Intro to Scripting in Unity Engine

- As many of you may know, Unity Engine is one of the two big industry-standard game development engines, at the top alongside Unreal Engine. While Unreal Engine aims to be a technical marvel pushing the limits of gaming technology, Unity has always had different goals.

- Released in 2005 as a MacOS X exclusive app, Unity has had the mission of making game development as accessible as possible. Though Unreal Engine has technically existed since the 90s, it did not become freely accessible to the public until 2015, nearly 10 years later.

- This massive lead in exposure, paired with its wealth of publicly available first-party educational content and community content, and unparalleled breadth of platform support and optimization features, Unity is still the preferred engine for many indie developers, artists, and professional developers alike.

Overview

As an introductory workshop, we will be making a simple feature suite for a 2D Platformer. Specifically, we will be making an animated sprite that can run and jump, and some tiles that will have some different effects – normal terrain, damaging terrain, and movement terrain (spring/trampoline).

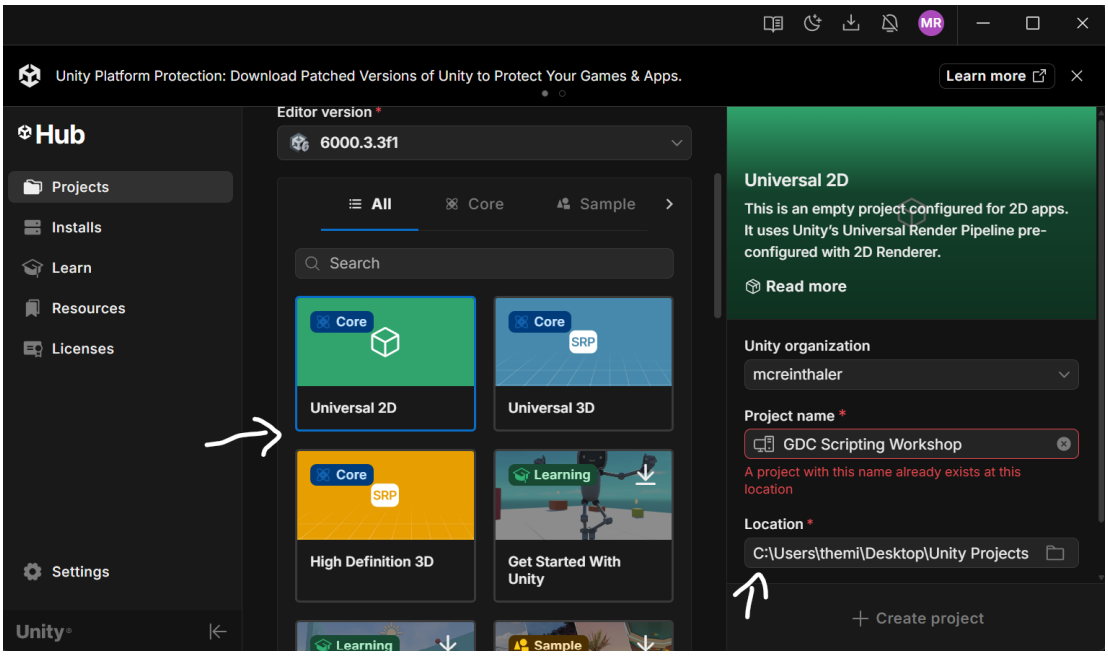
For anyone wanting to start from scratch (and reading this manual as a digital PDF) the raw assets we started with are linked here. The starting assets for the workshop will provided in a public repository on the MCC-GDC GitHub page.

The workshop will be focusing on these main aspects of programming and design:

- Class Inheritance and Overloading
- Movement and Collision
- Unity's Low-Level 2D Physics, Input, and State Machine/Animation Systems

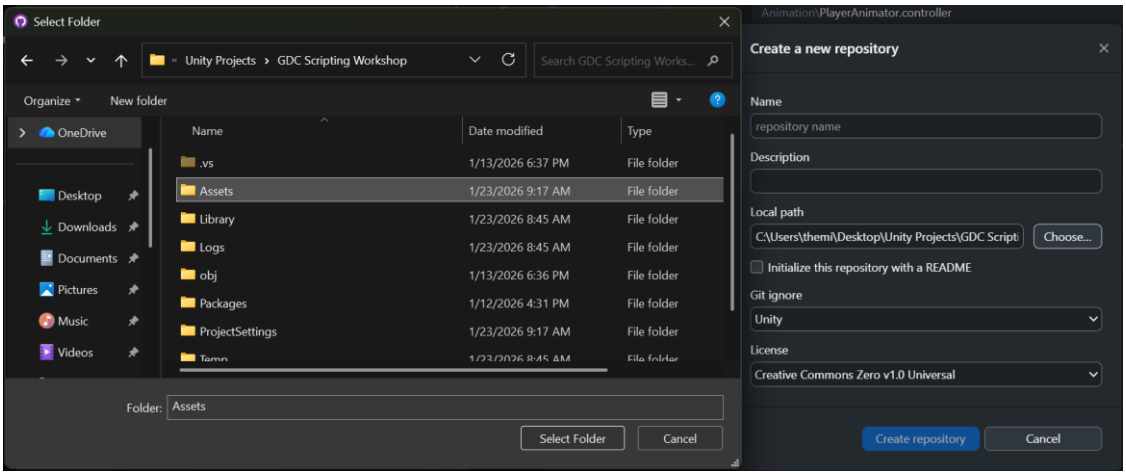
Creating Your Unity Project

- Open Unity Hub. In the top right corner, click New Project. Choose your preferred options for local or cloud storage, name it what you want, but make sure to select “Universal 2D” for the rendering template. Click “Create Project” and wait for the build to complete. (**Remember your file path you saved to!**) The Editor will open automatically when it is done.



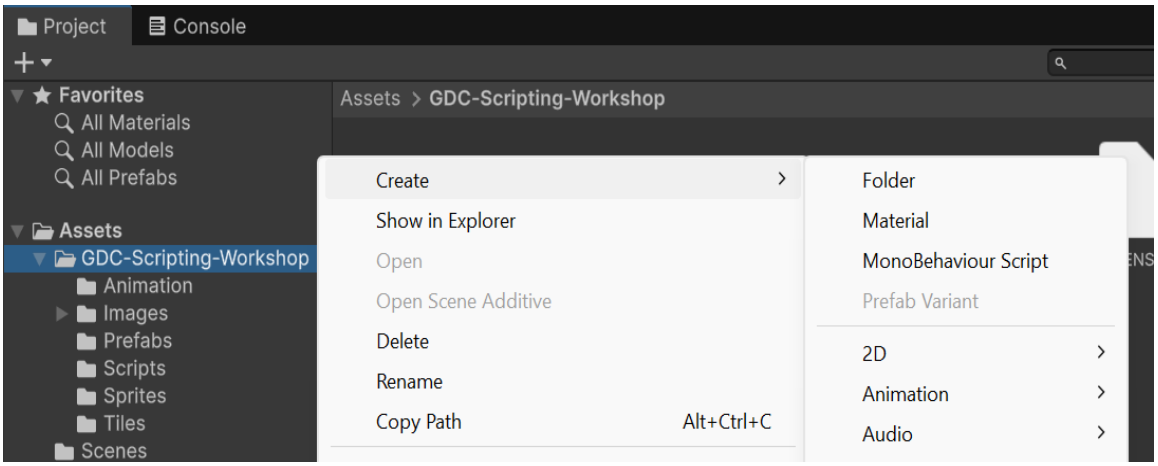
*Ignore the error message, I made the guidebook *after* the project

- If you want to use Git/GitHub to track your work – Once the editor opens, browse to your project folder at the Location you chose on creating the project. Enter the project directory and find your Assets folder. Copy the directory path. Then, open your preferred GitHub app and create a new repository or clone the starter content repository inside the Assets folder. Now, when you make new assets, make them inside the repository folder and Git will be able to track the changes. (Make sure to use the Unity .gitignore file)



Importing Assets

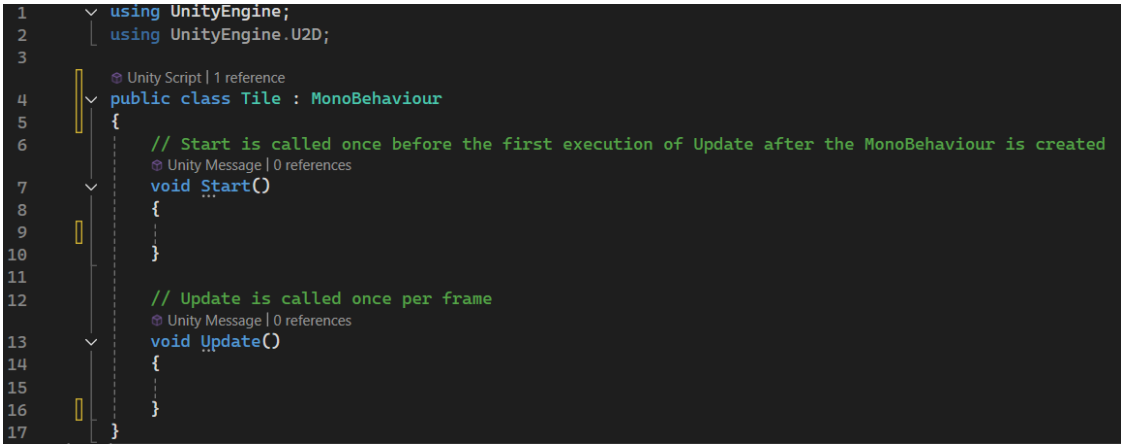
- Once you are in the editor and have your repository set up, you should see it pop up in your Assets folder in the editor. Once this is done, let's go ahead and create some folders for the different assets we'll be using. (To create a folder, right click the directory pane, hover over "Create", and click Folder. The name should be pre-highlighted for renaming.) Refer to image below for what to name your folders.
  - If you downloaded the starter content from GitHub, assets have been pre-sorted into the folders. If you've only downloaded the assets from Kenney, you will have to sort them yourself.



- Once this is done, browse the Tiles folder for the sprite you prefer to use, as we'll be starting off by making a normal terrain tile.

Making the Tile Parent Class

- In your scripts folder, right-click and select “Create > MonoBehaviour Script”. Name it “Tile”. MonoBehaviour is the name of the Unity system that gives game objects types, properties, and behaviors that can be unique or inherited. Once this script is created, double-click on it and Unity should open it up in your default IDE.
  - Microsoft Visual Studio 2022 is *highly recommended*. The new 2026 version does not yet have all of the same plugins and features for Unity, making it much easier to make mistake and much harder to troubleshoot.
- Once it opens, you should see something that looks like this. Here’s a little breakdown on what these mean:



  - Tile : MonoBehaviour
    - In C#, the colon symbol (:) is known as the Inheritance Operator, meaning, in this case, our Tile class *inherits* all the features of the MonoBehaviour class. This doesn’t mean much right now, but we’ll be touching on inheritance a lot pretty soon.
  - Start and Update functions
    - As said in the default comments, these are inherent functions to Unity’s game loop that allow you to set up and update the objects. These are the two inherited functions for *every* MonoBehaviour script, and the two most useful.

Customizing Our Tile Class

- The first two changes we're going to make to our Tile class are as follows.
    - Change "public class" to "public abstract class".
        - The "abstract" keyword for classes means that you cannot create objects of that specific class, but the class can still be inherited by other classes, and have access to its members and methods, known as "child" classes.
    - Add an abstract "OnCollisionEnter2D" method.
        - Abstract methods can be declared in an abstract class, and while this means the method cannot be defined in this class, any non-abstract class that inherits it *has to* define it in order to be valid.

```
public abstract class Tile : MonoBehaviour
{
    // Start is called once before the first execution of Update after the MonoBehaviour is created
    Unity Message | 0 references
    void Start()
    {

    }
    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {

    }

    Unity Message | 1 reference
    protected abstract void OnCollisionEnter2D(Collision2D collision);
}
```

- The "permission" keywords (public, protected) you see here are relevant. Here's what they mean.
    - Public – can be accessed or changed by any object in the program during runtime. In Unity, you can change a member's value in the Editor, by default. (We'll see this later)
    - Protected – Can only be accessed by itself or by children of its class.
    - Private – Can only be accessed by itself.
        - Not displayed here, but private is the default protection level of classes. Therefore, Start and Update are private, as they do not specify otherwise.

- Now, we'll add some members to make sure our tiles behave properly. Since our Tile class is abstract, we're just making sure every child class we make of it has to have the necessary parts.
    - Sprite – A graphical representation of our tile
    - BoxCollider2D – To enable collision logic

```
public abstract class Tile : MonoBehaviour
{
    private SpriteRenderer Sprite;
    private Collider2D Collider;
```

- Pro Tip: All member variables/properties should always be declared above/before any methods (member functions)

- And since our properties are private, that means we can't directly set it in the Editor. So, in your Start function, change the protection level to "protected" and add the following lines of code:

```
protected void Start()
{
    if(GetComponent<Collider2D>() != null) //Check to see if there's already a collider component
    {
        Collider = GetComponent<Collider2D>(); //If there is, use that one
    }
    else
    {
        Collider = gameObject.AddComponent<BoxCollider2D>(); //Otherwise, use BoxCollider as default
    }   //If you don't want a collider, the Start function can be overridden in an inherited class

    Sprite = GetComponent<SpriteRenderer>();

}
```

- You don't need to add the green comment lines, these are just explanations
- By default, a collider will configure itself to match the size of the object it's attached to. If you need to have a different size, you should probably add collider component to the object and configure it like that.

- Then, save your file. Go back to the editor and create a new MonoBehaviour script. Name it "GroundTile". This will be our normal, walkable terrain. Open your script and replace the MonoBehaviour inheritance with "Tile". You may notice some errors pop-up. This will happen because we haven't yet added all of our members required by the abstract class we're inheriting.

- Add the following code to make the error disappear

```
Unity Message | 1 reference
protected override void OnCollisionEnter2D(Collision2D collision)
{

}
```

- The “override” keyword here is necessary to “get rid” of the abstract type of the inherited function, so that we can define it. We’re not going to add any code here, but it is something we have to do.

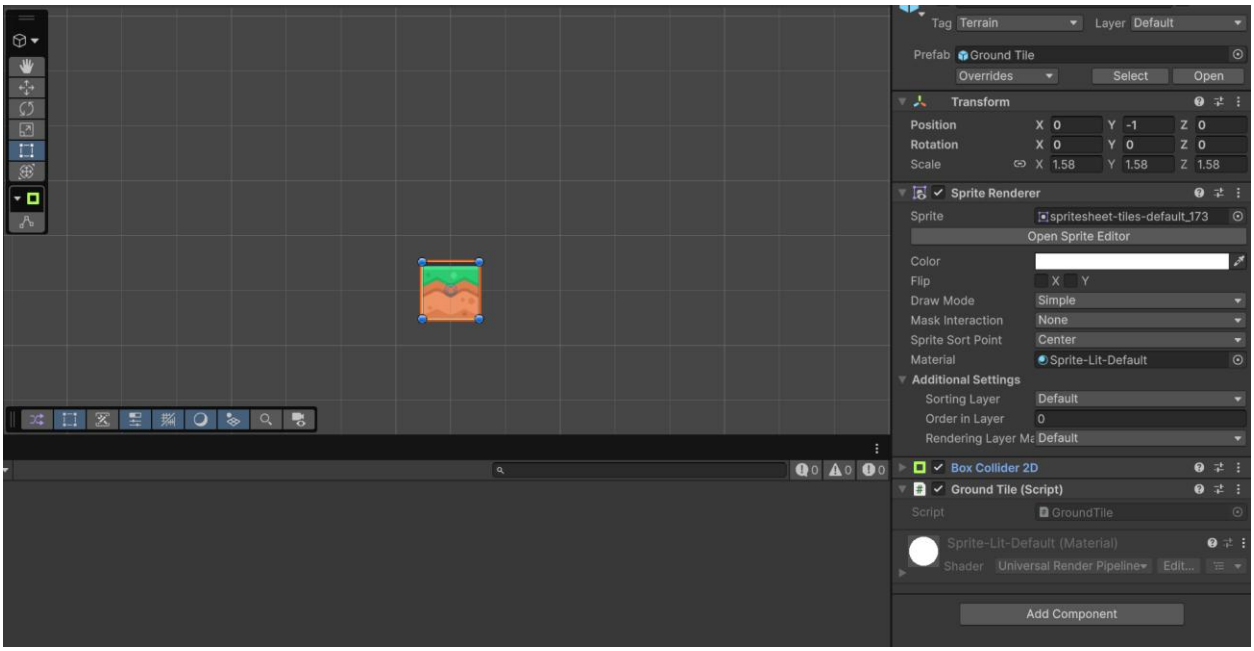
- Then, alter your Start function to be a “new” function, so that it can call base.Start and call the GetComponent we added in the Tile parent class. You can also add extra code on top of that, and it won’t affect other child classes of Tile, if you see fit.

```
new void Start()
{
    base.Start();
    gameObject.tag = "Terrain";
}
```

* Setting the tag to “Terrain” is used in the PlayerCharacter script later

- Now, return to the editor. Choose your favorite sprite for a terrain tile (just make sure it’s square for now) and click-and-drag it into the scene. You’ll notice it automatically becomes a game object with a Sprite Renderer component already attached. (That’s why we didn’t mess with the Sprite member in the script)
  - Note: I changed the Scale of my terrain tile to 1.58 in X and Y. This gets it pretty close to 1x1 size in Unity units, but mileage may vary depending on machine. Adjust how you see fit.

- Once you've done this, click on the new object in the Hierarchy pane, then click-and-drag your GroundTile script from your Assets folder to the object's Inspector panel. This will attach the script to the object and *voila!* You have an object of type GroundTile. Now, click the "Add Component" Button and search for "BoxCollider2D". Add the component. This isn't necessary because our Start function in our Tile class adds one if there isn't one, but you can if you want a different Collider shape or settings.
    - Note: If you want to change the start function to have a different shape or have no collider at all



- Once you've attached the script and (optional) box collider, go into the Hierarchy. Rename the object "Ground Tile" and drag it into the Prefabs folder in the Asset window. The object should turn Blue, as it is now saved as a "Prefab", which essentially means you can now easily make copies of your ground tile, and edit the properties of it's components in a way that will effect every duplicate you make. Conversely, if you edit an *instance* of a prefab (the actual object in the Scene/Hierarchy) it won't affect the original prefab.

- Now, we will move on to making the Player Character.

Making the Player Character

- It wouldn't be a platformer without a little guy to move around. First things first, go into the Sprites folder and take a look at the spritesheet for the player characters. Choose your favorite one and drag the "Idle" image into the scene. (The one standing and facing forward) This will, like before, create a game object with a preattached Sprite Renderer. Grab the pre-provided PlayerController script and attach it to this object. (Or make your own)

- While we're here, also click the "Add Component" button and add a "Rigidbody2D" and a "CapsuleCollider2D" component. Then, in the Animations folder, find the PlayerAnimator component and attach it to the player character object. Finally, drag the game object into the Prefabs folder to save it as a prefab. Delete the object in the hierarchy.

- In the pre-provided PlayerController script, you should see some pre-provided members and methods, mainly to handle animation and initialize using the input system.

```
using System;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.LowLevelPhysics2D;
Unity Script | 0 references
public class PlayerControllerStarter : MonoBehaviour // Remove "Starter" from the class name before starting the section
{
    private InputAction moveAction; // Standard map is WASD
    private InputAction jumpAction; // Standard map is Spacebar
    private Animator Animator;
    private SpriteRenderer Sprite;
    private Rigidbody2D rb;

    public float moveSpeed = 6f;    // Adding a value to a public member sets the default value
    public float maxSpeed = 6f;     // It can still be accessed/changed in the Editor
    public float jumpForce = 150f;  // If you do change it in the editor, it will override the default value

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    Unity Message | 0 references
    void Start()
    {
        moveAction = InputSystem.actions.FindAction("Move");
        jumpAction = InputSystem.actions.FindAction("Jump");
        Animator = GetComponent<Animator>(); // Animator controller is pre-provided in the Animation folder
        rb = GetComponent<Rigidbody2D>(); // Enables the physics system for the object
        Sprite = GetComponent<SpriteRenderer>();
    }
    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        var moveValue = moveAction.ReadValue<Vector2>(); // Listener for any Move input action

        if (jumpAction.WasPressedThisFrame()) // Listener for any Jump input action
        {                                     // Using this specific function means the jump listener won't repeat if held
            if (!Animator.GetBool("IsJumping"))
            {
                Animator.SetBool("IsJumping", true);
            }
        }
    }

    Unity Message | 0 references
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if(collision.gameObject.CompareTag("Terrain"))
        {
            if (Animator.GetBool("IsJumping"))
                Animator.SetBool("IsJumping", false);
        }
    }
```

- As a preface, there are multiple ways you can go about designing the movement here, but we are going to do it in a way that employs the unity Low Level Physics system. (Notice the “using UnityEngine.LowLevelPhysics2D” tag in the file header.)
- This is the system that interacts with the Rigidbody components if they’re attached to Game Objects, as our Player Character object does.
- Rather than moving the character to specific positions based on input, the Low Level Physics system allows us to add and/or set Force and Velocity properties to our character. This gives that feeling of speeding up, slowing down, and smooth gravity trajectory.

- You may already see that we have some member values for moveSpeed, jumpForce, and maxSpeed. This will help us dial in our max running speed, how fast we accelerate, and how high we jump. Since they’re public, we’ll be able to adjust the values in the editor after we have the movement logic added.

- The commands that read from the input system are already provided, so we’re going to be extending those pieces of code to get our movement, starting with running. The “Move” action by default is mapped to the WASD keys, and Jump mapped to the spacebar.

- The “ReadValue” function we’re calling returns a 2D Vector (x, y) with values -1 or 1 in either dimension, depending on which keys we’re pressing. X handles horizontal, -1 going to the left and 1 going to the right, and -1 and 1 going down and up in the Y dimension, respectively (if no button pressed, the value is 0). Knowing this, and the fact that we don’t want to move in the Y direction unless we’re jumping, use the moveValue and moveSpeed variables to set the player character’s linearVelocity in the X axis. (Hint: linearVelocity is a property of Rigidbody)

- Once you’ve accomplished that, go back into the editor and line up some tiles for you to run on and test out your new movement logic. If it’s too fast or too slow, or seems like it’s not moving at all, try adjusting the moveSpeed property in the PlayerCharacter inspector pane (should be visible on the script component. Make sure they’re public!). There’s not necessarily a perfect or correct way to do this, so if it works, it works!

- In testing your horizontal movement, you may notice that the player character is running backwards instead of turning around. Once you're happy with your movement, add this block of code to your update *after* your movement logic to fix that.

```
if (rb.linearVelocityX < 0)
{
    Sprite.flipX = true;
}
else
{
    Sprite.flipX = false;
}
```

- It just checks every frame to see if the character is moving backwards, and if it is, flips the image to face the opposite way.

- Moving on, let's add our jumping logic. To think of it, we'll want jumping to behave a little differently. It's not really a smooth, constant driving force like running, but more of a one-time burst of movement in the vertical direction. There's a method to do this: Rigidbody.AddForce (There's also AddForceX or Y, you should know which one to use)

- Using the AddForce method and the jumpForce member, add some jump logic is the body of the if (jumpAction.WasPressedThisFrame()) block.

- **Challenge**: Figure out a way to prevent multiple jumps.

Finishing a Level: Spawn and Win Points

*Warning: this section is quite a spike in the complexity of programming principles we're using

- The last thing we need to make for a "functional" level is a place for our player character to spawn, and a point we have to reach to beat the level. We will be doing this via a Level Manager script. Make two new MonoBehaviour scripts in your Scripts folder named "LevelManager" or something similar. Create an empty game object in the hierarchy, name it "LevelManager" and attach the script to it. Then, open the script in your IDE.

- Let's take a moment to list out what we want our Level Controller to handle:
    - Spawning/Despawning the PlayerCharacter
    - Tracking individual spawn points/checkpoints
    - Tracking the Win State/Condition
    - Resetting the level between different instances of it

- First things first, we'll set up the member variables we need to handle all of this

```
public static LevelController Instance { get; private set; }
5 references
public PlayerController PlayerInstance { get; private set; }

public PlayerController PlayerCharacterPrefab;
```

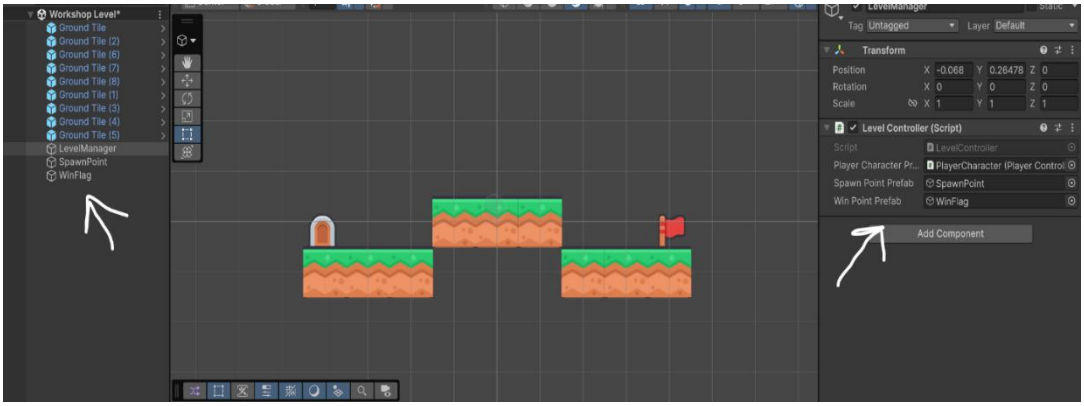
- And then add some functions to handle the events we want it to handle

Note: Instance-based level design is a concept that deserves its own workshop. For the sake of brevity and time, we're using it now to offload our logic of when to call the Spawn and WinGame functions to the game objects that observe those conditions (the Player Character and WinFlag.

```
void Awake()
{
    if (Instance != null)
    {
        Destroy(Instance);
    }

    Instance = this;
}
```

```
1 reference
void Spawn(Vector3 location)
{

}
0 references
public void WinGame()
{

}
```

- Now that we have our objects defined, let's place them in the level. In the Tiles spritesheet, there are plenty of options for objects to use. Personally, I'll choose a door for my spawn point and a red flag to be my win point. Once they're added in your level, drag the objects from the hierarchy into the member slots in the LevelController script in the inspector. If you haven't yet, you can also take this time to set up your arrangement of terrain tiles.



- Make sure to delete your PlayerCharacter object from the level Hierarchy, as now we'll make the Level Controller handle spawning it.

```
void Start()
{
    SpawnLocation = SpawnPoint.transform.position;
    Spawn(SpawnLocation);
}
```

```
void SpawnPlayer(Vector3 location)
{
    if (PlayerInstance != null)
    {
        Destroy(PlayerInstance.gameObject);
    }

    PlayerInstance = Instantiate(PlayerCharacterPrefab, location, Quaternion.identity);
}
```

*Quaternion means rotation, Quaternion.identity means 0 rotation

*We do need to destroy the game object the player is attached to. When referencing a "PlayerController" object by itself, it's actually only referencing the script component. We need to destroy all other components associated as well

- In your PlayerCharacter prefab, in the "Additional Settings" dropdown, change the "Order In Layer" property to 3, or your player character will render behind the spawn point sprite.

- Now, we can add an OnDeath function to our PlayerCharacter that will call the Spawn function of our level controller to add Death/Respawn behavior. Now, we'll make a damaging tile to test if that logic works.

Making a Damaging (Hazard) Tile

Time for a bit of a challenge:

- In order to see if our level controller handles spawning/respawning correctly, we'll need to add an instance of our player character "dying". To do this, let's make a tile that kills the player character on collision.
- In the Tiles sheet, find a tile that looks dangerous and drag and drop it into the Editor. Drag it into the prefabs folder and make a new "HazardTile" script to attach to it. Make sure it inherits "Tile".
- Now we're going to overload the tile's OnCollisionEnter method to deal damage to/kill the player character. Here are some tips:
    - Death logic should be handled by the PlayerController script. Therefore, you can either have the OnCollisionEnter function call the OnDamage/OnDeath function (make sure it's public) or find a different way to initiate the method (Extension methods? Messages? There are many)
    - Use "<Component>.enabled = false" to disable any component you don't want acting during the death animation (disabling the collider disables collision, disabling the rigidbody disables gravity/physics, etc.)
- Here's the design that I decided to go for:
    - The collision with the tile checks if it's the player via CompareTag(), and if it is, apply a force to give it that classic "bounce off" effect (don't sue me Nintendo)
    - Collider.enabled = false disables the collider so it'll fall through the map tiles
    - Delay 2.5 seconds (Delay's parameter is in milliseconds) before respawning the player

```
Unity Message | 1 reference
protected override void OnCollisionEnter2D(Collision2D collision)
{
    var obj = collision.gameObject;
    if (obj.CompareTag("Player"))
    {
        obj = LevelController.Instance.PlayerInstance.gameObject;
        var rb = obj.GetComponent<Rigidbody2D>();
        rb.AddForce(new(50, 200));

        LevelController.Instance.PlayerInstance.OnDamageTaken();
    }
}
```

*In the HazardTile script

```
1 reference
public async void OnDamageTaken()
{
    var camera = GetComponentInChildren<Camera>();
    collider.enabled = false;
    this.gameObject.SetActive(false);

    await System.Threading.Tasks.Task.Delay(2500);

    LevelController.Instance.Respawn();
    Destroy(camera.gameObject);
}
```

*In the PlayerController script

*Notice the use of the "async" and "await" keywords. This is necessary to utilize the Task.Delay function, similar to sleep() in Python or SetTimeout() in Javascript if you're familiar.

- You can/should add a new animation clip in the animator to set the sprite to a new one on damage/death. This is a bit outside of the purview of this workshop and isn't too hard to do.

Adding UI and Winning the Game