ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐỒ ÁN MÔN HỌC

KỸ THUẬT LẬP TRÌNH NÂNG CAO

Giảng viên hướng dẫn: Ths. Huỳnh Quốc Thịnh

Họ và tên sinh viên:

Nguyễn Xuân Hoàng	21200009
Lê Đức Duy	21200071
Vương Nguyễn Tiến Dũng	21200066

Thành phố Hồ Chí Minh – 2024

MỤC LỤC

DANH MỤC HÌNH ẢNH

DANH MỤC BẢNG

Chương 1: Giới thiệu nội dung và thiết kế trò chơi

Về mặt nội dung, trò chơi tạo dựng một khhung cảnh hầm ngục nơi người chơi phải đối diện với những sinh vật nguy hiểm, không dừng lại ở đó khi đã thoát khỏi hầm ngục thì vẫn còn đó những nguy hiểm đang chờ đợi người chơi.

Về thiết kế, trò chơi được thiết kế hai màn chơi độc lập nhưng vẫn có liên kết với nhau về nội dung. Màn đầu được thiết kế 2D và màn thứ hai là 3D mang lại trải nghiệm độc đáo khác nhau.

Chương 2: Thiết kế 3D stage

2.1. Main menu

2.1.1. ChooseStage

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

public class ChooseStage : MonoBehaviour
{

    public void Stage1()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }


    public void Stage2()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 2);
    }




    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Hàm `ChooseStage()` trong đoạn mã này quản lý việc chọn và tải các màn chơi cụ thể trong trò chơi. Nó cung cấp các phương thức để tải các màn chơi khác nhau.

Hàm `Stage1()`: Dùng để tải màn chơi 1. Tải cảnh có chỉ mục tiếp theo trong build index. `SceneManager.GetActiveScene().buildIndex` trả về chỉ mục của cảnh hiện tại, và việc thêm 1 vào chỉ mục này sẽ tải cảnh tiếp theo. Ví dụ, nếu cảnh hiện tại có chỉ mục là 0, thì hàm này sẽ tải cảnh có chỉ mục 1.

Hàm `Stage2()`: Dùng để tải màn chơi 2. Tải cảnh có chỉ mục cách hai vị trí so với cảnh hiện tại trong build index. `SceneManager.GetActiveScene().buildIndex` trả về chỉ mục của cảnh hiện tại, và việc thêm 2 vào chỉ mục này sẽ tải cảnh có chỉ mục cộng thêm 2. Ví dụ, nếu cảnh hiện tại có chỉ mục là 0, thì hàm này sẽ tải cảnh có chỉ mục 2.

2.1.2. Main menu

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

public class MainMenu : MonoBehaviour
{
    public void Play()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }
    public void Quit()
    {
        Application.Quit();
        Debug.Log("Player Has Quit The Game");
    }


    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Hàm `MainMenu()` trong đoạn mã này quản lý các chức năng của menu chính của trò chơi. Nó cung cấp các phương thức để bắt đầu trò chơi và thoát khỏi ứng dụng.

Hàm `Play()`: Dùng để bắt đầu trò chơi. Tải cảnh tiếp theo trong build index. `SceneManager.GetActiveScene().buildIndex` trả về chỉ mục của cảnh hiện tại, và việc thêm 1 vào chỉ mục này sẽ tải cảnh tiếp theo. Ví dụ, nếu cảnh hiện tại là cảnh thứ nhất (chỉ mục 0), thì hàm này sẽ tải cảnh thứ hai (chỉ mục 1).

Hàm `Quit()`: Dùng để thoát trò chơi. để thoát khỏi ứng dụng. Lưu ý rằng lệnh này chỉ hoạt động khi trò chơi được build và chạy dưới dạng một ứng dụng độc lập; nó sẽ không hoạt động trong Unity Editor. Ghi lại thông điệp "Player Has Quit The Game" trong bảng điều khiển Debug để biết rằng hàm Quit đã được gọi.

2.1.3. Pause menu

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

public class PauseMenu : MonoBehaviour
{
    [SerializeField] GameObject pauseMenu;

    public void Pause()
    {
        pauseMenu.SetActive(true);
        Time.timeScale = 0f;
    }


    public void Home()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene("MainMenu");

    }


    public void Resume()
    {
        pauseMenu.SetActive(false);
        Time.timeScale = 1f;
        Debug.Log("Game Resumed");

    }



    public void Restart()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
```

```
    }




    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Hàm `PauseMenu()`: Trong đoạn mã này quản lý các chức năng của menu tạm dừng trong trò chơi. Nó cung cấp các phương thức để tạm dừng trò chơi, quay lại menu chính, tiếp tục chơi và khởi động lại cấp độ hiện tại.

Hàm `Pause()`: Dùng để tạm dừng trò chơi. Kích hoạt menu tạm dừng `(pauseMenu.SetActive(true))`. Dừng thời gian của trò chơi bằng cách đặt `Time.timeScale` thành 0, nghĩa là mọi hoạt động trong trò chơi sẽ dừng lại.

Hàm `Home()`: Quay trở lại màn hình chính. Khôi phục thời gian của trò chơi về bình thường bằng cách đặt `Time.timeScale` thành 1. Tải lại cảnh "`MainMenu`" bằng `SceneManager.LoaadScene("MainMenu").`

Hàm `Resume()`: Vô hiệu hóa menu tạm dừng bằng hàm:

`(pauseMenu.SetActive(false)).`

Khôi phục thời gian của trò chơi về bình thường bằng cách đặt `Time.timeScale` thành 1. Ghi lại thông điệp "`Game Resumed`" trong bảng điều khiển Debug.

Hàm `Restart()`: Khởi động lại cấp độ hiện tại và khôi phục thời gian của trò chơi về bình thường bằng cách đặt `Time.timeScale` thành 1.

Tải lại cảnh hiện tại bằng:

`SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex)`

với chỉ mục của cảnh hiện tại:

`(SceneManager.GetActiveScene().buildIndex).`

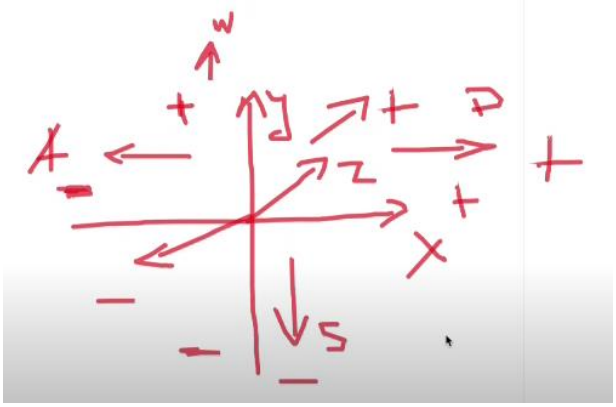
2.2. Những script về nhân vật

2.2.1. Player movement and jump

```
//PlayerMovement.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    private CharacterController character_Controller;
    private Vector3 move_Direction;//calculate xac dinh cai direction ma ta se di chuyen

    public float speed = 5f;//bien cho toc do nhan vat
    public float gravity = 20f;//khai bao trong luc(gravity) hay cac physic law cho vat the

    public float jump_Force = 10f;//khai bao luc nhay cua nhan vat theo chieu doc(y)
    private float vertical_Velocity;



    void Awake()
    {
        character_Controller = GetComponent<CharacterController>();//lay cac thong tin(component) cua
character controller ben "player"
    }


    // Update is called once per frame
    void Update()
    {
        MoveThePlayer();
    }
    void MoveThePlayer()
    {
        move_Direction = new Vector3(Input.GetAxis(Axis.HORIZONTAL), 0f, Input.GetAxis(Axis.VERTICAL));//goi
bien public tu phia ben class Axis qua player movement cho tien
        //vertical up and down,horizontal is left and right

        //print("HORIZONTAL: " + Input.GetAxis("Horizontal"));//neu an cac nut A D tren ban phim
        move_Direction=transform.TransformDirection(move_Direction);//xac dinh local space
```

```
    move_Direction *= speed* Time.deltaTime;//giup tang toc do man hinh
    ApplyGravity();
    character_Controller.Move(move_Direction);//dung de nhan vat di chuyen




  }//move player
void ApplyGravity()
  {
    if (character_Controller.isGrounded)
    {
      vertical_Velocity -= gravity * Time.deltaTime;//giup giu cho nhan vat o duoi dat
      //jump
      PlayerJump();
    }
    else
    {
      vertical_Velocity-=gravity*Time.deltaTime;
    }
     //move_Direction.y=vertical_Velocity * Time.deltaTime;
     move_Direction.y = vertical_Velocity;




    }//apply gravity
  void PlayerJump()
  {
    if (character_Controller.isGrounded && Input.GetKeyDown (KeyCode.Space))
    { //neu nhan vat o tren mat dat va nhan key "space" tren ban phim
      vertical_Velocity = jump_Force;

    }



  }
       }//class
```

`Hàm MoveThePlayer()`: Hàm trên được sử dụng để xác định mặt phẳng 3D cho nhân vật di chuyển tương đương với A S D W.



Mô tả chuyển động của nhân vật

`move_Direction`: Đây thường là một vector (kiểu Vector3 hoặc Vector2) biểu thị hướng di chuyển.

`speed`: Một giá trị float đại diện cho tốc độ di chuyển của đối tượng.

`Time.deltaTime`: Một giá trị float đại diện cho thời gian đã trôi qua kể từ khung hình cuối cùng. Điều này đảm bảo rằng chuyển động không phụ thuộc vào tốc độ khung hình.

`PlayeerJump()`: hàm này sẽ giúp cho Player có thể nhảy khi nhấn **space** và khi nhân vật đang ở trên bề mặt của map với `jump_Force` =10f.

2.2.2. Player attack

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

public class PlayerAttack : MonoBehaviour
{
    private WeaponManager weapon_Manager;

    public float fireRate = 15f;
    private float nextTimeToFire;
    public float damage = 20f;

    private Animator zoomCameraAnim;
    private bool zoomed;

    private Camera mainCam;

    private GameObject crosshair;
```

```
    private bool is_Aiming;


    [SerializeField]
    private GameObject arrow_Prefab, spear_Prefab;

    [SerializeField]
    private Transform arrow_Bow_StartPosition;

    void Awake()
    {
        weapon_Manager = GetComponent<WeaponManager>();

        zoomCameraAnim =
transform.Find(Tags.LOOK_ROOT).transform.Find(Tags.ZOOM_CAMERA).GetComponent<Animator>();
        crosshair = GameObject.FindWithTag(Tags.CROSSHAIR);

        mainCam=Camera.main;

    }
// Start is called before the first frame update
void Start()
{

}

// Update is called once per frame
void Update()
{
    WeaponShoot();
    ZoomInAndOut();


}

void WeaponShoot()
 {
     //if we have assault rifle
     if (weapon_Manager.GetCurrentSelectWeapon().fireType == WeaponFireType.MULTIPLE)
     {
         //if we press and Hold left mouse click AND
         //if Time is greater than the nextTimeToFire
         if (Input.GetMouseButton(0) && Time.time > nextTimeToFire)
         {
```

```
        nextTimeToFire = Time.time + 1f / fireRate;
        weapon_Manager.GetCurrentSelectWeapon().ShootAnimation();

        BulletFired();
    }
}
else
{
    if (Input.GetMouseButtonDown(0))

    {//handle axe
        if (weapon_Manager.GetCurrentSelectWeapon().tag == Tags.AXE_TAG)
        {
            weapon_Manager.GetCurrentSelectWeapon().ShootAnimation();

        }
        //handle shoot
        if (weapon_Manager.GetCurrentSelectWeapon().bulletType == WeaponBulletType.BULLET)
        {
            weapon_Manager.GetCurrentSelectWeapon().ShootAnimation();
            BulletFired();

        }
        else
        {//we have an arrow or bow

            if (is_Aiming)
            {
                weapon_Manager.GetCurrentSelectWeapon().ShootAnimation();
                if(weapon_Manager.GetCurrentSelectWeapon().bulletType == WeaponBulletType.ARROW)
                {
                    ThrowArrowOrSpear(true);
                    // throw arrow
                }
                else if (weapon_Manager.GetCurrentSelectWeapon().bulletType == WeaponBulletType.SPEAR)
                {
                    //throw spear
                   ThrowArrowOrSpear(false);
                }

            }

        }
```

```
        }//if input get mouse button 0
    }
            }
void ZoomInAndOut()
 {
     //we are going to aim wit our camera on the weapon
     if (weapon_Manager.GetCurrentSelectWeapon().weapon_Aim == WeaponAim.AIM)
     {
         //if we press and  hold  right mouse button
         if (Input.GetMouseButtonDown(1))
         {
             zoomCameraAnim.Play(AnimationTags.ZOOM_IN_ANIM);
             crosshair.SetActive(false);
         }

         //release the right button click
         if (Input.GetMouseButtonUp(1))
         {
             zoomCameraAnim.Play(AnimationTags.ZOOM_OUT_ANIM);
             crosshair.SetActive(true);
         }
     }//if we need to zoom the weapon
     if (weapon_Manager.GetCurrentSelectWeapon().weapon_Aim == WeaponAim.SELF_AIM)
     {
         if (Input.GetMouseButtonDown(1))
         {
             weapon_Manager.GetCurrentSelectWeapon().Aim(true);
             is_Aiming = true;
         }
         if (Input.GetMouseButtonUp(1))
         {
             weapon_Manager.GetCurrentSelectWeapon().Aim(false);
             is_Aiming = false;
         }
     }//weapon self aim


             }//zoom in and out
void ThrowArrowOrSpear(bool throwArrow)
   {
       if (throwArrow)
       {
           GameObject arrow = Instantiate(arrow_Prefab);
           arrow.transform.position = arrow_Bow_StartPosition.position;
```

```
            arrow.GetComponent<ArrowBowScript>().Launch(mainCam);

        }
        else
        {
            GameObject spear  = Instantiate(spear_Prefab);
            spear.transform.position = arrow_Bow_StartPosition.position;

            spear.GetComponent<ArrowBowScript>().Launch(mainCam);
        }

    }

    void BulletFired() {
        RaycastHit hit;

        if (Physics.Raycast(mainCam.transform.position, mainCam.transform.forward, out hit)) {
            if (hit.transform.tag == Tags.ENEMY_TAG) {

                hit.transform.GetComponent<HealthScript>().ApplyDamage(damage);

                print("We hit: " + hit.transform.gameObject.name);

            }
        }
    }

}//class
```

Các biến trong script PlayerAttack phục vụ nhiều mục đích khác nhau liên quan đến việc xử lý tấn công bằng vũ khí, nhắm bắn, và hiệu ứng camera trong Unity.

`weapon_Manager`: Một thể hiện của lớp WeaponManager, có khả năng chịu trách nhiệm quản lý vũ khí của người chơi, bao gồm việc chuyển đổi giữa các vũ khí và lấy vũ khí hiện tại được chọn.

`fireRate`: Một giá trị kiểu float xác định tần suất mà người chơi có thể bắn vũ khí. Ví dụ, fireRate là 15f nghĩa là người chơi có thể bắn 15 phát mỗi giây.

`nextTimeToFire:` Một giá trị kiểu float giữ thời gian khi nào người chơi có thể bắn phát tiếp theo. Nó được sử dụng để triển khai tốc độ bắn, đảm bảo rằng vũ khí không bắn thường xuyên hơn mức cho phép bởi fireRate.

damage: Một giá trị kiểu float đại diện cho lượng sát thương mỗi phát bắn hoặc tấn công gây ra cho mục tiêu.

zoomCameraAnim: Một thành phần Animator để xử lý các hoạt ảnh zoom của camera. Nó có thể được sử dụng để tạo hiệu ứng zoom khi người chơi nhắm bắn.

zoomed: Một cờ boolean chỉ ra liệu camera hiện tại có đang được zoom hay không.

mainCam: Một tham chiếu tới camera chính trong cảnh. Nó có thể được sử dụng để điều khiển các hành vi của camera, như zoom hoặc điều chỉnh trường nhìn khi nhắm bắn.

crosshair: Một đối tượng GameObject đại diện cho phần tử giao diện ngắm bắn, giúp người chơi nhắm. Nó thường được hiển thị ở trung tâm của màn hình.

is_Aiming: Một cờ boolean chỉ ra liệu người chơi có đang nhắm bắn hay không. Nó có thể ảnh hưởng đến các hành vi khác nhau, như zoom của camera và hiển thị giao diện ngắm bắn.

arrow_Prefab: Một tham chiếu tới prefab được sử dụng cho mũi tên. Prefab là các GameObject đã được cấu hình sẵn có thể được tạo ra trong trò chơi.

spear_Prefab: Một tham chiếu tới prefab được sử dụng cho giáo. Giống như prefab của mũi tên, nó là một GameObject đã được cấu hình sẵn.

arrow_Bow_StartPosition: Một Transform đại diện cho vị trí bắt đầu khi bắn mũi tên. Đây có thể là vị trí mà mũi tên được tạo ra khi người chơi bắn chúng.

2.2.3. PlayerStat

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

public class PlayerStats : MonoBehaviour
{

    [SerializeField]
    private Image health_Stats, stamina_Stats;



    public void Display_HealthStats(float healthValue)
    {
        healthValue /= 100f;
        health_Stats.fillAmount =healthValue;
```

```
    }
    public void Display_StaminaStats(float staminaValue)
    {
        staminaValue /= 100f;
        stamina_Stats.fillAmount =staminaValue;

    }


    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Hàm `PlayerStats()` trong đoạn mã này quản lý và hiển thị các chỉ số sức khỏe và thể lực của người chơi trong trò chơi. Nó cung cấp các phương thức để cập nhật và hiển thị các chỉ số này trên giao diện người dùng (UI)

Biến thành viên:

`health_Stats`: Biến lưu trữ thành phần Image đại diện cho thanh chỉ số sức khỏe trên UI.

`stamina_Stats`: Biến lưu trữ thành phần Image đại diện cho thanh chỉ số thể lực trên UI.

2.3. Enemy script

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

public class EnemyManager : MonoBehaviour
{

    public static EnemyManager instance;
```

```
    [SerializeField]
    private GameObject boar_Prefab, cannibal_Prefab;

    public Transform[] cannibal_SpawnPoints, boar_SpawnPoint;

    [SerializeField]
    private int cannibal_Enemy_Count,boar_Enemy_Count;


    private int initial_Cannibal_Count, initial_Boar_Count;


    public float wait_Before_Spawn_Enemies_Time = 10f;


    void Awake()
    {
        MakeInstance();

    }
    // Start is called before the first frame update
    void Start()
    {
        initial_Cannibal_Count = cannibal_Enemy_Count;
        initial_Boar_Count = cannibal_Enemy_Count;

        SpawnEnemies();

        StartCoroutine("CheckToSpawnEnemies");

    }
void MakeInstance()
{
    if (instance == null)
    {
        instance=this;
    }
}

 void SpawnEnemies()
{
    SpawnCannibal();
    SpawnBoars();
}
```

```
void SpawnCannibal()
{
    int index = 0;
    for(int i=0;i<cannibal_Enemy_Count;i++)
    {
        if(index >= cannibal_SpawnPoints.Length)
        {
            index = 0;
        }

        Instantiate(cannibal_Prefab, cannibal_SpawnPoints[index].position,Quaternion.identity);
        index++;

    }
    cannibal_Enemy_Count = 0;

}
void SpawnBoars()
{
    int index = 0;
    for (int i = 0; i < boar_Enemy_Count; i++)
    {
        if (index >= boar_SpawnPoint.Length)
        {
            index = 0;
        }

        Instantiate(boar_Prefab, boar_SpawnPoint[index].position, Quaternion.identity);
        index++;

    }
    boar_Enemy_Count = 0;
            }
IEnumerator CheckToSpawnEnemies()
   {
       yield return new WaitForSeconds(wait_Before_Spawn_Enemies_Time);
       SpawnCannibal();

       SpawnBoars();
       StartCoroutine("CheckToSpawnEnemies");

   }
```

```
    public void EnemyDied(bool cannibal)
    {
        if (cannibal)
        {
            cannibal_Enemy_Count++;
            if(cannibal_Enemy_Count > initial_Cannibal_Count)
            {
                cannibal_Enemy_Count =initial_Cannibal_Count;
            }
        }
        else
        {
            boar_Enemy_Count++;
            if(boar_Enemy_Count > initial_Cannibal_Count)
            {
                boar_Enemy_Count=initial_Cannibal_Count;
            }
        }

    }
    public void StopSpawning()
    {
        StopCoroutine("CheckToSpawnEnemies");
    }

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

    }
}
```

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

public class EnemyManager : MonoBehaviour
{

    public static EnemyManager instance;

    [SerializeField]
    private GameObject boar_Prefab, cannibal_Prefab;
```

```
public Transform[] cannibal_SpawnPoints, boar_SpawnPoint;

[SerializeField]
private int cannibal_Enemy_Count,boar_Enemy_Count;


private int initial_Cannibal_Count, initial_Boar_Count;


public float wait_Before_Spawn_Enemies_Time = 10f;
```

Các biến được sử dụng để quản lý kẻ thù trong trò chơi Unity. Dưới đây là giải thích chi tiết về mục đích của từng biến:

`public static EnemyManager instance`: Đây là một biến tĩnh, dùng để tạo ra một instance duy nhất của lớp EnemyManager. Điều này giúp dễ dàng truy cập và quản lý các đối tượng kẻ thù từ các lớp khác trong game.

`[SerializeField] private GameObject boar_Prefab,`

`cannibal_Prefab`: Hai biến này là tham chiếu đến prefab (mẫu) của kẻ thù "boar" và "cannibal". Prefab là các đối tượng mẫu trong Unity, dùng để tạo ra các instance của đối tượng trong game.

`public Transform[] cannibal_SpawnPoints, boar_SpawnPoint`: Hai mảng này chứa các vị trí (Transform) mà kẻ thù "cannibal" và "boar" sẽ được sinh ra (spawn) trên bản đồ.

`[SerializeField] private int cannibal_Enemy_Count,`

`boar_Enemy_Count:` Hai biến này lưu số lượng kẻ thù "cannibal" và "boar" hiện tại trong game. Chúng được gán giá trị thông qua Inspector trong Unity.

`private int initial_Cannibal_Count, initial_Boar_Count`: Hai biến này lưu số lượng ban đầu của kẻ thù "cannibal" và "boar". Chúng có thể được dùng để khởi tạo hoặc thiết lập lại số lượng kẻ thù trong game.

`public float wait_Before_Spawn_Enemies_Time = 10f`: Biến này xác định khoảng thời gian chờ (tính bằng giây) trước khi sinh ra các kẻ thù mới sau mỗi đợt spawn. Nó giúp tạo ra khoảng nghỉ giữa các lần xuất hiện của kẻ thù, đảm bảo rằng trò chơi không bị quá tải với quá nhiều kẻ thù cùng một lúc.

Mục đích của hàm `SpawnCannibal`**:** Hàm này có nhiệm vụ chính là quản lý việc sinh ra các kẻ thù "cannibal" tại các vị trí chỉ định trong game. Các kẻ thù sẽ được sinh ra theo

vòng lặp và được đặt tại các điểm spawn tuần tự. Nếu hết các điểm spawn, quá trình sinh ra sẽ quay lại từ đầu danh sách điểm spawn.

Mục đích của hàm `SpawnBoars`**:** Hàm này có nhiệm vụ chính là quản lý việc sinh ra các kẻ thù "boar" tại các vị trí chỉ định trong game. Các kẻ thù sẽ được sinh ra theo vòng lặp và được đặt tại các điểm spawn tuần tự. Nếu hết các điểm spawn, quá trình sinh ra sẽ quay lại từ đầu danh sách điểm spawn.

Chương 3: Thiết kế 2D stage

3.1. Player script

3.1.1. Player movement

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

public class Playermovement : MonoBehaviour
{
    [SerializeField] float speed = 1f;
    [SerializeField] float jump_heigh = 1f;
    [SerializeField] float Attack2Cooldown = 0;
    private Rigidbody2D player;
    private Animator player_animation;

    bool Player_isGround;
    bool Fall_detect = false;
    bool Double_jump = false;

    private float speedIn = 0f;
    private float attack2CooldownTimer = Mathf.Infinity;

    private void Awake()
    {
        player = GetComponent<Rigidbody2D>();
        player_animation = GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKey(KeyCode.LeftShift))
```

```
{
    speedIn = speed * 2f;
}
else speedIn = speed;

float HorizontalInput = Input.GetAxis("Horizontal");

player.velocity = new Vector2(HorizontalInput * speedIn, player.velocity.y);

if (HorizontalInput > 0f ) transform.localScale = new Vector3(3.5f, 3.5f, 3.5f);
if (HorizontalInput < 0f ) transform.localScale = new Vector3(-3.5f, 3.5f, 3.5f);

if (Input.GetKeyDown(KeyCode.Space) && (Player_isGround ^ Double_jump))
{
    player.velocity = new Vector2(player.velocity.x, jump_heigh);
    player_animation.SetTrigger("Jump");
    Player_isGround = false;
    Double_jump = !Double_jump;
}
if (player.velocity.y < -2f)
{
    Fall_detect = true;
    Player_isGround = false;
}
else Fall_detect = false;

player_animation.SetBool("Player_run_check", HorizontalInput != 0f);
player_animation.SetBool("Player_isGrounded", Player_isGround);
player_animation.SetBool("Fall", Fall_detect);

//attack movement

attack2CooldownTimer += Time.deltaTime;

if (Input.GetKeyDown(KeyCode.Mouse0))
{
    player_animation.SetBool("Attack1", true);
}
else player_animation.SetBool("Attack1", false);

if (Input.GetKeyDown(KeyCode.Mouse1))
{
    if (attack2CooldownTimer >= Attack2Cooldown)
```

```
        {
            player_animation.SetBool("Attack2", true);
            attack2CooldownTimer = 0;
        }
    }
    else player_animation.SetBool("Attack2", false);
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Ground_map")
    {
        Player_isGround = true;
        Double_jump = false;
    }

}
}
```

Đoạn mã này xử lý chuyển động, nhảy và hoạt ảnh tấn công của người chơi trong một trò chơi 2D Unity. Người chơi có thể di chuyển sang trái hoặc phải, nhảy (với tính năng nhảy kép), và thực hiện hai loại tấn công.

Biến thành viên:

`float speed`: Tốc độ di chuyển của người chơi.

`float jump_heigh`: Chiều cao của cú nhảy.

`float Attack2Cooldown`: Thời gian hồi chiêu giữa các lần thực hiện Attack2.

`Rigidbody2D player`: Thành phần Rigidbody2D gắn với người chơi, được sử dụng cho các tính toán vật lý.

`Animator player_animation`: Thành phần Animator dùng để điều khiển hoạt ảnh của người chơi.

`bool Player_isGround`: Biến xác định liệu người chơi có đang đứng trên mặt đất hay không.

`bool Fall_detect`: Biến xác định liệu người chơi có đang rơi hay không.

`bool Double_jump`: Biến xác định liệu người chơi có thể thực hiện nhảy kép hay không.

`float speedIn`: Tốc độ hiện tại của người chơi, có thể thay đổi khi nhấn phím Shift.

`float attack2CooldownTimer`: Bộ đếm thời gian cho lần tấn công thứ hai, khởi đầu bằng giá trị vô cực.

`Awake():` Phương thức này được gọi khi đối tượng được khởi tạo. Nó lấy các thành phần Rigidbody2D và Animator từ đối tượng người chơi.

`Update():` Phương thức này được gọi mỗi khung hình. Nó xử lý các hành động và trạng thái của người chơi như di chuyển, nhảy và tấn công.

`OnCollisionEnter2D(Collision2D collision):` Phương thức này được gọi khi người chơi va chạm với đối tượng khác. Nó xác định liệu người chơi có đang trên mặt đất hay không dựa trên thẻ của đối tượng va chạm. Phần này đảm bảo rằng người chơi có thể nhảy lại khi tiếp đất và đặt lại biến **Double_jump** để cho phép nhảy kép trong lần tiếp theo.

3.1.2. Player attack

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

public class Player_attack : MonoBehaviour
{
    // Start is called before the first frame update
    [SerializeField] private int Damage1;
    [SerializeField] private int Damage2;

    [SerializeField] private Collider2D SwordCollider;

    private int DamageToEnemy;

    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Mouse0)) DamageToEnemy = Damage1;
        if (Input.GetKeyDown(KeyCode.Mouse1)) DamageToEnemy = Damage2;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Enemy")
```

```
    {
        Debug.Log(collision.tag + " " + collision.GetComponent<Enemy_Health>().currentHealth);
        collision.GetComponent<Enemy_Health>().TakeDamage(DamageToEnemy);
        //if (Input.GetKeyDown(KeyCode.Mouse1))
collision.GetComponent<Enemy_Health>().TakeDamage(Damage2);

    }
  }
}
```

Đoạn mã này xử lý các hành động tấn công của người chơi trong một trò chơi 2D Unity. Người chơi có thể tấn công bằng hai loại đòn khác nhau, gây sát thương lên kẻ địch khi va chạm với chúng.

Biến thành viên:

`int Damage1`: Sát thương của đòn tấn công thứ nhất (nhấn chuột trái).

`int Damage2`: Sát thương của đòn tấn công thứ hai (nhấn chuột phải).

`Collider2D SwordCollider`: Bộ va chạm của vũ khí, được sử dụng để phát hiện va chạm với kẻ địch.

`int DamageToEnemy`: Sát thương hiện tại sẽ gây ra cho kẻ địch, thay đổi dựa trên đòn tấn công được sử dụng.

`Start()`: Phương thức này được gọi khi đối tượng được khởi tạo. Trong đoạn mã này, phương thức Start không thực hiện bất kỳ hành động nào.

`Update()`: Phương thức này được gọi mỗi khung hình. Nó kiểm tra các phím bấm để xác định loại đòn tấn công và cập nhật giá trị sát thương tương ứng.

Nếu nhấn phím `Mouse0` (chuột trái), cập nhật sát thương hiện tại là `Damage1`.

Nếu nhấn phím `Mouse1` (chuột phải), cập nhật sát thương hiện tại là `Damage2`.

`OnTriggerEnter2D(Collider2D collision)`: Phương thức này được gọi khi bộ va chạm của vũ khí va chạm với một đối tượng khác. Nó kiểm tra nếu đối tượng va chạm có thẻ "`Enemy`" và gây sát thương cho kẻ địch. Nếu đối tượng va chạm có thẻ là "`Enemy`", thực hiện các hành động sau:

In ra thẻ của đối tượng và máu hiện tại của kẻ địch.

Gọi phương thức `TakeDamage` của đối tượng `Enemy_Health` để gây sát thương cho kẻ địch với giá trị `DamageToEnemy`.

Phần này đảm bảo rằng khi vũ khí của người chơi va chạm với kẻ địch, kẻ địch sẽ nhận sát thương tương ứng với loại đòn tấn công được sử dụng.

3.1.3. Health

```
using System;
using UnityEngine;

public class Health : MonoBehaviour
{
    [SerializeField] public float startingHealth;
    public bool dead = false;
    public float currentHealth { get; private set; }
    private Animator anim;

    private void Awake()
    {
        currentHealth = startingHealth;
        anim = GetComponent<Animator>();
    }

    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
        if (currentHealth > startingHealth) currentHealth = startingHealth;
        if (currentHealth < 0) currentHealth = 0;

        if (currentHealth > 0)
        {
            // take hurt
            anim.SetTrigger("Hurt");
        }
        else
        {
            // go dead
            anim.SetTrigger("Die");
            GetComponent<Playermovement>().enabled = false;
            dead = true;
        }
    }
    public void AddHealth(int health)
    {
        currentHealth += health;
        if (currentHealth > startingHealth) currentHealth = startingHealth;
        if (currentHealth < 0) currentHealth = 0;
    }
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.H)) { currentHealth = startingHealth; }
    }
```

```
}
```

Đoạn mã này xử lý máu của một đối tượng trong trò chơi Unity. Nó kiểm tra máu, cho phép đối tượng nhận sát thương, hồi phục máu và xử lý các trạng thái như "hurt" và "die".

`float startingHealth`: Máu ban đầu của đối tượng.

`bool dead`: Biến xác định liệu đối tượng đã chết hay chưa.

`float currentHealth`: Máu hiện tại của đối tượng, chỉ có thể được truy cập từ bên ngoài.

`Animator anim`: Thành phần Animator dùng để điều khiển hoạt ảnh của đối tượng.

`Awake()`: Phương thức này được gọi khi đối tượng được khởi tạo. Nó thiết lập máu hiện tại bằng máu ban đầu và lấy thành phần Animator từ đối tượng.

`TakeDamage(int damage)`: Phương thức này được gọi khi đối tượng nhận sát thương. Nó giảm máu hiện tại theo giá trị sát thương, sau đó cập nhật hoạt ảnh và trạng thái của đối tượng.

`AddHealth(int health)`: Phương thức này được gọi khi đối tượng nhận thêm máu. Nó tăng máu hiện tại theo giá trị máu được thêm vào.

Phần này đảm bảo rằng đối tượng có thể nhận sát thương, hồi phục máu, và kích hoạt các hoạt ảnh và trạng thái tương ứng dựa trên máu hiện tại.

3.2. Enemy script

3.2.1. MeleeEnemy

```
using UnityEngine;

public class MeleeEnemy : MonoBehaviour
{
    [SerializeField] private CapsuleCollider2D capsuleCollider;
    [SerializeField] private LayerMask playerLayer;
    [SerializeField] private Enemy_group MeleeEnemy_details;

    private float attackCooldown;
    private float range;
    private float distance;
    private int damage;
    private int criticalDamage;

    private float cooldownTimer = Mathf.Infinity;
    private int randomAttack;

    private Animator anim;
```

```
    private Health playerHealth;

    private Patroll enemyPatrol;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<Patroll>();
    }
    private void Start()
    {
        attackCooldown = MeleeEnemy_details.attackCooldown;
        range = MeleeEnemy_details.detectRange;
        distance = MeleeEnemy_details.detectDistance;
        damage = MeleeEnemy_details.damage;
        criticalDamage = MeleeEnemy_details.criticalDamage;
    }
    private void Update()
    {
        cooldownTimer += Time.deltaTime;

        //attack only player in sight
        if (PlayerInSight())
        {
            if (cooldownTimer >= attackCooldown)
            {
                //attack
                if (!playerHealth.dead)
                {
                    cooldownTimer = 0;
                    if (randomAttack <= 3) anim.SetTrigger("Attack1");
                    else anim.SetTrigger("Attack2");
                }
            }
        }

        if(enemyPatrol != null) enemyPatrol.enabled = !PlayerInSight();

    }
    private bool PlayerInSight()
    {
        RaycastHit2D hit = Physics2D.BoxCast(capsuleCollider.bounds.center + transform.right * distance *
(transform.localScale.x / Mathf.Abs(transform.localScale.x)),
            new Vector3(capsuleCollider.bounds.size.x * range, capsuleCollider.bounds.size.y,
capsuleCollider.bounds.size.z), 0, Vector2.left, 0, playerLayer);

        if (hit.collider != null) playerHealth = hit.collider.GetComponent<Health>();

        return (hit.collider != null);
```

```
    }
    private void DamagePlayer()
    {
        if (PlayerInSight())
        {
            //Damage player health
            randomAttack = Random.Range(0, 10);
            if (randomAttack <= 3) playerHealth.TakeDamage(damage);
            else playerHealth.TakeDamage(criticalDamage);
        }
    }
    private void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireCube(capsuleCollider.bounds.center + transform.right * distance * (transform.localScale.x
/ Mathf.Abs(transform.localScale.x)),
            new Vector3(capsuleCollider.bounds.size.x * range, capsuleCollider.bounds.size.y,
capsuleCollider.bounds.size.z));
    }
}
```

Đoạn mã này xử lý hành vi của một enemy cận chiến trong trò chơi Unity. Enemy này có thể phát hiện người chơi trong phạm vi xác định, tấn công người chơi khi có thể và tuần tra khi không phát hiện thấy người chơi.

Biến thành viên:

`CapsuleCollider2D capsuleCollider`: Bộ va chạm của enemy, được sử dụng để phát hiện người chơi.

`LayerMask playerLayer`: Lớp đối tượng của người chơi, dùng để kiểm tra va chạm.

`Enemy_group MeleeEnemy_details`: Thông tin chi tiết về enemy cận chiến, bao gồm thời gian hồi chiêu, phạm vi phát hiện, khoảng cách phát hiện, sát thương và sát thương chí mạng.

`float attackCooldown`: Thời gian hồi chiêu giữa các đòn tấn công.

`float range`: Phạm vi phát hiện người chơi.

`float distance`: Khoảng cách phát hiện người chơi.

`int damage`: Sát thương của đòn tấn công thông thường.

`int criticalDamage`: Sát thương của đòn tấn công chí mạng.

`float cooldownTimer`: Bộ đếm thời gian để theo dõi thời gian hồi chiêu, khởi đầu bằng giá trị vô cực.

`int randomAttack`: Biến ngẫu nhiên để xác định loại đòn tấn công.

`Animator anim`: Thành phần Animator dùng để điều khiển hoạt ảnh của enemy.

`Health playerHealth`: Thành phần Health của người chơi, được sử dụng để giảm máu của người chơi khi bị tấn công.

`Patroll enemyPatrol`: Thành phần Patroll dùng để điều khiển hành vi tuần tra của enemy.

`Awake()`: Phương thức này được gọi khi đối tượng được khởi tạo. Nó lấy các thành phần Animator và Patroll từ đối tượng enemy.

`Start()`: Phương thức này được gọi khi đối tượng bắt đầu hoạt động. Nó khởi tạo các biến bằng các giá trị từ MeleeEnemy_details.

`Update()`: Phương thức này được gọi mỗi khung hình. Nó tăng bộ đếm thời gian hồi chiêu, kiểm tra xem người chơi có trong tầm nhìn hay không và thực hiện tấn công nếu có thể.

Nếu người chơi trong tầm nhìn và thời gian hồi chiêu đã hết, thực hiện tấn công bằng cách kích hoạt hoạt ảnh "`Attack1`" hoặc "`Attack2`" dựa trên giá trị ngẫu nhiên.

Vô hiệu hóa hoặc kích hoạt hành vi tuần tra dựa trên việc người chơi có trong tầm nhìn hay không.

`PlayerInSight()`: Phương thức này kiểm tra xem người chơi có trong tầm nhìn của enemy hay không bằng cách sử dụng BoxCast là một hàm được xây dựng sẵn trong unity

`DamagePlayer()`: Phương thức này gây sát thương cho người chơi khi người chơi trong tầm nhìn. Biến randomAttack dùng để cho phép ngẫu nhiên enemy gây sát thương thường hay trọng kích, có 30% khả năng gây sát thương trọng kích. Và biến randomAttack sẽ được đưa vào hàm phương thức update() để thay đổi hoạt ảnh cho phù hợp với từng loại sát thương.

`OnDrawGizmos()`: Phương thức này vẽ một hình hộp trong chế độ xem Scene của Unity để trực quan hóa phạm vi phát hiện của enemy. Phương thức này dùng trong thiết kế để nhận biết tầm nhìn của enemy, không hiển thị trên phần game chính.

Phần này đảm bảo rằng enemy cận chiến có thể phát hiện người chơi, tấn công khi có thể và tuần tra khi không phát hiện thấy người chơi.

3.2.2. Patroll

```
using UnityEngine;
```

```
public class Patroll : MonoBehaviour
{
    [SerializeField] private Transform leftEdge;
    [SerializeField] private Transform rightEdge;
    [SerializeField] private Transform enemyTaget;
    [SerializeField] private Animator anim;

    [SerializeField] private Enemy_group Enemy_details;

    private float Speed;
    private float idleDuration;

    private bool MovingLeft = false;
    private float idleTimer;

    private void Start()
    {
        Speed = Enemy_details.Speed;
        idleDuration = Enemy_details.idleDuration;
    }
    private void Update()
    {
        if (MovingLeft)
        {
            if (enemyTaget.position.x >= leftEdge.position.x) MoveInDirection(-1);
            else
            {
                idleTimer += Time.deltaTime;
                if (idleTimer >= idleDuration) MovingLeft = !MovingLeft;
                anim.SetBool("moving", false);
            }
        }
        else
        {
            if (enemyTaget.position.x <= rightEdge.position.x) MoveInDirection(1);
            else
            {
                idleTimer += Time.deltaTime;
                if (idleTimer >= idleDuration) MovingLeft = !MovingLeft;
                anim.SetBool("moving", false);
            }
        }
    }

    private void MoveInDirection(int direction)
    {
        idleTimer = 0;
        anim.SetBool("moving", true);
```

```
        enemyTaget.localScale = new Vector2(Mathf.Abs(enemyTaget.localScale.x) * direction,
enemyTaget.localScale.y);
        enemyTaget.position = new Vector2(enemyTaget.position.x + Time.deltaTime * direction * Speed,
enemyTaget.position.y);
    }

    private void OnDisable()
    {
        anim.SetBool("moving", false);

    }
}
```

Đoạn mã này xử lý hành vi tuần tra của một enemy trong trò chơi Unity. Enemy di chuyển qua lại giữa hai điểm được xác định trước và dừng lại trong một khoảng thời gian cố định tại mỗi điểm trước khi quay đầu di chuyển lại.

Biến thành viên:

`Transform leftEdge`: Điểm giới hạn bên trái mà enemy tuần tra đến.

`Transform rightEdge`: Điểm giới hạn bên phải mà enemy tuần tra đến.

`Transform enemyTaget`: Điểm mục tiêu mà enemy di chuyển tới.

`Animator anim`: Thành phần Animator dùng để điều khiển hoạt ảnh của enemy.

`Enemy_group Enemy_details`: Thông tin chi tiết về enemy, bao gồm tốc độ di chuyển và thời gian chờ.

`float Speed`: Tốc độ di chuyển của enemy.

`float idleDuration`: Thời gian chờ khi enemy đến một trong hai giới hạn tuần tra.

`bool MovingLeft`: Biến xác định hướng di chuyển của enemy, ban đầu được đặt là di chuyển sang trái.

`float idleTimer`: Bộ đếm thời gian chờ khi enemy đến một trong hai giới hạn tuần tra.

`Start()`: Phương thức này được gọi khi đối tượng bắt đầu hoạt động. Nó khởi tạo các biến Speed và idleDuration bằng các giá trị từ Enemy_details.

`Update()`: Phương thức này được gọi mỗi khung hình. Nó xử lý việc di chuyển của enemy và thời gian chờ khi enemy đến một trong hai giới hạn tuần tra.

Nếu enemy đang di chuyển sang trái và chưa đạt đến giới hạn bên trái, gọi `MoveInDirection(-1)` để tiếp tục di chuyển sang trái.

Nếu enemy đang di chuyển sang phải và chưa đạt đến giới hạn bên phải, gọi `MoveInDirection(1)` để tiếp tục di chuyển sang phải.

Nếu enemy đã đạt đến một trong hai giới hạn, tăng idleTimer theo thời gian trôi qua. Nếu `idleTimer` lớn hơn hoặc bằng `idleDuration`, đổi hướng di chuyển `(MovingLeft = !MovingLeft)` và đặt hoạt ảnh moving thành false.

`MoveInDirection(int direction):` Phương thức này xử lý việc di chuyển của enemy theo hướng được chỉ định và đặt lại bộ đếm thời gian chờ.

Đặt lại `idleTimer` bằng 0.

Đặt hoạt ảnh `moving` thành `true`.

Điều chỉnh hướng của enemy bằng cách thay đổi `localScale` của `enemyTaget`.

Di chuyển `enemyTaget` theo hướng chỉ định với tốc độ được xác định bởi `Speed`. **`OnDisable():`** Phương thức này được gọi khi đối tượng bị vô hiệu hóa. Nó đặt hoạt ảnh moving thành false.

3.2.3. Enemy health

```
using UnityEngine;

public class Enemy_Health : MonoBehaviour
{
    [SerializeField] private Enemy_group Enemy_details;
    [SerializeField] private Rigidbody2D rigid_body;
    private float startingHealth;
    public bool dead = false;
    public float currentHealth { get; private set; }
    private Animator anim;


    private void Awake()
    {
        startingHealth = Enemy_details.startingHealth;
        currentHealth = startingHealth;
        anim = GetComponent<Animator>();
    }

    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
```

```
        if (currentHealth > startingHealth) currentHealth = startingHealth;
        if (currentHealth < 0) currentHealth = 0;

        if (currentHealth > 0)
        {
            // take hurt
            anim.SetTrigger("Hurt");
        }
        else
        {
            // go dead
            if (!dead)
            {
                anim.SetTrigger("Die");
                GetComponentInParent<Patroll>().enabled = false;
                GetComponent<MeleeEnemy>().enabled = false;
                rigid_body.simulated = false;
                dead = true;

            }
        }
    }
    private void Update()
    {

    }
}
```

Đoạn mã này quản lý máu của enemy trong trò chơi Unity. Khi enemy nhận sát thương, mã sẽ giảm máu hiện tại của enemy, hiển thị hoạt ảnh bị thương hoặc chết, và vô hiệu hóa các thành phần điều khiển di chuyển và tấn công của enemy khi nó chết.

Biến thành viên:

`Enemy_group Enemy_details`: Thông tin chi tiết về enemy, bao gồm máu ban đầu.

`Rigidbody2D rigid_body`: Thành phần Rigidbody2D của enemy, dùng để xử lý vật lý.

`float startingHealth`: Máu ban đầu của enemy, lấy từ Enemy_details.

`bool dead`: Biến đánh dấu enemy đã chết hay chưa.

`float currentHealth`: Máu hiện tại của enemy, chỉ có thể được truy cập bên ngoài lớp thông qua thuộc tính `currentHealth`.

`Animator anim`: Thành phần Animator dùng để điều khiển hoạt ảnh của enemy.

`Awake()`: Phương thức này được gọi khi đối tượng được tạo ra. Nó khởi tạo các biến `startingHealth` và `currentHealth` bằng các giá trị từ `Enemy_details`, và lấy thành phần `Animator`.

`TakeDamage(int damage)`: Phương thức này được gọi khi enemy nhận sát thương. Nó giảm máu hiện tại của enemy theo giá trị sát thương, và xử lý các hoạt ảnh tương ứng.

Phần này đảm bảo rằng enemy có thể nhận sát thương, hiển thị hoạt ảnh bị thương khi còn sống, và vô hiệu hóa các thành phần điều khiển di chuyển và tấn công khi chết.

3.2.4. Enemy group

```
using UnityEngine;

public class Enemy_group : MonoBehaviour
{
    // Start is called before the first frame update
    [Header ("Details")]
    [SerializeField] public int damage;
    [SerializeField] public int criticalDamage;
    [SerializeField] public float startingHealth;
    [SerializeField] public float attackCooldown;
    [SerializeField] public float detectRange;
    [SerializeField] public float detectDistance;

    [Header ("Patrolling")]
    [SerializeField] public float Speed;
    [SerializeField] public float idleDuration;
}
```

Chứa các biến điều chỉnh thông số cho enemy, được gọi trong 3 script `Patroll.cs`, `Enemy_health.cs`, `MeleeEnemy.cs` và được truyền thông qua `Serialize Field` của unity. Sử dụng câu lệnh dưới đây để gọi:

```
[SerializeField] private Enemy_group Enemy_details
```

3.3. Những script khác

3.3.1. Heal collect

```
using UnityEngine;

public class HealCollect : MonoBehaviour
{
    [SerializeField] private int healthValue;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        Debug.Log("collide with object");
```

```
        if (collision.tag == "Player")
        {
            Debug.Log("collide with player");
            collision.GetComponent<Health>().AddHealth(healthValue);
            gameObject.SetActive(false);

        }
    }
}
```

Đoạn mã này quản lý hành vi của một vật phẩm hồi máu trong trò chơi Unity. Khi người chơi va chạm với vật phẩm hồi máu, máu của người chơi sẽ được tăng lên và vật phẩm hồi máu sẽ biến mất khỏi trò chơi.

3.3.2. Health bar

```
using UnityEngine;
using UnityEngine.UI;

public class HealthBar : MonoBehaviour
{
    [SerializeField] private Health playerHealth;
    [SerializeField] private Image currentHealthBar;
    private void Update()
    {
        currentHealthBar.fillAmount = playerHealth.currentHealth / playerHealth.startingHealth;
    }
}
```

Đoạn mã này cập nhật máu hiện tại của người chơi lên thanh máu trong UI.

Hình 3.1 Vật phẩm hồi máu và thanh máu

3.3.3. Camera controller

```
using UnityEngine;
```

```
public class Cameracontroller : MonoBehaviour
{
    [SerializeField] private float Camera_speed = 2f;
    public Transform player;
    private int left_right;

    [SerializeField] private float offset_x = 2f;
    [SerializeField] private float offset_y = 2f;

    Vector3 velocity = Vector3.zero;

    private void FixedUpdate()
    {
        if ((player.position.x > -8f && player.position.x < 6f) && (player.position.y > -43f && player.position.y < -
36f))
        {
            transform.position = Vector3.SmoothDamp(transform.position, new Vector3(-1.2f, -39.3f, -10f), ref
velocity, Camera_speed);
            //this.transform.position = new Vector3(-1.2f, -39.3f, -10f);
        }
        else
        {
            if (Input.GetAxis("Horizontal") > 0f)
            {
                transform.position = Vector3.SmoothDamp(transform.position, new Vector3(player.position.x +
offset_x, player.position.y + offset_y, -10f), ref velocity, Camera_speed);
                left_right = 1;
            }
            if (Input.GetAxis("Horizontal") < 0f)
            {
              transform.position = Vector3.SmoothDamp(transform.position, new Vector3(player.position.x -
offset_x, player.position.y + offset_y, -10f), ref velocity, Camera_speed);
                left_right = -1;
            }
            if (Input.GetAxis("Vertical") < 0f)
                transform.position = Vector3.SmoothDamp(transform.position, new Vector3(player.position.x,
player.position.y - 2 * offset_y, -10f), ref velocity, Camera_speed);

            else
                transform.position = Vector3.SmoothDamp(transform.position, new Vector3(player.position.x +
left_right*offset_x, player.position.y + offset_y, -10f), ref velocity, Camera_speed);

        }
    }
}
```

Nếu người chơi di chuyển sang phải `(Input.GetAxis("Horizontal") > 0f)`, camera sẽ theo sau với một khoảng cách bù trừ theo trục x và y.

Nếu người chơi di chuyển sang trái `(Input.GetAxis("Horizontal") < 0f)`, camera sẽ theo sau với một khoảng cách bù trừ ngược lại.

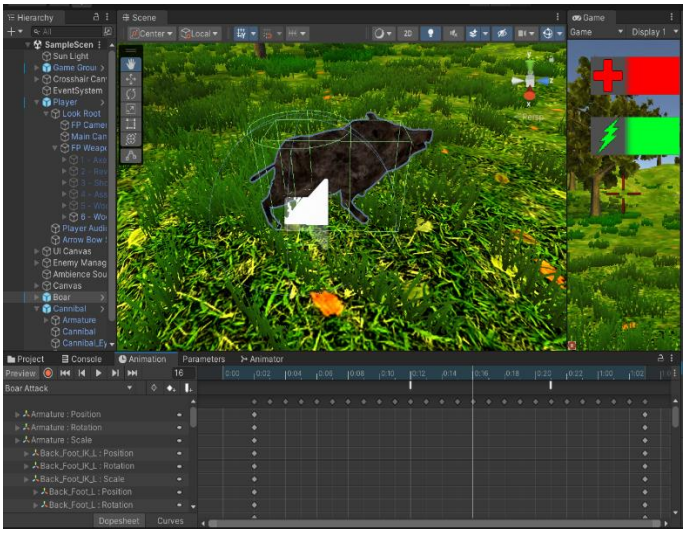
Nếu người chơi di chuyển xuống `(Input.GetAxis("Vertical") < 0f)`, camera sẽ theo sau với một khoảng cách bù trừ theo trục y lớn hơn.

Trong các trường hợp khác, camera sẽ theo sau người chơi với khoảng cách bù trừ đã lưu trong biến `left_right`.

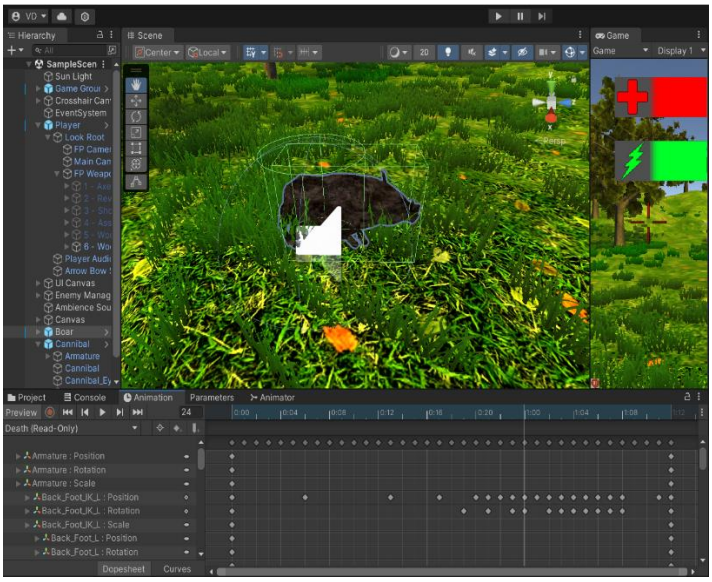
Hình 3.2 Vị trí spawn nhân vật

Chương 4: Một số hình ảnh trong dự án

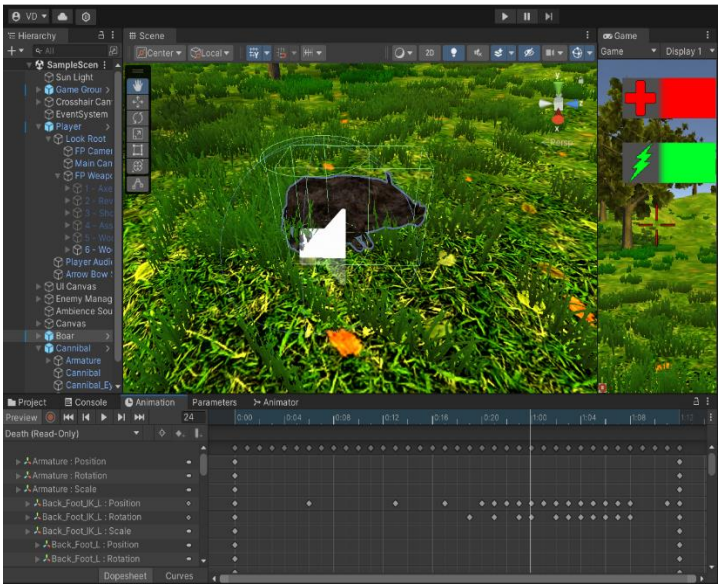
4.1. 3D stage



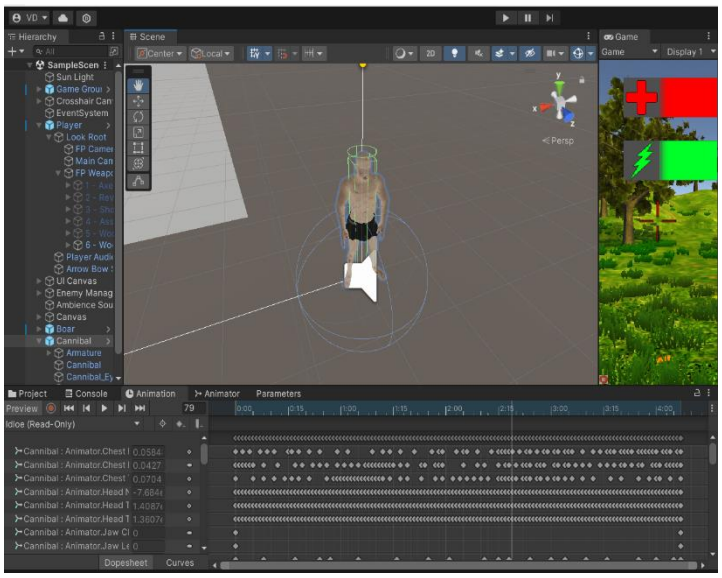
Hình 4.1 Enemy attack (boar)



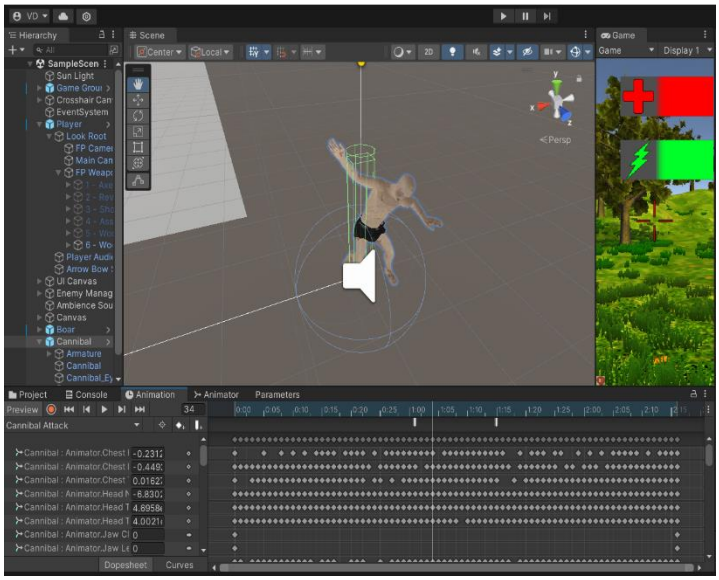
Hình 4.2 Enemy ildle (boar)



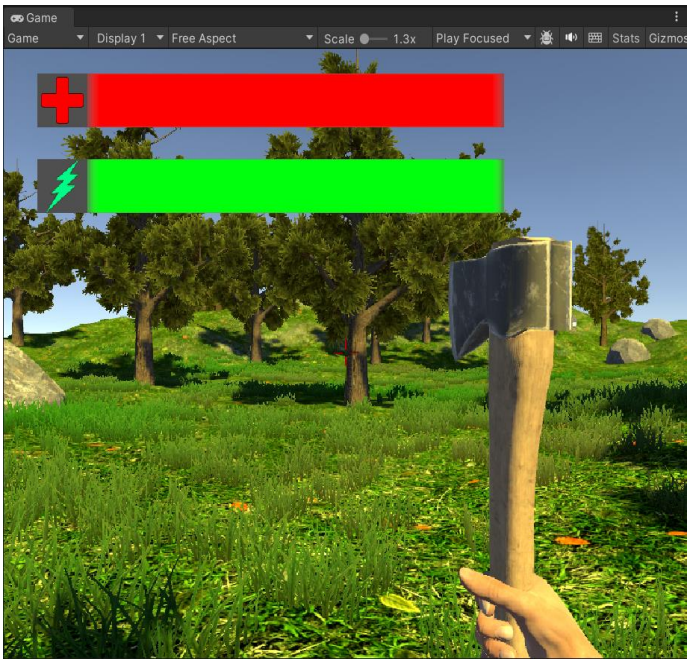
Hình 4.3 Enemy death (boar)



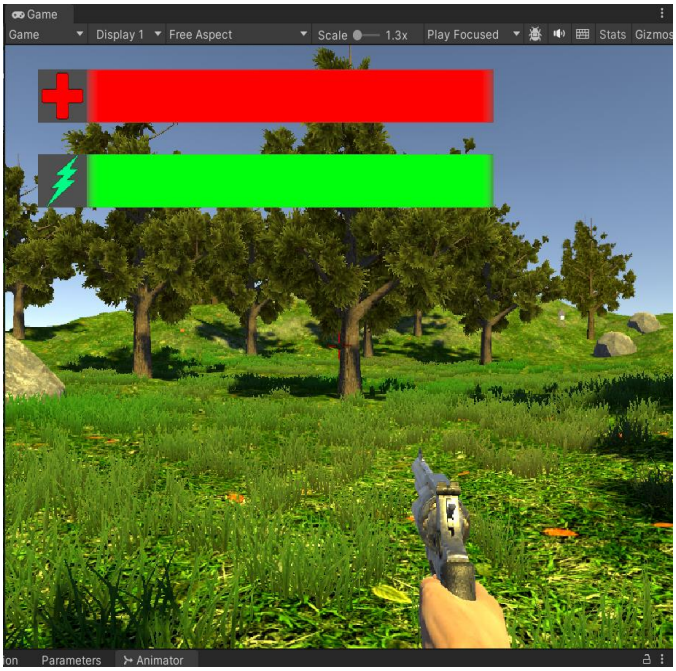
Hình 4.4 Enemy ildle (canibal)



Hình 4.5 Enemy attack (canibal)



Hình 4.6 Vũ khí trong 3D stage (rìu)



Hình 4.7 Vũ khí trong 3D stage (súng)

4.2. 2D stage

Hình 4.8 Hình ảnh trong 2D stage

Hình 4.9 Hình ảnh trong 2D stage

Hình 4.10 Hình ảnh trong 2D stage