# Unity for Newborns

JANUARY 01, 2021

# Important Notice

The contents of these labs are made to teach and stress some learning points and for mere “practice”, e.g: getting used to Unity layout, terminologies, etc.

By no means we claim that they are the best practice, in fact some of the ways may be convoluted and they won’t be done exactly this way by experienced coder but we can’t teach the “best practices” right away because it relies on many prior knowledge.

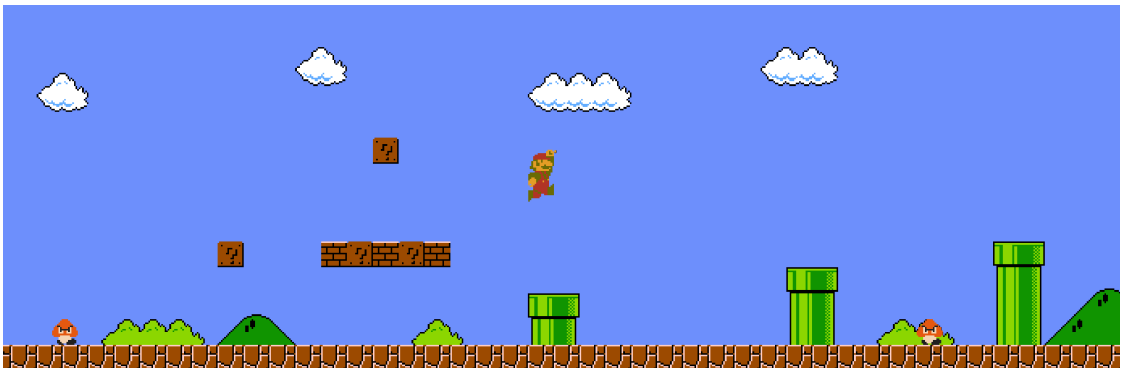
Experienced coders: keep in mind that you too were once a *noob*. You made mistakes.

*If you realise that some parts are troublesome or ineffective, then good for you. It means that you’re **smart** and **experienced**, and from now on you can embark on the journey to customize it to a more fitting way: simpler, better, more efficient, whatever it is. You can tell our teaching team your personal opinion, and constructive criticism is always welcome after class. We however expect a certain kind of mutual respect during the lab hours.*

# Learning Objectives: 2D Basics

- **Unity editor basics**: layout, arrangements of project files
- Edit **scene**, add **GameObject** & elements, create **prefabs**
- C# **scripting** basics
- **Unity Life Cycle** introduction and callback functions: Update(), Start(), OnTriggerEnter(), etc.
- Physics **simulation** basics: RigidBody2D, Collider2D
- **Binding** keys for input
- **UI elements**: Canvas, Text, Button
- Basic experience on **events**: OnClick() for Button

# The Classic Super Mario Game



The goal of this simple 4-week lab is to recreate basics this classic platform game: **Mario**, step by step and complete it by the end of Week 4. We will try to rebuild World 1-1 as closely as possible, although due to constraints of time, some features may be omitted. In Week 5 & 6, we upgrade our skills to explore Unity3D.

# Preparation

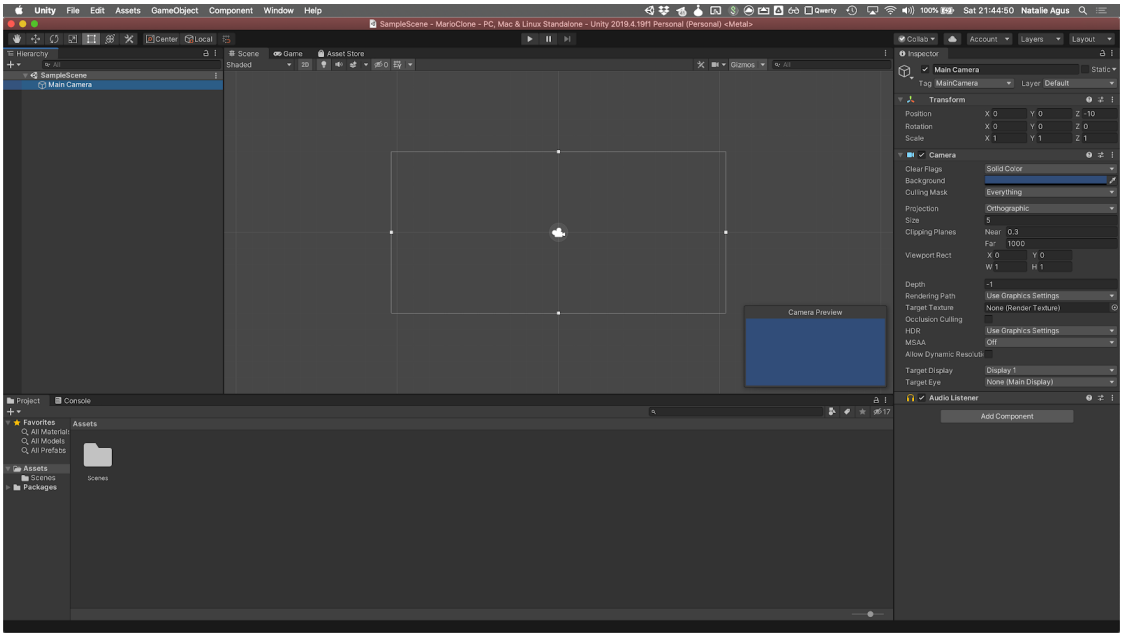
**Download the starter asset** from your **course handout**, under “Class Calendar” heading, *Week 1 Session 3 row.* This is a starter asset that you can import to your project and complete the lab. It contains all required images, sound files, etc so we can safe time searching all these stuffs.

***We do not own any of the assets and are simply using them for non-profit educational purposes.***

*Note for experts in Unity:* If you’re already an expert in Unity, you’re welcome to implement the final state of the project in any way you want, and implement the Checkoff. Jump to the Checkoff heading right away for more information.

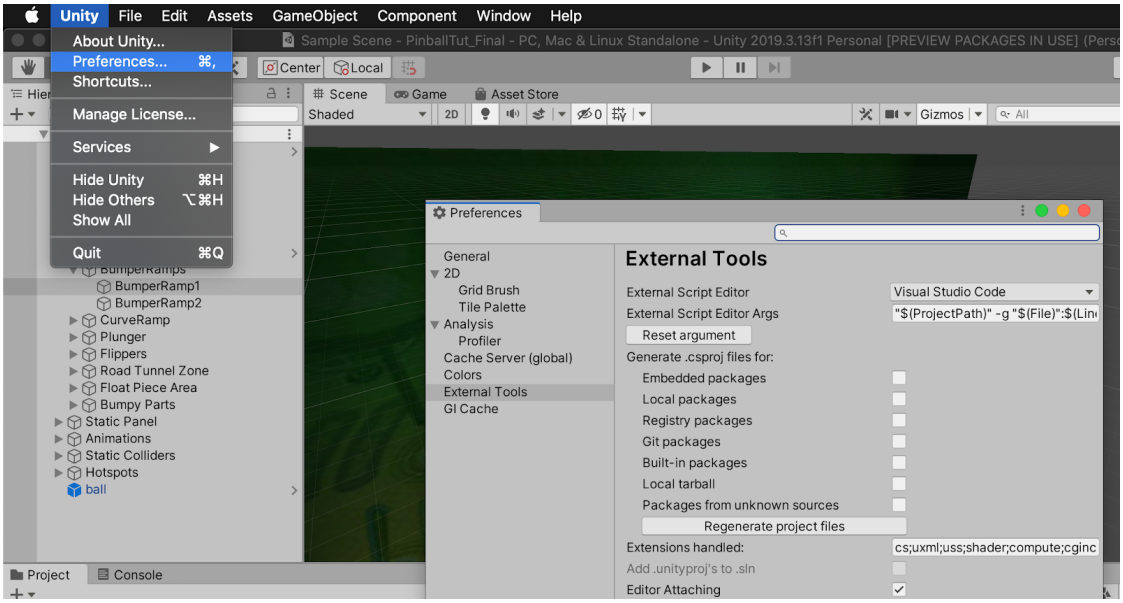
# Create New Project

Before we begin, create a **new unity project**.




Then, import the asset you downloaded. You should see a list of assets on the Project tab in the Unity editor.

It is also crucial to have a good code editor. Add your own script editor as shown. *Visual Studio Code is recommended.*



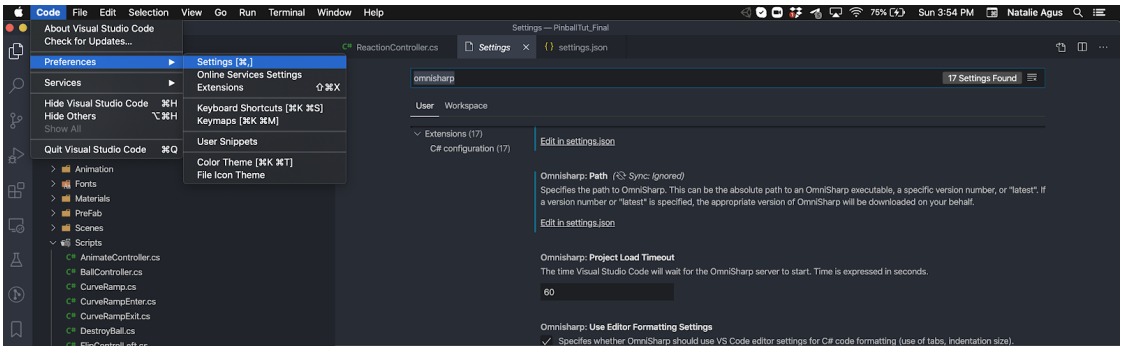

See the guide : https://code.visualstudio.com/docs/other/unity. For Mac users, simply: `brew install mono` on top of what the guide above told you to do. If you are NOT familiar with installing SDK or frameworks or editing `.json` files then just use the easy **MonoDevelop IDE** instead (heavier, not as pretty, but saves you the hassle).

Finally, check if `dotnet` is installed:

```
~ dotnet --info                                                    ✔ at 21:48:32
.NET SDK (reflecting any global.json):
 Version:   5.0.103
 Commit:    72dec52dbd

Runtime Environment:
 OS Name:     Mac OS X
 OS Version:  10.15
 OS Platform: Darwin
 RID:         osx.10.15-x64
 Base Path:   /usr/local/share/dotnet/sdk/5.0.103/

Host (useful for support):
  Version: 5.0.3
  Commit:  c636bbdc8a

.NET SDKs installed:
  3.1.406 [/usr/local/share/dotnet/sdk]
  5.0.103 [/usr/local/share/dotnet/sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.App 3.1.12 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
  Microsoft.AspNetCore.App 5.0.3 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 3.1.12 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]
  Microsoft.NETCore.App 5.0.3 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]

To install additional .NET runtimes or SDKs:
  https://aka.ms/dotnet-download
```

If you're using VSCode, edit `settings.json` to include the following:

```
"omnisharp.defaultLaunchSolution": "latest",
"omnisharp.path": "latest",
"omnisharp.monoPath": "",
"omnisharp.useGlobalMono" : "always"
```




# Housekeeping

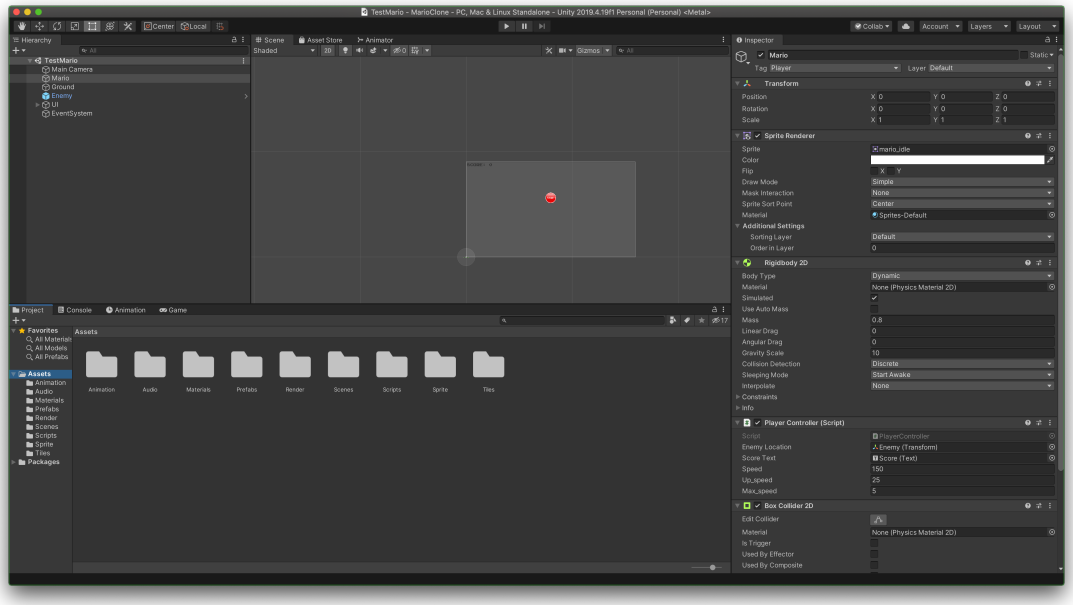
To work better, we need to set up the UI in a more comfortable way. We need at least the following windows:

1. Inspector (to have an overview of all elements in a GameObject),
2. Game Hierarchy (to have an overview of all GameObjects in the scene)
3. Scene Editor (to add GameObjects and place them)
4. Project (to show all the files),
5. Console (to see printed output)
6. Game Screen (to test your game)

Go to “Window” and select the windows that you want. Explore the options and arrange the tabs in any way you like.

Then, go to **Scenes** folder and rename your scene in a way that it makes more sense than “*SampleScene*”. Create **two more folders** called “**Scripts**”, and “**Prefabs**” under “**Assets**”. We will learn about them soon.

This is how my layout looks like.

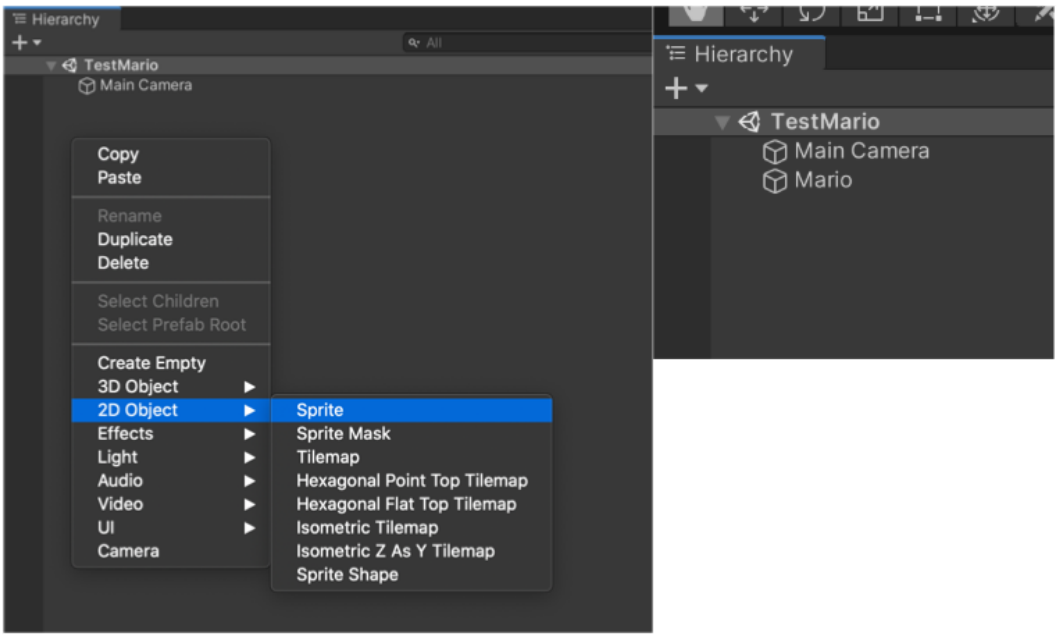


# The Basics

## Add GameObjects to the Scene

Everything you see on the game scene is a **GameObject**. It is the base class of all entities in the Unity Scene.

**Let's add Mario to the scene.** Right click in the Hierarchy tab and create a 2D Object with **Sprite** component. Change its name to "Mario".

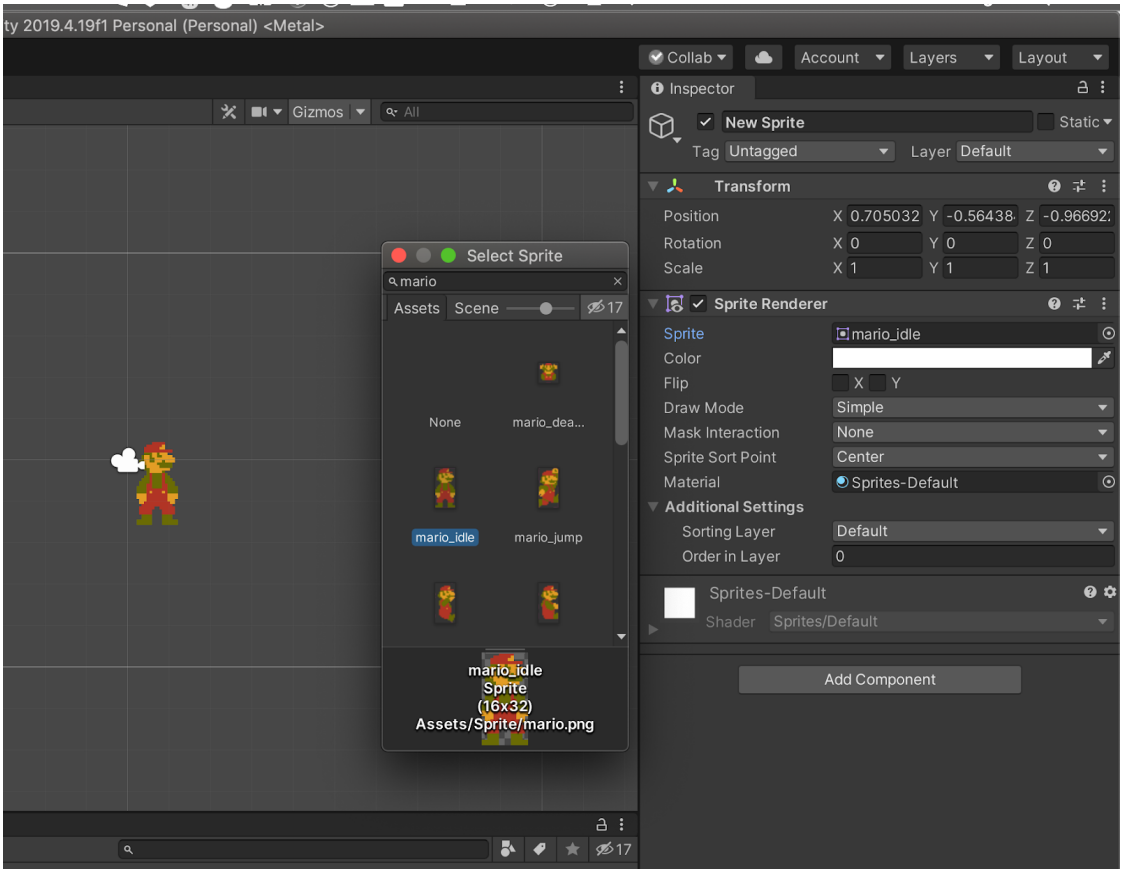



# The Inspector

On the **Inspector**, head to the `SpriteRenderer` element and add the *mario_idle* sprite. You should see something like this in the end. The inspector is pretty straightforward. It contains all elements attached to a particular gameobject:

- `Transform` dictates where this GameObject is in the Scene. You can change its coordinates with respect to the Camera Object so you can view it from differnt angles.
- `SpriteRenderer` works for dictating what the object should look like in 2D.

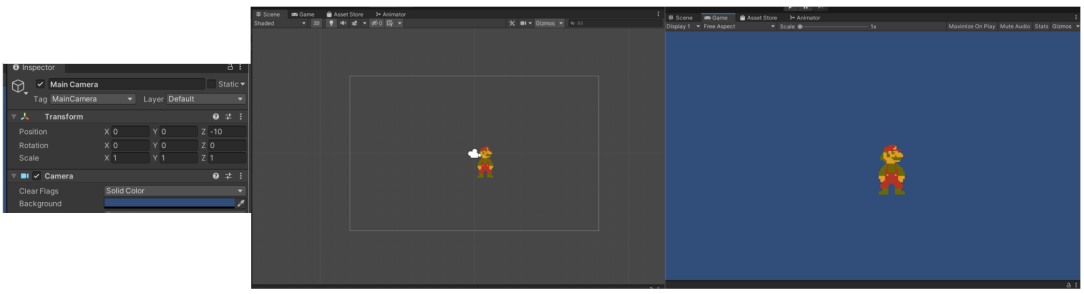
We can attach more elements to the GameObject, such as `RigidBody` for physics simulation. More on that later.




# The Camera

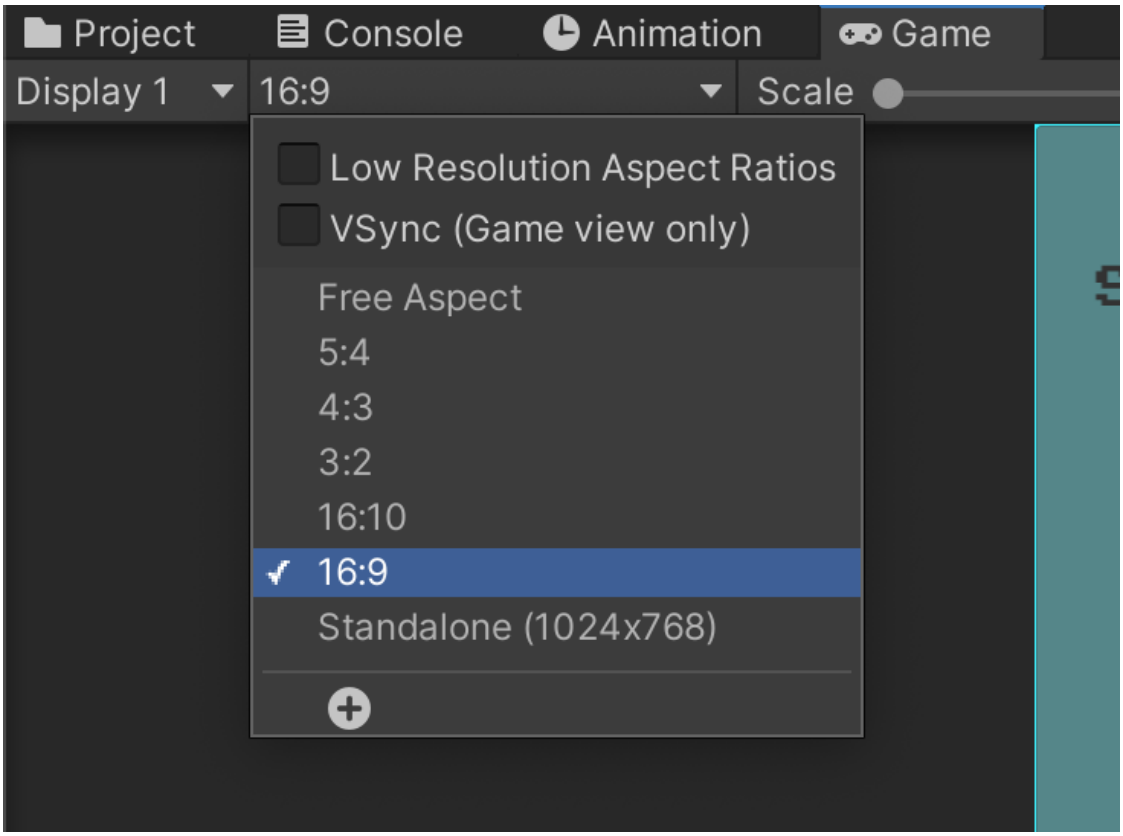
The “Scene” is the entire game world. What users can “observe” depends on the main camera. Notice that there’s always a Main Camera GameObject in your scene.

In your “Game” tab, you may see Mario with a blue background, but in the Scene tab, its Mario with a grey background. That’s due to your **camera setting.** We can adjust the camera to give different POV to the user or background color.



It is worth noting that you might want to set the **game aspect ratio**.

- Click the drag-down menu as shown and select the option that you're most comfortable with.
- E.g: selecting "Free Aspect" means that your window size will affect the camera "view".

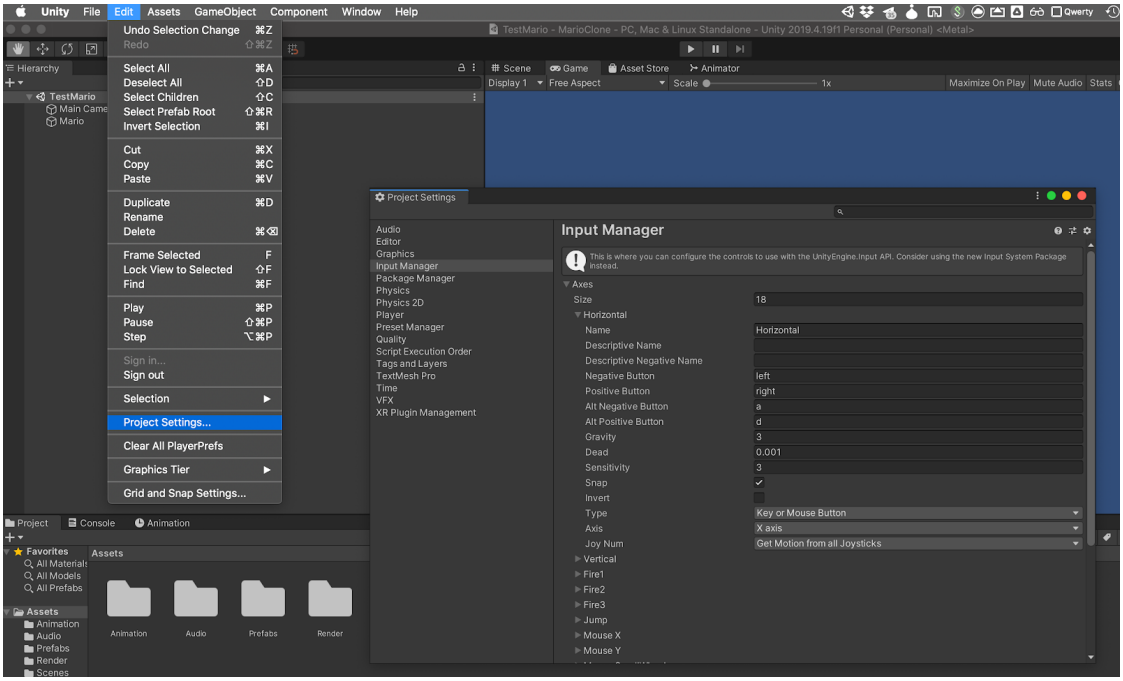



# Setting up Inputs

Go to **Edit » Project Settings** and click on **Input Manager**.

We want to test if we can control the movement of Mario using the keys "a" and "d" for movement to the left and right respectively.

- Check if the setting of "horizontal" axis is true. You can add your own key bindings here and name it.
- Later in the script, you can decide what to do if a certain named key is pressed.




# Creating a Script

Right click inside the **Scripts** folder in the **Project** window, create a new C# script `PlayerController.cs`. Here we will programmatically control Mario. Open the script with an editor of your choice.

**Let's attempt to move Mario via the script.** Firstly, Add `Rigidbody2D` component in the **Inspector** and set:

- `Gravity Scale` to `0`,
- `Linear Drag` to `3`
- `BodyType` to `Dynamic`

You are free to play with other parameters.

We can then control this component from the script. Paste the following inside `PlayerController.cs`:

```
public float speed;
private Rigidbody2D marioBody;
// Start is called before the first frame update
void  Start()
```

```
{
        // Set to be 30 FPS
        Application.targetFrameRate =  30;
        marioBody = GetComponent<Rigidbody2D>();
}
```

Add the script to Mario (Add Component » `PlayerController` script) then set `speed` to `70` in the Inspector. Afterwards, implement the **event callback** `FixedUpdate` as shown:

```
void  FixedUpdate()
{
        float moveHorizontal = Input.GetAxis("Horizontal");
        Vector2 movement = new Vector2(moveHorizontal, 0);
        marioBody.AddForce(movement * speed);
}
```

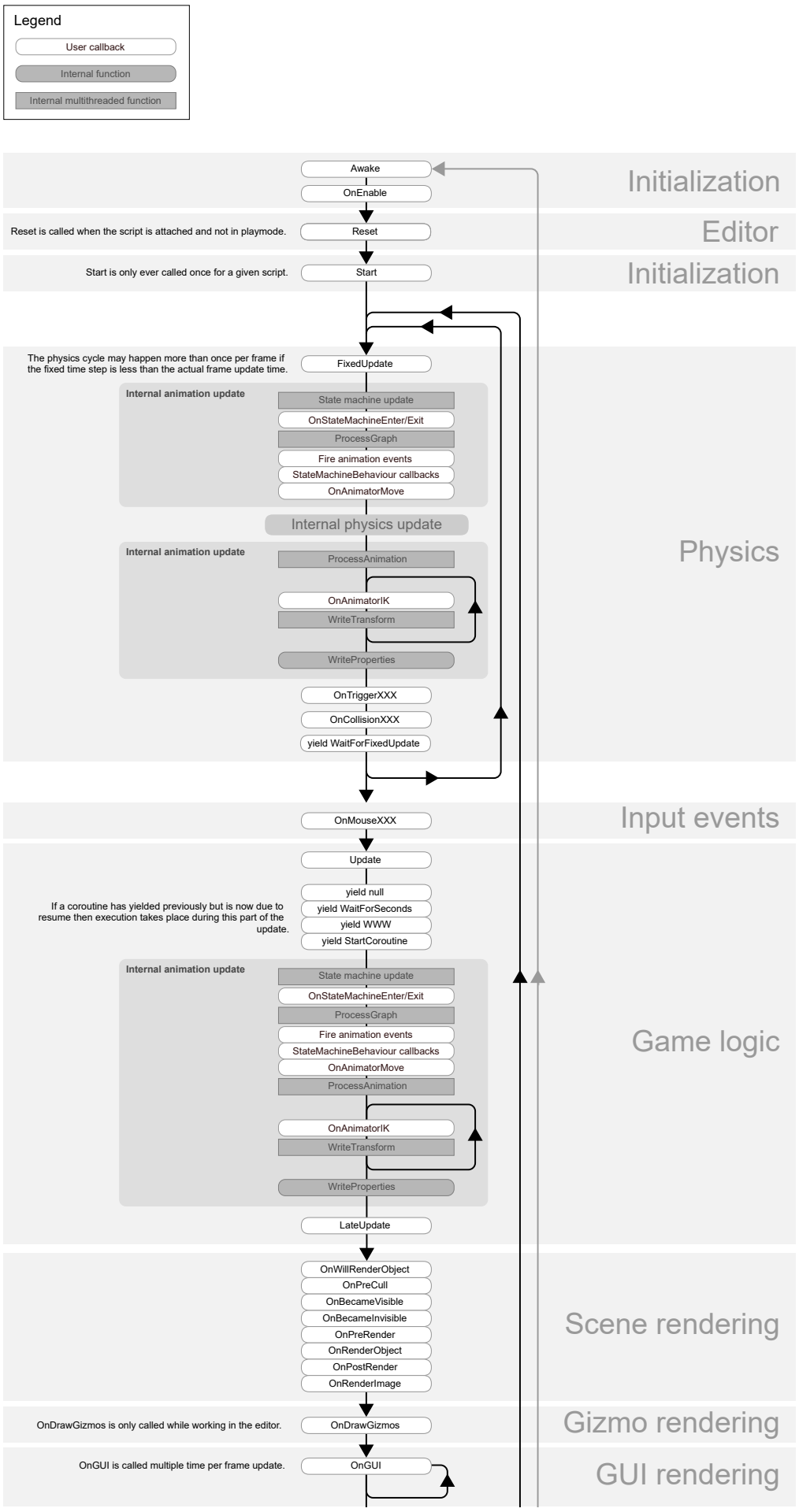
You can **test run** that now Mario can be moved to the left and to the right using the keys “a” and “d” respectively. *However it doesn’t feel quite right. We will fix this later but first, lets learn about Unity event functions.*

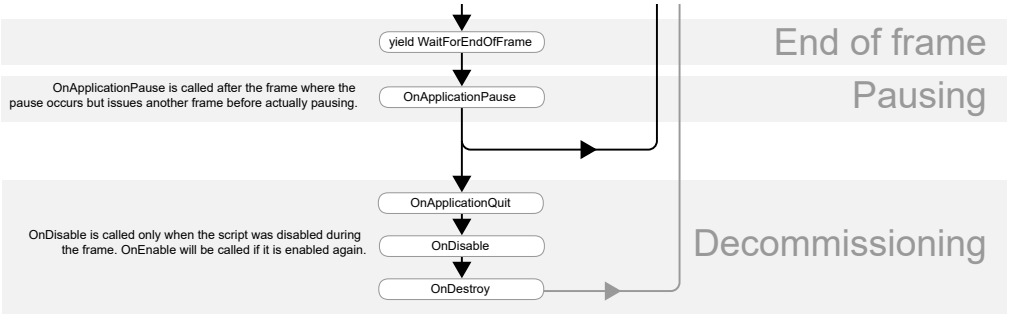
# Unity Order Execution of Event Functions.

We can implement the event functions in the script that’s attached to a particular GameObject. Notice that in the script we created,

- It inherits from MonoBehaviour, the base class from which every Unity script derives.
- It comes with two functions for you to implement if you want: `Start()` and `Update()`.

`Start()` is always called once in the beginning when the GameObject is instantiated, and then `Update()` is called per frame. **This is where you want to implement your game logic.** The diagram below shows the order of execution of event functions. They run on a single Unity main thread.


Legend
User callback
Internal function
Internal multithreaded function
Initialization
Awake
OnEnable
Editor
Reset is called when the script is attached and not in playmode.
Reset
Initialization
Start is only ever called once for a given script.
Start
Physics
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
Internal physics update
Internal animation update
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
Input events
OnMouseXXX
Game logic
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
LateUpdate
Scene rendering
OnWillRenderObject
OnPreCull
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Gizmo rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
GUI rendering
OnGUI is called multiple time per frame update.
OnGUI




The manual for all event functions can be found here. It is crucial for you to read it when the need arises, so you dont put weird things like implementing regular Physics simulation under `OnDestroy`.

Usually we don't touch all of them. One of the more common ones to implement are: `Start`, `Update`, `FixedUpdate`, `LateUpdate`, `OnTrigger`, `OnCollision`, `OnMouse` `OnDestroy`, and some internal animation state machines if you use `AnimationControllers`. We will learn that in the next series.

# RigidBody2D Setting

Now back to the issue how Mario's movement doesn't seem quite right. He seems to be *sliding*. We would expect him to stop the moment we lift the key, wouldn't we?

Setting the `BodyType` to `Dynamic` allows the **physics engine to simulate** forces, collisions, etc on the body. Since we're adding Force to Mario's body, it will obviously "glide" until the drag forces it to stop.

> Setting BodyType to `Kinematic` allows movement unders simulation but under very specific user control, that is you want to compute its behavior yourself – simulating Physics under your own rule instead of relying on Unity's Physics engine. Read more the documentation here.

# Improving Mario GameObject

## Stop Mario

To prevent this “sliding” feature that’s not very intuitive for platform game like this, we need to

- Set its velocity to 0 when key “a” or “d” is lifted up
- **Clamp** his speed to a maximum value so he doesn’t run faster and faster when we hold that “a” or “d” button.

Add the global variable `maxSpeed` and implement `FixedUpdate()` in `PlayerController.cs` .

```
public float maxSpeed = 10;
// FixedUpdate may be called once per frame. See documentation for details.
void FixedUpdate()
    // dynamic rigidbody
    float moveHorizontal = Input.GetAxis("Horizontal");
    if (Mathf.Abs(moveHorizontal) > 0){
        Vector2 movement = new Vector2(moveHorizontal, 0);
        if (marioBody.velocity.magnitude < maxSpeed)
                marioBody.AddForce(movement * speed);
    }
    if (Input.GetKeyUp("a") || Input.GetKeyUp("d")){
        // stop
        marioBody.velocity = Vector2.zero;
    }
}
```

*Note: if Mario still slides, try to export the game and check if it is not due to some lag in debug mode.*

## Make Mario Jump

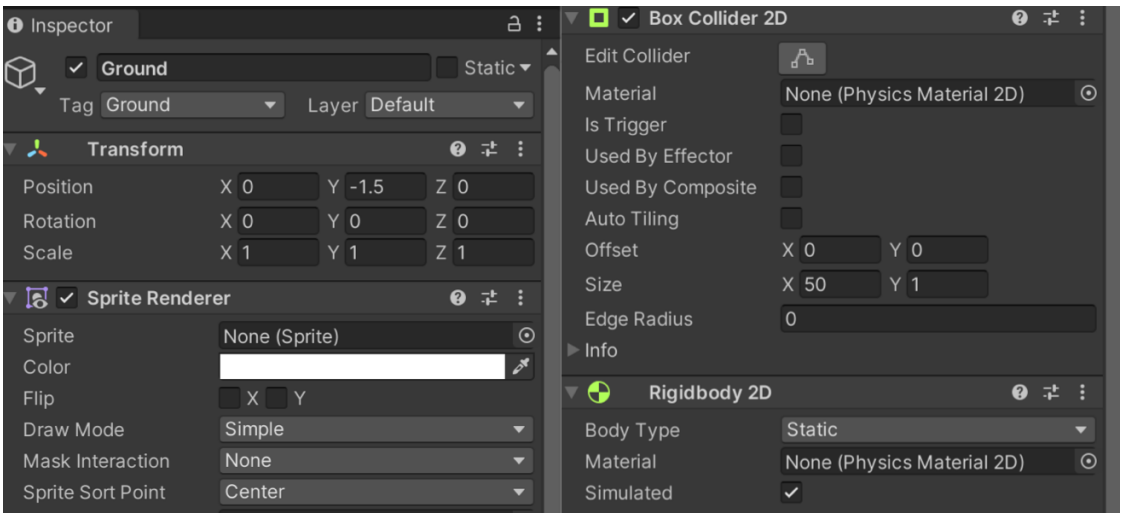
Let's make him jump to a fixed height whenever we press the **Spacebar** key once. We can leverage on the physics engine for this, but we need to enable gravity. Otherwise we have to make the Kinematics computation ourselves.

> Nobody's stopping you to do that, but due to time constraints let's not reinvent the wheel.

**Set `GravityScale` to `1`.** If you press play now, Mario will fall to **oblivion**.

We need to add some sort of a "floor" to prevent him from falling down.

- Create a new 2D Sprite GameObject and name it Ground. Add the components:
  1. `BoxCollider2D`
  2. `Rigidbody2D` (set to `static` type)
- Add a Tag called "Ground"
- Set its Position and match the components setting as shown.




Now we implement the `Collider` **callback** function called `OnCollision2D` in `PlayerController.cs`. The idea is that if Mario is on the ground, and if spacebar is pressed, we will add an Impulse force upwards. Pressing spacebar again **will not cause Mario to double jump.**

We need to have some kind of **state** variable for this. Add the following code to `PlayerController.cs`:

```
    private bool onGroundState = true;

    // called when the cube hits the floor
    void OnCollisionEnter2D(Collision2D col)
    {
        if (col.gameObject.CompareTag("Ground")) onGroundState = true;
    }
```

and the following inside `FixedUpdate()` method:

```
        if (Input.GetKeyDown("space") && onGroundState){
            marioBody.AddForce(Vector2.up * upSpeed, ForceMode2D.Impulse);
            onGroundState = false;
        }
```

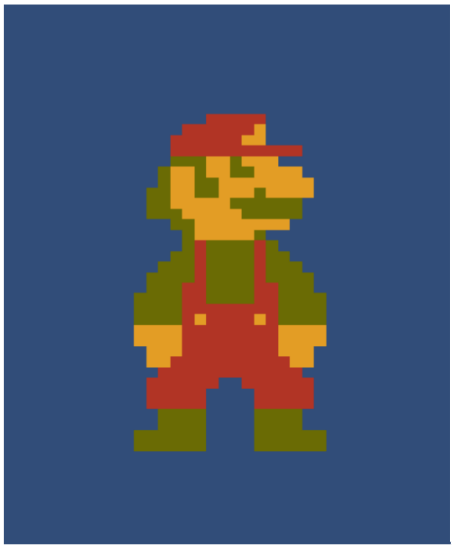
You can improve the controls and adjust the parameters: `speed`, `upSpeed`, and `maxSpeed` accordingly to get the right "**feel**". It can take quite a lot of time to get the **kinesthetics** right, but it is an important part of your journey in making a good game.

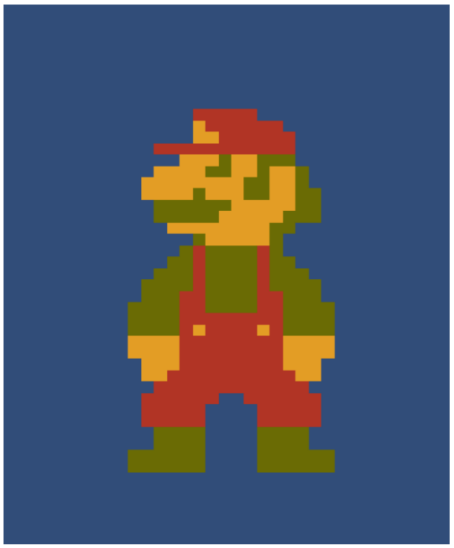
Focus more on these details instead of "expanding" your game. We don't require you to create a 1-hour long game, but rather a short and well designed game.

> Invest your time wisely.

# Flip Mario

Now let's fix mario's facing. If he is going to the left, he should be facing the left side and vice versa.





The direction he's facing should conform to the **last pressed key**.

We can do this by enabling the `flipX` property of its `SpriteRenderer` whenever key "a" is pressed, and disabling it whenever key "d" is pressed. Add these two global variables to the script:

```
private SpriteRenderer marioSprite;
private bool faceRightState = true;
```

**We have to control the `SpriteRenderer` component via the script.** You can pretty much get any component via `GetComponent<type>()` method in the script attached to the game object. Instantiate the `MarioSprite` under the `Start()` method:

```
MarioSprite = GetComponent<SpriteRenderer>();
```

Finally, implement the following under `Update` and not `FixedUpdate` since this logic has nothing to do with the Physics Engine:

```
    // toggle state
    if (Input.GetKeyDown("a") && faceRightState){
```

```
        faceRightState = false;
        marioSprite.flipX = true;
    }

    if (Input.GetKeyDown("d") && !faceRightState){
        faceRightState = true;
        marioSprite.flipX = false;
    }
```

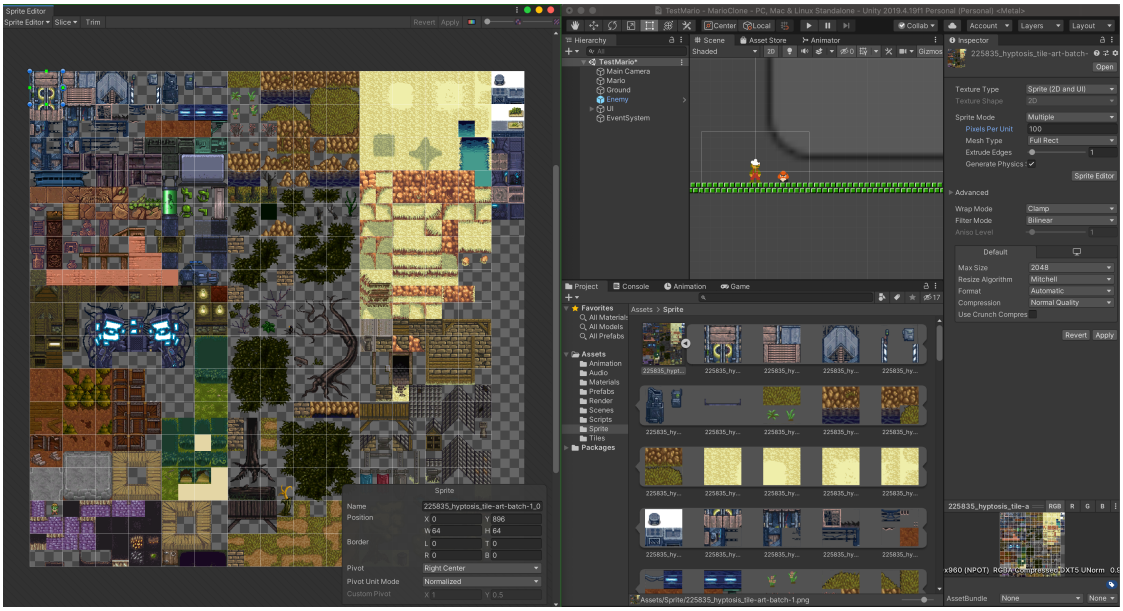
Your Mario will now face right and left accordingly as “a” or “d” is pressed.

# Importing Multiple Sprites

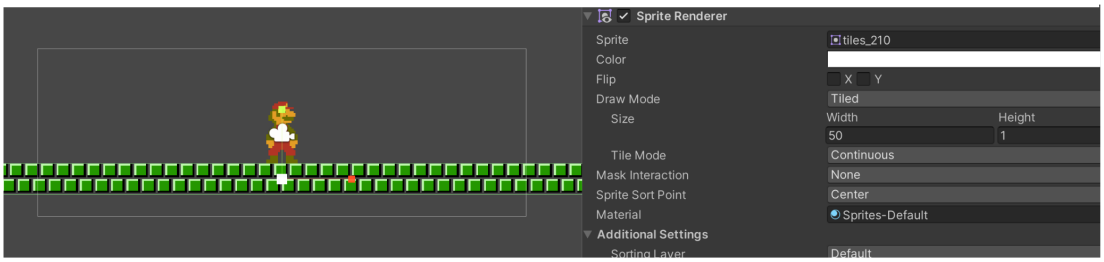
Right now our ground is simply “invisible”. We need a sprite for it. In fact, we need **a lot of sprites for this game**, e.g: the tile, enemies, obstacles, etc. We do not want to import them one by one. One way to import sprites quickly is to arrange **multiple sprites in a single image**, arranged nicely in area of x by x pixels. You can extract them all quickly using **Unity Sprite Editor**:

- Drag your 2D Tilemap asset to the Asset/Sprites folder.
- In the inspector, set the SpriteMode into multiple
- Click **Sprite Editor**

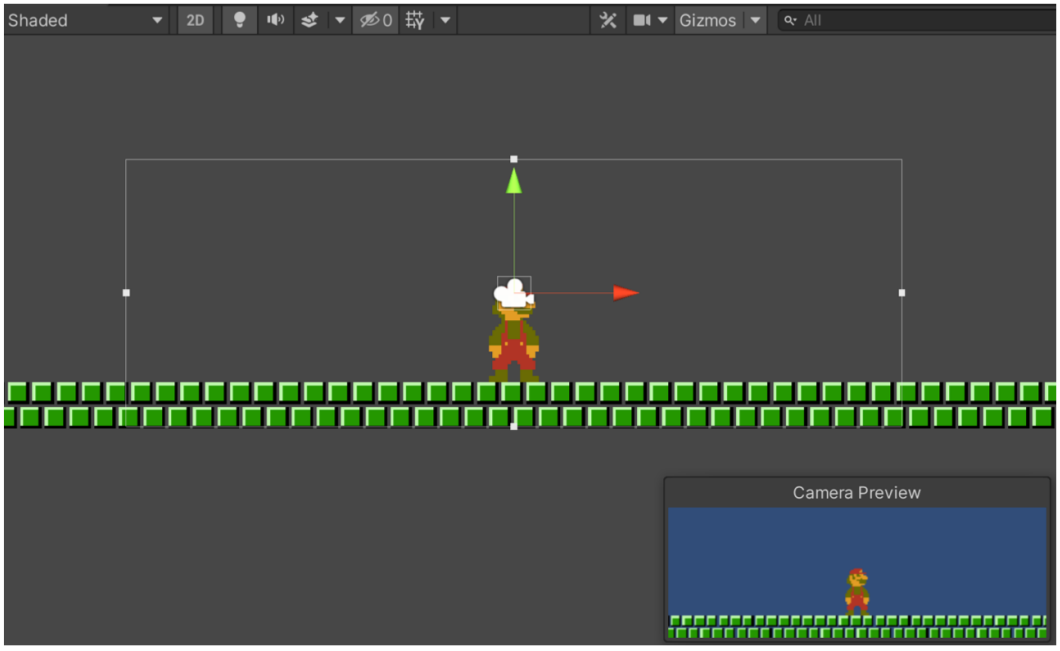
A Sprite Editor window will pop up and you can simply **slice** the sprites accordingly, and **apply** the changes. Notice that now your single sprite becomes many smaller ones that you can use for your GameObject’s renderer.



Set the `SpriteRenderer` component of `Ground` object to contain the following settings. We simply want to draw one of the sprites over it in `Tiled` fashion, so it repeats itself along the X axis.




Don’t forget to **adjust** the Camera’s `Transform` so that the ground is nicely flushed to the bottom of the screen.

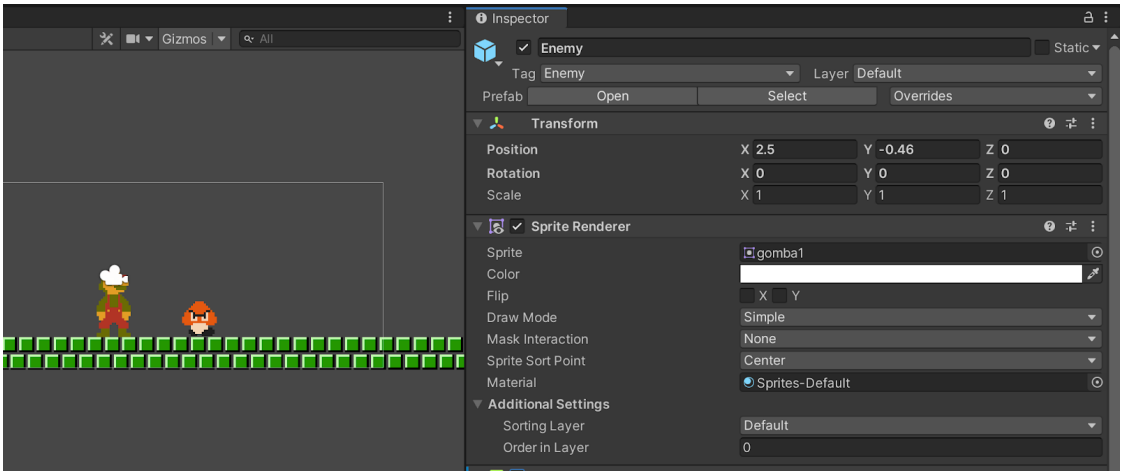



> You can also adjust the background color to be something else that's more aesthetically pleasing. In the Camera's inspector, change its `BackgroundColor`. We will learn later on how to create a better background, such as parallax background. Experiment with other camera settings as well, as as its Viewport Rect, Culling Mask, Clear Flags, etc.

# Adding Obstacle

Now its time to create the Enemy.

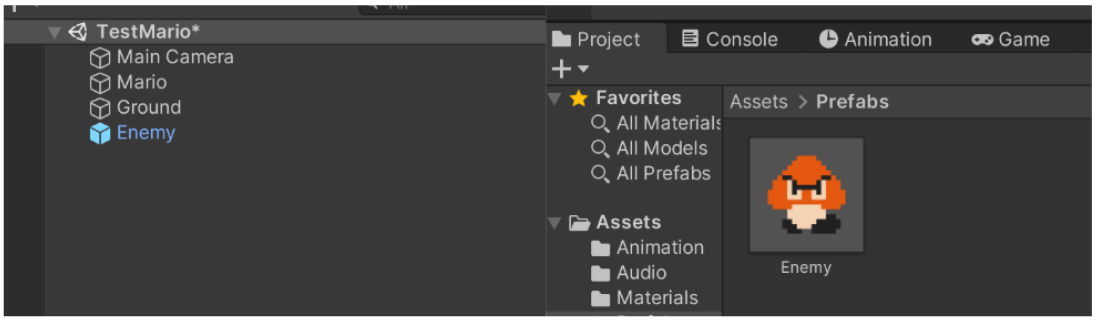
- Create a 2D Object » Sprite onto the scene, name it `Enemy`
- Put `gomba1` as its sprite, edit its `Transform` and you should see the following on your scene.
- Change its **Tag** to `Enemy` (create it).




# Create Prefabs

Drag the `Enemy` GameObject to the folder we created earlier called “Prefabs”. Notice how the logo of the cube will become **blue**. **Prefab is basically a reusable game object**. From now on, you can **change the prefab master** by **clicking on the prefab in this Prefab folder**, and all of your copies placed on **any Scene** will reflect the change.

> This is particularly good for items that are instantiated many times in the scene, such as this `Enemy`.




# Move the enemy

Let’s say we need the enemy to patrol left and right up to a certain offset X from its starting position. Create a new script called `EnemyController.cs`. Instantiate the

following variables and implement the `Start()` function. Add also the following two methods:

```
private float originalX;
private float maxOffset = 5.0f;
private float enemyPatroltime = 2.0f;
private int moveRight = -1;
private Vector2 velocity;

private Rigidbody2D enemyBody;

void Start()
{
    enemyBody = GetComponent<Rigidbody2D>();
    // get the starting position
    originalX = transform.position.x;
    ComputeVelocity();
}
void ComputeVelocity(){
    velocity = new Vector2((moveRight)*maxOffset / enemyPatroltime, 0);
}
void MoveGomba(){
    enemyBody.MovePosition(enemyBody.position + velocity * Time.fixedDeltaTime);
}
```

The idea is to allow the enemy to patrol up to `5.0` units to *the left and to the right*, and **change** direction accordingly when the max offset distance is reached. We also want to control its `speed`.

We can compute the **required** velocity by dividing supposed distance travelled with time, and then compute the position at each `Time.fixedDeltaTime`. Then, we can move the enemy to the calculated position: **`original_position_vector + velocity_vector * delta_time`**

Finally, since we do not need to perform a full-blown physics simulation on the enemy, **we can set its `RigidBody2D` `BodyType` to `Kinematic`.** We are simply

moving it to patrol around desired location, and perhaps later on to detect "collision".

Then implement `Update()` method in `EnemyController.cs` .

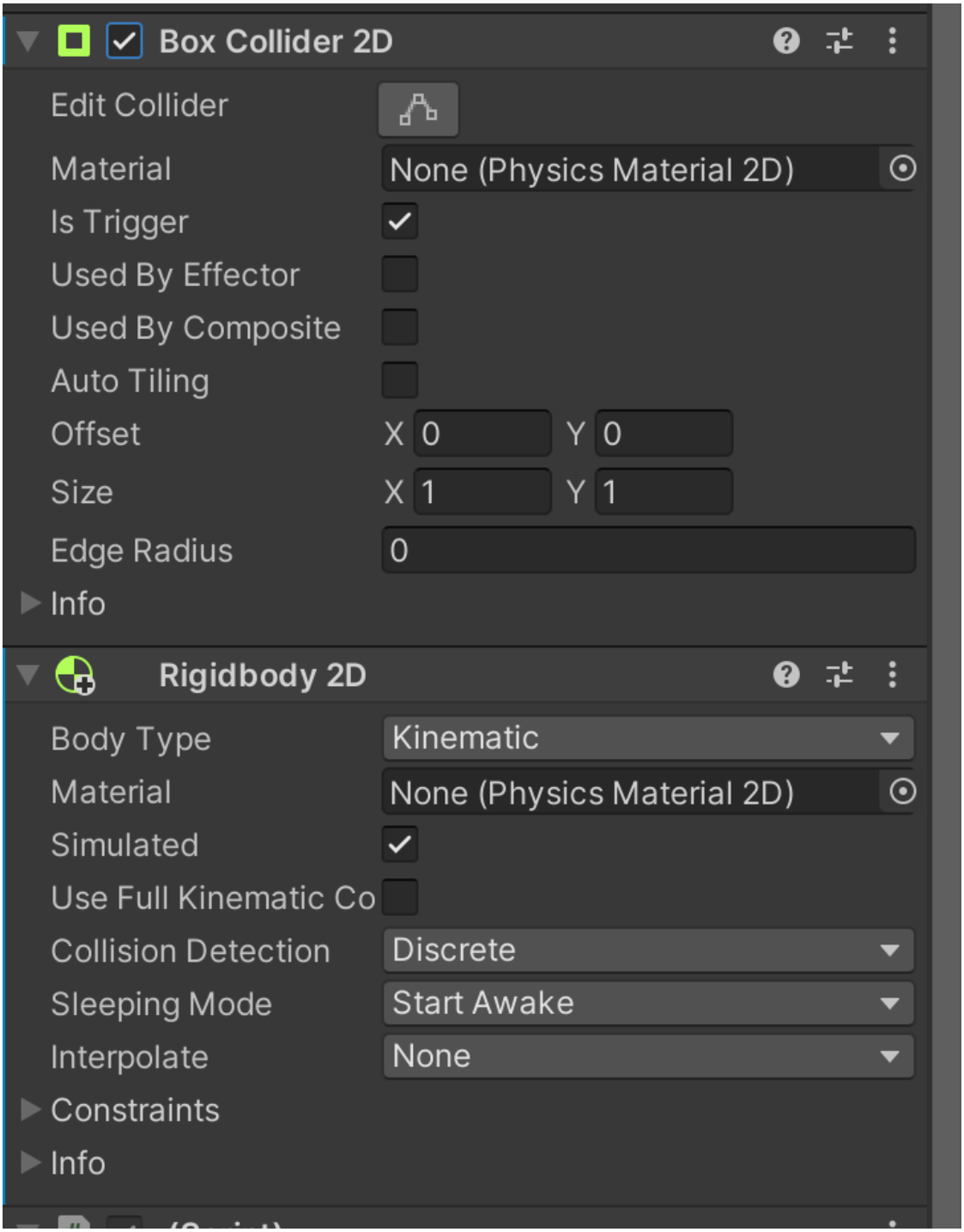
```
 void Update()
{
    if (Mathf.Abs(enemyBody.position.x - originalX) < maxOffset)
    {// move gomba
        MoveGomba();
    }
    else{
        // change direction
        moveRight *= -1;
        ComputeVelocity();
        MoveGomba();
    }
}
```

The idea:

- `If` Gomba isn't too far away from its starting position yet, move it to the designated direction
- `Else` , flip direction

# Collision with Enemy

Add a `Collider` component to the `Enemy` GameObject. Its inspector should now contain these components:




Now intuitively, we want our character to be "**damaged**" when it collides with the Enemy, but **not for the two bodies to push each other or simulate Physics.** The way to do this is to set the collider attached at the enemy's GameObject as [Trigger](https://docs.unity3d.com/540/Documentation/ScriptReference/Collider-isTrigger.html) (tick that `IsTrigger` option in `BoxCollider2D` element).

If a `Collider` collides with another `Collider` that is a `Trigger`, then "collision effect" will **not** be computed, and rather the callback `OnTriggerEnter` will be invoked (on **both** GameObject).

Implement the callback function `OnTriggerEnter2D` in `PlayerController.cs`:

> You can implement it in `EnemyController.cs` as well, and probably trigger an `Event` but for now let's choose the simple way first.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.CompareTag("Enemy"))
    {
        Debug.Log("Collided with Gomba!");
    }
}
```

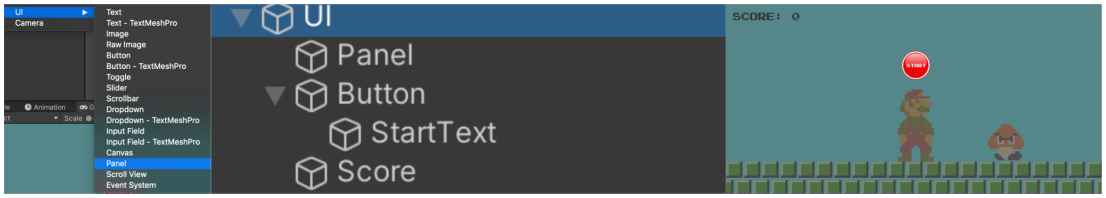
By now, you should see the message "*Collided with Gomba*" printed out in the console whenever Mario collides with Gomba.

# UI Elements

Of course any game should have some kind of "start" button and a scoring system. To have our game looks something like the screenshot below (right), we need 3 GameObjects:

- Panel
- Button
- Text (for Score)

Create following GameObject hierarchy as shown in the middle image and rename them accordingly. Right click at the hierarchy, then click **UI » Panel**, etc.




The `UI GameObject` is the parent object of all other UI GameObjects in this scene. We need to first determine which **coordinate system** to use.

- Set the `RenderMode` of its Canvas element as `Screen Space - Overlay` so that the coordinate system is mapped to the preview of the game.

> If you set it as `WorldSpace` then you need to use absolute coordinates to position the text and the button.

# Panel

The Panel serves as an “overlay” above all game objects in the scene. Click on it and see the Inspector. Edit its color and alpha to have a transparent layer over your game screen space.

# Score Text

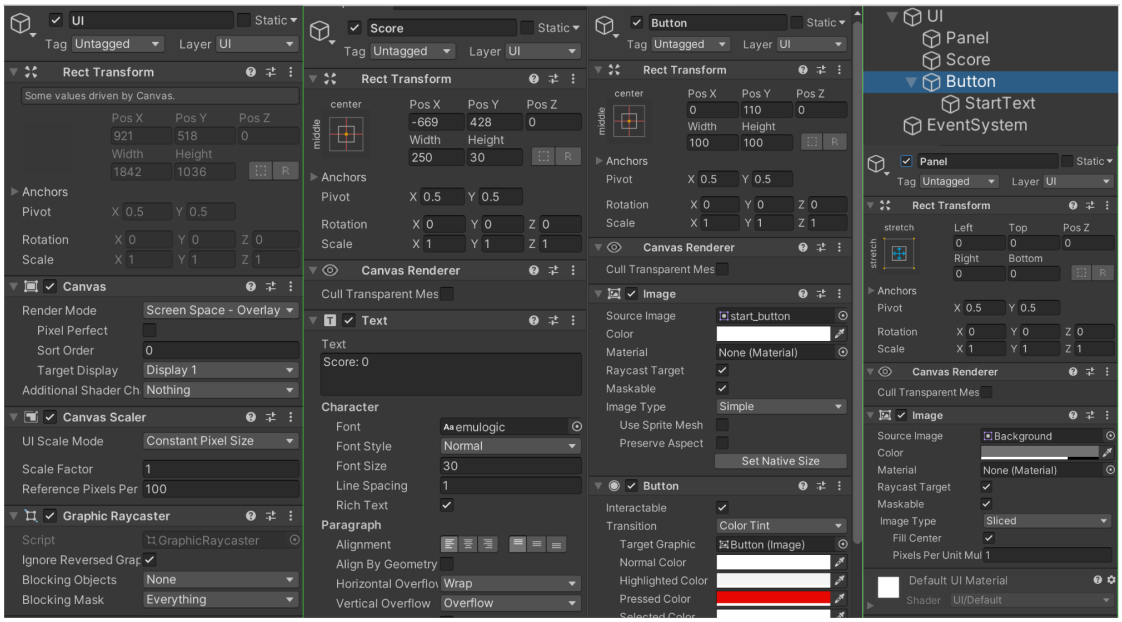
Then, play around with the `scoreText` setting:

- Change font size,
- PosX, PosY,
- Change font,
- Alignment, etc…
- *Horizontal Overflow*: Wrap is handy to prevent “missing” text when the font is too big for the text box.

# Button

For `Button`, you can download your own button image and change its text. Otherwise, use the default Unity button image. You can also **dictate how it should look like when pressed or remain selected**. Most importantly, you can choose the callback function when button is clicked. Also don't forget to modify the **text (child of Button)**.

Here's the inspector setting of all GameObjects mentioned above.




# UI Menu Logic: Button Callback

We want the entire overlay to disappear when the Start Game button is clicked. We need a function that will be called whenever the button is pressed.

Create a C# script named `MenuController.cs` and attach it to `UI` GameObject (that parent object). We want the game to not start at all before the button is pressed, so we need to do this in the `Awake()` function (recall Unity event functions):

```
void Awake()
{
    Time.timeScale = 0.0f;
}
```

Then, implement a **callback** function for the button called `StartButtonClicked()`. Here we iterate each children of **UI** and **disable** them so they're not rendered on the Scene anymore:

```
public void StartButtonClicked()
{
    foreach (Transform eachChild in transform)
    {
        if (eachChild.name != "Score")
        {
            Debug.Log("Child found. Name: " + eachChild.name);
            // disable them
            eachChild.gameObject.SetActive(false);
            Time.timeScale = 1.0f;
        }
    }
}
```

In the above, we basically set the `timescale` of the game to `0` in the beginning and set it to `1` *after button is pressed*. We also iterate through all of UI's children and *disable* all of them except the `scoreText`.

# Transform

The `transform` component of any GameObject is implemented as a **hierarchical** data structure.

- Using `transform` component we can traverse GameObject hierarchies in our scene quite conveniently,
- We can find a particular GameObject which we know is a *child* of some known GameObject reference.

**The reason why transform is implemented as a hierarchy is because when we transform any parent object, the change is reflected on ALL of its children.**

Each transformation from the parent is **propagated** to the children recursively, forming a transformation **stack** for each children (and all their children too) under

a gameObject.

Here's some quick info from Unity's official documentation:

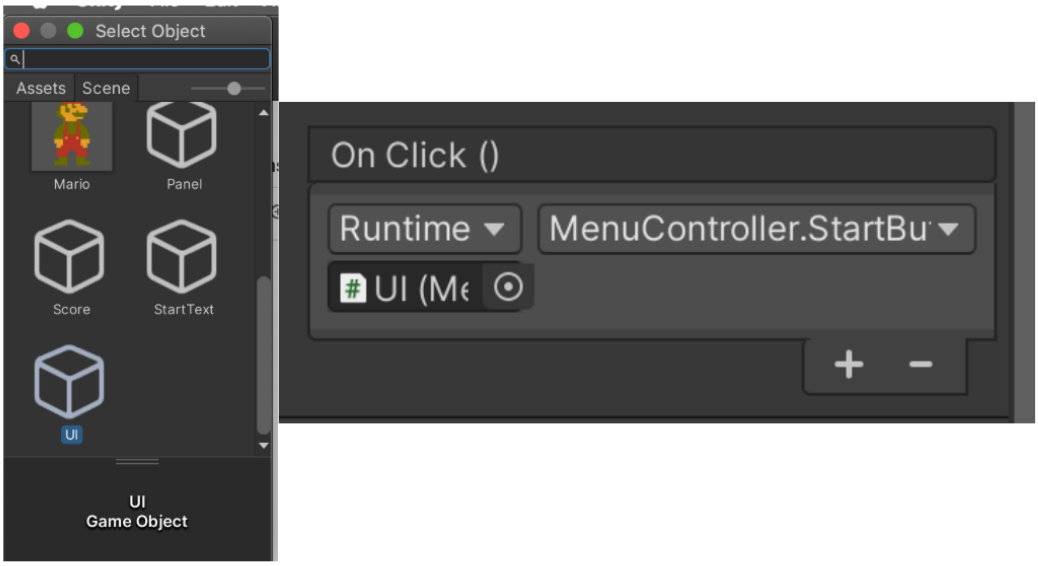
> Unity internally represents each transform hierarchy, i.e. a root and all it's deep children, with its own packed data structure. This data structure is resized when the number of transforms in it exceeds its capacity.

If you're interested to read up more about the mathematical details of **hierarchical transformation**, refer to this document.

# Final Touches for UI Logic

To stitch all the logic together:

- Attach `MenuController.cs` to **UI** GameObject.
- Then on **Button** GameObject, navigate the Inspector under Button element, and click the **circle** at the OnClick() callback.
- Find **UI** GameObject under the **Scene** tab (not Assets!)
- Afterwards, the function StartButtonClicked will appear in the dropdown menu.



> Remember to implement the callback function for the button as a `public` method else you won't see it in the dropdown.

# Scoring System

A game will not be complete without some kind of scoring or reward system. One way to "**count**" a score is to count how many time Mario has *successfully jumped over Gomba*. To do this, we need to know where Gomba is at all times, and of course the **reference** to the **scoreText** GameObject.

Add these variables in `PlayerController.cs` :

```
public Transform enemyLocation;
public Text scoreText;
private int score = 0;
private bool countScoreState = false;
```

Then in the `Update()` function of `PlayerController.cs`, add the following check:

```
// when jumping, and Gomba is near Mario and we haven't registered our score
 if (!onGroundState && countScoreState)
 {
     if (Mathf.Abs(transform.position.x - enemyLocation.position.x) < 0.5f)
     {
         countScoreState = false;
         score++;
         Debug.Log(score);
     }
 }
```

> We need that `countScoreState` to **not** increment the score *too many times*, but only once per jump because we know that Mario is unable to

> perform double jump.

Set `countScoreState` to be true when “space” key is pressed under the `FixedUpdate()` function:

```
    if (Input.GetKeyDown("space") && onGroundState)
    {
        marioBody.AddForce(Vector2.up * upSpeed, ForceMode2D.Impulse);
        onGroundState = false;
        countScoreState = true; //check if Gomba is underneath
    }
```

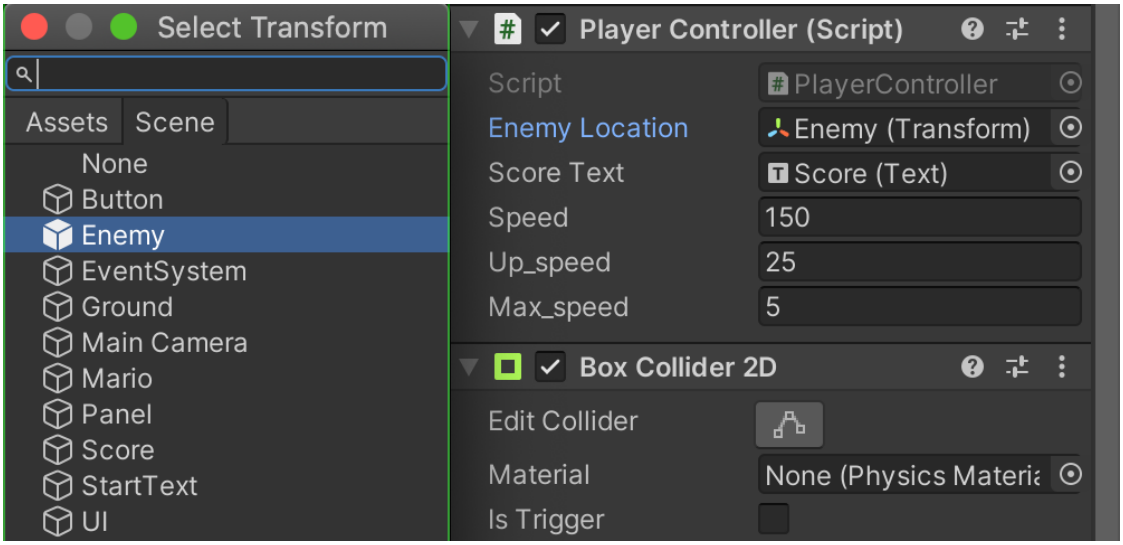
Finally, we need to check when Mario lands on the ground. We can do this by checking collision with the **Ground**:

```
void OnCollisionEnter2D(Collision2D col)
{
    if (col.gameObject.CompareTag("Ground"))
    {
        onGroundState = true; // back on ground
        countScoreState = false; // reset score state
        scoreText.text = "Score: " + score.ToString();
    };
}
```

> Note: we use Tag here to easily find GameObjects on the scene. Surely there’s fancier ways out there, using Inherintance and whatnot. However during development stage and considering that our peers are mostly beginner, let’s just use the most convenient tools available. Your gaming computers shall not have any performance problems for this simple small game.

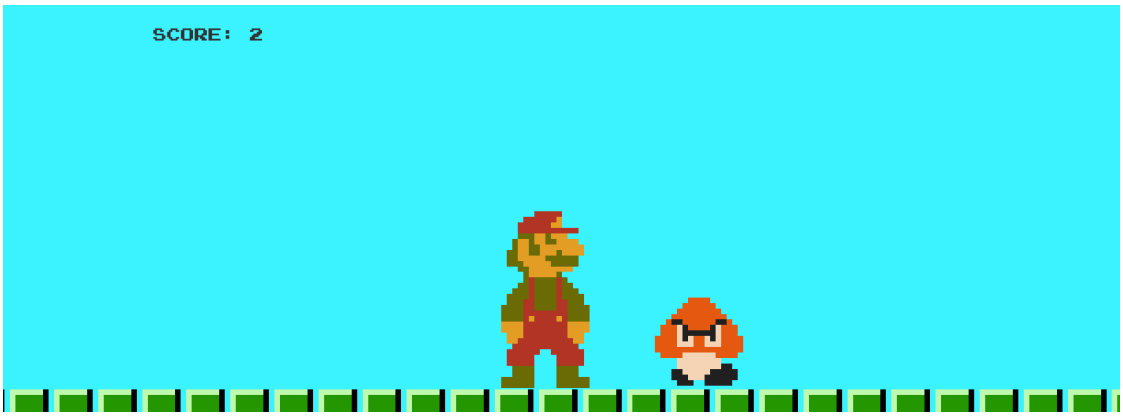
Finally, we can conveniently **link up scoreText** and **Enemy Transform** object references in **Mario’s** inspector. Click the circle and select the appropriate

GameObjects from the Scene tab.




You can adjust `speed`, `upSpeed`, and `maxSpeed`, as well as **Mario's** `Mass` and `GravityScale` to make the movement feels natural.

- The scoreText should increase whenever Mario successfully jumps over Gomba and
- The game shall stop abruptly when Mario collides with Gomba.

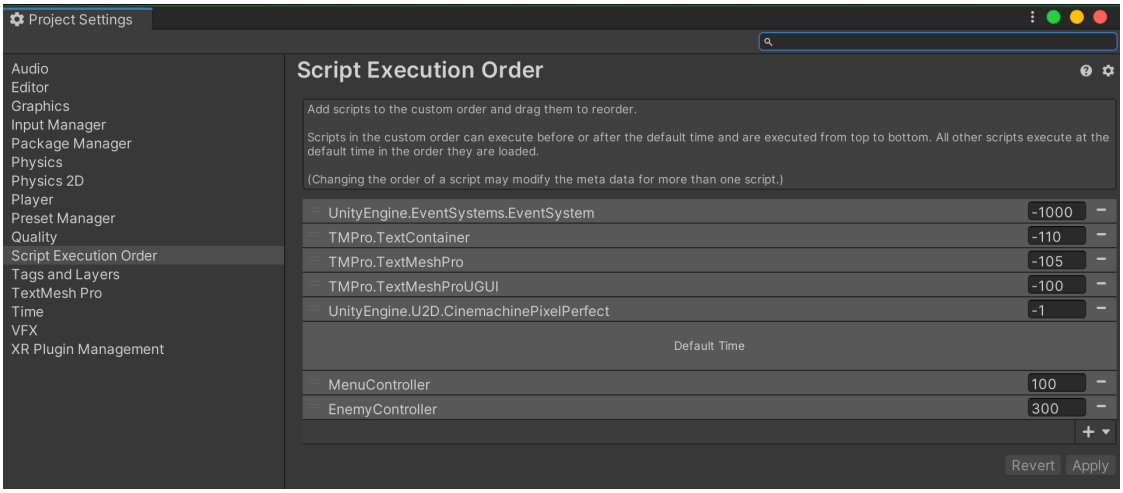



# Script Execution Order

We can tell Unity which scripts to execute first, that is Unity will call the `Awake()` functions it needs to invoke in the order that you want, and then repeatedly call `Update()` in the same order.

Go to Edit » Project Settings then select the Script Execution Order category. You may choose to add any script you want and define its order of execution (higher number means it will be run later, you can use any positive integer. Unity will only care about its relative value).

In the screenshot below, we want the menu script to be run first before the enemy's script.




# Checkoff

Review our lab recording (the playlist for all recordings can be found in our course handout) to know the final state of the project before proceeding to implement the checkoff. The gif below also summarises the desired end state for this lab:


Once you've implemented everything in this handout, then for **checkoff** you're required to implement a **restart** mechanism when game is ***over***. You need to reset everything: score, player position, etc.

*You're free to implement it in any way. It will not affect your checkoff score.* The grading for this lab is **binary** (completed / not completed).

***Read the course handout to find out the checkoff procedure for each lab.***

# Next

It's been hours but we are nowhere near a completed game (unless of course you have prior experience with Unity):

- No **sound effect** or **animation** (lack of visual feedback)
- No **platforms** implemented yet (it's a platformer game!)
- There's **no game manager** of any sort, and `score` is sloppily stored in `PlayerController.cs`
- There's no *centralised* way for keeping track of **states** (score, player state, etc)

- The “Enemy” is kinda predictable or boring, and Mario’s scoring system doesn’t work that way.
- …etc

We will try to improve our game and learn some common C# coding practices in the next few parts.

---

PREVIOUS POST
Unity for Babies