Load addressable assets bằng scripts

Tóm tắt

Trong bài học này chúng ta sẽ tìm hiểu một vài cách để load addressable asset vào game và làm thế nào để giải phóng chúng khỏi bộ nhớ bằng scripts. Chúng tôi đã đính lèm một số tài liệu tham khảo và các phần ôn tập về những khái niệm lập trình mà bạn sẽ sử dụng với hệ thống Addressables.

Kết thúc bài này chúng ta sẽ:

- Định nghĩa một thao tác bất đồng bộ (asynchronous operation).
- Xác định các cách khác nhau để tham chiếu đến một addressable trong code
- Xác định các phương pháp khác nhau để load và release các addressable một cách bất đồng bộ.

Tổng quan

Trong hướng dẫn này, bạn sẽ học cách load và unload addressable vào game thông qua code. Trong suốt bài thực hành này, bạn sẽ tiến hành loại bỏ phần lớn source code sử dụng Resource API trong dự án Loady Dungeons và thay bằng Addressables API.

Trước khi bắt đầu

Trước khi bắt đầu chúng ta hãy cùng ôn tập lại những khái niệm quan trọng trong lập trình và đặc biệt cần thiết với Addressables.

L?p trình b?t ??ng b?

Trong Addressables API, nhiều tác vụ thực hiện việc load asset và dữ liệu sau đó trả về kết quả. Khi các tài nguyên hoặc dữ liệu đó nằm trên một máy chủ, các tác vụ này sẽ cần thêm thời gian để hoàn thành. Để tránh làm giảm hiệu suất và gây trì hoãn các tác vụ khác. Hệ thống Addressables sử dụng các thao tác bất đồng bộ, điều này có nghĩa là nhiều tác vụ có thể chạy đồng thời.

Để triển khai các thao tác bất đồng bộ với Addressables API bạn sẽ cần hiểu rõ những khái niệm sau (những khái niệm này nằm trong project Intermediate scripting):

- **Coroutines** cho phép bạn thực thi một hàm trong nhiều frame thay vì chờ nó hoàn thành ngay trong một frame.
- **Delegates** là các container đại diện cho các hàm có thể truyền đi dưới dạng tham số hoặc sử dụng như các biến thông thường.
- **Events** là một dạng delegate chuyên biệt, có khả năng phát tín hiệu thông báo cho class khác biết rằng có một sự kiện

Structs

Addressables API sử dụng struct có tên là AsyncOperationHandle để giữ tác vụ bất đồng bộ bên trong code của bạn.

Struct là một kiểu tham trị trong C#, có thể được sử dụng để lưu các dữ liệu liên quan với nhau thành một nhóm. Chúng giống với class ở chỗ có thể chứa các filed và các method nhưng khác với class chúng được truyền theo giá trị thay vì truyền theo tham chiếu. Có nghĩa là khi bạn tạo một struct và truyền vào một method hoặc gán nó cho một biến, một bản sao của struct sẽ được tạo ra thay vì tạo ra tham chiếu trỏ tới đối tượng gốc.

Struct AsyncOperationHandle có một giá trị bool để xác định xem method thành công hay thất bại và trả về giá trị sau quá trình load

Tu?n t? hóa (Serialization)

Tuần tự hóa là quá trình chuyển đổi và lưu trữ dữ liệu giữa các phiên làm việc (sesstion) của một ứng dụng. Giải tuần tự hóa (Deserialization) là quá trình trích xuất dữ liệu đã lưu trữ đó để nó có thể được tái cấu trúc lại khi ứng dụng chạy ở những lần sau.

Trong hướng dẫn này, bạn không cần trở thành một chuyên gia tuần tự hóa. Bạn chỉ cần biết rằng Unity thực hiện tuần tự hóa dữ liệu trong quá trình build để nó có thể giải tuần tự hóa dữ liệu được lưu trữ đó khi ứng dụng chạy. Thuộc tính [SerializeField] có tác dụng đánh dấu các trường dữ liệu private là được tuần tự hóa và các trường đã được tuần tự hóa sẽ có thể hiển thị trong Inspector để bạn có thể chỉnh sửa như các biến public.

Khi bạn sử dụng Addressables, bạn sẽ phải sử dụng quá trình tuần tự hóa để tối ưu hóa cách dự án của bạn tiêu thụ tài asset.

Bạn có thể tìm hiểu thêm về tuần tự hóa trong Unity Manual.

Load một addressable prefab

Dưới đây là một vài cách để load một addressable prefabs với Addressables API. Hãy bắt đầu với việc load asset bằng cách sử dụng địa chỉ của nó, điều mà bạn sẽ thực hiện trong quá trình chuyển đổi script `PlayerConfigurator` sang sử dụng Addressable API.

Để viết lại `PlayerConfigurator.cs` với Addressables API thay vì Resources API, bạn có thể làm theo hướng dẫn dưới đây:

1. Từ cửa sổ Project, mở file `Assets > Scripts > PlayerConfigurator.cs`
2. Khai báo các namespace của Addressable cần sử dụng

   ```
   using UnityEngine.AddressableAssets;
   using UnityEngine.ResourceManagement.AsyncOperations;
   ```

   Bạn sẽ cần sử dụng namespace này để load asset thông qua địa chỉ của nó và truy cập vào `AsyncOperationHandle` được trả về.
3. Trong class PlayerConfigurator tạp một biến string mới đại diện cho địa chỉ. Đặt attribute [SerializeField] ở trước đó để bạn có thể serialize nó và chỉnh sửa giá trị ngay trong Inspector.

   ```
   [SerializeField] private string m_Address;
   ```

   Sau khi khai báo bạn có thể thấy biến này xuất hiện trong của sổ Inspector với tên là Address
4. Trong script hãy tìm dòng code dưới đây. Nó đang sử dụng Resources API

   ```
   private ResourceRequest m_HatLoadingRequest;
   ```

   Và thay nó bằng dòng code sử dụng Addressables API dưới đây:

   ```
   private AsyncOperationHandle<GameObject> m_HatLoadOpHandle;
   ```

   Dòng mới này khai váo biến `m_HatLoadOpHandle` là một `AsyncOperationHandle` của GameObject.
5. Trong hàm `SetHat()` hãy xóa toàn bộ phần thân của nó đi và thay bằng dòng code dưới đây

   ```
   m_HatLoadOpHandle = Addressables.LoadAssetAsync<GameObject>(m_Address);
   ```

   Dòng code này sẽ load bất đồng bộ prefab thông qua địa chỉ `m_Address`. Biến `m_HatLoadOpHandle` sẽ chứa một giá trị bool cho biết asset có được tải thành công hay không. Nếu load thành công, handle này cũng chứa kết quả của request tức là chính prefab chúng ta đang load.
6. Bởi vì asset được load bất đồng bộ, nên các dòng code phía sau nó sẽ được chạy mà không cần chờ `LoadAssetAsync` chạy xong. Vậy nên để biết được việc load có thành công hay không chúng ta phải đăng ký một event để biết khi nào load xong.

```
m_HatLoadOpHandle.Completed += OnHatLoadComplete;
```

Dòng code này dùng để đăng ký một event handler (chúng ta sẽ khai báo nó ở bước sau). Vì chúng ta cần biết khi nào asset được load xong nên ở đây chúng ta sử dụng event `Completed` của `AsyncOperationHandle`

7. Ở đây chúng ta sẽ viết hàm `OnHatLoadComplete()` thay cho hàm `OnHatLoaded()` để in ra trạng thái của handle. Hàm này được gọi bởi Addressables khi nó thực hiện xong yêu cầu tải prefab ở trên.

```
private void OnHatLoadComplete(AsyncOperationHandle<GameObject> asyncOperationHandle)
{
    Debug.Log($"AsyncOperationHandle Status: {asyncOperationHandle.Status}");
}
```

8. Trong hàm OnDisable(), xóa bỏ toàn bộ phần body và thay thế bằng dòng code dưới đây để hủy đăng ký sự kiện khi gameObject bị disable. Nó sẽ chặn biệc Addressables gọi `OnHatLoadComplete()` khi `PlayerConfigurator` không hoạt động.

```
m_HatLoadOpHandle.Completed -= OnHatLoadComplete;
```

9. Save script và quay lại Unity.
10. Để test script này, mở scene Assets > Scenes > Level_00.
11. Trong cửa sổ Project, chọn prefab Assets > Prefabs > Player.
12. Nhìn sang cửa sổ Inspector, tìm component `PlayerConfigurator` và đặt giá trị cửa biến Address thành địa chỉ của chiếc mũ. Ở phần trên chúng ta đã đặt là Hat_BunnyEars.
13. Play game và nhìn vào cửa sổ Console, bạn sẽ thấy dòng thông báo "*AsyncOperationHandle Status: Succeeded*".
    Tại thời điểm này, asset được được load thành công. Tuy nhiên, bạn sẽ chưa nhìn thấy nó trên scene vì bạn chưa instantiate nó.
14. Giờ là lúc bạn instantiate prefab mà bạn vừa load lên. Viết lại hàm `OnHatLoadComplete()` như dưới đây:

```
private void OnHatLoadComplete(AsyncOperationHandle<GameObject> asyncOperationHandle)
{
    if (asyncOperationHandle.Status == AsyncOperationStatus.Succeeded)
    {
        Instantiate(asyncOperationHandle.Result, m_HatAnchor);
    }
}
```

15. Vào lại Play mode và bạn sẽ thấy player di chuyển cùng cái mũ.
16. Bạn có thể thử với những địa chỉ khác mà chúng ta đã tạo ở bài hướng dẫn trước.

> Lưu ý: Trong ví dụ này Address là một string và nó sẽ không xác thực xem bạn có nhập đúng địa chỉ hay không. Bạn chỉ biết khi bạn thực sự chạy load. Bạn có thể nhập bất cứ thứ gì và điều đó có thể gây ra lỗi khi chạy game. Hãy đảm bảo rằng bạn nhập chính xác địa chỉ của prefab muốn load.

Load một addressable prefab với AssetReference

Asset Reference là một kiểu dữ liệu dùng để tham chiếu đến addressable asset. Nó được thiết kế để sử dụng như một trường dữ liệu có thể serializable trong MonoBehaviour hay ScriptableObject. Khi bạn thêm một AssetReference vào trong các class này, bạn có thể gán một địa chỉ cho nó trong cửa sổ Inspector thông qua một công cụ chọn. Những lựa chọn này được giới hạn nghiêm ngặt, chỉ cho phép chọn các asset đã được đánh dấu là addressable.

Về mặt hình thức, bạn sẽ khai báo các AssetReference vào script giống hệt như khi bạn sử dụng tham chiếu trực tiếp, tức là có thể thông qua các trường public hoặc các trường private có sử dụng attribute [SerializableField]. Tuy nhiên, AssetReference không lưu trữ trực tiếp một tham chiếu đến asset. Thay vào đó, AssetReference lưu trữ mã định danh GUID của asset đó. GUID này được hệ thống Addressables sử dụng để lưu trữ và truy xuất đối tượng trong runtime.

Lợi ích cốt lõi của việc sử dụng AssetReference so với việc dùng địa chỉ dạng string là nó bắt buộc bạn chỉ được chọn asset thông qua Inspector. Điều này giúp tránh các rủi ro như không đánh có như sai chính tả...

Tiếp theo chúng ta sẽ viết lại `PlayerConfigurator` sử dụng AssetReference thay cho địa chỉ dạng string.

1. Vẫn là mở file `Assets > Scripts > PlayerConfigurator.cs`
2. Thay `m_Address` bằng dòng dưới đây:

   ```
   [SerializeField] private AssetReference m_HatAssetReference;
   ```
3. Trong hàm `SetHat()` chúng ta thay phần body dòng code sau để load asset:

   ```
   if (!m_HatAssetReference.RuntimeKeyIsValid()) return;
   m_HatLoadOpHandle = m_HatAssetReference.LoadAssetAsync<GameObject>();
   m_HatLoadOpHandle.Completed += OnHatLoadComplete;
   ```

   `RuntimeKeyIsValid` thực hiện kiểm tra xem các giá trị được chọn có phải là một địa chỉ hợp lệ không. Trong trường hợp này nó sẽ chặn `LoadAssetAsync` nếu `AssetReference` bị bỏ trống (None);
4. Save script và quay trở lại với Level_00;
5. Trong cửa sổ Project, chọn prefab Assets > Prefabs > Player.
6. Trong cửa sổ Inspector, tìm đến component PlayerConfigurator. Bây giờ trường Address đã được thay thế bằng trường Hat Asset Reference. Tại đây hãy chọn cái mũ mà bạn muốn.
7. Play game và xem nhân vật có di chuyển cùng cái mũ bạn chọn không.
8. Giờ bạn có thể thử chọn các AssetReference khác và quan sát kết quả.

Bạn cũng có thể đọc thêm về AssetReference tại đây.

Load một addressable prefab với AssetReferenceGameObject

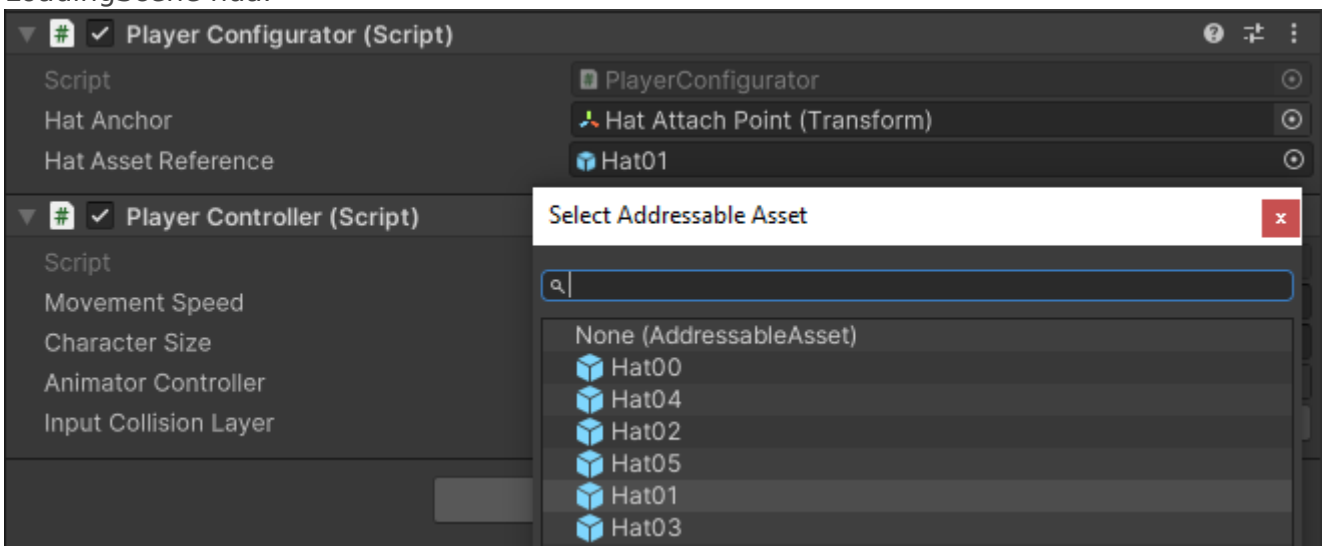
Công cụ chọn của AssetReference cho phép chọn tất cả các asset được đánh dấu là addressable nên bạn có thể thấy nó cho phép bạn chọn cả texture, scene... Mặc dù được phép chọn nhưng nếu bạn cố load chúng như những GameObject nó sẽ gây ra lỗi trong code của Addressables.

Bạn có thể muốn quản lý chặt chẽ hơn nữa và chỉ cho phép chọn các prefab từ công cụ chọn. Để làm được điều này bạn có thể sử dụng `AssetReferenceGameObject` thay vì `AssetReference`.

Tiếp tục viết lại `PlayerConfigurator` nhé.

1. Mở file `Assets > Scripts > PlayerConfigurator.cs`
2. Thay `m_HatAssetReference` bằng dòng dưới đây:

   ```
   [SerializeField] private AssetReferenceGameObject m_HatAssetReference;
   ```
3. Save script và quay trở lại Unity
4. Trong cửa sổ Project, chọn prefab Assets > Prefabs > Player.
5. Trong cửa sổ Inspector, tìm đến component PlayerConfigurator. Mở picker lên và chọn một cái mũ bạn muốn. Chú ý xem hiện giờ bạn không thể chọn LoadyDungeonsLogo hay LoadingScene nữa.



6. Play game và thử cái mũ mới.

Bạn cũng có thể đọc thêm về AssetReferenceGameObject tại đây.

Revision #1
Created 2026-04-18 15:03:00 UTC by ThanhDV
Updated 2026-04-19 16:19:47 UTC by ThanhDV