Unity for Children

JANUARY 04, 2021

- Learning Objectives: Game Management and C# Basics
- Introduction
- AudioMixer
  - AudioMixer Groups
  - Effects
    - Lowpass and Highpass Effect
    - Pitch Shifter
    - SFX Reverb
    - Duck Volume
    - Send Effect
  - Routing AudioSource Output
    - Checkoff Information
  - Snapshots
  - Views
  - More AudioMixers
- Object Pooling
  - C#: Enumeration Types
  - C#: Helper Class
  - C#: List
  - Getting the Pooled Object
  - C#: Static Variable
  - Calling GetPooledObject()
  - Using ObjectPooler.cs

- Scriptable Objects
- Creating Managers
  - Scoring System
  - GameManager.cs
  - CentralManager.cs
  - C#: Static variable
  - Create EnemyController.cs Script for Enemy Prefab
  - Damaging Player
  - C#: Delegates and Events
    - Checkoff information
  - PowerupManager.cs
  - C#: Interface
  - PowerupManager.cs
    - PowerupUI
  - C#: Switch statements
- Checkoff
- Next

Learning Objectives: Game Management and C# Basics

- **Managing Audio**: Introduction to AudioMixer
- **Optimising** the game: Object Pooling
- **Managing** the game:
  - Data containers using **scriptable objects**
  - Creating global game state and manager
  - Creating a hierarchy of managers: Central Manager, Powerup Manager, Spawn Manager, and Game Manager
- **C# Basics:**
  - Enums

- List
- Switch statements
- Static variables
- **Interface and inheritance:** for managing multiple object types (consumables, enemies, etc),
- **Delegates** and **events**

Introduction

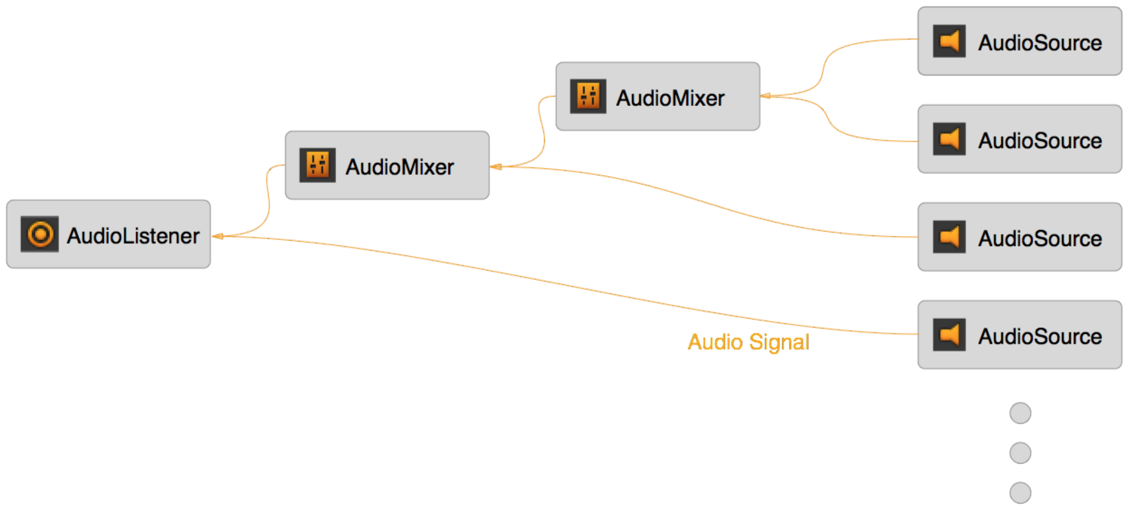
The main purpose of this Lab is to introduce a few tools that can be used to manage the game better. For example, right now we have game states spread all over various scripts, audio source spread everywhere on each object, hard-to-read game logic, etc. We can improve the structure of the game better with the help of AudioMixer, ScriptableObject, Unity Event, and a few other C# basics.

If you already know most of the learning objectives of the Lab, feel free to head straight to the Checkoff section. As usual, you don't have to follow everything in this Lab step by step if you have your own preferences to manage your game. However it is imperative that you still know what is the content of this Lab as they are part of our course syllabus.

AudioMixer

The Unity AudioMixer allows you to **route** each AudioSource output in your scene, **mix**, apply **effects** on any of them and perform **mastering**. You can apply effects such as volume attenuation and pitch altering for each individual AudioSource. For example, you can reduce the background music of the game when there's a cutscene or conversation. We can mix any number of input signals from each AudioSource, and have exactly **one** output in the Scene: **the AudioListener**

usually attached to the MainCamera. The figure below illustrates this scenario, taken from the Official Unity AudioMixer Documentation:



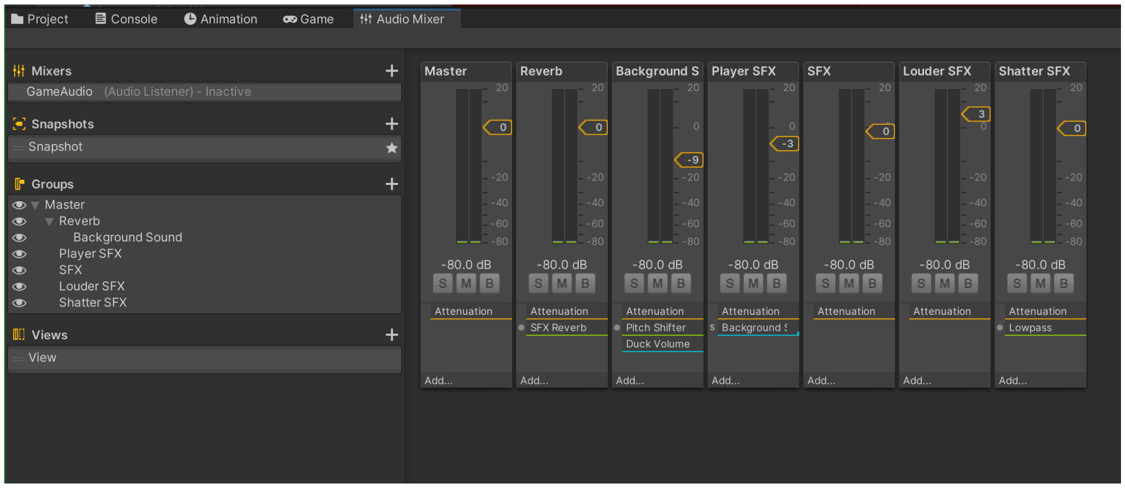
We will not be going in depth about DSP concepts in Audio, and we assume that it is your job to find out which effects are more suitable for your project. However we will still be picking a few examples to illustrate how to utilize the AudioMixer. They include: lowpass and highpass, pitch shifting, reverb, and duck volume.

Click Window » Audio » AudioMixer to have the AudioMixer tab open. On the left, there are four sections: **Mixers**, **Groups**, **Snapshot**, and **Views**. The first section is called Mixers, and they contain the overview of all AudioMixers in your Project. The output of each mixer is routed to a Scene's default **Audio Listener** typically attached on the **Main Camera**.

> You can also route the output of one AudioMixer to another AudioMixer if your project is complex. Specifics can be found in the official documentation.

AudioMixer Groups

On the third section, there's **Groups**. You can create several AudioMixer groups and form a *hierarchy*. Each group creates another bus in the Audio Group strip view, as shown:



The purpose of having these groups is so that the output of each AudioSource can be routed here, and effects can be applied. Effects of the parent group is applied to all the children groups. Each group has a Master group by default, which controls the attenuation (loudness) of all children Audio groups.

Create five audio groups as shown in the screenshot above. You will only have Attenuation effect applied on each of them in the beginning, set at 0 dB. The attenuation is simply *relative* to one another, applied on a relative unit of measurement *decibels*.

> **TL;DR** If you have sound group A at 0 dB, sound group B at 1 dB and sound group C at -2 dB, that means sound group B is the loudest among the three. How much louder is sound group B as compared to A? It depends on your perception. A *change* of 10 **dB** is accepted as the difference in level that is perceived by most listeners as "**twice as loud**" or "half as **loud**"

You can set the initial attenuation for each group to vary by default. For example, the volume of Background Sound by default is less (at -9 dB) than Player SFX (at -3 dB).

Effects

You can apply various effects within each audio group. The effects are applied from top to bottom, meaning the **order of effects** can impact the **final** output of the sound. You can drag each effect in the strip view (the view shown in the screenshot above) to reorder them.

Lowpass and Highpass Effect

Click on Shatter SFX and **add** the **Lowpass** effect in the Inspector. You can set two properties:

- Cutoff frequency
- Resonance

The cutoff frequency indicates the frequency that is allowed to pass the filter, so the output will contain range of frequencies from zero up to this cutoff frequency. As for resonance, you can leave the value as 1.00 unless you're familiar with it. It dictates how much the filter's self-resonance is **dampened**.

> From Unity documentation: Higher lowpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

An audio with lowpass filter effect applied will sound more dull and less sharp, for example it is ideal for effects where the game character throw blunt objects around.

On the contrary, **Highpass** effect allows us to pass any frequency **above** the cutoff. If you'd like to pass through certain bands, let's say between 1000 Hz to 5000 Hz, then you can apply a lowpass at 5000Hz, and then a highpass at 1000Hz. The order doesn't matter in this case, as long as you set the right frequency cutoff for each type of effect.

Pitch Shifter

The **pitch shifter effect** allows you to change the pitch of the group **without** causing the output to sound sped up or slowed down. This is formally called as **pitch scaling**: the process of changing audio pitch without affecting its speed. It is commonly used in games to improve the game feel by making sound effects *less repetitive* and by helping to convey information to the player, for example:

- **Collision information:** the speed, material of items colliding
- **Importance** of events
- **Response** to player inputs and actions

This *pitch shifter effect* **different** from the **pitch slider** located at the top of the group inspector. The regular pitch slider changes the pitch of the audio file by manipulating the sampling frequency of the audio output.

> If the system's sampling rate was set at 44.1 kHz and we used a 22.05 kHz audio file, the system would read the samples *faster* than it should. As a result, the audio would sound sped up and higher-pitched. The inverse also can happen. Playing an audio file slower than it should will cause it to sound slowed down and lower-pitched.

Add the pitch shifter effect on Background Sound group and adjust the properties accordingly:

- **Pitch**: adjust this slider to determine the pitch multiplier
- **FFT size, Overlap, and Max Channels**: real-time pitch shifting is done using Fourier Transform.

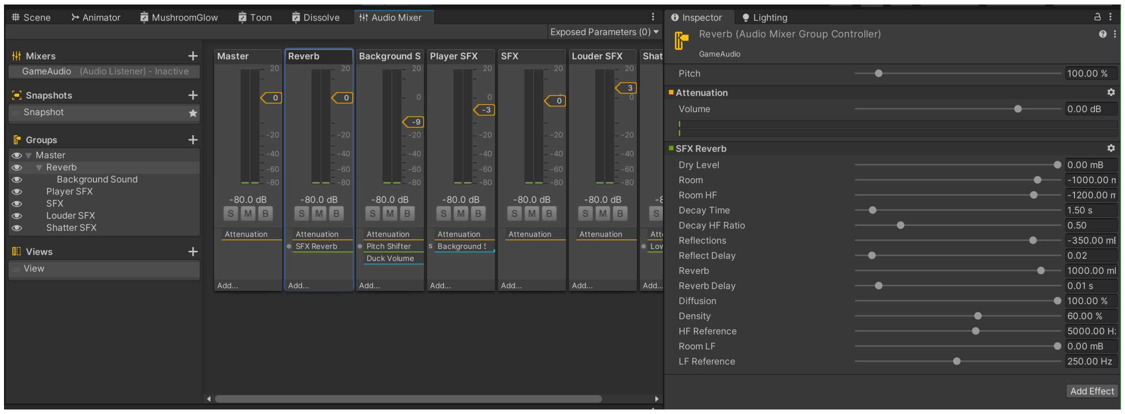
  > You're not required to know the details behind pitch shifting DSP. If you're interested in the details of such implementation, you may refer to the Python implementation here. Since Unity is not open source, we do not know the exact implementation of these effects, but we can certainly learn to implement our own pitch shifter or any other audio effects by reading other research papers online.

SFX Reverb

This effect causes the sound group to be as if it is played within a room, having that complex echo effect. Each room sounds differently, depending on the amount of things that exist in a room, for example listening to music in the bathroom (high reverb) sounds different than listening to the same music in an open field. Routing all audio output through the same reverb filter has the effect as if each of the audio files are played in the same room.

This is particularly handy if you have different sections in your game, e.g: cave, open field, wooden houses, etc. For example, the player's footsteps will sound different in each scenario, despite having the same audio file for footsteps.

Add a new audio group called **Reverb**, add add this SFX Reverb effect to the group. Set the properties as follows and set it as the parent group of the Background Sound.



Reverb (Audio Mixer Group Controller)
GameAudio

Pitch	100.00 %
Attenuation	
Volume	0.00 dB
SFX Reverb	
Dry Level	0.00 mB
Room	-1000.00 n
Room HF	-1200.00 m
Decay Time	1.50 s
Decay HF Ratio	0.50
Reflections	-350.00 mE
Reflect Delay	0.02
Reverb	1000.00 ml
Reverb Delay	0.01 s
Diffusion	100.00 %
Density	60.00 %
HF Reference	5000.00 H:
Room LF	0.00 mB
LF Reference	250.00 Hz

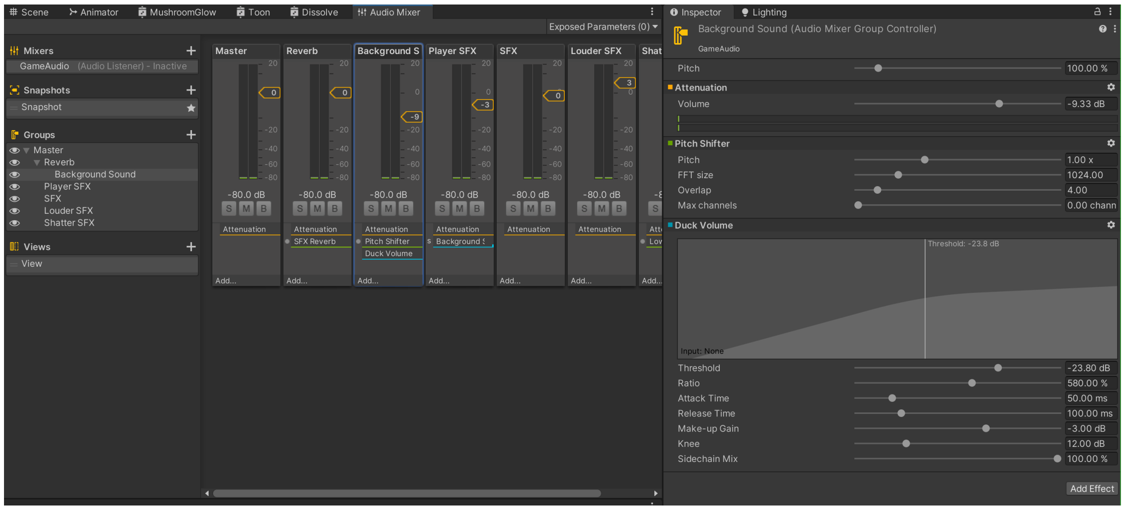
This will give us an "empty room" reverb effect for the Background Sound group since the latter is the child group of the former.

SFX Reverb has many properties, and it is best if you find presets online to get the room effect that you want without going into the details. The documentation provides a brief description of each property, but it doesn't seem like much help for beginners who aren't familiar with the term.

Duck Volume

Another very useful effect is duck volume, that allows the group's volume to automatically reduce when something else is playing above a certain threshold. For example, if we want to reduce the background music whenever the player is jumping.

Add Duck Volume effect in Background Sound group, and set its threshold to be -65 dB.



Background Sound (Audio Mixer Group Controller)
GameAudio

Pitch 100.00 %

Attenuation
Volume -9.33 dB

Pitch Shifter
Pitch 1.00 x
FFT size 1024.00
Overlap 4.00
Max channels 0.00 chann

Duck Volume
Threshold: -23.8 dB
Input: None
Threshold -23.80 dB
Ratio 580.00 %
Attack Time 50.00 ms
Release Time 100.00 ms
Make-up Gain -3.00 dB
Knee 12.00 dB
Sidechain Mix 100.00 %

Add Effect

This configuration means:

- Any input over -65db will cause the volume to go down (**Threshold**)
- The volume will go down by quite a high amount (**Ratio**)
- The volume will go down very quickly (**Attack** time) at 100 ms and, after the alert has gone below -25db, go back to normal somewhat quickly (**Release** time).

Leave the other properties as is unless you know what they means. Right now we do not need it, and we are confident that you can learn them independently next time when you need it.

> You will be faced with the warning that there's no send source connected for now. We will tackle it in this immediate next section.

Send Effect

Now the final thing to do is to **Send** an input to this Duck Volume effect, which is the source that can cause this audio group's volume to *duck*. Click Player SFX and add the Send effect. Select Background Sound as its Receive target and set its send level to 0.00 dB (so the background sound unit can receive the full amount of Player SFX output).

Routing AudioSource Output

Now go back to your scene and click on every type of AudioSource present in your scene, and set the output to each group accordingly. We can also create new ones whenever we deem appropriate. Below are the lists of AudioSources that we can have as example:

- **AudioSource** with theme.mp3 as `clip` and Background Sound group as `output`, attached to MainCamera gameobject.
- **AudioSource** with smb_jump-small.mp3 as `clip` and Player SFX group as `output`, attached to Mario gameobject.
- **AudioSource** with shatter sound effect as `clip` (download it yourself) and Shatter SFX, attached to the the BreakableBrick gameobject.

You are free to create more audio mixer groups and effects, and route any AudioSource output to any group that you deem fit.

Checkoff Information

As part of Checkoff, you are required to create your own AudioMixer groups, apply 1-2 interesting effects on each group, download your own audio clips and demo it in your screen recording (make sure you record the sound as well!).

Snapshots

Snapshot is basically a saved state of your Audio Mixer.

> From Unity documentation: Snapshot is a great way to define moods or themes of the mix and have those moods change as the player progresses through the game.

For example, changing the parameters of SFX Reverb on different stages in the game, depending on the location of the player. It is more convenient than manually changing all SFX Reverb parameters one by one during runtime through scripts.

You can change between snapshots programmatically using the following code:

```
private AudioMixerSnapshot snapshot1;
public AudioMixer mixer;

// instantiate somewhere in the code
snapshot1 = mixer.FindSnapshot("Snapshot1_name");

// then transition
snapshot1.TransitionTo(.5f); //transition to snapshot1
```

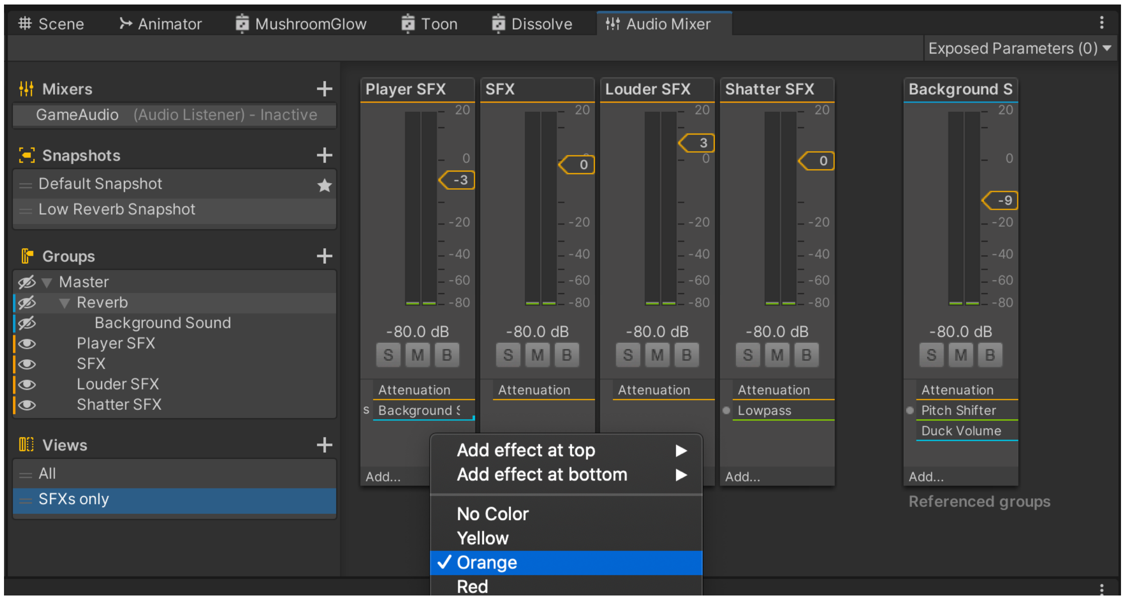
We already have one snapshot by default, the one with the star symbol. This is our **starting** snapshot (the state that will be used when we start the game). To create more snapshots, simply click the + button beside it and give it a good name. Highlight (click) on the new snapshot and start changing any parameters within the AudioMixer: volume, pitch, send level, wet-mix level, effect **parameters**.

> Snapshot won't work with new effects or new group within the Mixer, only the parameters. If you create or delete groups or effects, it will be reflected on all Snapshots.

Views

The last concept that can make your life easier if you have a complex project is to separate your views so you can focus on groups that matter for your current work. You can create a new View by clicking the + button, rename the view and click on it. Then start hiding some groups that you don't want to see within this view.

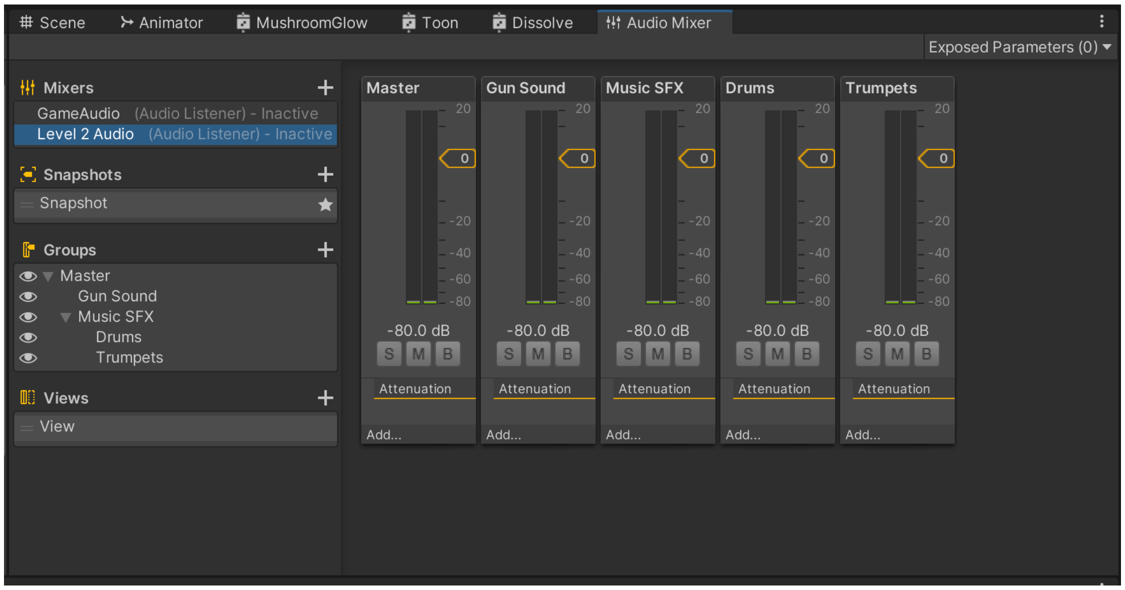
For example, here we have SFX views that show only SFX related audio groups, and hide the background sound group:



You can also **color code** each group (right click on the group) to visually separate your audio groups.

More AudioMixers

You can create more AudioMixers, and route its output through existing AudioMixers (if you don't then all output of the mixers will be routed to the Scene's AudioListeners). This is useful to have better modularity and management if you have many stages in your game. For example, you can create a new AudioMixer for so-called "level 2" of your game:



If you need to route the output of *Level 2 Audio* to GameAudio, **drag** its entry under Mixers and **place it on GameAudio**, then select the group that you want use to receive the output of *Level 2 Audio*. This way we can **apply chain of audio effects** conveniently, with clear visual separation between each group and mixers.

> Unless you have complex Audio system in your game, creating multiple groups and arranging their hierarchy is usually sufficient. You don't really need to route multiple mixers together. You can use different mixers for different set of AudioSources in completely different scenes instead.

Object Pooling

Object pooling is a design pattern that you can use to Object Pooling to optimize your projects by lowering the burden that is placed on the CPU when having to rapidly **instantiate** and **destroy** GameObjects. It is particularly useful for top-down bullet-spray games, or games that have swarms of monsters that are constantly created and destroyed at runtime.

Although Super Mario Bros do not have anything that needs to be spawned and destroyed many times (think hundreds!) at runtime, we can try to apply this concept to manage various enemies in the game.

The main idea of Object Pooling is as follows:

- Instantiate `N` objects at `Start()` , but render them **inactive**. Place all of them in a *pool*.
- Activate each objects at runtime accordingly, instead of instantiating new ones. This action removes *available* objects from the *pool*.
- After we are done with these objects, deactivate them. This essentially returns the object back to the *pool*, ready to be reused next time.
- The *pool* may run out of objects to be activated eventually, and we can expand the pool at runtime. This requires instantiation of new gameObjects obviously, so try to reduce the need to do so and instantiate enough relevant game objects at `Start()` .

Create a **new** script called ObjectPooler.cs.

C#: Enumeration Types

We begin by stating the types of objects that we can instantiate in our Pool. We can skip this step and use Tags instead, but it might be more efficient to use `enum` type.

> An *enumeration type* (or *enum type*) is a value type defined by a set of named constants of the underlying integral numeric type.

As an example, we have two enemy types: gomba and green. You can define your own integral numeric type on the right hand side, but for sanity sake let's use something intuitive as follows:

```
public  enum ObjectType{
        gombaEnemy =  0,
        greenEnemy =  1
}
```

Paste the above code outside `ObjectPooler` class in `ObjectPooler.cs` script.

C#: Helper Class

We can define other classes as well, without having to inherit MonoBehavior. These classes are our helper class. We need to create two of these:

- A class to define the data structure of an Object *metadata* to be spawned into the pool
- A class to define the data structure of an Object in the pool

For the former, we can define something like this:

```
[System.Serializable]
public  class ObjectPoolItem
{
        public  int amount;
        public  GameObject prefab;
        public  bool expandPool;
        public  ObjectType type;
}
```

The attribute `[System.Serializable]` indicates that a class or a struct can be serialized. In laymen terms: **visible** and **customisable** in the **inspector** when declared as a public instance `public ObjectPoolItem i` later on.

For the latter, we have the following:

```
public  class ExistingPoolItem
{
        public  GameObject gameObject;
        public  ObjectType type;

        // constructor
        public  ExistingPoolItem(GameObject gameObject, ObjectType type){
                // reference input
                this.gameObject  =  gameObject;
                this.type  =  type;
        }
}
```

C#: List

Under ObjectPooler class, we declare the following variables to hold different kinds of object *metadata* in the pool and references to objects spawned in the pool:

```
public  List<ObjectPoolItem> itemsToPool; // types of different object to pool
public  List<ExistingPoolItem> pooledObjects; // a list of all objects in the pool,
```

So for example, if we have **two** types of monsters: Gomba and Green, the number of elements in `itemsToPool` is exactly **two** `ObjectPoolItems` , each containing the metadata for each monster in the form of `ObjectPoolItem` .

Now suppose we want to have at maximum **three** Gombas and **six** Green spawned (but deactivated at first), then the number of elements in pooledObjects is **nine**, each in the form of `ExistingPoolItem` .

We then can implement the `Awake()` method of ObjectPooler class, where we spawn all items for each `ObjectPoolItem` and put them inside `pooledObjects` List.

```
void  Awake()
{
        pooledObjects  =  new  List<ExistingPoolItem>();
        foreach (ObjectPoolItem item in  itemsToPool)
        {
                for (int i =  0; i  <  item.amount; i++)
                {
                        // this 'pickup' a local variable, but Unity will not remove
                        GameObject pickup = (GameObject)Instantiate(item.prefab);
                        pickup.SetActive(false);
                        pickup.transform.parent  =  this.transform;
                        ExistingPoolItem e =  new  ExistingPoolItem(pickup, item.typ
                        pooledObjects.Add(e);
                }
        }
    }
```

Notice that we created a **local** variable `e` , that contains a **reference** to a newly instantiated `ExistingPoolItem` object *inside this for-loop*, and then we add that reference to `pooledObjects` list. In good old C, you will end up with the horrible `segmentation fault` because this reference will be *out of scope*. C# variables are also generally **scoped** within the nearest set of `{}` s. However since we are adding it to the List, C# is **smart enough** to know that you still need the content of `e` later on using `pooledObjects[i]` and therefore allocates the memory space differently. `e` does not exist anymore, i.e: you can't simply `Debug.Log(e)` after the loop exists, but its content stays in `pooledObjects` . It is out of our syllabus, but we just want you to take a moment to *appreciate* how this feature makes our lives so much easier.

Anyway, you can have a neater code with inline instantiation instead:

```
pooledObjects.Add(new  ExistingPoolItem(pickup, item.type));
```

Getting the Pooled Object

Create a public method to return GameObject reference of one of the requested objects in the pool when available:

```
public  GameObject  GetPooledObject(ObjectType type)
{
        // return inactive pooled object if it matches the type
        for (int i =  0; i  <  pooledObjects.Count; i++)
        {
                if (!pooledObjects[i].gameObject.activeInHierarchy  &&  pooledObject
                {
                        return  pooledObjects[i].gameObject;
                }
        }
        return null;
}
```

There's no method `returnPooledObject` because the *user* of the pooled object should render the object back as inactive. Deactivating the pooled objects essentially "returns" the object back to the pool, which makes it convenient to use.

> Any script controlling the pooled object must be aware of this setup, such as resetting all its state and `transform.position` when deactivated so that it can be reused another time.

We can also expand the Pool if we didn't instantiate enough objects in the Pool at `awake`. Notice that `ObjectPoolItem` as the attribute `bool expandPool`. We can add the following check after the `for` loop but before `return null` to instantiate the requested object at runtime if `expandPool == true`.

```
// this will be called when no more active object is present, item to expand pool i
foreach (ObjectPoolItem item in itemsToPool)
{
        if (item.type == type)
        {
                if (item.expandPool)
                {
                        GameObject pickup = (GameObject)Instantiate(item.prefab);
                        pickup.SetActive(false);
                        pickup.transform.parent  =  this.transform;
                        pooledObjects.Add(new  ExistingPoolItem(pickup, item.type));
                        return  pickup;
                }
        }
}
```

The drawback with this method is that you have to loop through both `pooledObjects` and `itemsToPool` whenever this method is called *and* there's no available object to return. You can use better data structures to avoid this, but nevertheless it shouldn't be a problem when run on modern computers.

C#: Static Variable

Finally, we want to be able to access the ObjectPooler instance fast and since there should only be one instance of this per Scene, we can create a static variable in ObjectPooler.cs too and set it in `Awake` :

```
public static ObjectPooler SharedInstance

void Awake(){
        SharedInstance = this;
        // other instructions
        ....
}
```

Calling GetPooledObject()

To demonstrate how we can utilise the object pooler, create a new script `SpawnManager.cs` . This script will call `GetPooledObject()` method in `ObjectPooler.cs` (obtain objects from the pool). Add the following method in `SpawnManager.cs` :

```
void  spawnFromPooler(ObjectType i){
        // static method access
        GameObject item =  ObjectPooler.SharedInstance.GetPooledObject(i);
        if (item  !=  null){
                //set position, and other necessary states
                item.transform.position  =  new  Vector3(Random.Range(-4.5f, 4.5f),
                item.SetActive(true);
        }
        else{
                Debug.Log("not enough items in the pool.");
        }
}
```

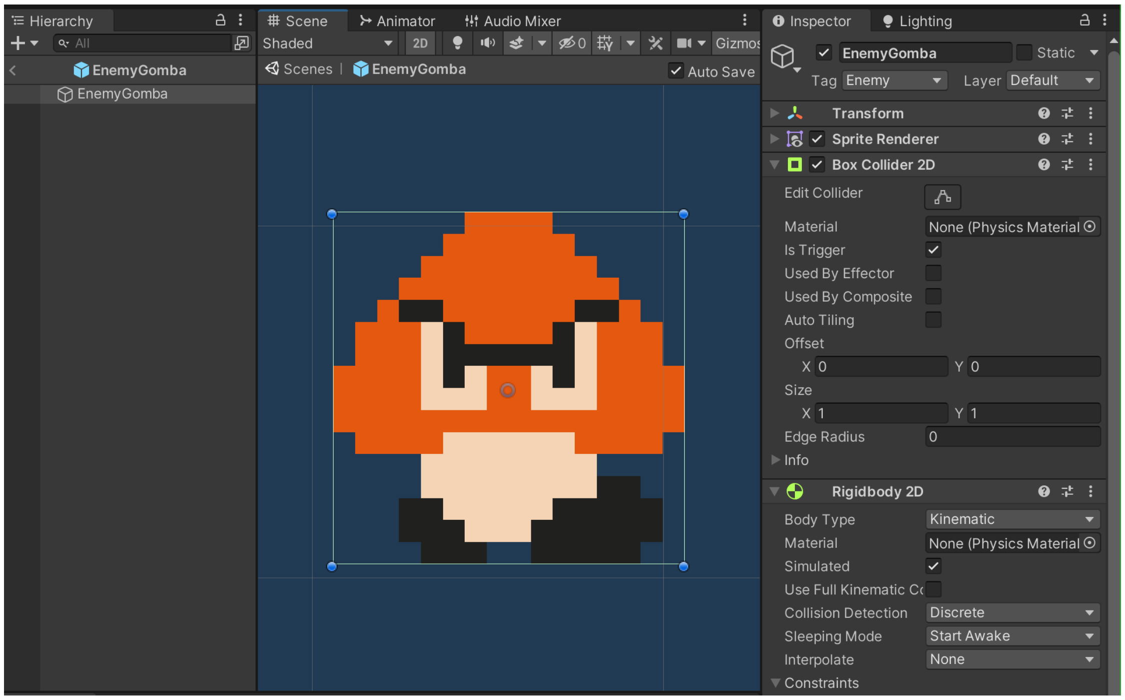
> Notice that we do not need to get the instance reference for `ObjectPooler` beforehand since we utilise the `static ObjectPooler` reference we created earlier.

Then we can simply test spawning something in `Awake()` :

```
// spawn two gombaEnemy
for (int j =  0; j  <  2; j++)
        spawnFromPooler(ObjectType.gombaEnemy);
```

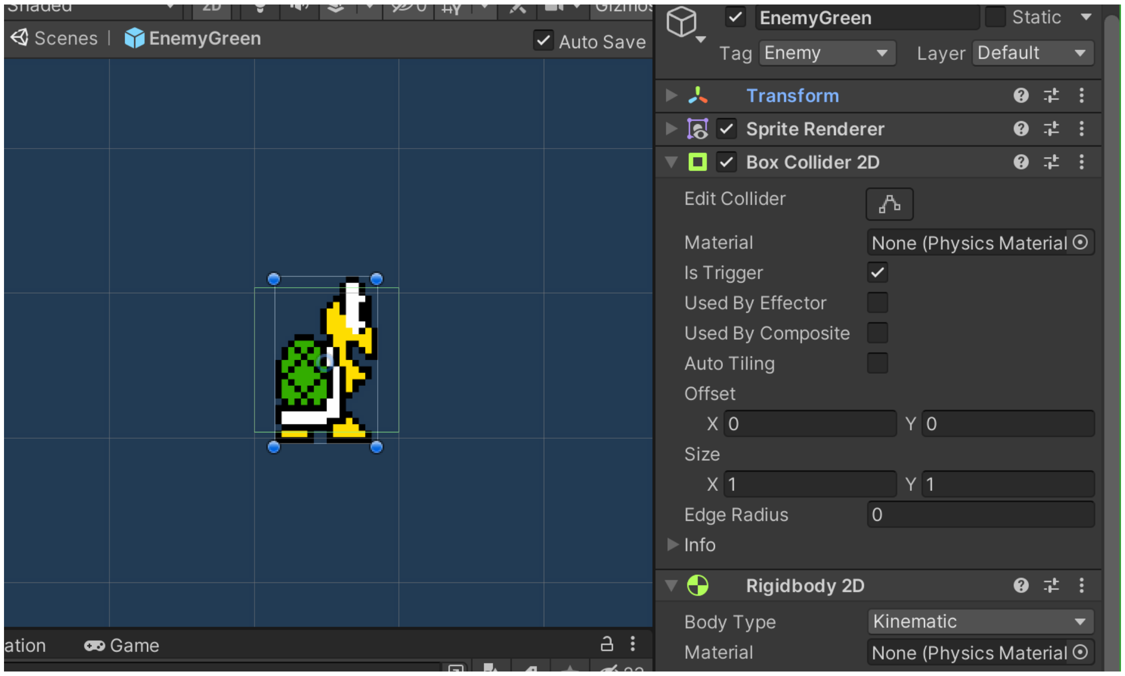
Using ObjectPooler.cs

Before we can use `ObjectPooler.cs` as a component, we need to prepare the prefabs for each object type: `gombaEnemy` and `greenEnemy` . Create these two prefabs with any sprite you want. Here's our sample for `gombaEnemy` :



EnemyGomba Static
Tag Enemy Layer Default
Transform
Sprite Renderer
Box Collider 2D
Edit Collider
Material None (Physics Material
Is Trigger ✓
Used By Effector
Used By Composite
Auto Tiling
Offset
X 0 Y 0
Size
X 1 Y 1
Edge Radius 0
Info
Rigidbody 2D
Body Type Kinematic
Material None (Physics Material
Simulated ✓
Use Full Kinematic Co
Collision Detection Discrete
Sleeping Mode Start Awake
Interpolate None
Constraints

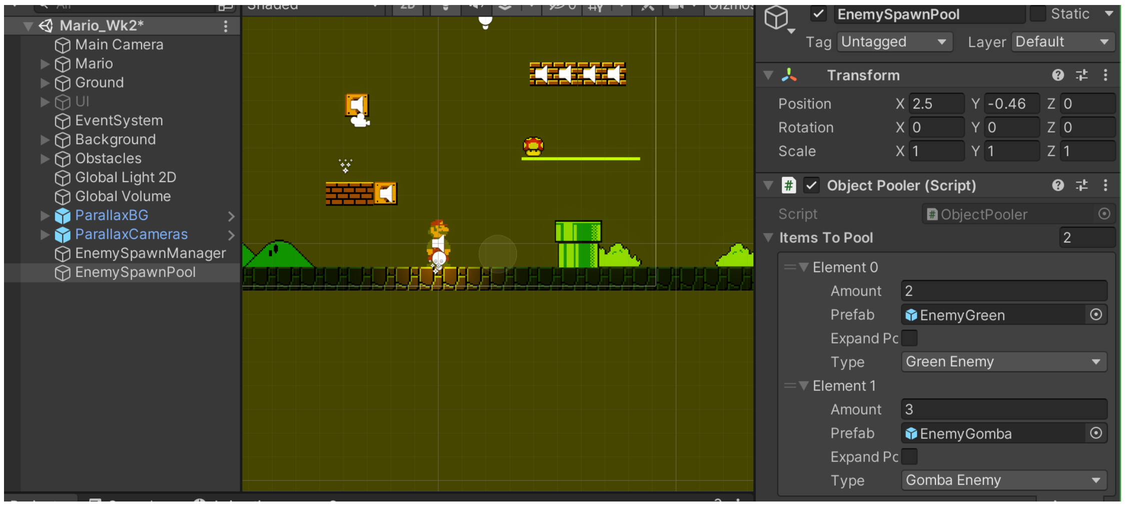
and our sample for `greenEnemy` :



EnemyGreen Static
Tag Enemy Layer Default
Transform
Sprite Renderer
Box Collider 2D
Edit Collider
Material None (Physics Material
Is Trigger ✓
Used By Effector
Used By Composite
Auto Tiling
Offset
X 0 Y 0
Size
X 1 Y 1
Edge Radius 0
Info
Rigidbody 2D
Body Type Kinematic
Material None (Physics Material

We added Trigger Collider and Kinematic Rigidbody components to each of the prefab because we want to detect collision with the player later on in the script, etc. You can add any other components that you want depending on how you want to control these enemies.

Then create an Empty game object in the scene, name it `EnemySpawnPool` and attach `ObjectPooler.cs` script on it. Set up the serializable `itemsToPool` variable accordingly:



EnemySpawnPool Static
Tag Untagged Layer Default

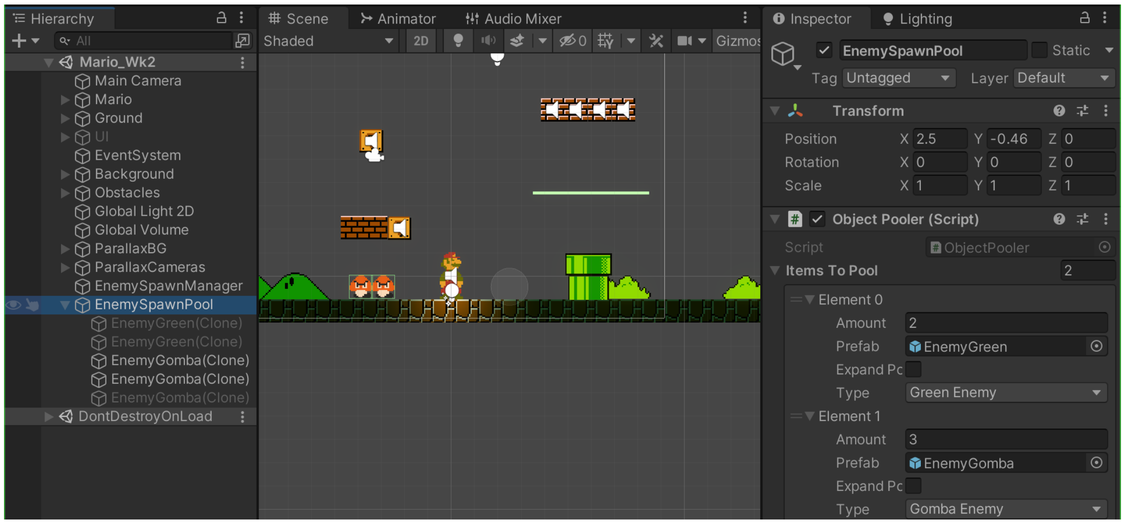
Transform
Position X 2.5 Y -0.46 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Object Pooler (Script)
Script ObjectPooler
Items To Pool 2
Element 0
Amount 2
Prefab EnemyGreen
Expand Pc
Type Green Enemy
Element 1
Amount 3
Prefab EnemyGomba
Expand Pc
Type Gomba Enemy

> If you have more enemyTypes, feel free to implement those as you deem fit.

Finally, create another Empty GameObject in the scene called `EnemySpawnManager` as shown in the screenshot above, and attach the `SpawnManager.cs` script as its component.

Test run and you shall have two cute `gombaEnemy` spawned in your scene (or whichever `enemy type` you decided to test):



EnemySpawnPool Static
Tag Untagged Layer Default

Transform
Position X 2.5 Y -0.46 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Object Pooler (Script)
Script ObjectPooler
Items To Pool 2
Element 0
Amount 2
Prefab EnemyGreen
Expand Pc
Type Green Enemy
Element 1
Amount 3
Prefab EnemyGomba
Expand Pc
Type Gomba Enemy

Under `EnemySpawnPool`, you can also observe other inactive game objects that are available in your *pool*. You can test further by spawning more enemies at runtime or test the `expandPool` feature. With that, we conclude this section about Object Pooling.

Scriptable Objects

A ScriptableObject is a **data container** that you can use to **save** large amounts of data, independent of class instances. An example scenario where this will be useful is when your game needs to instantiate tons of Prefab with a Script component that stores unchanging variables. We can save memory by storing these data in a ScriptableObject instead and these Prefabs can refer to the content of the ScriptableObject at runtime.

ScriptableObject is also useful to store standard values for your game, such as reset values, max health for each character, cost of items, etc. In the later weeks, we will also learn how to utilise ScriptableObjects to create a handy Finite State Machine.

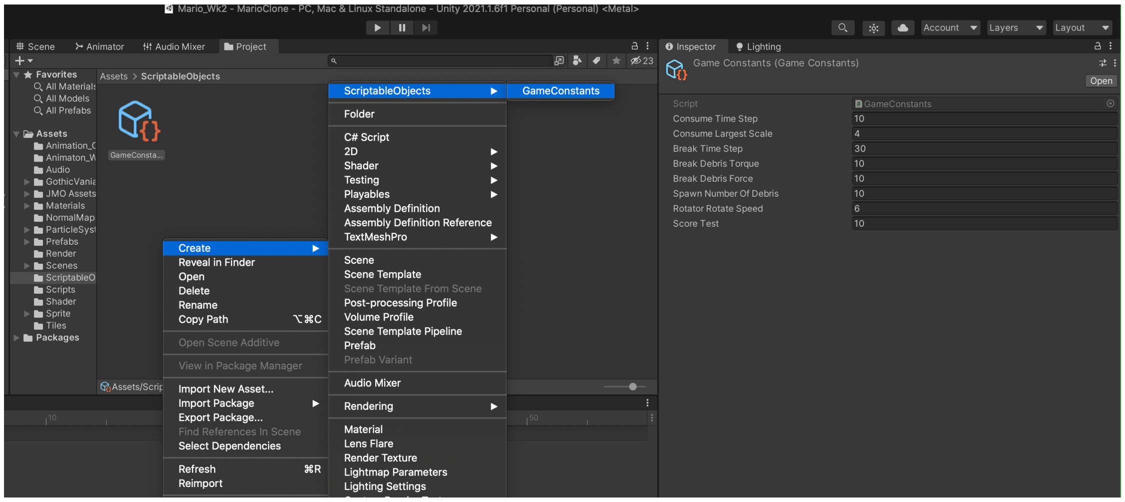
To begin creating this data container, create a new script and call it `GameConstants.cs`. Instead of inheriting `MonoBehavior` as usual, we let it inherit `ScriptableObject`:

```
using UnityEngine;

[CreateAssetMenu(fileName =  "GameConstants", menuName =  "ScriptableObjects/GameCor
public  class GameConstants : ScriptableObject
{
        // set your data here
}
```

The header `CreateAssetMenu..` allows us to create instances of this class in the Project. Proceed by declaring a few constants that might be useful for your project inside the class, for example:

```
// for Scoring system
int currentScore;
int currentPlayerHealth;

// for Reset values
Vector3 gombaSpawnPointStart = new Vector3(2.5, -0.45, 0); // hardcoded location
// .. other reset values

// for Consume.cs
public  int consumeTimeStep =  10;
public  int consumeLargestScale =  4;

// for Break.cs
public  int breakTimeStep =  30;
public  int breakDebrisTorque =  10;
public  int breakDebrisForce =  10;

// for SpawnDebris.cs
public  int spawnNumberOfDebris =  10;

// for Rotator.cs
public  int rotatorRotateSpeed =  6;

// for testing
public  int testValue;
```

Now you can instantiate the object by right clicking on the Project window then » Create, since we have declared `CreateAssetMenu`.



Game Constants (Game Constants)

Script	GameConstants
Consume Time Step	10
Consume Largest Scale	4
Break Time Step	30
Break Debris Torque	10
Break Debris Force	10
Spawn Number Of Debris	10
Rotator Rotate Speed	6
Score Test	10

You can also edit the values of the variables in the inspector. These values persist, so you can store something in these data containers during runtime, such as the player's highest score.

To use it in any script, simply declare it as a public variable and link it up in the inspector. Below is a simple sample script that can rotate the object it is attached to, at a rotational speed as set in `gameConstants` .

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

public  class Rotator : MonoBehaviour
{
        public  GameConstants gameConstants;
        private  Vector3 rotator;

        // Start is called before the first frame update
        void  Start()
        {
                rotator  =  new  Vector3(0, gameConstants.rotatorRotateSpeed, 0);
        }

        // Update is called once per frame
        void  Update()
        {
                //Rotate
                this.transform.rotation  =  Quaternion.Euler(this.transform.eulerAng
        }
    }
```

You can change these values conveniently to find the right "feel" for your game during testing.

Creating Managers

There are many design patterns to create a game manager. You can even have a hierarchy of managers in the game, depending on the role of each manager: managing score, managing enemies, managing powerups, etc. However, sometimes we may cut corners because we simply do not have enough time to read about all of them. In this section, we will create a working game manager, and we a few C# tricks. By no means we claim that this is the best way to create a game manager, and it is simply a tool to introduce to you the concept of a basic *manager*, and a few fancy C# stuffs.

> If you already know most of the concepts taught in this section, then the best way to proceed from here and learn further is to reverse engineer how others implement their managers. You can start with several advanced Unity tutorial projects freely available online.

The idea is to make sure that we have a centralised way to manage score, enemies, players, and pretty much everything important in the Scene. Suppose we want a game with three basic system:

1. **Scoring system:** enemies spawned from the pool can be killed and this will increase the score.
2. **Damage system:** player will die if it collides with the enemies from the side before managing to kill the enemy.

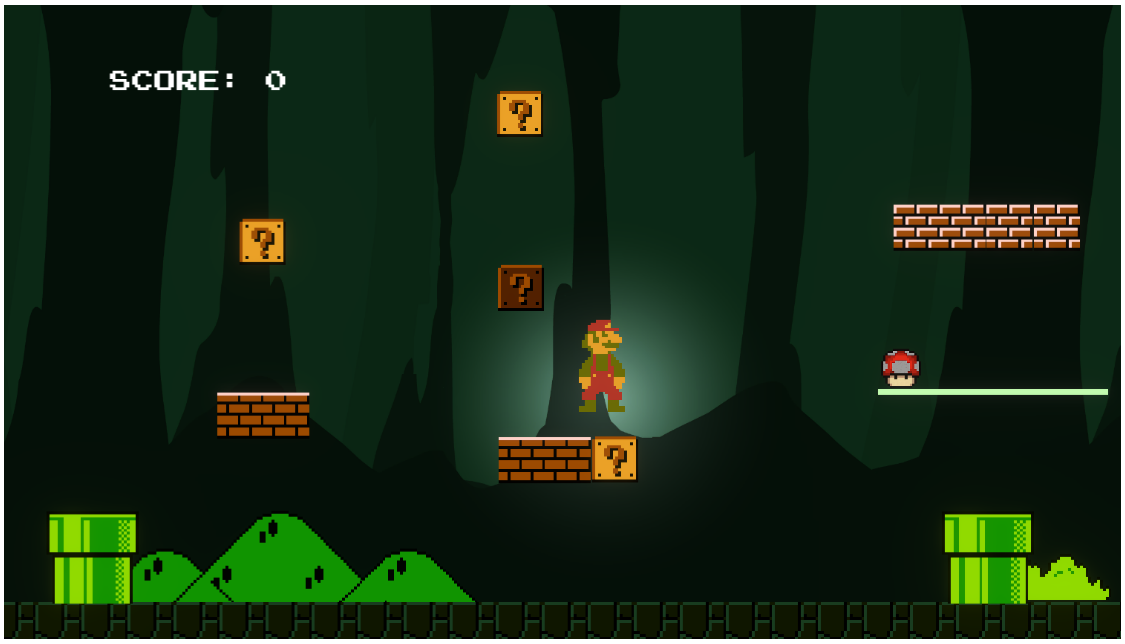
3. **Powerup system:** player can collect powerups and use it to boost something for a fixed period of time.

There has to be a script to manage all these game-wide logistics, which we can call the *managers*, while scripts attached to the prefabs or gameobjects simply manage the gameobject it is attached to.

Scoring System

Previously in our first lab, we manage the ScoreText UI directly from the PlayerController script. We need a better way to do this: if Player scores anything, it has to notify a manager to update the score value.

Ensure that you have a UI Text score in your Scene, placed somewhere convenient in an *Screen Space - Overlay* render mode as such:



GameManager.cs

Create a new script called `GameManager.cs` , declare and implement the following:

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

public  class GameManager : MonoBehaviour
{
        public  Text score;
        private  int playerScore =  0;

        public  void  increaseScore(){
                playerScore  +=  1;
                score.text  =  "SCORE: "  +  playerScore.ToString();
        }
}
```

Obviously the method `increaseScore` will have to be called whenever the player legally kill one of the enemies. We will have eventually a few other managers in the game, such as `SpawnManager` and `PowerupManager` , and we do not want any script to be able to call methods from any manager. Therefore we need some kind of `CentralManager` that helps to call other manager methods such as this `increaseScore` so that we have less management headache.

CentralManager.cs

Create a new script called `CentralManager.cs` and implement the following:

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

// this has methods callable by players
public  class CentralManager : MonoBehaviour
{
        public  GameObject gameManagerObject;
        private  GameManager gameManager;
        public  static  CentralManager centralManagerInstance;
        void  Awake(){
                centralManagerInstance  =  this;
        }
        // Start is called before the first frame update
        void  Start()
        {
                gameManager  =  gameManagerObject.GetComponent<GameManager>();
        }

        public  void  increaseScore(){
                gameManager.increaseScore();
        }
}
```

C#: Static variable

The `CentralManager` has a reference to current GameManager instance in the scene, and this can be easily set in in the inspector. However, since every other script has to refer to the current `CentralManager` instance in the Scene for help, then a `static` variable: `centralManagerInstance` is declared, which value is set to `this` upon instantiation of `CentralManager` . This way we do not have to link up the current `CentralManager` reference to every script in the Scene.

`centralManagerInstance` is a static variable, any other script can **conveniently** refer to `this` instance of `CentralManager` using the class:

```
CentralManager.centralManagerInstance.method(parameters...)
```

Create EnemyController.cs Script for Enemy Prefab

Initially we have created `gombaEnemy` and `greenEnemy` prefabs. We now need a script to control their movements and behaviour. It is up to you how you want to implement this script, but for example let's just implement one similar to our first lab, that is to create enemies that patrol back and forth between Point A and B, where Point A and B are at at most 5 x-units away from its initial `transform.position.x`.

Here’s a sample quick implementation to move the enemy back and forth for a fixed distance. Set up the appropriate constants for `maxOffset` and `enemyPatrolTime` in `gameConstants` accordingly.

```
using System.Collections;
using UnityEngine;

public  class EnemyController : MonoBehaviour
{
        public  GameConstants gameConstants;
        private  int moveRight;
        private  float originalX;
        private  Vector2 velocity;
        private  Rigidbody2D enemyBody;

        void  Start()
        {
                enemyBody  =  GetComponent<Rigidbody2D>();

                // get the starting position
                originalX  =  transform.position.x;

                // randomise initial direction
                moveRight  =  Random.Range(0, 2) ==  0  ?  -1  :  1;

                // compute initial velocity
                ComputeVelocity();
        }

        void  ComputeVelocity()
        {
                        velocity  =  new  Vector2((moveRight) *  gameConstants.maxO
        }

        void  MoveEnemy()
        {
                enemyBody.MovePosition(enemyBody.position  +  velocity  *  Time.fixe
    }

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

Now we need to check if it collides with the Player. Ensure that the BoxCollider2D of the enemy prefabs has `isTrigger` property **checked**. Then implement `OnTriggerEnter2D` method:

```
    void  OnTriggerEnter2D(Collider2D other){
        // check if it collides with Mario
        if (other.gameObject.tag  ==  "Player"){
            // check if collides on top
            float yoffset = (other.transform.position.y  -  this.transfo
            if (yoffset  >  0.75f){
                KillSelf();
            }
            else{
                // hurt player, implement later
            }
        }
    }
```

The idea is to check if the player's `y` location is higher than the enemy's (that's how we know the Player is *stomping* the enemy from above). If yes, then we the enemy is killed:

```
void  KillSelf(){
        // enemy dies
        CentralManager.centralManagerInstance.increaseScore();
        StartCoroutine(flatten());
        Debug.Log("Kill sequence ends");
}
```

A little Coroutine called `flatten` gradually animate the enemy being flattened onto the ground as follows:

```
IEnumerator  flatten(){
        Debug.Log("Flatten starts");
        int steps =  5;
        float stepper =  1.0f/(float) steps;

        for (int i =  0; i  <  steps; i  ++){
                this.transform.localScale  =  new  Vector3(this.transform.lc

                // make sure enemy is still above ground
                this.transform.position  =  new  Vector3(this.transform.posi
                yield  return  null;
        }
        Debug.Log("Flatten ends");
        this.gameObject.SetActive(false);
        Debug.Log("Enemy returned to pool");
        yield  break;
}
```

The value `gameConstants.groundSurface` is the global y-position of a point directly above the ground. In our case, the groundSurface is right at `y = -1` .

> Note that Coroutines are not any form of multi Threading and it does not allow any kind of Parallel execution.

The debug messages are printed in this order: > * `Flatten starts` , > * `KillSequenceEnds` , > * `Flatten ends` , then > * `Enemy returned to pool` , This which shows that the method `KillSelf` ** does NOT have to wait until Coroutine flatten ends** to resume execution, thus allowing **asynchronous** execution.

However, `onTriggerEnter2D` can only be called again once `KillSelf` exits because `KillSelf` is synchronous with `OnTriggerEnter2D` .

At each `yield` in `flatten` , control is **returned** to Unity and it will still call **other functions** in `Enemy Controller` script such as `Update` and `FixedUpdate` . It is important to **never** call too many Coroutines during `Update` or `FixedUpdate` as it may cause performance issues.

To test run, create a few gameObjects a parent gameObject named `Managers` with two children (and no other components):

- GameManager with `GameManager.cs` attached
- CentralManager with `CentralManager.cs` attached

Set up the inspectors appropriately: link up `Score` Text in. `GameManager.cs` and `GameManager` gameobject in `CentralManager.cs` . Do not forget to link up `gameConstants` whenever you refer to it in the enemy prefabs or everywhere else in your code when you refer to it.

Test run and you should observe the enemies getting flattened whenever the Player lands right on top if it, and the score is increased.

Damaging Player

Create another method in `CentralManager.cs` which will be called whenever the Player collides with the enemies sideways:

```
public  void  damagePlayer(){
        gameManager.damagePlayer();
}
```

Now implement the same method in `GameManager.cs` :

```
public  void  damagePlayer(){
        OnPlayerDeath();
}
```

C#: Delegates and Events

What is `OnPlayerDeath` ? It is not a method implemented in `GameManager.cs` but it is an **event**, of which all of its **subscribers** will cast the subscribed method whenever `GameManager` calls `OnPlayerDeath()` .

Event is a special type of **delegate**. A delegate is a **reference pointer** to a method. It allows us to treat method as a variable and pass method as a variable for a callback. When a delegate gets called, it notifies all methods that **reference** the delegate.

> The basic idea behind them is exactly the same as a subscription magazine. Anyone can subscribe to the service and they will receive the update at the right time automatically.

You can declare a delegate with the `delegate` keyword and specifies its signature (return type and parameters):

```
public  delegate  returnType name(parameter1, parameter2, ...);
```

Declare the following delegate in `GameManager` .cs:

```
public  delegate  void gameEvent();
```

To allow other scripts to subscribe to this delegate, we need to create an instance of that delegate, using the `event` keyword:

```
public static event gameEvent OnPlayerDeath;
```

> Note that we can also use the delegate directly using its name: `public static gameEvent OnPlayerDeath` , but **without** the keyword `event` then `OnPlayerDeath()` can be **cast** by anyone (unless it is not `public` , but that will mean that not every other script can subscribe to it). If we want only the *owner* of the delegate to cast, such as this `GameManager` , then the `event` keyword is used.

We instantiate two events from `gameEvent` delegate. Any other script can now subscribe to this event. Open `EnemyController.cs` and implement the following under the `Start()` method.

> Remember that the event `OnPlayerDeath` is set as **static**, so we can conveniently refer to them using the classname `GameManager` instead of finding the instance reference during runtime.

```
// subscribe to player event
GameManager.OnPlayerDeath += EnemyRejoice;
```

`EnemyRejoice` is simply a method in in `EnemyController.cs` implemented as:

```
// animation when player is dead
void EnemyRejoice(){
        Debug.Log("Enemy killed Mario");
        // do whatever you want here, animate etc
        // ...
}
```

`EnemyRejoice` has the **same signature** as `delegate gameEvent` in `GameManager` , so it can subscribe to `OnPlayerDeath` event derived from `gameEvent` delegate.

You can have multiple methods subscribed to the same event (that's the point of subscription!), so open `PlayerController.cs` and implement a method that you want to be called when Mario dies:

```
void  PlayerDiesSequence(){
        // Mario dies
        Debug.Log("Mario dies");
        // do whatever you want here, animate etc
        // ...
}
```

Then subscribe to the event at `Start()` in `PlayerController.cs` :

```
GameManager.OnPlayerDeath  +=  PlayerDiesSequence;
```

Finally, when `damagePlayer()` is called in `GameManager.cs` , this will cast the **delegate** `OnPlayerDeath()` , which in turn will call `EnemyRejoice()` and `PlayerDiesSequence()` **both** as they're both subscribed to the event. The order of execution depends on the order of subscription, and they're **sequential**.

You can learn more about delegate and events from Unity tutorials here.

Now propagate these chain of function calls at the `EnemyController.cs` by implementing the **else** clause in the `OnTriggerEnter2D` function we did earlier:

```
void  OnTriggerEnter2D(Collider2D other){
        // check if it collides with Mario
        if (other.gameObject.tag  ==  "Player"){
                // check if collides on top
                float yoffset = (other.transform.position.y  -  this.transform.posit
                if (yoffset  >  0.75f){
                        KillSelf();
                }
                else{
                        // hurt player
                        CentralManager.centralManagerInstance.damagePlayer();
                }
        }
  }
```

Test run and you should at least see the Debug messages printed out whenever Player collides with the enemy in any way except from stomping it from above.

Checkoff information

Create a new event that's to be casted when `increaseScore()` in GameManager is called, such that it results in spawning of one new enemy. The class that should subscribe to this new event is `SpawnManager.cs` we created earlier.

PowerupManager.cs

Create a new script called `PowerupManager.cs` which will be used to manage the powerups that Mario can *collect* and *consume* anytime at will.

Before that, open the Consumable Mushroom prefab that you have created in the previous lab and another copy. Rename them both into `RedMushroom` and `OrangeMushroom` and select two different sprites (you can name it however you want actually, it does not matter. We will only use these names for easy referencing in this tutorial).

You should already have a script that controls its direction of patrol as per previous lab. You might now want to add a **collision** check with Player, and a state `bool collected` to indicate that the powerup is **collected** so that **you don't need to move the mushroom anymore when collected**. Also add some kind of visual feedback upon collision. Here's a simple example: the mushroom will slightly enlarge and then scaled in.

C#: Interface

It is very common in a game to have various types of powerups, but they should have common methods that will be called by other scripts such as `cast` or `consume`, etc. To do this more uniformly, we can utilise an `interface`.

Create a new script called `ConsumableInterface.cs`, where we can declare method signatures:

```
using UnityEngine;

public  interface ConsumableInterface{
        void  consumedBy(GameObject player);
}
```

Now create two more scripts, `RedMushroom.cs` and `OrangeMushroom.cs` that will implement this interface:

```
using System.Collections;
using UnityEngine;

public  class RedMushroom : MonoBehaviour, ConsumableInterface
{
        public  Texture t;
        public  void  consumedBy(GameObject player){
                // give player jump boost
                player.GetComponent<PlayerController>().upSpeed  +=  10;
                StartCoroutine(removeEffect(player));
        }

        IEnumerator  removeEffect(GameObject player){
                yield  return  new  WaitForSeconds(5.0f);
                player.GetComponent<PlayerController>().upSpeed  -=  10;
        }
}
```

As you can see, the RedMushroom, when consumed will give the player jump boost for 5 seconds.

The OrangeMushroom will give the player a speed boost for 5 seconds:

```
using System.Collections;
using UnityEngine;

public  class OrangeMushroom : MonoBehaviour, ConsumableInterface
{
        public  Texture t;
        public  void  consumedBy(GameObject player){
                // give player jump boost
                player.GetComponent<PlayerController>().maxSpeed  *=  2;
                StartCoroutine(removeEffect(player));
        }

        IEnumerator  removeEffect(GameObject player){
                yield  return  new  WaitForSeconds(5.0f);
```

```
            player.GetComponent<PlayerController>().maxSpeed  /=  2;
        }
    }
```

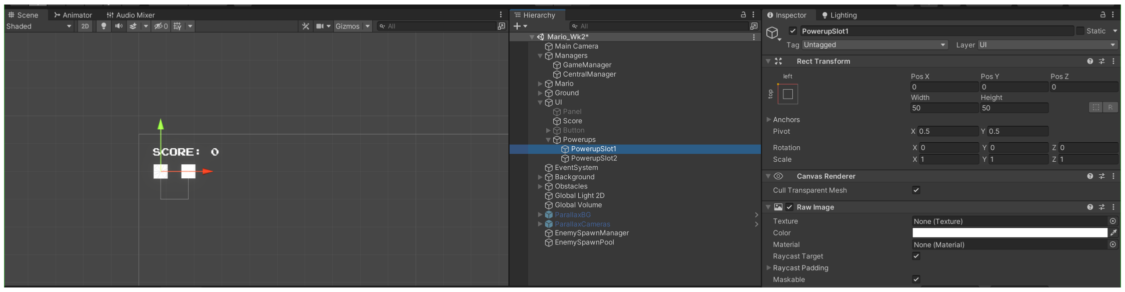
Attach `RedMushroom.cs` and `OrangeMushroom.cs` to your respective consumable prefabs (the prefab that's spawned and patrolling around after hitting ? boxes) respectively.

Set `Texture t` in the inspector to be a picture you want to indicate in the UI later on on the availability of this powerup.

PowerupManager.cs

PowerupUI

Now what is left is to indicate how many powerups are usable, and to bind "consume" of powerup with a key. Create a UI gameobject as follows:



```
Hierarchy
Mario_Wk2*
  Main Camera
  Managers
    GameManager
    CentralManager
  Mario
  Ground
  UI
    Panel
    Score
    Button
    Powerups
      PowerupSlot1
      PowerupSlot2
  EventSystem
  Background
  Obstacles
  Global Light 2D
  Global Volume
  ParallaxBG
  ParallaxCameras
  EnemySpawnManager
  EnemySpawnPool
```

```
Inspector                Lighting
✓ PowerupSlot1                              Static
Tag Untagged             Layer UI
Rect Transform
left       Pos X     Pos Y     Pos Z
top        0         0         0
           Width     Height
           50        50
Anchors
Pivot      X 0.5     Y 0.5
Rotation   X 0       Y 0       Z 0
Scale      X 1       Y 1       Z 1
Canvas Renderer
Cull Transparent Mesh   ✓
✓ Raw Image
Texture          None (Texture)
Color
Material         None (Material)
Raycast Target   ✓
Raycast Padding
Maskable         ✓
```

The two children objects: PowerupSlot1 and PowerupSlot2 have RawImage as a component. We will load RedMushroom texture at Slot2 and OrangeMushroom texture at Slot1 whenever Mario collected them (but not yet consumed).

Create a script called `PowerUpManager.cs`:

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

public  class PowerUpManager : MonoBehaviour
{
        public  List<GameObject> powerupIcons;
        private  List<ConsumableInterface> powerups;

        // Start is called before the first frame update
        void  Start()
        {
                powerups  =  new  List<ConsumableInterface>();
                for (int i =  0; i<powerupIcons.Count; i++){
                        powerupIcons[i].SetActive(false);
                        powerups.Add(null);
                }
        }
}
```

Here we store a list of existing powerups *interfaces*, and initialise them as *null* at first because Mario hasn't collected anything. We also disable the UI placeholder.

Write two methods to add and remove powerups from the list:

```
public  void  addPowerup(Texture texture, int index, ConsumableInterface i){
        Debug.Log("adding powerup");
        if (index  <  powerupIcons.Count){
                powerupIcons[index].GetComponent<RawImage>().texture  =  texture;
                powerupIcons[index].SetActive(true);
                powerups[index] =  i;
        }
}

public  void  removePowerup(int index){
        if (index  <  powerupIcons.Count){
        powerupIcons[index].SetActive(false);
        powerups[index] =  null;
        }
}
```

C#: Switch statements

Finally, add these two methods in `PowerUpManager.cs` to consume the powerups when key Z or X is pressed. You can set your own key binding. This is just an example:

```
void  cast(int i, GameObject p){
        if (powerups[i] !=  null){
                powerups[i].consumedBy(p); // interface method
                removePowerup(i);
        }
}

public  void  consumePowerup(KeyCode k, GameObject player){
        switch(k){
                case  KeyCode.Z:
                        cast(0, player);
                        break;
                case  KeyCode.X:
                        cast(1, player);
                        break;
                default:
                        break;
        }
}
```

`consumedBy` is a method that's guaranteed to be implemented by any class having this `ConsumableInterface`. **It is a good way to implement a standardized method without any ambiguity on how to call it**, not to mention that it is neat to have such standardized interface for similar object types and abstract out the implementation of its effects within the method. It will be quite a nightmare if each powerup has its own method signature.

Now obviously `consumePowerup` and `addPowerup` should be called from outside `PowerUpManager.cs` script. Remember that we do not necessarily want any other script to call any Managers as they please, so we need to add these methods to `CentralManager.cs` and let other Scripts call them via `CentralManager`:

```
// add reference to PowerupManager
public  GameObject powerupManagerObject;
private  PowerUpManager powerUpManager;
```

```
// instantiate in start
powerUpManager  =  powerupManagerObject.GetComponent<PowerUpManager>();

public  void  consumePowerup(KeyCode k, GameObject g){
        powerUpManager.consumePowerup(k,g);
}

public  void  addPowerup(Texture t, int i, ConsumableInterface c){
        powerUpManager.addPowerup(t, i, c);
}
```

The script that will call `addPowerup` is both RedMushroom.cs and OrangeMushroom.cs, attached to the consumable prefab and called when they collide with Mario:

```
void  OnCollisionEnter2D(Collision2D col)
{
        if (col.gameObject.CompareTag("Player")){
                // update UI
                CentralManager.centralManagerInstance.addPowerup(t, index, this);
                GetComponent<Collider2D>().enabled  =  false;
        }
}
```

where `index` can be set to `0` or `1` depending on which key, `z` or `x` you choose to activate the powerups.

The script that will call `userPowerup` is `PlayerController.cs` , under the `Update()` method:

```
if (Input.GetKeyDown("z")){
        CentralManager.centralManagerInstance.consumePowerup(KeyCode.Z,this.gameObje
}

if (Input.GetKeyDown("x")){
        CentralManager.centralManagerInstance.consumePowerup(KeyCode.X,this.gameObje
}
```

As a summary:

- When Mario collides with Red/Orange Mushroom, `OnCollision2D` callback in `Red/OrangeMushroom.cs` will call `addPowerup` in CentralManager, and pass the reference for Texture `t` and `this instance` eventually to `PowerupManager`. This will show the UI that a powerup is available.
- When Mario presses `z` or `x`, it will call `consumePowerup` in CentralManager, and pass the reference of itself. This will eventually reach `PowerupManager`'s `consumePowerup` and it will call the `consumedBy` method implemented in both `RedMushroom` and `OrangeMushroom` any of these powerups are available.

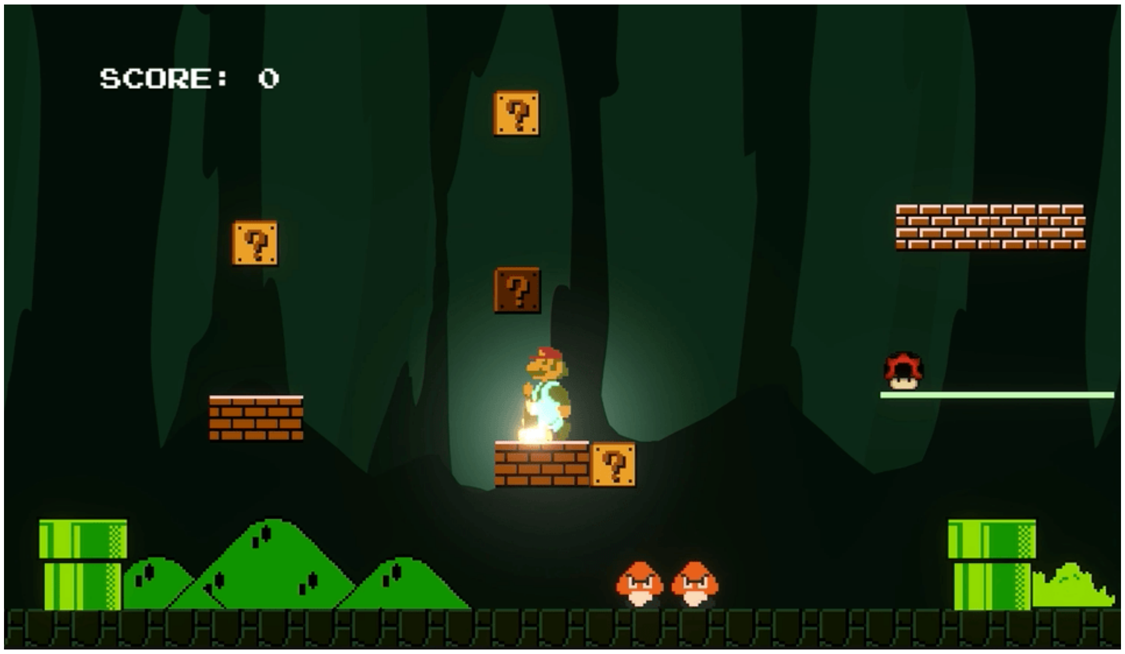
Checkoff

The gif summarises what you need for checkoff. There's a lot of things that need to be done so read each point carefully.

1. **Use the ObjectPooler** and *spawn* 2 enemies at a time. The Player can "Kill" the enemy any way you want, such as stepping from the top. This will **increase score** and spawn **one new enemy**. Make sure you show that you use an object pooler by recording the *Hierarchy* as well.
2. Create some "**interactive**" **bricks** to collect coins or whatever rewards that will **increase score** but *spawn one new enemy*. It is up to you to determine how many max enemies can be present at a time. In the gif below, we limit total number of gomba enemies and green enemies to be 5.
3. Player will "**die**" under certain conditions (up to you), and this will cause the enemy to "**rejoice**". Both actions of player dying and enemy rejoicing must be clear to the player. In the gif sample below, player will die if it collides with the enemy from anywhere except from the top. The enemies have this cute little dance when the player dies, and player's death is animated as well.
4. Use **AudioMixer** in your project, in any way you want.
5. **Powerups**: create two different powerups from anywhere e.g: the question boxes. Player can "collect" it as shown, and "cast" it later. Bind the "casting" to

two distinct keys, signifying that the player *consumes* it. The effect should disappear after a fixed number of seconds. For example, the red mushroom allows the player to jump higher for 5 seconds. The max number of powerups that can be collected can be fixed to 1 for each type for simplicity.

6. Usage of **ScriptableObject** to store some unchanged constants like enemy speed, fixed patrol locations, etc. This won't be visible easily in the game, so you need to record yourself clicking to the scriptable object instance and **show** that the values are used by other scripts.



It is alright if your screen recording for this checkoff takes more time.

Next

In the next lesson, we will further polish our game by learning how to transition between Scenes and implementing a Combo Manager. We will also learn another way to perform asynchronous execution using async and await, which is fundamentally different from Coroutines, along with some issues to watch out for when using them such as *memory leak*.

NEXT POST
Unity for Toddlers