2D Unity User Guide

About

This it a 2D Unity user guide that acts as a tutorial in making a 2D platformer. I will go through 2D games design along with general tips and tricks for solo and team-based projects. Without further ado, let's begin!

Index

1. Installation

Before installing, note that Unity is **completely free**, if you or your company makes less than $100 000 (USD).

Follow the link here and download the free installer (the plus version is definitely not necessary). Follow the instructions and download the installer. The Unity installer is simple and easy to use nevertheless, this video for Windows is a tutorial in installing it. Also note that you will may need a Unity account or just use your Google/Facebook account to sign in.

If you want a new version of Unity, go to the Unity Hub and click the **Installs** tab on the left and **ADD** (top right) the desired version (it is recommended that you pick a version with **long term support** or LTS); however do be warned that this tutorial was made in **2019.3** so in newer versions, things might have moved places or removed entirely 😣. Afterwards, the installation will take quite a while, even with a fast internet connection.

When installing Unity, it might ask you to install **Visual Studio** along-side Unity. If you have Visual Studio, do not install again, it should automatically detect the current VS on the system. Otherwise, it is optional and MonoDevelop (comes with Unity) is good enough for handling coding in Unity.

2. General

Unity is a 3D game engine built in **C#** but can be used for 2D. Before you freak out about programming, let me assure, you it is quite trivial. If you have programmed before especially in C++, it is a huge plus. If you have not, don't worry, it's just a bunch of copy and pasting. The most important thing is don't be frustrated and

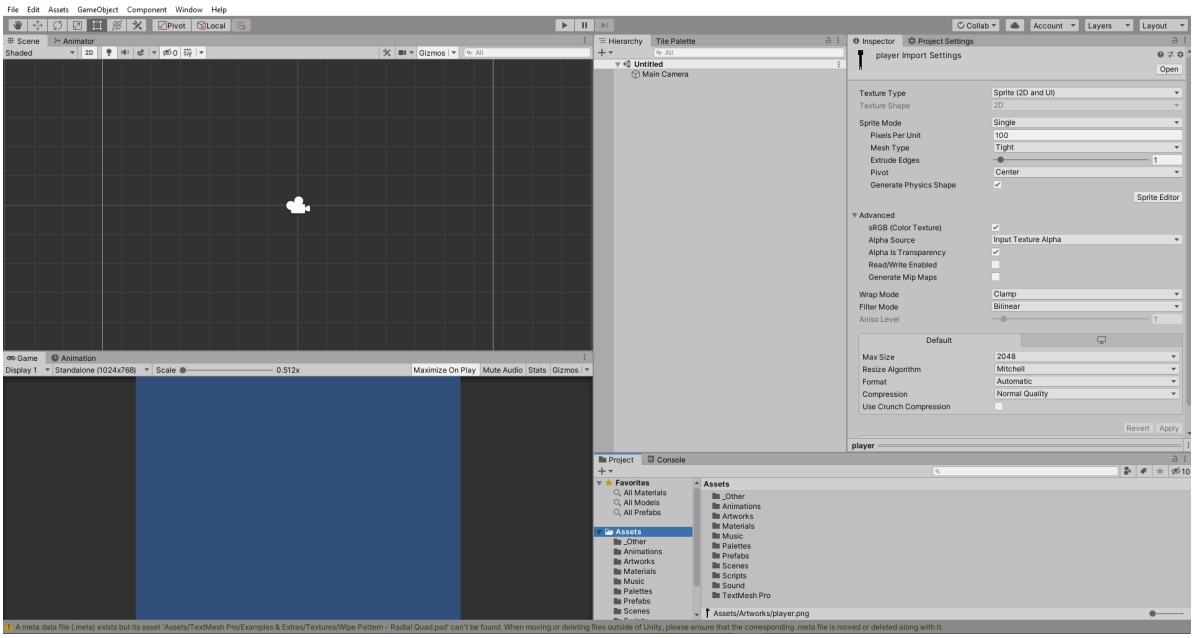
```
Debug.Log() // logs to Unity console
```

is your friend.

Being organized in Unity is a must, especially for big projects. Have organized folder will help a ton, in addition, using empties as folders within scene is *+1 organization*.

3. Unity Editor Overview

Unity is a panel based application. These panels can be rearranged however you like and saved as default.



This is my default setup. The **Scene** panel is the most important viewport. This is where you modify the scene. The **Game** scene only turns on if you hit play. It is a preview of what the user will see if you build the current scene.

On the right hand side, we have the **Project** directory (folder) panel on the bottom. This is where all your assets are (ie art, music, scripts). The **Hierarchy** is like a layers panel in Photoshop. It is an arrangement of all elements in the scene. For every element of the scene can be hidden from the view by hitting eye button when hovering on the element. The lock button is the pointer icon. The **Inspector** contains all modifiable aspect of an element along with info. This is where you drag scripts in the element.

Other panels include a console, which will output when the game is run. The **Animator** and **Animation** go hand in hand and creates animation for sprites.

2D

4. General

So you've decide to make a 2D game. Great! Who needs modelling and lighting anyways?

2D games are secretly 3D, what does that mean? Think of Unity 2D as a bunch of layers like in Photoshop, Gimp, After Effects, or Illustrator. The closest to the camera is picked up first and blocks the ones behind it. As a 2D world, lighting is global (unless you want enable an experimental local lighting feature). To start a new 2D game, click **New** and select 2D on the pop-up screen and use the desired directory (or follow my cloning guide so to not start from starch). This may take some time, but after Unity finishes installing itself, we can get started. Now let's make a player and make it move!

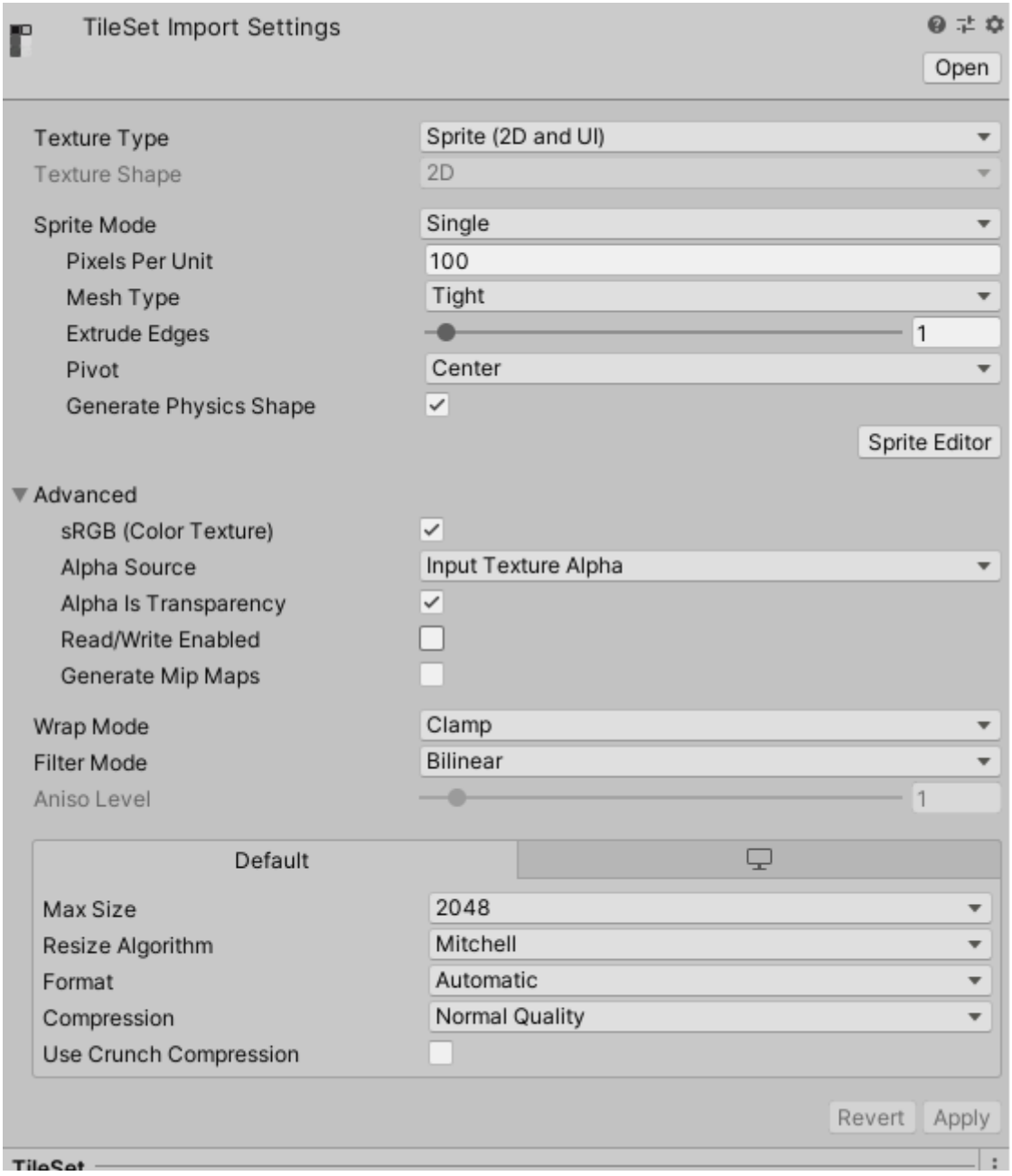
*Note, part of the tutorial (2D) follows this repo: Unity-Tutorial-2D

5. Sprites

Sprites are what make up the visuals of the game. These could be png or jpeg (recommended because of small size) file. You can make your own and drag them into an **Artworks folder**.

If you have multiple sprites on a picture file, you have what is called a **sprite sheet**. These could be useful because it saves space. Unity comes with a sprite editor that can cut the sprite sheet into multiple sprites.

However, before we get to that, let us take a look at the Sprite setting



From the top header, we can see some info on the item

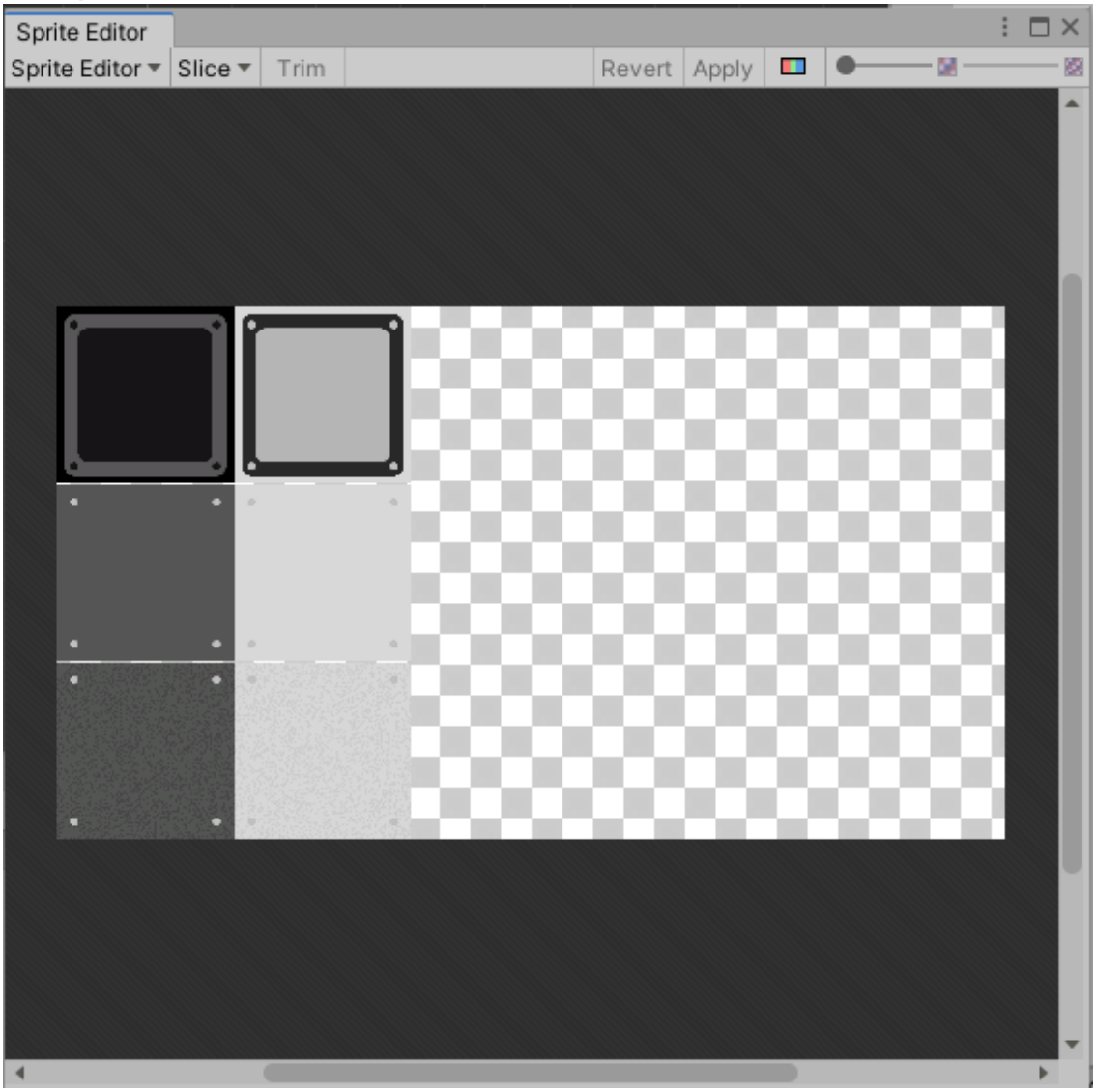
From the header we also see:

- Texture Type - Usually will only use sprite (2D image) or normal map (for 3D depth)
- Sprite mode - as single texture or sprite sheet
    - Adjust the pixels per unit (Pixels per unit should be 1-1) (In my case, it is 358 px per tile)
    - Pivot - where the center of object is
- Wrap Mode - how the image is displayed (ie repeated or clamped(/cut) at the board)
- Filter Mode - Point (equivalent to pixel perfect or nearest neighbour in Photoshop) or Bilinear (natural scaling with edge softening)
- Format - Format of image (8bit, 16 bit, 32 bit colour with or without alpha)

* Note, if a setting was not mentioned, it is not that important at the beginner level.

Multiple Sprites

1. Select Multiple in **Sprite Mode**
2. Click **Spirit Editor** button



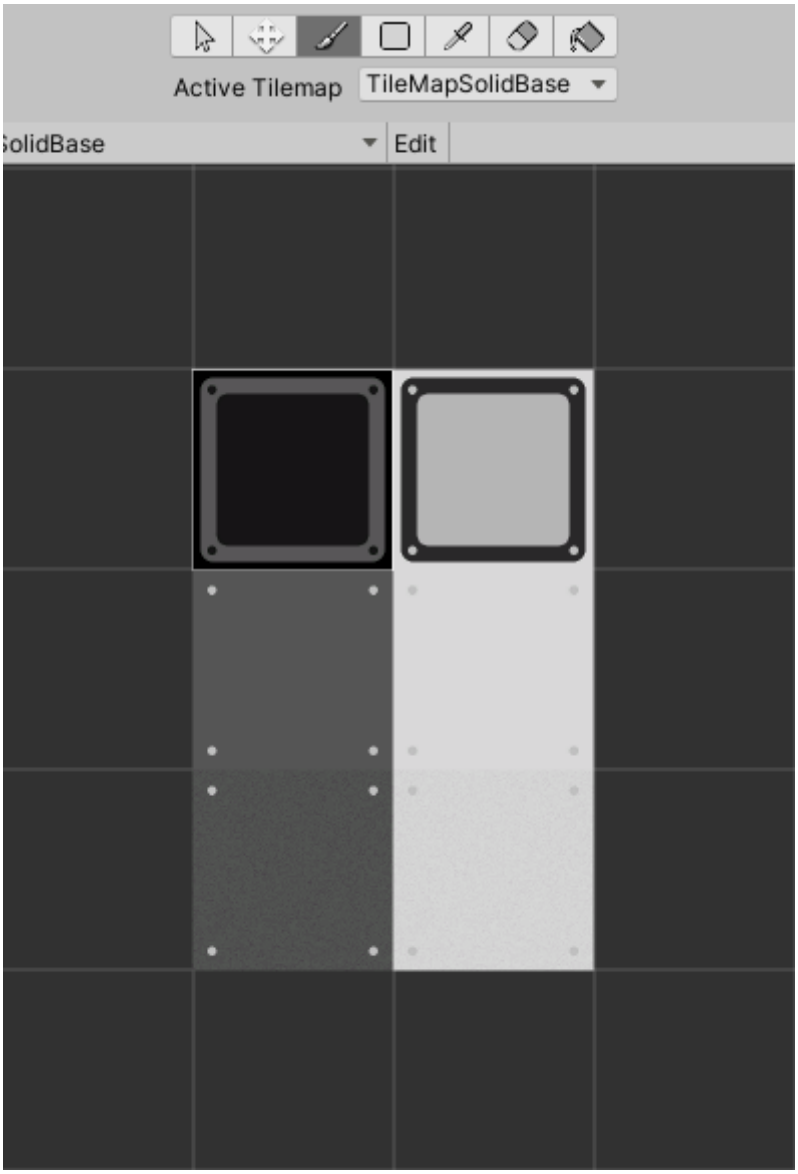
3. In sprite editor, click Slice -> Slice (This is an automatic slicer and is usually good enough, if not, manually move the transform bound boxes to fit or define split at distance apart (must change to **Grid by Size Count**, not **Automatic**) with the size of tile in **Pixel Size**.
4. Click apply
5. Close window or drag to some widget
6. You can inspect every tile by expanding the original image and a bunch of tiles with name pictureName_0 to # of tiles with be a child of it

Now, with a sprite, you can drag it into the scene

To create a tile map, follow the steps below:

1. GameObject -> 2D Object -> Tile Map
2. Open Tile Palette, by going to Window -> 2D -> Tile Palette
3. Create New Palette and save it
4. Drag all individual tiles into widget or drag parent spritesheet (and save)

Now with the tile palette, you can draw on the scene, reorganize everything and many other things



From the top, there are many icons, we will go through each of them,

- Curser - selects tiles from scene
- Move - Moves tiles in palette (Only if you click **Edit**)
- Brush - paints on scene
- Square/Rect - Selects multiple tiles or one to paint from
- Eye dropper - Selects tile from scene
- Eraser - Erases
- Paint Bucket - Floods with active brush (Like Photoshop)

To layer tiles:

- Right click on the tile map -> 2D -> Tilemap for however many more layers you need

To set a layer as a solid with collision:

1. go to the layer you want to make solid
2. Add Component -> search add Tilemap Collider 2D

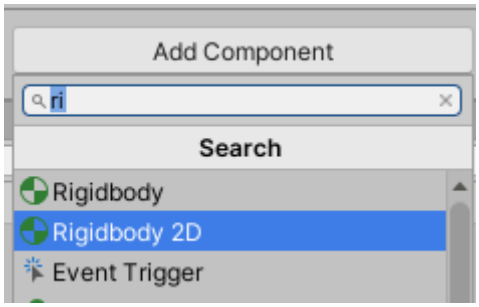
Resources: Brackeys

6. Movement

Movement is critical in all games, whether the movement is limited to left or right, or games that in 3D. On the internet there are may sources that claim the perfect jump, but only you can decide that based on what type of gameplay you want.

To start off with, set your player to be a rigid body with colliders:

1. Select the player and add a Rigidbody 2D



2. Open up the Rigidbody 2D and go to constraints. If you want the player to not rotate in 2D, then select **Freeze Rotation Z**
3. Otherwise, add a **Collider 2D** to fit the player. Depending on how your player's form, different shapes may be better. Try the different collider options to see which ones fit best. Most of the time, a **Box Collider 2D** will work just fine. Also note that you can use multiple colliders, but remember, this is computationally more expensive.
4. Also note that if you are not satisfied with Unity's default collider size, you can change the size by changing the **size** attribute

Now for the movement:

1. make a new folder called **scripts**
2. right click -> Create -> C# Script -> call it **movement** (although you can pick any name you want, this is standard)

Next, you want to open the script by double clicking on it

This will either open the script in MonoDevelop or Visual Studio (to find out more about scripting, go to the scripting section)

In the movement class, put

```
// inits
// hoziontal speed
public float speed = 10f;

// vertical jump
public float jumpForce = 10f;

// different jump heights for how long jump button is pressed
// change it as you change gravity
public float jumpCheck = .2f;
// is ground check
public float checkRadius = 0.3f;

// ground jump check
public bool isGrounded;
public Transform feetPos;
public LayerMask whatIsGround;

// so that infin jumps are not a thing
```

```
private float jumpCheckCounter;
private bool jumping;
private float moveInput;

// inits rigid body
private Rigidbody2D rb;

void Start() {
    rb = GetComponent<Rigidbody2D>();
}

void Update() {

    // bool to see if overlap of floor layer and feet object
    isGrounded = Physics2D.OverlapCircle(feetPos.position, checkRadius,
whatIsGround);

    // if is touching ground and jump button pressed, add jump force
    if (Input.GetButtonDown("Jump") && isGrounded == true) {
        jumping = true;
        jumpCheckCounter = jumpCheck;
        rb.velocity = Vector2.up * jumpForce;
    }

    if (Input.GetButton("Jump") && jumping == true) {
        // can jump only once
        if (jumpCheckCounter > 0) {
            Debug.Log(jumpCheckCounter);
            rb.velocity = Vector2.up * jumpForce;
            jumpCheckCounter -= Time.deltaTime;
        }

    }
    else {
        // once jumping is not true, set jump to false
        jumping = false;
    }

    if (Input.GetButtonUp("Jump")) {
        jumping = false;
    }

    // flips player
    if (moveInput > 0) {
        transform.eulerAngles = new Vector3(0, 0, 0);
    }
    else if (moveInput < 0) {
        transform.eulerAngles = new Vector3(0, 180, 0);
    }
}
void FixedUpdate() {
    // gets horizontal input from Unity (1 = right, -1 = left)
    moveInput = Input.GetAxisRaw("Horizontal");
    rb.velocity = new Vector2(moveInput * speed, rb.velocity.y);
}
```

Save and exit. Now, when you click on the player, movement script, there will be options for you to tinker with.

Finally for jumping you need to:

1. Set ground layer to ground objects that can be jumped on
2. Make a isGrounded empty object to be placed at the feet of the player to check if there is **ground layer** beneath the player to jump off of

* Note that this script has a jump that can be a multitude of heights depending on how long the jump button is held down for.

Also, for the player to not stick to the walls while jumping, you must:

1. Add a new physics material 2D
2. Change friction to 0 and bounciness to whatever you want it to be
3. Apply to player in the collider

Resource: https://www.youtube.com/watch?v=j111eKN8sJw

7. Camera

The camera will capture things on from the scene to project onto the play window.

The most important setting for a camera object is the size. Changing the size will change the view for the user.

Background

A background can be achieved by placed in the camera and setting the **Order in Layer** to some negative number, such that it is behind the foreground objects

Camera Follow

When implementing a camera, there are two options, however, we will through the harder option first to get use to how cameras work

1. We will create a camera follow script with the following:

```
// selects targe to be fixed on
public Transform target;
public float smoothing = 0.12f;

// because it is fixed to player, we want to move in from by 10 layers be
default
public Vector3 offset = new Vector3(0f,0f,-10f);

void FixedUpdate() {
    Vector3 desiredP = target.position + offset;

    // interpolate movement
    Vector3 smoothP = Vector3.Lerp(transform.position, desiredP, smoothing);
    transform.position = smoothP;

    // will move in direction of target
    // comment out if you don't like the jitters
    transform.LookAt(target);
```

```
}
```

You will have to drag the player to target and change the smoothing value if you like more or less smoothing

OR

2. Use the Cinemachine Extension

To use Cinemachine, one must first install the extension as it does not come in with Unity natively. To do this,

1. window -> Package Manager -> **Search:** Cinemachine -> click install

Now to use it:

1. Cinemachine -> Create 2D Camera
2. Embed CM vcam into main camera (optional)
3. Go to CM vcam and put the player in the **Follow**
4. Try it out
5. If you want the character directly in the center of the camera, turn down all **damping** and **dead zone** to 0 (these settings are in **Body**)
6. **Screen** x and y, offsets from center
7. **Soft zone** changes the max distance from center before snapping the player to edge
8. **Dead zone** is how far you can move the character before the camera moves along-side the player
9. **Damping** is the smoothness of the camera movements
10. **Look ahead** is the direction the player is moving towards
11. Play around with these setting

Resource: https://www.youtube.com/watch?v=MFQhpwc6cKE or https://www.youtube.com/watch?v=2jTY11Am0Ig

8. Prefab

A prefab is simply a clone that can be dragged to the scene

This means that things from other scenes can be reused

To make a prefab, just drag the desired prefab object from scene to the prefabs folder

9. Particle System

A particle system adds an extra layer of immersion that is quite simple in Unity.

To add a particle system:

1. Right click hierarchy -> Effects -> Particle System
2. Remember, default is some circled with soft edges
3. To make custom particles,
    1. Go to photo editor of chose (Adobe Photoshop, illustrator, or GIMP work great)
    2. Make your single particle or collection of particles

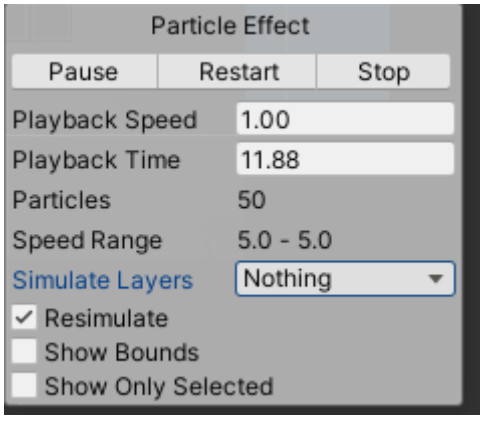
3. Save
4. To use the custom particles, go to renderer -> material -> sprite default
5. Now enable **Texture Sheet Animation**
6. Mode -> ~~Grid~~ -> sprites
7. Select objects

There are near infinite options in the inspector for particle systems. Most options are self explanatory. Some of the most common options to change are:

- Loop - for ambient particles that need to on screen all the time
- Start Color Options
    - Random Color -> randomly selects a color
    - Gradient -> flows from one color to the next
    - Random between two colors/gradients -> randomly picks a color in gradient or one of the two colors
- Prewarm - already has some particles and does not start with no particles
- Start Speed - Speed of particles
- Start Lifetime - how long the particle lives in the scene
- Start Size - size of particle
- Emission -> Rate over Time - how many particles spawn over time
- Rotation over Lifetime -> rotates over time
- Collision -> collides with solid objects
- Trail -> creates trails of particle as it moves
- Color over Lifetime -> you can fade the particles in and out
- Shape -> Randomize Direction - changes starting velocity vector
- Shape -> Spherize Direction - Moves out from center

* Note for some options like start speed, lifetime, size, etc, you can pick from an interval

If there is some setting I haven't mentioned but look interesting, refer to the Unity docs. To test particles, use the Particle Effects window. This will control preview



10. Collectibles

There are two parts to a collectible:

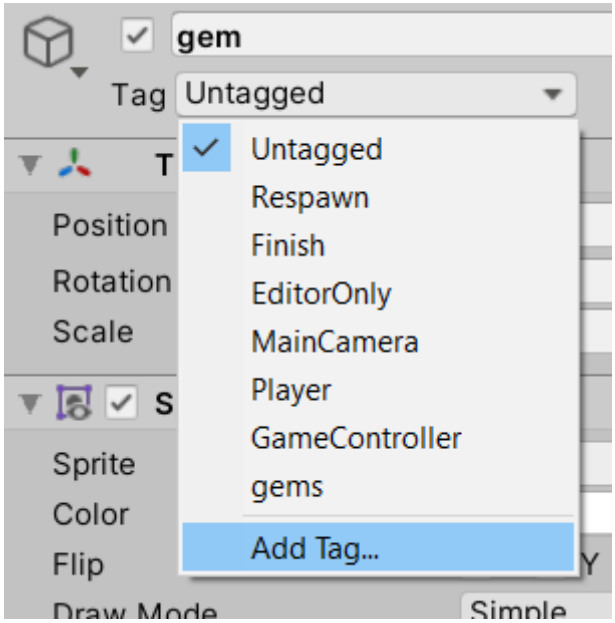
1. The collection of the item

2. The storage of the item in inventory

We will start with 1. the collection of an item

1. Find a sprite or animation to use as collectable. For simplicity, I will use a sprite like this:

2. Drag gem into level
3. Add collider (I think box collider works best here) and check **Is Trigger**
4. Add a new tag and select it of the gem



5. Within the player movement, we need to detect the collision overlaps and if an element of the tag overlaps the player, destroy object

   This is done by putting the following code somewhere in the player movement class (not in an update or start method)

```
private void OnTriggerEnter2D(Collider2D collision) {
    if (collision.gameObject.CompareTag("gems")) {
        Destroy(collision.gameObject);
    }
}
```

6. (Optional) If you want to reuse the gem, drag in Prefab folder

Now we must consider the storage of the item in inventory. Note this can be as easy as a counter or a full fledged inventory

I will do a counter for simplicity:

* Note while we are adding a counter, it is also a convenient time to add a sound of collecting the item.

1. Resources for audio clip: here & here but I used PlayOneShot
2. Optionally, a collecting animation or particles could also be implemented

3. However, we will jump directly to a collectables counter
    1. Create a text layer by clicking UI -> Text - TextMeshPro
    2. Click on canvas -> Render Mode -> Screen Space - Camera
    3. Assign Render Camera to current Main camera
    4. Change layering order on canvas so it is in front
    5. Position it right. You may also chose to have a image accompanying the counter
    6. Create a collectablesManager script with the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class collectablesManager : MonoBehaviour
{
    public static collectablesManager instance;
    public TextMeshProUGUI text;
    public int score = 0;
    // Start is called before the first frame update
    void Start()
    {
        if (instance==null) {
            instance = this;
        }
    }

    // Update is called once per frame
    public void changeScore(int gemValue) {
        score += gemValue;
        text.text = "x" + score.ToString();
    }
}
```

    7. Create empty to house the collectablesManager script and drag the script in
    8. Drag the text score into the Text field
    9. Now gems script and populate with the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class gems : MonoBehaviour
{
    // Start is called before the first frame update
    public int value = 1;
    private void OnTriggerEnter2D(Collider2D collision) {
        if (collision.gameObject.CompareTag("Player")) {
            collectablesManager.instance.changeScore(value);
        }
    }
}
```

*Note by using this, you must have one collider with the player tag or the collectable might count twice. Optionally, you can make a new empty game object with the player tag than triggers the collection

Resource: https://www.youtube.com/watch?v=DZ-3g31jk90

11. Animation

Animation is critical in creating any game that does not look static. In Unity, animation is handled by Animator and Animation windows. The **Animator** is a node based system while the **Animations** is a timeline based.

To make an animation, simply:

1. Drag more than 1 picture into the scene
2. Save your animations in the animations folder

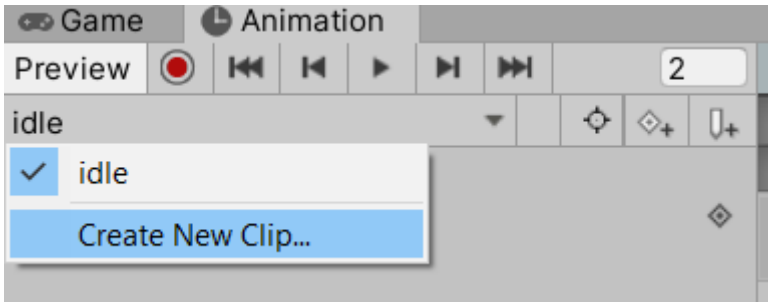
You have made your first animation

To adjust your animation, go to the animator window and clip on the orange node.

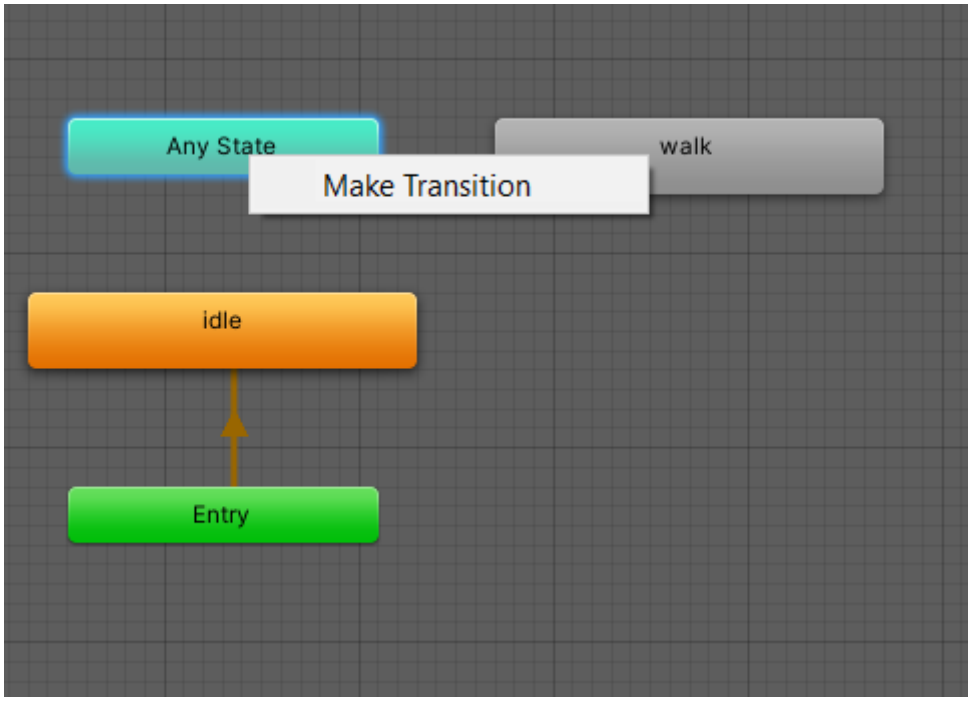
You can change the speed here

To add different actions connected to the original animation:

1. Go to the Animations window when the original animation is selected
2. Click on animation dropdown -> Create New Clip...



3. Save the new animation
4. It is recommended that you change the sample frame to the original animation's sample frame
5. Drag all the new set of pictures to the timeline in the **Animations** window
6. To change the pictures per second, drag the blue bar (all frames must be selected ie in blue)
7. Now to move from any one animation (or state as it is called), go to the **Animator** window
8. Right click on the **Any State** node -> Make Transition



9. We will link the idle animation to the other transitions, therefore, after selecting make transition, click on idle and this will connect any state to idle
10. Also make transition to walk (or any other animation) from any state
11. To link these states up, go to **Parameters** tab in the Animator window and click **+** -> int
12. Name it
13. Select transition Any State to idle
14. in the inspector, expand Settings and
    1. Uncheck **Fixed Duration** if you want no transition duration
    2. **Change Transition** Duration to 0
    3. Uncheck can transition to self
    4. In Conditions, click **+**
    5. Set it equal to 0
    6. Do the same thing for **Any State** to any other animation to the same parameter except change step 5 to equal to some other number
15. Now to connect the animations with the code so it changes with the actions performed by the player
16. Go to the player movement script
17. Add the following:

```
private Animator animator; //attribute in class

// add the following in Start
void Start() {
    animator = GetComponent<Animator>();
}

// add the following in FixedUpdate
void FixedUpdate() {
    // if moving, then idle, else set to moving animation
    if (moveInput != 0) {
        animator.SetInteger("AnimValue", 1);
    } else {
        animator.SetInteger("AnimValue", 0);
```

```
    }
}
```

* Note if you don't want the animation to loop, go to the animation in where you saved your animation and uncheck **Loop Time**

12. Events

Events are add much needed interactions into a game, but keep in mind that every event requires a lot of work. With this in mind, let's implement a button that triggers a platform.

1. Have a button and platform object ready. If not, feel free to use the ones below

2. If you want, drag the two buttons as an animation so you can change the sprites when touched, but I will not go through the animations
3. Add a button tag onto the button
4. Create a button script and paste the following into the script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class button : MonoBehaviour
{
    // renderer and timer
    public GameObject somePlatform;

    // change button state
    private Animator animator;

    // audio
    public AudioSource audioSource;
    public AudioClip clip;
    public float volume = 0.2f;

    // timer for how long the platform stays
    private bool timerOn = false;
    public float timeLeft = 8.0f;

    private void Start() {
        animator = GetComponent<Animator>();
    }

    private void FixedUpdate() {
        // when pressed, wait for timer to go down, then revert
        if (timerOn) {
            timeLeft -= Time.deltaTime;
            if (timeLeft < 0) {
                animator.SetInteger("button", 0);
                somePlatform.SetActive(false);
                timerOn = false;
                timeLeft = 8.0f;
```

```
            }
        }
    }

    // Start is called before the first frame update
    private void OnTriggerEnter2D(Collider2D collision) {
        // when collision with obj with player tag, set button to pressed
down,
        // play tone, set platform active and activate timer to turn the
things back to normal
        if (collision.gameObject.CompareTag("Player")) {
            animator.SetInteger("button", 1);
            audioSource.PlayOneShot(clip, volume);
            somePlatform.SetActive(true);
            timerOn = true;
            timeLeft = 8.0f;
        }
    }
}
```

5. Click on script, drag audio stuff or remove all audio parts from script and set the object to **Some Platform**, also set the timer for when it reverts back to normal (auto is 8 seconds)

As you can see, using what we already know, we can get a lot of events to work! I know this may be challenging and you may run into bugs (hell, even I ran in many bugs), but persistent and challenge yourself and good luck!

13. Pixelated

Perhaps you want your game to have a pixelized feel, thankfully, it is not too much work to do that with Unity.

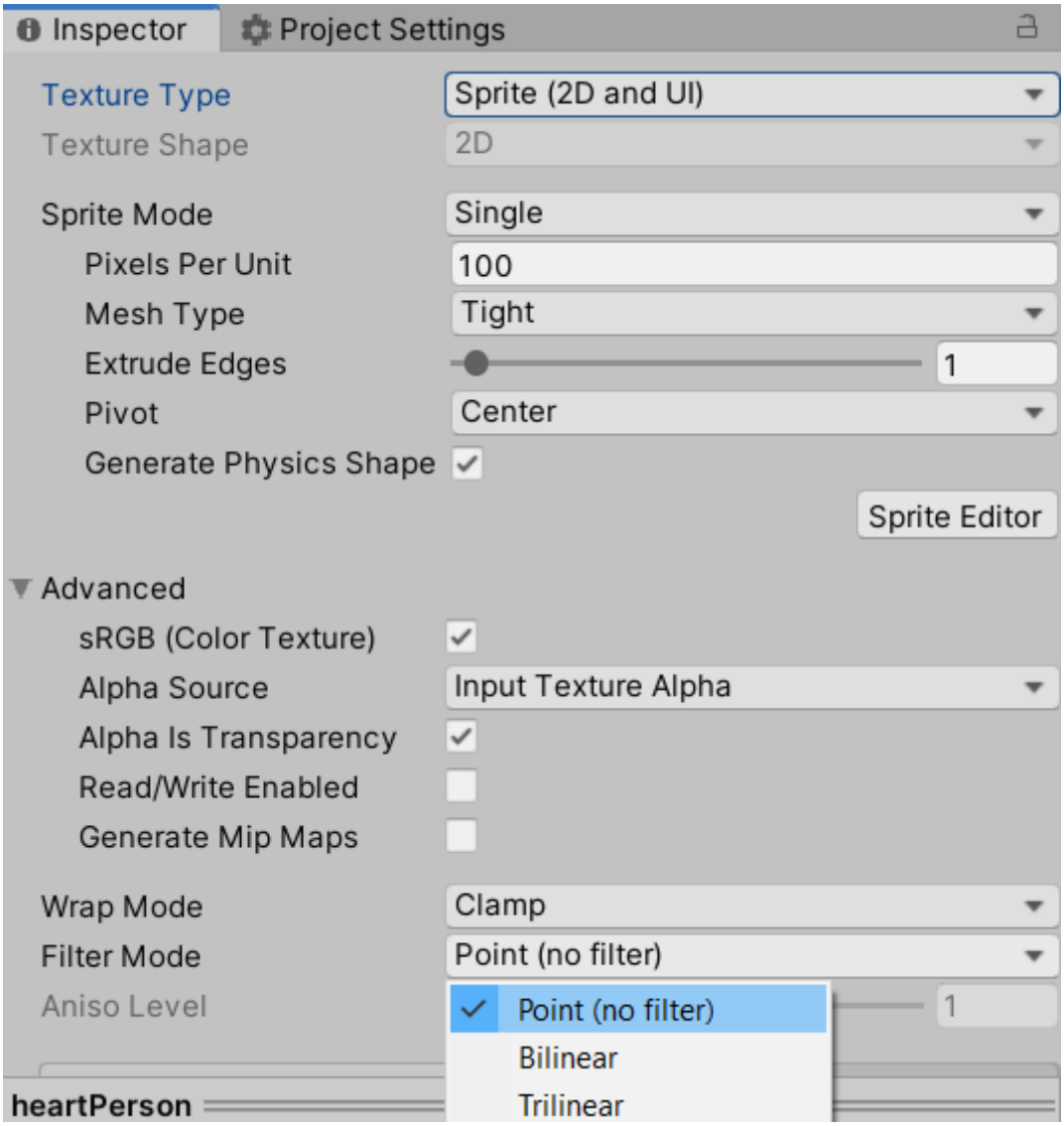
General rules and tips

Some extensions (especially with cameras) have options to enable **pixel perfect**. Whenever presented with that option, select it.

Otherwise when importing your pixel artwork or spritesheet;

* If you need pixel art to practice this section, take this:

Click on Filter Mode -> Point (no filter)



The point mode is like a nearest neighbour in Photoshop. This way, if you scale it, you will get the pixels to snap sharply.

You may also change the pixels per unit to your desired size.

14. Music & Sounds

Music is curial to a game's level of emersion. While I will not go into music theory and music in general, I will help you get familiar with how Unity handles sounds. This is something Unity does really well, so not much can be said in the chapter.

If you need music to practice with, use main.wav in the **test_music** directory. Also note that this music is made by me and I give you permission to use it under creative commons license

First, when dealing with sound, not much should be changed and when playing a sound, in the script that triggers the sound, just put the following in

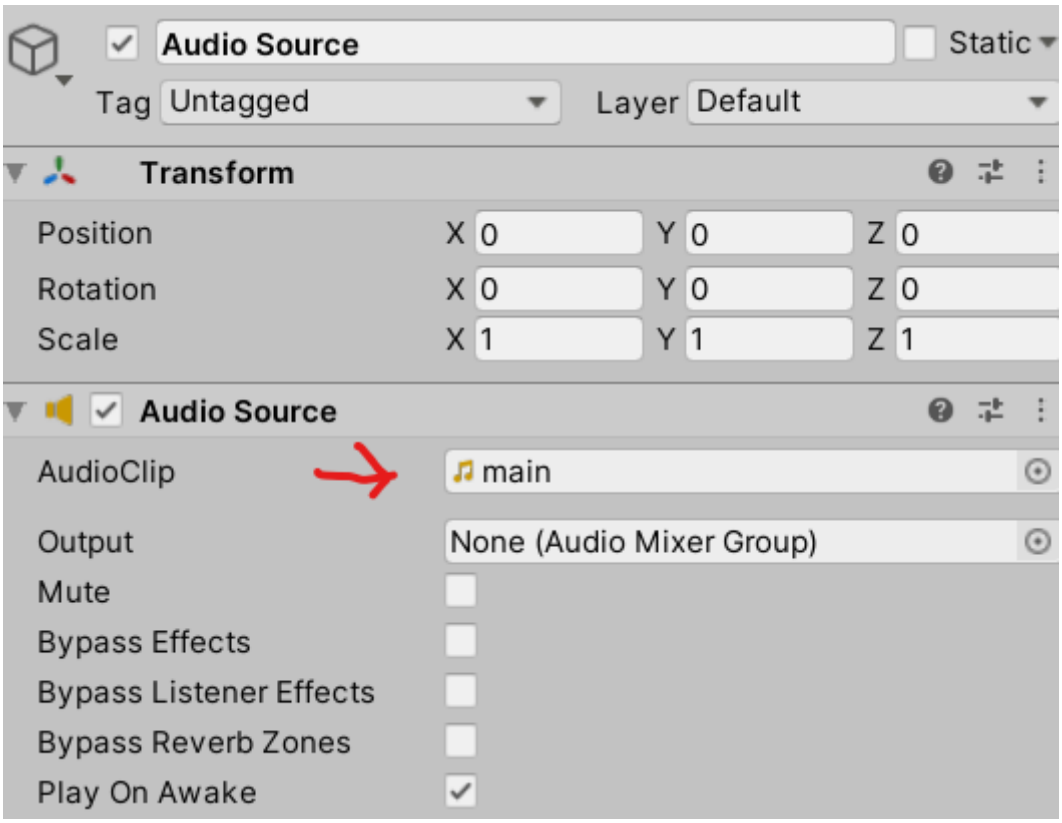
```
// audio
public AudioSource audioSource;
public AudioClip clip;
public float volume = 0.2f; // volume

// to play sound
audioSource.PlayOneShot(clip, volume);
```

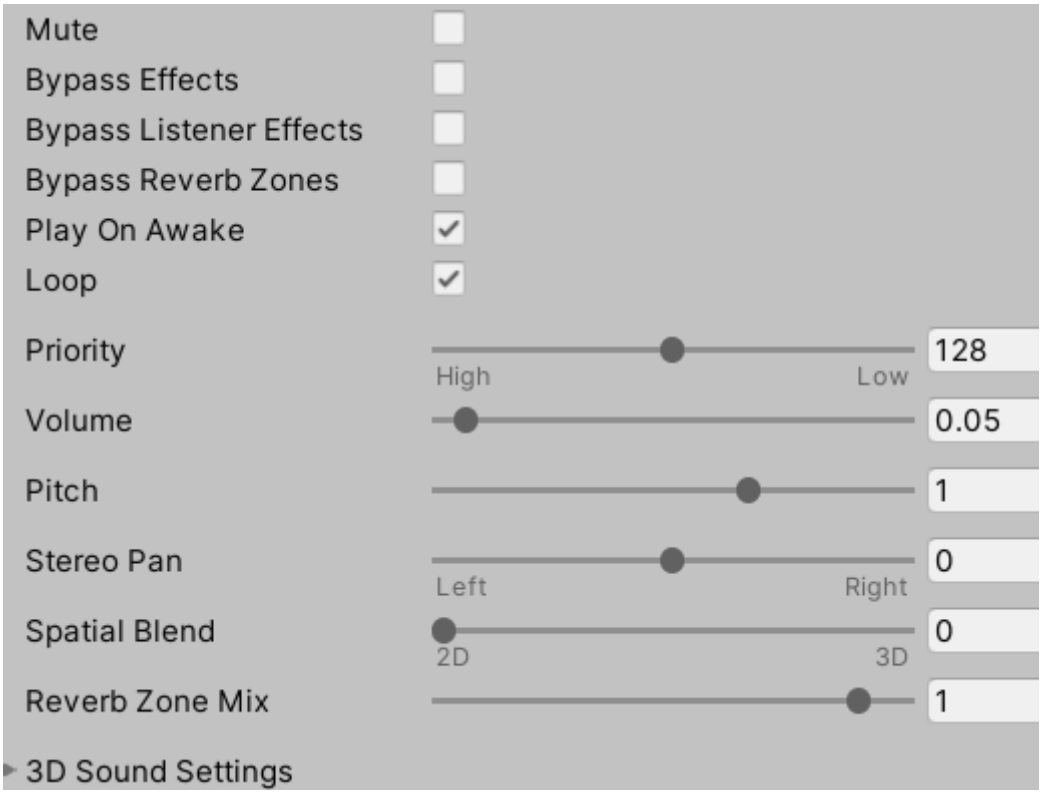
The 3 options are

1. AudioSource - where the sound is omitted from
2. AudioClip - the audio file
3. volume - the volume of the clip

For music, it is recommended that a **Audio Source** object is created and a script added to control the music and when to stop it

Otherwise, if starting and stopping the music is not a concern, just drag the music clip into the Audio Clip box and you are done





Below are some more options. Some important ones are the

1. Play on Awake - plays when scene is loaded
2. loop - loops track
3. Volume - volume of track
4. Pitch - shifts pitch (not recommended)

15. Parallax

Parallax is a popular way of displaying movement, especially if the game is a bit static.

What parallax does is like a old film reel, but instead of new frames, it is a repeat of the old frame(s).

1. Within the main camera, drag in your parallax object (I will be using clouds)
2. Clone/ copy and paste another version of the parallax object and set it just right and left of the original object if you want it to go right to left. Note the theses should be a child of the original object
3. Create new parallax script and put this in it:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class parallax : MonoBehaviour
{
    private float length;
    private float startpos;

    public GameObject cam;
    public float parallex;
```

```
    // Start is called before the first frame update
    void Start()
    {
        // if you want to do it by the y, just change x to y
        startpos = transform.position.x;

        length = GetComponent<SpriteRenderer>().bounds.size.x;
    }

    // Update is called once per frame
    void Update()
    {
        float dis = cam.transform.position.x * parallex;
        float tmp = cam.transform.position.x * (1-parallex);

        transform.position = new Vector3(startpos + dis,
transform.position.y, transform.position.z);

        if (tmp > startpos+length) {
            startpos += length;
        } else if (tmp < startpos - length) {
            startpos -= length;
        }
    }
}
```

4. Drag the script on the original parallax object
5. Drag the main camera in the camera slot and play with the parallax number (between 1 to 0) to see how much parallax you want

Resource: https://www.youtube.com/watch?v=zit45k6CUMk

16. Odds and Ends

For changing controls, go to Project Settings -> Input Manager and you can change your input from there

Linking Scenes

Linking scenes is quite easy and just requires knowledge of the following:

1. When moving to another scene (this will usually involve a detection collider), simply put

```
using UnityEngine.SceneManagement;

// and this to load a new or the same scene

SceneManager.LoadScene('#scene name here');
```

Death Zones

For this exercise, I will create a death zone if the player is out of bounds, however, this could easily be modified such that they are spikes.

1. Create an empty object to house the colliders for out of bounds
2. Make the colliders are a series of sprites or a tile map. As long as a collider can be set to it such that Unity can detect when the Player's colliders collide with the out of bounds colliders, it is fine
3. Create a new tag. I will call it **death**
4. Create script **death** and paste in the following

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class death : MonoBehaviour
{
    // collides with object tagged death
    private void OnTriggerEnter2D(Collider2D collision) {
        if (collision.gameObject.tag == "death") {
            Die();
        }
    }

    // kills player and resets
    void Die() {
        Destroy(transform.parent.gameObject);
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}
```

   * Optionally, you can just add this to the movement player script

5. Drag to player collider

Enemies

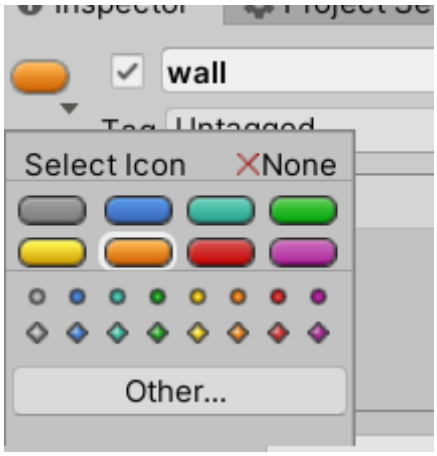
Enemies are an important yet complicated thing. They can be static, or mobile. Have easy or tough AI, or creates projectiles. For this example I will do medium-easy example of a creature in a box moving back and forth

1. Drag Sprite or Animation in the scene
2. New creatureMove script and paste in the following

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class creatureMove : MonoBehaviour
{
    // speed of creature
    public float speed = 5f;
```

```
    public bool smartTurn = true;

    // wall detection
    public Transform start, end;

    public bool collision;

    private Rigidbody2D body;

    // Start is called before the first frame update
    void Start()
    {
        body = GetComponent <Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
        collision = Physics2D.Linecast(start.position, end.position, 1 <<
LayerMask.NameToLayer("Ground"));

        if (collision == smartTurn) {
            transform.localScale = new Vector3(-transform.localScale.x,
transform.localScale.y, 1);
        }

        body.velocity = new Vector2(transform.localScale.x, 0) * speed;

    }
}
```

3. Drag script to creature
4. Create an empty object as a child of the creature
5. To see the empty object that will act as a wall detector, press the cube icon and click on a large tag



6. Place empty object in front of the creature such that if that empty interacts a wall, the creature will turn
7. Drag the creature itself into the creature box and the empty in the wall box
8. Change speed to desired amount and check **Smart Turn** to make the creature turn instead of falling off an ledge

9. If you want the creature to not walk off the edge, also cone the empty object and put it bellow the feet of the player so it is always touching the ground. Once it is not touching the ground, it should turn around

Pausing

Pausing with Unity can be made simple with one command.

1. Create script pause and drag it on your player (optionally, you can just do this in the movement folder, but this will be more organized)
2. Create a pause UI element (I will be doing a simple Pause text)
3. Paste in the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class pause : MonoBehaviour
{
    public static bool paused = false;

    public GameObject pauseMenu;

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.P)) {
            Pause();
        }
    }

    void Pause() {
        if (!paused) {
            pauseMenu.SetActive(true);
            Time.timeScale = 0f;
        } else {
            pauseMenu.SetActive(false);
            Time.timeScale = 1f;
        }
        paused = !paused;
    }
}
```

4. Place the UI element in the Pause menu box

Resource: https://www.youtube.com/watch?v=JivuXdrIHK0

Quit

To quit game simply add the following onto the player:

```
  if (Input.GetKey("escape")) // change escape to any other character if you
like
  {
      Application.Quit();
  }
```

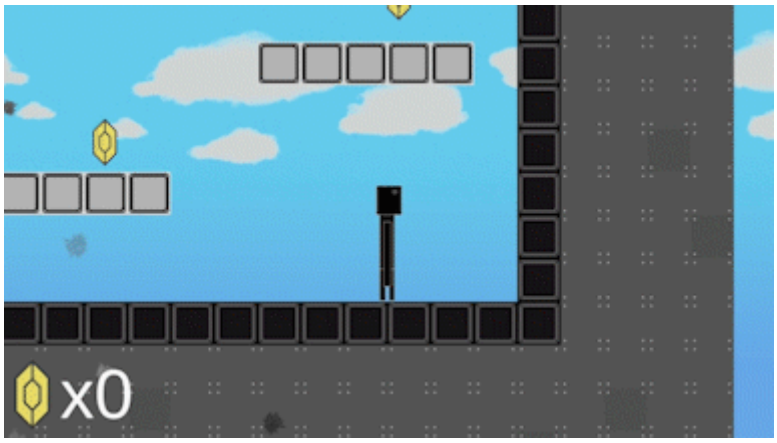
in the update method

* Note, of course more logic could be added to make this more complicated

17. End Note

I hope you had a good taste of Unity 2D. Of course, this was just a very brief introduction and there are many more aspects of Unity not covered; ie UI buttons. You can go to my Unity 3D course below if you want to learn more, or start creating your 2D game. There are many more resources online for you to check out if you need help, or want inspiration. These resource can be found in section 11 and thank you for joining me with this course. If you have any feedback, feel free to go to the GitHub and open a discussion. Again, thanks for viewing and good luck to your future Unity endeavors.

End Product/Demo



Back to Top

18. Title Screen

One of the last things to do is to create a title screen

See here for title screen section

19. Scripting

Unity is based on C# and is very much a c-styled programming language. That means it resembles programs that are written in C, C++, or Java.

The main thing to know though is Unity's implementation of C#

If you create a script, automatically, it will be drawn from a template that looks similar to the following:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class movement : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

You will notice that at the top there are imports from Unity's other libraries using the **using** keyword. Afterwards, Unity defines the script as a class*. The class is defined as public so other scripts and Unity can assess and utilize this class. The name you gave the script will follow the **class** keyword. Afterwards, **MonoBehaviour** is defined as what movement inherits from. Think of this as your script **extending** from class MonoBehaviour.

* Note a class is a data structure that holds a collection of information on the class (ie. its attributes and methods).

Below that and indented, we see void Start() and void Update(). As the comments suggest, void Start() occurs before the first frame update and void Update() is called once per frame. Note **void** means there will not be a return value (ie. no return statement).

Also, there are different methods of MonoBehaviour such as **FixedUpdate()** that may update more or less than once per frame. This will be important for any physics related functions

A list of other functions and methods that can be used can be found on the Unity Documentation site, however, the most important thing about learning and debugging Unity is to use

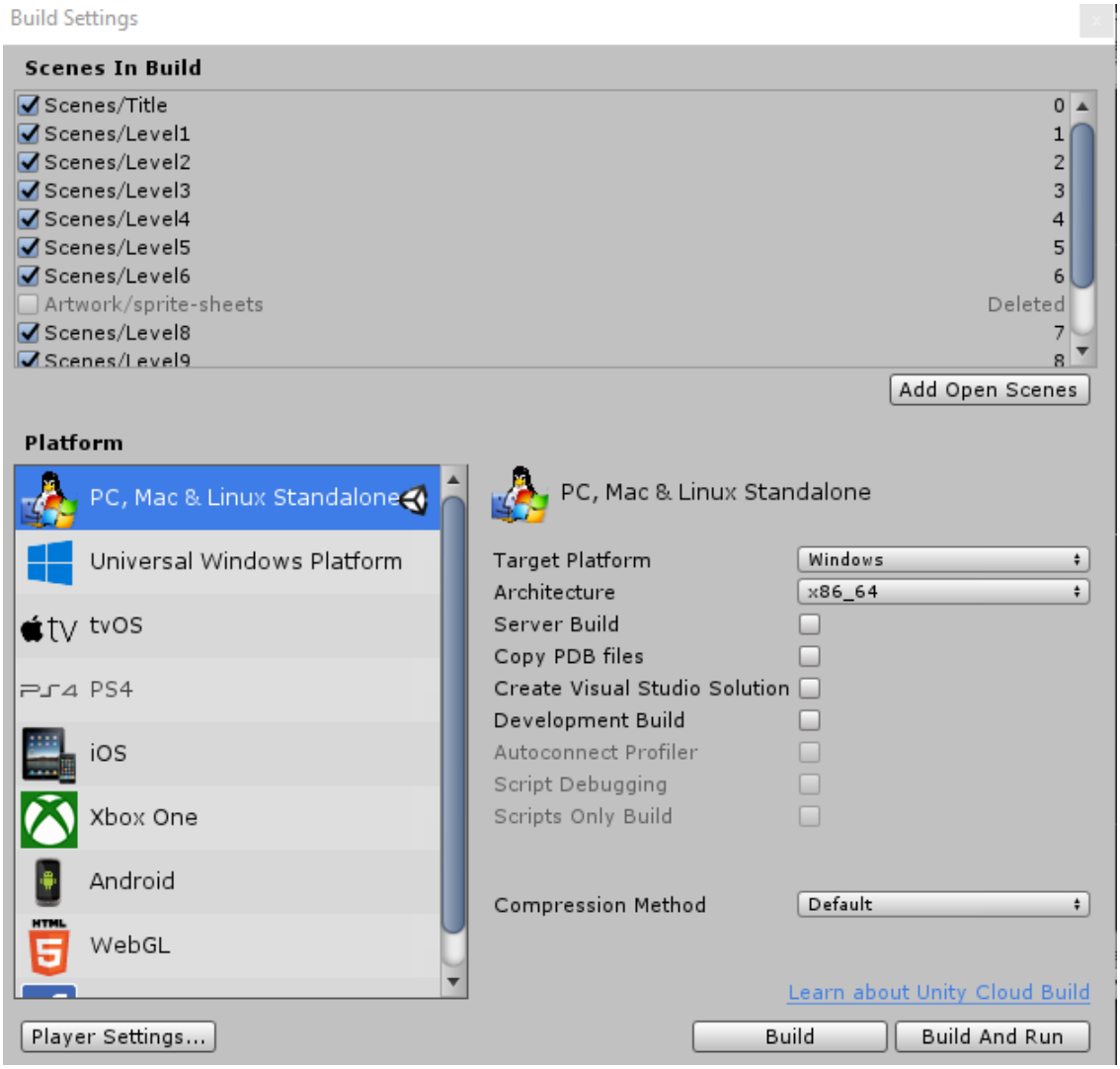
```
Debug.Log() // logs to Unity console
```

When using this, after running the game, it will output to Unity's debug console

20. Build

So you are finished your game. You need to disturbed the game. This is done through building the game.

1. Go to **File**, **Build Settings...**
2. Select the scenes that you want to build



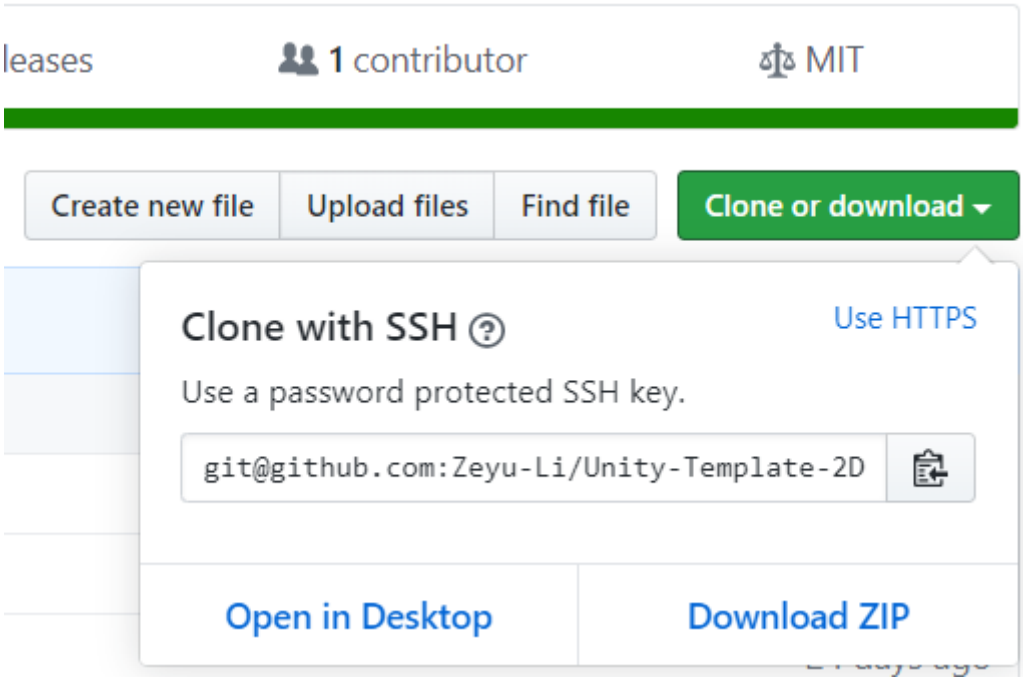
3. Select the desired **Platform**
4. You can customize the company name, product name, icon, and cursor
5. **Build** (**And Run** if you want to run it)
6. Select the folder and wait for it to build

* Note if you did not add an exit game button, the only way to exit is to close the program externally or Alt-f4

21. Cloning Guide

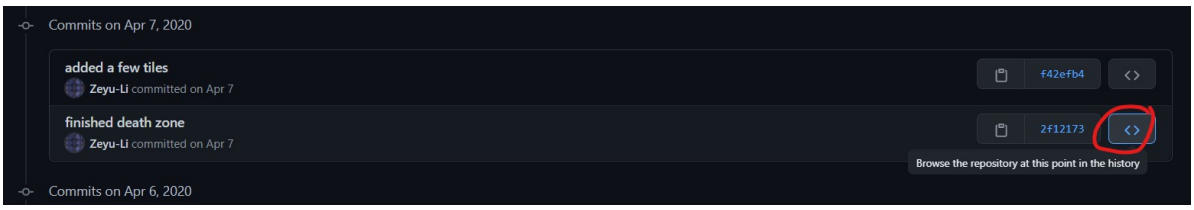
To download the finished project:

1. Go to https://github.com/Zeyu-Li/Unity-Tutorial-2D
2. Download ZIP after clicking Clone or download



3. Unzip and move to desired directory
4. Go on Unity Hub and click **Add** and located to directory
5. The project should appear in Projects and it is done
6. Click on the project to open it

* Note if you want the project at different steps click on commits or https://github.com/Zeyu-Li/Unity-Tutorial-2D/commits/master and between Feb 26, 2020 `added logo` commit to April 12, 2020 `finished Unity template game` is when various steps where completed. To download the project file at these times, click on the bracket thing (see below) and repeat above from step 2

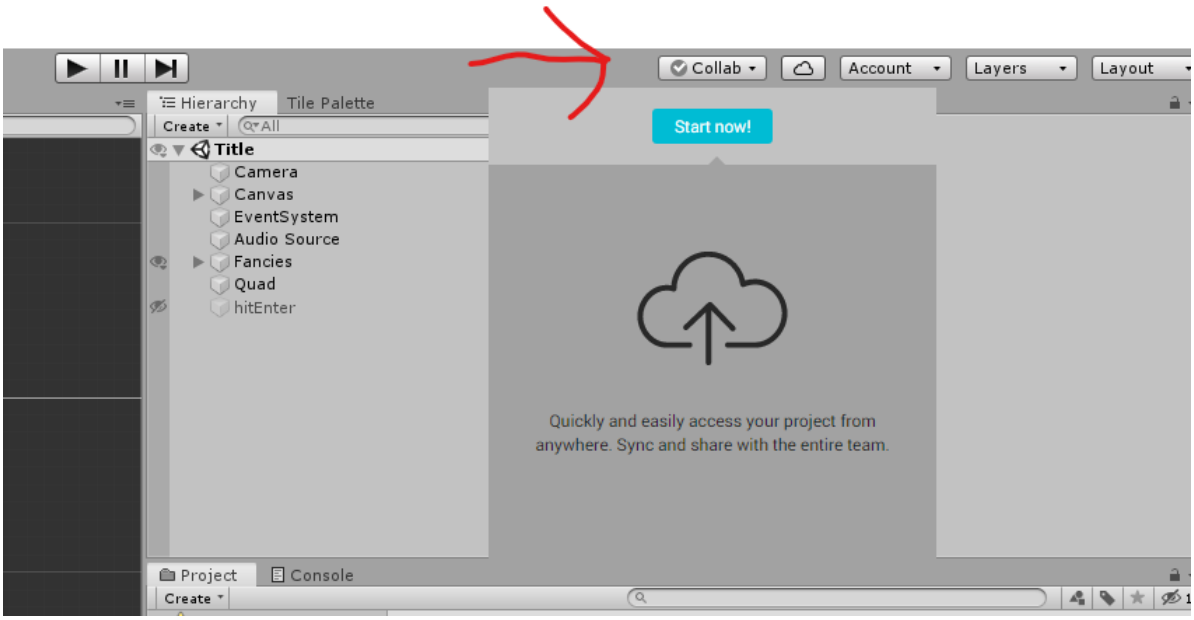


22. Collaboration

Unity's Collab

Collaborating will friends can be tough will Unity, especially with git/GitHub, but it you are careful, things could turn out fine.

If you need to collaborate, the best way is to use Unity's Collab feature



However, note that this only supports 4 people (which is usually enough because only people that will contribute to the codebase should need it)

GitHub

Otherwise if you decide to GitHub, here are some general practices:

1. Everyone should work on there own branches and work on there own scenes working towards **Prefabs** (ie one person on movement, one on moving platforms, etc.)
2. Optionally, have one person on the master (final) level to peace it together
3. Use this gitignore or clone from my projects on GitHub
4. Have the core gameplay done first (especially at a Game Jam)
5. Get to know Git/GitHub and how version control works
6. Remember you can go back to a previous working version
7. Have fun and don't fight with your team members

Back to Top

23. Resources

- The Unity User Manual provides some great documentation with code that can for the most part be copied and pasted
- Brackey's YouTube channel
- Lynda.com - If you have a library card, there is a high probability that you have access to Lynda with their organization deal
- Google is your best friend
- Ask on Reddit or Stack Exchange, don't worry they won't bite