Unity

Unity Camera

orthographicSize & aspect

https://docs.unity3d.com/ScriptReference/Camera-orthographicSize.html

https://docs.unity3d.com/ScriptReference/Camera-aspect.html

public float **orthographicSize**;
public float **aspect**;

Mô tả

Camera.orthographicSize

Là **1/2** kích thước của camera khi ở chế độ **orthographic**.

Thuộc tính `orthographicSize` xác định độ lớn vùng nhìn thấy được của một orthographic Camera. Để chỉnh sửa `orthographicSize` bạn phải set Camera projection thành orthorgraphic.

Chiều cao của vùng nhìn là (`orthographicSize` * 2). Unity tính chiều rộng của vùng nhìn bằng cách sử dụng orthorgraphicSize và tỷ lệ của Camera.

Unity bỏ qua orthographicSize nếu camera không phải orthorgraphic. Thay vào đó hãy sử dụng **fieldOfView**.

Camera.aspect

Là tỷ lệ khung hình (chiều rộng/chiều cao).

Mặc định tỷ lệ này tự động được tính dựa theo tỷ lệ màn hình. Ngay cả khi camera không render full màn hình. Nếu bạn thay đổi aspect của camera, giá trị này sẽ giữ nguyên cho đến khi bạn gọi `camera.ResetAspect()`. Camera.aspect sẽ được đặt lại thành tỷ lệ khung hình của màn hình.

Giữ `orthographicSize` theo chiều rộng màn hình.

```
private Camera mainCamera;
private Vector2 referenceResolution = new Vector2(1080, 1920);

private void ResizeCamera()
{
    float screenWidth = Screen.width;
    float screenHeight = Screen.height;

    float baseHorizontalSize = mainCamera.orthographicSize * referenceResolution.x /
referenceResolution.y;
    float size = baseHorizontalSize * screenHeight / screenWidth;

    mainCamera.orthographicSize = size;
}
```

Unity Addressable Asset System

Unity Addressable Asset System | Package Manager UI website (unity3d.com)

Addressable Asset System cung cấp một cách đơn giản để load assets bằng “địa chỉ”. Nó xử lý các chi phí quản lý bằng cách đơn giản hóa việc tạo và triển khai các gói nội dung.
Addressable Asset System sử dụng tải bất đồng bộ để hỗ trợ tải ứng dụng từ bất kỳ đâu với mọi collection. Dù đang sử dụng tham chiếu trực tiếp, asset bundles, hoặc Resources folders.
Addressable Assets cung cấp một cách đơn giản và linh hoạt.

Summary

Asset là gì???

Asset là nội dung được sử dụng để tạo ra game hoặc ứng dụng. Asset có thể là prefab, texture, material, audio clip, animation...

Địa chỉ là gì???

Địa chỉ xác định vị trí nơi mà cái gì đó “cư trú”. Ví dụ, khi gọi điện thoại, số điện thoại sẽ được xem như một địa chỉ. Cho dù bạn ở nhà, chỗ làm, Hà Nội hay Sài Gòn thì mọi người đều có thể kết nối với bạn qua số điện thoại.

Addressable Asset là gì???

Khi một asset được đánh dấu là `addressable`, asset đó được gọi là addressable asset (nghe lú vl) và asset đó có thể được được gọi từ bất kỳ đâu. Dù cho addressable asset được đặt ở local hay trên network, hệ sẽ định vị và trả về asset đó. Bạn có thể load một addressable bằng địa chỉ của nó hoặc load nhiều addressable sử dụng một nhãn (label) của nhóm mà bạn khai báo.

Tại sao phải quan tâm???

Cách truyền thống gây ra nhiều khó khăn trong việc cấu trúc asset và load chúng một cách hiệu quả.
Sử dụng addressable assets rút ngắn thời gian cấu trúc cho phép bạn có nhiều thời gian hơn để thiết kế, lập trình, và kiểm thử ứng dụng. Với Addressable Assets bạn định danh assets bằng addressable và tải ló :))))

Addressable Asset System giải quyết vấn đề gì???

- **Thời gian lặp lại**: Tham chiếu đến nội dung bằng địa chỉ của nó là siêu hiệu quả. Với một địa chỉ được tham chiếu hệ thống chỉ việc truy xuất nó. Tối ưu hóa content mà không cần

thay đổi code.

- **Quản lý sự phụ thuộc**: Hệ thống không chỉ trả về nội dung theo địa chỉ mà còn trả về mọi sự phụ thuộc của nội dung đó. Hệ thống sẽ thông báo cho bạn khi nào nội dung sẵn sàng. Mọi meshes, shaders, animations... sẽ được tải trước khi nội dung được trả về.
- **Quản lý bộ nhớ**: địa chỉ không chỉ load asset mà còn unload chúng. Tham chiếu tự động được tính toàn và một profiler mạnh mẽ sẽ cho bạn biết những vấn đề tiềm ẩn với bộ nhớ.
- **Đóng gói nội dung**: Bởi vì hệ thống lập bản đồ và hiểu các chuỗi phụ thuộc phức tạp, nó cho phép đóng gói hiệu quả các gói ngay cả khi asset được thay đổi hoặc đổi tên. Asset có thể được chuẩn bị một cách dễ dàng cho cả triển khai cục bộ và từ xa để hỗ trợ nội dung có thể tải về (DLC) và giảm kích thước ứng dụng.

Với những game có sẵn thì sao???

Addressable Asset cung cấp một lộ trình nâng cấp cho dù bạn đang sử dụng tham chiếu trực tiếp, Resources folder hoặc Asset Bundles.

Overview

Addressable Assets Overview | Package Manager UI website (unity3d.com)

Addressable Assets bao gồm 3 gói:

- Addressable Assets (chính)
- Resource Manager
- Scriptalbe Build Pipeline

Resource Manager và Scriptable Build Pipeline sẽ tự động được cài đặt cùng Addressable Assets

Khái niệm

- **Address** – Xác định một asset để dễ dàng truy xuất trong run-time.
- **AddressableAssetData directory** – Lưu trữ metadata của addressable Asset trong thư mục Assets của dự án.
- **Asset Group** – Một tập hợp addressable Assets sẵn sàng cho xử lý build-time.
- **Asset Group Schema** – Khai báo một tập dữ liệu có thể gán cho một nhóm và sử dụng trong build.
- **AssetReference** – Một đối tượng hoạt động như một tham chiếu trực tiếp, nhưng với khởi tạo hoãn lại (deferred initialization) (Ví dụ như lazy loading). AssetReference lưu trữ GUID dưới dạng địa chỉ mà bạn có thể load theo yêu cầu.
- **Asynchronous Loading** – cho phép vị trí của Asset và các phụ thuộc (local, remote, generated) thay đổi trong suốt quá trình phát triển mà bạn không cần thay đổi code. Tải bất đồng bộ (async loading) là nền tảng của Addressable Asset System.
- **Build Script** – chạy các Asset Group Processors để đóng gói Asset và cung cấp các ánh xạ giữa các địa chỉ (Address) và vị trí của tài nguyên (Resource Location) cho Resource Manager.
- **Labal** – cung cấp thêm một định danh cho addressable Asset để định danh các Asset tương tự nhau trong run-time. VD `Addressables.DownloadDependenciesAsync("spaceHazards");`

Getting started

Getting started with Addressable Assets | Package Manager UI website (unity3d.com)

Yêu cầu

- Unity 2018.3 hoặc mới hơn.

Đánh dấu asset là addressable

Có 2 cách để đánh dấu đối tượng là addressable asset. Khi package Addressable Assets được cài đặt bạn có thể đánh dấu một asset là addressable trong **Inspector** hoặc kéo thả vào cửa sổ **Addressables**.

Trong cửa sổ **Inspector** của đối tượng. Chọn Address và nhập tên để định danh Asset.image.png

Để mở **Addressables** chọn **Window > Asset Management > Addressable Assets**.
Kéo đối tượng từ folder Asset trong cửa sổ Project vào một nhóm trong tab Asset của cửa sổ Addressables.

image-3.png

Địa chỉ mặc định của asset là đường dẫn đến asset trong dự án. Ví dụ *Assets/images/myImage.png*. Bạn có thể sử dụng **Addressables** để thay đổi địa chỉ thành bất kỳ tên unique nào.

Để thay đổi địa chỉ, double click vào address của asset và nhập địa chỉ mới.

Trong lần đầu sử dụng Addressable Asset, hệ thống lưu lại một số dữ liệu asset edit-time và run-time cho project trong `Assets/AddressableAssetsData` (nên được thêm vào version control).

Build Game

Addressables cần build nội dung của bạn vào file mà game có thể chạy trước khi bạn build game. Bước này không được thực hiện tự động. Bạn cần build nội dung thông qua UI hoặc API.

- UI: Mở Addressables. Chọn Build > Build Player Content
- API: `AddressableAssetSettings.BuildPlayerContent()`

Load hoặc Instantiate bằng địa chỉ

Bạn có thể Load hoặc Instantiate một asset trong run-time. Load một asset sẽ load toàn bộ các ràng buộc vào bộ nhớ (bao gồm cả dữ liệu Asset Bundle nếu có). Nó cho phép bạn sử dụng Asset khi cần. Instantiate tải asset và ngay lập tức thêm nó vào scene.

Truy cập một asset trong script sử dụng địa chỉ dạng string:
Addressables.LoadAssetAsync<GameObject>(“AssetAddress”);
hoặc
Addressables.InstantiateAsync(“AssetAddress”);

`LoadAssetAsync` and **`InstantiateAsync`** là xử lý bất đồng bộ. Bạn có thể gọi một callback để làm việc với asset ngay khi n được tải xong. Xem AsyncOperationHandle để biết thêm về hoàn thành chờ.

image-4.png

Sub-Assets và Component

Sub-Assets và Component là các trường hợp đặc biệt đáng để xem xét cho load asset.

- Component – bạn không thể load game object thông qua các component của nó. Bạn phải load/instantiate GameObject và sau đó lấy component bạn cần.
- Sub-Assets – Load sub-asset được hỗ trợ nhưng thông qua cú pháp đặc biệt. Ví dụ sub-asset là sprite trong sprite sheet hoặc animation clip trong một FBX. Cú pháp sẽ là: `Addressables.LoadAssetAsync<IList<Sprite>>("MySpriteSheetAddress");`

Sử dụng AssetReference

AssetReference cung cấp cơ chết để truy cập asset mà không cần đến địa chỉ dạng string.

Để truy cập Addressable Asset sử dụng AssetReference:

1. Chọn 1 asset.
2. Trong Inspector, thêm script bạn cần.
3. Trong script thêm `public AssetReference assetReference;`
4. Quay lại Inspector. Thêm asset vào script trên.

image-8.png

image-6.png

Load Addressable Asset bằng object reference

Để load AssetReference bằng object reference:
`AssetRefMember.LoadAssetAsync<GameObject>();`
Hoặc
`AssetRefMember.InstantiateAsync(pos, rot);`

LoadAssetAsync and **InstantiateAsync** là xử lý bất đồng bộ. Bạn có thể gọi một callback để làm việc với asset ngay khi n được tải xong. Xem AsyncOperationHandle để biết thêm về hoàn thành chờ.

Dữ liệu cục bộ trong StreamingAssets

Addressables cần một số file trong run-time để biết load cái gì và load như thế nào. File này được sinh ra khi build Addressable data và lưu trong folder **StreamingAssets**. Đây mà một folder đặc biệt vì tất cả file trong thư mục này đều được thêm vào build. File không được thêm vào folder này ngay lập tức trong khi build. Thay vào đó chúng được lưu trong **Library**. Khi build game file đươc chuyển qua để build xong đó bị xóa đi. Điều này được thực hiện để người dùng có thể build đa nền tảng. Nhưng yên tâm nó chỉ bao gồm những dữ liệu liên quan đến build đó.

Ngoài dữ liệu Addressables cụ thể, bất kỳ nhóm nào build dữ liệu để sử dụng local cũng sẽ sử dụng vị trí tổ chức sắp xếp cụ thể cho nền tảng của thư viện đó. Để đảm bảo điều này hoạt động, đường dẫn build game nên được đặt thành biến bắt đầu bằng **[UnityEngine.AddressableAssets.Addressables.BuildPath]** và đường dẫn load nên bắt đầu bằng **{UnityEngine.AddressableAssets.Addressables.RuntimePath}.**

Tải xuống trước

Gọi phương thức `Addressables.DownloadDependenciesAsync()` để load các phụ thuộc cho địa chỉ (address) hoặc nhãn (label). Tiêu biểu là asset bundle.

Cấu trúc `AsyncOperationHandle` được trả về bao gồm cả thuộc tính `PercentComplete` – có thể được sử dụng để hiển thị quá trình tải xuống. Bạn có thể sử dụng `PercentComplete` để hiển thị thanh trạng thái và chờ đến khi nội dung được tải xong.

Nếu bạn muốn hỏi người dùng có bằng lòng tải xuống hay không. Bạn có thể sử dụng `Addressables.GetDownloadSize()`. Nó sẽ cho bạn biết dung lượng dữ liệu cần tải xuống. Dung lượng này bao gồm tất cả các gói đã được tải về trước đó (kể cả khi gói đó vẫn được lưu trong cache).

Mặc dù việc tải xuống trước có thể có lợi. Nhưng trong nhiều trường hợp bạn có thể chọn không tải trước. Ví dụ:

- Ứng dụng bao gồm rất nhiều nội dung online, và người dùng thường chỉ tương tác với một phần nội dung đó.
- Bạn có một ứng dụng phải kết nối mạng để hoạt động. Ví dụ như game online. Nếu tất cả dữ liệu của ứng dụng nằm trong một gói nhỏ, bạn có thể chọn chờ và tải xuống khi cần.

Bạn cũng có thể chọn sử dụng một phần của chức năng này. Thay vì sử dụng `PercentComplete` và chờ đến khi tải xong. Bạn có thể tải xuống sau đó tiếp tục ứng dụng. Bạn nên tạo màn hình chờ khi dữ liệu thực sự được tải xuống. Nếu không ứng dụng sẽ vẫn phải chờ cho đến khi tải xuống hoàn tất.

Build đa nền tảng

Khi build nội dung cho Addressable, Asset Bundles được tạo bao gồm Addressable Assets. Asset Bundles phụ thuộc vào nền tảng nên bạn sẽ cần build lại cho mỗi nền tảng mà bạn nhắm tới.

Mặc định, khi Addressable được build dữ liệu cho nền tảng sẽ được lưu vào các thư mục con của đường dẫn Addressable. Đường dẫn runtime sẽ tính đến các folder nền tảng này và trỏ đến dữ liệu được chỉ định.

Chú ý: Khi trong Play Mode trên Editor. Nếu sử dụng script **Addressable Packed Play Mode**, Addressable sẽ tải dữ liệu cho nền tảng mục tiêu bạn đang chọn. Có nghĩa là nếu dữ liệu không tương thích với nền tảng mục tiêu trên editor thì rất có thể vẫn đề này cũng xảy ra trên thiết bị.

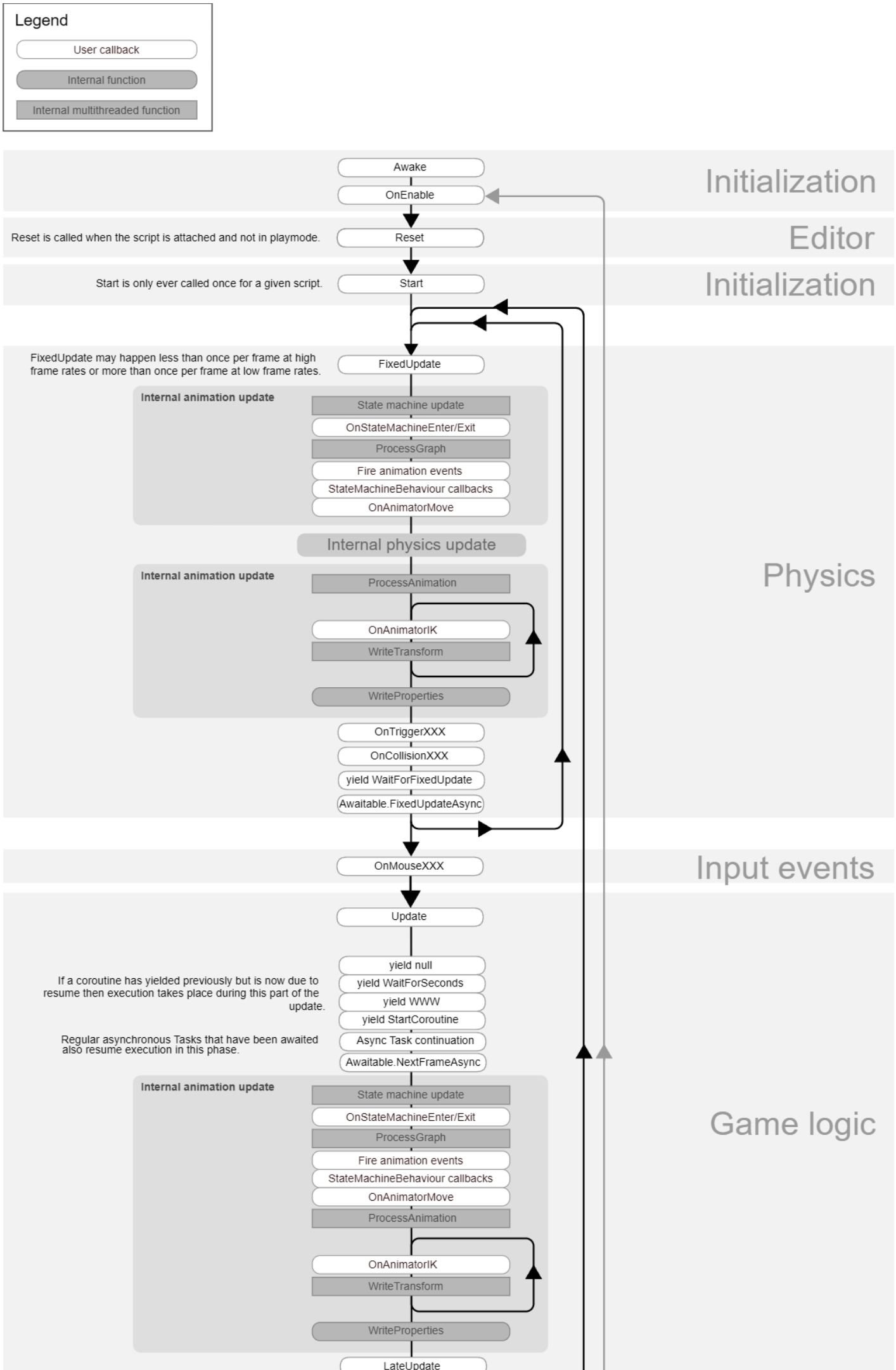
MonoBehaviour và ScriptableObject

Tiêu chí	MonoBehaviour	ScriptableObject
Bản chất	Component **gắn lên GameObject** trong Scene/Prefab	**Asset dữ liệu** độc lập (không gắn GameObject)
Vòng đời	Theo GameObject: `Awake` → `OnEnable` → `Start` → `Update`... → `OnDisable`/`OnDestroy`	Theo Asset: tồn tại như file `.asset`; không có loop mặc định
Khởi tạo	Tạo qua `AddComponent<T>()` hoặc trên Prefab/Scene	Tạo file bằng **Create > ScriptableObject** hoặc `CreateInstance<T>()`
Chứa gì	**Logic/Hành vi** + tham chiếu Component/Transform trong Scene	**Dữ liệu**; có thể chứa logic nhẹ (không phụ thuộc Scene)
Update loop	**Có** (`Update`, `FixedUpdate`, `LateUpdate`...)	**Không**
Serialization	Serialize theo **Scene/Prefab**	Serialize thành **asset**; dễ **chia sẻ giữa nhiều scene**
Tham chiếu Scene	Truy cập trực tiếp GameObject/Component	**Không nên** giữ tham chiếu trên scene (dễ null/leak); giữ **data thuần**
Dùng lại/chia sẻ	Instance gắn với từng GameObject	**Một asset dùng cho nhiều đối tượng/scene** (data-driven)
Bộ nhớ/GC	Tạo/hủy theo vòng đời object trong Scene	Ít cấp phát runtime; phù hợp **giảm trùng lặp dữ liệu**
Use-case chính	Controller, behaviour, input, AI, UI view, spawn...	Config, stat, item database, lookup table, event channel, state container
Làm việc trong Editor	Thường test trong Play Mode	Dễ chỉnh sửa/so sánh như **asset**
Phụ thuộc	Phụ thuộc Scene/Prefab hierarchy	**Độc lập** với Scene; tốt cho **DI/data pipeline**
Hạn chế	Khó chia sẻ state giữa nhiều scene nếu không singleton/DI	Không có lifecycle update; **không** tham chiếu scene trực tiếp
Dùng khi	Cần logic chạy mỗi frame, tương tác thế giới	Cần **dữ liệu chung/cấu hình** tái sử dụng, tách data khỏi logic.

Awake và Start

Tiêu chí	Awake()	Start()
Thời điểm gọi	Ngay khi **script được nạp/khởi tạo** (khi scene load hoặc `Instantiate` prefab)	**Khung hình đầu tiên** khi component **được enable** sau `Awake`/`OnEnable`
Yêu cầu enable	Gọi **dù component bị Disable**	Chỉ gọi khi **component Enable**
Mục đích chính	Khởi tạo **nội bộ**: tạo dữ liệu, tìm & cache reference trong **cùng GameObject**	Khởi tạo **phụ thuộc ngoại**: cần đảm bảo **các đối tượng khác đã Awake xong**
Thứ tự giữa các object	**Không đảm bảo** thứ tự Awake giữa các object khác nhau	**Không đảm bảo** thứ tự Start giữa các object khác nhau
Gọi lại khi Instantiate	Có (mỗi lần `Instantiate`)	Có (khi instance được enable và tới frame đầu)
Best practice	Cache component (`GetComponent`), set giá trị mặc định, đăng ký dịch vụ/DI	Kết nối hệ thống khác, subscribe event ngoài, logic cần đảm bảo đối tượng khác đã sẵn sàng
Lưu ý	Gọi API phụ thuộc đối tượng khác chưa sẵn sàng	Công việc nặng mỗi frame; tránh dùng nếu chỉ cần ở `Awake`

MonoBehaviour Life Circle


Legend
User callback
Internal function
Internal multithreaded function
Initialization
Awake
OnEnable
Editor
Reset is called when the script is attached and not in playmode.
Reset
Initialization
Start is only ever called once for a given script.
Start
Physics
FixedUpdate may happen less than once per frame at high frame rates or more than once per frame at low frame rates.
FixedUpdate
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
Internal physics update
Internal animation update
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
Awaitable.FixedUpdateAsync
Input events
OnMouseXXX
Game logic
Update
yield null
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield WaitForSeconds
yield WWW
yield StartCoroutine
Regular asynchronous Tasks that have been awaited also resume execution in this phase.
Async Task continuation
Awaitable.NextFrameAsync
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
LateUpdate

FixedUpdate, Update và LateUpdate

Tiêu chí	FixedUpdate()	Update()	LateUpdate()
Thời điểm gọi	Theo **nhịp cố định** của physics (mặc định 0.02s)	**Mỗi khung hình** (phụ thuộc FPS)	**Sau mọi** `Update()` trong cùng frame
Tần suất	Có thể chạy **0, 1, hoặc nhiều lần** trong một frame để "đuổi kịp" thời gian vật lý	1 lần/frame	1 lần/frame, **sau** `Update()`
delta time dùng	`Time.fixedDeltaTime`	`Time.deltaTime`	`Time.deltaTime`
Gắn với	**Vật lý** (PhysX): `Rigidbody`, lực, va chạm	**Logic frame**: input, AI nhẹ, UI, timer	**Hậu xử lý** sau Update: camera follow, căn chỉnh vị trí/rotation
timeScale = 0	**Dừng** (physics dừng)	Dừng (trừ khi dùng *unscaled time*)	Dừng (trừ khi dùng *unscaled time*)
Nên dùng cho	Vật lý, tính toán ổn định theo thời gian	Đọc **input**, cập nhật gameplay, hoạt ảnh thủ công, UI	Cập nhật **camera** theo đối tượng đã cập nhật xong; sắp xếp thứ tự/đồng bộ transform
Tránh	Đọc input (không theo frame), thao tác `Transform` trực tiếp gây xé với Update	Gọi API physics bước lớn/không ổn định theo FPS	Logic làm thay đổi dữ liệu mà `Update()` còn phụ thuộc
Ghi chú	Nếu FPS thấp, `FixedUpdate` có thể chạy **nhiều lần/frame**; giữ logic vật lý ở đây để kết quả ổn định	Logic phụ thuộc hiển thị/frame-accurate nên đặt ở đây	Dùng để **"đi sau cùng"**: camera mượt, ràng buộc cuối cùng sau Update

Collision Detection

Khái niệm

- Cách Unity/PhysX (3D) hoặc Box2D (2D) xác định **hai collider có chạm nhau** trong bước mô phỏng.
- Mục tiêu: **tránh "xuyên vật"** (tunneling) khi vật thể **di chuyển nhanh** hoặc collider **mỏng**.

Rigidbody 3D

Chế độ	Mô tả	Dùng khi
Discrete	Mặc định. Kiểm tra tại **các mẫu thời gian rời rạc** (mỗi bước physics). Nhanh nhất nhưng có thể xuyên nếu quá nhanh/mỏng.	Hầu hết vật thể di chuyển chậm/trung bình.
Continuous	Kiểm tra **swept** giữa hai vị trí (last→new) để giảm xuyên qua **static colliders**.	Vật thể di chuyển nhanh, va tường mỏng.
Continuous Dynamic	Nâng cấp của Continuous cho **Rigidbody động nhanh** va vào **đối tượng continuous** khác. Tốn chi phí hơn.	Đạn/vật thể rất nhanh, va chạm động–động.
Continuous Speculative	Dự đoán va chạm (*speculative*) ở bước kế tiếp. Ổn định, chi phí vừa, hiệu quả với vật nhỏ/nhanh.	Vật thể nhỏ, tốc độ cao, cần chi phí hợp lý.

> Continuous/Continuous Dynamic **tốn hiệu năng** hơn Discrete; chỉ bật cho vật thể thật sự cần.

Rigidbody 2D

Chế độ	Mô tả	Dùng khi
Discrete	Mặc định, rẻ. Có thể xuyên khi rất nhanh.	Vật thể chậm/trung bình.
Continuous	Giảm xuyên qua bằng kiểm tra swept 2D.	Vật thể nhanh hoặc collider mỏng.

Trigger vs Collision

Loại	Dùng cho	Callback
Collision (va chạm)	Vật thể **chặn nhau** (phản lực, ma sát…)	`OnCollisionEnter/Stay/Exit`
Trigger	**Không chặn**, chỉ để bắt sự kiện	`OnTriggerEnter/Stay/Exit`

Best Practice

- **Vật nhanh/nhỏ**: dùng **Continuous/Continuous Dynamic** (3D) hoặc **Continuous** (2D).
- **Giữ collider đơn giản** (hộp/cầu/capsule) để giảm chi phí và ổn định tiếp xúc.
- **Di chuyển Rigidbody bằng physics** (`AddForce`, `MovePosition`) — tránh `transform.Translate` để không bỏ qua mô phỏng.
- **Interpolate** (Rigidbody Interpolate) chỉ mượt hình ảnh, **không** thay đổi va chạm.
- Kiểm soát va chạm bằng **Layer Collision Matrix**; tắt những lớp không cần để tiết kiệm.
- Nếu vẫn bị “xuyên”: tăng **Fixed Timestep** (mô phỏng dày hơn), giảm tốc độ đỉnh, hoặc **tăng độ dày** collider.