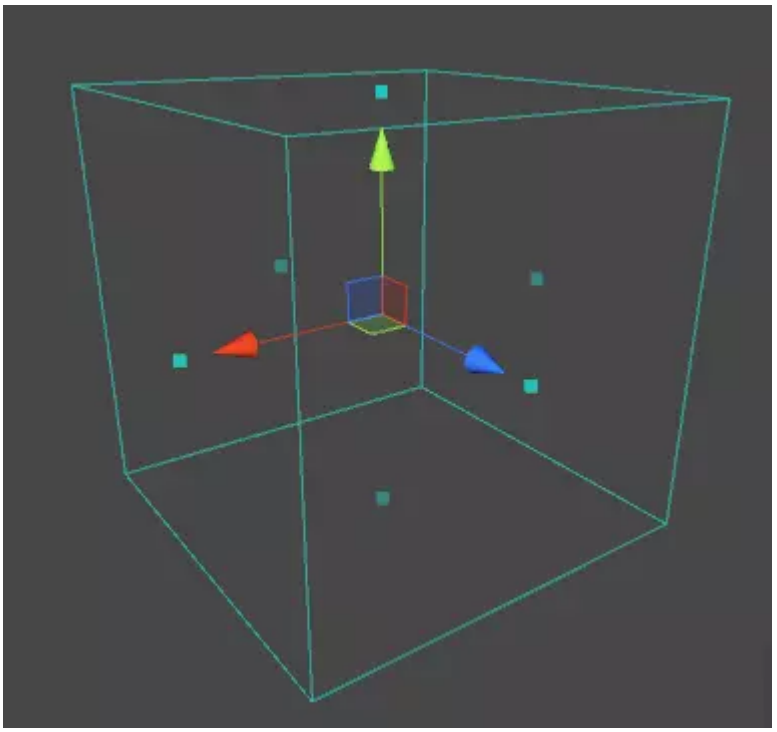
Các thành phần hình học

Trước khi viết scripts, mình sẽ giúp các bạn về các kiến thức nền tảng, các định nghĩa cần biết:

Vector3

Chắc các bạn đều đã được học về vector trong hình học không gian ở cấp 3 (hoặc cấp 2), tương tự Unity cũng vậy. Chúng ta sẽ sử dụng concept Vector3 gồm (x, y, z) để định nghĩa một vector trong không gian game.



Để tạo một Vector3, sử dụng toán tử new và 3 tham số x, y, z:

```
Vector3 u = new Vector3 (1 , 2.0f , -3);
```

Chúng ta cũng có một số Vector được hỗ trợ sẵn:

```
Vector3 a = Vector3.left; // (-1, 0, 0)

Vector3 b = Vector3.up; // (0, 1, 0)
```

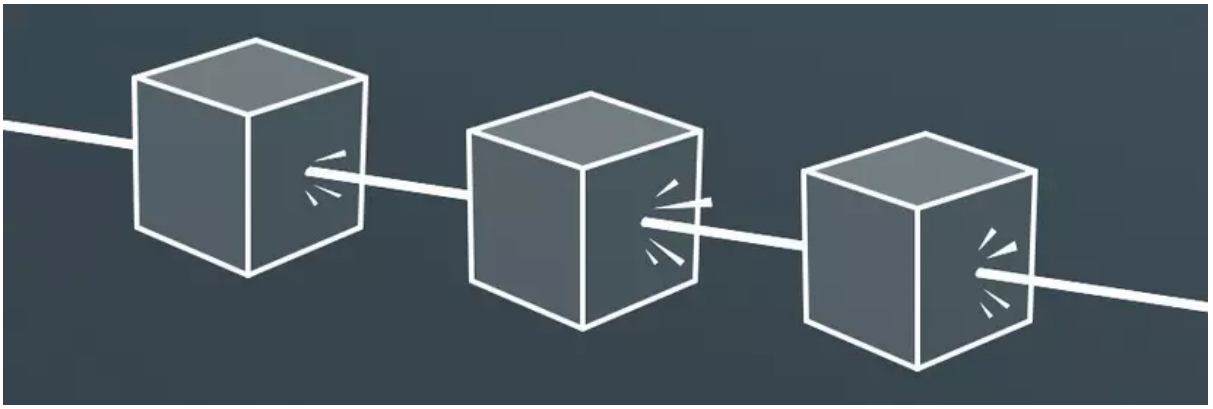
```
Vector3 c = Vector3.forward; // (0, 0, 1)

Vector3 d = Vector3.zero; // (0, 0, 0)
```

Trong Unity thì 3 cột x, y và z được định nghĩa như sau: x là cột nằm ngang có chiều từ trái sang phải, y là cột dọc có chiều từ dưới lên và cuối cùng là z có chiều từ bạn tới màn hình máy tính.

Ray

Ray đơn giản nó là một tia (đường) được định nghĩa bởi 1 điểm gốc gọi là origin và một vector hướng. Bạn có thể tưởng tượng như một tia laser bắn ra từ khẩu súng vậy, nòng của súng là origin và vector hướng là hướng súng đang “chĩa” đến trong không gian 3D.



Ray

Bạn có thể khai báo một Ray bằng cách đưa 2 tham số là origin và direction như sau:

```
// Ray bắn từ vị trí (0, 0, 0), có hướng là (1, 0, 0) "->"
Ray ray = new Ray ( Vector3.zero , Vector3.right );
```

Chúng ta sẽ thảo luận ứng dụng của ray khi tới phần ray-casting nhé.

Quaternion

Quaternion là cấu trúc dùng để biểu diễn hướng xoay của vật trong không gian 3D, tuy nhiên đây không phải là cách duy nhất cung cấp bởi Unity.

Chắc hẳn bạn đã quen với góc xoay: xoay 30 độ quanh trục x (30, 0, 0), xoay 30 độ quanh trục x và 90 độ theo trục y (30, 90, 0). Góc xoay này được biểu diễn bằng cấu trúc gọi là góc euler hay eulerAngles.

Có lẽ góc eulerAngles sẽ quen thuộc với chúng ta hơn là Quaternion, đa số trường hợp khai báo hay sử dụng góc xoay, mình cũng sẽ sử dụng eulerAngles.

Vì vậy thay vì khai báo thẳng giá trị Quaternion, mình sẽ chuyển từ góc euler sang góc quaternion:

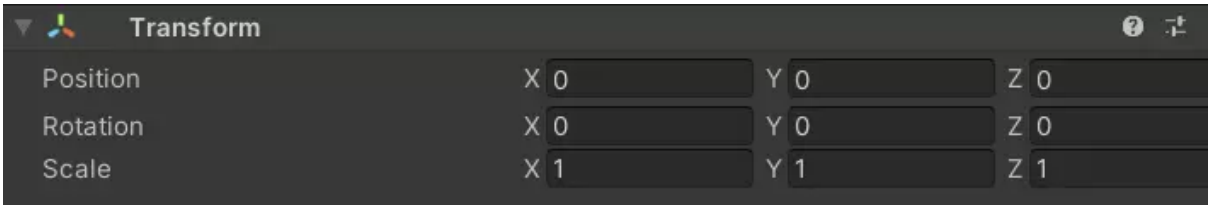
```
Quaternion q1 = Quaternion.Euler (0 , 30 , 0) ;

// xoay 30 độ quanh trục y
Quaternion q2 = Quaternion.AngleAxis (30 , Vector3.up );
```

2 kiểu khai báo trên đều cho ra 1 Quaternion giống nhau nhé.

Transform

Mình đã nhắc tới component Transform rất nhiều ở các bài trước, bởi nó là component được “gắn chặt” vào bất kỳ gameObject nào trên Scene.



Transform Component

Component này sẽ lưu trữ 3 thứ cơ bản sau: vị trí (position), góc xoay (rotation) và tỉ lệ (scale) của một gameObject, thông qua component này bạn có thể truy cập position cũng như rotation bằng cách: transform.rotation, transform.position.

Transform cũng cung cấp các hàm giúp bạn thay đổi position và rotation dễ dàng hơn như Translate(), Rotate():

```
// di chuyển vị trí theo trục y 1 đơn vị
transform.Translate (new Vector3 (0, 1 ,0));

// xoay quanh trục y 30 độ
transform.Rotate (0, 30 ,0);
```

“Các hàm này di chuyển và xoay theo global space hay local space?”

Nếu các bạn đã biết về local space và global space trong Unity thì không cần lo về vấn đề này, bạn có thể cung cấp thêm một tham số là enum Space { Self, World } để chỉ rõ là space nào.

Nhắc lại một chút, chúng ta đã biết về mối quan hệ phân cấp ở bài trước trên tab Hierachy, nếu bạn di chuyển gameObject parent, các children cũng sẽ di chuyển theo và giữ một vị trí tương đối với parent của nó.

Để tham chiếu tới parent, bạn có thể sử dụng component Transform: transform.parent, hoặc thay đổi parent: transform.SetParent(otherParent). Đồng thời từ parent bạn cũng có thể loop qua toàn bộ children của nó.

MonoBehaviour

Như mình đã nói ở đầu bài viết, để tạo một component, chúng ta cần viết một class kế thừa MonoBehaviour

“Tại sao kế thừa MonoBehaviour lại trở thành Component?”

Đây là framework của Unity, chúng ta chỉ cần làm theo thôi, nếu vẫn thắc mắc thì bạn có thể xem cây kế thừa của nó: MonoBehaviour : Behaviour : Component.

Để tạo một scripts, bạn có thể tạo trực tiếp trong gameObject trên Inspector hoặc chuột phải vào tab Project, chọn Create và C# Script.

Cấu trúc của MonoBehaviour

Khi khởi tạo một scripts, giả sử như class tên Temp chẳng hạn, Unity sẽ tạo sẵn template cho các bạn để hướng dẫn cách sử dụng:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Temp : MonoBehaviour
{
    // Start is called before the first frame update
    void Start() {}

    // Update is called once per frame
    void Update() {}
}
```

Nào hãy cùng quan sát một chút đoạn code trên.

Đầu tiên chúng ta có các lệnh `using` để sử dụng các chức năng trong namespace đó như System.Collection để sử dụng các cấu trúc dữ liệu như ArrayList, Stack, Queue, HashTable,… nhưng chúng là của C#.

Chúng ta chỉ quan tâm đến `using UnityEngine;`, nhờ namespace này mà chúng ta có thể sử dụng các concepts mà mình đã hướng dẫn ở trên kể cả MonoBehaviour.

Như vậy là bạn đã tạo được component Temp, chúng ta có thể kéo thả nó vào gameObject bất kỳ hoặc chọn Add Component ở tab Inspector và chọn Temp.

Viết gì trong Scripts?

Đa số các class script trong Unity đều liên quan đến 3 thứ: khởi tạo state (các giá trị của component), chạy update để thực hiện logic theo vòng đời của game, quản lý state.

Template trên đã cung cấp cho bạn 2 hàm rỗng là `Start()` và `Update()` để bạn viết vào, tất nhiên bạn cũng có thể xóa chúng đi. Giả sử component của bạn không cần khởi tạo, bạn có thể xóa hàm Start() đi.

- `Start()`: khi chương trình bắt đầu, hàm này sẽ được gọi trước Update() để khởi tạo các giá trị của component (class).
- `Update()`: chạy liên tục vào mỗi frame, giả sử game của bạn là 60fps, như vậy hàm này sẽ thực thi 60 lần trong 1s, nếu bạn chưa biết về cách game hoạt động có thể đọc qua một chút về fps (frame per second) nhé.

Awake và Start

Mình đã có một bài viết nói về Awake, Start và một số hàm cơ bản trong MonoBehaviour.

Có 2 functions điển hình trong Unity được sử dụng với mục đích khởi tạo đó là `Start()` và `Awake()`, cả 2 hàm này đều chỉ được gọi một lần tương ứng với bất kỳ gameObject nào.

Awake sẽ được gọi đầu tiên, và thực thi ngay khi gameObjects được khởi tạo bởi Unity, trừ khi gameObject đó bị deactive.

“gameObject bị deactive là gì?”

Mỗi gameObject của chúng ta đều có trạng thái active hoặc deactive, được định nghĩa bằng thuộc tính `bool gameObject.activeSelf`.

“Enable/ disable và active/ deactive khác gì nhau?”

Theo mình biết, enable/ disable chỉ sử dụng cho component, nó chỉ dừng component trong gameObject, còn active/ deactive sẽ có tác dụng với gameObject (hoặc toàn bộ components)

Bạn có thể disable từng component riêng lẻ như Rigidbody, Collider,… như vậy các logic trong các components này hầu hết sẽ không được chạy. Ví dụ bạn có component SpriteRenderer dùng để vẽ vật 2D, nếu bạn disable nó đi thì component này sẽ không render ảnh cho bạn.

Khi deactive một gameObject, toàn bộ các components sẽ vào trạng thái inactive và dừng chạy logic như bị disable vậy. Tuy nhiên khi components vào trạng thái deactive không có nghĩa là nó disabled nhé.

"Vậy thì liên quan gì tới Start và Awake?"

- Khi bạn deactive một gameObject, cả Awake() và Start() sẽ không được gọi.
- Khi bạn active một gameObject, và component bị disable, Awake() được gọi nhưng Start() thì không.
- Khi bạn active một gameObject và component được enable, Awake() gọi sau đó đến Start().

Update

Như mình đã đề cập, hàm `Update()` sẽ được gọi trong mỗi frame, 1 giây có thể có 50, 60,… frames. Số lượng frames trong một giây là không có định, vì vậy số lần gọi của nó trong 1 giây cũng không cố định.

Update() được gọi sau Awake() và Start(), khi các bước khởi tạo đã hoàn tất.

Mặc dù không ổn định nhưng bạn có thể cài đặt fps mục tiêu mà bạn hướng tới bằng: `Application.targetFrameRate = 60;` chẳng hạn. Tuy nhiên có thể máy với cpu (/gpu) ổn định, tốc độ tốt sẽ chạy 60 fps, người có cpu yếu có thể sẽ chạy 30 fps.

"Giả sử mình tăng tốc độ nhân vật trong Update(), người có cpu ổn định thì tốc độ nhân vật sẽ tăng 60 lần trong 1s, cpu yếu sẽ là 30 lần trong 1s, tốc độ nhân vật bị ảnh hưởng bởi fps không?"

Đúng vậy, người có 60 fps sẽ chạy nhanh hơn 30 fps trong 1s nếu:

```
private void Update()
{
    // di chuyển sang phải 1 đơn vị mỗi lần Update()
    transform.Translate(new Vector3(1, 0, 0));
}
```

Để fix trường hợp này, chúng ta cần một điều kiện sao cho trong trường hợp 60 fps, tốc độ tăng trong Update() sẽ chậm hơn so với tốc độ tăng trong Update() của 30 fps.

Như vậy để chúng bằng nhau trong 1s thì phải: `60 * x * new Vector3(1, 0, 0) = 30 * x * new Vector3(1, 0, 0);`

May mắn là Unity cung cấp cho chúng ta số x, được gọi là `Time.deltaTime`, biểu diễn thời gian của 1 frame.

"Vậy thì sao?"

Thời gian của 1 frame đối với:

- 60 FPS: 1s / 60 = 0.00166667 (s)
- 30FPS: 1s / 30 = 0.03333333 (s)

Nếu bạn thay x vào, công thức của chúng ta đã đúng: `60 * 1/60 * Vector3 == 30 * 1/30 * Vector3`.

Và đó là lý do tại sao phải nhân Time.deltaTime khi di chuyển:

```
private void Update()
{
    transform.Translate(new Vector3(1, 0, 0) * Time.deltaTime);
}
```

Update và FixedUpdate

Unity còn cung cấp cho chúng ta một hàm tương tự như Update, tuy nhiên có thêm tiền tố "Fixed", nếu bạn đã đọc Update ở trên thì chúng ta biết rằng Update "không ổn định" (do yếu tố fps, giật lag).

Nhưng `FixedUpdate()` sẽ cực kỳ ổn định và được gọi hầu như đúng số lượng frame trong 1s, default của nó là 50 fps với `Time.fixedDeltaTime` là 0.02f, bạn có thể sửa trong settings.

"Vậy tại sao lại dùng FixedUpdate?"

Update của chúng ta không thể áp dụng được cho các yếu tố vật lý nếu FPS không ổn định, ví dụ như vận tốc của một vật bị rơi chẳng hạn, bạn sẽ muốn vận tốc thay đổi theo gia tốc (g) của vật trong mỗi khoảng thời gian đều đặn.

Vì vậy, các xử lý code về vật lý hãy đặt chúng trong FixedUpdate nhằm kết hợp với các hàm vật lý khác của Unity như OnCollisionXXX, OnTriggerXXX,… bạn có thể đọc thêm Execution orders để biết thêm các thông tin khác.

“Còn loại Update nào không?”

LateUpdate

LateUpdate được gọi sau khi tất cả các Update() trong frame đó đã được gọi xong, và trước khi màn hình được redrawing.

Có một use-case duy nhất mà mình sử dụng LateUpdate() đó là di chuyển của Camera, khi bạn muốn camera di chuyển theo nhân vật chẳng hạn, bạn sẽ không biết Update() của camera hay Update() của Player chạy trước, trong khi bạn muốn camera phải bắt kịp vị trí của Player.

Vì vậy đặt xử lý follow player ở LateUpdate(), camera có thể follow Player sau khi Player đã di chuyển xong xuôi (trong Update).