**區分實例new PlayerInput()和直接呼叫PlayerInput的差異性**

1. 如果你使用 playerInput = new PlayerInput(); **創建了一個新的PlayerInput 實例，**通常你會訪問特定動作地圖和動作的方式是使用直接的屬性訪問**，**

**如 playerInput.PlayerAttack.HeavyAttack.triggered;。**這種方式直接映射到特定的動作和動作地圖。

**playerInput = new PlayerInput();**

**bool takeBallTriggered= playerInput.PlayerAttack.HeavyAttack.triggered;**

1. **如果你使用的是 Unity的Input System 而不是直接創建新的實例，**則通常更推薦使用通用的方式，即 playerInput.actions["TakeBall"].triggered;。這種方式更靈活，因為你可以在運行時動態訪問不同的動作。

**bool takeBallTriggered = playerInput.actions["TakeBall"].triggered;**

**實作投射球**

[**https://www.youtube.com/watch?v=dUxSTcPkT9w**](https://www.youtube.com/watch?v=dUxSTcPkT9w)

**這邊利用了，AnimationCurve，**就是說這一個球，移動的軌跡，例如Y軸好了，

他的數字變動可以由Curve預先決定

newPosition.y =

ballFlyHeight \* curve.Evaluate(distanceToFollow / totalDistance);

這行代碼使用了Unity中的Evaluate方法，該方法用於根據輸入值在曲線上進行插值計算。

curve 是一個曲線物件，通常是AnimationCurve類型的物件，用於定義一個曲線形狀。

distanceToFollow 是當前距離，而 totalDistance 是總距離。

distanceToFollow / totalDistance 是一個標準化的值，表示當前距離在總距離中的相對位置（範圍在0到1之間）。

curve.Evaluate(...) 使用標準化的相對位置作為輸入，返回曲線上對應位置的值。這個值乘以 ballFlyHeight 就是新位置 newPosition.y 的值

也就是說AnimationCurve決定數字變動的變動程度或軌跡，至於裡面放甚麼數字自行決定。在這個特定的代碼中，curve 物件的形狀（即曲線的插值方式）將影響最終計算的 newPosition.y。在使用 AnimationCurve時，您可以通過在 Unity 編輯器中調整曲線的關鍵幀來自訂曲線的形狀。每個關鍵幀都有一個時間和一個值，這些關鍵幀之間的插值決定了曲線的形狀。這使您能夠創建各種各樣的曲線，以根據需要調整數值的變化。

目標:

滑鼠點擊地點，為球的落地地點，利用RayCast針對螢幕與滑鼠間投射

Ray ray = cam.ScreenPointToRay(Input.mousePosition);

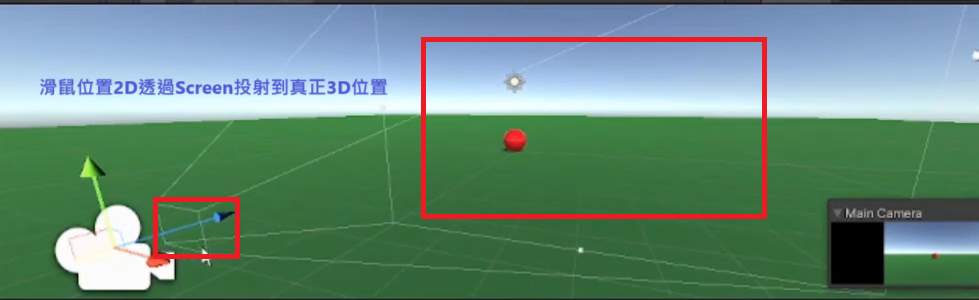
是代表 滑鼠的位置為終點，從螢幕當起點 投射雷射

cam.ScreenPointToRay(Input.mousePosition) 的作用是將滑鼠在螢幕上的位置轉換為一條從攝像機原點發射出去的射線。這個射線的方向是由攝像機的位置和方向決定的，起點是攝像機的位置，終點是滑鼠在螢幕上的位置對應的三維空間中的點。這條射線通常用於進行射線投射，例如用於檢測滑鼠點擊的物件或進行物理射線檢測等

Input.mousePosition 返回的是滑鼠在螢幕上的2D座標。這個2D坐標通常是以螢幕左下角為原點，x軸向右為正，y軸向上為正的坐標系統。

然而，當使用 cam.ScreenPointToRay(Input.mousePosition) 將這個2D坐標轉換為一條射線時，這條射線的終點是滑鼠在螢幕上的位置對應的三維空間中的點。這意味著射線從攝像機的位置穿過滑鼠在螢幕上的位置，並延伸到攝像機所在的3D場景中。

簡單來說，滑鼠位置是在2D螢幕座標中，但由於投射射線的目的，該滑鼠位置會轉換為3D空間中的一條射線，該射線的起點是攝像機的位置，終點是3D空間中與滑鼠在螢幕上位置相對應的點。



RaycastHit hitpo;

Ray ray = cam.ScreenPointToRay(Input.mousePosition);

if(Physics.Raycast(ray,out hitpo))

{

endPoint = hitpo.point;

if(coroutine == null)

{

coroutine = followBall();

StartCoroutine(coroutine);

}

}

IEnumerator followBall()

    {

        pathVector3 = endPoint - ballTransform.position;

        totalDistance = pathVector3.magnitude;//向量取長度

        pathVector3.Normalize();

**//會使pathVector3 成為一個長度為 1 的單位向量。**

**//標準化後的向量方向保持不變，但其長度變為 1。**

**//其實就等於pathVector3 / pathVector3.magnitude**

        distanceToFollow = 0f;

        //ballRadius = ballTransform.localScale.y / 2.0f;

        newPosition = ballTransform.position;//起點球的位置，終點滑鼠的投射

        while (distanceToFollow <= totalDistance)//紀錄過程中移動的距離

        {

        deltaPosition = pathVector3 \* (ballSpeed \* Time.deltaTime);

        newPosition += deltaPosition;

        distanceToFollow += deltaPosition.magnitude;

        newPosition.y =  ballFlyHeight\*curve.Evaluate(distanceToFollow/totalDistance);

**//因為curve.Evaluate只是獲取0~1的數字，因此要乘上flyHeight，每一段的y高度**

**//而這段高度的表現，就是AnimationCurve的圖形表現，可以自由調整**

        ballTransform.position = newPosition;

        yield return null;

**//表示不做任何事情，又再進行一次while,也可以在這邊進行設定幾秒，去做其他程式碼**

**//再回來while,例如yield return new WaitForSeconds(3f);**

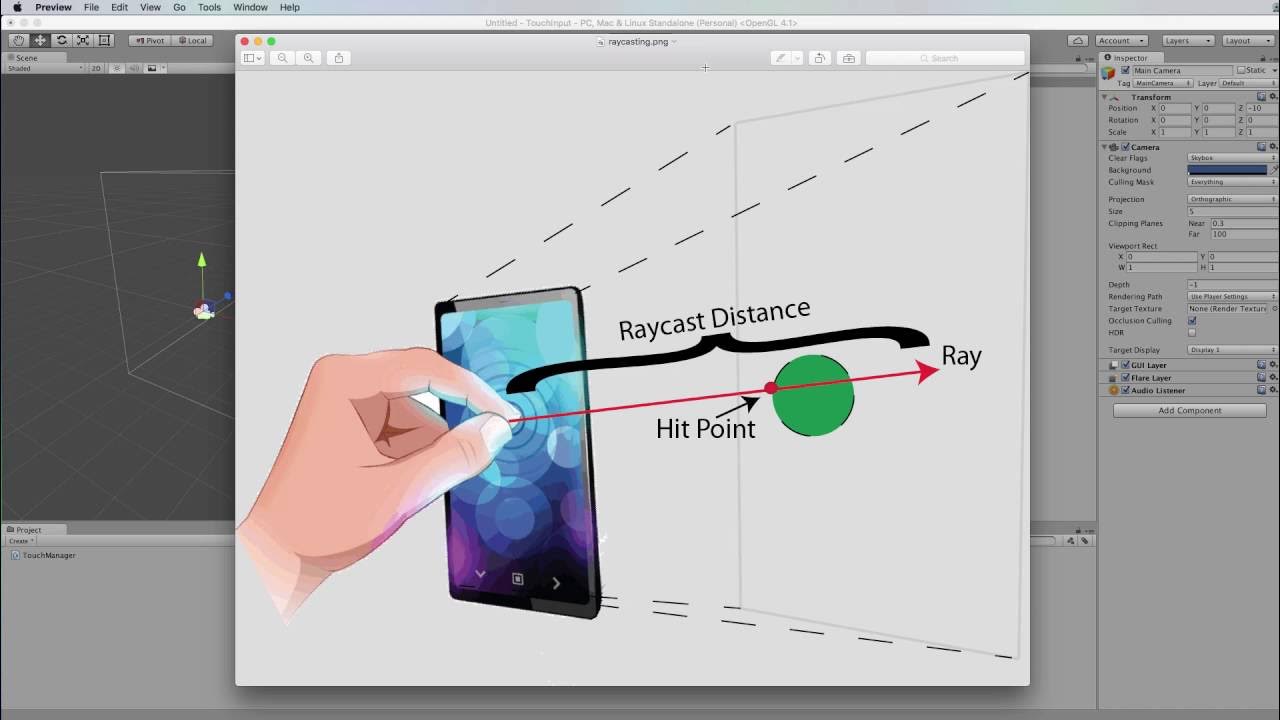
**}**

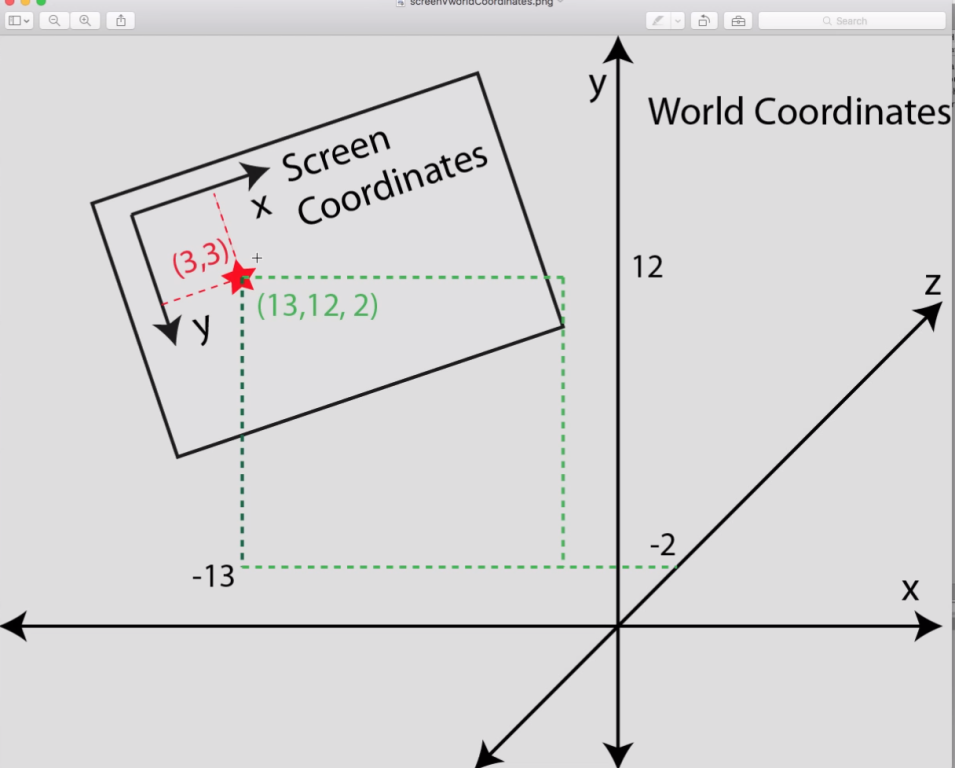
        coroutine = null;

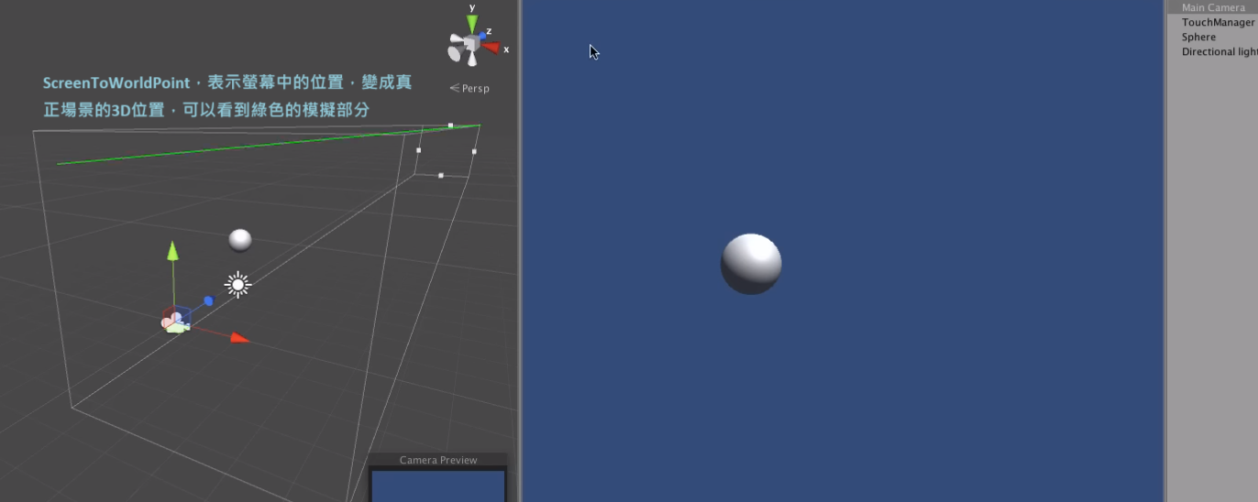
    }

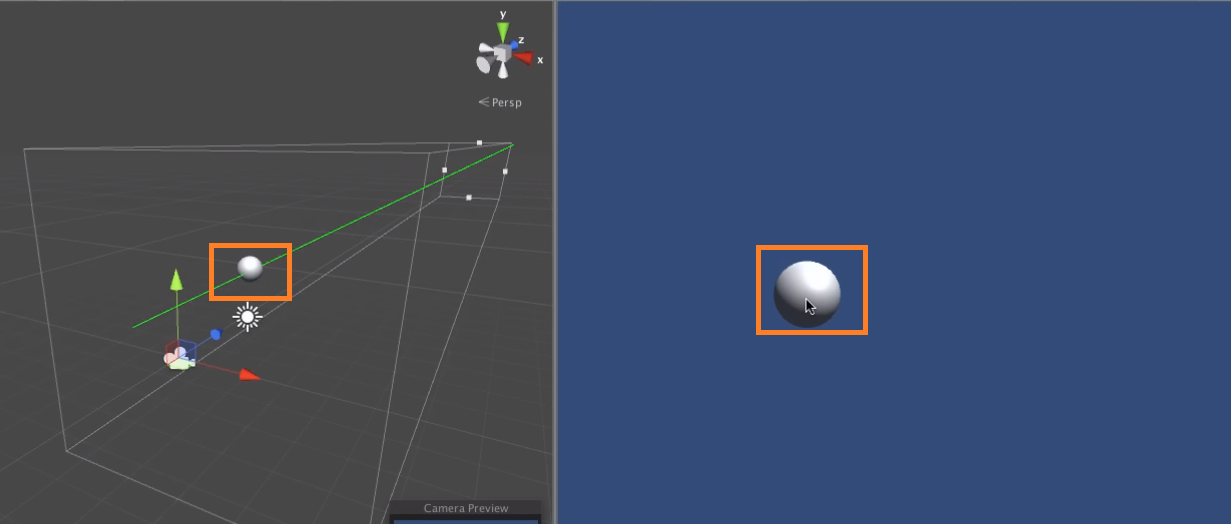
**詳細介紹螢幕點和世界點的雷射關係**

https://www.youtube.com/watch?app=desktop&v=7QomGnOyQoY









<https://www.youtube.com/watch?v=K4DMCseZA08>

實作劃出投射線

public class TrajectProjectile : MonoBehaviour

{

   public Transform shootPosition;

   public GameObject bomb;

   public float launchSpeed = 10.0f;

   [Header("Line Drawing")]

   public LineRenderer lineRender;

   public int linePoints=175;

   public float timeInterValPoints=0.01f;

   void Update()

   {

        if(lineRender !=null)

        {

            if(Input.GetMouseButton(1))

            {

                DrawLine();

                lineRender.enabled = true;

            }

            else

            {

                lineRender.enabled = false;

            }

        }

        if(Input.GetMouseButtonDown(0))

        {

             bomb = Instantiate(bomb,shootPosition.position,shootPosition.rotation);

             bomb.GetComponent<Rigidbody>().velocity = launchSpeed\*shootPosition.up;

        }

   }

   public void DrawLine()

   {

      Vector3 origin = shootPosition.position;

      Vector3 startVelocity = launchSpeed \* shootPosition.up;

      lineRender.positionCount = linePoints;

      float time = 0.0f;

      for(int i=0; i<linePoints;i++)

      {

**//計算物理公式距離: S = Vt + 1/2 \* g \* t^2**

**//這邊取z軸，因為球的前方為z方向，如果是x則要改一下**

            var tempY = (startVelocity.y \* time) + ((Physics.gravity.y)/2 \* time \* time);

            var tempX = (startVelocity.x \* time) + ((Physics.gravity.z)/2 \* time \* time);

            Vector3 points = new Vector3(tempX,tempY,0);

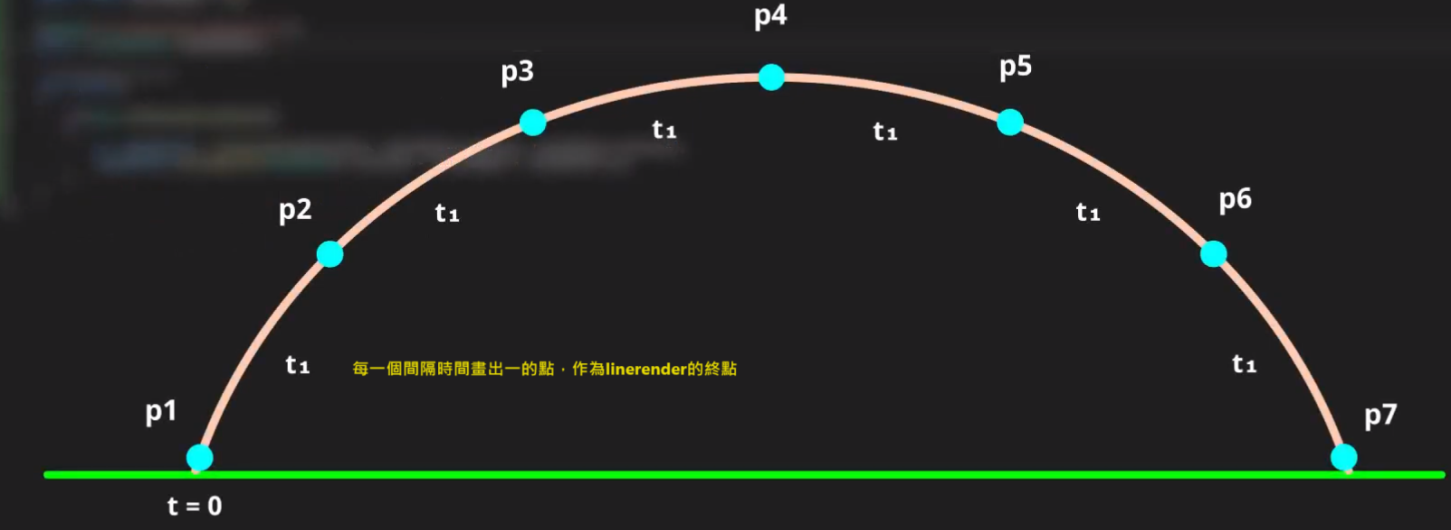
            lineRender.SetPosition(i,origin + points);

            time += timeInterValPoints;

      }

   }

}



<https://www.youtube.com/watch?v=03GHtGyEHas>

**實作投射準確地點**

****

要注意，**shootPosition，不可以離主角太近，不然產生炸彈的時候，會互相碰撞**，讓主角位置會改變，不然就是炸彈彼此之間太靠近。因此要拉遠一點。

shootPosition必須要隨著cursor轉動而轉動，不然他會停留在某一個地方

shootPosition.rotation = Quaternion.LookRotation(cursor.transform.position,Vector3.up);

//射擊的位置，旋轉要繞著y軸，看著Cursor旋轉，不然位置只能待在同一邊，也就是說shootPosition除了會隨著人物轉動而轉動之外，它自己也會繞著cursor轉方向，類似轉頭。

cursor.transform.position = hitInfo.point + Vector3.up\*0.1f;//自己調整要不要往上加一點

而cursor自己也會隨著滑鼠移動而移動

為了人物可以在cursor轉動的時候，也跟著轉動，轉到那方向丟炸彈，要注意人物會歪一邊，因此又要用eular角度來做。先用LooKRotation配合Vector3.up取得四維數

Quaternion targetRotation = Quaternion.LookRotation(targetDistance, Vector3.up);

**targetRotation.eulerAngles** = new Vector3(0, **targetRotation.eulerAngles.y**, 0)

// **只保留 Y 軸旋轉，不會動到其他軸，這樣人物就不會歪一邊**

this.transform.rotation = Quaternion.Slerp(this.transform.rotation,**targetRotation**,25\*Time.deltaTime);

**Vector3 CalculateVelocity(Vector3 target,Vector3 origin,float time)**

**{**

**//定義出x,y的距離**

**Vector3 distance = target-origin;**

**Vector3 distanceXZ = distance;**

**distanceXZ.y = 0;//這個距離，只有x和z而已**

**//創造出個軸的距離**

**float y\_distance = distance.y;**

**//float xz\_distance = distanceXZ.magnitude;**

**float xz\_distance = distance.magnitude;**

**//取得速度V\*t = s, 因此 V = s / t**

**float Vxz = xz\_distance / time;**

**float Vy = (y\_distance / time) + 1/2 \* Mathf.Abs(Physics.gravity.y) \* time ;**

**//原始為 Sy = Vy\*t - 1/2 \* g \* t \* t 移項整理  Vy = Sy / t + 1/2 \* g \* t**

**//將-9.81f變成正數**

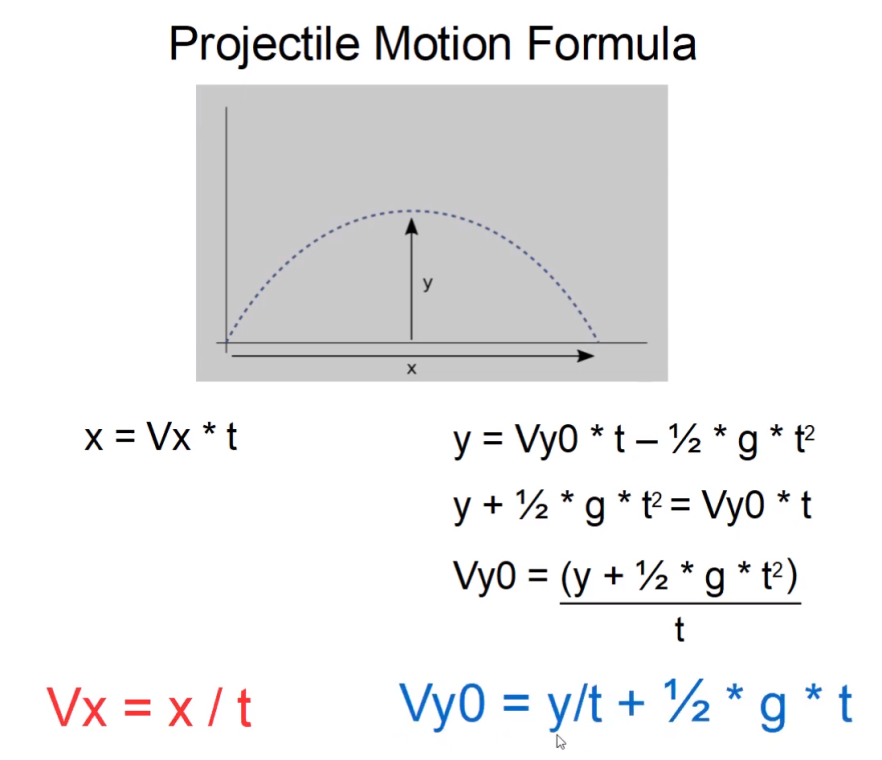
**Vector3 result = distanceXZ.normalized;//標準化0~1**

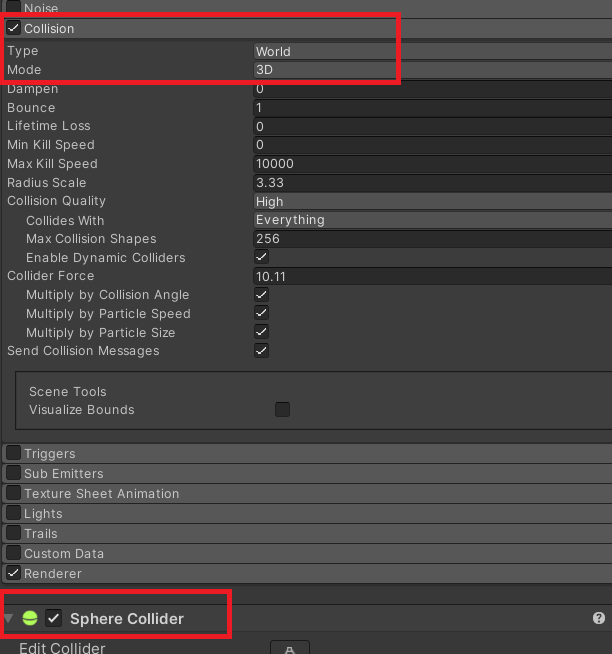
**result \*=Vxz;**

**result.y = Vy\*0.001f;**

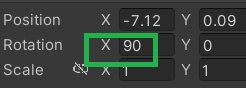
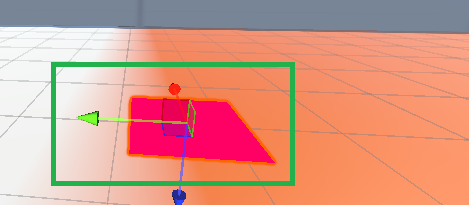
**return result;**

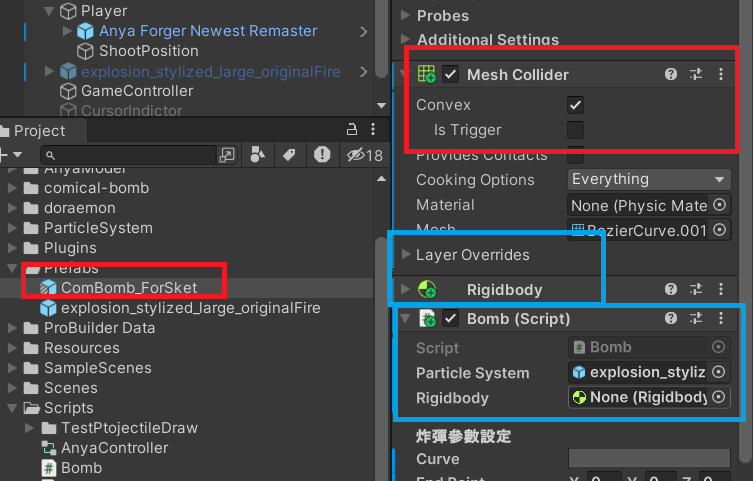
**}**





爆炸的粒子效果，如果要碰撞記得要把Collision打勾，並且也要把send Collsion Message打勾，如果還是沒有碰撞的話，再加上Collider也可以。



****

**炸彈要有mesh Collider**

**才可以放入程式OnCollisionEnter**

**爆炸的產生點，利用Collision.contact.point來製作**

void OnCollisionEnter(Collision collision)

     {

        if (collision.collider.tag == "Target")

            {

**//在 Unity 的碰撞器（Collider）中，collision.collider.tag 用於獲取碰撞的遊戲對象的標簽。**

**//如果您想要獲取碰撞點（collision point），可以使用 collision.contacts。**

**//這是一個包含碰撞點信息的數組，每個元素都是一個 ContactPoint 結構，包含了碰撞點的位置等信息。**

                ContactPoint contact = collision.contacts[0];

                Vector3 b\_position = contact.point;

**//注意contact.point是一個碰撞一堆點的儲存方式，每一個contact儲存很多訊息**

**//例如**

**// 1.point: 一個 Vector3，表示碰撞點在世界坐標系中的位置。**

**// 2.normal: 一個 Vector3，表示碰撞點的法線方向。**

**// 3.separation: 一個浮點數，表示碰撞點的分離距離。**

**// 4.thisCollider: 對撞擊事件中碰撞點所屬的碰撞器的引用。**

**// 5.otherCollider: 對撞擊事件中碰撞點所在碰撞對的另一側碰撞器。**

**//collision.contacts[0] 獲取第一個碰撞點的信息，然後取得裡面的point訊息**

**//collision.contacts 中的每個 ContactPoint 結構都包含一個名為 point 的 Vector3，表示碰撞點的世界坐標。**

**//所以在上述示例中，Vector3 collisionPoint = contact.point;**

**//會將碰撞點的世界坐標存儲在 collisionPoint 變量中。您可以將此變量用作生成爆炸物體的位置。**

**//注意，contact.point 是世界坐標，因此如果您的爆炸物體需要在本地坐標系中生成，**

**//您可能需要進行額外的轉換。**

**//例如，使用 transform.InverseTransformPoint(contact.point) 將世界坐標轉換為本地坐標**。

                GameObject obj = Instantiate(particleSystem,b\_position,Quaternion.identity);

                Destroy(obj,1f);

                gameControllerManager.IsHitting = true;

                         }

     }

void OnCollisionEnter(Collision collision)

{

if (collision.contacts.Length > 0)

**//打到的點有東西才算，通常不加這個條件也是可以，因為打到一定會有接觸點**

{

ContactPoint firstContact = collision.contacts[0];

Vector3 collisionPoint = firstContact.point;

// 獲取第一個碰撞點的位置

Vector3 collisionNormal = firstContact.normal;

// 獲取第一個碰撞點的法線方向

float separationDistance = firstContact.separation; // 獲取分離距離等

// 在這裡可以使用碰撞點的資訊進行其他處理

}

**注意世界軸和人物的本身軸方向**

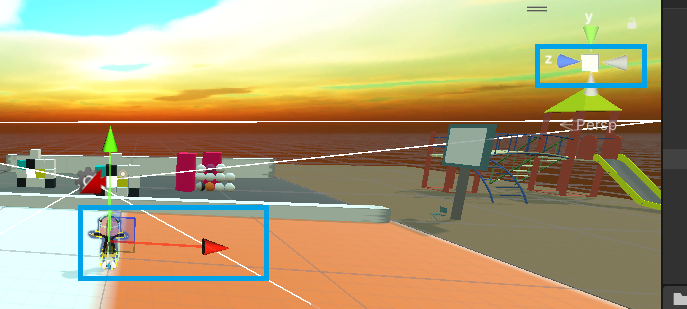
playerControllerMove.rb.AddForce((transform.TransformDirection(Vector3.forward)+Vector3.up)\*10f,ForceMode.Impulse);

**//因為相機前方是X軸，但人物前方是Z軸，如果單純用Vector3.forward,人物會往右邊跳，因為相機的前方式x軸，剛好是人物右邊**

**//因此要利用TransformDirect(Vector3.forward)，將人物前方和世界軸的前方一致性**

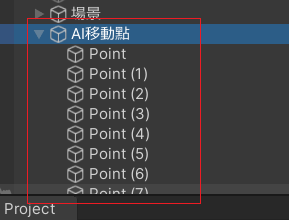
**//可以比較一下單純用Vector3.forward或者是TransformDirection的效果**

**//這邊也可以不要往前跳，單純往上跳就可以Vectro3.up，只是為了更逼真跑到一半往前跳**



**AI隨機移動**

1. 簡單的方式:



**//[]Transform 和 List<Transform> 都是用來存儲 Transform，但它們有一些重要的區別：**

**//固定大小 vs. 動態大小：**

**//[]Transform（陣列）的大小是固定的，一旦創建，無法更改其大小。您必須在創建時指定其大小，並且無法在運行時動態添加或刪除元素。**

**//List<Transform>（動態列表）的大小是動態的，可以在運行時動態添加或刪除元素。**

**//用List來取得waypoints，可以自由調整傳送點數目**

override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)

    {

       gameControllerManager = FindObjectOfType<GameControllerManager>();

       GameObject go = GameObject.FindGameObjectWithTag("WayPoints");

**//只會找到第一個有tag的物件，第二個找不到**

       foreach(Transform t in go.transform)

       {**//找到這個tag底下所有的子物件空白物件，取出它的transform**

            movePoints.Add(t)**;//然後List加入進去**

       }

    }

 override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)

    {

        animator.transform.position = Vector3.MoveTowards(animator.transform.position,movePoints[randomPoint].position,moveSpeed\*Time.deltaTime);

**//MoveTowards保持Y軸不變動，指朝向X,Z方向平面移動**

        gameControllerManager.TurnToOpponent(animator);

        if(Vector3.Distance(animator.transform.position,movePoints[randomPoint].position)<=0.2f)

        {

**//到達目的地的距離0.2f就當作是到達了，最好別用animator.positoin == movePoint這樣判斷 //因為很難數字剛剛好相等，這樣永遠都是false**

            if(waitTime<0f)

            {

                    randomPoint = Random.Range(0,movePoints.Count);

**//從空白物件的數目//0~總數目，產生一個整數，變成移動點的位置，AI在走過去就可**

                    waitTime = startTime;

            }

            else

            {

                waitTime -=Time.deltaTime;

            }

        }

    }

1. 直接利用

//另一種做法可以思考一下，創造一個empty的waypoint

//例如public Transform waypoint它的位置是 waypoint.position = Random(自訂範圍)

//這樣它的位置就是隨機，人物移動到達也是隨機，就可以省去場上建立不必要的waypoints了

**真正做法是利用NavMeshAgent的AI製作**

**利用NavMeshAgent它會自動轉向，這樣比較有逼真。**

製作AI移動的時候需要注意事項

如果只是單純用

override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)

    {

**movePosition = new Vector3 (Random.Range(minX,maxX),0,Random.Range(minZ,maxZ));**

**agent.SetDestination(movePosition);**

}

這樣乍看之下好像每一秒AI都會移動到新的位置，**但是這樣由於地點是隨機的，有可能AI還沒到下一個位置的時候，它位置又備更新了一次，等於是原地打轉**，**這樣很難移動，因此要給它一些緩衝時間，讓他有時間走到隨機指派的位置後，再更新位置**

因此要改成這樣:

**startTime = startTime + Time.deltaTime;**

**if (startTime>=2.0f)**

       {

             movePosition = new Vector3 (Random.Range(minX,maxX),0,Random.Range(minZ,maxZ));

             agent.SetDestination(movePosition);

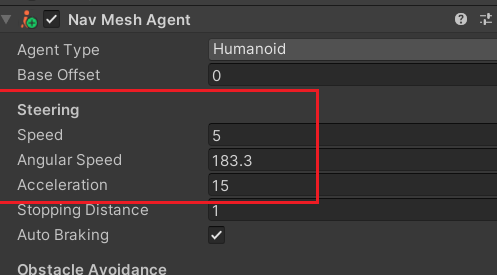
             startTime=0.0f;

       }

**因此要改一下起始時間至少大於startTime > 2秒後，它才有時間走到被隨機指派的位子**

**然後再被更新一次新的位置，一旦更新後startTime又重新計算一次2秒鐘**

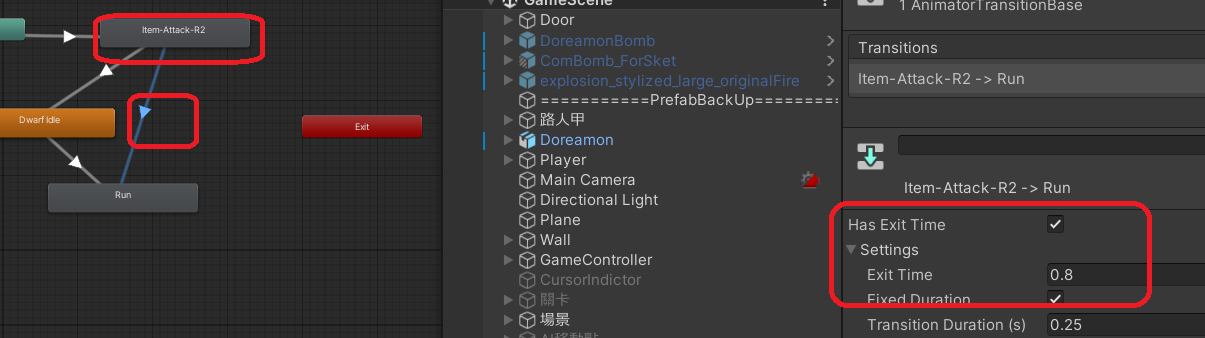
**這樣AI移動位子會比較順，不會還沒走到位子，又被更新新的位子導致原地打轉。**

****

再控制速度不可太快，這邊的速度

就是移動到地點的速度，Angular speed是轉動速度，人物會轉向該目標地點。

**動畫中的Exit Time注意事項**

****

Exti Time表示動畫要撥放到百分比多少，才會離開，如果沒有勾選，那麼動畫隨時可以離開被切斷，如果有勾選Exit Time表示，要撥放到一定比例才離開，**這種丟球動作**

**應該丟完在離開該動畫，不然會有丟到一半就切斷，因此用Exit Time比較合適**

**利用Animator抓取物件，作為參數傳遞**

rigidbody = GetComponent<Rigidbody>();

 gameControllerManager = FindObjectOfType<GameControllerManager>();

animator = GameObject.FindGameObjectWithTag("Player").GetComponentInChildren<Animator>()

**炸彈啟動(搜尋場上的東西)**

public void LockTransform(Animator animator,bool isLocked)

**//鎖住被炸彈炸到的時候，不會被彈開,由animation state給它參數,animator的特性，用animator抓取元件**

    {

        if(isLocked)

        {**//發現因為player的animator和rigidbody不是在同一階層，**

**//因此要用InParent或InChildren才抓的到，因為是用animator來抓物件。**

**//如果Animator和Rigidbody是在同一個物件，無論是用InParent或者直接GetComponent都會抓的到，**

**//因此不影響哆啦A夢的物件抓取，雖然它的Rigidbody和Animator同一物件，但是用InParent仍可以抓的到**

            rb = animator.GetComponentInParent<Rigidbody>();

            rb.isKinematic = true;//**鎖住物理碰撞不被影響**

**接著碰撞後，把找到的animator丟給GameController**

void OnCollisionEnter(Collision collision)     {

        if (collision.collider.tag == "Player")

                ContactPoint contact = collision.contacts[0];

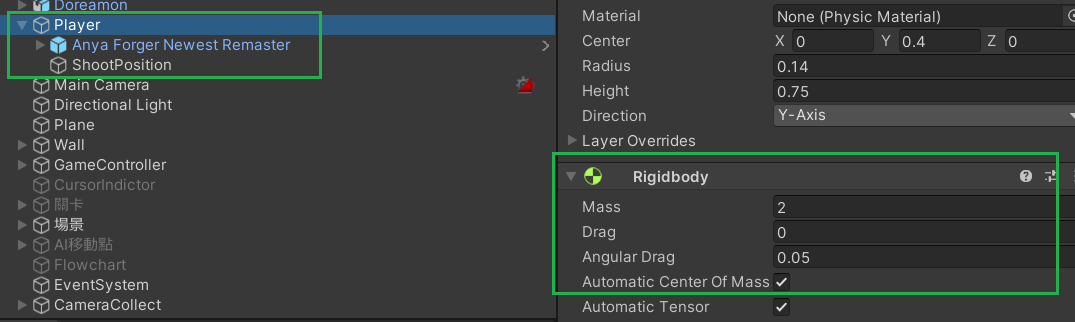
                Vector3 b\_position = contact.point;

                gameControllerManager.LockTransform(**animator**,true);

**//啟動被炸彈打到後鎖住Rigidbody**

**接著碰撞後，把找到的animator丟給GameController**

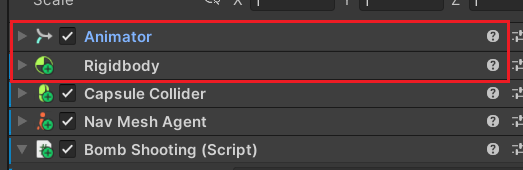
抓取animator的時候，要注意rigidbody以及animator有沒有在同一個階層身上



Player的animator和rigidbody兩者是沒有在同一階層身上，

**因此要用animator.GetComponentInparent<Rigidbody>();**

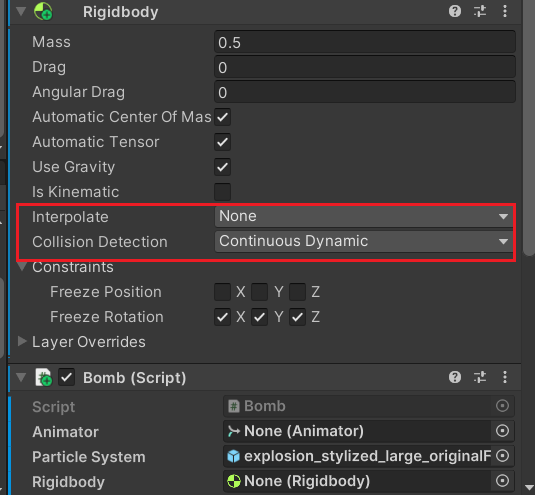
但是敵人身上的Animator 和 Rigidbody在同一個階層身上



**animator.GetComponent<Rigidbody>();**

**最後統一，無論有沒有在同一階層身上，如果都用animator.GetComponentInparent<Rigidbody>();都會抓的到物件**

**當Rigidbody要配上OnEnterCollision的精準度問題**



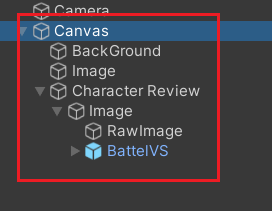
如果偵測的東西沒有太快速，用Discrete的Collsion Detetion就好了

但是這邊是炸彈會跑來跑去

因此改用Continuous Dynamic動態處裡偵測，會比較精準，扣血才不會delay

**製作人物選單UI**

<https://www.youtube.com/watch?v=ZvkIfey70Po&list=PLD_vBJjpCwJv1pVhDpjLeLcEHUKqDrBoc&index=1&t=897s>



UI的物件順序，越底層，顯示的越前面

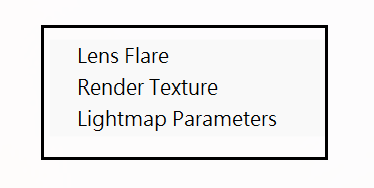
在Unity中，Render Texture（渲染紋理）是一種用於將相機渲染結果保存為紋理的功能。它允許你在遊戲中以各種方式處理和使用相機的渲染輸出。以下是關於Unity中Render Texture的一些基本資訊：

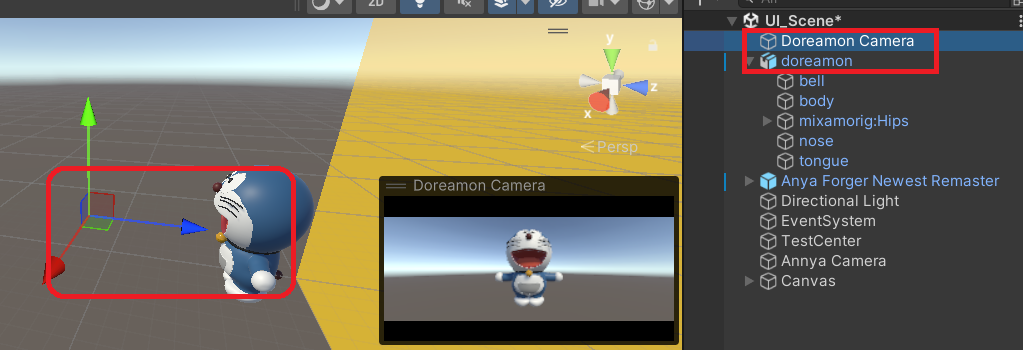
創建 Render Texture： 你可以在Unity編輯器中創建Render Texture。在Project視圖中，右鍵點擊空白處，選擇Create -> Rendering -> Render Texture。然後，你可以設置Render Texture的解析度和其他屬性。

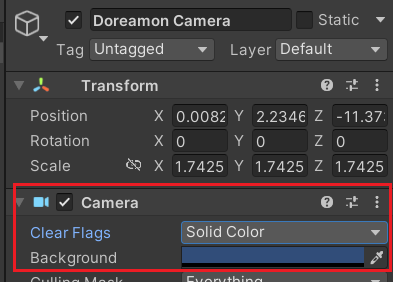
附加 Render Texture 到相機： 要使用Render Texture，你需要將其附加到相機上。在相機的Inspector視圖中，找到Target Texture屬性，將創建的Render Texture拖拽到這個屬性上。

使用 Render Texture： 一旦Render Texture附加到相機上，該相機的渲染結果將被保存到Render Texture而不是螢幕上。你可以將Render Texture用作材質的主紋理，將其顯示在GUI上，或者用於其他需要渲染紋理的場合

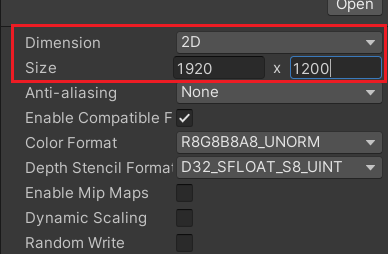
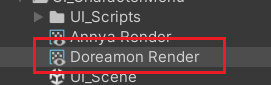
**簡單來說，就是相機必須要呈現的畫面，無論它是在哪一個背景之下**

1. **建立一台專屬人物的相機，也就是Camera，一台相機只有一個Render Texture**
2. **在asset的create建立**
3. **加入3D模型，把相機調整到可以照到你要的3D模型**

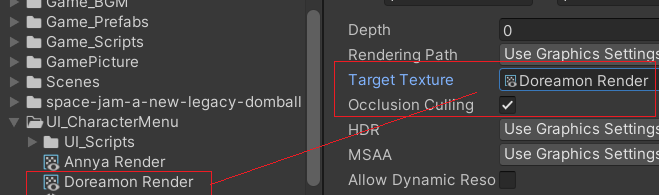
****

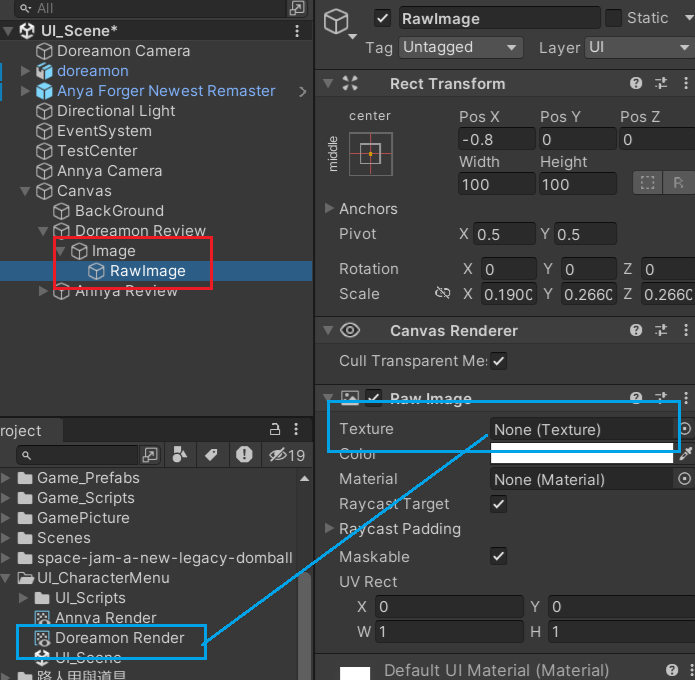
把Camera的 Clear Flags改成Solid Color目的是只要看到人物計就可以



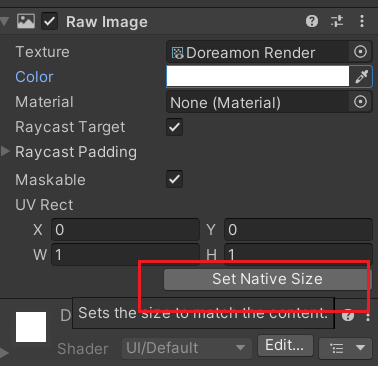
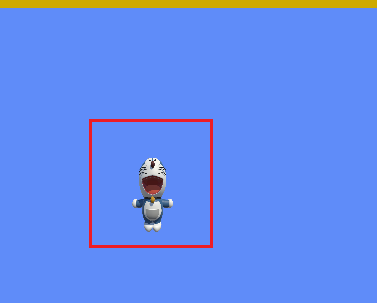
調整Render Texture的大小解析度



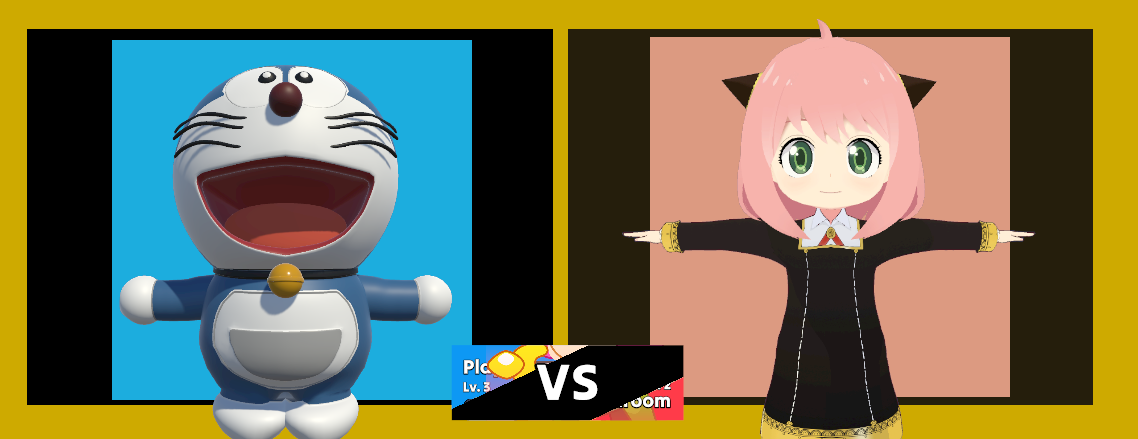
**接著調整完解析度後，把RenderTexure 拉到Camera的Target Texture**



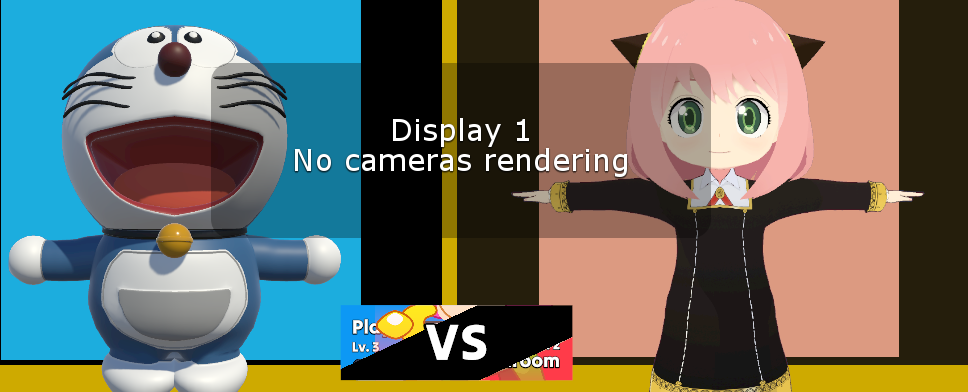
接著新增UI🡪Raw Image，把剛剛的Texture Render拉入關係



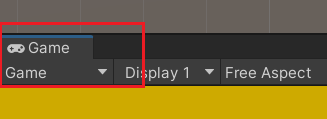
發現出現的畫面很小，因此改成Set Native Size按下去後，自己調整大小



這些3D模型可以在加上動作，直接在相機上會出現，因為這些模型都是在場景上面

如果出現在左邊這樣有No Camera

Renendering的話，

就在  
**按右鍵，取消waring if no camera renendering**

****

在Canvas特別的是，越底層，顯示越前面，但是一旦調整好圖層關係後，就不用再把該物件放在最底層了，可以獨立拉出來，不會影響圖層，因此圖層只要確定後，就可以不用在附屬關係了。

接著寫一個程式碼讓滑鼠移動可以轉動該人物

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.InputSystem;**//記得放入，不然InputSystem無法運做**

public class CharacterRotation\_UI : MonoBehaviour

{

    public  PlayerInput playerInput;

    public Vector2 cameraInput;

    public float rotationSpeed=4.0f;

    public Vector3 currentRotatnion;

    public Vector3 targetRotation;

    void Start()

    {

        playerInput = GetComponent<PlayerInput>();

        currentRotatnion = this.transform.eulerAngles;/**/掛在該人物身上，當然是用本身transform**

        targetRotation = this.transform.eulerAngles;

    }

    void Update()

**{//這邊沒有用New Class()的做法，而是直接用actions.trigger or ReadValue**

        cameraInput = playerInput.actions["Camera"].ReadValue<Vector2>();

        if (cameraInput.x >0)

        {

            targetRotation.y = targetRotation.y+5.0f;

        }

        else if(cameraInput.x < 0)

        {

             targetRotation.y = targetRotation.y -5.0f;

        }

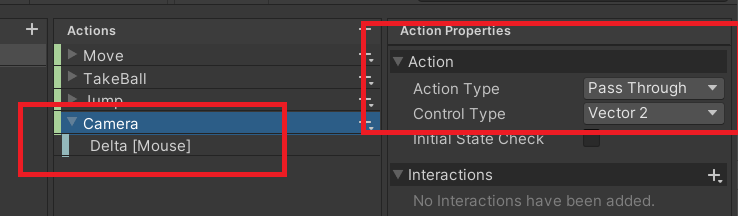
**//這邊的Vector3.Lerp和Quaternion.Slerp是一樣的意思，只是一個資料型態是Vector3,一個是Quaternion型態不同**

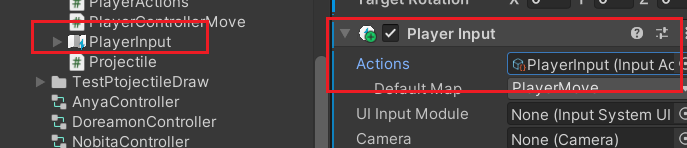
        currentRotatnion = Vector3.Lerp(currentRotatnion,targetRotation,rotationSpeed\*Time.deltaTime);

        this.transform.eulerAngles = currentRotatnion;

    }

}





**Player Input是自己必須新增的元件，再把他們拉入關係**

**Mathf.Clamp01 限制值在 0 和 1 之間。**

**如果值小於 0，則返回 0；如果值大於 1，則返回 1；否則返回原始值。**

**Mathf.Clamp 允許你指定任意範圍。**

**它接受三個參數：要限制的值，範圍的最小值和範圍的最大值。如果值小於最小值，**

**則返回最小值；如果值大於最大值，則返回最大值；否則返回原始值。**