Bài 2: Tối ưu mã nguồn C# trong Unity

MỤC TIÊU

- Giới thiệu chung
- Ngôn ngữ trong Unity
- **Lớp components**
- **Truy xuất thuộc tính của script**
- **Các phương thức cơ bản**
- **Debugging**
- **Tối ưu hóa mã nguồn**

Nội dung

- Ngôn ngữ trong Unity
- C#
- Syntax của C# trong game
  - Điều khiển GameObject với C#
- **Lớp components**
- **Truy xuất thuộc tính của script**
- **Các phương thức cơ bản**
- **Debugging**
- **Tối ưu hóa mã nguồn**

Lớp Components

- Là lớp cơ bản nhất được tích hợp vào các GameObject.
- Lớp Component bao gồm các thuộc tính lưu trữ những thông tin cơ bản và quan trọng nhất đối với một GameObject như vị trí của đối tượng trong không gian 3D, độ xoay của đối tượng, tên, tag ...v...v

Thuộc tính	Chú thích
transform	Thông tin về vị trí, độ xoay
rigidbody	Giả lập khối lượng và trọng lượng cho GameObject.
camera	Truy xuất đến các thuộc tính của class Camera nếu GameObject được tích hợp một camer
light	Truy xuất đến các thuộc tính của class Light nếu GameObject được tích hợp một Light.
animation	Truy xuất đến các thuộc tính của class Animation nếu GameObject bao gồm các animation.
constantForce	Truy xuất đến các thuộc tính của class ConstantForce nếu GameObject được tích hợp ConstantForce.

Lớp Components

Thuộc tính	Chú thích
renderer	Truy xuất đến các thuộc tính của class Renderer nếu GameObject được tích hợp Renderer.
audio	Truy xuất đến các thuộc tính của class Audio nếu GameObject được tích hợp Audio.
guiText	Truy xuất đến các thuộc tính của class GUIText nếu GameObject được tích hợp GUIText
networkView	Truy xuất đến các thuộc tính của class NetworkView nếu GameObject được tích hợp NetworkView.
guiTexture	Truy xuất đến các thuộc tính của class GUITexture nếu GameObject được tích hợp GUITexture.
collider	Truy xuất đến các thuộc tính của class Collider nếu GameObject được tích hợp Collider (được dung đểkiểm tra các va chạm).
hingeJoint	Truy xuất đến các thuộc tính của class HingeJoint nếu GameObject được tích hợp HingeJoint.
particleEmitter	Truy xuất đến các thuộc tính của class ParticleEmitter nếu GameObject được tích hợp ParticleEmitter.
particleSystem	Truy xuất đến các thuộc tính của class ParticleSystem nếu GameObject được tích hợp ParticleSystem.

Lớp Components

Thuộc tính	Chú thích
gameObject	Truy xuất đến đối tượng mà nó tích hợp đến.
tag	Tag của component.

- **Chú ý:**

Mặc định khi khởi tạo một đối tượng, đối tượng đó sẽ mang tên "GameObject" và được tích hợp sẵn thành phần Transform.

Lớp Components

Phương thức	Chú thích
GetComponent	Trả về kiểu Component được tích hợp vào đối tượng.
GetComponentInChildren	Trả về kiểu Component được tích hợp vào đối tượng hoặc con của đối tượng.
GetComponentsInChildren	Giống GetComponentsInChildren nhưng trảvềnhiều
Component	GetComponents Giống GetComponent nhưng trả về nhiều Component
CompareTag	Đối tượng có thuộc tag nào không?
SendMessageUpwards	Gữi yêu cầu thực hiện một phương thức đến các thành phần được tích hợp trong cùng một đối tượng.
SendMessage	Gữi yêu cầu thực hiện một phương thức đến các thành phần được tích hợp trong cùng một đối tượng.
BroadcastMessage	Gữi yêu cầu thực hiện một phương thức đến các đối tượng con.

Truy xuất đến thuộc tính của script

- Bắt đầu bằng một ví dụ cơ bản, đoạn code sau sẽ di chuyển một đối tượng thông qua input và một biến hiệu chỉnh tốc độ.

```
using UnityEngine;
using System.Collections;
public class Move : MonoBehaviour
{
    public float speed = 5.0F;//tốc độ của đối tượng
void Start () {}
// Update is called once per frame
void Update ()
{
        float x = Input.GetAxis("Horizontal") * Time.deltaTime * speed; // tốc độ theo cấu trúc ngang
        float z = Input.GetAxis("Vertical") * Time.deltaTime * speed; // tốc độ theo cấu trúc ngang
        transform.Translate(x,0,z);// di chuyển theo vector (x,0,z)
}
}
```

Truy xuất đến thuộc tính của script

- **Chú ý:** Sau khi tích hợp script trên vào một đối tượng, thuộc tính speed của class Move được public ra ngoài để ta có thể hiệu chỉnh trực tiếp bằng cách click chuột và nhập thông số, bản thân các script nên được xây dựng như một tool để tiện lợi cho việc hiệu chỉnh và tránh can thiệp vào mã nguồn, ngoài ra còn đơn giản hóa việc sử dụng script.

Truy xuất đến thuộc tính của script

- Trong quá trình phát triển chúng ta sẽ có nhu cầu về việc hiệu chỉnh tốc độ thông qua code, sử dụng phương thức GetComponent để truy xuất đến thuộc tính của một thành phần bất kỳ được tích hợp trong đối tượng.

```
using UnityEngine;
using System.Collections;
public class SetSpeed : MonoBehaviour
{
// Use this for initialization
void Start () {}
   // tăng tốc độ mỗi khi player nhấn phím U
void Update ()
{
      if (Input.GetKey(KeyCode.U))
      {
         Move moveScript = gameObject.GetComponent<Move>;// truy xuất đến lớp Move
         moveScript.speed += 0.5F;// tăng tốc độ
      }
}
}
```

Truy xuất đến thuộc tính của script

- **Giải thích:** Sau khi tích hợp script SetSpeed vào cùng một đối tượng với script Move, thuộc tính gameObject sẽ truy xuất đến đối tượng chứa 2 script này và thông qua phương thức GetComponent để truy xuất đến một thành phần bất kỳ được tích hợp vào đối tượng, cụ thể ở đây là script Move.
- **Chú ý:** Tại script C# không thể truy xuất đến script Javascript thông qua phương thức GetComponent.

Các phương thức cơ bản

- Khi khởi tạo một script, mặc định nó đã chứa các phương thức Start(), Update(), đây là hai phương thức rất hữu dụng, ngoài ra còn một số phương thức khác được liệt kê dưới đây:
- **Update():** Những đoạn code thuộc phương thức này sẽ được gọi lại mỗi frame (khung hình).
- **FixedUpdate():** Giống với Update nhưng ta có thể hiệu chỉnh được số khung hình ban đầu, và không bị ảnh hưởng khi tần số frame không ổn định.
- **Awake():** Những đoạn code thuộc phương thức này được gọi khi script ở giai đoạn khởi tạo, thường được dùng để thiết lập hoặc tải dữ liệu ban đầu cho các Component.

Các phương thức cơ bản

- **Start():** Được gọi thực hiện trước phương thức Update() nhưng lại sau Awake(), Khác nhau cơ bản giữa Start() và Awake() là phương thức Start() chỉ được gọi khi script không bị đình chỉ hoạt động (enabled).
- **OnCollisionEnter():** Những đoạn code thuộc phương thức này sẽ được thực thi khi đối tượng chứa nó bao gồm một Collider và va chạm với một Collider hoặc Rigidbody của một đối tượng khác.
- **OnMouseDown():** Những đoạn code thuộc phương thức này sẽ được thực thi khi chuột của người chơi click vào một đối tượng có tích hợp thành phần GUIElement hoặc Collider.

Debugging

- Unity cung cấp lớp Debug để hỗ trợ lập trình viên theo dõi và kiểm soát các lỗi, ở đây chúng ta quan tâm đến phương thức Debug.Log().
- Phương thức Log() cho phép người dùng gửi một thông tin đến Unity Console nhằm mục đích:
  - Chứng minh rằng đoạn mã này đang được thực hiện.
  - Báo cáo tình trạng hiện tại của biến.
- Trở lại ví dụ về lớp SetSpeed, kiểm tra tốc độ của đối tượng mỗi khi người chơi tăng tốc.

Debugging

- Trở lại ví dụ về lớp SetSpeed, kiểm tra tốc độ của đối tượng mỗi khi người chơi tăng tốc.

```
using UnityEngine;
using System.Collections;
public class SetSpeed : MonoBehaviour
{
// Use this for initialization
void Start () {}
// Update is called once per frame
void Update ()
{
    if (Input.GetKey(KeyCode.U))
    {
        Move moveScript = gameObject.GetComponent<Move>;// truy xuất đến lớp Move
        moveScript.speed += 0.5F;// tăng tốc độ
        Debug.Log(moveScript.speed);
    }
}
}
```

Tối ưu hóa mã nguồn:

Sử dụng các thành phần của GameObject thông qua biến tĩnh:

- Như đã tìm hiểu ở các mục trên, bản thân một GameObject luôn được tích hợp sẵn các thành phần để lưu trữ thông tin của chúng trong không gian 3D, ở đây chúng ta sẽ lấy ví dụ cụ thể với thành phần transform.

```
using UnityEngine;

using System.Collections;

public class Example : MonoBehaviour

{

void Update ()

        {

                transform.Translate(0,0,0);

        }

}
```

Tối ưu hóa mã nguồn:

Sử dụng các thành phần của GameObject thông qua biến tĩnh:

- Đoạn mã trên được sửa lại như sau:

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
        private Transform myTransform;
         void Awake()
        {
                myTransform = transform;
        }
        void Update ()
        {
                myTransform.Translate(0,0,0);
        }
}
```

Tối ưu hóa mã nguồn:

Sử dụng các thành phần của GameObject thông qua biến tĩnh:

- Mỗi khi chúng ta gọi một thành phần bất kỳ của GameObject, hệ thống Unity sẽ tốn thời gian để duyệt qua các thành phần được tích hợp trong GameObject để cuối cùng trả về kết quả thích hợp.
- Để tiết kiệm khoản thời gian đó nên sử dụng thành phần của GameObject thông qua biến tĩnh.

Tối ưu hóa mã nguồn:

Sử dụng mảng tĩnh

- Các lớp ArrayList hay Array thì rất dễ sử dụng, chúng ta có thể dễ dàng thêm một phần tử vào mảng và sử dụng các phương thức, nhưng chi phí phải trả để hệ thống thực hiện điều đó là rất cao.
- Thay vì vậy, việc sử dụng các mảng được cấu trúc sẵn sẽ giảm bớt công việc cho hệ thống, vì các phần tử trong mảng có cùng kiểu và độ dài của mảng đã được xác định từ trước, chúng ta chỉ mất thời gian để xác định độ dài mảng cần thiết, bù lại chúng ta tiết kiệm được một khoản chi phí khá lớn.

Tối ưu hóa mã nguồn:

Sử dụng mảng tĩnh

- Chú ý: Hạn chế sử dụng for, foreach, nên sử dụng while để đạt được tốc độ tối ưu.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
   private Vector3[] positions;
   void Awake()
   {
      positions= new Vector3[100];
      int i = 0;
      while (i < 100)
           {
                positions[i] = Vector3.zero;
                i++;
           }
   }
}
```

Tối ưu hóa mã nguồn:

Hiệu chỉnh tần số sử dụng phương thức

- **Đặt vấn đề:** Khi người chơi ở quá xa, kẻ địch sẽ ởtrạng thái ngủ tạm thời.
- **Giải thích:** Đoạn mã trên sẽ xét khoảng cách giữa người chơi và kẻ địch tại mỗi frame, nhưng điều đó là không cần thiết, trung bình một giây hệ thống chạy từ 30 đến 40 khung hình, chi phí bỏ ra quá lớn nhưng hiệu quả lại không cao.

```
using UnityEngine;

using System.Collections;

public class Example : MonoBehaviour

{

   public Transform target;

void Update ()

  {

      if (Vector3.Distance(transform.position, target.position) > 100)

      return;

  }

}
```

Tối ưu hóa mã nguồn:

Yield và Coroutine

- Các Coroutine cho phép chúng ta đình trệ hay làm trễ việc thực thi một đoạn mã hoặc một phương thức, áp dụng để xử lý vấn đề nêu trên.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    public Transform target;
    void Start()
    {
        StartCoroutine("TestDistance");
    }
    IEnumerable TestDistance()
    {
        while (true)
        {
            if (Vector3.Distance(transform.position, target.position) > 100)
            {
                Debug.Log("Khong lam gi");
            }
            yield return new WaitForSeconds(2);
        }
    }
}
```

Tối ưu hóa mã nguồn:

Yield và Coroutine

- **Giải thích:** Việc thực thi vòng lập while sẽ bị chậm lại 2 giây sau mỗi lần lặp do tác động của yield, việc xét khoảng cách giữa người chơi và kẻ địch sẽ được thực thi 2 giây 1 lần, tiết kiệm được nhiều chi phí và giảm nhẹ công việc cho hệ thống.


THANKS FOR
WATCHING!