# **2 cách tham chiếu đến class khác**

# **1.tham chiếu bằng tên class và biến trung gian**

Giả sử muốn truy cập để lấy thông tin 1 class là info.cs

Tạo 1 class khác và tạo biến trung gian là :[Serializedfield] info getinfo;

Getinfo sẽ có thể lấy đc tất cả các biến và phương thức có phạm vi public trở lên

# **2.tham chiếu bằng namespace**

## **A. mô hình CMMGUV**

### **Core:**

Chịu trách nhiệm cho các tính năng cốt lõi và không phụ thuộc vào bất kỳ module nào khác.

Chứa các lớp, hàm và utilities dùng chung cho toàn bộ dự án.

Ví dụ: Quản lý vòng đời, quản lý tài nguyên, quản lý cấu hình, quản lý sự kiện, v.v.

### **Model:**

Chịu trách nhiệm cho các lớp mô tả dữ liệu và logic nghiệp vụ.

Chứa các lớp đại diện cho các thực thể trong game, chẳng hạn như nhân vật, kẻ địch, vũ khí, v.v.

Không chứa bất kỳ logic giao diện người dùng hoặc cơ chế game play nào.

### **Mechanic:**

Chịu trách nhiệm cho các hệ thống và tính năng game play.

Chứa các lớp và logic liên quan đến các cơ chế game play như di chuyển, tấn công, phòng ngự, v.v.

Tương tác với các lớp trong namespace Model để thực hiện các tính năng game play.

### **GamePlayer:**

Chịu trách nhiệm cho logic điều khiển và hành vi của người chơi.

Chứa các lớp và logic liên quan đến việc kiểm soát các hành động của người chơi, chẳng hạn như nhập liệu, điều khiển nhân vật, v.v.

Tương tác với các lớp trong namespace Mechanic để thực hiện các hành động của người chơi.

### **UI:**

Chịu trách nhiệm cho giao diện người dùng và các tính năng liên quan.

Chứa các lớp và logic liên quan đến các phần tử giao diện như menu, HUD, popup, v.v.

Tương tác với các lớp trong namespace Model và Mechanic để hiển thị và cập nhật thông tin cho người chơi.

### **View:**

Chịu trách nhiệm cho việc hiển thị các thực thể trong game.

Chứa các lớp và logic liên quan đến việc vẽ, hiệu ứng, animation, v.v. cho các đối tượng trong game.

Tương tác với các lớp trong namespace Model để lấy dữ liệu và hiển thị chúng.

## **B. Mô hình MVC**

Đọc lại phần A.

Model tương ứng với thành phần "model".

View tương ứng với "view" và "UI".

Controller tương ứng với "core", "mechanic" và "gameplay".

## **Ví dụ**

### **1.core**

namespace Core

{

public class GameManager : MonoBehaviour

{

public void InitGame()

{

// Khởi tạo game

}

public void QuitGame()

{

// Kết thúc game

}

}

public static class ResourceManager

{

public static void LoadAsset(string assetPath)

{

// Tải tài nguyên

}

}

}

### **2.model**

namespace Model

{

public class Player

{

public int Health { get; set; }

public int Damage { get; set; }

public void TakeDamage(int amount)

{

// Xử lý nhận sát thương

}

}

public class Enemy : MonoBehaviour

{

public float MoveSpeed { get; set; }

public int Damage { get; set; }

public void Attack(Player player)

{

// Thực hiện tấn công người chơi

}

}

}

### 3.Mechanic

using Model;

namespace Mechanic

{

public class MovementController : MonoBehaviour

{

private Player \_player;

private void Start()

{

\_player = GetComponent<Player>();

}

public void MovePlayer(Vector3 direction)

{

// Di chuyển người chơi

}

}

public class CombatSystem : MonoBehaviour

{

private Player \_player;

private Enemy \_enemy;

private void Start()

{

\_player = GetComponent<Player>();

\_enemy = FindObjectOfType<Enemy>();

}

public void Attack()

{

// Thực hiện tấn công

\_enemy.Attack(\_player);

}

}

}

### 4.gameplay

using Mechanic;

namespace GamePlayer

{

public class PlayerController : MonoBehaviour

{

private MovementController \_movementController;

private CombatSystem \_combatSystem;

private void Start()

{

\_movementController = GetComponent<MovementController>();

\_combatSystem = GetComponent<CombatSystem>();

}

private void Update()

{

// Xử lý nhập liệu và điều khiển người chơi

\_movementController.MovePlayer(GetInputDirection());

if (Input.GetButtonDown("Fire1"))

{

\_combatSystem.Attack();

}

}

private Vector3 GetInputDirection()

{

// Lấy hướng di chuyển từ input

return new Vector3(Input.GetAxis("Horizontal"), 0f, Input.GetAxis("Vertical"));

}

}

}

### 5.UI

using Model;

using UnityEngine;

using UnityEngine.UI;

namespace UI

{

public class HUDController : MonoBehaviour

{

public Text healthText;

private Player \_player;

private void Start()

{

\_player = FindObjectOfType<Player>();

}

private void Update()

{

// Cập nhật thông tin HP cho người chơi

healthText.text = "Health: " + \_player.Health;

}

}

}

### **6.view**

using Model;

using UnityEngine;

namespace View

{

public class PlayerView : MonoBehaviour

{

private Player \_player;

private void Start()

{

\_player = FindObjectOfType<Player>();

UpdatePlayerView();

}

private void UpdatePlayerView()

{

// Cập nhật hiển thị cho nhân vật người chơi

transform.position = \_player.transform.position;

GetComponent<Renderer>().material.color = Color.blue;

}

}

}

# **2. khi nào cần update, fix update và late update**

## **Update():**

Được gọi một lần mỗi frame, ngay sau khi frame được vẽ.

Thích hợp để xử lý các tương tác người dùng, cập nhật trạng thái game, hoặc các tác vụ không phụ thuộc vào thời gian.

Ví dụ: Kiểm tra đầu vào, cập nhật vị trí của vật thể dựa trên di chuyển của người dùng.

## **FixedUpdate():**

Được gọi với một tần suất cố định, độc lập với tốc độ khung hình.

Thích hợp để cập nhật các vật lý, như chuyển động, va chạm, lực, v.v.

Ví dụ: Cập nhật vị trí và vận tốc của vật thể dựa trên lực và khối lượng.

## **LateUpdate():**

Được gọi sau khi tất cả các Update() trong frame đã được gọi.

Thích hợp để cập nhật các thành phần phụ thuộc vào trạng thái của các vật thể khác, như camera theo dõi nhân vật.

Ví dụ: Cập nhật vị trí của camera để theo dõi nhân vật sau khi nhân vật đã được cập nhật vị trí trong Update().

## **Một số trường hợp điển hình khi nên sử dụng FixedUpdate() và LateUpdate():**

FixedUpdate():

Cập nhật vật lý, như di chuyển, va chạm, lực.

Cập nhật trạng thái của các thành phần liên quan đến vật lý, như Rigidbody.

LateUpdate():

Cập nhật camera để theo dõi nhân vật hoặc đối tượng khác.

Cập nhật UI để phản ảnh trạng thái của game.

Cập nhật các thành phần phụ thuộc vào trạng thái của các thành phần khác.

# **3.XML doc cơ bản**

## **<summary>:**

<summary> là một phần của XML-doc comment, được sử dụng để mô tả một phần của code, như một phương thức, lớp hoặc thuộc tính.

Mục đích của <summary> là cung cấp thông tin mô tả về chức năng và mục đích của code, giúp người đọc hiểu rõ hơn.

Thông tin trong <summary> sẽ được hiển thị khi sử dụng IDE như Visual Studio để xem tài liệu của code.

<summary> được bao quanh bởi cặp thẻ /// hoặc /\*\* \*/.

/// <summary>

/// Turns the main menu on or off.

/// </summary>

## **<param>:**

Thẻ <param> được sử dụng để mô tả một tham số đầu vào của một phương thức.

Cú pháp: <param name="tên\_tham\_số">Mô tả tham số</param>

## **<return>:**

Thẻ <return> được sử dụng để mô tả giá trị trả về của một phương thức.

Cú pháp: <returns>Mô tả giá trị trả về</returns>

Ví dụ:

/// <summary>

/// Calculates the area of a rectangle.

/// </summary>

/// <param name="width">The width of the rectangle.</param>

/// <param name="height">The height of the rectangle.</param>

/// <returns>The area of the rectangle.</returns>

public double CalculateRectangleArea(double width, double height)

{

return width \* height;

}

# **4.Spawn quái**

Đầu tiên, tạo một Scriptable Object để lưu trữ thông tin vị trí của enemy:

using UnityEngine;

[CreateAssetMenu(fileName = "EnemySpawnData", menuName = "Scriptable Objects/Enemy Spawn Data")]

public class EnemySpawnData : ScriptableObject

{

public Transform[] spawnPositions;

}

Tạo một prefab của enemy và một script để quản lý việc spawn enemy:

using UnityEngine;

public class EnemySpawner : MonoBehaviour

{

public GameObject enemyPrefab;

public EnemySpawnData spawnData;

private void Start()

{

SpawnEnemies();

}

private void SpawnEnemies()

{

foreach (Transform position in spawnData.spawnPositions)

{

Instantiate(enemyPrefab, position.position, Quaternion.identity);

}

}

}

# **5.Parralax layer**

movementScale: Một Vector3 xác định tỉ lệ di chuyển của lớp này so với camera. Nếu movementScale là Vector3.one, lớp sẽ di chuyển cùng với camera. Nếu movementScale là Vector3(0.5f, 0.5f, 0.5f), lớp sẽ di chuyển với tốc độ bằng một nửa tốc độ di chuyển của camera.

\_camera: Một Transform tham chiếu đến camera chính.

Awake(): Một hàm được gọi khi component được tạo. Trong đây, nó gán camera chính vào biến \_camera.

LateUpdate(): Một hàm được gọi ở cuối mỗi khung hình. Trong đây, nó cập nhật vị trí của Transform của component sao cho tương ứng với vị trí camera nhân với movementScale.

using UnityEngine;

namespace Platformer.View

{

/// <summary>

/// Used to move a transform relative to the main camera position with a scale factor applied.

/// This is used to implement parallax scrolling effects on different branches of gameobjects.

/// </summary>

public class ParallaxLayer : MonoBehaviour

{

/// <summary>

/// Movement of the layer is scaled by this value.

/// </summary>

public Vector3 movementScale = Vector3.one;

Transform \_camera;

void Awake()

{

\_camera = Camera.main.transform;

}

void LateUpdate()

{

transform.position = Vector3.Scale(\_camera.position, movementScale);

}

}

}

# **6.Telepost**

using UnityEngine;

public class TeleportManager : MonoBehaviour

{

public Transform player;

private const string TELEPORT\_POSITION\_X = "TeleportPositionX";

private const string TELEPORT\_POSITION\_Y = "TeleportPositionY";

private const string TELEPORT\_POSITION\_Z = "TeleportPositionZ";

public void SaveTeleportPosition()

{

// Lưu vị trí hiện tại của player vào PlayerPrefs

PlayerPrefs.SetFloat(TELEPORT\_POSITION\_X, player.position.x);

PlayerPrefs.SetFloat(TELEPORT\_POSITION\_Y, player.position.y);

PlayerPrefs.SetFloat(TELEPORT\_POSITION\_Z, player.position.z);

}

public void TeleportToSavedPosition()

{

// Lấy vị trí teleport từ PlayerPrefs và teleport player đến đó

float x = PlayerPrefs.GetFloat(TELEPORT\_POSITION\_X, player.position.x);

float y = PlayerPrefs.GetFloat(TELEPORT\_POSITION\_Y, player.position.y);

float z = PlayerPrefs.GetFloat(TELEPORT\_POSITION\_Z, player.position.z);

player.position = new Vector3(x, y, z);

}

}