Unity and C# Design patterns and concepts.

Unity là một công cụ phát triển game rất dễ bắt đầu, giao diện đơn giản và được cung cấp API giúp bạn xây dựng trò chơi nhanh chóng hơn so với các công cụ khác, điều này làm cho Unity trở nên phổ biến và được sử dụng để tạo ra nhiều trò chơi.

Tuy nhiên, dễ dàng sử dụng Unity chỉ đúng khi bạn có kiến thức cơ bản về mẫu lập trình và các khái niệm liên quan. Nếu không, việc gỡ lỗi ứng dụng của bạn có thể trở thành ác mộng. Để tránh điều đó, tôi xin giới thiệu cho bạn một hướng dẫn nhỏ, hy vọng nó sẽ giúp ích cho dự án của bạn trong tương lai.

Singleton:

Singleton là một mẫu thiết kế gây tranh cãi, bạn có thể đã nghe hoặc sử dụng nó nhiều lần trước đây. Mẫu thiết kế này giới hạn việc tạo các phiên bản của một lớp chỉ thành một đối tượng duy nhất. Singleton thường được sử dụng để phối hợp và quản lý các chức năng và chỉ cần một đối tượng duy nhất.

Nếu bạn tự hỏi lợi ích của việc sử dụng mẫu Singleton so với các lớp tĩnh, thì lớp tĩnh được tải lười biếng khi được tham chiếu lần đầu tiên và phải có hàm khởi tạo tĩnh trống, điều này có thể dẫn đến việc gây lỗi trong mã nếu không cẩn thận. Đối với mẫu Singleton, nó thực hiện nhiều công việc và phương thức khởi tạo tĩnh làm cho chúng không thể thay đổi.

Hơn nữa, Singleton có thể triển khai một giao diện, trong khi lớp tĩnh không thể, cho phép bạn có một đối tượng trò chơi với các thành phần để tổ chức tốt hơn. Bạn cũng có thể kế thừa Singleton từ các lớp cơ sở.

Tuy nhiên, bạn cần cẩn thận khi sử dụng Singleton, như tên gọi của nó, các đối tượng Singleton hoạt động độc lập và không nên phụ thuộc vào các lớp khác. Nếu bạn sử dụng Singleton, hãy đảm bảo nó có thể hoạt động một mình. Dưới đây là một cách triển khai Singleton:

```
public class MonoSingleton<T> : MonoBehaviour where T : MonoBehaviour {
    protected static T instance;
```

```
static public bool isActive {
    get {
        return instance != null;
    }
}

public static T Instance
{
    get {

        if (applicationIsQuitting)
        {
            return null;
        }

        if(instance == null)
        {
            instance = (T) FindObjectOfType(typeof(T));

            if (instance == null)
            {
                GameObject go = new GameObject("Singleton-" + typeof(T).Name);
                DontDestroyOnLoad(go);
                instance = go.AddComponent<T>();
            }
        }
        return instance;
    }
}

void OnEnable()
{
    if (FindObjectsOfType(typeof(T)).Length > 1)
    {
        for (int i = 0; i < FindObjectsOfType(typeof(T)).Length; i++)
        {
            if (FindObjectsOfType(typeof(T))[i] != this.gameObject.GetComponent<T>())
            {
                Destroy(((T)(FindObjectsOfType(typeof(T))[i])).gameObject);
            }
        }
    }
}

private static bool applicationIsQuitting = false;

void OnApplicationQuit()
{
```

```
        applicationIsQuitting = true;
        instance = null;
    }
}
```
Gọi singleton:

Tạo lớp singleton của bạn và kế thừa từ lớp trên, sau đó gọi nó bằng cách sử dụng Instance:

public class MyClass : MonoSingleton<MyClass>
Sau đó, bạn có thể gọi singleton từ bất kỳ script nào trong scene của bạn, ví dụ: một hàm StartGame.

MyClass.Instance.StartGame();
Unity sẽ cố gắng tìm đối tượng trong scene của bạn, và nếu không tìm thấy, một phiên bản sẽ được tạo ra và nó sẽ không bị hủy bỏ cho đến khi bạn thoát ứng dụng. Rất tiếc là không có cách tốt để loại bỏ từ khóa "Instance" ngay tại đó.

Pooling:

Có một số trường hợp khi bạn phải tạo ra một đối tượng nhiều lần, ví dụ như viên đạn bắn từ súng, điều này có thể làm tải ứng dụng của bạn trở nên rất nặng, mỗi lần tạo mới giảm FPS hơn nữa.

Để tránh việc tạo nhiều phiên bản, thường sử dụng kỹ thuật pooling. Pooling là một kỹ thuật quản lý tài nguyên, cho phép tái sử dụng các đối tượng, điều này quan trọng khi bạn tạo và hủy nhiều đối tượng.

Cơ bản, bạn tạo ra một số đối tượng khi scene bắt đầu vào một "bể" (pool) và ẩn chúng khỏi tầm nhìn trực tiếp. Sau đó, khi cần tạo đối tượng, đối tượng sẽ được lấy từ bể và sử dụng thay vì tạo mới. Điều này giảm thiểu nguy cơ làm chậm ứng dụng bằng cách tạo nhiều phiên bản mới. Sau khi đối tượng hoàn thành công việc của nó, nó sẽ được trả lại bể, chờ cuộc gọi tiếp theo.

Dưới đây là một hệ thống Pool rất tốt mà bạn có thể sử dụng cho ứng dụng của mình, tất cả các công đoạn thuộc quyền sở hữu của Sumit Das tại SwiftFinger Games.

https://github.com/6ameDev/Advanced-Pooling-System

Delegates:

Delegate, như tên gọi, là một kiểu biến tham chiếu, nó giữ tham chiếu của các phương thức có cùng chữ ký với kiểu được khai báo. Điều đẹp đẽ về delegate là nó có thể thay đổi trong thời gian chạy, khi được gọi, nó thông báo cho tất cả các phương thức tham chiếu đến nó.

Để giải thích rõ hơn, delegate giống như một danh sách các hàm, và khi bạn gọi nó, nó sẽ gọi các hàm được tham chiếu đến nó.

Hãy bắt đầu bằng cách tạo một lớp có delegate và một sự kiện từ nó:

```
public class EventManager : MonoBehaviour
{
   public delegate void ClickAction(GameObject g);
   public static event ClickAction OnClicked;
}
```

Delegate gọi là ClickAction truyền một GameObject cho phép chúng ta xác định đối tượng trò chơi mà nó đến từ. Sự kiện cho phép bạn giao tiếp từ một đối tượng tới các lớp khác thông qua delegate. Bằng cách này, bạn có thể đăng ký các phương thức xử lý sự kiện và khi sự kiện xảy ra, tất cả các phương thức được đăng ký sẽ được gọi.

Để đăng ký một phương thức xử lý sự kiện, bạn chỉ cần gọi phương thức AddListener trên sự kiện tương ứng:

```
public class ClickHandler : MonoBehaviour
{
   private void OnEnable()
   {
      EventManager.OnClicked += HandleClick;
   }

   private void OnDisable()
   {
      EventManager.OnClicked -= HandleClick;
   }

   private void HandleClick(GameObject g)
   {
      // Xử lý sự kiện khi bị nhấp chuột
   }
}
```

Trong ví dụ trên, phương thức HandleClick sẽ được gọi khi sự kiện OnClicked xảy ra. Chúng ta đăng ký phương thức này bằng cách sử dụng toán tử += và gỡ đăng ký bằng cách sử dụng toán tử -=.

Khi bạn muốn kích hoạt sự kiện, bạn chỉ cần gọi nó và truyền các tham số tương ứng:

```
public class ClickTrigger : MonoBehaviour
{
   private void Update()
   {
      if (Input.GetMouseButtonDown(0))
      {
         GameObject clickedObject = GetClickedObject();
         EventManager.OnClicked?.Invoke(clickedObject);
      }
```

```
    }

    private GameObject GetClickedObject()
    {
        // Lấy đối tượng trong game được nhấp chuột
    }
}
```

Trong đoạn mã trên, chúng ta kiểm tra xem nút chuột trái có được nhấp xuống không. Nếu có, chúng ta lấy đối tượng trong game được nhấp chuột và gọi sự kiện OnClicked, truyền đối tượng đó làm tham số.

Delegate và sự kiện điều khiển linh hoạt và mạnh mẽ trong việc giao tiếp giữa các phần của ứng dụng Unity. Bạn có thể sử dụng chúng để xử lý sự kiện như nhấp chuột, va chạm, hoặc bất kỳ hành động nào khác mà bạn muốn đồng bộ hoặc truyền dữ liệu giữa các đối tượng trong game.

Hy vọng những thông tin trên có thể giúp bạn hiểu rõ hơn về cách sử dụng Singleton, Pooling và Delegates trong Unity để phát triển trò chơi của mình. Nếu bạn có bất kỳ câu hỏi hoặc cần thêm thông tin, hãy cho tôi biết!