CUHK
Computer Science
FYP 2020-21
Final Report

FYP Project CHC 2001: Game Development

Kwan Ting Fung 1155110979
Li Kin Sing 1155108893

December, 2020

Table of Contents

1. Game Description

1.1. Basic Idea

Our goal is to create a third-person 3D dungeon game, which should look like Dark Souls from Fromsoftware, but with skills-tree like a classic MMORPG. Player has to defeat the monsters and find out the path to the level boss in that maze. After defeating monsters, the player gains EXP or gets equipment randomly. After defeating the level boss, the player will be allowed to open the next maze. At a certain point of EXP, the player should get the skill point for upgrading his skills-tree.

Skills-tree includes different sword skills or magic to strengthen their character. Each player could enjoy their unique different fighting style.

However, usually upgrading skills and player stats are twin-blades, being fun in the beginning, and becoming bored at late game in classic RPG, meanwhile the bosses stats keeps rising, which becomes a stat-chasing game always. Thus, we try to have a game design where players could re-assign their skill-points any time, so that players could enjoy the whole combat system rather than chasing for stats.

The other goal is to have better fighting experience in PVE. In my point of view, Dark Souls, Sekiro have a slower attack speed, so it is just a matter of correct attack timing. Therefore, we hope that our game could offer another way of combat style, which is speedy, and a variety of magics.

1.2. Gameplay

Players will have basic equipment at the beginning to explore the dungeon. Collect different equipment and fight with monsters to level up. Every floor is a maze and there are monsters on the map.

Whenever a player dies, he/she will return to the start point of that floor, and the map will be reset. Players could loot the dead body of the monsters or treasure boxes. At the end of each floor, the player will need to fight with the boss and get to the next floor.

Furthermore, players can learn different sword skills or magic to strengthen their character. Each player can have a different fighting style.

1.3. Game Engine

There are many game engines nowadays. It is pointless to choose which engine we should use if we just want to present the game idea. Hence, the choice of game engine should be focused on the ease of use, efficiency, license, servers...etc.

We should choose Unity as our project's game engine, mainly with 3 aspects, graphics, scripts, and resources.

Reason to choose Unity

For graphics, although we really hope that our game could have pretty 3A models to fulfil our fantasy, we are not 3d artist and could not made those stuff in fast, so the possible ways are use free model from asset store or use low-poly style models, which we could found out lots of free resources, and able for us making it by ourselves.

For scripts, Since our team is formed by two programmers, we believe coding is much faster than visual programming (blueprints) in efficiency and logic design so visual programming won't be our consideration. Moreover, Unity is better as it is using C#, while Unreal is using C++, unless we are experienced C++ programmer, C# is better for bug checking.

For resources, Unity is much much better than Unreal, like communities, tutorials, supports, 3d resources, animations, etc. And there are so many plug-ins that are free to add in, higher flexibility. If we choose Unreal, just simply not enough support, too hard to learn or handle, although Unreal is open source, we don't have enough knowledge and experience to do change to it.

Most of our game features could be implemented by all game engines as we know, or maybe code by ourselves. In short, the choice is mainly about availability and quantity of free materials.

Comparing Unity to Unreal, Unity has games like Human: Fall Flat, Rick and Morty: Virtual Rick-ality, Ori ; Unreal has many 3A games like Tekken7, Gears5. It is pretty obvious that Unity should fit our projects in terms of man-power and the ease of choosing free materials, and Unreal will be much better unless we understand the source code of it and be able to rewrite the functions.

For other engines

1. cryEngine is recently free of charge, but it mainly focuses on fps games, or visual effects, and open-world environments. But we are not going to make huge open-world, fps, and it is a really small user base and too less knowledgeable people to ask for help.

2. godot is an open-source game engine, the advantage is free of charge and we could see all source code. Their scripting language is GDscript, which is a python-like language, and should be easy to use. It has a better UI development, better scene system, node system than Unity. However, godot has a worse rendering for 3D graphics, smaller user base and community, not even one showcase. So it might not be a good choice to gain experience for game development.

Final decision

One very important point is, there are very few tutorials and forums when compared with those famous game engines(Unreal/Unity). Moreover, there are few assets in their store, which means you might need to create many assets or plug-in by yourself.

Thus, for a 2-man project, Unity should be the best choice.

1.4. Classification

- Bonfire

Bonfire is a save-point actually. But it needs to be found by the player and activate it. After being activated, the woods are ignited and give a huge lighting. Then, each time the player dies, the player can reborn from the last visited Bonfire and restore his statistics. The player could use one bonfire to transit to another bonfire. Thus, this should be similar to Souls' like game.

2. Development Schedule

2.1. Task

<u>Functional Minimum (Works To be done in this semester, the basic game)</u>

It should be our prototype of the game, and presented at the end of the fall semester. It should contain:

- level editing
- animated player character
- animated enemies
- basic melee fight system
- UI, menu, and options
- basic sounds and lighting

*All of these will not have so much visual effects or delightful textures.

<u>Low Target (If times available, some of which will be done, otherwise, next semester)</u>

It should be done in the spring term of semester, likes updating some visual effects, or additional modes or functions.

- some skills-tree with animation
- sound for all stuff
- additional monster types
- storyline
- delightful lighting

<u>High Target (Considering in the next semester)</u>

If times allowed, a visual update and more variety of enemies should be made.

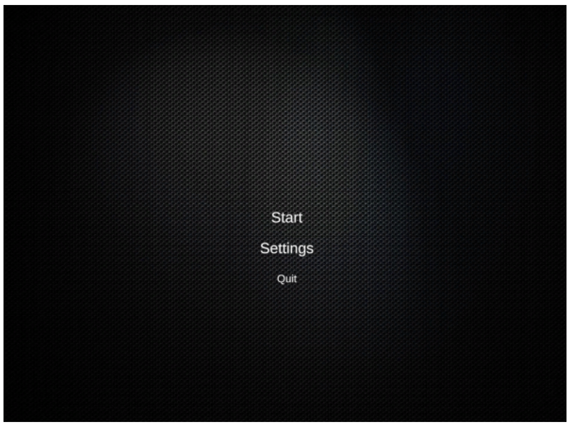
- playable map for multi-players
- different shader techniques
- characters with better collision handling
- variety in combat system
- more maps
- playable with a matching system

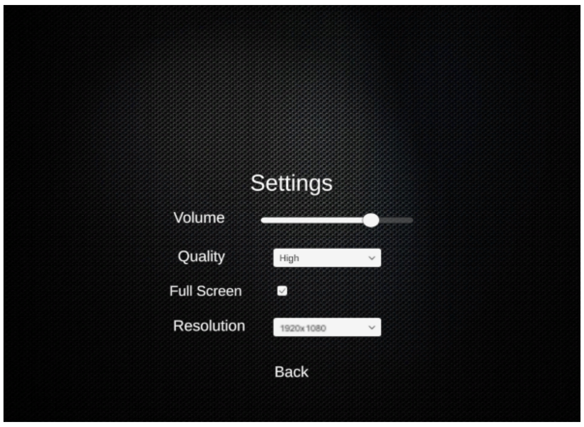
2.2. Schedule

Sep: Planning + Basic Menu + Basic AI + basic animated enemies + Camera Control
Oct: Basic animated Character + Save & Load system + melee system + UI
Nov: Level Editing + Sound effect + lighting
Dec: 2/12 Term-end Deadline

3. Implementaion:

1. Basic Main Menu




Settings

Volume

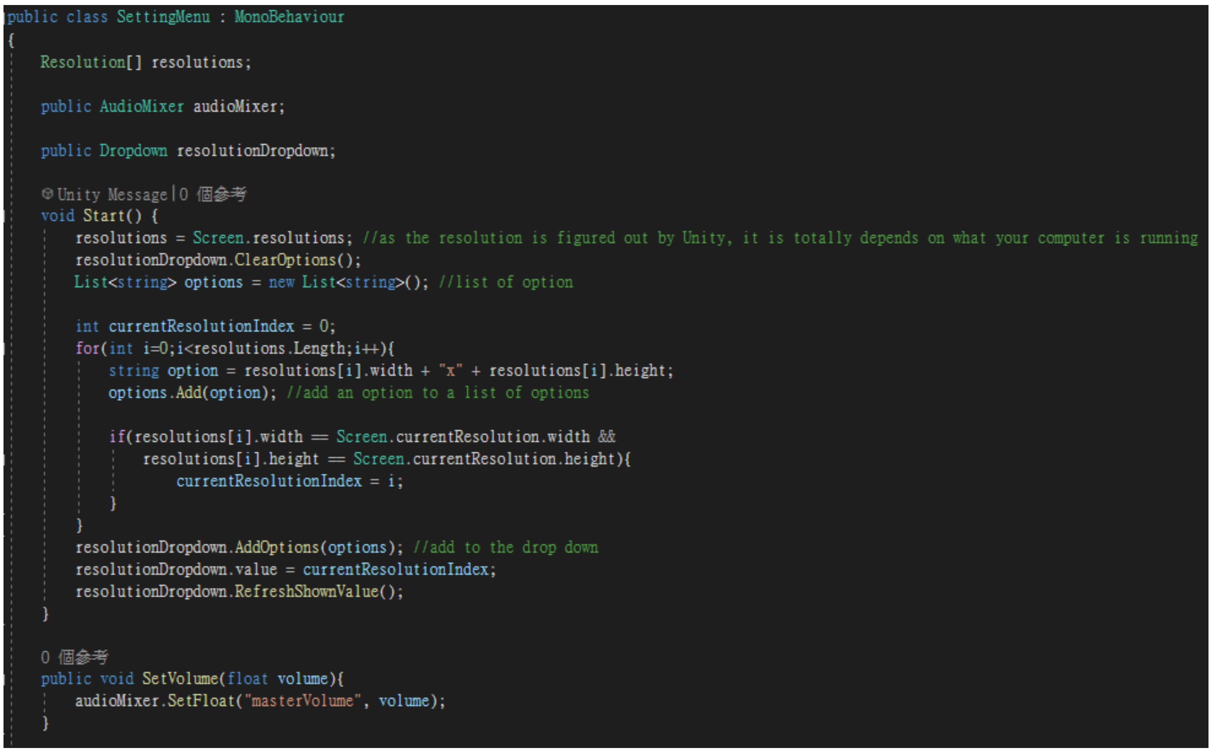
Quality High

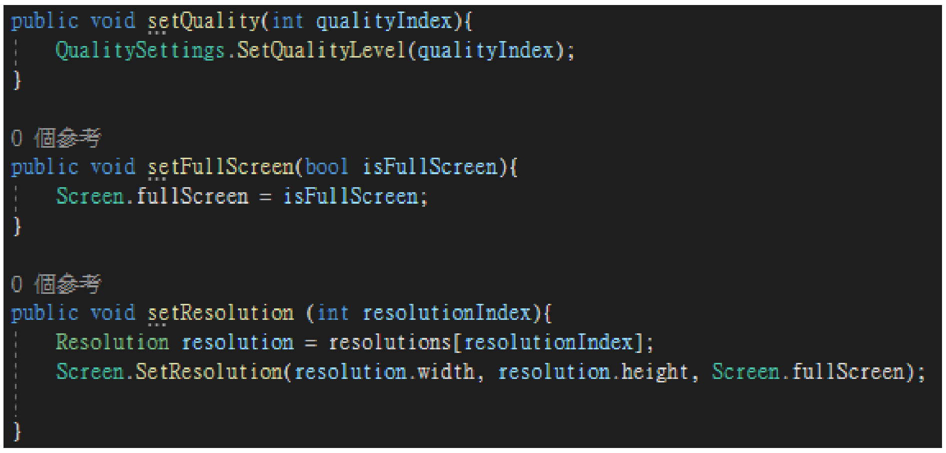
Full Screen

Resolution 1920x1080

Back

The basic main menu includes all basic functions that are essential to a game, i.e. start game, quit game, change graphic and volume settings.

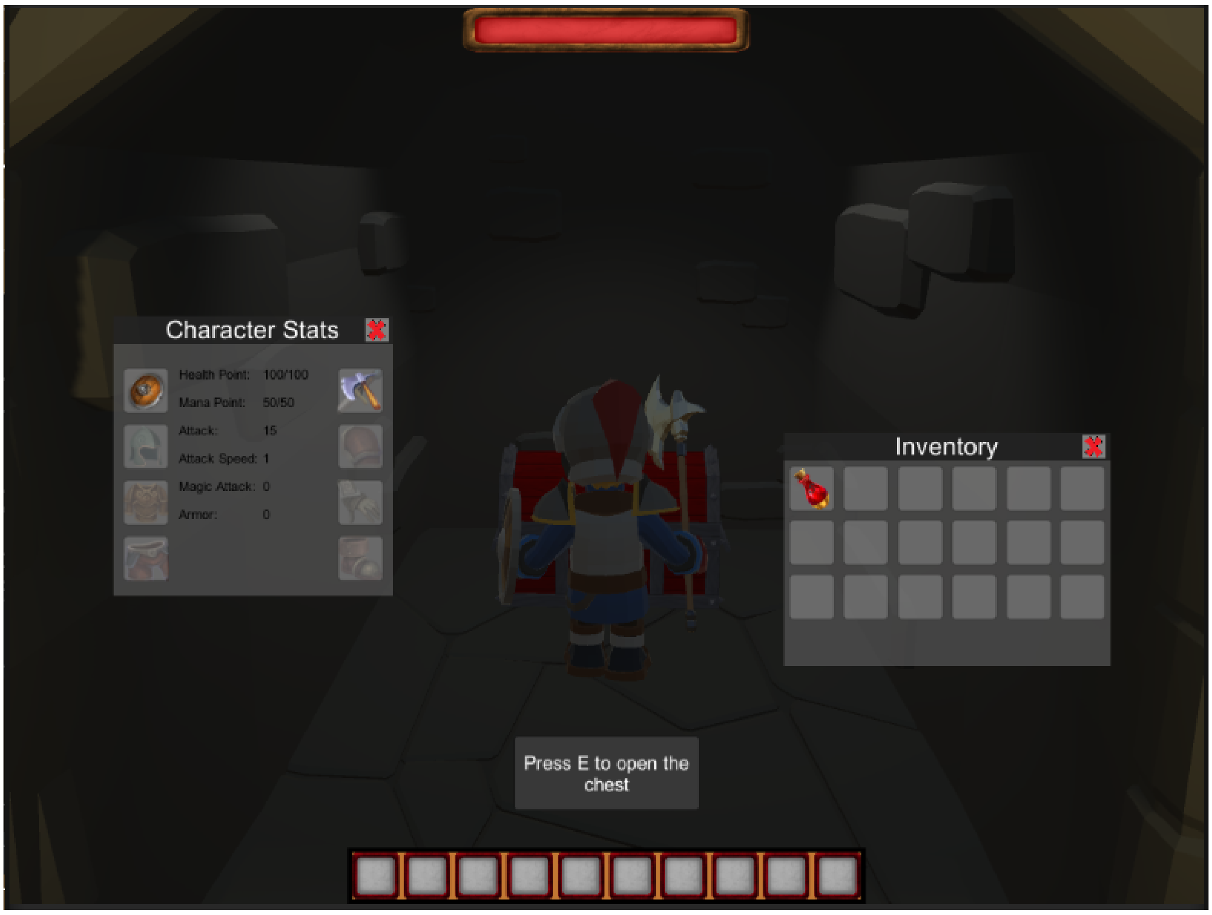

```
public class SettingMenu : MonoBehaviour
{
    Resolution[] resolutions;

    public AudioMixer audioMixer;

    public Dropdown resolutionDropdown;

    ⊕Unity Message|0 個參考
    void Start() {
        resolutions = Screen.resolutions; //as the resolution is figured out by Unity, it is totally depends on what your computer is running
        resolutionDropdown.ClearOptions();
        List<string> options = new List<string>(); //list of option

        int currentResolutionIndex = 0;
        for(int i=0;i<resolutions.Length;i++){
            string option = resolutions[i].width + "x" + resolutions[i].height;
            options.Add(option); //add an option to a list of options

            if(resolutions[i].width == Screen.currentResolution.width &&
                resolutions[i].height == Screen.currentResolution.height){
                    currentResolutionIndex = i;
            }
        }
        resolutionDropdown.AddOptions(options); //add to the drop down
        resolutionDropdown.value = currentResolutionIndex;
        resolutionDropdown.RefreshShownValue();
    }

    0 個參考
    public void SetVolume(float volume){
        audioMixer.SetFloat("masterVolume", volume);
    }
```



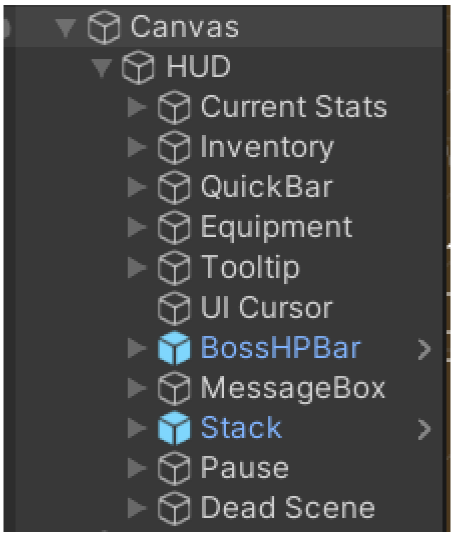
```
public void setQuality(int qualityIndex){
    QualitySettings.SetQualityLevel(qualityIndex);
}

0 個參考
public void setFullScreen(bool isFullScreen){
    Screen.fullScreen = isFullScreen;
}

0 個參考
public void setResolution (int resolutionIndex){
    Resolution resolution = resolutions[resolutionIndex];
    Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);

}
```

However, the graphic quality settings should be reserved and used later in spring semester for the final progress, i.e. optimization. The quality would be set inside Unity settings and defines different game textures or materials as different quality indexes.

2. Basic UI



Health Point: 100/100
Mana Point: 50/50
Attack: 15
Attack Speed: 1
Magic Attack: 0
Armor: 0

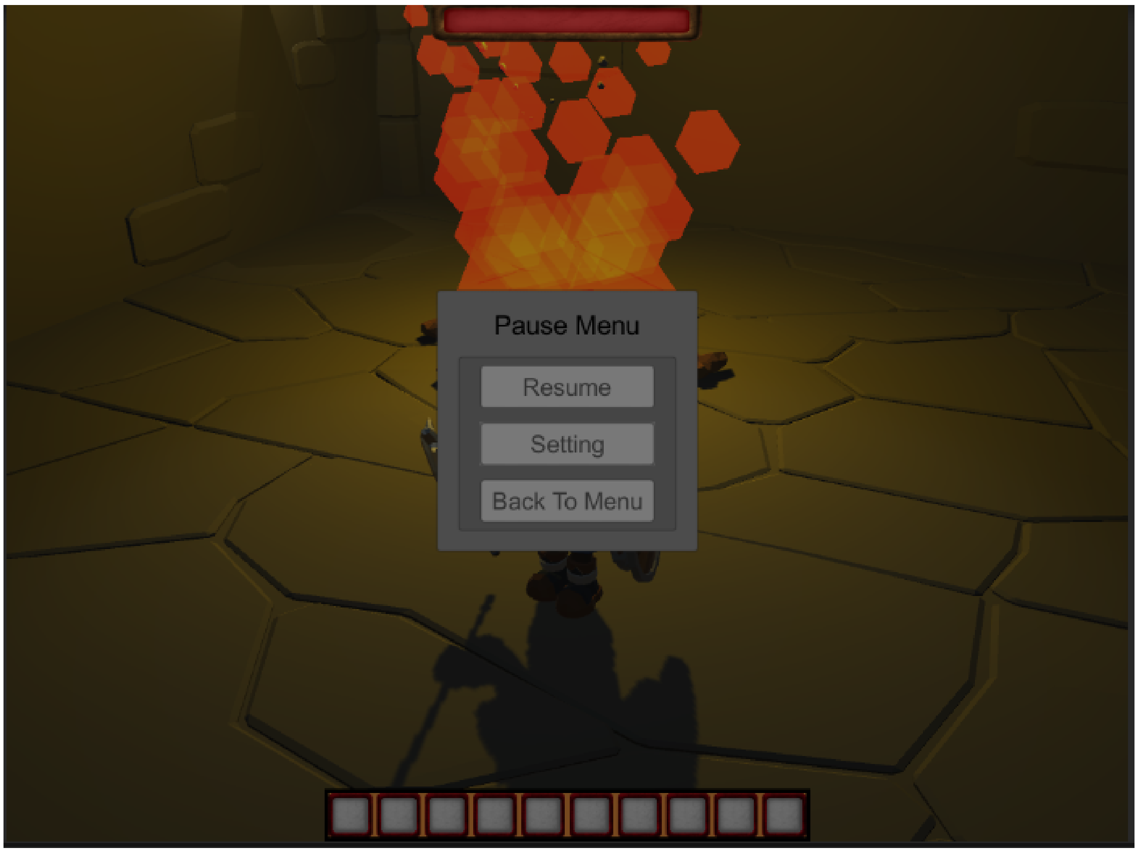


```
Canvas
  HUD
    Current Stats
    Inventory
    QuickBar
    Equipment
    Tooltip
    UI Cursor
    BossHPBar
    MessageBox
    Stack
    Pause
    Dead Scene
```

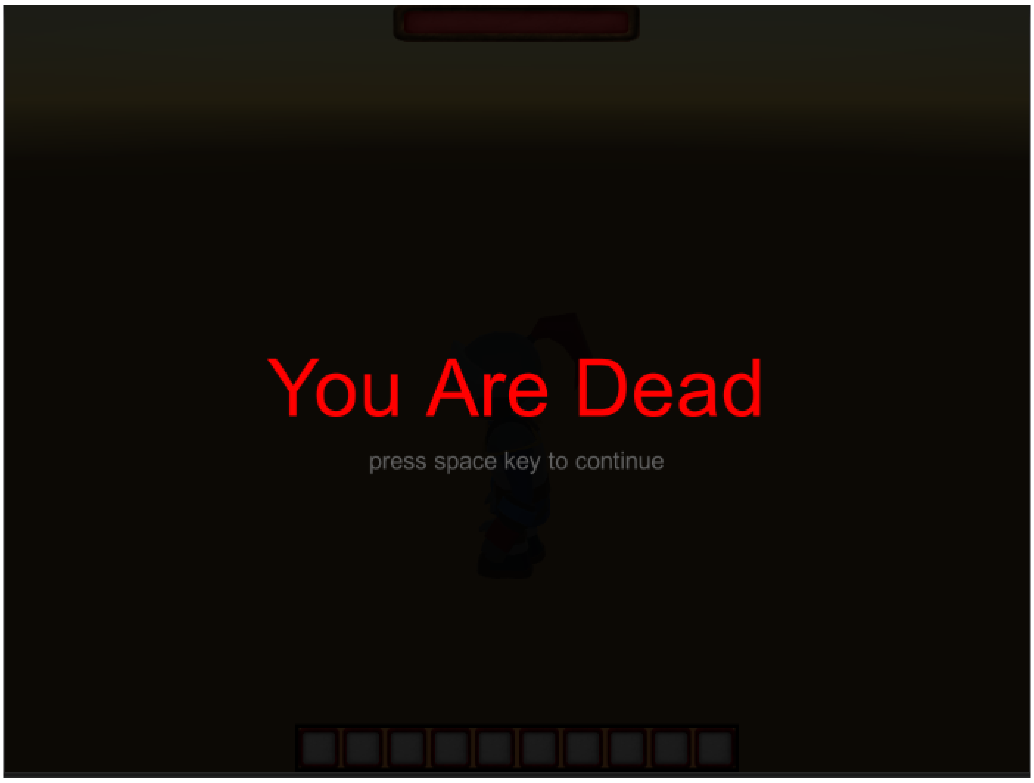
The basic UI contains the following:

- **Current Stats**: The health point bar that will show the health percentage information of your character in real time by using the slider.
- **Inventory**: An item container contains all the items (including usable items and equipment) and you can use the item by right clicking on its icon. Player can also drag and close the window by dragging the title bar and clicking the close icon. This can be opened and closed by using key: I. (asset)

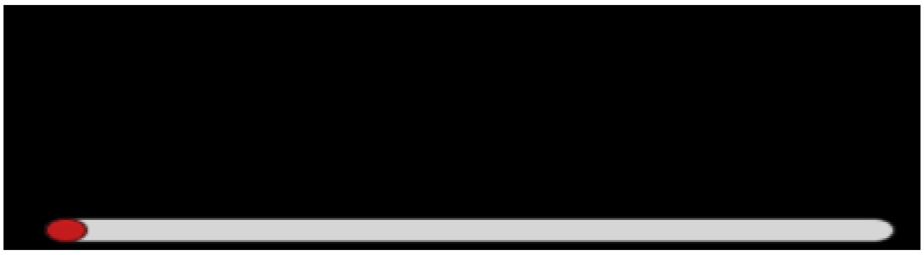
- **QuickBar**: A quickBar of the inventory which you can drag and drop the item in inventory to here and use it. You can use the shortcut (1,2,3,4..0) to use the item in your inventory. (asset)
- **Equipment**: A window shows all the equipped items and stats of your character in real time. This can be opened and closed by using key: C.
- **Tooltip**: A window shows about the detail of the item that your mouse is pointing to. (asset)
- **UI Cursor**: Use for dragging the item to another slot in the inventory system. (asset)
- **BossHPBar**: Only show when you enter the boss fight zone and show the current Boss health information.
- **Message Box**: It has two modes to display the information. The first one is to show a message to give a hint that what player can press in order to interact with the object. The second is to show what they have got from the chest.
- **Stack**: Use for inventory system. (asset)



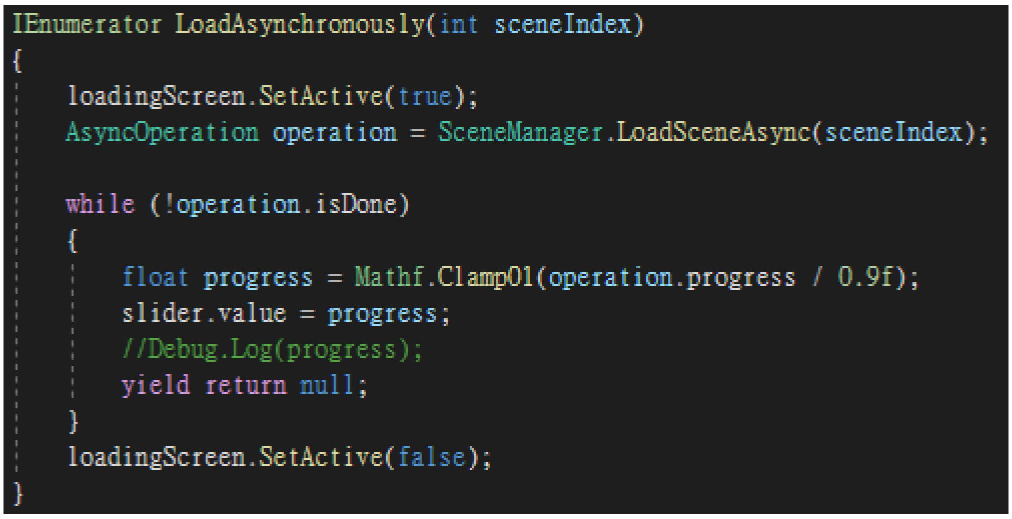
- **Pause**: Show the pause menu and the game pause at the same time, and the player can choose to resume the game, open the setting, or quit the game.



- **Dead Scene**: This UI will display when the player is dead.
- **Loading Screen:** This UI will display the progress of loading levels or the main menu.



Loading screen is implemented by coroutine, for monitoring the progress.

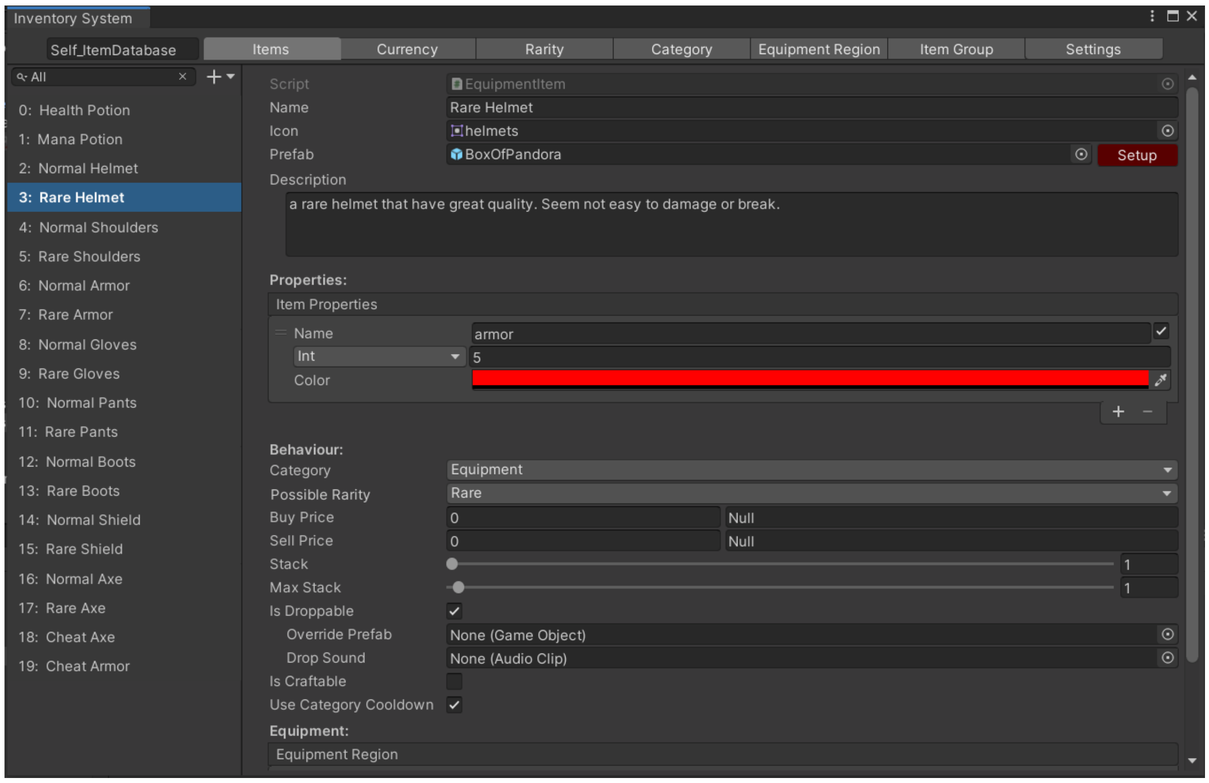


```
IEnumerator LoadAsynchronously(int sceneIndex)
{
    loadingScreen.SetActive(true);
    AsyncOperation operation = SceneManager.LoadSceneAsync(sceneIndex);

    while (!operation.isDone)
    {
        float progress = Mathf.Clamp01(operation.progress / 0.9f);
        slider.value = progress;
        //Debug.Log(progress);
        yield return null;
    }
    loadingScreen.SetActive(false);
}
```

3. Inventory system and database (asset)

The inventory system and the database are supported by the unity asset store (Item & Inventory System) provided by Devion Games.

Here show the database for this system:


Inventory System

Self_ItemDatabase | Items | Currency | Rarity | Category | Equipment Region | Item Group | Settings

All

0: Health Potion
1: Mana Potion
2: Normal Helmet
3: Rare Helmet
4: Normal Shoulders
5: Rare Shoulders
6: Normal Armor
7: Rare Armor
8: Normal Gloves
9: Rare Gloves
10: Normal Pants
11: Rare Pants
12: Normal Boots
13: Rare Boots
14: Normal Shield
15: Rare Shield
16: Normal Axe
17: Rare Axe
18: Cheat Axe
19: Cheat Armor

Script	EquipmentItem
Name	Rare Helmet
Icon	helmets
Prefab	BoxOfPandora
Description	a rare helmet that have great quality. Seem not easy to damage or break.

Properties:

Item Properties

Name	armor ✓
Int	5
Color	🟥

Behaviour:

Category	Equipment	
Possible Rarity	Rare	
Buy Price	0	Null
Sell Price	0	Null
Stack	1	
Max Stack	1	
Is Droppable	✓	
Override Prefab	None (Game Object)	
Drop Sound	None (Audio Clip)	
Is Craftable		
Use Category Cooldown	✓	

Equipment:

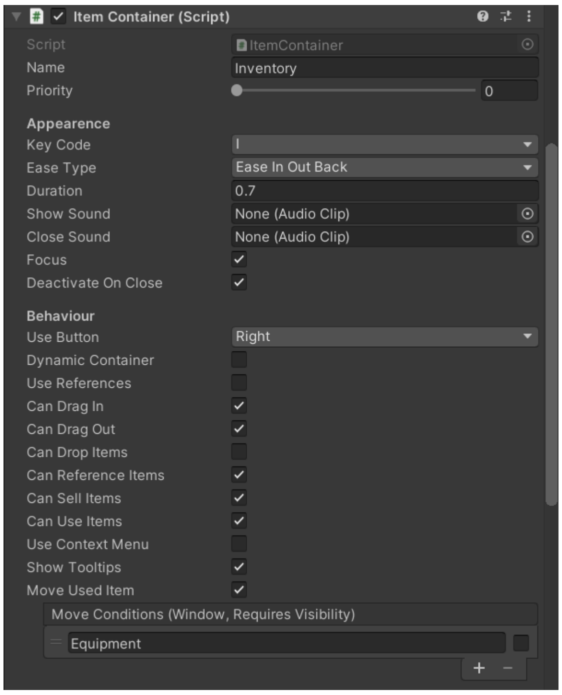
Equipment Region

We can define items in this database by filling all the details above for every item. For example, above is a screenshot of an equipment item.

- The behaviour needs to interact with other scripts in the asset. In our project, we mainly use this asset for the database and inventory use, so some behaviour is useless for us. The Only consideration is the stack amount of the item.
- The Equipment Region is defined by us too and used to interact with equipment slots to show the current equipment information of the character.

If you are defining a usable item, you can even define some function that will execute when you use the item.

The inventory needs to attach the item container(script) and item collection(script) to indicate the feature of the inventory.

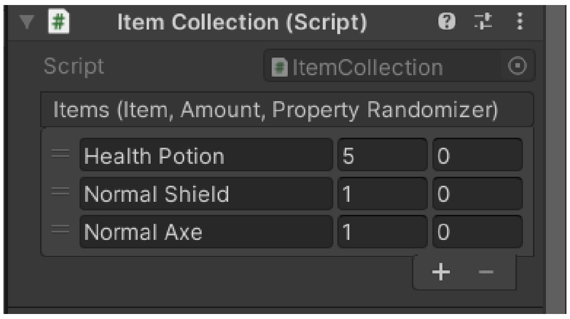


Item Container (Script)

Script	ItemContainer
Name	Inventory
Priority	0
Appearence	
Key Code	I
Ease Type	Ease In Out Back
Duration	0.7
Show Sound	None (Audio Clip)
Close Sound	None (Audio Clip)
Focus	✓
Deactivate On Close	✓
Behaviour	
Use Button	Right
Dynamic Container	
Use References	
Can Drag In	✓
Can Drag Out	✓
Can Drop Items	
Can Reference Items	✓
Can Sell Items	✓
Can Use Items	✓
Use Context Menu	
Show Tooltips	✓
Move Used Item	✓

Move Conditions (Window, Requires Visibility)

Equipment	



Item Collection (Script)

Script: ItemCollection

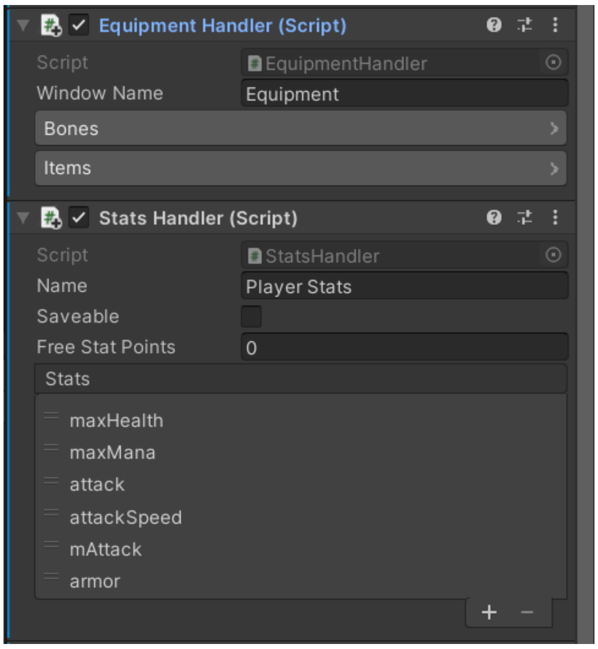
Items (Item, Amount, Property Randomizer)

Health Potion	5	0
Normal Shield	1	0
Normal Axe	1	0

The property for the equipment will interact with other scripts that attached to your character below:

```
private void OnAddItem(Item item, Slot slot)
{
    foreach (ObjectProperty property in item.GetProperties()) {
        if (property.SerializedType == typeof(int) || property.SerializedType == typeof(float)) {
            float value = System.Convert.ToSingle(property.GetValue());
            SendMessage("AddModifier", new object[] { property.Name, value, (value <= 1f && value >= -1f) ? 1 : 0, item }, SendMessageOptions.DontRequireReceiver);
        }else
        if(property.SerializedType == typeof(string)) {
            GetComponentInChildren<PlayerInventory>().EquipItem(System.Convert.ToString(property.GetValue()));
        }
    }
}

private void OnRemoveItem(Item item, int amount, Slot slot)
{
    foreach (ObjectProperty property in item.GetProperties()) {
        if (property.SerializedType == typeof(int) || property.SerializedType == typeof(float)) {
            SendMessage("RemoveModifiersFromSource", new object[] { property.Name, item }, SendMessageOptions.DontRequireReceiver);
        }
        else
        if (property.SerializedType == typeof(string)) {
            EquipmentItem mItem = item as EquipmentItem;
            if (mItem.Region.Count <= 1) {
                if (mItem.Region[0].Name == "Left hand") {
                    GetComponentInChildren<PlayerInventory>().UnEquipItem("Left hand");
                }
                else
                if (mItem.Region[0].Name == "Right hand") {
                    GetComponentInChildren<PlayerInventory>().UnEquipItem("Right hand");
                }
            }
            else {
                GetComponentInChildren<PlayerInventory>().UnEquipItem("Both hand");
            }
        }

    }
}
```

When the player equip an equipment item, the database will call the OnAddItem function in the Equipment Handler.
The function will search the property name in the Stat Handler that we defined before and add the value that is defined in the item's property. If the item is a weapon or shield, the script will call the EquipItem method inside the Playerinventory.
When the player unequips the equipment item, the stat in the Stat Handler will change back to the init value. If the item is a weapon or shield, the script will call the UnEquipItem method inside the Playerinventory.



Equipment Handler (Script)

Script	EquipmentHandler
Window Name	Equipment

Bones

Items

Stats Handler (Script)

Script	StatsHandler
Name	Player Stats
Saveable	☐
Free Stat Points	0

Stats

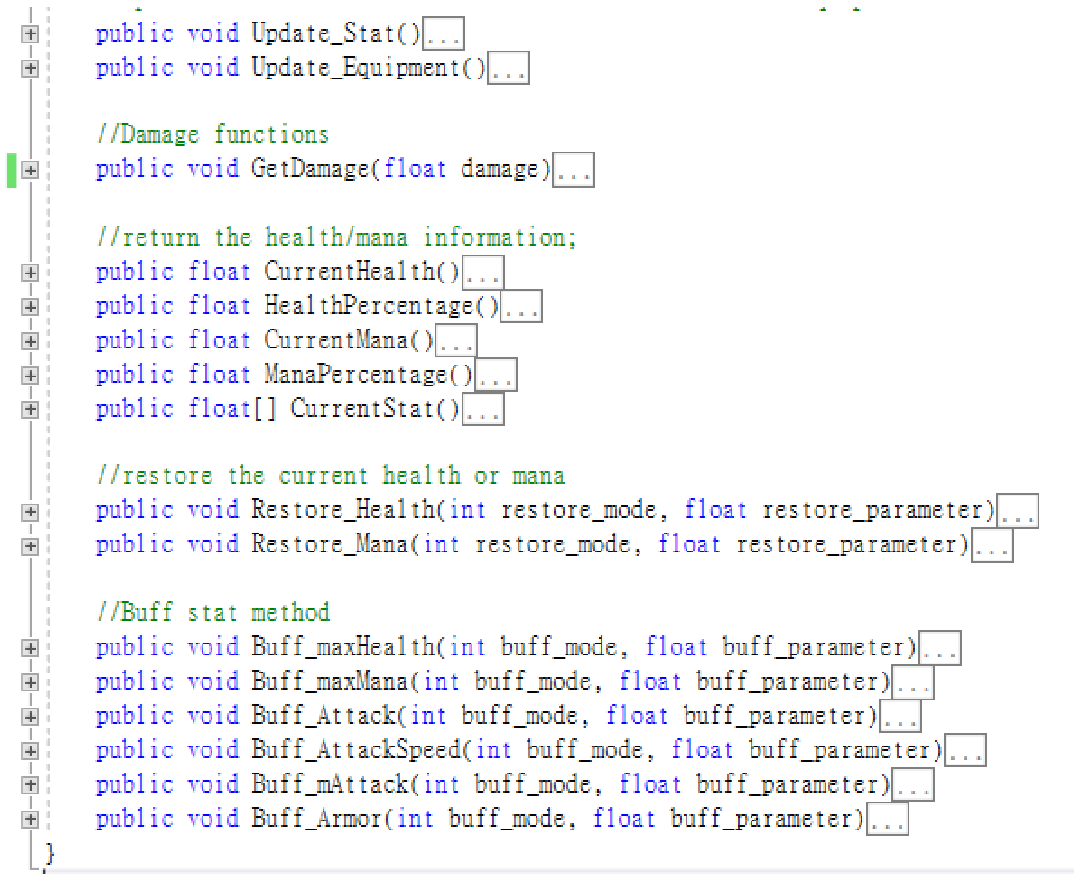
- maxHealth
- maxMana
- attack
- attackSpeed
- mAttack
- armor

4. Stat system

The stat system simply includes the statistics of the character that can be used to calculate different events of the character. I defined the following six basic stat for the character:

- maxHealth
- maxMana
- attack
- attackSpeed
- mAttack (magic attack)
- armor

These six attributes can interact with other objects and script to manipulate the character related functions. There are many functions for it to communicate with other object:



```
    public void Update_Stat()...
    public void Update_Equipment()...

    //Damage functions
    public void GetDamage(float damage)...

    //return the health/mana information;
    public float CurrentHealth()...
    public float HealthPercentage()...
    public float CurrentMana()...
    public float ManaPercentage()...
    public float[] CurrentStat()...

    //restore the current health or mana
    public void Restore_Health(int restore_mode, float restore_parameter)...
    public void Restore_Mana(int restore_mode, float restore_parameter)...

    //Buff stat method
    public void Buff_maxHealth(int buff_mode, float buff_parameter)...
    public void Buff_maxMana(int buff_mode, float buff_parameter)...
    public void Buff_Attack(int buff_mode, float buff_parameter)...
    public void Buff_AttackSpeed(int buff_mode, float buff_parameter)...
    public void Buff_mAttack(int buff_mode, float buff_parameter)...
    public void Buff_Armor(int buff_mode, float buff_parameter)...
}
```

Let me introduce the functions with different usage:

- Update data
  There are four kinds of data in the system, they are the basic, equipment, buff and the final calculated properties. The Update_Stat() and Update_Equipment() functions are used to update the final and equipment properties. Therefore we can always get the latest data of the character's stat.
- Damage
  There are functions to handle the damage calculation between the enemies and player. For example, damage from player to enemy will invoke attack/mAttack

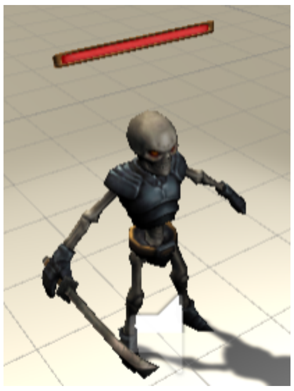
data of player and the armor data of the enemy. So when a damage event happens, these two functions are called.

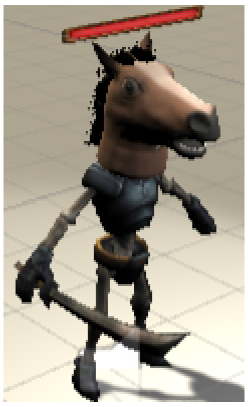
- Return stats information
  To show the private stats of the character to do the calculation or display function.
- Restore health/mana function
  Used for potion items to recover the character stat.
- Buffs functions
  Set the additional buffs information for the character and recalculation the current stats of the player's character.

5. Basic animated enemies

<u>normal enemy</u>

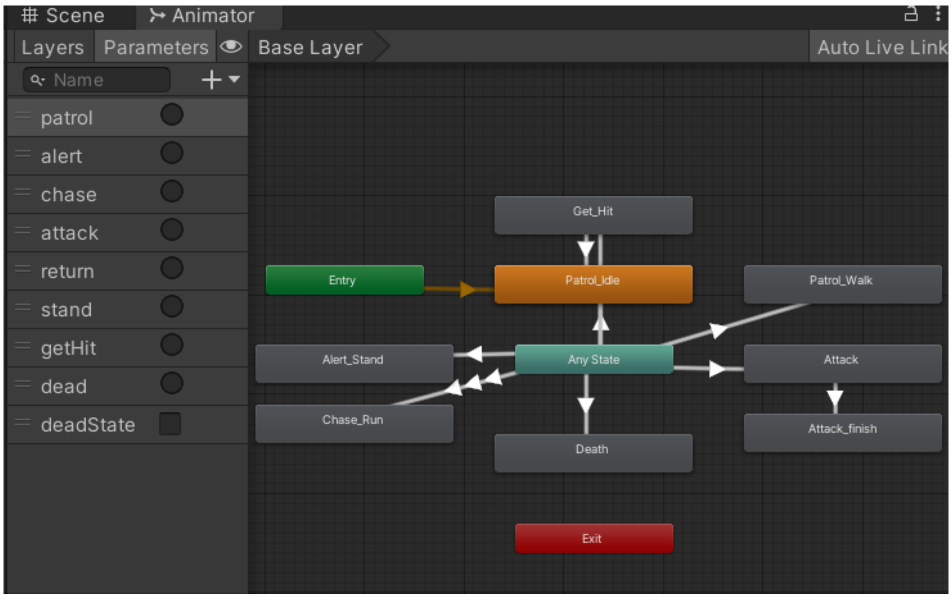
There are 2 kinds of animated enemies designed in the project. They are all grabbed in unity asset stores. Following are the first kind of enemies (normal monster):





They are skeleton, borisMonster and mummy respectively. The monster wearing the horse head is like the elite of that monster, they will be much stronger and do more damage to the player.

Since they are just some normal enemies in the game, so i just simply implement the animator like the following and use the enemy AI script to control it:

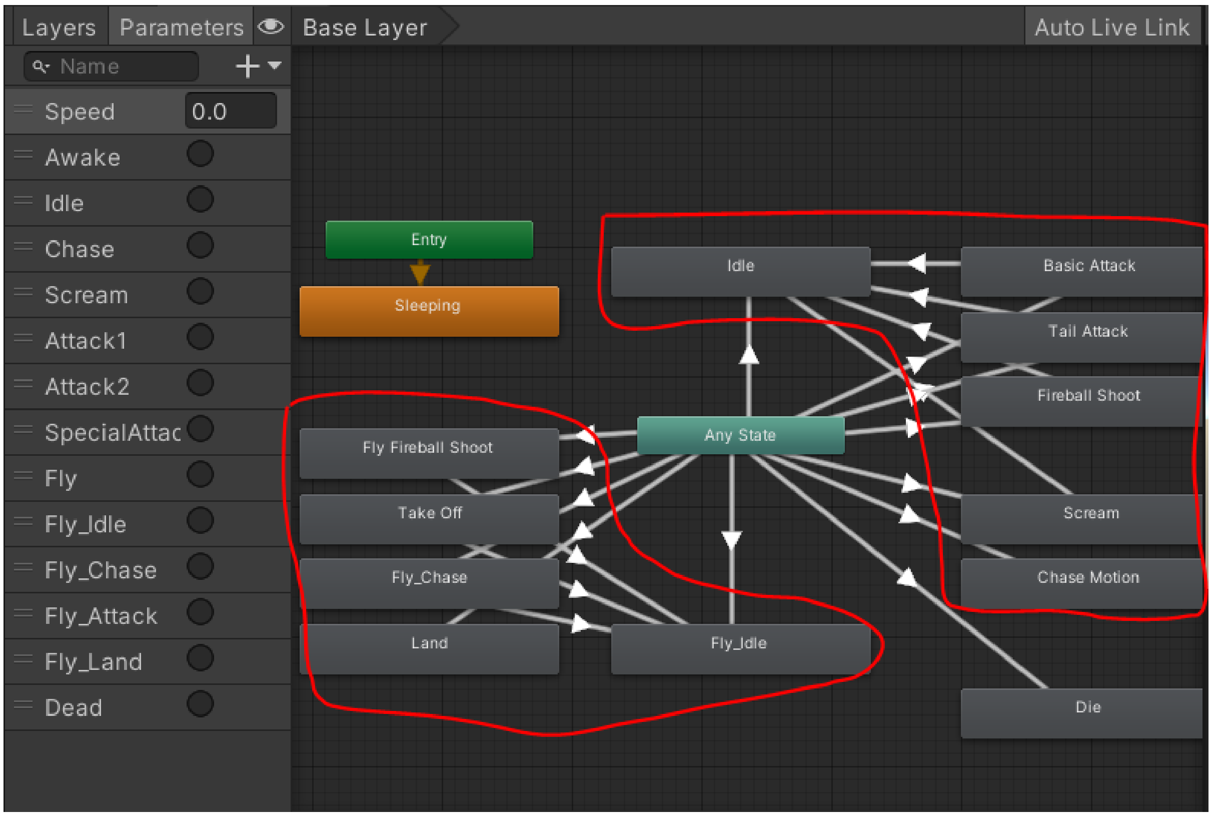


All normal enemies should have the same parameters name and same state but different animations.

Boss

The second kind of enemy is the Boss, and it look like this:

Since it is a boss, he has more attack patterns. The animator becomes much more complicated in order to do the action.



There are two sets of actions which are related to the different action state of the boss (flying and walking) (circle by red pen). The Boss will first go to the sleeping action until I send some trigger to change its action state. More detail will be introduced in the Basic enemy AI part.

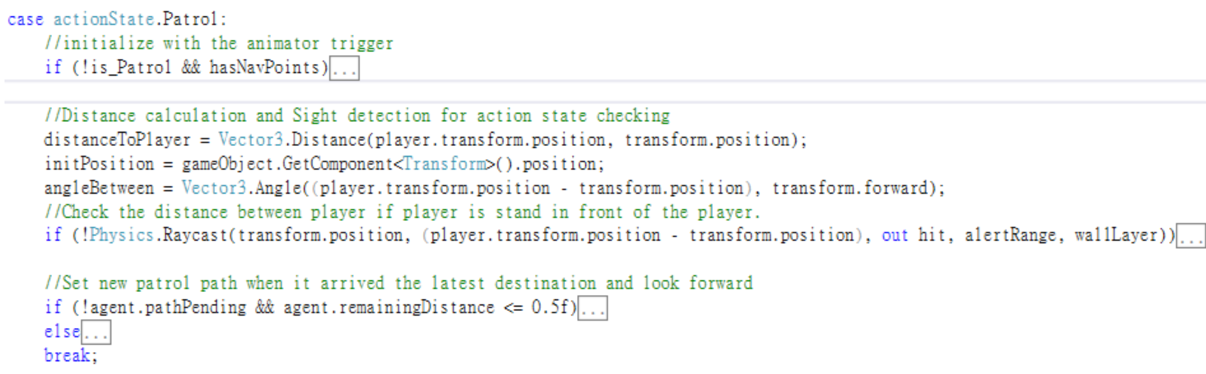
6. Basic enemy AI

I have created 2 different AI scripts for those two kinds of enemies mentioned before. Let me introduce them one by one in terms of their action states.

Normal enemy

All the normal enemy share the same action state as following:

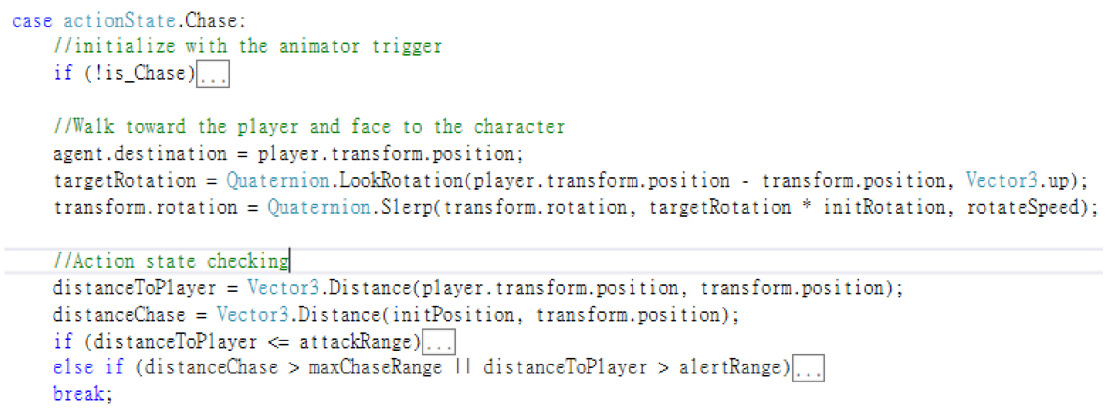
- Patrol



```
case actionState.Patrol:
    //initialize with the animator trigger
    if (!is_Patrol && hasNavPoints)...

    //Distance calculation and Sight detection for action state checking
    distanceToPlayer = Vector3.Distance(player.transform.position, transform.position);
    initPosition = gameObject.GetComponent<Transform>().position;
    angleBetween = Vector3.Angle((player.transform.position - transform.position), transform.forward);
    //Check the distance between player if player is stand in front of the player.
    if (!Physics.Raycast(transform.position, (player.transform.position - transform.position), out hit, alertRange, wallLayer))...

    //Set new patrol path when it arrived the latest destination and look forward
    if (!agent.pathPending && agent.remainingDistance <= 0.5f)...
    else...
    break;
```

The patrol action state is the initial state of the monster. it will initialize the animator of the monster and do the state checking. if it is still inside this action state, the code will check whether the gameObject arrived at the patrol point or not. if it arrived, it would stay at the point for a while and start the patrol to the next defined point.

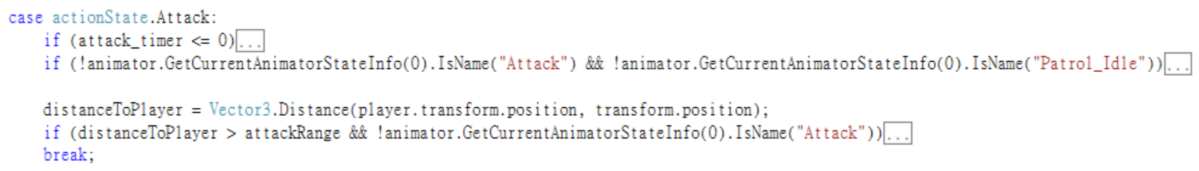
- Chase



```
case actionState.Chase:
    //initialize with the animator trigger
    if (!is_Chase)...

    //Walk toward the player and face to the character
    agent.destination = player.transform.position;
    targetRotation = Quaternion.LookRotation(player.transform.position - transform.position, Vector3.up);
    transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation * initRotation, rotateSpeed);

    //Action state checking
    distanceToPlayer = Vector3.Distance(player.transform.position, transform.position);
    distanceChase = Vector3.Distance(initPosition, transform.position);
    if (distanceToPlayer <= attackRange)...
    else if (distanceChase > maxChaseRange || distanceToPlayer > alertRange)...
    break;
```

Monster will change to this chase state when the player gets closer to the monster in the alert state. The monster starts chasing the player and checks whether it gets into the attack range or not. Also if the monster chases too far away from the original position, it will go back there and return to its patrol state.

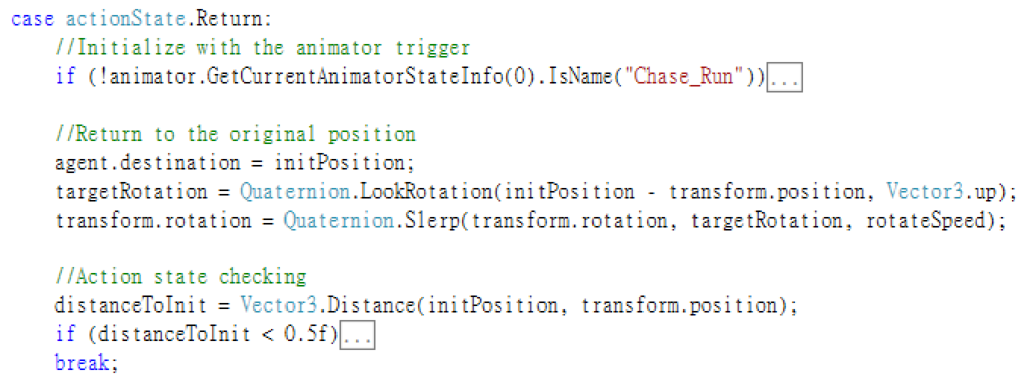
- Attack



```
case actionState.Attack:
    if (attack_timer <= 0)...
    if (!animator.GetCurrentAnimatorStateInfo(0).IsName("Attack") && !animator.GetCurrentAnimatorStateInfo(0).IsName("Patrol_Idle"))...

    distanceToPlayer = Vector3.Distance(player.transform.position, transform.position);
    if (distanceToPlayer > attackRange && !animator.GetCurrentAnimatorStateInfo(0).IsName("Attack"))...
    break;
```

Monsters start attacking the player when the player gets into its attack range. The only thing the monster needs to do in this state is to do the action state checking and attack again if the cooldown of attack is finished. if the player gets far away again, then it returns to the chase state and stops attacking.

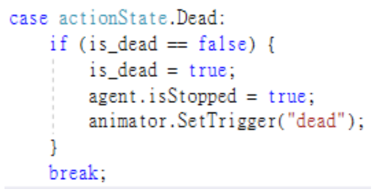
- <u>Return</u>



```
case actionState.Return:
    //Initialize with the animator trigger
    if (!animator.GetCurrentAnimatorStateInfo(0).IsName("Chase_Run"))...

    //Return to the original position
    agent.destination = initPosition;
    targetRotation = Quaternion.LookRotation(initPosition - transform.position, Vector3.up);
    transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, rotateSpeed);

    //Action state checking
    distanceToInit = Vector3.Distance(initPosition, transform.position);
    if (distanceToInit < 0.5f)...
    break;
```

The monster starts to return to its original point.

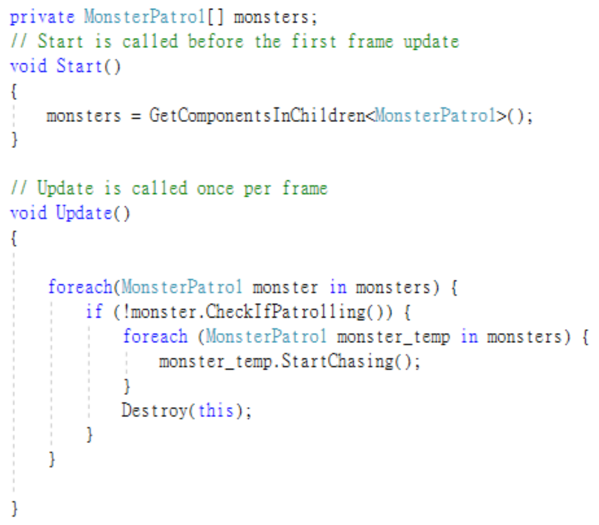
- <u>Dead</u>



```
case actionState.Dead:
    if (is_dead == false) {
        is_dead = true;
        agent.isStopped = true;
        animator.SetTrigger("dead");
    }
    break;
```

The monster health drops below 0 and plays the dead animation and doesn't move anymore.

Since the normal monster always works as a group, when any of them see the player, all of them should know me and start chasing me and attack me immediately. Here is how it works:

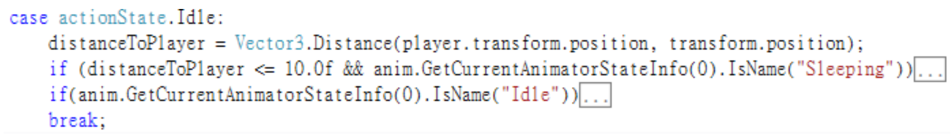


```
private MonsterPatrol[] monsters;
// Start is called before the first frame update
void Start()
{
    monsters = GetComponentsInChildren<MonsterPatrol>();
}

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

    foreach(MonsterPatrol monster in monsters) {
        if (!monster.CheckIfPatrolling()) {
            foreach (MonsterPatrol monster_temp in monsters) {
                monster_temp.StartChasing();
            }
            Destroy(this);
        }
    }

}
```

We first group a set of monsters and place them under an empty object. Use this code to get all the monsters in that group and get their action state. If anyone is changed to chasing mode, then all the monsters start chasing the player.

Boss

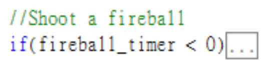
- Idle



```
case actionState.Idle:
    distanceToPlayer = Vector3.Distance(player.transform.position, transform.position);
    if (distanceToPlayer <= 10.0f && anim.GetCurrentAnimatorStateInfo(0).IsName("Sleeping"))[...]
    if(anim.GetCurrentAnimatorStateInfo(0).IsName("Idle"))[...]
    break;
```

The boss will always sleep until the player enters the boss zone, so this state aims to do the action state checking.

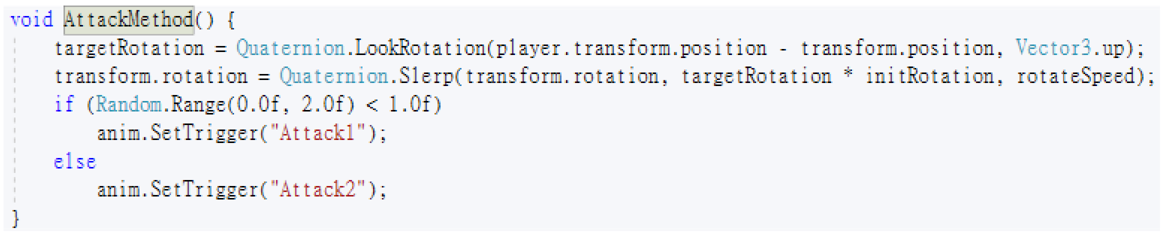
- Chase



```
//Shoot a fireball
if(fireball_timer < 0)[...]
```

The chase action state is similar with the normal enemy chase state. The main difference is that the boss may shoot a fireball to damage the player.

- Attack



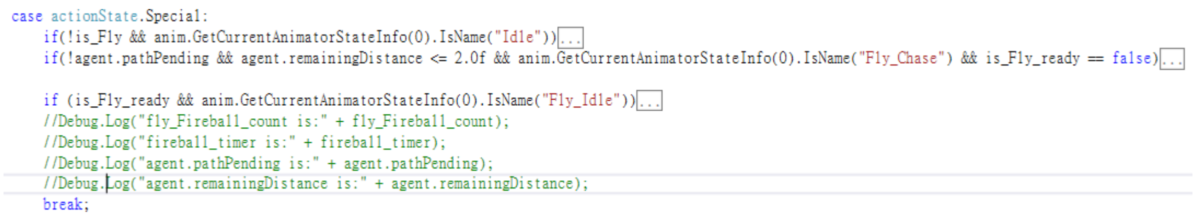
```
void AttackMethod() {
    targetRotation = Quaternion.LookRotation(player.transform.position - transform.position, Vector3.up);
    transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation * initRotation, rotateSpeed);
    if (Random.Range(0.0f, 2.0f) < 1.0f)
        anim.SetTrigger("Attack1");
    else
        anim.SetTrigger("Attack2");
}
```

- The attack state is also something similar to the normal enemy attack state but it has two different attack actions. The script will randomly choose one and do the animation.

- Dead

This state is the same as the normal enemy dead state.

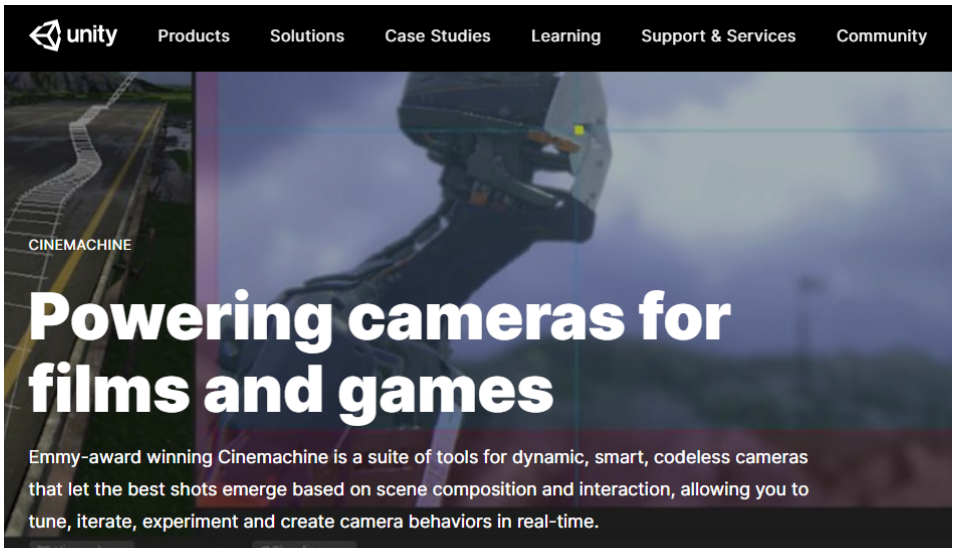
- Special



```
case actionState.Special:
    if(!is_Fly && anim.GetCurrentAnimatorStateInfo(0).IsName("Idle"))[...]
    if(!agent.pathPending && agent.remainingDistance <= 2.0f && anim.GetCurrentAnimatorStateInfo(0).IsName("Fly_Chase") && is_Fly_ready == false)[...]

    if (is_Fly_ready && anim.GetCurrentAnimatorStateInfo(0).IsName("Fly_Idle"))[...]
    //Debug.Log("fly_Fireball_count is:" + fly_Fireball_count);
    //Debug.Log("fireball_timer is:" + fireball_timer);
    //Debug.Log("agent.pathPending is:" + agent.pathPending);
    //Debug.Log("agent.remainingDistance is:" + agent.remainingDistance);
    break;
```

The boss will come into this state when its health drops below 50%. the boss will fly to the sky and go to the four corners of the room. It will shoot three fireballs at each corner and land on the ground. Players need to walk around and survive from this special mode and wait for it to land.

7. Camera Control

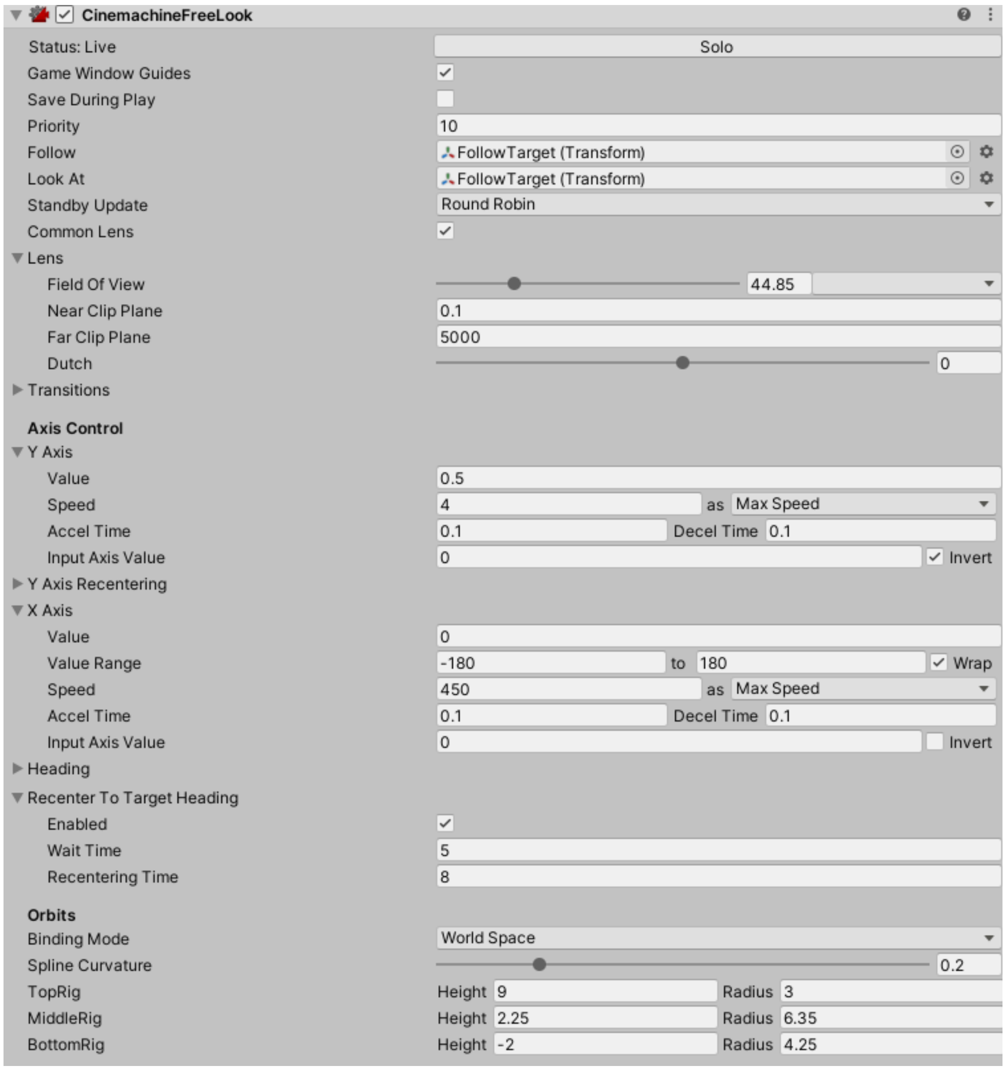


CINEMACHINE

Powering cameras for films and games

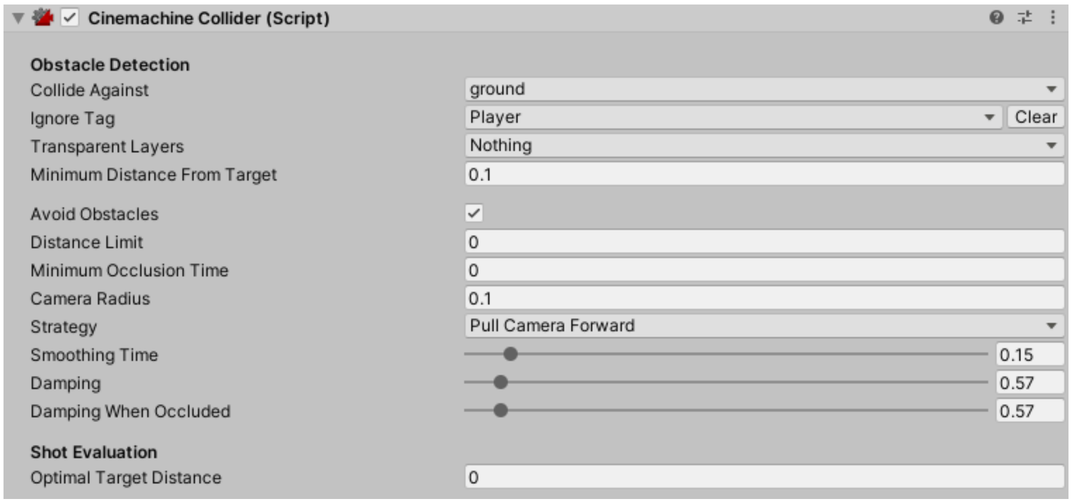
Emmy-award winning Cinemachine is a suite of tools for dynamic, smart, codeless cameras that let the best shots emerge based on scene composition and interaction, allowing you to tune, iterate, experiment and create camera behaviors in real-time.

For camera control, we import Cinemachine, a Unity-registry package. The cinemachine offers almost all stuff we need for a game, i.e. following target, axis control, rig, damping, camera collision, etc.



CinemachineFreeLook

Status: Live	Solo
Game Window Guides	✓
Save During Play	
Priority	10
Follow	FollowTarget (Transform)
Look At	FollowTarget (Transform)
Standby Update	Round Robin
Common Lens	✓
Lens	
Field Of View	44.85
Near Clip Plane	0.1
Far Clip Plane	5000
Dutch	0
Transitions	
Axis Control	
Y Axis	
Value	0.5
Speed	4 as Max Speed
Accel Time	0.1 Decel Time 0.1
Input Axis Value	0 ✓ Invert
Y Axis Recentering	
X Axis	
Value	0
Value Range	-180 to 180 ✓ Wrap
Speed	450 as Max Speed
Accel Time	0.1 Decel Time 0.1
Input Axis Value	0 Invert
Heading	
Recenter To Target Heading	
Enabled	✓
Wait Time	5
Recentering Time	8
Orbits	
Binding Mode	World Space
Spline Curvature	0.2
TopRig	Height 9 Radius 3
MiddleRig	Height 2.25 Radius 6.35
BottomRig	Height -2 Radius 4.25


Cinemachine Collider (Script)

Obstacle Detection

Collide Against	ground
Ignore Tag	Player
Transparent Layers	Nothing
Minimum Distance From Target	0.1
Avoid Obstacles	✓
Distance Limit	0
Minimum Occlusion Time	0
Camera Radius	0.1
Strategy	Pull Camera Forward
Smoothing Time	0.15
Damping	0.57
Damping When Occluded	0.57

Shot Evaluation

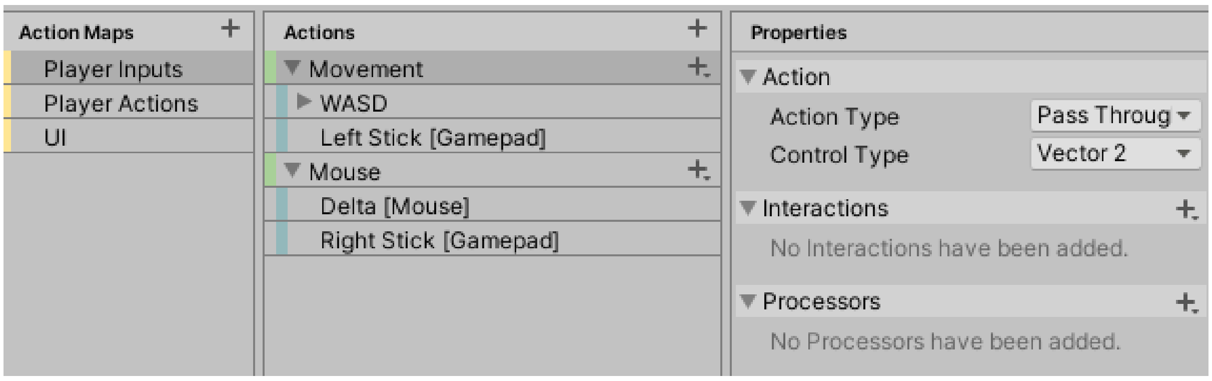
Optimal Target Distance	0

Although we have attempted to try implementing cameras by ourselves in the first week, after we found Cinemachine, we decided to use this as this is made by Unity, that means it should not have so many bugs, and the camera is way more smoother than we think, and it provides different strategy methods for updates and camera collision handling.

8. Basic animated main character

8.1. Input system

In this fyp, I try to use the newest input system in Unity. Basically two reasons, the old input system will be removed in later versions of Unity, thus, I think we need to learn the newest input system in advance. The second reason is, it is designed to separate the logical input our game code responds to from the physical actions the user takes, so that we have easier switching between different input devices.


Action Maps: Player Inputs, Player Actions, UI

Actions:
- Movement
  - WASD
  - Left Stick [Gamepad]
- Mouse
  - Delta [Mouse]
  - Right Stick [Gamepad]

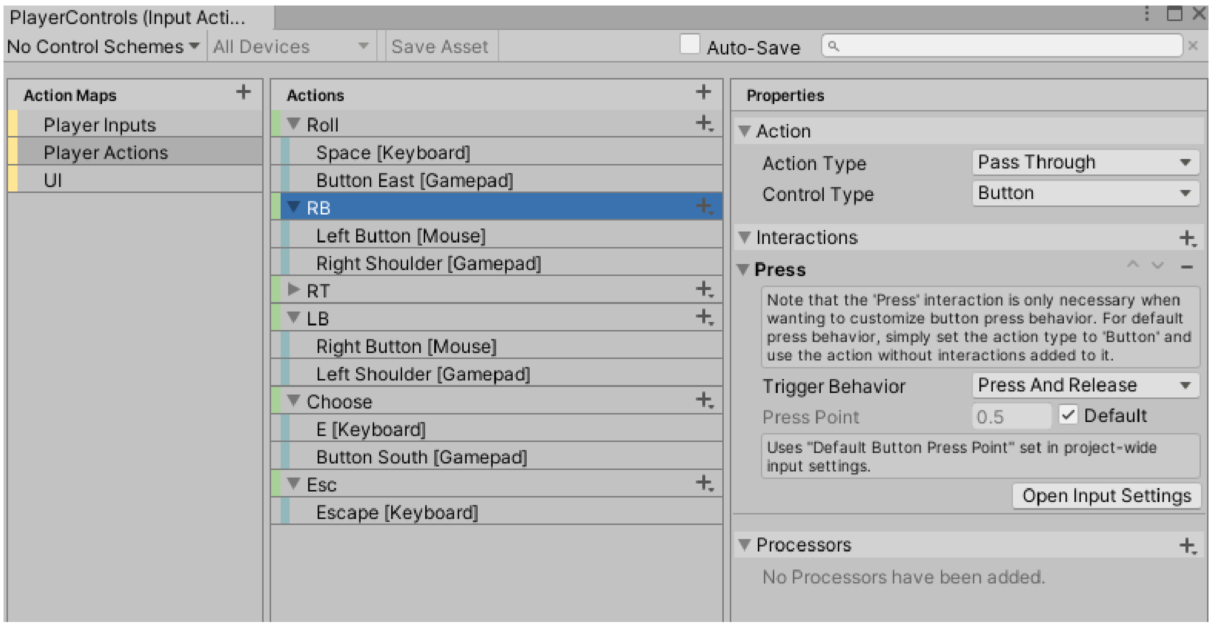
Properties

Action
- Action Type: Pass Throug
- Control Type: Vector 2

Interactions
- No Interactions have been added.

Processors
- No Processors have been added.

However, it seems that there are some bugs in Unity default new Input System 1.0.0, somehow binding of keys combo (e.g. shift + left click), or “Input.GetKeysDown” function in old input function has no equivalent function in 1.0.0 Input System. It claims that those are fixed in 1.0.1 patch during the mid-term of our semester, but we decide not to alter yet for safety. Thus, for now, the game could be played on mouse and keyboard only.



PlayerControls (Input Acti...

No Control Schemes ▾ | All Devices ▾ | Save Asset | ☐ Auto-Save

Action Maps
- Player Inputs
- Player Actions
- UI

Actions
- ▼ Roll
  - Space [Keyboard]
  - Button East [Gamepad]
- ▼ RB
  - Left Button [Mouse]
  - Right Shoulder [Gamepad]
- ▶ RT
- ▼ LB
  - Right Button [Mouse]
  - Left Shoulder [Gamepad]
- ▼ Choose
  - E [Keyboard]
  - Button South [Gamepad]
- ▼ Esc
  - Escape [Keyboard]

Properties

▼ Action

Action Type: Pass Through

Control Type: Button

▼ Interactions

▼ Press

Note that the 'Press' interaction is only necessary when wanting to customize button press behavior. For default press behavior, simply set the action type to 'Button' and use the action without interactions added to it.

Trigger Behavior: Press And Release

Press Point: 0.5 ☑ Default

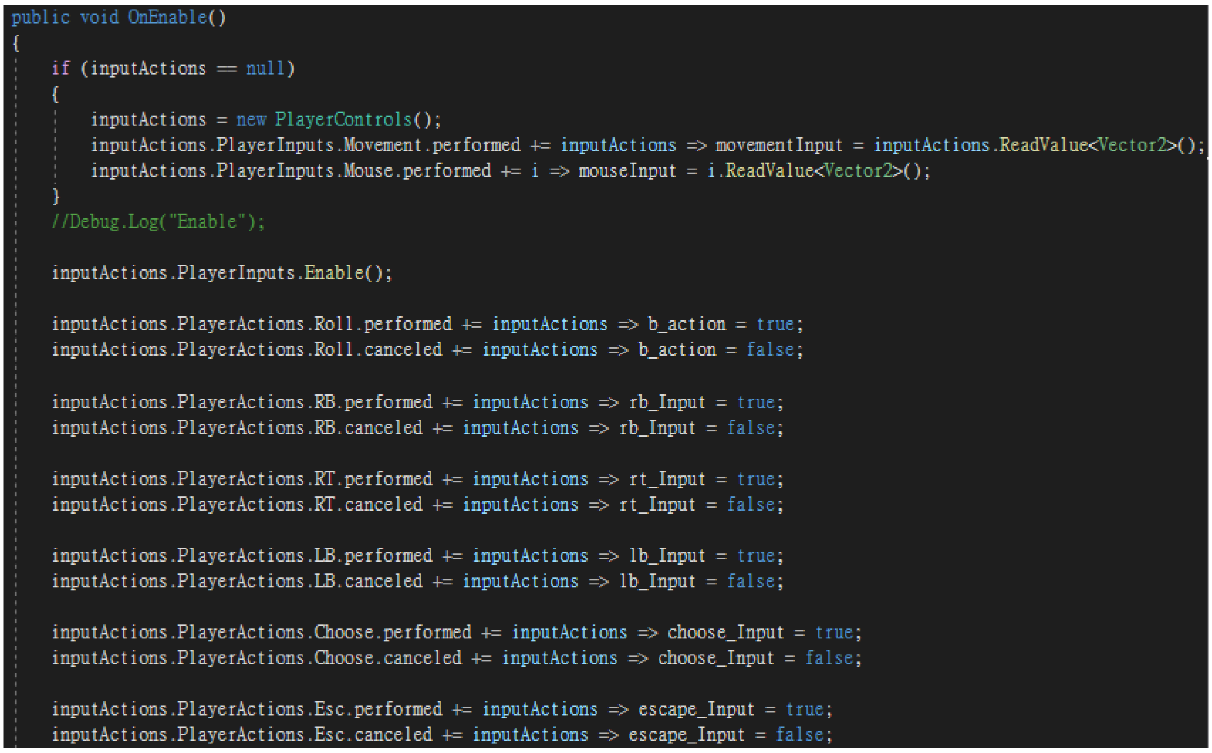
Uses "Default Button Press Point" set in project-wide input settings.

Open Input Settings

▼ Processors

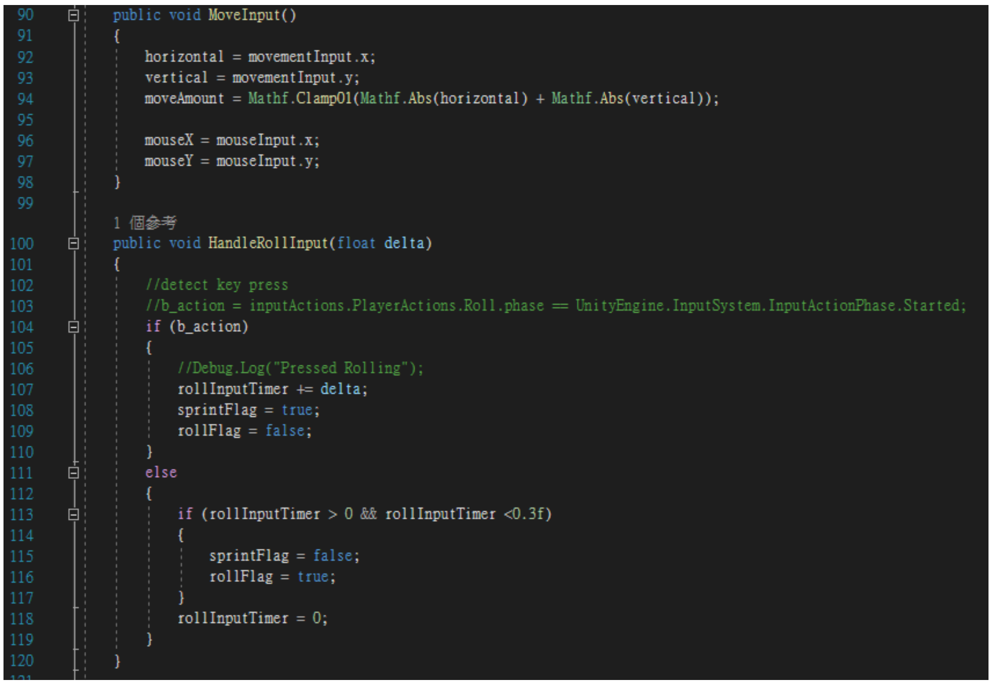
No Processors have been added.

Unity implements this new system by event-driven. All input is delivered as events and we can generate custom input by injecting events. Events are stored in unmanaged memory buffers and not converted to C# heap object. But usually we don’t have to deal with the actual code inside unless we are using custom input devices. We just need to register the input actions, and using a handler for those events, i.e.



```
public void OnEnable()
{
    if (inputActions == null)
    {
        inputActions = new PlayerControls();
        inputActions.PlayerInputs.Movement.performed += inputActions => movementInput = inputActions.ReadValue<Vector2>();
        inputActions.PlayerInputs.Mouse.performed += i => mouseInput = i.ReadValue<Vector2>();
    }
    //Debug.Log("Enable");

    inputActions.PlayerInputs.Enable();

    inputActions.PlayerActions.Roll.performed += inputActions => b_action = true;
    inputActions.PlayerActions.Roll.canceled += inputActions => b_action = false;

    inputActions.PlayerActions.RB.performed += inputActions => rb_Input = true;
    inputActions.PlayerActions.RB.canceled += inputActions => rb_Input = false;

    inputActions.PlayerActions.RT.performed += inputActions => rt_Input = true;
    inputActions.PlayerActions.RT.canceled += inputActions => rt_Input = false;

    inputActions.PlayerActions.LB.performed += inputActions => lb_Input = true;
    inputActions.PlayerActions.LB.canceled += inputActions => lb_Input = false;

    inputActions.PlayerActions.Choose.performed += inputActions => choose_Input = true;
    inputActions.PlayerActions.Choose.canceled += inputActions => choose_Input = false;

    inputActions.PlayerActions.Esc.performed += inputActions => escape_Input = true;
    inputActions.PlayerActions.Esc.canceled += inputActions => escape_Input = false;
```

After registering all events, the script contains different functions for handling these inputs, and output different flags for other scripts usage.



```
public void MoveInput()
{
    horizontal = movementInput.x;
    vertical = movementInput.y;
    moveAmount = Mathf.Clamp01(Mathf.Abs(horizontal) + Mathf.Abs(vertical));

    mouseX = mouseInput.x;
    mouseY = mouseInput.y;
}

1 個參考
public void HandleRollInput(float delta)
{
    //detect key press
    //b_action = inputActions.PlayerActions.Roll.phase == UnityEngine.InputSystem.InputActionPhase.Started;
    if (b_action)
    {
        //Debug.Log("Pressed Rolling");
        rollInputTimer += delta;
        sprintFlag = true;
        rollFlag = false;
    }
    else
    {
        if (rollInputTimer > 0 && rollInputTimer <0.3f)
        {
            sprintFlag = false;
            rollFlag = true;
        }
        rollInputTimer = 0;
    }
}
```

,
i.e. the HandleRollInput will handle if a single button is holding or just a press-and-release, then it could determine it is sprinting or rolling, these flags will be used in player locomotion.

8.2. Player Locomotion

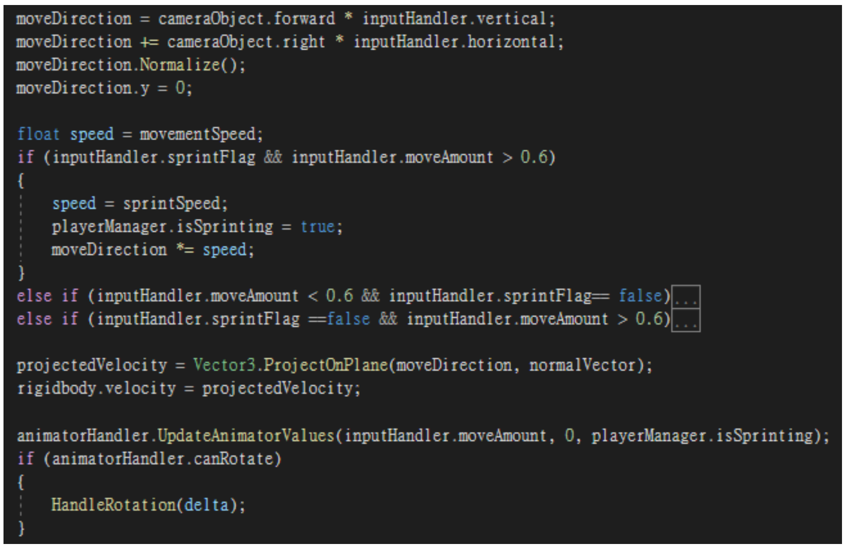
This should contain and handle all rigidbody movements, and rotations, carried by different events, i.e. player inputs for walking, sprinting, rolling. And it stores all velocity or movement speed statistics in this script.

Although it is recommended that we should just use an animated character in Unity assets store, I think one of the targets in our fyp is creating a good fighting experience, which cannot be found in Unity assets store, and should be made manually, no matter for learning or our fyp.

The player locomotion scripts will receive the output from the input-handler.

- **Player rotation** is handled by calculating the direction (product of the camera direction and the user inputs), the results will be assigned to the player's transform rotation.

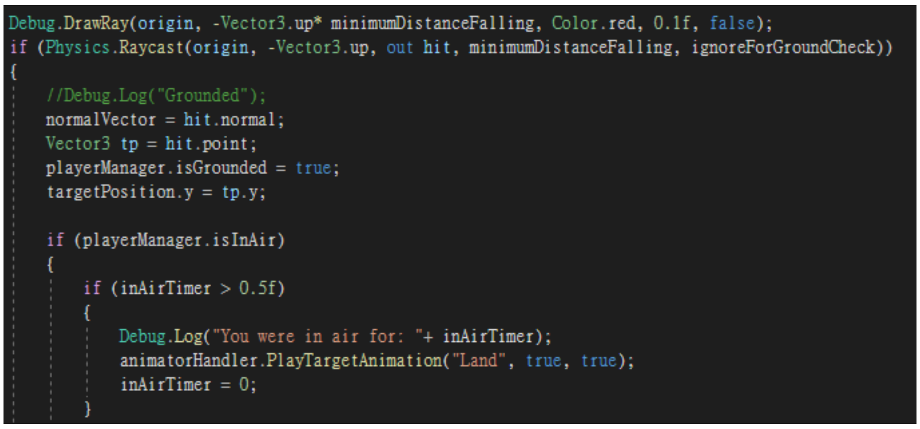
- For **basic movement**, the direction is based on camera direction and user input as well. After the velocity is assigned to our player based on the player's input, it calls the animator handler to play the corresponding animation, which will be explained in 7.3. animator handler. Different player's movement input should have different velocity, e.g. walk, sprint, run.



```
moveDirection = cameraObject.forward * inputHandler.vertical;
moveDirection += cameraObject.right * inputHandler.horizontal;
moveDirection.Normalize();
moveDirection.y = 0;

float speed = movementSpeed;
if (inputHandler.sprintFlag && inputHandler.moveAmount > 0.6)
{
    speed = sprintSpeed;
    playerManager.isSprinting = true;
    moveDirection *= speed;
}
else if (inputHandler.moveAmount < 0.6 && inputHandler.sprintFlag== false)...
else if (inputHandler.sprintFlag ==false && inputHandler.moveAmount > 0.6)...

projectedVelocity = Vector3.ProjectOnPlane(moveDirection, normalVector);
rigidbody.velocity = projectedVelocity;

animatorHandler.UpdateAnimatorValues(inputHandler.moveAmount, 0, playerManager.isSprinting);
if (animatorHandler.canRotate)
{
    HandleRotation(delta);
}
```

- For **rolling**, it is similar to basic movement, but it gives the direction only, no velocity shall be assigned. The rolling will be based on the animation final delta position. But the final position could be handled manually (without delta position) if it has a better user experience and we could change the method in the next semester.

- For **falling,** it uses a ray cast under the player's push collider (which is for physical collision and movements). It detects a certain distance from the player, if it is in-air for some time, it will trigger the animation "falling". If it does not exceed the trigger falling time, it should have nothing happen for animation. But, no matter which types of falling, it must have gravity for falling if it is in-air.



```
Debug.DrawRay(origin, -Vector3.up* minimumDistanceFalling, Color.red, 0.1f, false);
if (Physics.Raycast(origin, -Vector3.up, out hit, minimumDistanceFalling, ignoreForGroundCheck))
{
    //Debug.Log("Grounded");
    normalVector = hit.normal;
    Vector3 tp = hit.point;
    playerManager.isGrounded = true;
    targetPosition.y = tp.y;

    if (playerManager.isInAir)
    {
        if (inAirTimer > 0.5f)
        {
            Debug.Log("You were in air for: "+ inAirTimer);
            animatorHandler.PlayTargetAnimation("Land", true, true);
            inAirTimer = 0;
        }
```

If it has been in-air, and suddenly on ground, it will trigger "landing" animation.

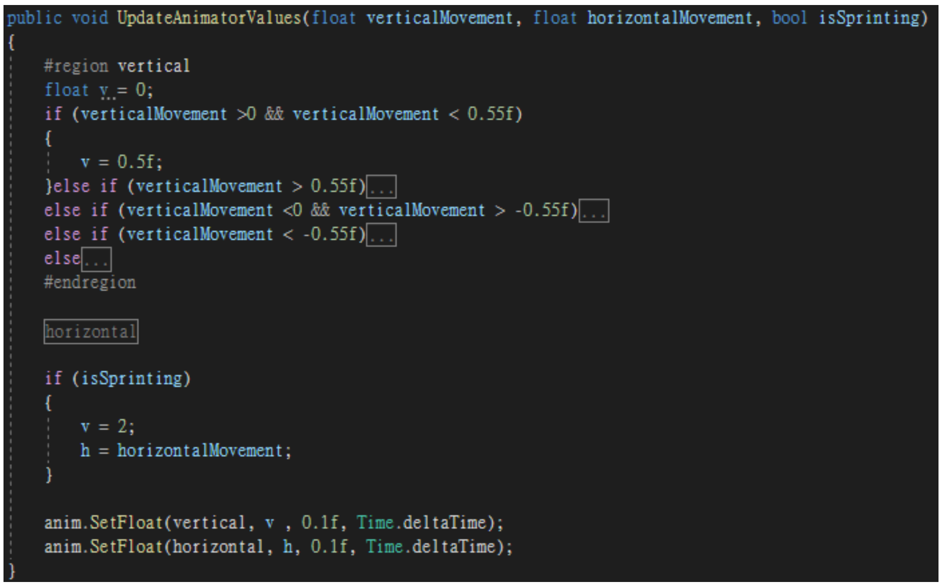
- For attacks, to make a good fighting experience, it should not have standing attacks only. Thus, the character moves when attacking.

This uses a player manager to deal with the boolean in the animator, the isAttacking boolean will be inserted into the animation itself, true if attacking. The attackRotate is in this player locomotion script, and prevents it from receiving input and changing the direction. Thus, the <u>attack</u> should have <u>one-direction movement</u>.

8.3. Animator Handler

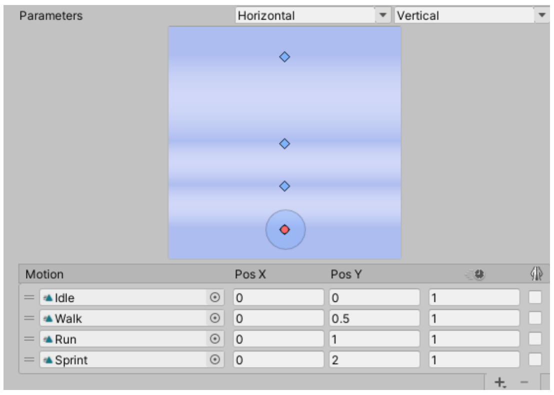
This script will be called when it needs animator and associated animations, it contains different functions which could be inserted inside some frame of animation, i.e. set bool for animator.

1. Update locomotion value for blend tree



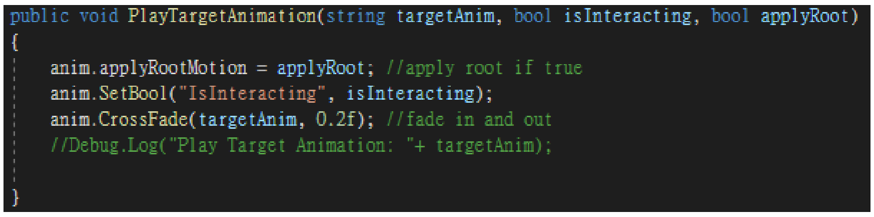
```
public void UpdateAnimatorValues(float verticalMovement, float horizontalMovement, bool isSprinting)
{
    #region vertical
    float v = 0;
    if (verticalMovement >0 && verticalMovement < 0.55f)
    {
        v = 0.5f;
    }else if (verticalMovement > 0.55f)...
    else if (verticalMovement <0 && verticalMovement > -0.55f)...
    else if (verticalMovement < -0.55f)...
    else...
    #endregion

    horizontal

    if (isSprinting)
    {
        v = 2;
        h = horizontalMovement;
    }

    anim.SetFloat(vertical, v , 0.1f, Time.deltaTime);
    anim.SetFloat(horizontal, h, 0.1f, Time.deltaTime);
}
```

It is called after handling velocity in player locomotion, receiving inputs and flags from the input handler, and output related values for the blend tree.



Motion	Pos X	Pos Y	
Idle	0	0	1
Walk	0	0.5	1
Run	0	1	1
Sprint	0	2	1

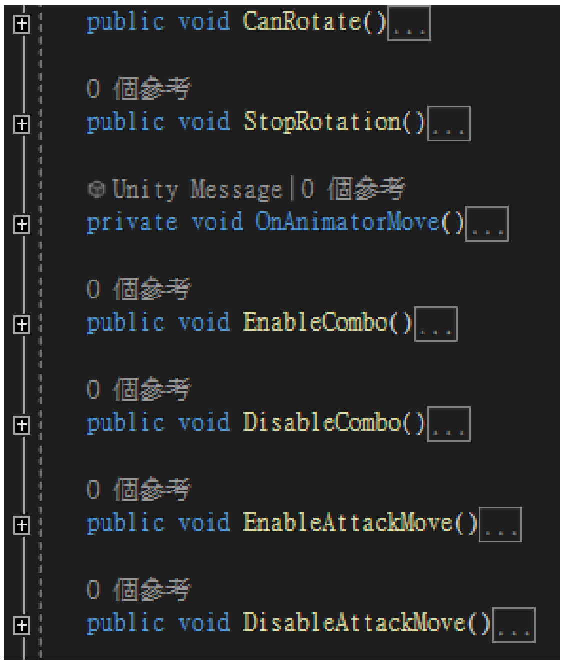
2. Target animation handling



```
public void PlayTargetAnimation(string targetAnim, bool isInteracting, bool applyRoot)
{
    anim.applyRootMotion = applyRoot; //apply root if true
    anim.SetBool("IsInteracting", isInteracting);
    anim.CrossFade(targetAnim, 0.2f); //fade in and out
    //Debug.Log("Play Target Animation: "+ targetAnim);

}
```

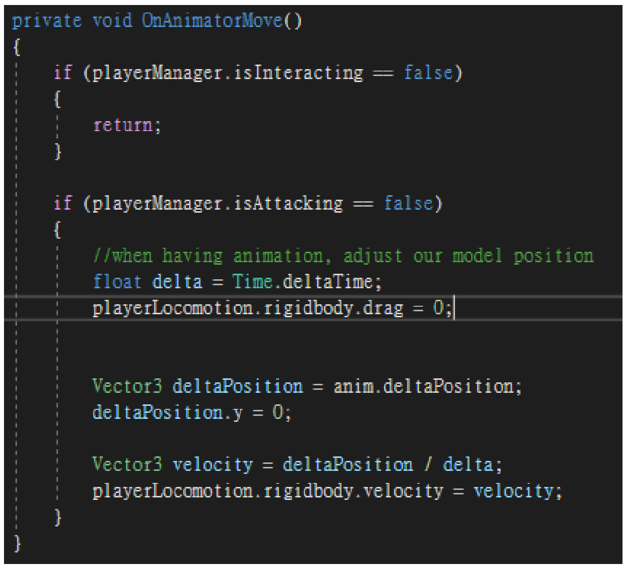
It handles all animation clips, sets them into root motion by scripts, each animation could set one more boolean: **isInteracting**, such that it could prevent interfering to other animation. CrossFade helps fade in and out by interpolate and results in smoother animation transition.

3. Other functions for animation in frames



```
public void CanRotate()...

0 個參考
public void StopRotation()...

Unity Message | 0 個參考
private void OnAnimatorMove()...

0 個參考
public void EnableCombo()...

0 個參考
public void DisableCombo()...

0 個參考
public void EnableAttackMove()...

0 個參考
public void DisableAttackMove()...
```

All of these mostly are for changing the boolean of animator during animation frames. OnAnimatorMove() is used with **isInteracting** for preventing interference.

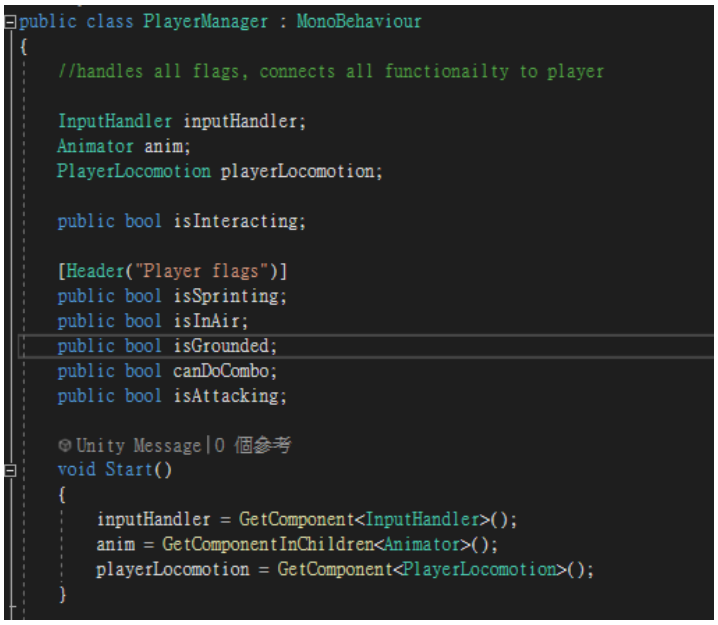


```
private void OnAnimatorMove()
{
    if (playerManager.isInteracting == false)
    {
        return;
    }

    if (playerManager.isAttacking == false)
    {
        //when having animation, adjust our model position
        float delta = Time.deltaTime;
        playerLocomotion.rigidbody.drag = 0;


        Vector3 deltaPosition = anim.deltaPosition;
        deltaPosition.y = 0;

        Vector3 velocity = deltaPosition / delta;
        playerLocomotion.rigidbody.velocity = velocity;
    }
}
```

This script also forces all character positions to stick with the animation (delta position), except attacking.

8.4. Player Manager

All the above scripts, their functions will be run in player manager. The function will be run in FixedUpdates, as all rigidbody should be in fixed update as recommended by Unity for not causing jittering.



```
public class PlayerManager : MonoBehaviour
{
    //handles all flags, connects all functionailty to player

    InputHandler inputHandler;
    Animator anim;
    PlayerLocomotion playerLocomotion;

    public bool isInteracting;

    [Header("Player flags")]
    public bool isSprinting;
    public bool isInAir;
    public bool isGrounded;
    public bool canDoCombo;
    public bool isAttacking;

    Unity Message|0 個參考
    void Start()
    {
        inputHandler = GetComponent<InputHandler>();
        anim = GetComponentInChildren<Animator>();
        playerLocomotion = GetComponent<PlayerLocomotion>();
    }
```

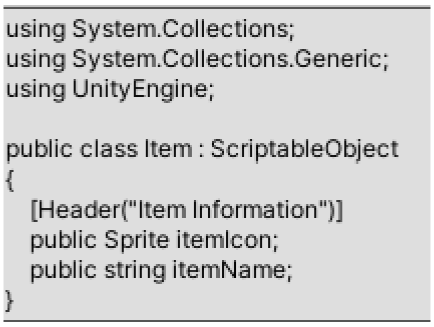
The player manager contains all necessary flags monitoring the player’ state for easier bug check. And it has a LateUpdate for resetting some flags by time.

9. Melee system

9.1. Weapon

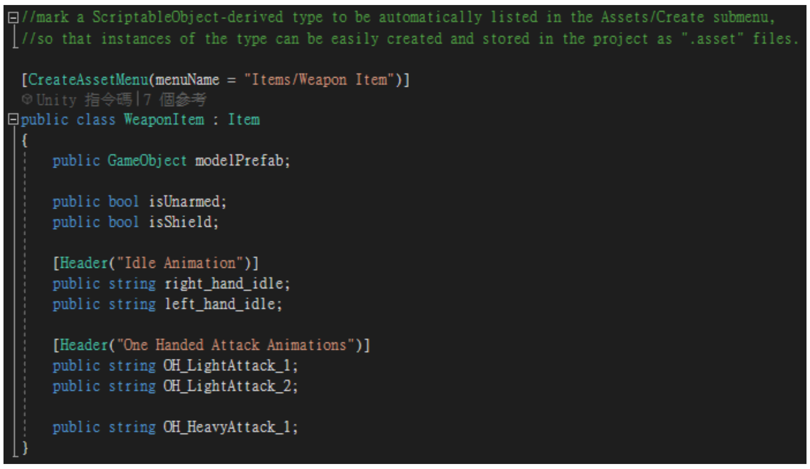
Using scriptable objects to store weapons.

Item:



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

public class Item : ScriptableObject
{
    [Header("Item Information")]
    public Sprite itemIcon;
    public string itemName;
}
```

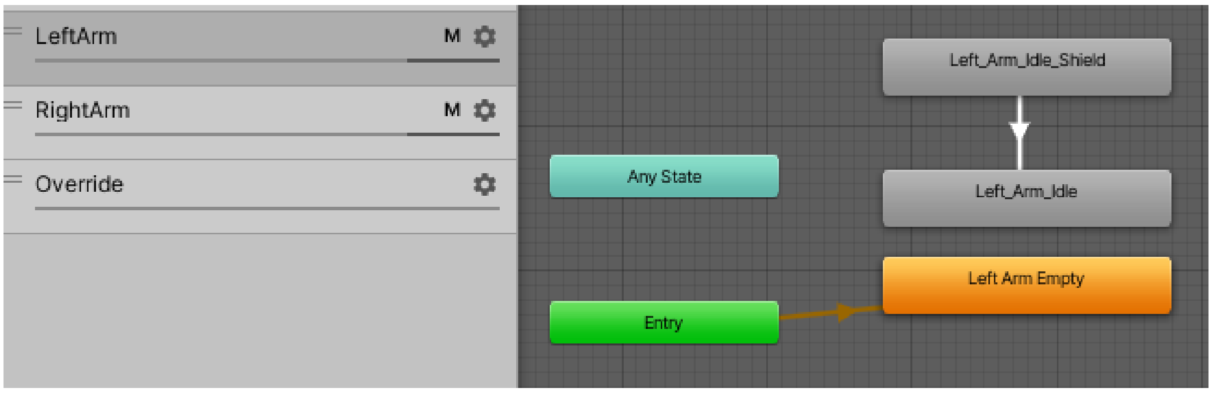
Weapon Item: (inherited from Item)



```
//mark a ScriptableObject-derived type to be automatically listed in the Assets/Create submenu,
//so that instances of the type can be easily created and stored in the project as ".asset" files.

[CreateAssetMenu(menuName = "Items/Weapon Item")]
Unity 指令碼|7 個參考
public class WeaponItem : Item
{
    public GameObject modelPrefab;

    public bool isUnarmed;
    public bool isShield;

    [Header("Idle Animation")]
    public string right_hand_idle;
    public string left_hand_idle;

    [Header("One Handed Attack Animations")]
    public string OH_LightAttack_1;
    public string OH_LightAttack_2;

    public string OH_HeavyAttack_1;
}
```

So each item could contain a name and an icon. Each weapon could have their specified attack or idle animations.

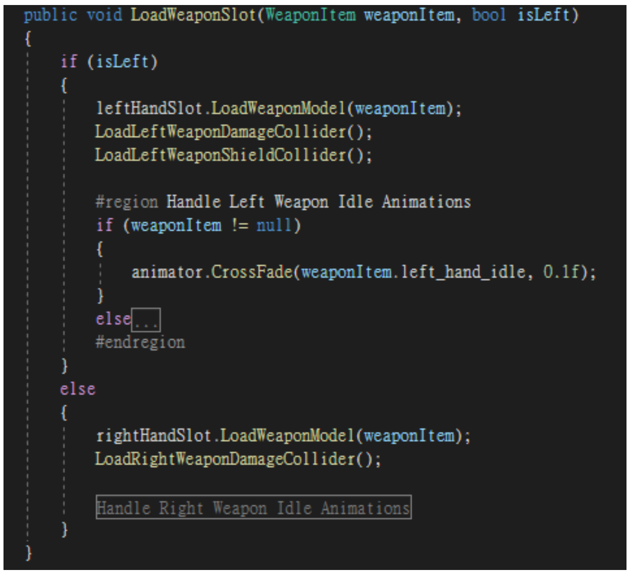
Idle animation is made by avatar mask, that means I divide the idle animation into separate hands.



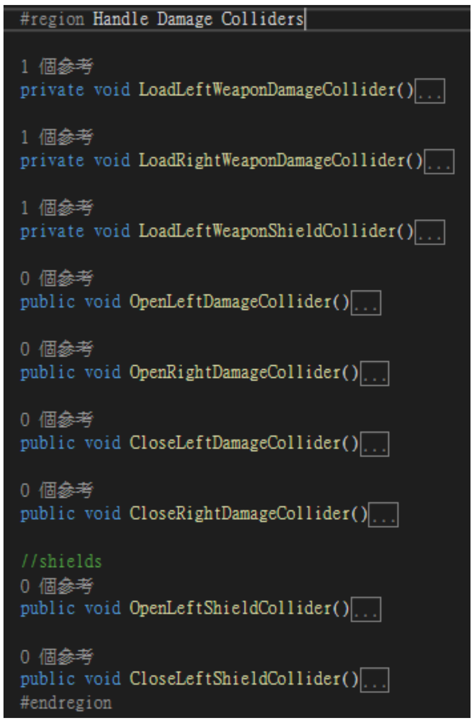
This could allow some animation for a particular arm while moving, i.e. shielding and move at the same time. And it could allow ranged attack with moving such as arrows shooting, which shall be done in the next semester.

9.2. Weapon Slot Manager

This manages loading of weapons and provides function for enabling or disabling different kinds of colliders in animation frames, i.e. hitbox and hurtbox.
The position of slots have been specified into the rig of character, as a type of weaponSlot, i.e. leftHandSlot.



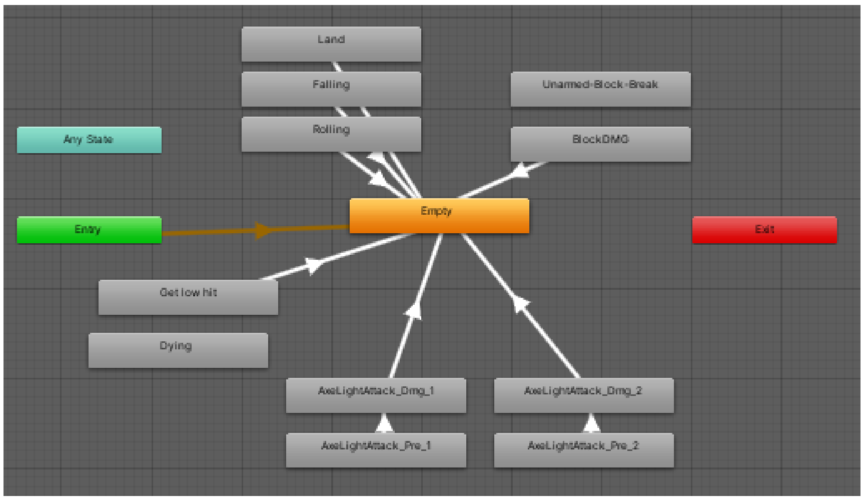
```
public void LoadWeaponSlot(WeaponItem weaponItem, bool isLeft)
{
    if (isLeft)
    {
        leftHandSlot.LoadWeaponModel(weaponItem);
        LoadLeftWeaponDamageCollider();
        LoadLeftWeaponShieldCollider();

        #region Handle Left Weapon Idle Animations
        if (weaponItem != null)
        {
            animator.CrossFade(weaponItem.left_hand_idle, 0.1f);
        }
        else...
        #endregion
    }
    else
    {
        rightHandSlot.LoadWeaponModel(weaponItem);
        LoadRightWeaponDamageCollider();

        Handle Right Weapon Idle Animations
    }
}
```



```
#region Handle Damage Colliders

1 個參考
private void LoadLeftWeaponDamageCollider()...

1 個參考
private void LoadRightWeaponDamageCollider()...

1 個參考
private void LoadLeftWeaponShieldCollider()...

0 個參考
public void OpenLeftDamageCollider()...

0 個參考
public void OpenRightDamageCollider()...

0 個參考
public void CloseLeftDamageCollider()...

0 個參考
public void CloseRightDamageCollider()...

//shields
0 個參考
public void OpenLeftShieldCollider()...

0 個參考
public void CloseLeftShieldCollider()...
#endregion
```

9.2. Animator

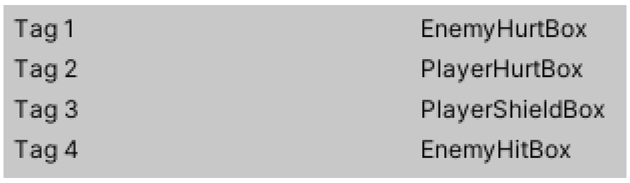
As mentioned before, it uses avatar masks and different layers,
the attack/ hurt/ dying animation will be in this overwrite layer.



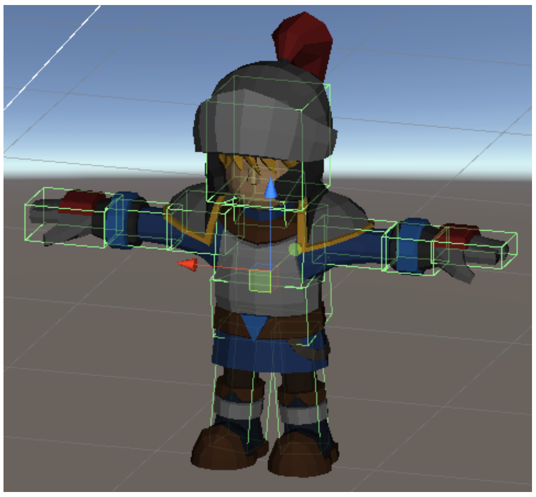
Every animation ends will return to empty, and empty state will reset all boolean flags specified, i.e. IsInteracting.

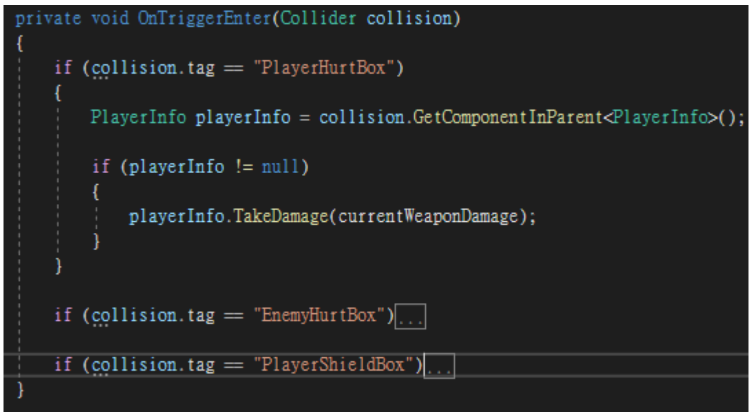
9.3. Hitbox and Hurtbox

Hitbox and hurtbox is inserted manually by collider and corresponding tags into the rig or weapon prefabs.

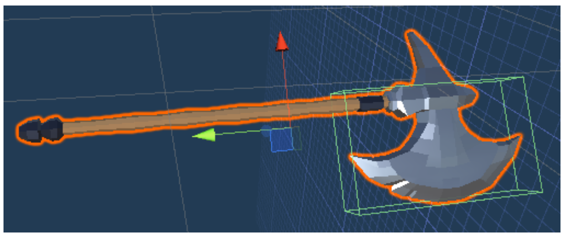


Tag 1	EnemyHurtBox
Tag 2	PlayerHurtBox
Tag 3	PlayerShieldBox
Tag 4	EnemyHitBox





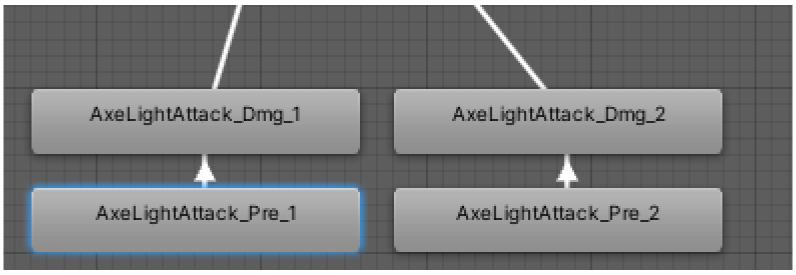
```
private void OnTriggerEnter(Collider collision)
{
    if (collision.tag == "PlayerHurtBox")
    {
        PlayerInfo playerInfo = collision.GetComponentInParent<PlayerInfo>();

        if (playerInfo != null)
        {
            playerInfo.TakeDamage(currentWeaponDamage);
        }
    }

    if (collision.tag == "EnemyHurtBox")...

    if (collision.tag == "PlayerShieldBox")...
}
```

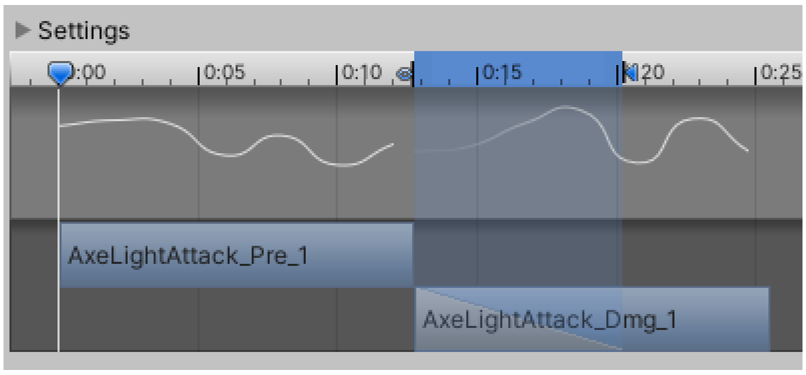


The above script is for a damage collider, when attacking, the collider will be enabled, and following the animated rig, if it detects a specific hurt box or shield, it plays their corresponding animation.

Shield box is implemented as the same method as damage collider.

9.4. Combo Attack

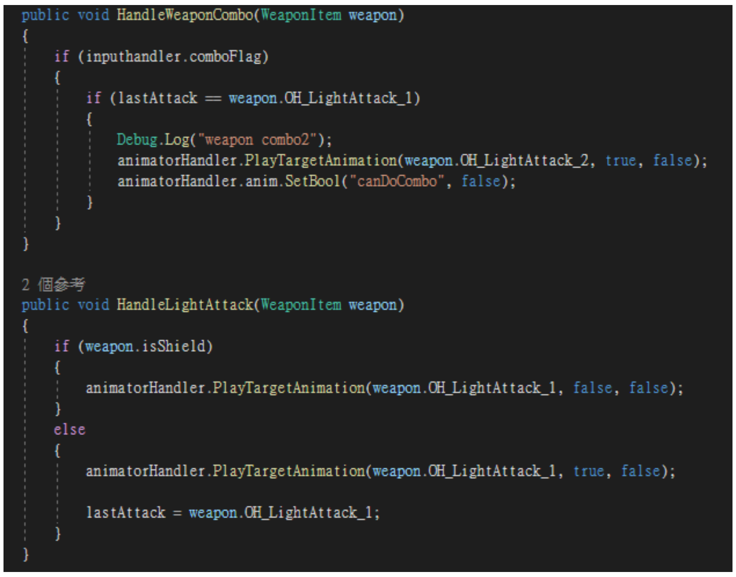




As some attack locomotion and some collision functions are called in animation frames, each transition will have 0 transition offsets.

An attack is divided into preparation, and actual attack, so that players could change direction just after preparing, the attack locomotion movement will take place in actual attack only.

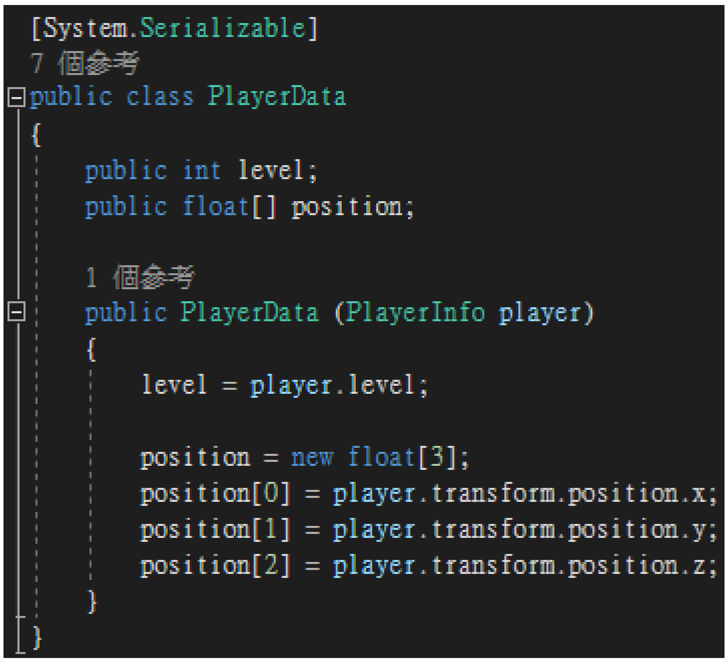
The attack input and flags are handled by the input handler, the combo time is defined in first attack animation frames. After that, if there is attack input, calling the function in “playerAttacker.cs”. i.e.



```
public void HandleWeaponCombo(WeaponItem weapon)
{
    if (inputhandler.comboFlag)
    {
        if (lastAttack == weapon.OH_LightAttack_1)
        {
            Debug.Log("weapon combo2");
            animatorHandler.PlayTargetAnimation(weapon.OH_LightAttack_2, true, false);
            animatorHandler.anim.SetBool("canDoCombo", false);
        }
    }
}

2 個參考
public void HandleLightAttack(WeaponItem weapon)
{
    if (weapon.isShield)
    {
        animatorHandler.PlayTargetAnimation(weapon.OH_LightAttack_1, false, false);
    }
    else
    {
        animatorHandler.PlayTargetAnimation(weapon.OH_LightAttack_1, true, false);

        lastAttack = weapon.OH_LightAttack_1;
    }
}
```

10. Save & Load system

10.1 Saving Player's related stuff



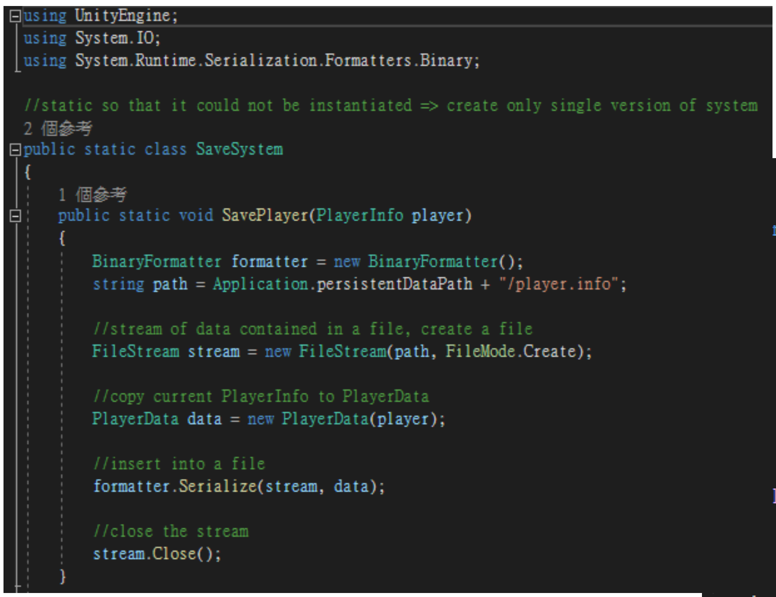
```
[System.Serializable]
7 個參考
public class PlayerData
{
    public int level;
    public float[] position;

    1 個參考
    public PlayerData (PlayerInfo player)
    {
        level = player.level;

        position = new float[3];
        position[0] = player.transform.position.x;
        position[1] = player.transform.position.y;
        position[2] = player.transform.position.z;
    }
}
```

The save & load system is implemented by ourselves for more freedom and customizable.

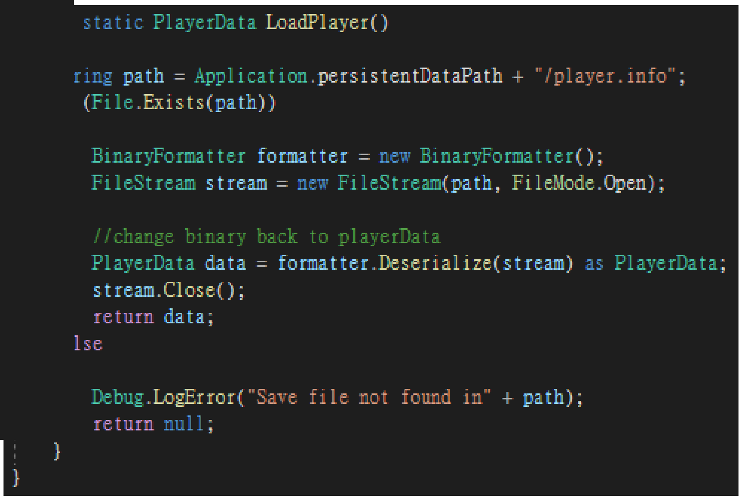
Currently, we only save and load maximum health, player levels, positions, and inventory (weapons).

Saving level progress and positions are implemented by creating a serializable abstract data type, and creating a static class handling, creating, saving or loading this data type, which we called a SaveSystem class.

All player data will be stored in a filename that we could name it by ourselves, we called it .info. This is a binary file, serialized by System.Runtime.Serialization.Formatters.Binary. After that, we could save or load player data when we call the functions in player info class.

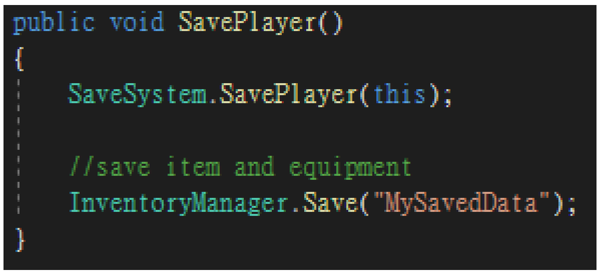


```
using UnityEngine;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

//static so that it could not be instantiated => create only single version of system
2 個參考
public static class SaveSystem
{
    1 個參考
    public static void SavePlayer(PlayerInfo player)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        string path = Application.persistentDataPath + "/player.info";

        //stream of data contained in a file, create a file
        FileStream stream = new FileStream(path, FileMode.Create);

        //copy current PlayerInfo to PlayerData
        PlayerData data = new PlayerData(player);

        //insert into a file
        formatter.Serialize(stream, data);

        //close the stream
        stream.Close();
    }
```

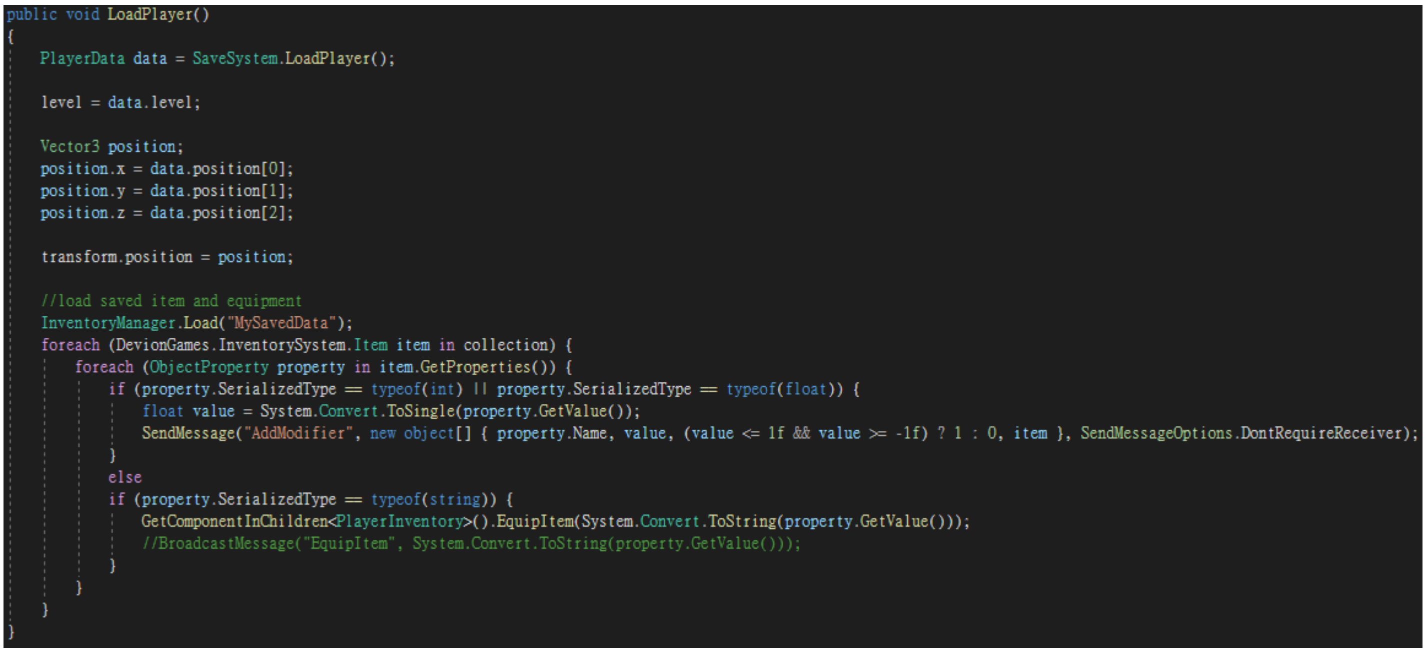


```
 static PlayerData LoadPlayer()

ring path = Application.persistentDataPath + "/player.info";
 (File.Exists(path))

  BinaryFormatter formatter = new BinaryFormatter();
  FileStream stream = new FileStream(path, FileMode.Open);

  //change binary back to playerData
  PlayerData data = formatter.Deserialize(stream) as PlayerData;
  stream.Close();
  return data;
lse

  Debug.LogError("Save file not found in" + path);
  return null;
    }
}
```

Then, the above self-implemented save & load system will be used with the imported Devion Games' save & load system, which they are saving through player preference (PlayerPref), saving their statistic and inventory system.



```
public void SavePlayer()
{
    SaveSystem.SavePlayer(this);

    //save item and equipment
    InventoryManager.Save("MySavedData");
}
```



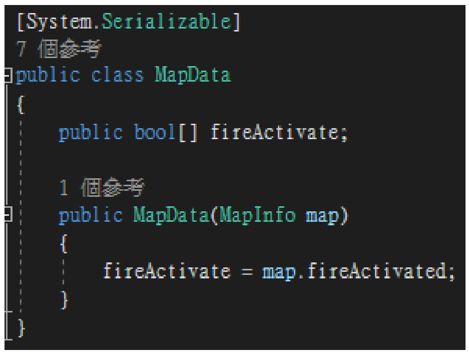
```
public void LoadPlayer()
{
    PlayerData data = SaveSystem.LoadPlayer();

    level = data.level;

    Vector3 position;
    position.x = data.position[0];
    position.y = data.position[1];
    position.z = data.position[2];

    transform.position = position;

    //load saved item and equipment
    InventoryManager.Load("MySavedData");
    foreach (DevionGames.InventorySystem.Item item in collection) {
        foreach (ObjectProperty property in item.GetProperties()) {
            if (property.SerializedType == typeof(int) || property.SerializedType == typeof(float)) {
                float value = System.Convert.ToSingle(property.GetValue());
                SendMessage("AddModifier", new object[] { property.Name, value, (value <= 1f && value >= -1f) ? 1 : 0, item }, SendMessageOptions.DontRequireReceiver);
            }
            else
            if (property.SerializedType == typeof(string)) {
                GetComponentInChildren<PlayerInventory>().EquipItem(System.Convert.ToString(property.GetValue()));
                //BroadcastMessage("EquipItem", System.Convert.ToString(property.GetValue()));
            }
        }
    }
}
```

The loaded equipment item will be used to calculate and update the character stat and also change the motion of the character to be the equipped mode.

10.2 Saving Map's data

We use the same way to create map's data, like memorizing whether the player has unlocked the door, or they have fired the Bonfire, or they have opened their treasures yey.

e.g.



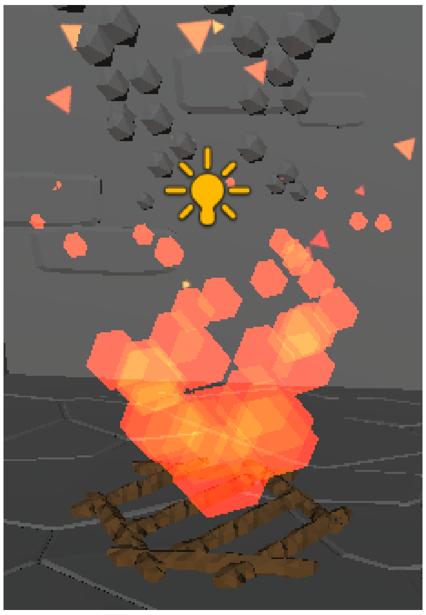
```
[System.Serializable]
7 個參考
public class MapData
{
    public bool[] fireActivate;

    1 個參考
    public MapData(MapInfo map)
    {
        fireActivate = map.fireActivated;
    }
}
```

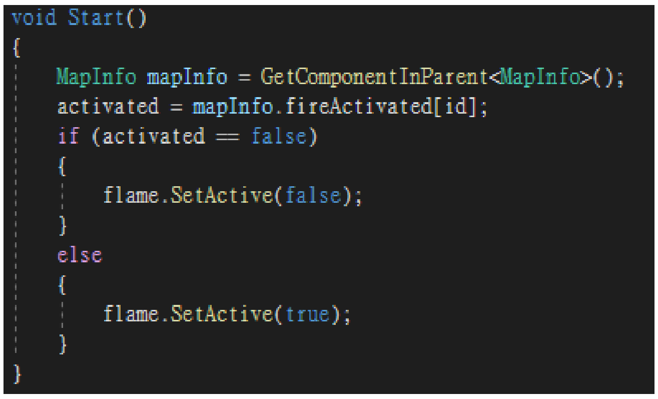
11. Bonfire

As mentioned in 1.4. Classification, Bonfire is a checkpoint that should be found and activated by the player. After being activated (fired), the player can use it to save their progress.

If the player is dead, after showing up the related UI, he can be reborn in the last visited Bonfire. Meanwhile, the Bonfire can be used to reset the whole level (monster is restored to default state and statistics), and transit between Bonfire.

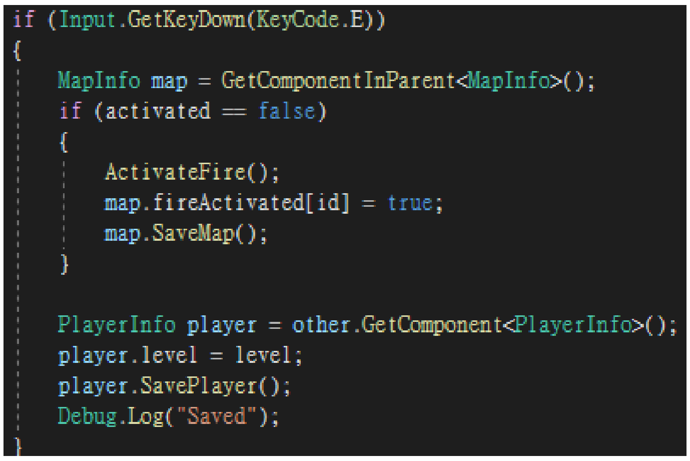
(before activated)


(after activated)



```
void Start()
{
    MapInfo mapInfo = GetComponentInParent<MapInfo>();
    activated = mapInfo.fireActivated[id];
    if (activated == false)
    {
        flame.SetActive(false);
    }
    else
    {
        flame.SetActive(true);
    }
}
```

This ensures if the player has activated it or not.



```
if (Input.GetKeyDown(KeyCode.E))
{
    MapInfo map = GetComponentInParent<MapInfo>();
    if (activated == false)
    {
        ActivateFire();
        map.fireActivated[id] = true;
        map.SaveMap();
    }

    PlayerInfo player = other.GetComponent<PlayerInfo>();
    player.level = level;
    player.SavePlayer();
    Debug.Log("Saved");
}
```

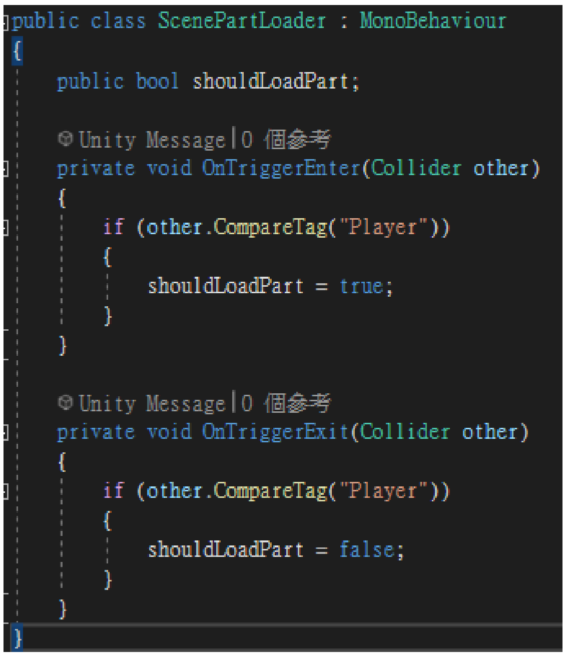
This script is for saving the map data and the player's information (current progress, position, etc.)

12. Level Editing:

12.1 additive scene loading

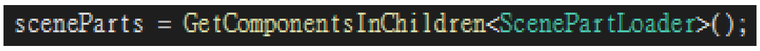
To simulate an open-world game like dark souls, it should not have a scene-loading screen when you are crossing maps in sequence. As there are no free assets for this function, therefore, we implement the additive scene loading by ourselves. Meanwhile, this method allows our team to develop separate levels independently, saving time.

Generally, there are two methods implementing additive scene loading, by distance or by trigger. In this fyp, we decide to use **by trigger**. We set **triggers for each level** or areas in the game detecting where the player is.

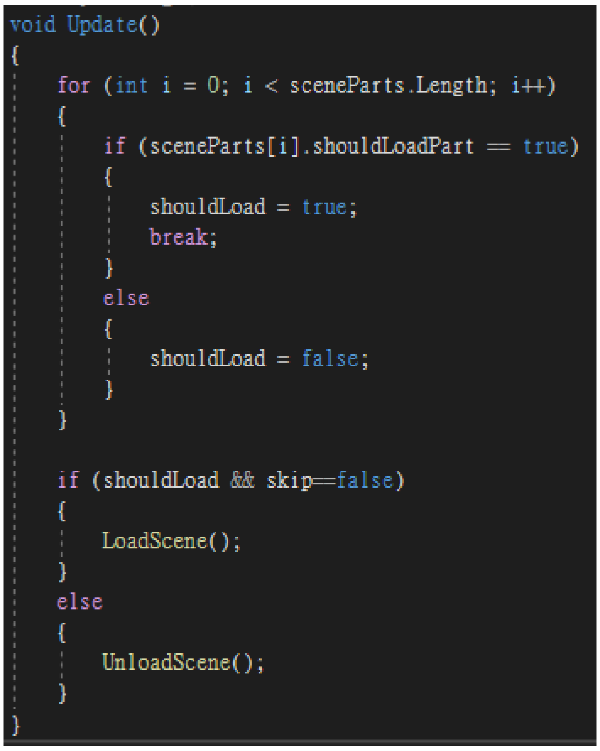


```
public class ScenePartLoader : MonoBehaviour
{
    public bool shouldLoadPart;

    ⚙Unity Message|0 個參考
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            shouldLoadPart = true;
        }
    }

    ⚙Unity Message|0 個參考
    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            shouldLoadPart = false;
        }
    }
}
```

If the player enters one of the triggers, it starts additive loading.

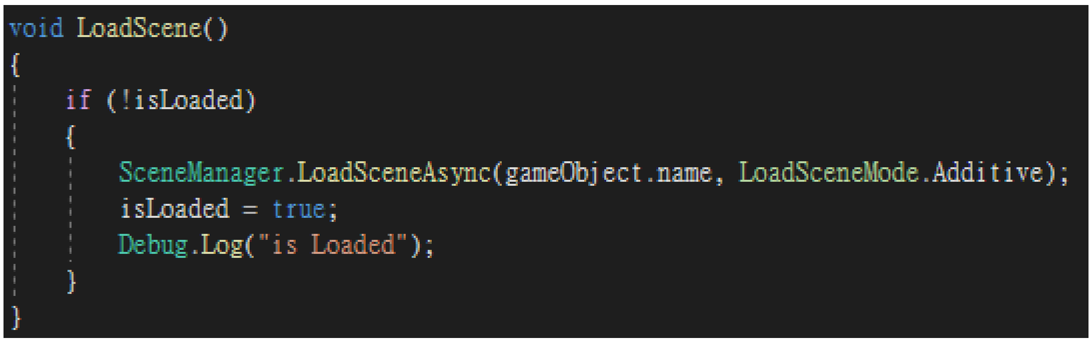


```
sceneParts = GetComponentsInChildren<ScenePartLoader>();
```

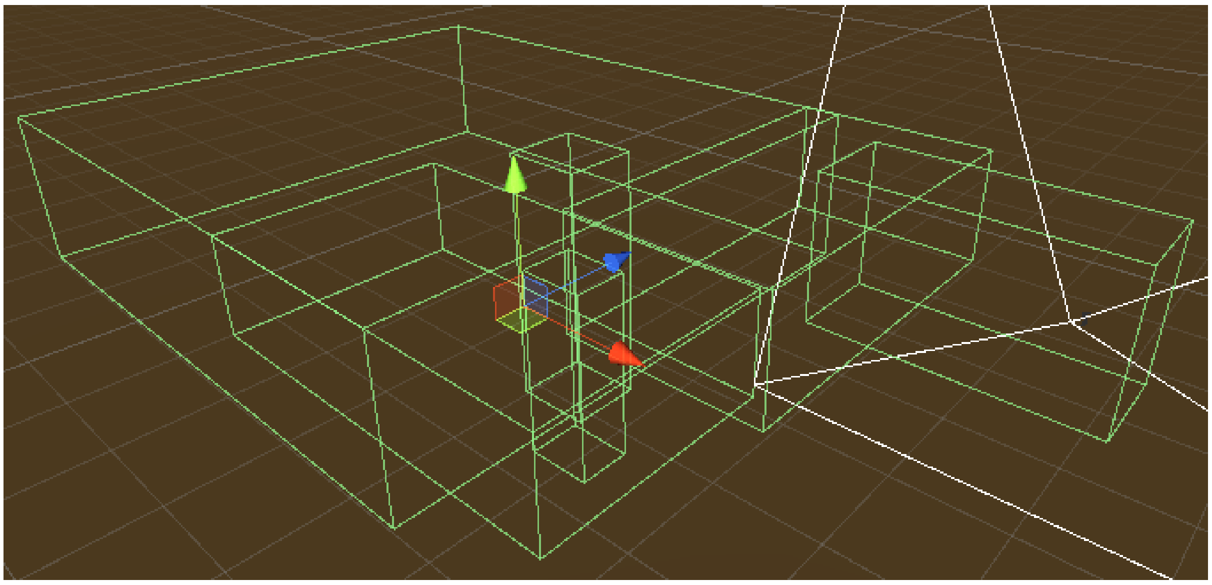


```
void Update()
{
    for (int i = 0; i < sceneParts.Length; i++)
    {
        if (sceneParts[i].shouldLoadPart == true)
        {
            shouldLoad = true;
            break;
        }
        else
        {
            shouldLoad = false;
        }
    }

    if (shouldLoad && skip==false)
    {
        LoadScene();
    }
    else
    {
        UnloadScene();
    }
}
```

Using the trigger method will be better as we can handle manually, so that has a better optimization, as my distance will still create unused loaded scenes.



```
void LoadScene()
{
    if (!isLoaded)
    {
        SceneManager.LoadSceneAsync(gameObject.name, LoadSceneMode.Additive);
        isLoaded = true;
        Debug.Log("is Loaded");
    }
}
```

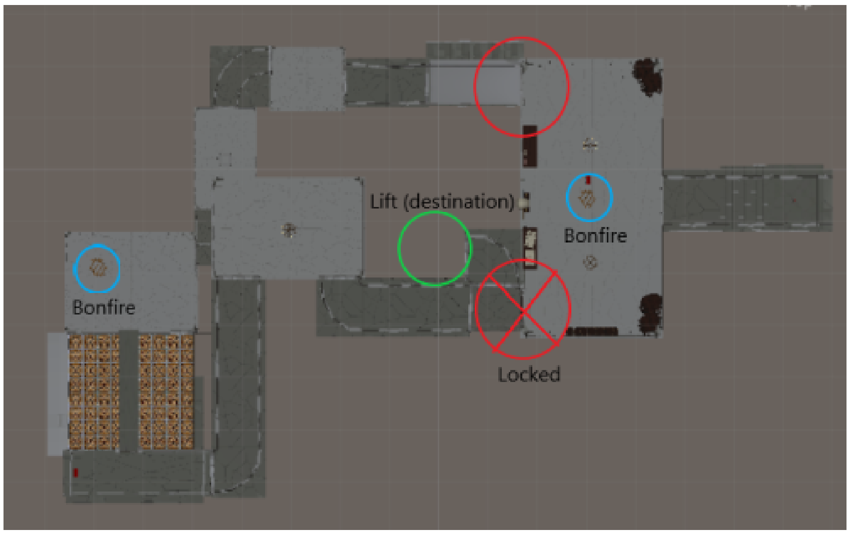


Hence, after we designed and made a scene, we just need to add triggers and related managers with the corresponding scene name, additive loading shall handle it.

12.2 Level Design

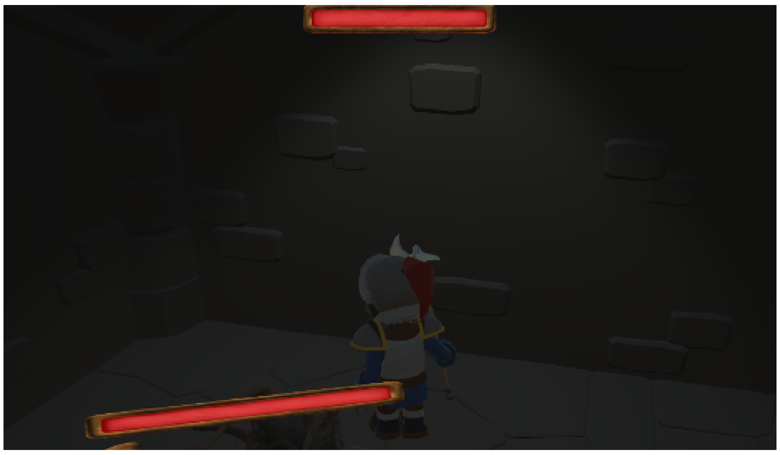
The first level is designed for the demonstration of our fyp, hence, the size should be suitable enough for presenting the features, without huge repetition or boring in running levels.

Thus, our level in this semester would be a simple lock-puzzle, that means the “two-path” of the level is actually one only, and leading the player to unlock and go with our game flow.



After running through the whole level one, the player could unlock the door, which provides the short-cut to the lift (the boss room) from the nearest, and the 1st Bonfire (save-point).

Throughout the map, there are some traps and delightful monster patrol points prepare for you to suffer. To enhance this effect, we use fog for our level and use grey/black as our main theme. To compensate, it will have vision around the player, helping the player to discover.



13. Sound

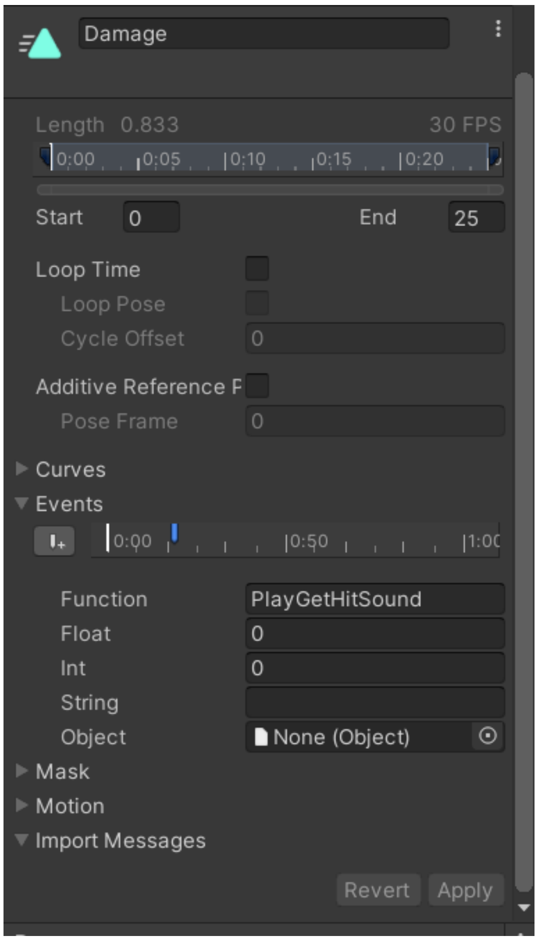
There are mainly three kinds of sound in the sound, one is the monster sound, one is the footstep sound for the player and the last one is the background music. Let me introduce them one by one in the following:

One Common script for both monster and human to play the sound:

```
public AudioClip attackSound;
public AudioClip getHitSound;
public AudioClip die;
public AudioClip footstep;

private AudioSource audioSource;

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

public void PlayAttackSound()...
public void PlayGetHitSound()...
public void PlayDieSound()...
public void PlayFootstepSound()...
```

Sound Effect Player (script name) defines the attack sound, get hit sound, die sound or footstep sound of the object if it needs the sound. Other scripts can play the sound once by running the method below.

<u>Monster Sound</u>:



Damage

Length 0.833 30 FPS

Start 0 End 25

Loop Time
Loop Pose
Cycle Offset 0
Additive Reference F
Pose Frame 0

Curves
Events

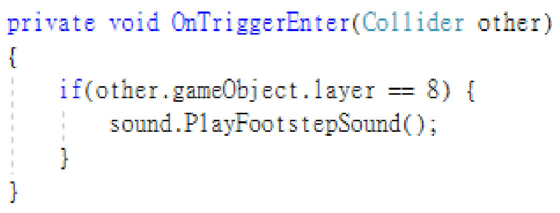
Function PlayGetHitSound
Float 0
Int 0
String
Object None (Object)

Mask
Motion
Import Messages

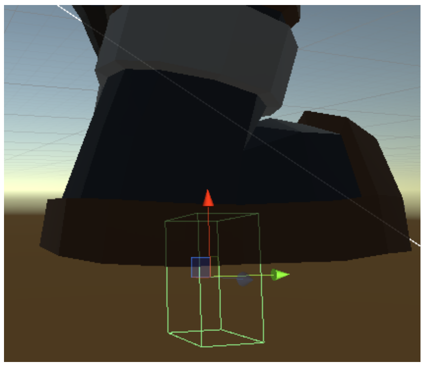
It is the animation of the monster when it gets damaged. it will use the PlayGetHitSound method in the Sound Effect Player to play the get hit sound for the monster.

Other sounds like the attack sound and die sound are also using the same method to play the sound.

<u>Footstep Sound for player</u>:
Create a Trigger Box under the foot and use it to detect whether the player is walking on the ground or not, and



```
private void OnTriggerEnter(Collider other)
{
    if(other.gameObject.layer == 8) {
        sound.PlayFootstepSound();
    }
}
```



<u>Background Music</u>:
Used the bgm of the sakura of rice and ruin soundtrack that I buy from steam. The bgm will keep looping. It will change when the player enters the boss room and starts the battle.

4. Reference

- Item & Inventory System
  https://assetstore.unity.com/packages/tools/gui/item-inventory-system-45568

- Free Ui Pack
  https://assetstore.unity.com/packages/2d/gui/icons/free-ui-pack-170878
  Used to create UI

- RPG inventory icons
  https://assetstore.unity.com/packages/2d/gui/icons/rpg-inventory-icons-56687
  Used for items

- Boris Monster
  https://assetstore.unity.com/packages/3d/characters/creatures/boris-monster-179425

- Chibi Mummy
  https://assetstore.unity.com/packages/3d/characters/chibi-mummy-60462

- Fantasy Monster - Skeleton
  https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy-monster-skeleton-35635

- Four Evil Dragons Pack HP
  https://assetstore.unity.com/packages/3d/characters/creatures/four-evil-dragons-pack-hp-79398

- Fire & Spell Effects
  https://assetstore.unity.com/packages/vfx/particles/fire-explosions/fire-spell-effects-36825
  Used for boss attack

- Potions, Coin And Box of Pandora Pack
  https://assetstore.unity.com/packages/3d/props/potions-coin-and-box-of-pandora-pack-71778

- Free pack of medieval weapons
  https://assetstore.unity.com/packages/3d/props/weapons/free-pack-of-medieval-weapons-136607
  used for item

- Low poly fantasy warrior:

use axe only
https://assetstore.unity.com/?q=asoliddev&orderBy=1

- Kevin Iglesias - Melee Warriors FREE Pack:
  use shield animation only
  https://assetstore.unity.com/packages/3d/animations/melee-warriors-free-pack-165785

- ExplosiveLLC: RPG Character Mecanim Animation Pack FREE
  use unarmed-idle and get hit animation only
  https://assetstore.unity.com/packages/3d/animations/rpg-character-mecanim-animation-pack-free-65284

- Dungeon Mason: RPG Hero PBR HP Polyart
  Used in main character model
  https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy/rpg-hero-pbr-hp-polyart-121480

- Cinemachine
  Package registered inside Unity officially

- Mixamo
  https://www.mixamo.com/#/
  Used in attack, rolling and dying animations.

- Monsters Sounds Pack
  https://assetstore.unity.com/packages/audio/monsters-sounds-pack-45615
  Used for both monsters' and boss's sound

- 44.1 General Library (Free Sample Pack)
  https://assetstore.unity.com/packages/audio/sound-fx/44-1-general-library-free-sample-pack-145873
  Used the footstep sound

- Blades & Bludgeoning Free Sample Pack
  https://assetstore.unity.com/packages/audio/sound-fx/blades-bludgeoning-free-sample-pack-179306
  used in attack sound

- Treasure Chest Maker
  https://assetstore.unity.com/packages/3d/props/treasure-chest-maker-20850
  used for creating the chest

- sakura of rice and ruin
  http://sakunaofriceandruin.com/
  used as bgm

- Broken Vector - Ultimate Low Poly Dungeon
  https://assetstore.unity.com/packages/3d/environments/dungeons/ultimate-low-poly-dungeon-143535
  used in level editing mainly

- Sugar assets - Dungeon Traps
  https://assetstore.unity.com/packages/3d/environments/dungeons/dungeon-traps-50655
  used in level-editing (traps)

Thank you.