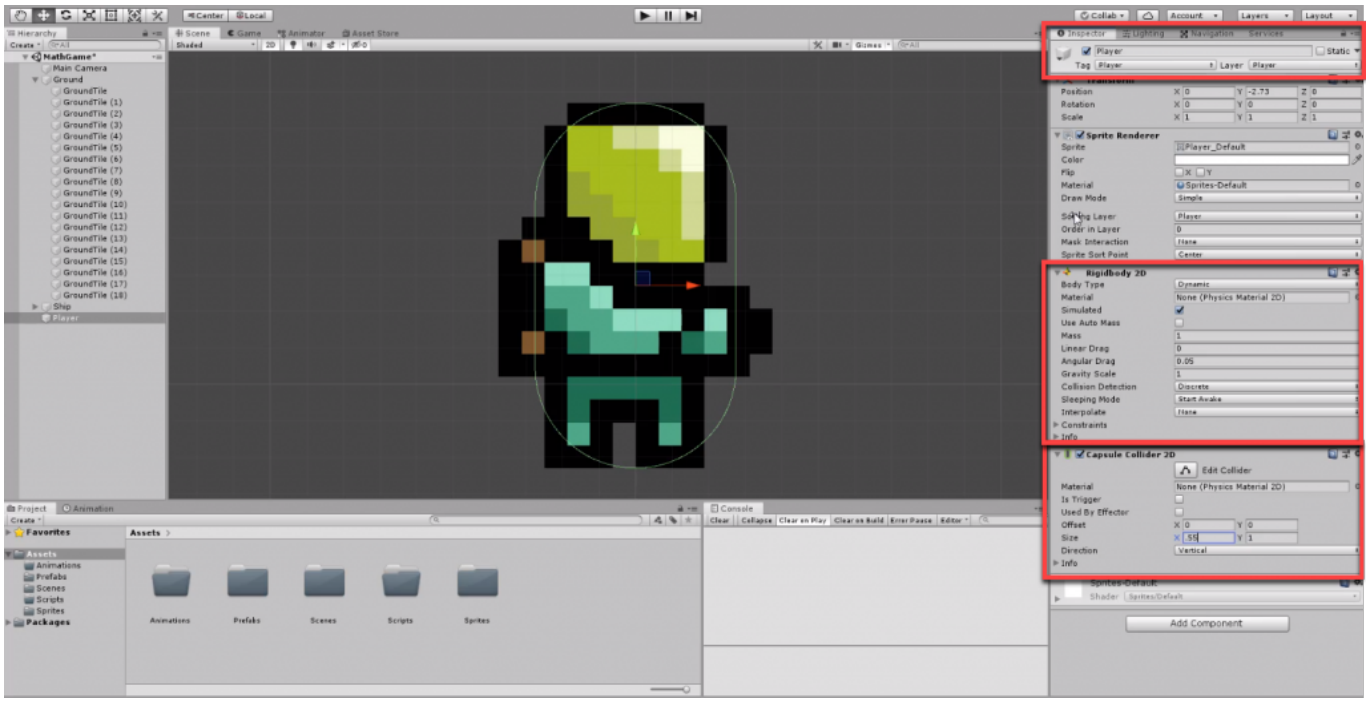
Creating the Player

It's now time to create our player. First, let's drag the **Player_Default** sprite (> *Sprites*) into the scene.

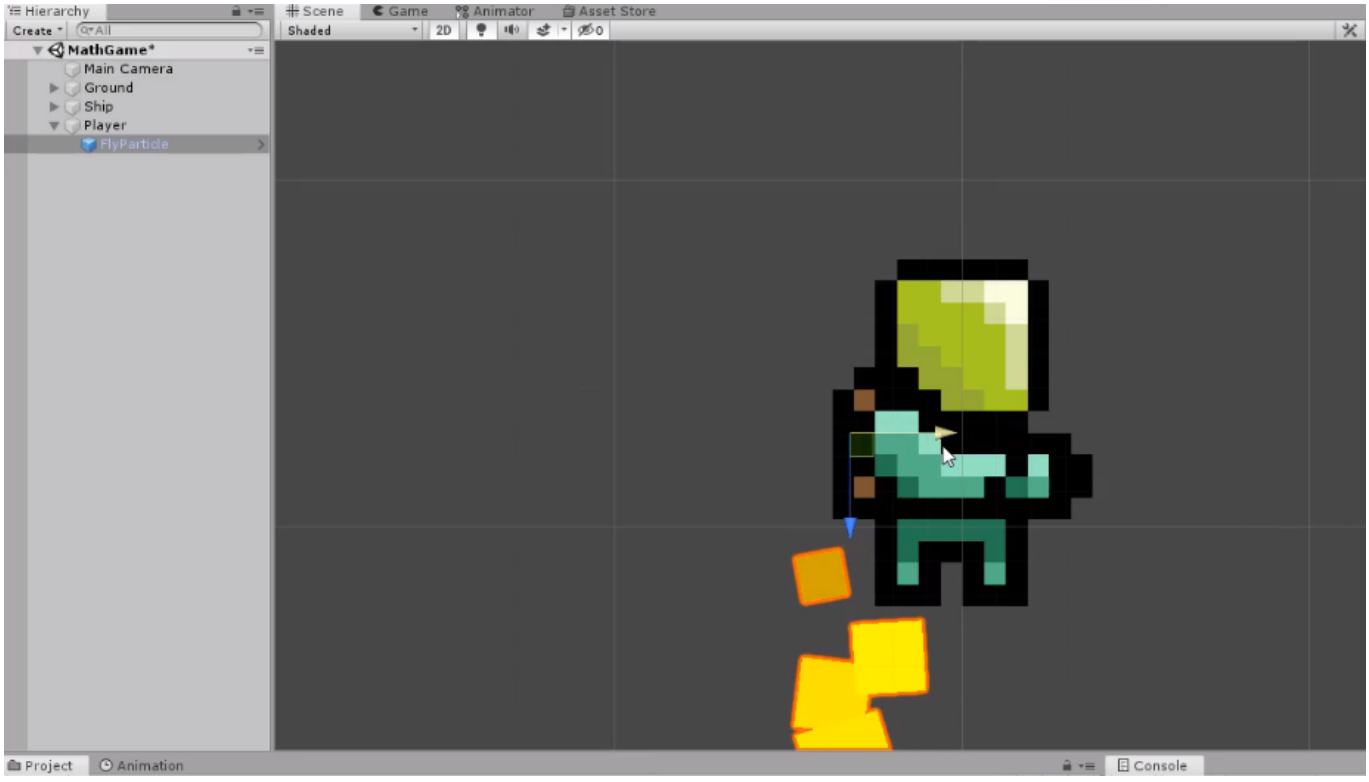


For the object, there's a few things we need to do.

- Rename the object to **Player**
- Set the tag, layer and sorting layer to **Player**
- Add a **Rigidbody2D** component
    - Freeze the Z rotation **Constraint**
- Add a **CapsuleCollider2D** component
    - Set the X size to *0.55*

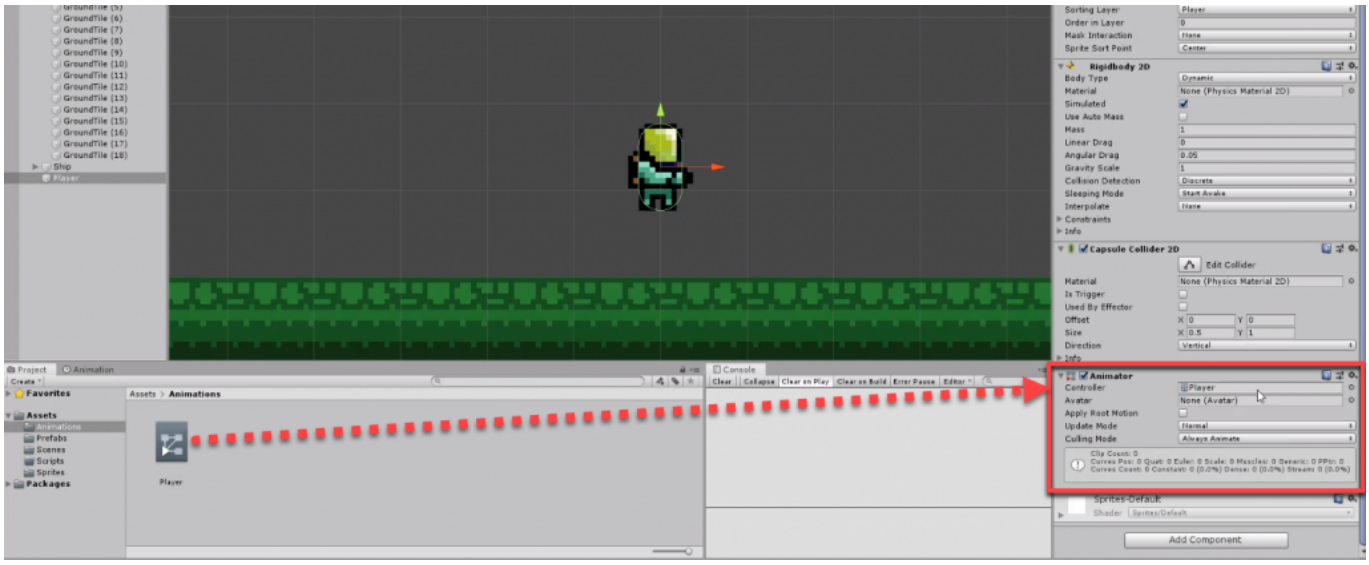


Next, let's drag the **FlyParticle** (> *Prefabs*) into the scene as a child of **Player**. Position it so the particles are flying down from the player's backpack.



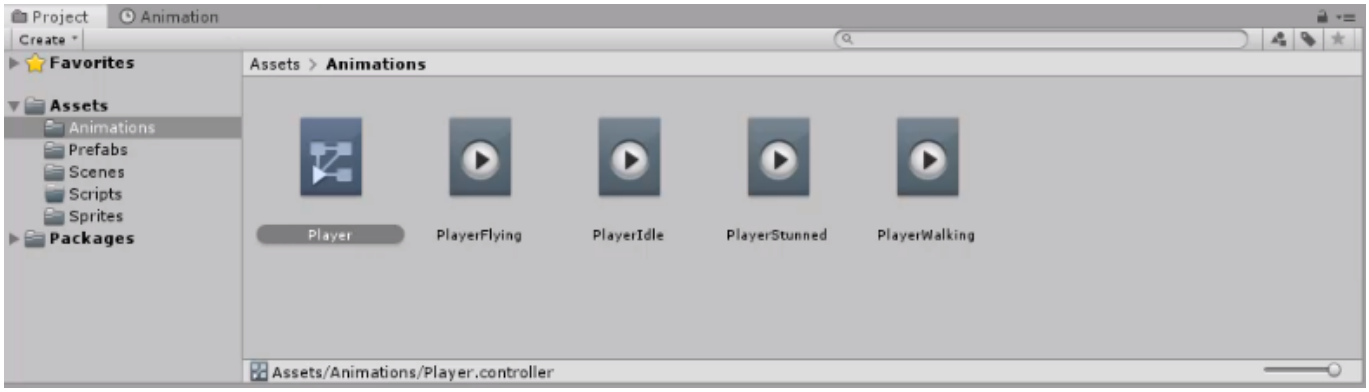
Animating the Player

We're going to be creating a few animations for our player. To do this we need to put add an **Animator** component to the player object. Then in the **Animations** folder (create one if you don't have one) create a new **Animation Controller** (right click **Project** > *Animation Controller*) and call it **Player**. Attach this to the player's Animator.

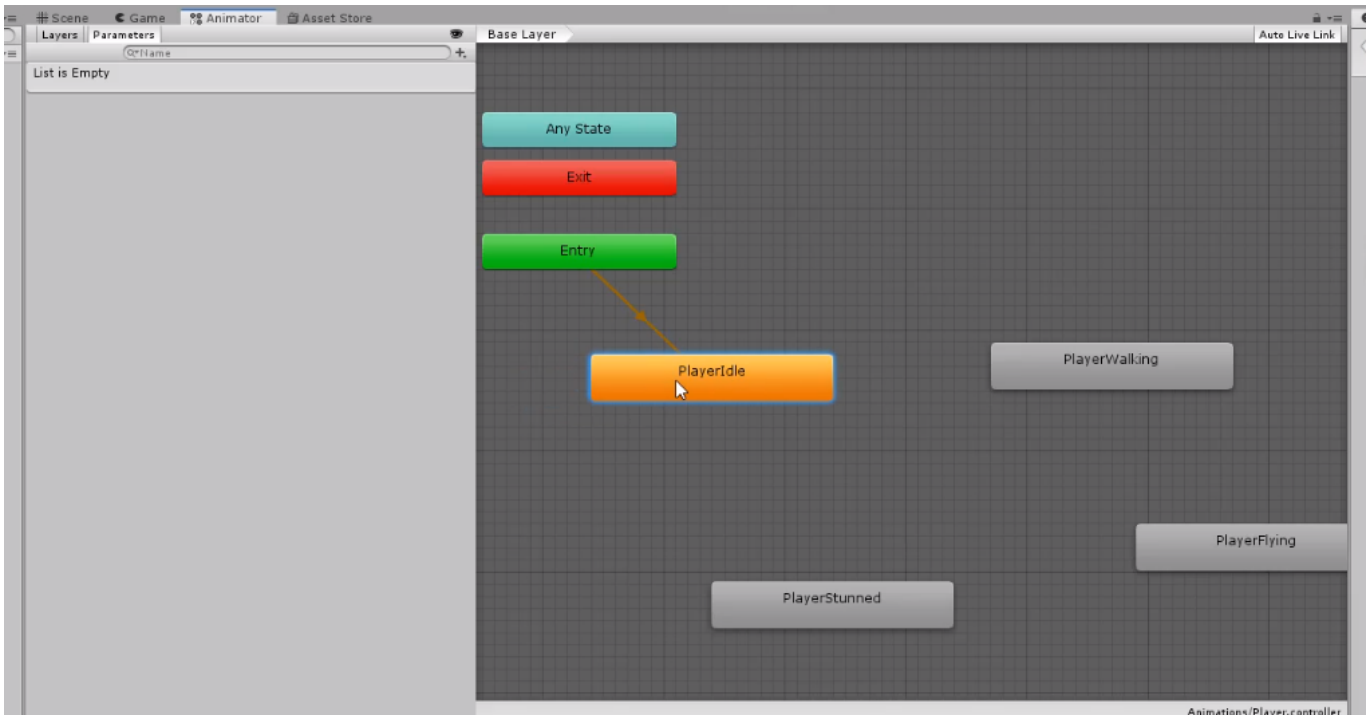


Now create 4 animation clips (right click **Project** > *Animation Clip*). These are:

- PlayerFlying
- PlayerIdle
- PlayerStunned
- PlayerWalking

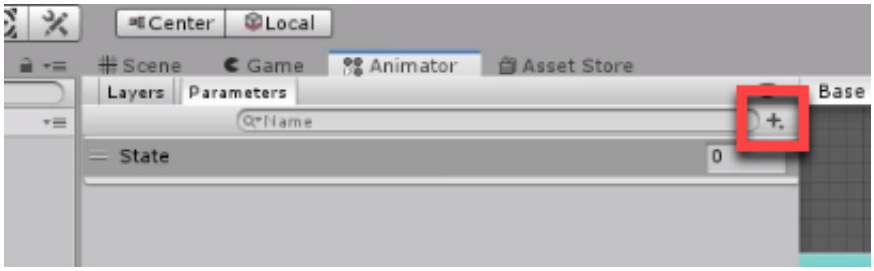


What we're going to do now is create an animation state machine. This basically links animations together and switches between them based on certain parameters. We need to open the **Animator** window (*Window* > *Animation* > *Animator*) – then double click on the **Player** animation controller to open it up. Here, we want to drag in the 4 animation clips. **Make sure the PlayerIdle animation is the first one in, so it's made the default one.**

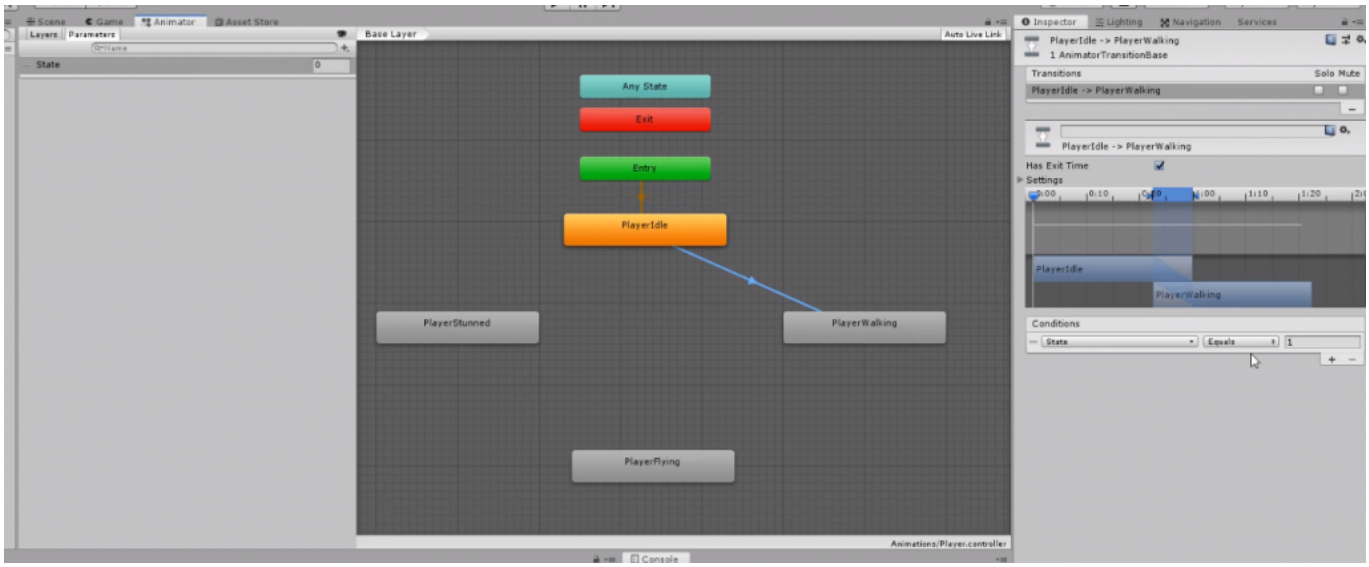


Now to switch between the animations, we're going to be checking a number parameter – State. In the top left of the **Animator** window, select **Parameters** and add a new int parameter. Call it **State**. For us, we're having 4 different states:

- 0 = idle
- 1 = walking
- 2 = flying
- 3 = stunned

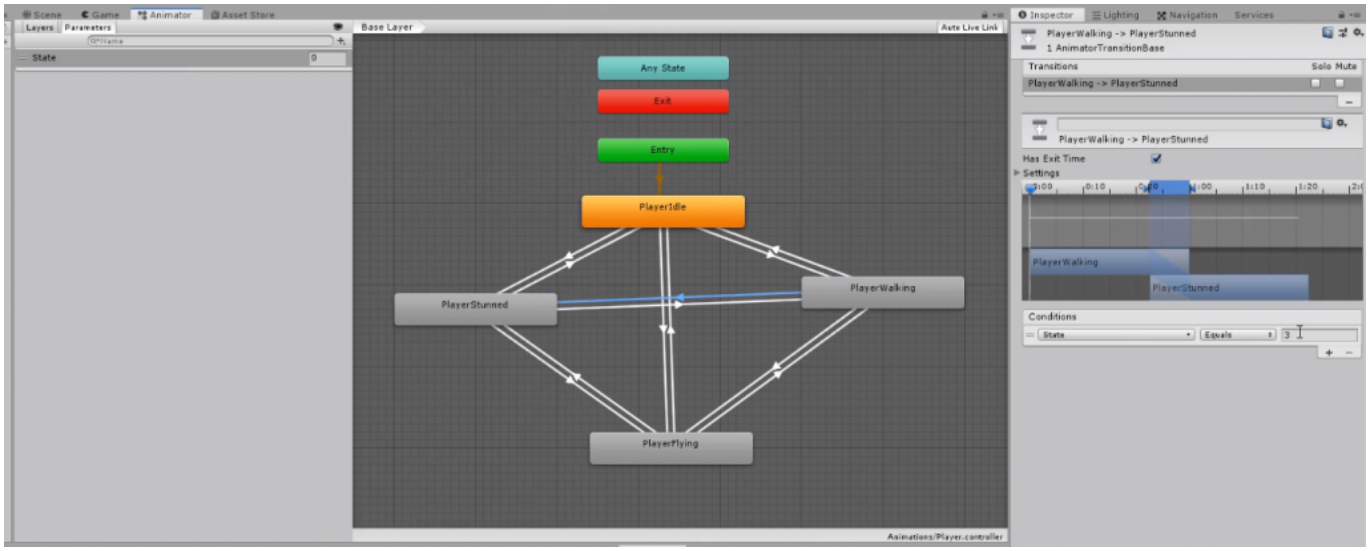


Let's begin! Right click on **PlayerIdle** and select **Make Transition**. Then we want to click on the **PlayerWalking** state. This will connect the two with a state transition. Selecting the connection, we can add a condition – state equals 1.

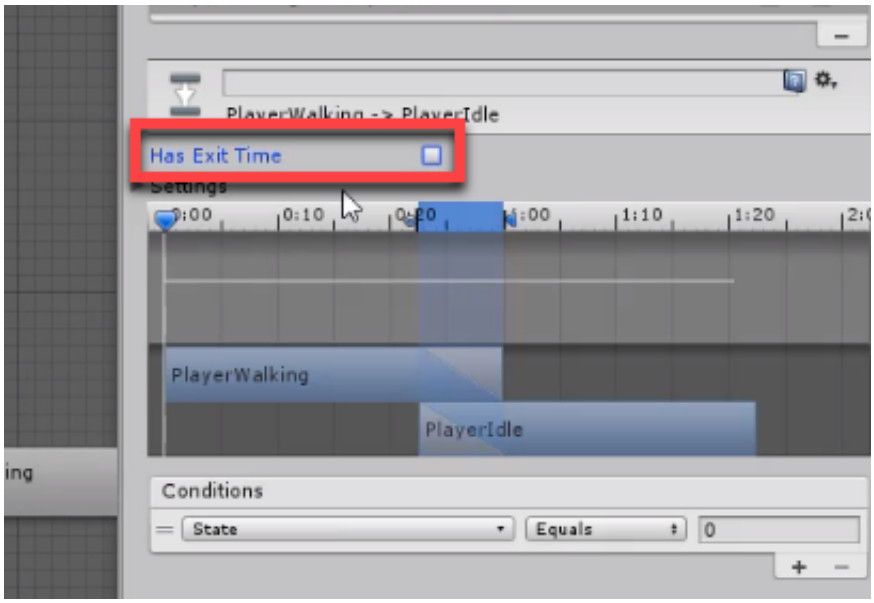


Create a connection between each state now. Make sure that the conditions are the correct states!

- PlayerIdle -> PlayerWalking (State = 1)
- PlayerIdle -> PlayerFlying (State = 2)
- PlayerIdle -> PlayerStunned (State = 3)
- PlayerWalking -> PlayerIdle (State = 0)
- PlayerWalking -> Player Flying (State = 2)
- PlayerWalking -> PlayerStunned (State = 3)
- PlayerFlying -> PlayerIdle (State = 0)
- PlayerFlying -> PlayerWalking (State = 1)
- PlayerFlying -> PlayerStunned (State = 3)
- PlayerStunned -> PlayerIdle (State = 0)
- PlayerStunned -> PlayerWalking (State = 1)
- PlayerStunned -> PlayerFlying (State = 2)



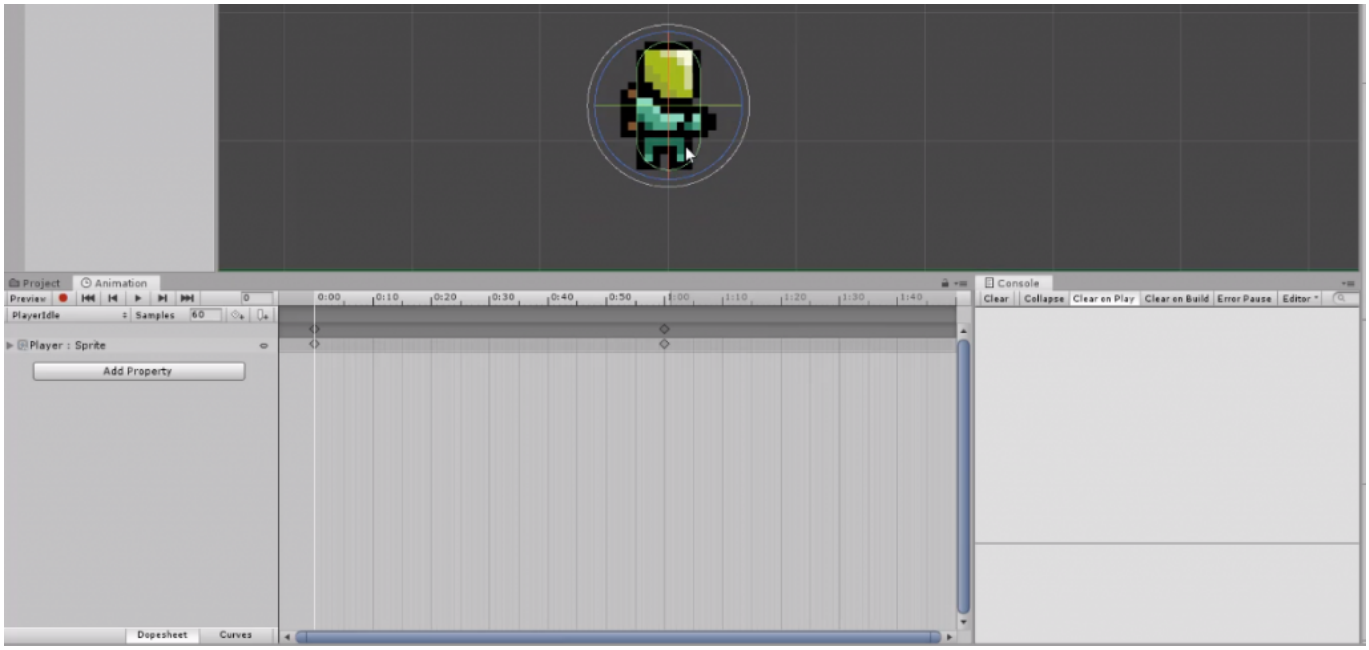
For each of the state transitions, we want to disable **Has Exit Time**. This means there won't be any transition between the animations – it will be instant. When working with pixel art frame animation, that's what we want.



In the next lesson, we'll be animating the player.

Player Idle Animation

First up is the **PlayerIdle** animation. This is just going to be playing the **Player_Default** sprite. To do this, click on the **Add Property** button and select the player's SpriteRenderer's Sprite property. This will create two keyframes – place them around 0.2 seconds apart.

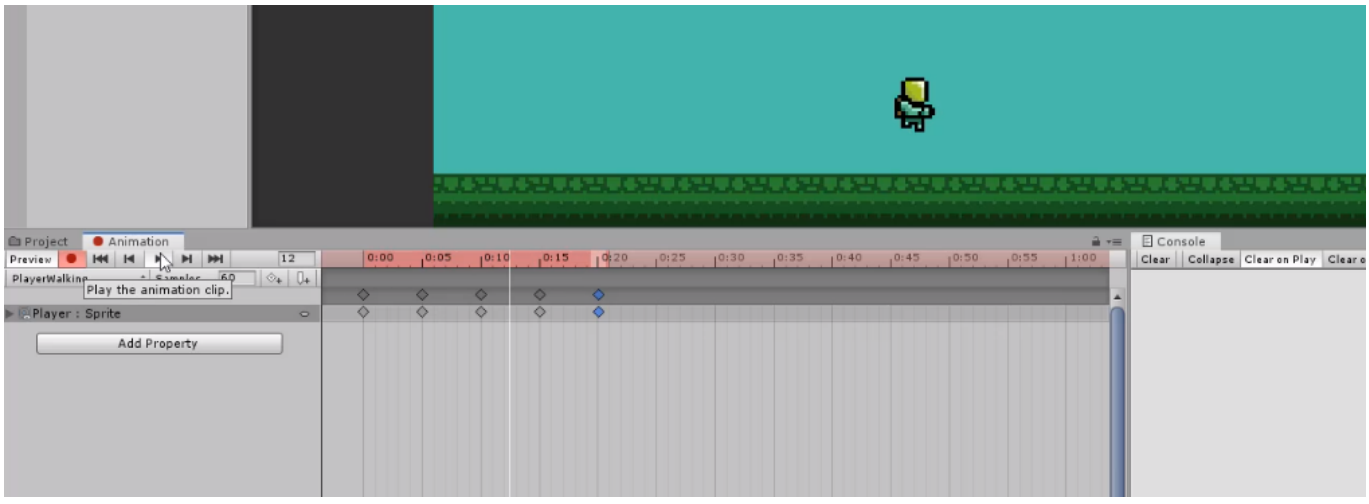


Player Walking Animation

Now let's create the **PlayerWalking** animation. This is 5 keyframes and goes through 3 different sprites.

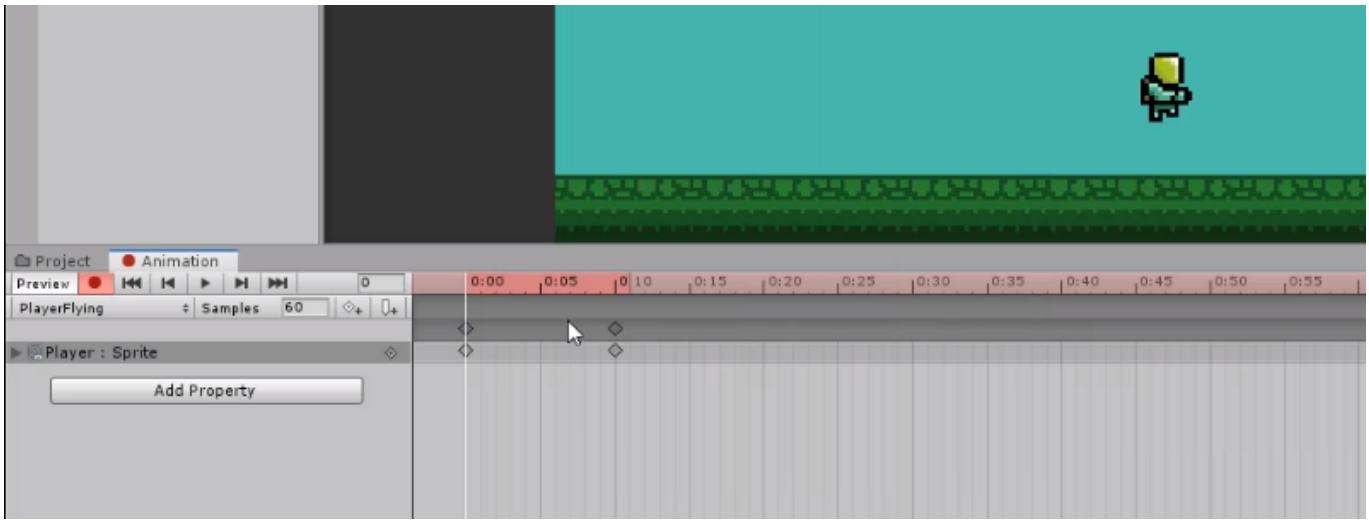
- Frame 1 = **Player_Walk1**
- Frame 2 = **Player_Default**
- Frame 3 = **Player_Walk2**
- Frame 4 = **Player_Default**
- Frame 5 = **Player_Default**

Make each frame 0.05 seconds apart with a total length of 0.2 seconds.



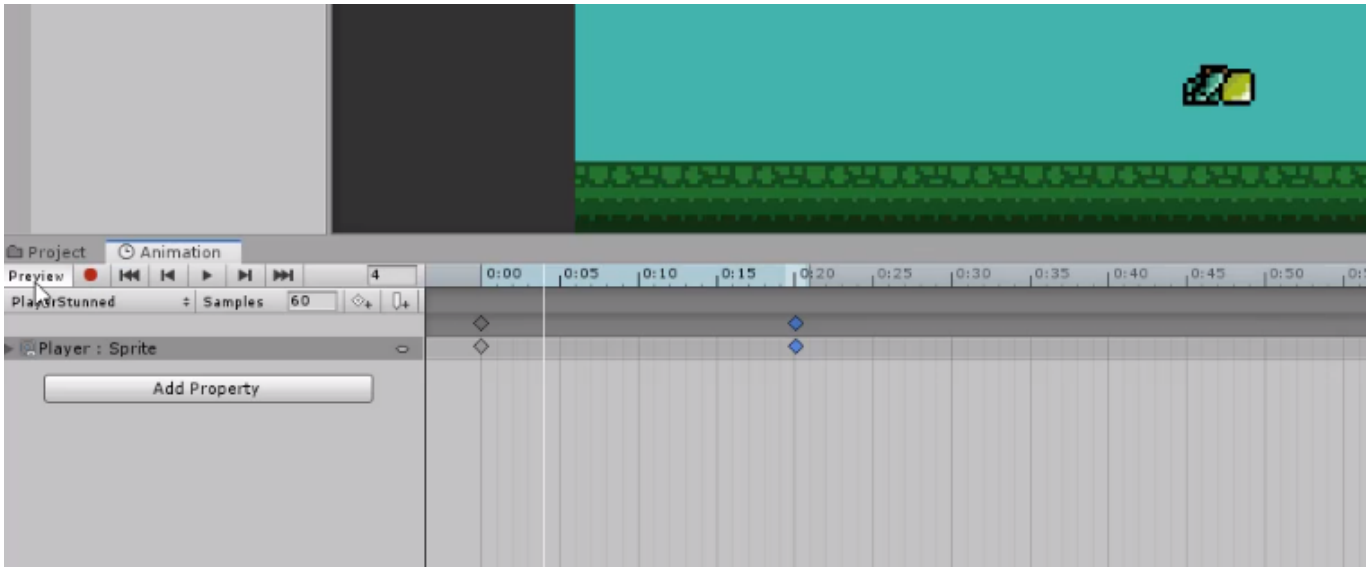
Player Flying Animation

The **Player_Flying** animation is similar to the idle one, yet it's just the **Player_Walk1** sprite.



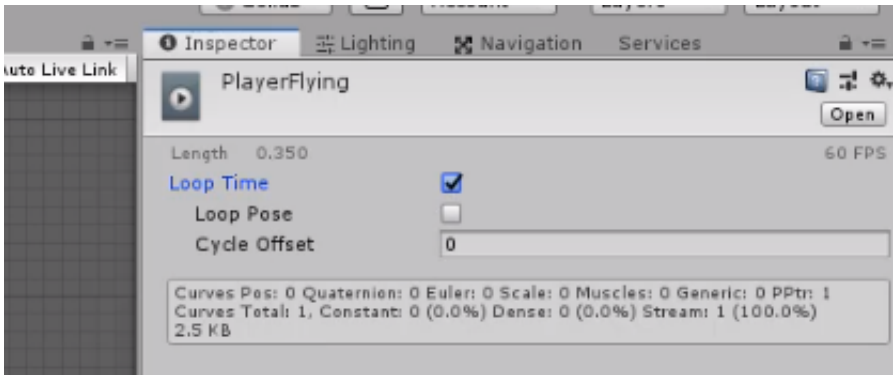
Player Stunned Animation

The **Player_Stunned** animation is again similar to the previous two, with the sprite being **Player_Stunned**.



Looping Animations

Right now, the animations play but they stop. This is because they're not looping. To fix this, for each of the animation clips, enable **Loop Time**.



Now the animations are setup! We can't really see them in action yet, so in the next lesson we'll be creating the player controller.

Scripting the Character Controller

Let's now begin scripting the character controller. Create a new folder called **Scripts** and inside of that, create a new C# script (right click **Project** window > *Create* > *C# Script*) called **PlayerController**.

The first thing we're going create, is our **PlayerState** enumerator. This is basically a list of all the available player states.

```
public enum PlayerState
{
    Idle = 0,
    Walking = 1,
    Flying = 2,
    Stunned = 3
}
```

Now let's create a variable to hold our current state.

```
// player's current state
public PlayerState curState;
```

Then we have our move speed and jetpack force.

```
// player's horizontal velocity
public float moveSpeed;

// player's upwards jetpack force
public float flyingForce;
```

We also need to know if the player is grounded. This boolean will be updated each frame.

```
// is the player on the ground?
private bool grounded;
```

When the player gets stunned, there's info we need.

```
// duration of stun
public float stunDuration;
private float stunStartTime;
```

Finally, we need to get our components.

```
// components
public Rigidbody2D rig;
```



```
public Animator anim;
public ParticleSystem jetpackParticle;
```

Laying Out the Functions

What we're going to do now, is create all of the functions we're going to be using. First up is **FixedUpdate**. This is where we're going to do physics based stuff.

```
void FixedUpdate () { }
```

CheckInputs checks for keyboard inputs, calling the respective functions.

```
// checks for user inputs
void CheckInputs () { }
```

SetState checks on things like velocity, grounded, etc to figure out what state the player is in.

```
// setting the player's state and changing animation
void SetState () { }
```

Move moves the player horizontally.

```
// move the player horizontally
void Move () { }
```

Fly adds force upwards.

```
// add force upwards to the player
void Fly () { }
```

Stun stuns the player, restricting input and forcing the downwards.

```
// stuns the player
public void Stun () { }
```

IsGrounded shoots a raycast downwards to check if the player is standing on a ground tile.

```
// returns true if grounded, false otherwise
bool IsGrounded ()
{
    return true;
```



```
}
```

OnTriggerEnter2D gets called when the object's collider enters the collider of another. We're using this to check if the player gets hit by an obstacle.

```
void OnTriggerEnter2D (Collider2D collision) { }
```

In the next lesson, we'll start to script these functions.

FixedUpdate Function

Let's start with the **FixedUpdate** function. First, let's set the grounded boolean to be the result of the **IsGrounded** function.

```
grounded = IsGrounded();
```

Then let's check for inputs.

```
CheckInputs();
```

Now we need to check if the player's stunned. If the stun duration has exceeded the stun time, un-stun them.

```
// is the player stunned?
if(curState == PlayerState.Stunned)
{
    // has the player been stunned for the duration
    if(Time.time - stunStartTime >= stunDuration)
    {
        curState = PlayerState.Idle;
    }
}
```

CheckInputs Function

In the **CheckInputs** function, let's only check for inputs if we're not stunned. At the end let's also update our state.

```
if(curState != PlayerState.Stunned)
{
    Move();

    // flying
    if(Input.GetKey(KeyCode.UpArrow))
        Fly();
    else
        jetpackParticle.Stop();
}

// update our state
SetState();
```

SetState Function

The **SetState** function will set the player's curState based on various factors. First, let's only update our state if we're not stunned.


```
// don't do anything if the player is stunned
if(curState != PlayerState.Stunned)
{

}
```

Inside of that, let's first check for setting the Idle state. We're idle if we're not moving and we're grounded.

```
// idle
if(rig.velocity.magnitude == 0 && grounded)
    curState = PlayerState.Idle;
```

We're Walking if our horizontal velocity is not 0 and we're grounded.

```
// walking
else if(rig.velocity.x != 0 && grounded)
    curState == PlayerState.Walking;
```

We're flying if our velocity isn't 0 and we're not grounded.

```
// flying
if(rig.velocity.magnitude != 0 && !grounded)
    curState = PlayerState.Flying;
```

Finally, outside of the if statement at the end of the function – let's update our animator to play the correct animation.

```
// tell the animator we've changed states
anim.SetInteger("State", (int)curState);
```

In the next lesson, we'll be working on the movement functions.

Move Function

Let's work on the **Move** function. First, we need to get the horizontal axis. This is basically a number ranging from -1 to 1 triggered by the 'A' & 'D' buttons, or the 'Left Arrow' & 'Right Arrow' buttons.

```
// get horizontal axis
float dir = Input.GetAxis("Horizontal");
```

We also want to flip the player's sprite based on the direction they're moving.

```
// flip player to face moving direction
if(dir > 0)
    transform.localScale = new Vector3(1, 1, 1);
else if(dir < 0)
    transform.localScale = new Vector3(-1, 1, 1);
```

Finally, let's set our rigidbody's velocity to make them move.

```
// set the rigidbody velocity
rig.velocity = new Vector2(dir * moveSpeed, rig.velocity.y);
```

Fly Function

In the **Fly** function, we'll add force upwards to the player.

```
// add force upwards
rig.AddForce(Vector2.up * flyingForce, ForceMode2D.Impulse);
```

Let's also play the jetpack particle.

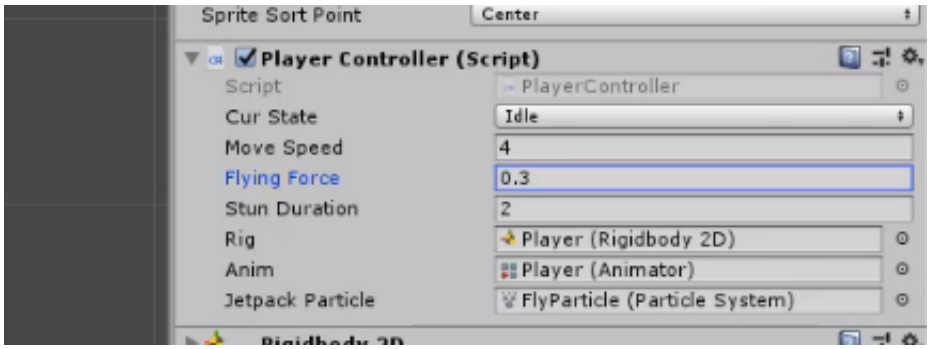
```
// play jetpack particle
if(!jetpackParticle.isPlaying)
    jetpackParticle.Play();
```

Back to the Editor

With the function's we've setup – let's test it out in the editor! The best values I found for the player controller are:

- **Move Speed** = 4
- **Flying Force** = 0.3
- **Stun Duration** = 2

Also make sure to attach the three components.



In the next lesson, we'll be finishing of our functions.

Stun Function

The **Stun** function will be called when we get hit by an obstacle or enter the wrong problem tube. First, let's set our state to Stunned.

```
curState = PlayerState.Stunned;
```

Then we want to throw the player down to the ground.

```
rig.velocity = Vector2.down * 3;
```

And finally, set the stun start time to be now and stop the jetpack particle.

```
stunStartTime = Time.time;
jetpackParticle.Stop();
```

IsGrounded Function

IsGrounded returns either true or false. This is depending on whether or not the player is standing on the ground. To begin, let's shoot a raycast downwards.

```
// shoot a raycast down underneath the player
RaycasstHit2D hit = Physics2D.Raycast(new Vector2(transform.position.x, transform.pos
ition.y - 0.55f), Vector2.down, 0.3f);
```

Let's check if the collider hit anything – if so was it an object tagged as **Floor**? If so, return true – we did hit the floor.

```
// did we hit anything
if(hit.collider != null)
{
    // was it the floor?
    if(hit.collider.CompareTag("Floor"))
    {
        return true;
    }
}
```

At the end of the function, if we hit nothing, return false.

```
return false;
```

OnTriggerEnter2D Function

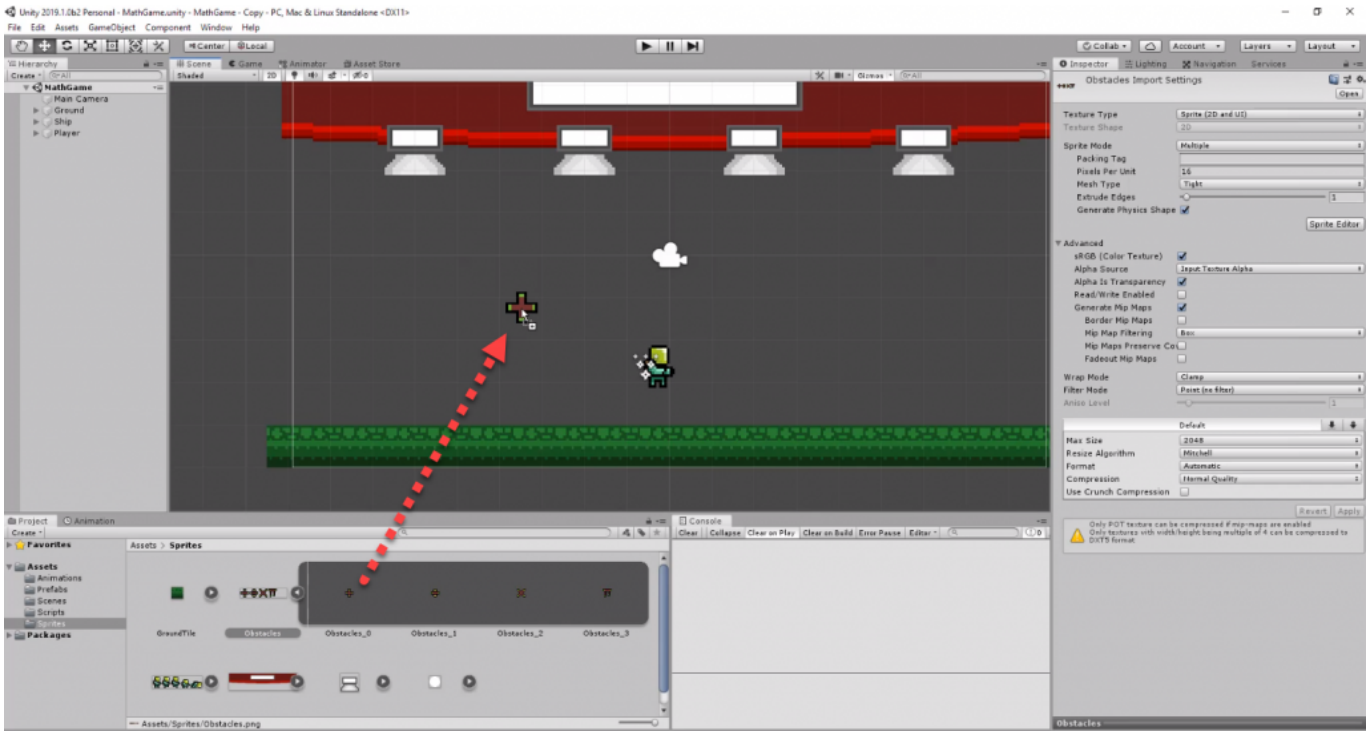
In the **OnTriggerEnter2D** function, let's check if we hit an obstacle. If so, get stunned.

```
if(curState != PlayerState.Stunned)
{
    // did we trigger an obstacle?
    if(collision.gameObject.CompareTag("Obstacle"))
    {
        Stun();
    }
}
```

Back in the editor, we can play the game now and see that our player controller is complete!

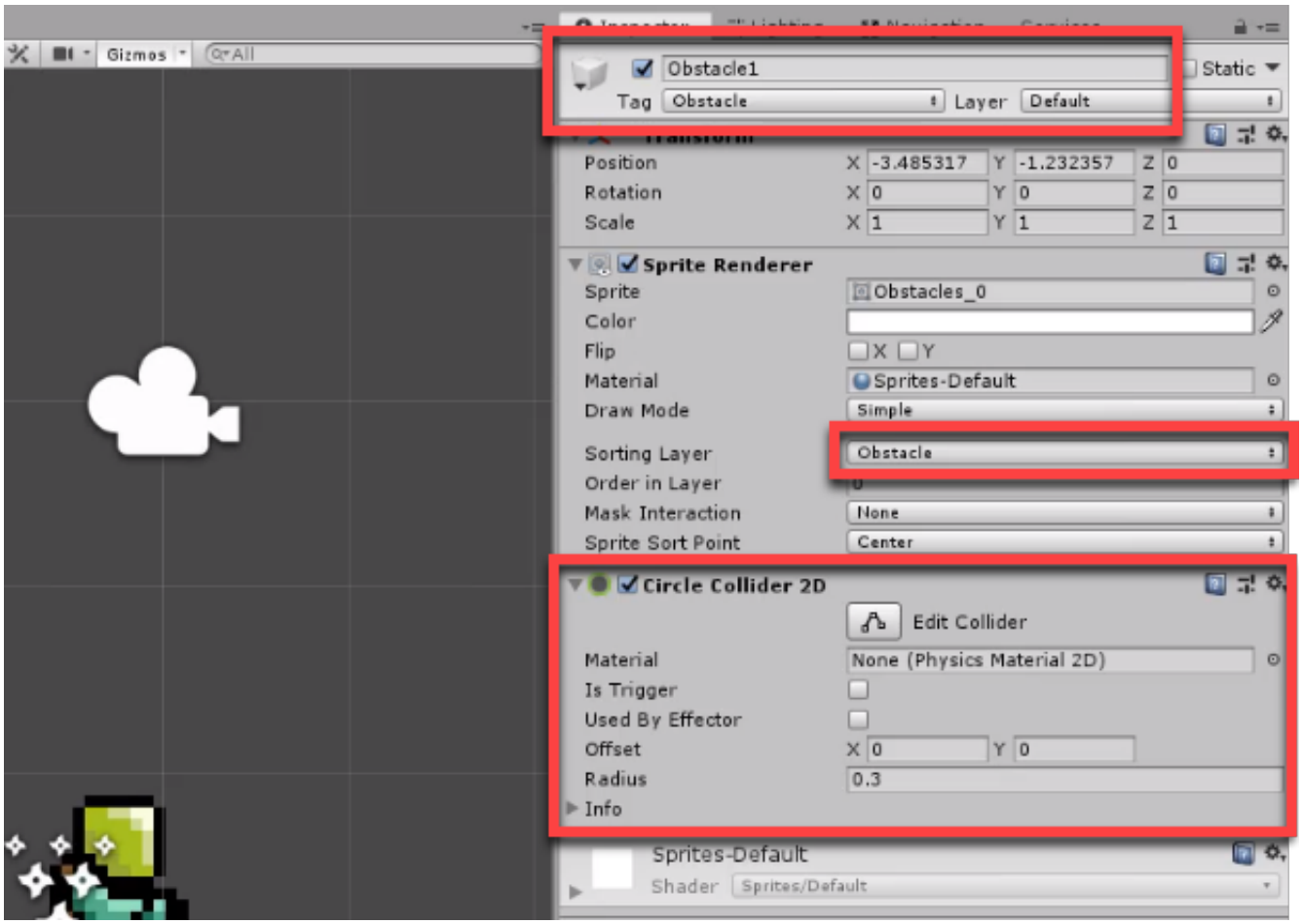
Creating an Obstacle

In this lesson, we'll be creating the obstacles. First, let's drag the **Obstacles_0** sprite into the scene (> *Sprites*).



To set the object up, let's change a few properties.

- Rename the object to **Obstacle1**
- Set the **Tag** to **Obstacle**
- Set the **Sorting Layer** to **Obstacle**
- Add a **CircleCollider2D** component with a radius of *0.3*
    - Enable **Is Trigger** (not seen in image)



Scripting the Obstacle

Let's now start on scripting the obstacle. Create a new C# script (right click **Project** > *Create* > *C# Script*) and call it **Obstacle**.

Our variables are just going to hold the direction to move in, the speed and how long until the object is alive for.

```
// direction to move in
public Vector3 moveDir;

// speed of the obstacle
public float moveSpeed;

// rotation speed
public float rotateSpeed;

// time before destroying object
public float aliveTime = 4.0f;
```

What we're going to do is invoke the **Deactivate** function in the **OnEnable** function. Invoke is basically a function call but with a delay. Here, we're delaying it by the alive time. Then in the **Deactivate** function, we're just going to disable the object. This is because we're going to *pool* these objects later on.

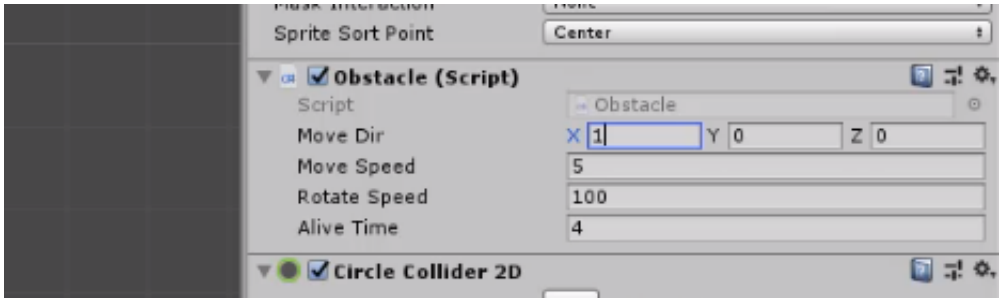

```
void OnEnable ()
{
    CancelInvoke("Deactivate");
    Invoke("Deactivate", aliveTime);
}

void Deactivate ()
{
    gameObject.SetActive(false);
}
```

Now in the **Update** function, let's move the object in the move direction and rotate it over time.

```
void Update ()
{
    // move the obstacle in the direction over time
    transform.position += moveDir * moveSpeed * Time.deltaTime;

    // rotate the obstacle overtime
    transform.Rotate(Vector3.back * moveDir.x * rotateSpeed * Time.deltaTime);
}
```

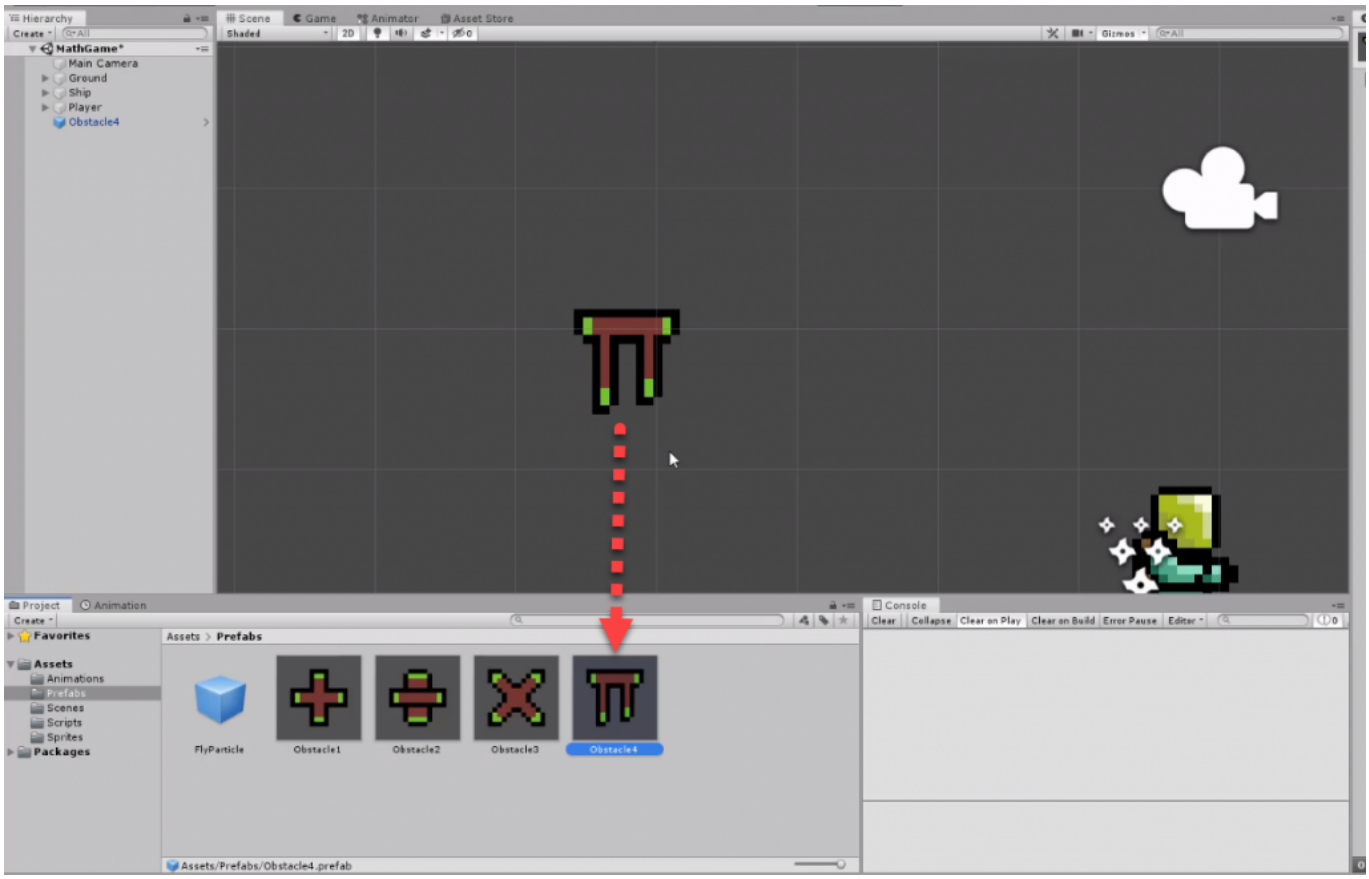
Back in the editor, let's add the script to the obstacle object. For the values...

- Set **Move Dir** to *1, 0, 0* (moving right for testing)
- Set **Move Speed** to *5*
- Set **Rotate Speed** to *100*
- Set **Alive Time** to *4*



If you press play, you should see the obstacle moving and rotating across the screen.

What we want to do now is save the object to the *Prefabs* folder (drag it into the **Prefabs** folder). Then change the sprite of the object in the scene to the next sprite, change the name, drag that into the *Prefabs* folder. Do this for all 4 obstacle sprites. We're doing this so we can spawn these in later. Make sure to set the **Move Dir** on each object to *0, 0, 0*.



In the next lesson, we'll be creating an object pooling system – allowing us to optimize object spawning.

What's an Object Pool?

An object pool is basically a collection of objects we instantiate at the start of the game and enable/disable them when we need them. The reason why we don't just call *Instantiate* to spawn an object is that it's quite expensive. Since we're creating the same set of 4 objects across the whole duration of the game, pooling them at the start will increase our performance. Although since this game is quite lightweight, it won't be that noticeable, although in larger scale game, pooling can be an important factor.

Scripting the Object Pool

For our variables, there are 3. First, we have **objectPrefabs**. This is an array of game objects with all the types of objects to spawn.

```
// array of all obstacle types
public GameObject[] objectPrefabs;
```

Next, we have the **poolSize**. The amount of objects we'll create at the start of the game.

```
// number of objects to have in pool
public int poolSize = 20;
```

pooledObjects is a list holding all of the spawned objects. These will be disabled, which we will then enable when we need to access one.

```
// list of all objects in the pool
private List<GameObject> pooledObjects = new List<GameObject>();
```

Creating the Pool

Let's start with creating our pooled objects. In the **Start** function, we'll call the **CreatePool** function.

```
void Start ()
{
    // create the pool's objects
    CreatePool();
}
```

Then in the **CreatePool** function, let's start by creating a loop that has **poolSize** iterations.

```
// instantiates all the pool objects
void CreatePool ()
{
    for(int i = 0; i < poolSize; i++)
    {

    }
```



```
}
```

Inside of the loop, let's Instantiate an object prefab. Which one? Well we're going to be creating them in order. e.g. the prefabs spawned will be: 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, etc... This is calculated easily by using a modulo operation (%). What this is, is basically the remainder of a division.

```
// instantiate the object
GameObject objectToSpawn = Instantiate(objectPrefabs[i % objectPrefabs.Length]);
```

Next, let's deactivate the object and add it to the **pooledObjects** list.

```
// deactivate it
objectToSpawn.SetActive(false);

// add to the pooled objects list
pooledObjects.Add(objectToSpawn);
```

Getting a Pooled Object

The function we're going call to get and "spawn" an object, we're going to use the **GetPooledObject** function. This will return a game object.

```
// returns a pooled object, ready for use
public GameObject GetPooledObject ()
{

}
```

First, let's get an inactive item from the **pooledObjects** list.

```
GameObject objectToSend = pooledObjects.Find(x => !x.activeInHierarchy);
```

Then let's check if we found an object. If there was no inactive object in the list, **objectToSend** will be null. If so, log an error saying the pool is too small. Otherwise, we'll activate the object. Finally, return the object.

```
// is there no object to send?
if(!objectToSend)
    Debug.Log("No more available objects in the pool! Increase Its size.");
else
    objectToSend.SetActive(true);

// return the object
return objectToSend;
```


In the next lesson, we'll be creating the script which will spawn and position the obstacles on either side of the screen.

Scripting the Obstacle Spawner

In this lesson, we'll be creating the obstacle spawner script. To begin, let's create a new C# script and call it **ObstacleSpawner**.

Our first variable is our object pool script which will spawn and contain our obstacles.

```
public ObjectPool obstaclePool;
```

Then we have the range values. These are the limits the where we can spawn objects.

```
// lowest point an object can spawn
public float minSpawnY;

// max height an object can spawn
public float maxSpawnY;

// left hand side X spawn pos
public float leftSpawnX;

// right hand side X spawn pos
public float rightSpawnX;
```

Lastly, we have the rate at which the obstacles are spawned.

```
// rate at which objects are spawned
public float spawnRate;
private float lastSpawnTime;
```

Setting Spawn Borders

leftSpawnX and **rightSpawnX** are values which are calculated in the script. This is because they're based on the left and right view borders of the camera.

```
void Start ()
{
    Camera cam = Camera.main;
    float camWidth = (2.0f * cam.orthographicSize) * cam.aspect;

    leftSpawnX = -camWidth / 2;
    rightSpawnX = camWidth / 2;
}
```

Spawning the Obstacles

In the **Update** function, we'll be calling the **SpawnObstacle** function every **spawnRate** seconds.


```
void Update ()
{
    if(Time.time - spawnRate >= lastSpawnTime)
    {
        lastSpawnTime = Time.time;
        SpawnObstacle();
    }
}
```

Let's create the **SpawnObstacle** function.

```
// creates a new obstacle
void SpawnObstacle ()
{

}
```

What we want to do first, is access an inactive object from the pool.

```
// get the obstacle
GameObject obstacle = obstaclePool.GetPooledObject();
```

Set the position using the **GetSpawnPosition** function (we'll create this next).

```
// set its position
obstacle.transform.position = GetSpawnPosition();
```

Then we want to access the obstacle's **Obstacle** script and set the move direction based on the spawn position.

```
// set the obstacle's direction to move
obstacle.GetComponent<Obstacle>().moveDir = new Vector3(obstacle.transform.position.x
 > 0 ? -1 " 1, 0, 0);
```

In the next lesson, we'll be finishing up on the obstacle spawner.

Getting a Spawn Position

Let's create the **GetSpawnPosition** function. This will return a position for the object to appear in.

```
// return a random position to spawn in
Vector3 GetSpawnPosition ()
{

}
```

We first want to get an X position. We're first generating a random number between 0 and 2 (2 is exclusive so the possible numbers will be 0 or 1). If it's 1, it will spawn on the left, or 0 will spawn on the right.

```
float x = Random.Range(0, 2) == 1 ? leftSpawnX : rightSpawnX;
```

Then we want to get a Y position. This will be a random number between the min and max spawn Y.

```
float y = Random.Range(minSpawnY, maxSpawnY);
```

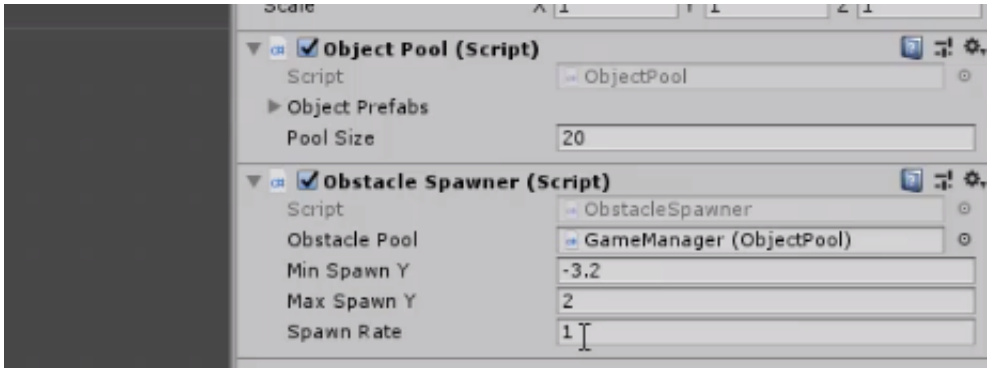
Finally, let's return the position.

```
return new Vector3(x, y, 0);
```

Back in the Editor

Back in the editor, let's make sure we have our **ObjectPool** and **ObstacleSpawner** scripts attached to the **GameManager** object (create an empty game object and call it GameManager).

- Drag the 4 obstacle prefabs into the **Object Prefabs** array
- Set the **Pool Size** to *20*
- Connect the **ObstaclePool** script to the **ObstacleSpawner**
- Set **Min Spawn Y** to *-3.2*
- Set **Max Spawn Y** to *2*
- Set **Spawn Rate** to *1*



In the next lesson, we'll start on creating the math problems.

Creating the Problem Class

To hold data for each problem, we're going to create a script called **Problem**. This will hold information such as the numbers, possible answers, correct tube, etc.

Let's start with the **MathsOperation** enumerator. This is basically a list of all the possible math operations we can use. For this project, we're going to focus on addition, subtraction, multiplication and division. You can add more though if you wish. Make sure that this is located outside and on top of the class as we need other scripts to be able to access it.

```
public enum MathsOperation
{
    Addition,
    Subtraction,
    Multipication,
    Division
}
```

Then inside of the **Problem** class, we can start with our two float variables. These are the first and second numbers of the math problem. For example: 15 + 33. 15 would be the first number and 33 would be the second number.

```
public float firstNumber;
public float secondNumber;
```

Next, we have the MathsOperation. Is this an addition question? division? etc.

```
public MathsOperation operation;
```

Then we want to have an array of all the possible answers. Since we have 4 problem tubes, this will be an array of 4 elements (we'll enter these in over in the editor).

```
public float[] answers;
```

Finally, we got our correct tube value. This will be the index number of a value in the **answers** array. Just so we know which answer was the right one.

```
public int correctTube;
```

At the top of the class (just the line above where the class is defined) we need to set this class as serializable. For us, this'll mean that we can open up and fill in the properties inside of the inspector.

```
[System.Serializable]
```


And that’s it for the **Problem** class. In the next lesson, we’ll start on the game manager, which will run the core game loop.

Game Manager Script

In this lesson, we're going to start on creating our game manager script. Create a new C# script in the *Scripts* folder, called **GameManager**. Let's start with our variables.

We first have our **problem** function. This is basically going to hold all of our problems in order.

```
// list of all the problems
public Problem[] problems;
```

Then we need an integer for our **curProblem**, so we know which problem we're currently on.

```
// current problem
public int curProblem;
```

Next, we want to know how many seconds the player will have for each problem, as well as the current time on the current problem.

```
// max time for each problem
public float timePerProblem;

// time remaining for the problem
public float remainingTime;
```

We also need the player object so we can stun them.

```
// player object
public PlayerObject player;
```

Finally, we have the instance of the script, allowing others to easily access it anywhere.

```
// instance
public static GameManager instance;

void Awake ()
{
    // set the instance
    instance = this;
}
```

Setting a Problem

In the **Start** function, let's set the initial function.

```
void Start ()
```

```
{
    // set the initial problem
    SetProblem(0);
}
```

The **SetProblem** function is called when we want to set a new problem for the player to solve. It takes in an integer parameter which is an index for the **problem** array. This will tell the UI to update and reset the remaining time.

```
// sets the current problem
void SetProblem (int problem)
{
    curProblem = problem;
    // set the ui text
    remainingTime = timePerProblem;
}
```

Losing

In the **Update** function, let's count down the **remainingTime**. If it reaches 0 – game over.

```
void Update ()
{
    remainingTime -= Time.deltaTime;

    // has the remaining time ran out?
    if(remainingTime <= 0)
    {
        Lose();
    }
}
```

The **Lose** function will be called when the remaining time reaches 0. This will pause the game and enable a UI lose screen.

```
void Lose ()
{
    Time.timeScale = 0.0f;
    // set the lose screen
}
```

In the next lesson, we're going to finish off the script.

Winning

Like when we lose, we also want to make a function for when we win. The **Win** function will be called once all of the problems have been answered correctly.

```
// called when the player answers all the questions
void Win ()
{
    Time.timeScale = 0.0f;
    // set a UI win screen
}
```

When the Player Enters the Tube

This script is also going to manage what happens when the player enters a tube. When a player does enter a tube's trigger, the **OnPlayerEnterTube** function will be called – sending over an index number for the tube (0 – 3).

```
// called when the player enters a tube
public void OnPlayerEnterTube (int tube)
{

}
```

The first thing we check, is if the tube the player entered was the right one. If so, call the **CorrectAnswer** function, otherwise call the **IncorrectAnswer** function.

```
// did the player enter the right tube?
if(tube == problems[curProblem].correctTube)
{
    CorrectAnswer();
}
else
{
    IncorrectAnswer();
}
```

The **CorrectAnswer** function either moves the player onto the next question, or wins the game.

```
void CorrectAnswer ()
{
    // is this the last problem?
    if(problem.Length - 1 == curProblem)
    {
        Win();
    }
    else
    {
        SetProblem(curProblem + 1);
```



```
    }
}
```

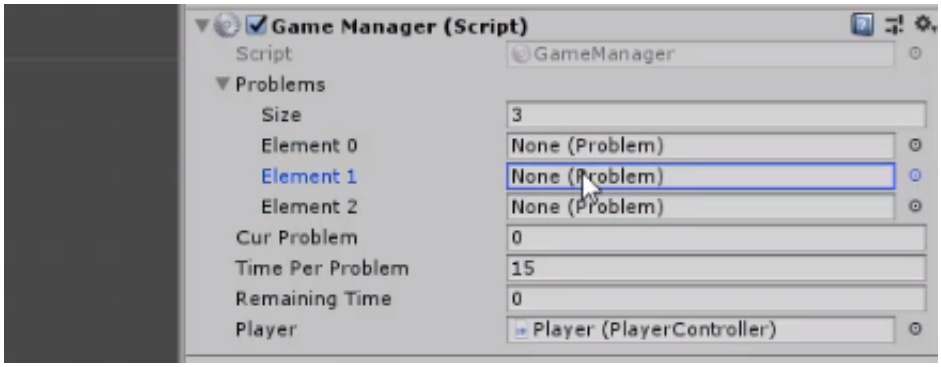
Then the **IncorrectAnswer** function stuns the player.

```
void IncorrectAnswer ()
{
    player.Stun();
}
```

Back in the Editor

Back in the editor, let's attach the script to the **GameManager** object.

- Set the **Problems** size to *3* (we'll talk about this soon)
- Set the **Time Per Problem** to *15*
- Attach the **PlayerController** script

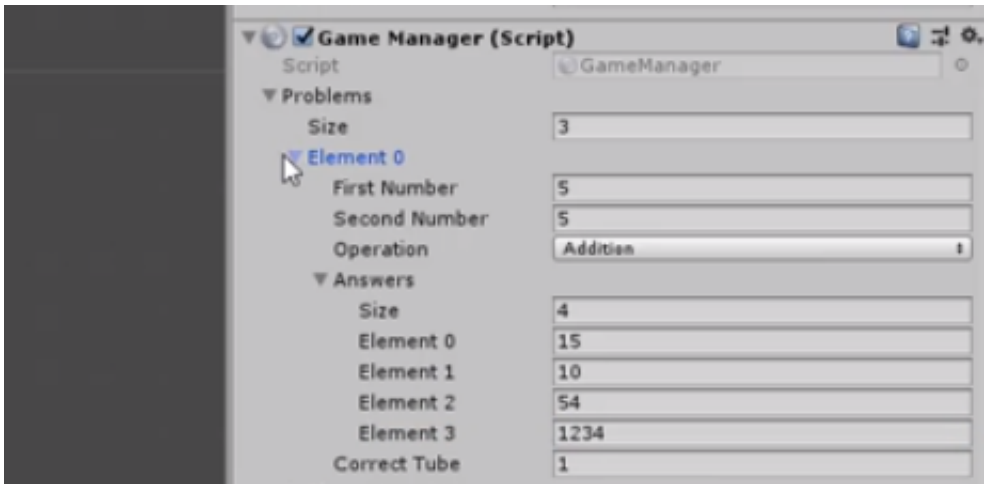


You may notice that we can't enter in any problems, even though we added in the **[System.Serializable]** property to the class. Something we need to do, is go to the **Problem** class and in the class definition – remove the **: MonoBehaviour**.

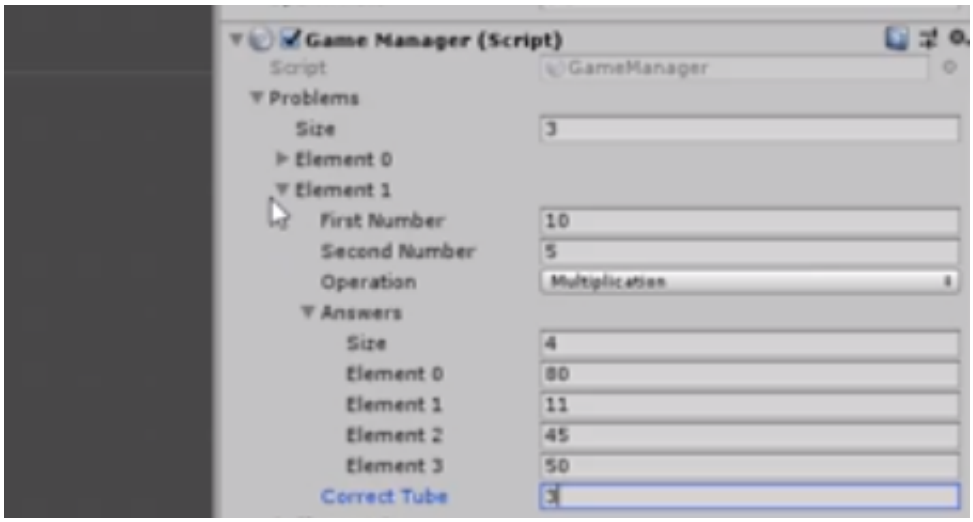
```
[System.Serializable]
public class Problem
{
    ...
```

Our first question is going to be as follows: **5 + 5**

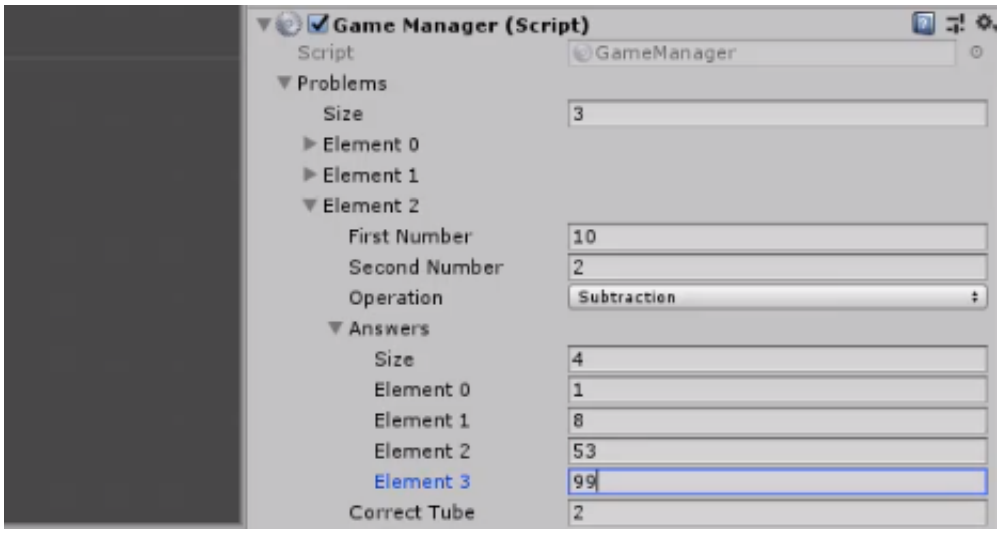
- Set the **First Number** to *5*
- Set the **Second Number** to *5*
- Set the **Operation** to **Addition**
- Set the **Answers** to *15, 10, 54, 1234*
- Set the **Correct Tube** to *1*



The second question is going to a multiplication one.



Finally, the third question is going to be subtraction.

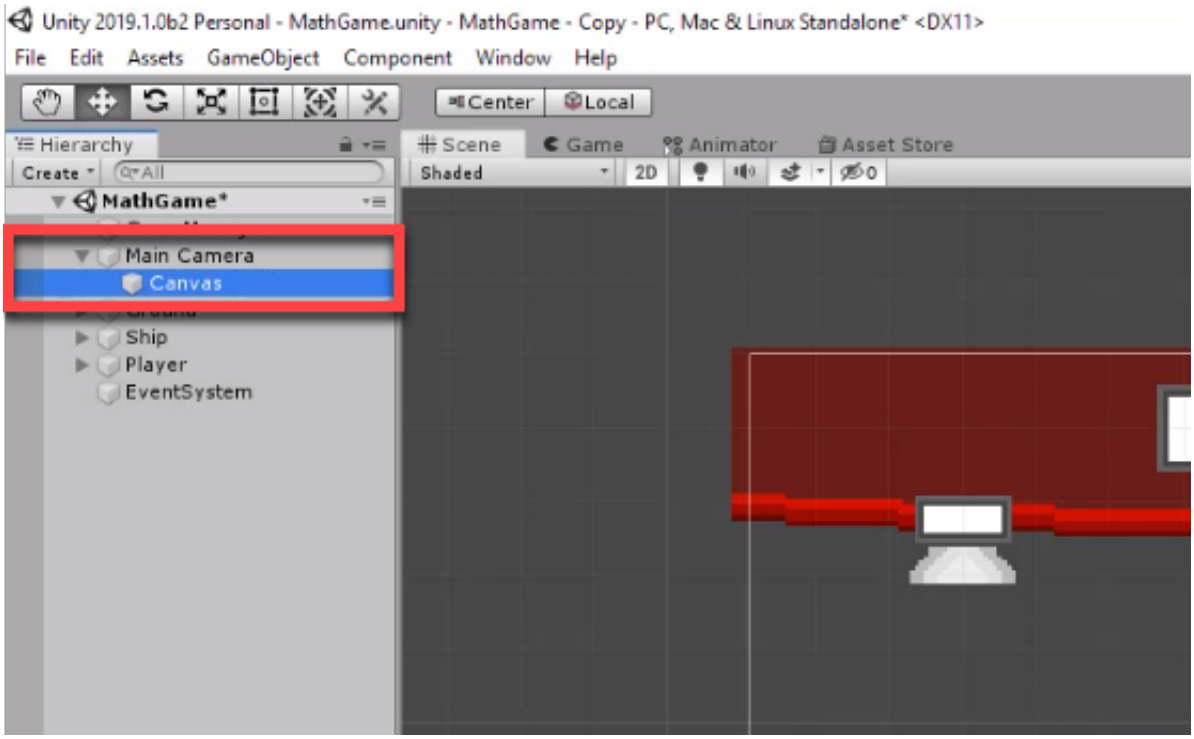


You can add as many questions as you like and they are fully up to you of what they are.

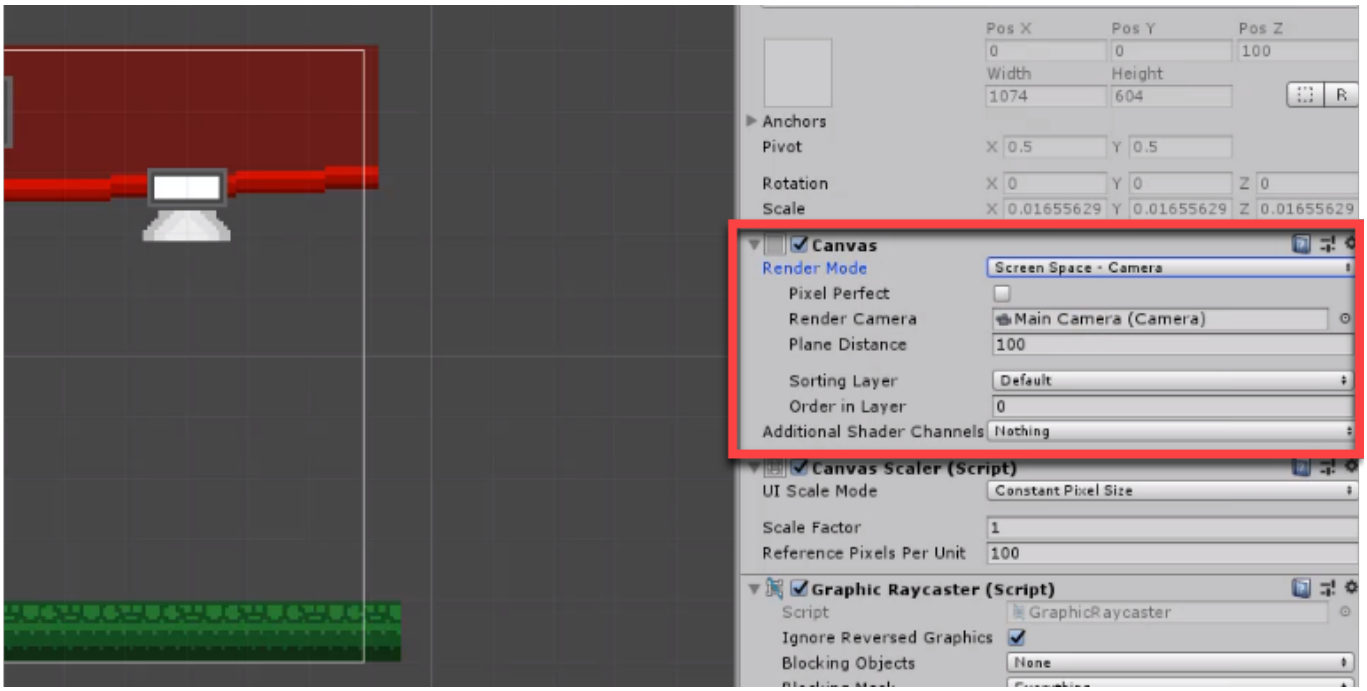
In the next lesson, we'll be starting on setting up the UI.

Setting Up the Canvas

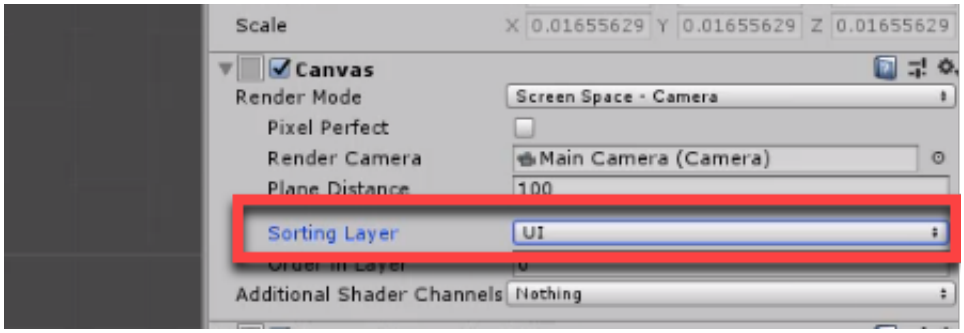
In this lesson, we'll be creating the UI elements. To begin, create a new canvas as a child of the camera (right click **Main Camera** > *UI* > *Canvas*). A canvas is basically a container for our UI elements.



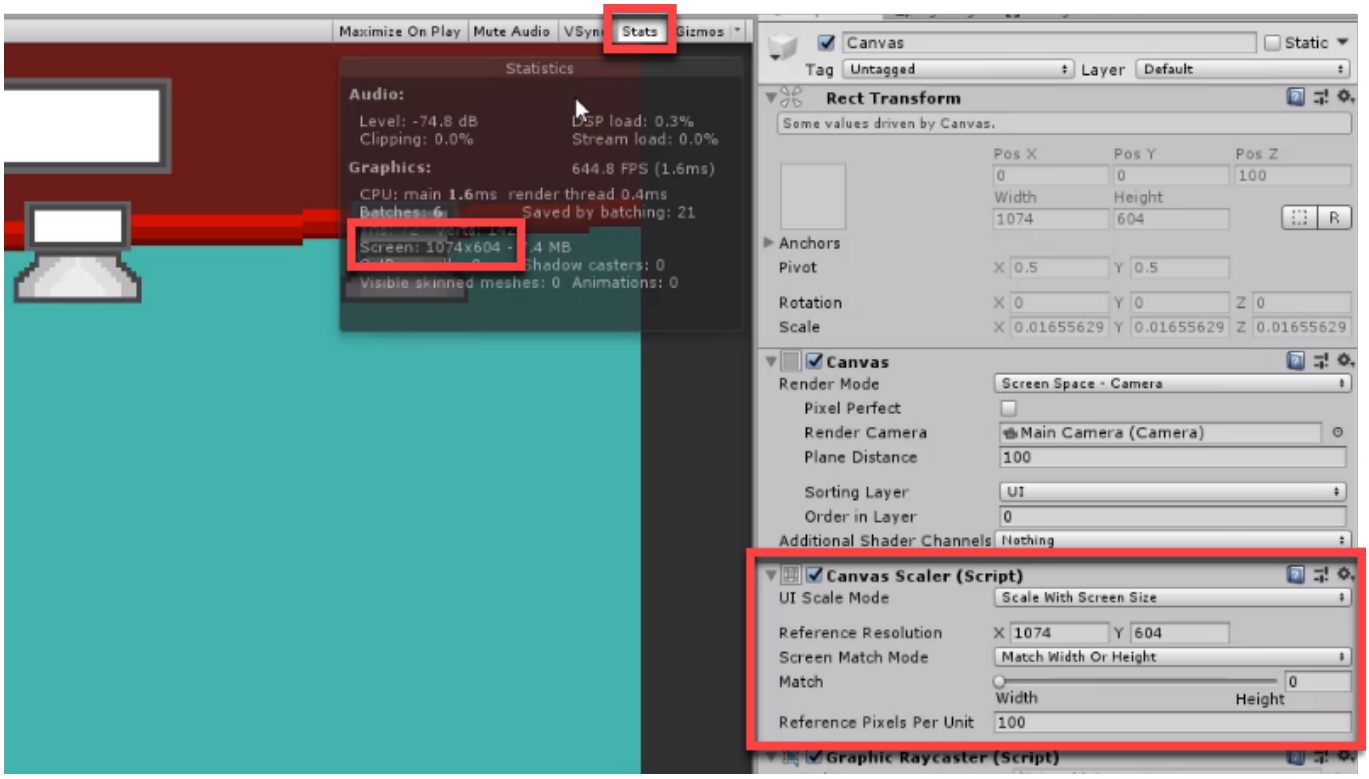
Since we're working in 2D we can use the camera screen space mode. This basically means that instead of having our canvas very large in the scene, it's mapped directly to the camera – making it easier to work with. To do this, select the canvas and in the **Inspector**, set **Render Mode** to Screen Space – Camera. Then drag in the main camera to the **Render Camera** property.



While we're here, let's also set the **Sorting Layer** to UI, so the UI elements will appear on top of everything else.

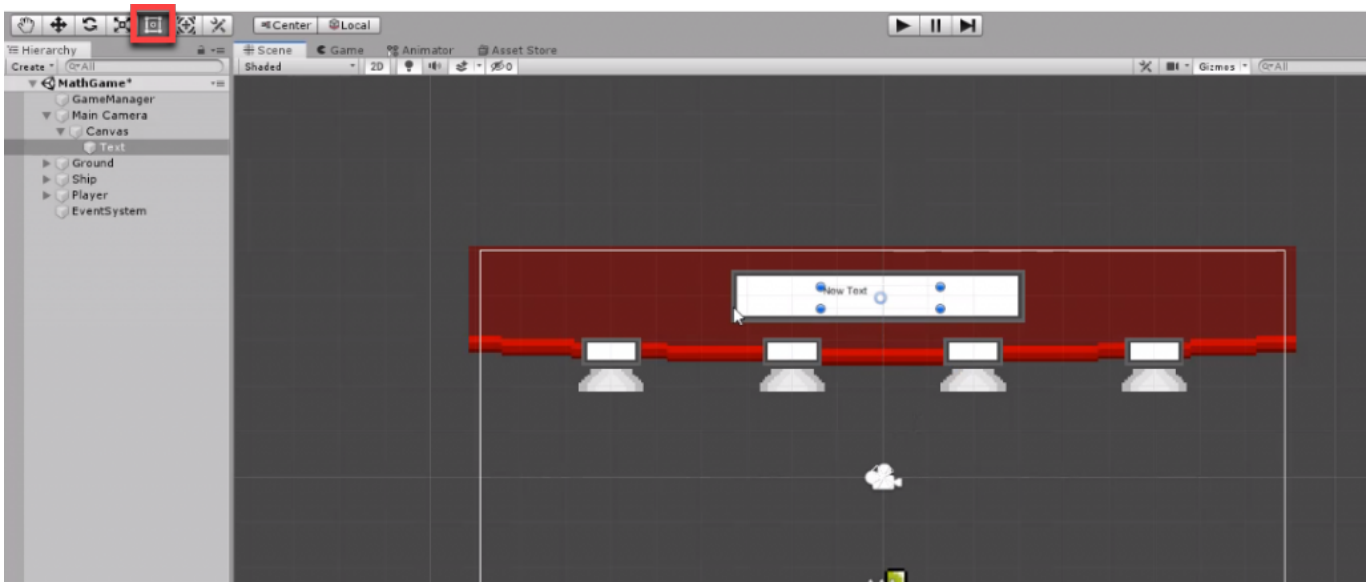


With the default UI, all elements are a set pixel size. Since our UI is going to be stuck to specific parts of the screen, we'll need to make it so the UI doesn't move when the screen resolution is changed. Selecting the canvas, set the **Reference Resolution** to the game view's resolution. You can find this info our by toggling the **Stats** window.



Problem Text

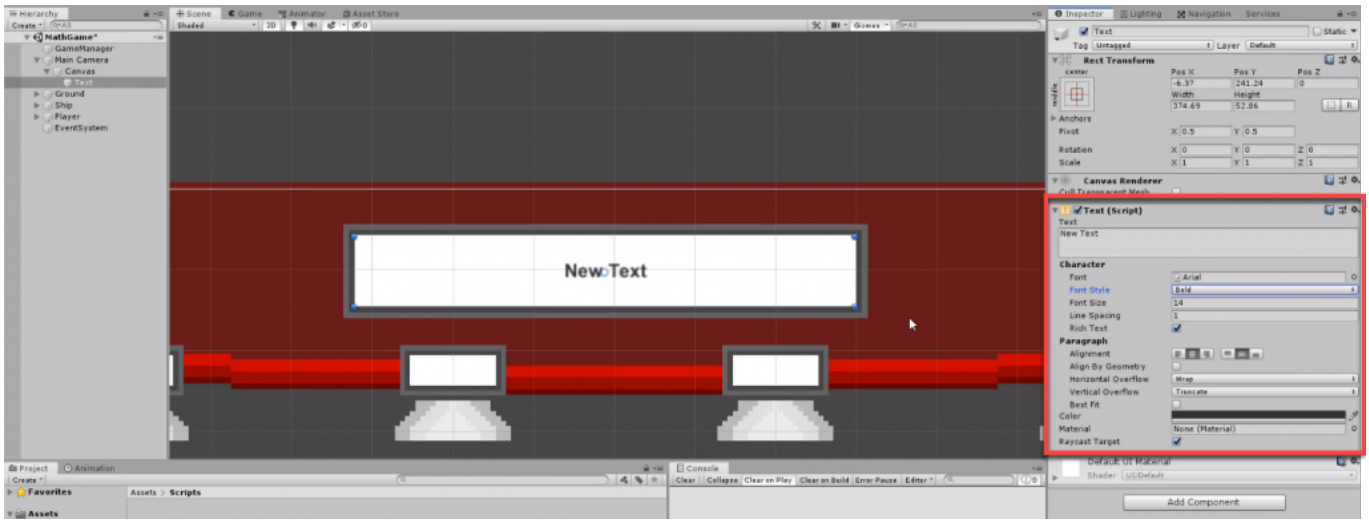
Next, let's create a new text element (right click on **Canvas** > *UI* > *Text*). This is going to be our problem text – displaying to the player the current problem.



We can toggle the **Rect Transform Gizmo** to modify UI elements (shortcut = '**T**'). The blue circles can be dragged to resize the rect transform. What we want to do is position it at the top of the screen and resize it to the bounds of the white rectangle. In the **Inspector**, let's modify the text a bit:

- Set **Font Style** to **Bold**
- Set **Font Size** to *30* (not seen in image)
- Set **Alignment** to middle and center

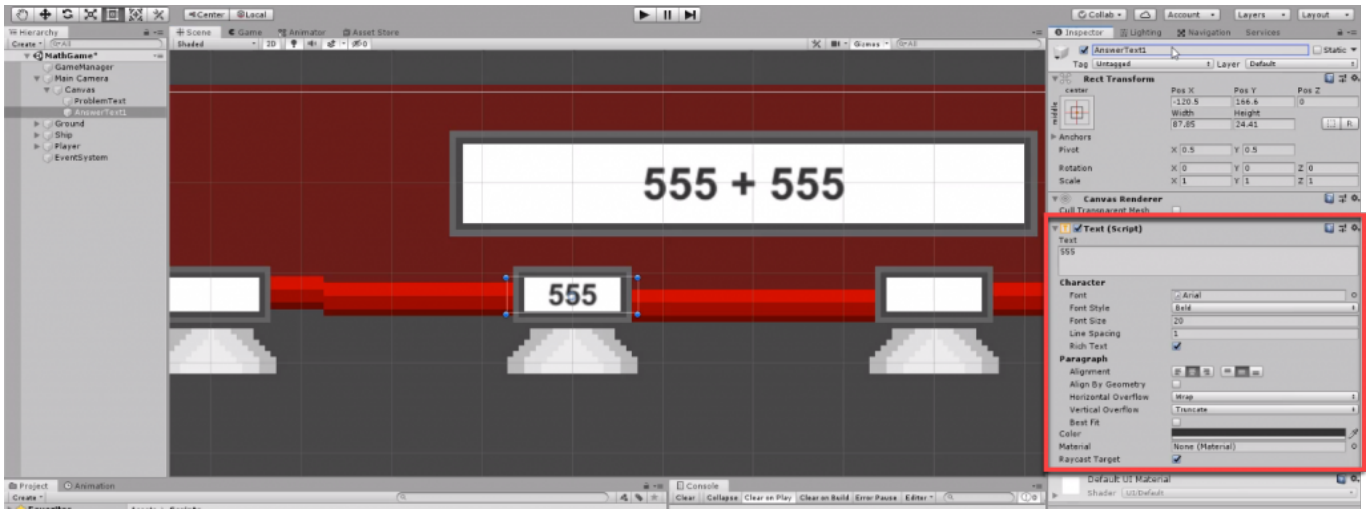
Let's also rename the text object to **ProblemText**.



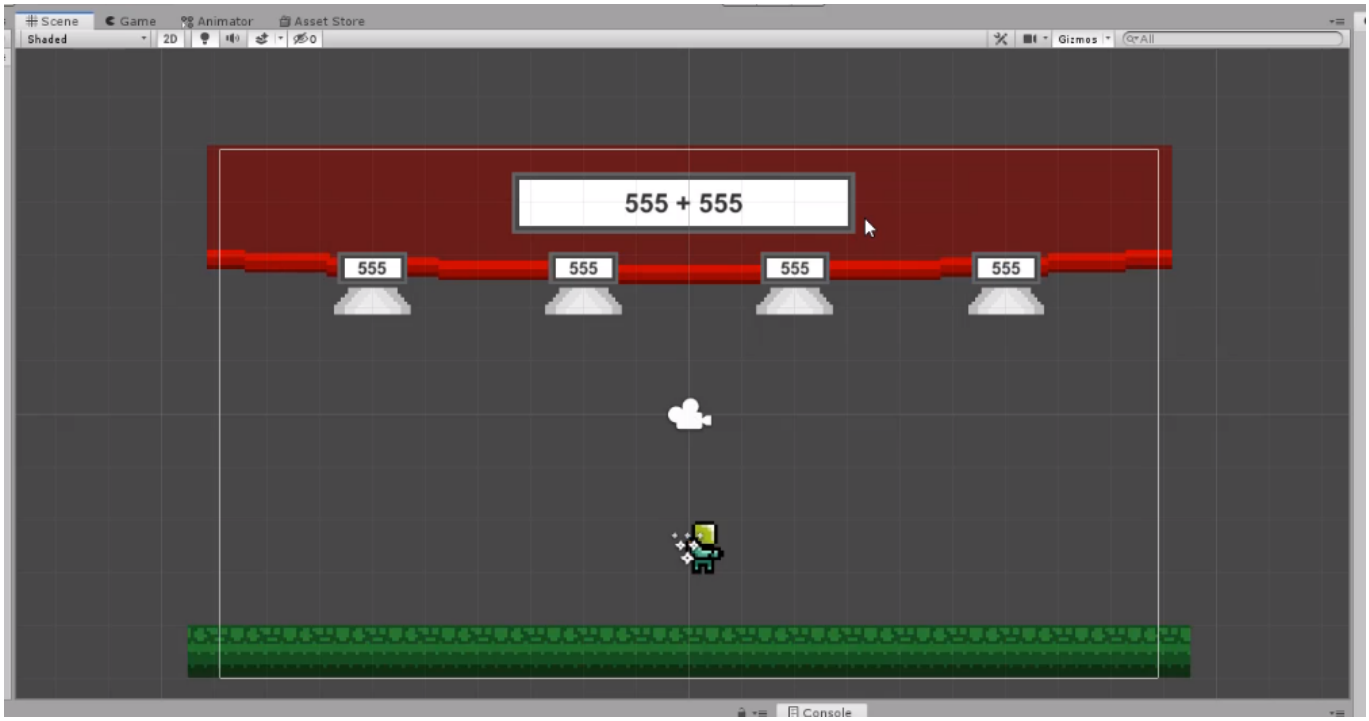
Answer Text

Now let's make the text elements for the answers. Duplicate the problem text (shortcut = '**Ctrl**' + '**D**') and call it **AnswerText1**.

- Set the **Font Size** to *20*
- Resize / reposition it to be over the white rectangle of the problem tube



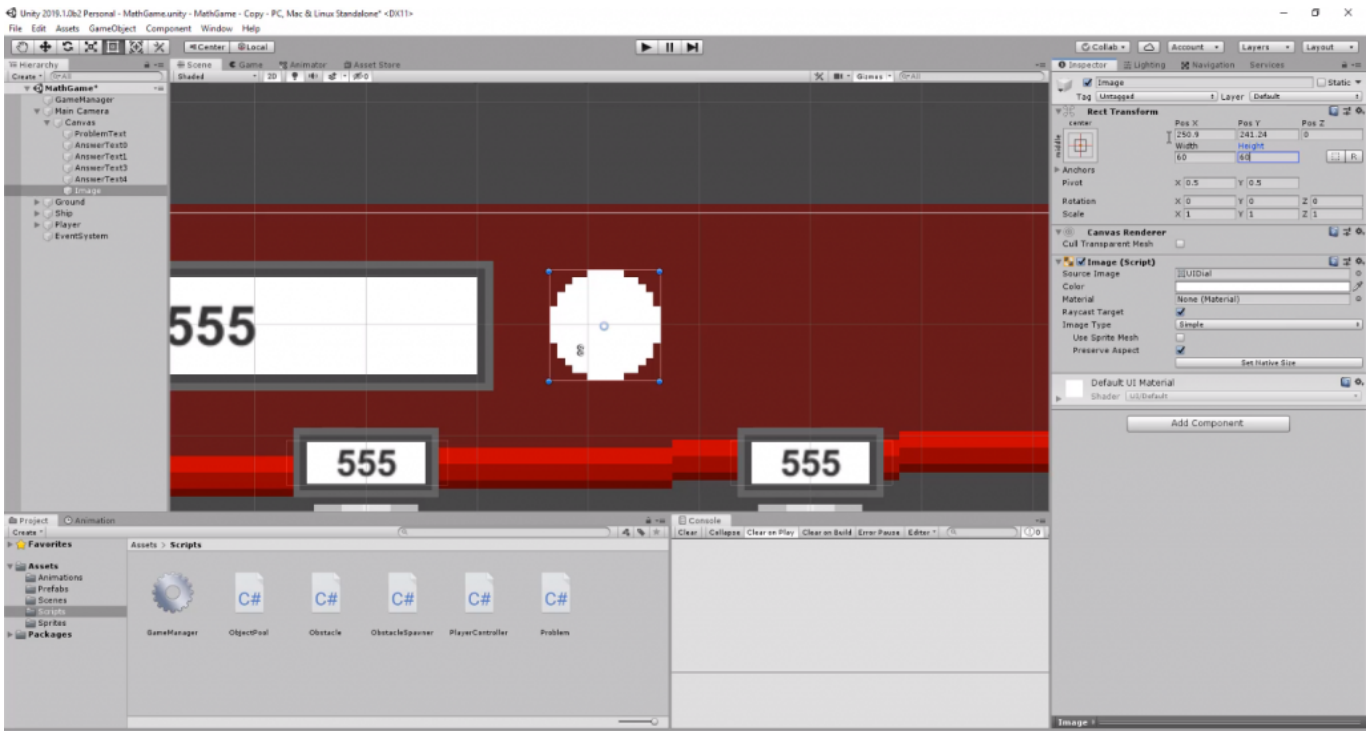
Duplicate this text for each of the problem tubes.



Time Dial

For the time dial, let's create a new image element (right click **Canvas** > *UI* > *Image*) and call it **TimeDial**.

- Set the **Width** and **Height** to *60*
- Position it to the right of the problem text



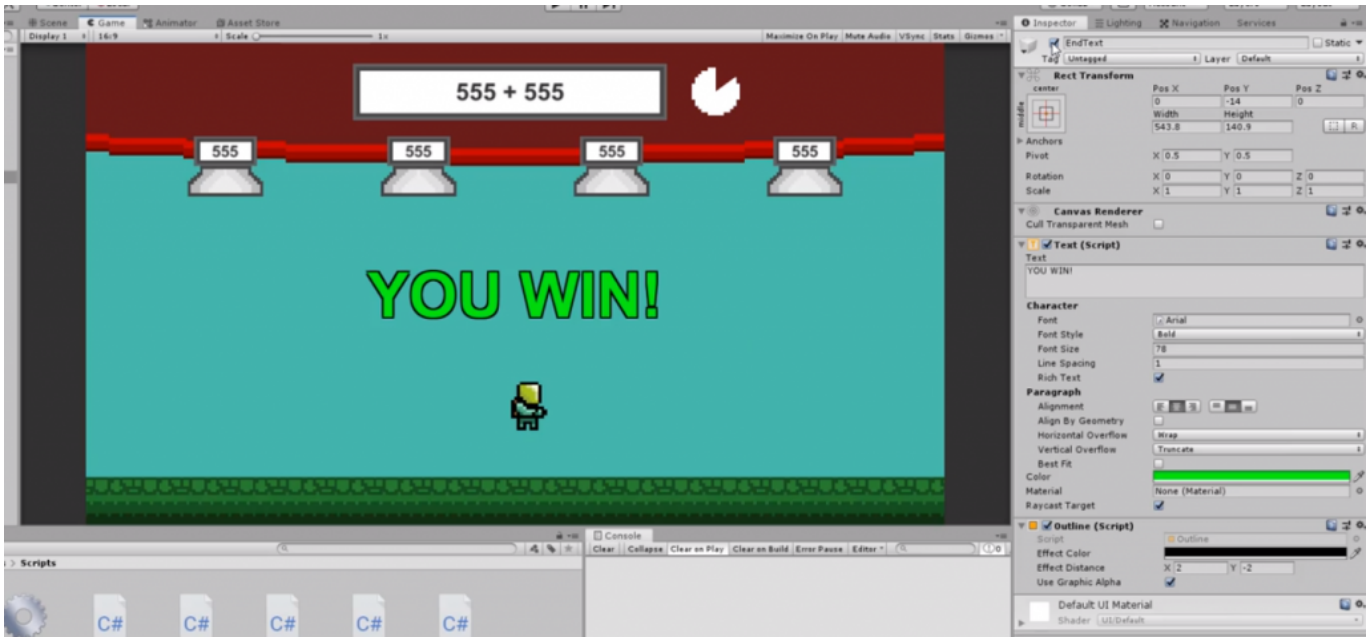
Let's now make it so the image can be "reduced" or tick down like a clock. To do this, set the image's **Image Type** to **Filled**.

- Set the **Fill Origin** to Top (not in image)
- Untick **Clockwise** (not in image)

End Screen

All we need to do now is create an end screen. For us, this is just going to be a large text element called **EndText**.

- Set the **Font Size** to *78*
- Add an **Outline** component
    - Set the **Effect Distance** to *2, -2*



In the next lesson, we'll be connecting the UI to a script.

UI Script

Create a new C# script called **UI**. Since we're using Unity's UI elements, we'll have to reference the UI namespace.

```
using UnityEngine.UI;
```

First, we have our text elements. The problem text and array of answer texts.

```
public Text problemText;
public Text[] answerText;
```

Then we have our time dial and time dial rate (i'll explain about this soon).

```
public Image timeDial;
private float timeDialRate;
```

Next, we have our end screen text.

```
public Text endText;
```

Finally, the instance of the script, so we can access it easily anywhere.

```
// instance
public static UI instance;

void Awake ()
{
    // set the instance
    instance = this;
}
```

Time Dial

In the **Start** function, let's set our **timeDialRate** value. This is basically a modifier we're going to apply to the current time on the problem when deciding what value to set the time dial's fill amount. The image's fill amount ranges in value from 0.0 to 1.0 – yet our remaining time can be 15 or anything. To normalize this value, we have it divided by 1. This means when we set the fill amount, we can easily decide what the value will be.

E.g. we have a time per problem of 15. That means when the current time is 6, the time dial will have a fill amount of 0.399.

```
void Start ()
{
```

```
    // calculate the time dial rate
    timeDialRate = 1.0f / GameManager.instance.timePerProblem;
}
```

Then in the **Update** function, we'll be setting the time dial's fill amount.

```
void Update ()
{
    // update the time dial
    timeDial.fillAmount = timeDialRate * GameManager.instance.remainingTime;
}
```

Setting the Problem Text

Let's now create the **SetProblemText** function, which will display the problem sent as a parameter.

```
// sets the ship UI to display the new problem
public void SetProblemText (Problem problem)
{

}
```

The first thing we need to do, is convert our operator enum to a string.

```
string operatorText = "";

// get the operator and convert it to a string
switch(problem.operation)
{
    case MathsOperation.Addition:
        operatorText = " + ";
        break;
    case MathsOperation.Subtraction;
        operatorText = " - ";
        break;
    case MathsOperation.Multiplication:
        operatorText = " x ";
        break;
    case MathsOperation.Division:
        operatorText = " ÷ ";
        break;
}
```

In the next lesson, we'll be finishing off the script.

Continuing the SetProblemText Function

Let's set the text of the problem text to be the first number, operator then second number.

```
// set the problem text to display the equation
problemText.text = problem.firstNumber + operatorText + problem.secondNumber;
```

Now let's loop through all the answers and set them to their respective answer texts.

```
// set the answers texts to display the correct and incorrect answers
for(int i = 0; i < answerTexts.Length; i++)
{
    answerTexts[i].text = problem.answers[i].ToString();
}
```

End Screen

The function **SetEndText** will be called if the player wins or loses the game. This sends over a bool as a parameter, dictating whether the player won or not.

```
// sets the win or lose text
public void SetEndText (bool win)
{

}
```

First, we enable the end text, then set the text and color depending on the sent boolean.

```
// enable the text
endText.gameObject.SetActive(true);

// did the player win?
if(win)
{
    endText.text = "You Win!";
    endText.color = Color.green;
}
// did the player lose?
else
{
    endText.text = "Game Over!";
    endText.color = Color.red;
}
```

Connecting to the GameManager Script

Now let's connect the UI to the **GameManager** script.

In the **SetProblem** function, let's call the UI's **SetProblemText** function.

```
UI.instance.SetProblemText(problems[curProblem]);
```

In the **Win** function, let's call the UI's **SetEndText** function.

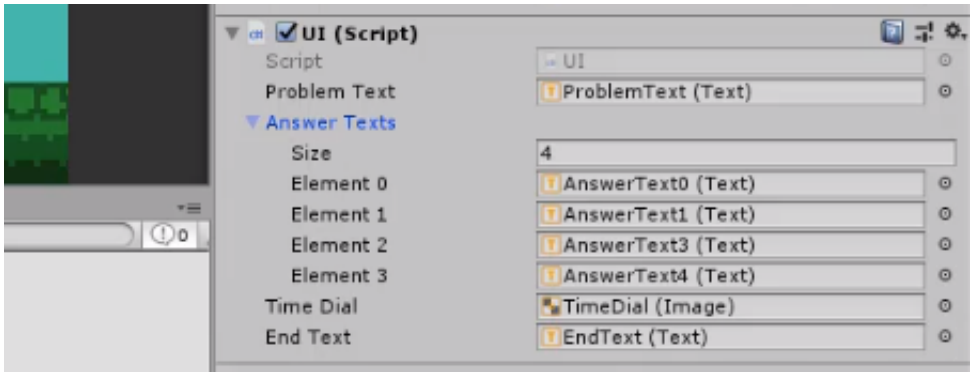
```
UI.instance.SetEndText(true);
```

In the **Lose** function, let's do the same.

```
UI.instance.SetEndText(false);
```

Back in the Editor

Back in the editor, let's attach the **UI** script to the **GameManager** object and fill in the properties.



If you press play, you can see that the first problem appears on the screen. In the next lesson, we'll be finishing up on the game, connecting the problem tubes.

Problem Tube Script

Create a new C# script called **ProblemTube**. This script will basically check for the player to enter its trigger, then tell the game manager.

The only variable we'll need to hold is the **tubeId**. It refers to the answers array (0 – 3).

```
public int tubeId;
```

Then, the only function we are going to use is **OnTriggerEnter2D**. This gets called when the object's collider enters the collider of another.

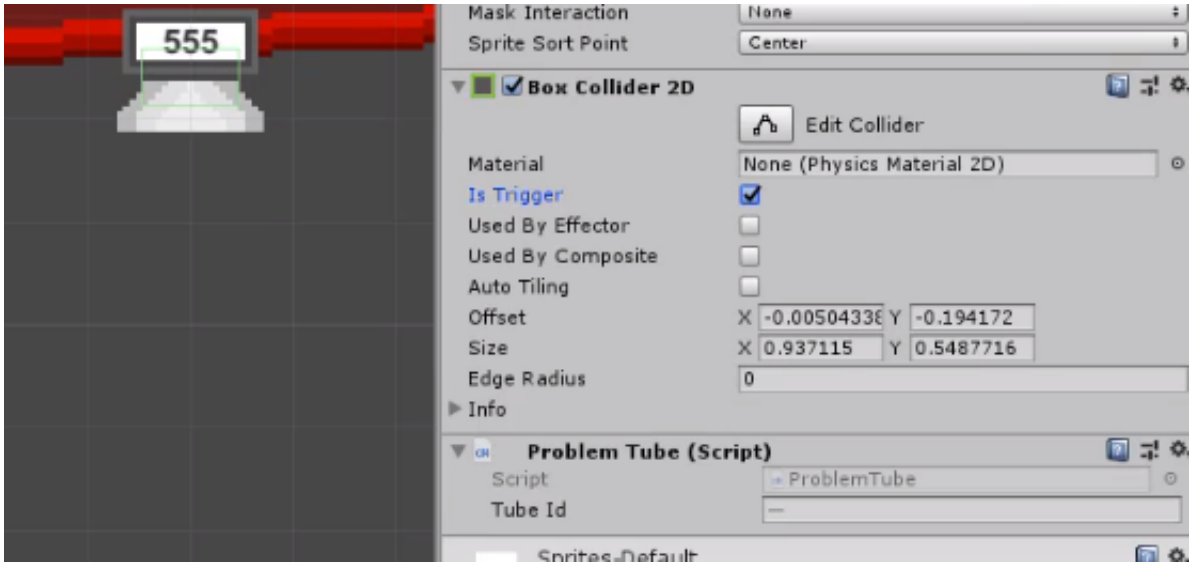
```
void OnTriggerEnter2D (Collider2D collision)
{

}
```

We check if the object that entered our collider was the player. If so, call the game manager's **OnPlayerEnterTube** function.

```
// was it the player?
if(collision.gameObject.CompareTag("Player"))
{
    // tell the game manager that the player has entered the tube
    GameManager.instance.OnPlayerEnterTube(tubeId);
}
```

Finishing Up the Problem Tubes

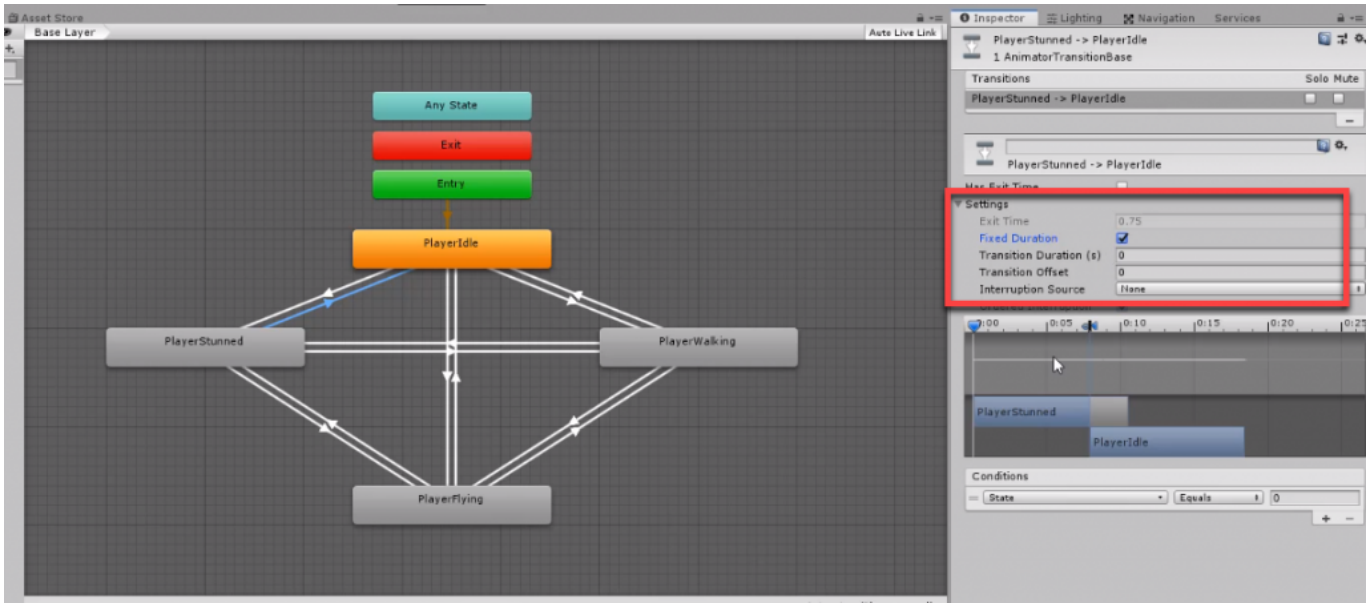
Back in the editor, let's attach the **ProblemTube** script to all of the tubes. Then set the **Tube Id** for each to be in order (0, 1, 2, 3). Also enable **Is Trigger** so the player will be able to enter it.



Fixing the Animations

If you press play, you'll see that the animations aren't quite right and there's a delay when the states change.

Go to the **Player** animation controller in the **Animator** window. For each of the state connections, open up the **Settings** tab and set **Transition Duration (s)** to 0.



Now the animations should be more snappy.

And that's the project done! Feel free to add more problems, more mechanics or change the existing ones to create your ideal project.