Álvaro Holguera
CIFP José Luis Garci
(Madrid)

UD 02
INTERACTIVIDAD Y MOVIMIENTO

Desarrollo de Entornos Interactivos Multidispositivo

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

INTERACTIVIDAD

Ahora que conocemos Unity, comencemos a realizar tareas básicas como posicionar objetos en el escenario, moverlos y dotar de interactividad a nuestro juego

Recuerda, todo lo que vamos a ver de aquí en adelante lo puedes completar en la documentación de Unity: "Help > Scripting Reference"

CREANDO VECTORES

Un Vector es una estructura usada para pasar posiciones en el espacio **2**D y **3**D, así como direcciones de desplazamiento.

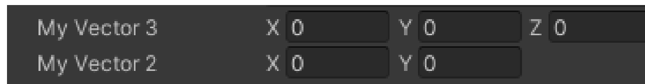
Vamos a crear 2 vectores, uno tridimensional y otro bidimensional, y probar a cambiar sus valores para mostrarlos en consola. Para ello, crea un script y asócialo a un objeto de la escena:

```
//Creamos un nuevo objeto de la clase Vector3
[SerializeField] Vector3 myVector3 = new Vector3();

//Hacemos lo mismo con un vector bidimensional
[SerializeField] Vector2 myVector2 = new Vector2();


// Start is called before the first frame update
void Start()
{
    //Mostramos en consola su valor
    print(myVector3);
    print(myVector2);
}
```

Como verás, he creado los objetos serializados para poder cambiar sus valores en Unity, que como comprobarás permite poner valores decimales en cada eje:



Qué es un vector

Hay 2 tipos principalmente: vectores tridimensionales, que contienen valores en los ejes X, Y, Z, y vectores bidimensionales con posiciones en X y en Y.

Son una clase heredada, por lo que si queremos crear un Vector debemos "instanciar" esa clase. Y a partir de ese momento tendremos acceso a sus atributos y métodos.

```
Vector3 miVector = new Vector3();
```

RECOMENDACIÓN: es una buena práctica consultar en la página de referencia lo que nos ofrecen las clases de Unity. Encontrarás una lista de atributos y métodos, con enlaces e información sobre lo que hacen, e incluso si necesitan librerías adicionales. Visita la página y mira qué ofrece la clase de Vector3 por ejemplo: https://docs.unity3d.com/ScriptReference/Vector3.html

También podemos darle directamente los valores mediante código, en el momento de crear el objeto:

```
3 of 3  Vector3(float x, float y, float z)
        Creates a new vector with given x, y, z components.

//Creamos un nuevo objeto de la clase Vector3
[SerializeField] Vector3 myVector3 = new Vector3(5f,2f,6f);
```

Fíjate cómo Visual Studio nos ayuda indicando qué valores tengo que poner, y de qué tipo, en este caso "float".

Al lanzar el juego verás que muestra en consola los valores que hayas puesto.

Operaciones con vectores

Los vectores pueden sumarse entre ellos, restarse, o incluso multiplicarse y dividirse.

Para ejemplificarlo, vamos a coger un Vector3 que marcará la posición inicial en el espacio, y otro vector que indica cuánto se va a desplazar (una unidad) y la dirección (en el eje Y, es decir, hacia arriba), y los sumaremos. Mira y analiza el siguiente código:

```
//Vector inicial, ubicado en las coordenadas 0,0,0
Vector3 initVector = new Vector3(0f, 0f, 0f);

//Vector que indica la dirección y la cantidad del desplazamiento
//Como verás, lo que hace es desplazars uno en el eje Y, hacia arriba
Vector3 desplVector = new Vector3(0f, 1f, 0f);

//Creamos un bucle y en cada ciclo suma el segundo vector al primero
for(int n = 0; n < 10; n++)
{
    print(initVector);

    //Lo sumamos
    initVector = initVector + desplVector;
}
```

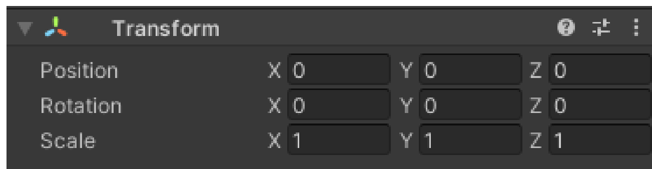
PARA NOTA: hay formas de crear vectores "al vuelo" usando los métodos de la clase con valores predefinidos. Mira los siguientes ejemplos y lánzalos a consola para comprobarlo (¿podíamos haber usado alguno para el ejemplo anterior?):

```
Vector3 newVector1 = Vector3.zero;
Vector3 newVector2 = Vector3.up;
Vector3 newVector3 = Vector3.right;
```

POSICIONANDO OBJETOS

Siempre que queremos modificar la posición, rotación o la escala de un objeto de la escena, tenemos que acceder a los atributos de su componente Transform, ya sea para tomar sus valores iniciales o para modificarlos

El componente "Transform" (que actúa como una clase) tiene a su vez 3 subclases (Poistion, Rotation, Scale) que a su vez tienen 3 atributos X-Y-Z.



En todo momento podemos conocer sus valores, así por ejemplo: "transform. position.y" nos devuelve el valor en el eje Y del objeto.

Prueba a crear un objeto en la escena y asóciale un script con este código. Lanza el juego y mira a ver qué pasa en la consola cuando mueves el objeto arriba y abajo:

```
// Update is called once per frame
void Update()
{
    float posY;
    posY = transform.position.y;
    print(posY);
}
```

Como verás, la variable "posY" toma el valor de la posición en el eje Y del objeto para mostrarlo en consola, y como está en el método "Update", se actualiza

IMPORTANTE: si te fijas, transform se escribe aquí con minúscula, porque es una instancia de la clase Transform. Cuando tengamos que dirigirnos a la clase, lo haremos escribiéndola en mayúscula.

Cambiar los valores

Siempre que queremos asignar una nueva posición, o rotación, o escala a nuestro objeto, debemos crear un nuevo "vector3", y especificar los nuevos valores, de esta forma:

```
transform.position = new Vector3(valorX,valorY,valorZ);
```

Por ejemplo, si queremos que un elemento comience siempre en la posición 0, debemos añadir en el método Start:

```
transform.position = new Vector3(0f,0f,0f);
```

o mejor aún:

```
transform.position = Vector3.zero;
```

Prueba ahora a mover el cubo antes de lanzar el juego, y dale a "Play". Verás que se pone en las coordenadas 0,0,0 al iniciar.

LA ESCALA

Desplazar otro objeto

Ahora vamos a cambiar la posición de un objeto, pero no el que tiene asociado el script, sino otro de la escena, para que veamos que tenemos un control absoluto de la escena.

Para ello, crea otro objeto en la escena, distinto al que tiene el script, y añade el siguiente código al script:

```
[SerializeField] Transform otherObject;

// Start is called before the first frame update
void Start()
{
    otherObject.position = new Vector3(10f, 10f, 10f);
}
```

Analicemos:

• Hemos creado una variable de tipo Transform (fíjate que aquí va con mayúscula, porque hace referencia a la clase), llamada "otherObject", que actuará como instancia de esa clase.

• Al ser serializada, nos permite en Unity arrastrar el objeto que queramos controlar al componente del Script. Al hacerlo, la variable "otherObject" se convierte en una instancia del componente Transform del otro objeto.

• Como buena instancia, tiene disponible todos sus atributos, por ejemplo el de posición. Eso nos permite cambiar su valor con un vecto3.

NOTA: El componente Transform es tan importante, que está disponible directamente en el código mediante el objeto "transform", sin tener que crear una instancia de la clase. Vermos más adelante que si queremos acceder a otros componentes, deberemos instanciarlos mediante el método GetComponent.

Si lanzamos el juego, veremos que el otro objeto se coloca en las coordenadas 10,10,10.

La escala

Si queremos cambiar la escala de un GameObject, debemos aplicar un Vector tridimensional a su atributo "localScale". Por ejemplo, de esta manera duplicamos el tamaño de nuestro objeto:

```
transform.localScale = new Vector3(2f, 2f, 2f);
```

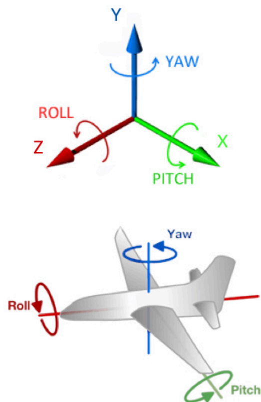
PARA NOTA: aunque queda mucho, que sepas que este atributo admite valores negativos, por lo que si escalamos un dibujo a -1 en el eje Y, le habremos dado la vuelta (flipeado). Recuérdalo.

EL PROBLEMA DE LA ROTACIÓN

Si bien la posición y la escala funcionan con Vectores tridimensionales sin problema, ya que las modificaciones se hacen en 3 ejes, la rotación es más compleja, ya que permite rotaciones en 4 ejes, los llamados Quaterniones. Por ello, el atributo "transform.rotation" pide un Quaternion, no un Vector3.

Su explicación excede los contenidos de esta Unidad, pero debemos saber que existe un sistema sencillo para describir la orientación en un plano 3D, usando un sistema de 3 coordenadas, son los ángulos Euler (llamados así porque son los que se usan en aeronáutica). Aplicado a los ejes de Unity quedaría así:

- Pitch: giro en torno al eje X (levantar y bajar el morro)
- Roll: giros en torno al eje Z (ladear el avión)
- Roll: giros en torno al eje Y (giro lateral, de alerón trasero)



Podemos construir un Quaternion para pasárselo al atributo transform.rotation usando el método Eulers de la clase Quaternion:

```
Quaternion.Euler (pitch, yaw, roll);
```

Los valores se expresan en grados.

Prueba este código en un cubo y verás cómo se gira 45° en todos los ejes:

```
Quaternion targetRotation = Quaternion.Euler(45f, 45f, 45f);
transform.rotation = targetRotation;
```

Ahora que lo comprendes, que sepas que la clase Transform incluye un método que hace este proceso de convertir un Vector3 en Quaterniones al vuelo, el "transform.eulerAngels". Prueba este código y verás que el resultado es el mismo:

```
transform.eulerAngles = new Vector3(45f, 45f, 45f);
```

Mas información sobre los quaterniones y los métodos que ofrecen la puedes encontrar en:

https://docs.unity3d.com/ScriptReference/Quaternion.html

PARA NOTA: siempre es bueno leer y comprender los ejemplos que ofrece la documentación de Unity. En esta página: https://docs.unity3d.com/ScriptReference/Transform-rotation.html encontrarás un ejemplo no solo de cómo rotar, sino de hacerlo suavemente usando el método Slerp de Quaternion.

PRACTICA

Haz pruebas, pero cambiando no solo la posición sino también la escala y la rotación del objeto que tiene el script y del otro objeto.

CREAR MOVIMIENTO

Hasta ahora, hemos cambiado las propiedades del objeto en el método Start, pero ¿qué ocurre si lo hacemos en el método Update? Nada porque siempre le damos el mismo valor. Pero, ¿y si ese valor cambia en cada fotograma?

Hagamos la prueba: crea una variable de tipo decimal que indicará nuestra posición en Z, dile a nuestro objeto que se posicione en todo momento en las coordenadas x = 0, y = 0, Z = el valor de la variable.

Ahora súmale 1 a esa variable en cada fotograma. Veremos qué ocurre. Aquí tienes el código:

```
//Creamos las variables
float posZ = 0f;
Vector3 newPos;

private void Update()
{
    //Asignamos el valor al Vector que determina nuestra posición
    newPos = new Vector3(0f, 0f, posZ);
    //Movemos el objeto
    transform.position = newPos;
    //Aumentamos uno al valor en Z
    posZ++;
}
```

Verás que el elemento sale disparado, y es lógico: le hemos dicho que se mueva un metro hacia atrás en cada fotograma.

Una forma de controlarlo es crear una variable float que establezca la velocidad, y multiplicar el vector “newPos” por esa variable (recuerda que se puede operar con vectores).

PARA NOTA: si el juego se ejecuta a 60fps, ese objeto se mueve a 60 mts/seg. Pero ¿y si nuestro ordenador va lento y el juego va a 30fps? ¿No resulta eso un problema, ya que la velocidad de los objetos no debe depender de los fps del juego? Esto lo resolveremos cuando hablemos del tiempo en Unity al final de esta unidad.

EJERCICIO: trata de recrear ese movimiento con la escala y la rotación.

Si consultas la clase Tranform, descubrirás que tiene una seríe de métodos públicos muy útiles:

Método Translate

Si recordamos, los vectores no solo indicaban posocioens, también direcciones de desplazamiento. Si le pasamos un vector a este método, movéra nuestro objeto en esa dirección.

NOTA: cuanto más longitud tiene el vector de desplazamiento, más rápido lo moverá.

Veamos un ejemplo:

```
// Update is called once per frame
void Update()
{
    Vector3 despl = new Vector3(0f, 1f, 0f);
    transform.Translate(despl);
}
```

En el código indicado, hemos creado un Vector3 que indica el desplazamiento (hacia arriba a una velocidad relativa de 1), y se lo hemos pasado al método Translate de transform.

En estos casos, es muy práctico los métodos que vimos de la clase Vector3: up, down, forward, left, right... De esta forma, el script anterior lo podemos reducir a:

```
transform.Translate(Vector3.up);
```

CREAR ROTACIÓN

Funciona exactamente igual que el anterior, y también recibe un Vector3 para aplicar la rotación.

Tienes que tener en cuenta que este método hace una conversión a ángulos Eulers del vector recibido, por lo que los valores que pases corresponderán con lo visto anteriormente: roll para el Z, pitch para el X y yaw para el Y. Por eso, este código hará que gire sobre su eje X:

```
transform.Rotate(Vector3.right);
```

NOTA: recuerda que las unidades de medida al rotar son los grados. Por eso notarás que no va tan rápido como el desplazamiento.

Pactica con el avión creado antes:

1. Crea variables para los ejes de desplazamiento y rotación. Haz que sean accesibles desde Unity

2. Añade scripts de traslación y rotación al avión, y asegúrate de que a los vectores aplicados les corresponden las variables que has creado para cada eje.

3. Lanza el juego y prueba a mover y rotar el avión cambiando los valores de las variables directamente en el panel del inspector. Prueba también valores negativos.

Prueba a rotar en varios ejes a la vez, y descubrirás lo que son los Quaterniones y por qué pilotar un avión no es tan fácil. Prueba a también el espacio de giro de local a global (Space.World) y verás que ocurre.

NOTA: Tanto en el método Translate como en Rotate, hay un segundo parámetro opcional que indica si los ejes deben tomarse en relación al mundo o al objeto, ya que si éste está girado, el concepto de "arriba" no es exactamente hacia arriba, salvo que sea respecto al mundo. Por defecto entiende que son relativos (Space.Self), pero si queremos que sea respecto al mundo deberemos añadir "Space.World". Así:

```
transform.Rotate(Vector3.up, Space.
World);
```

LookAt

La clase Transform tiene otro método muy útil, LookAt, que hace que el objeto se gire en todo momento para mirar a otro objeto (o más concretamente, a los parámetros de su componente Transform).

Para probarlo, vamos a realizar una prueba en nuestro avión. Crea un script como el que se muestra, y a la variable Transform arrastra el objeto al que quieras que mire. Lanza el juego y mueve el objeto que sirve de objetivo por el escenario:

```
//Variable Transform que contendrá el objeto al que mirar
[SerializeField] Transform target;

void Update()
{
    transform.LookAt(target);
}
```

PARA NOTA: Es bueno que conozcas algunas funciones avanzadas con vectores que te pueden resultar útiles:

- Vector unitario (nomralizado): Vector.normalize / Vector.normalized
- Magnitud (Magnitude): longitud del vector
- Distancia (Distance): distancia entre dos vectores
- Move hacia otro objeto (MoveTowards)
- Dot: compara como se miran dos elementos
- Cross: obtiene un vector perpendicular entre otros dosww

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

EL TIEMPO EN UNITY

Conozcamos a continuación, como podemos controlar el tiempo en Unity

LA CLASE TIME

El control del tiempo en Unity se realiza mediante la clase Time, y todos sus atributos y métodos.

Un ejemplo, el método Time.time nos devuelve el nº de segundos que han pasado desde que se inició el juego. Este parámetro es útil por ejemplo para crear ráfagas de disparo a intervalos regulares usando el método Update. Observa este ejemplo insparado en la documentación de Unity y trata de descifrarlo:

```
//Intervalo para el disparo
public float fireRate = 0.5f;

//Valor que indica cuándo será el siguiente disparo
private float nextFire = 0.0f;

//Booleana que nos dice si estamos disparando
bool disparando = true;

void Update()
{
    //Si estamos disparando y hemos llegado al tiempo del siguiente disparo
    if (disparando == true && Time.time > nextFire)
    {
        //Sumo el intervalo para indicar cuándo saldrá el siguiente
        nextFire = Time.time + fireRate;

        //Disparo
        print("BANG");
    }
}
```

Otro atributo muy similar es "fixedTime", que mide el tiempo pasado pero desde que se inicio el juego, no solo el objeto con el script.

Detener el tiempo

El atributo timeScale por defecto vale 1, lo que significa que todo lo que depende del paso del tiempo en el juego se moverá a una velocidad normal. Si cambiamos ese parámetro, haremos que todo eso cambie, y si lo ponemos a 0 el juego se quedará en pausa (muy importante reestablecerlo en algún momento, ya que ese cambio permanece incluso al cambiar de escena).

El problema del frameRate

Otra herramienta de gran utilidad es el método Time.deltaTime, que mide el tiempo pasado desde el anterior fotograma

Siempre hay que tener en cuenta que la función Update no funciona igual en todos los ordenadores, algunos lo ejecutarán más rápido y otros más lentos, dependiendo de su potencia, sin embargo, esta variable nos devuelve el tiempo pasado en segundos desde el anterior fotograma.

Siempre que queramos mover/rotar un objeto, debemos multiplicarlo por este parámetro para asegurarnos que se moverá a la misma velocidad en todos los ordenadores. Un ejemplo.

```
transform.Translate(Vector3.forward * Time.deltaTime)
```

También para medir distancias transcurridas:

```
distanceTraveled = distanceTraveled + Time.deltaTime
```

LAS CORRUTINAS

Uno de los problemas es que, al ser programación sincrónica, los bucles impiden que se siga ejecutando el código hasta que mp se han realizado todos los bucles. Es por eso que se ejecutan de golpe (normalmente, vemos el resultado del último ciclo de forma instantánea).

Otro problema se presenta si queremos realizar una acción de forma repetitiva, pero hacerlo en cada frame como hace el método "Update" consumiría demasiados recursos y queremos hacerlo cada 5 segundos.

Para evitar esto, existen las corrutinas mediante el tipo de funciones IEnumerator, que permiten ejecutar código de forma asíncrona. Se declaran de la siguiente manera:

```
IEnumerator Contador() { … }
```

Entre llaves estará el código que se ejecutará, pero hay que tener en cuenta que no e ejecuta en bucle, eso lo tenemos que crear nosotros, mediante un while o un for.

NOTA: en este caso sí se permiten bucles infinitos, ya sea pasando una condición que siempre sea true al "while", o un "for" en el que no se pasa ninguna condición para que se pare el bucle.

Dentro del bucle, se incluye un valor retornado que indica el tiempo que debe pasar hasta el siguiente ciclo. Podemos hacerlo en segundos mediante la clase WaitForSeconds a la que le pasamos entre paréntesis el nº de segundos (en este caso, hace que se ejecute cada segundo):

```
Yield return new WaitForSeconds(1f);
```

Si se pasa el siguiente código, la corrutina se ejecuta en cada fotograma, como el método Update:

```
yield return null;
```

Las corrutinas son invocadas mediante:

```
StartCoroutine("nombreDeLaCorrutina")
```

y detenidas mediante

```
StopCoroutine("nombreDeLaCorrutina");
```

Un ejemplo de corrutina que se lanza al comienzo y ella misma se detiene:

```
//Contador de ciclos
private int cont = 0;

void Start()
{
    //Iniciamos la corrutina
    StartCoroutine("Contador");
}

//Esta corrutina se ejecuta cada segundo
IEnumerator Contador()
{
    //Bucle infinito. También puede ser while(true)
    for ( ; ; )
    {
        //Sumamos uno al contador
        cont++;
        print("Segundos transcurridos: " + cont);
        if (cont == 10)
        {
            StopCoroutine("Contador");
        }

        //Indicamos que cada ciclo se ejecuta pasado 1 segundo
        yield return new WaitForSeconds(1f);

    }
}
```

MÉTODO INVOKE

Una alternativa a las corrutinas, cuando queremos ejecutar una función pasados unos segundos, es el método Invoke.

Se puede llamar en cualquier momento, y se le pasan dos variables: el nombre de la función que queremos ejecutar (string), y el nº de segundos que deben de pasar antes de ejecutarla (float). Ejemplo:

```
void Start()
{
    Invoke("LaunchProjectile", 2.0f);
}
```

Este ejemplo ejecuta la función "LaunchProjectile" a os 2 segundos de cargar el script. Podemos incluso meter el mismo Invoke dentro de la función, y el efecto final será similar a una corrutina.

NOTA: como las Corrutinas no son funciones como tal, si queremos lanzar una corrutina mediante Invoke tendremos que crear una función con el StartCoroutine dentro.

PRACTICA

Algo básico en cualquier juego o aplicación es la creación de contadores. ¿Serías capaz de crear un contador que vaya de 0 a 10, y al llegar a 10 empiece a contar hacia atrás hasta cero, y así de forma indefinida? Y para nota, y que se detenga al cumplir4 vueltas. Trata de hacerlo con corrutinas y con Invoke también.

PARA NOTA: existe una forma de crear algo similar a las corrutinas, mediante lo que se llaman funciones asíncronas, acompañadas del método await. Puedes descubrir más de cómo funcionan y por qué en algunos casos son mejores en este vídeo: https://www.youtube.com/watch?v=WY-mk-ZGAq8

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

INTERACTIVIDAD EN UNITY

Veamos algo básico en cualquier videojuego: cómo detectar las entradas del usuario y actuar en consecuencia

CLASE INPUT

Llegamos a uno de los apartados más importantes de cualquier videojuego o aplicación: la interactividad. Cómo detectar las entradas que nos ofrece el hardware (teclado, ratones, gamepads, pantallas táctiles, etc.) y traducirlas en comportamientos dentro del programa.

Hay dos sistemas principales que usa Unity para detectar la interactividad, y aunque veremos los dos, a partir de ahora utilizaremos el segundo:

1. Clase Input: viene incluido por defecto en Unity y al llamarlo en el método Update permite detectar cualquier entrada, como teclas, botones o, lo que es más habitual, ejes que configuramos en el InputManager de Unity

2. Nuevo "Input System" de unity. Requiere la instalación de un paquete y permite un mayor control sobre la detección de entradas, especialmente para dispositivos móviles, y una mayor compatibilidad entre plataformas.

Unity tiene el método input, que contiene un gran nº de métodos asociados, que como siempre conviene explorar. Aquí veremos los más habituales.

Al introducir una comprobación dentro del método Update, podremos verificar en todo momento si se pulsa la tecla, o qué valor adopta el eje de movimiento. La sintaxis es:

```
Inpupt.MétodoElegido(Tecla/Botón/Eje que queremos comprobar);
```

Veamos algunos ejemplos de los métodos disponibles:

GetKeyDown() / GetKeyUp

Utilizado para detectar el pulsado/despulsado de una tecla, por ejemplo, la barra espaciadora para disparar.

Devuelve true cuando es pulsada, pero no vuelve a devolverlo hasta que esa tecla se suelta y se vuelve a pulsar/despulsar.

Se debe especificar el código de la tecla, utlizando la siguiente sintaxis:

```
KeyCode.NombreDeLaTecla
```

En esta página puedes encontrar todos los nombres de los códigos: https://docs.unity3d.com/ScriptReference/KeyCode.html

NOTA: estos códigos pueden estar representados por una cadena de texto que configuraremos en Unity (ej.- "space") o por el método Keycode (ej.- Keycode.Space)

Un ejemplo:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("Space key was pressed.");
    }

    if (Input.GetKeyUp(KeyCode.Space))
    {
        Debug.Log("Space key was released.");
    }
}
```

Más información:

https://docs.unity3d.com/ScriptReference/Input.GetKeyUp.html

https://docs.unity3d.com/ScriptReference/Input.GetKeyDown.html

CLASE INPUT (II)

GetMouseButtonDown()/ GetMouseButtonUp()

Funciona exactamente igual que el anterior, pero en lugar del código de tecla indicaremos un nº para indicar el botón del ratón (0->botón izquierdo, 1->botón derecho, 2->botón del medio).

Ejemplo:

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
        Debug.Log("Pressed primary button.");

    if (Input.GetMouseButtonDown(1))
        Debug.Log("Pressed secondary button.");

    if (Input.GetMouseButtonDown(2))
        Debug.Log("Pressed middle click.");
}
```

GetButtonDown/GetButtonUp

Tiene la misma función que las anterior funciones para detectar cuándo se pulsa un botón y cuándo se deja de pulsar, pero se asocia a un botón del ratón o del joystick mediante el Input Manager de Unity, que acabamos de ver. Ejemplo usando el Axe llamado "Fire1":

```
public GameObject projectile;
void Update()
{
    if (Input.GetButtonDown("Fire1"))
        Instantiate(projectile, transform.position, transform.rotation);
}
```

Ya estás en condiciones de entender lo que hace este pequeño pedazo de código, ¿verdad?

GetKey() / GetKey() / GetMouseButton()/ GetMouseButton()

Como vemos la sintaxis es igual que los anteriores, pero quitando "Up" y "Down".

En estos casos, no espera a que la tecla o botón vuelva a su estado inicial para volver a lanzarse, sino que se ejecuta de nuevo en cada fotograma.

Esta tecla nos permite crear autodisparos. Un ejemplo:

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
        Debug.Log("Pressed primary button.");

    if (Input.GetMouseButtonDown(1))
        Debug.Log("Pressed secondary button.");

    if (Input.GetMouseButtonDown(2))
        Debug.Log("Pressed middle click.");
}
```

Si ejecutamos este código, veremos que nos manda el mensaje en cada fotograma, a diferencia del anterior que sólo lo mandaba una vez.

GetAxis()

Devuelve el valor del eje de movimiento, normalmente vinculados a las flechas del teclado o al joystick, en un rango de -1 a 1. Ideal para detectar si el objeto se mueve y a qué velocidad (aunque algunos ejes en algunos dispositivos pueden devolver entre 0 y 1, por ejemplo los RT y LT de Xbox).

Compruébalo con esta simple línea en el Update:

```
print(Input.GetAxis(“Vertical”));
```

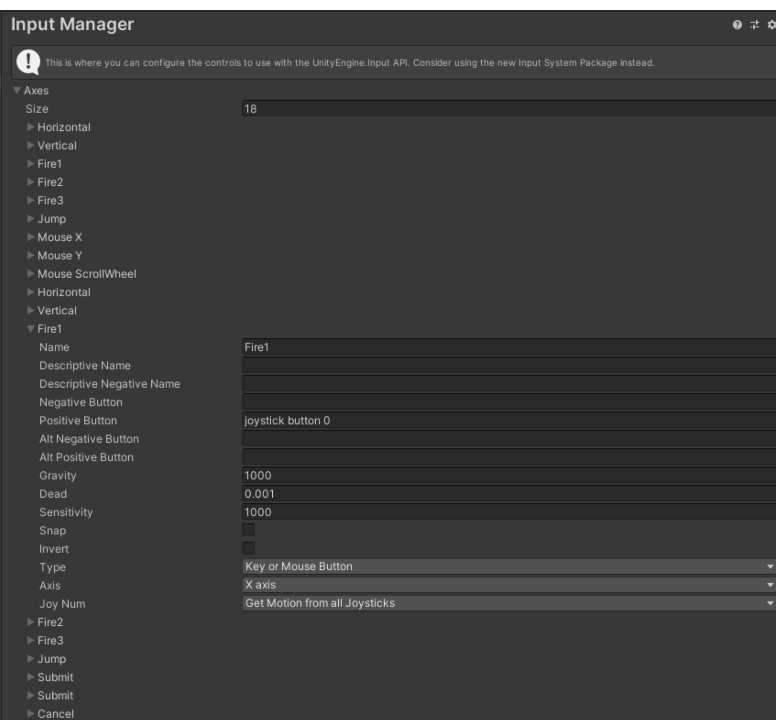
Creando una variable de tipo float para la velocidad ("speed"), y usando el método público Translate, podemos generar fácilmente desplazamientos:

```
float DesplY = Input.GetAxis(“Vertical”) * speed;
transform.Translate(Vector3.up * DesplY * Time.deltaTime);
```

INPUT MANAGER

Antes de ver otros métodos, entendamos esta herramienta de Unity. En ella podemos configurar tanto el nombre con el que se llama como las diferentes teclas o botones que lo lanzan en la ventana de "Edit>Project Settings>Input Manager"

Veremos que ya existen una serie de "funciones" (ó "Axes") configurados, pero podemos quitarlos o añadirlos clonando los existente con el botón derecho del ratón, o aumentando el "Size".



Parámetros:

• Name: importante porque es con el que se llamará usando la clase "Input".

• Negative Button / Positive Button: cuando queremos aignar dos botones a una misma función, pero uno da un valor positivo y el otro negativo (por ejemplo, en el eje "Vertical"). Tiene sus opciones alternativas

• Gravity: tiempo que tarda en volver el valor a 0 cuando el botón o el joystick se ha dejado de usar

• Dead: valor por debajo del cual se entiende que no está siendo usado.

• Speeed: velocidad a la que aumenta el valor hasta alcanzar el valor recibido de entrada

• Snap: cuando se aprietan dos botones con funciones diferentes, si debe volver al estado neutral

• Invert: alterna los valores (por ejemplo, si queremos invertir un jystick para que abajo sea hacia arriba)

IMPORTANTE: si nos fijamos, algunas funciones (Axes) están repetidas, ya que si creamos dos con el mismo nombre, pero una coge el valor del teclado y otra del joystick, al llamarla desde Unity permitirá usar tanto uno como otro a la hora de jugar.

En estas imágenes podemos ver la correspondencia con nuestro GamePad, tanto de botones como de ejes, dependiendo de si es en PC o en MAC:

Fuente: https://answers.unity.com/questions/1350081/xbox-one-controller-mapping-solved.html

XboxOne Controller in Unity (Windows)
(used Version: 2018.2.10f1)

(Source: https://tinyurl.com/yaskln58 and https://answers.unity.com/questions/1350081/xbox-one-controller-mapping-solved.html)

Group	Shortcut	Controller Button Name	Mapping in Unity	Return Value Range
Face Buttons	A	A	joystick button 0	
	B	B	joystick button 1	
	X	X	joystick button 2	
	Y	Y	joystick button 3	
Bumper	LB	Left Bumper	joystick button 4	
	RB	Right Bumper	joystick button 5	
Trigger	LT	Left Trigger	9th Axis	0 to 1
	LT	Left Trigger Shared Axis	3rd Axis	0 to 1
	RT	Right Trigger	10th Axis	0 to 1
	RT	Right Trigger Shared Axis	3rd Axis	0 to -1
	Back	View (Back)	joystick button 6	
	Start	Menu (Start)	joystick button 7	
Stick	LS_h	Left Stick "Horizontal"	X Axis	-1 to 1
	LS_v	Left Stick "Vertical"	Y Axis	1 to -1
	LT_B	Left Stick Button	joystick button 8	
	RS_h	Right Stick "HorizontalTurn"	4th Axis	-1 to 1
	RS_v	Right Stick "VerticalTurn"	5th Axis	1 to -1
	RS_B	Right Stick Button	joystick button 9	
DPAD	DPAD_h	DPAD – Horizontal	6th Axis	-1 (.64) 1
	DPAD_v	DPAD – Vertical	7th Axis	-1 (.64) 1

(Source: https://ritchielozada.com/2016/01/16/part-11-using-an-xbox-one-controller-with-unity-on-windows-10/)

XboxOne Controller in Unity (MacOS)
(used Version: 2018.2.10f1, Driver: **Xbox 360 Controller Driver** Version 0.16.9, FranticRain, Colin Munro)

Group	Shortcut	Controller Button Name	Mapping in Unity	Return Value Range
Face Buttons	A	A	joystick button 16	
	B	B	joystick button 17	
	X	X	joystick button 18	
	Y	Y	joystick button 19	
Bumper	LB	Left Bumper	joystick button 13	
	RB	Right Bumper	joystick button 14	
Trigger	LT	Left Trigger	5th Axis	0 to 1
	LT	Left Trigger Shared Axis	none	
	RT	Right Trigger	6th Axis	0 to 1
	RT	Right Trigger Shared Axis	none	
	Back	View (Back)	joystick button 10	
	Start	Menu (Start)	joystick button 9	
Stick	LS_h	Left Stick "Horizontal"	X Axis	-1 to 1
	LS_v	Left Stick "Vertical"	Y Axis	1 to -1
	LT_B	Left Stick Button	joystick button 11	
	RS_h	Right Stick "HorizontalTurn"	3th Axis	-1 to 1
	RS_v	Right Stick "VerticalTurn"	4th Axis	1 to -1
	RS_B	Right Stick Button	joystick button 12	
DPAD	DPAD_h	DPAD – Horizontal	none	
	DPAD_v	DPAD – Vertical	none	
DPAD	DPAD_up	DPAD – up	joystick button 5	
	DPAD_down	DPAD – down	joystick button 6	
	DPAD_left	DPAD – left	joystick button 7	
	DPAD_right	DPAD – right	joystick button 8	

EJERCICIO

Y estás preparado para comenzar a crear un juego de verdad, en el que coloquemos un "jugador" en la escena y lo movamos mediante nuestros controles.

Instrucciones

Abre el proyecto "Zaxxon" creado anteriormente, crea una nueva escena llamada "Level1" y en ella coloca tu prefab con la nave (mirando hacia el fondo de la escena y en coordenadas 0,0,0). Deberás ajustar la cámara del juego para que vea la nave desde atrás, en 3ª persona.

Añade un script llamado "PlayerController" al prefab de la escena, y añade interactividad para lograr:

1. Que se desplace arriba y abajo al pulsar el joystick izquierdo del Gamepad o los cursores del teclado. Truco: esos ejes ya están creados en el Input Manager, aunque puedes crear los tuyos para que responda también a las teclas AWSD

2. Añade rotación a la nave, usando los ejes que quieras (puede ser el joystick derecho del Gampad).

3. Añade velocidades de desplazamiento y de rotación, si es posible públicas para poder controlarlas externamente.

4. Añade restricción de movimiento para que no pueda salirse de ciertos límites. ¿Se te ocurre cómo podemos hacerlo? PISTA: recuerda que en todo momento sabemos en qué coordenadas está la nave y que los booleanos son muy útiles.

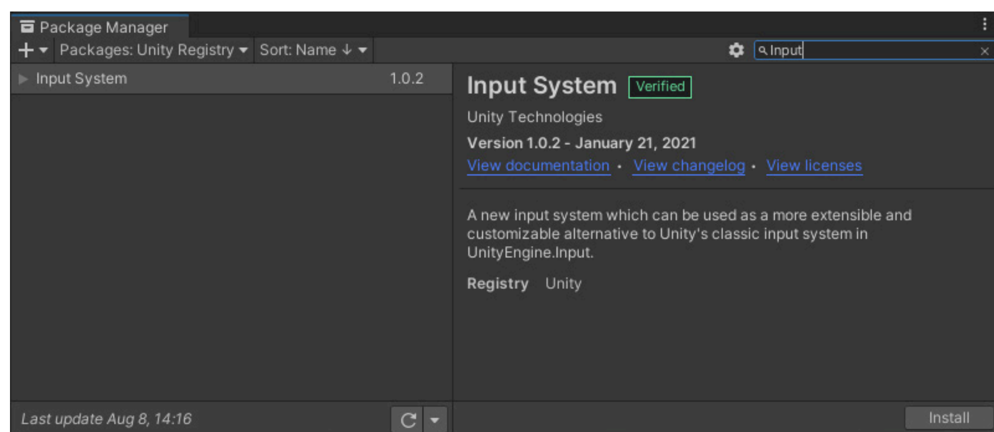
NOTA: acuérdate que este script lo tiene la instancia del Prefab, no el original. Si quisieses que todas las instancias tuvieran ese script, acuérdate del "Overrides".

NUEVO INPUT SYSTEM

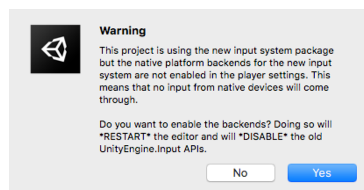
Unity dispone de un nuevo paquete de gestión de entradas que permite un mayor control sobre los dispositivos conectados y sus interacciones.

Instalación

Para usarlo, debemos buscar el paquete "Input System" en nuestra ventana de Window > Package Manager (seleccionando que muestre los paquetes de Unity Registry), e instalarlo.

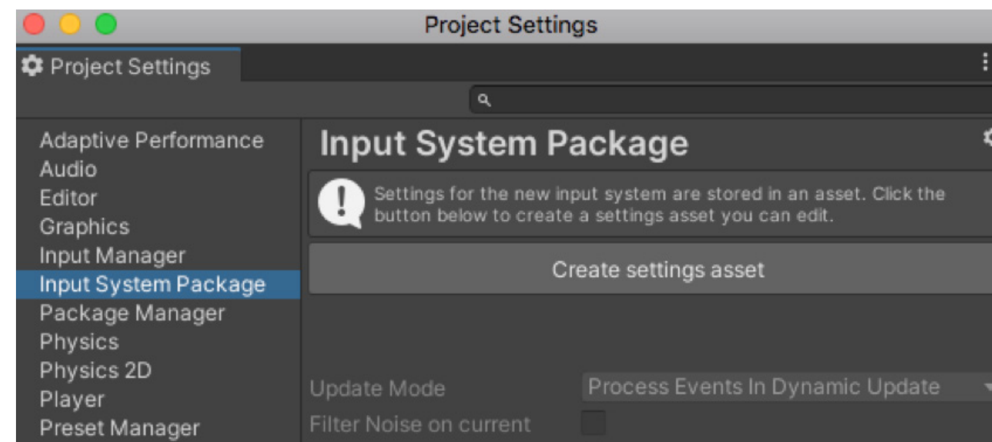


Nos avisará de que necesita activar los "backends" para que el nuevo sistema de entradas reconozca dispositivos nativos del sistema, y que necesitará reiniciar el proyecto. Le decimos que "Sí".



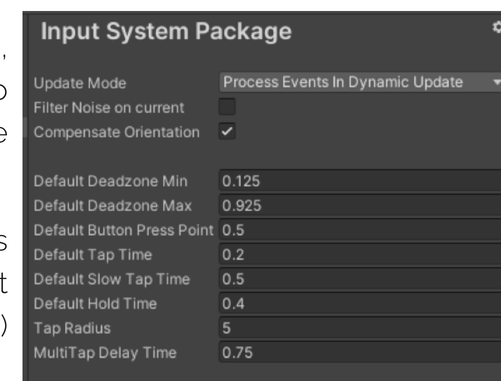
A partir de este momento el Input Manager de Unity, en "Edit>Project Settings", deja de estar disponible y tendremos que configurarlo a través de "Input System Package".

Para que se aplique, debemos crear un archivo de configuración inicial, que se guardará en nuestro proyecto, así que pulsamos en el botón "Create setting asset", que se guardará en la carpeta raíz de Assets con el nombre "InputSystem.inputsettings".



Ahora, en la ventana de Project Settings, podemos configurar el comportamiento general, o incluso indicar qué dispositivos de entrada son aceptados.

Podemos comprobar los dispositivos reconocidos por el sistema mediante el Input Debugger(Windows->Analysis->Input Debug) donde aparecerán todos los dispositivos.



CREANDO ACCIONES DE ENTRADA

Veamos cómo se configuran las entradas de datos en este sistema

Primero crearemos nuestro archivo de gestión de entradas. En el proyecto, pulsaremos con el ratón derecho y daremos a "Create->InputActions". Se creará un archivo al que podemos cambiarle el nombre, y al hacer doble click en él -o pulsando en el botón de "Edit Asset" del inspector- se abrirá el panel de gestión:

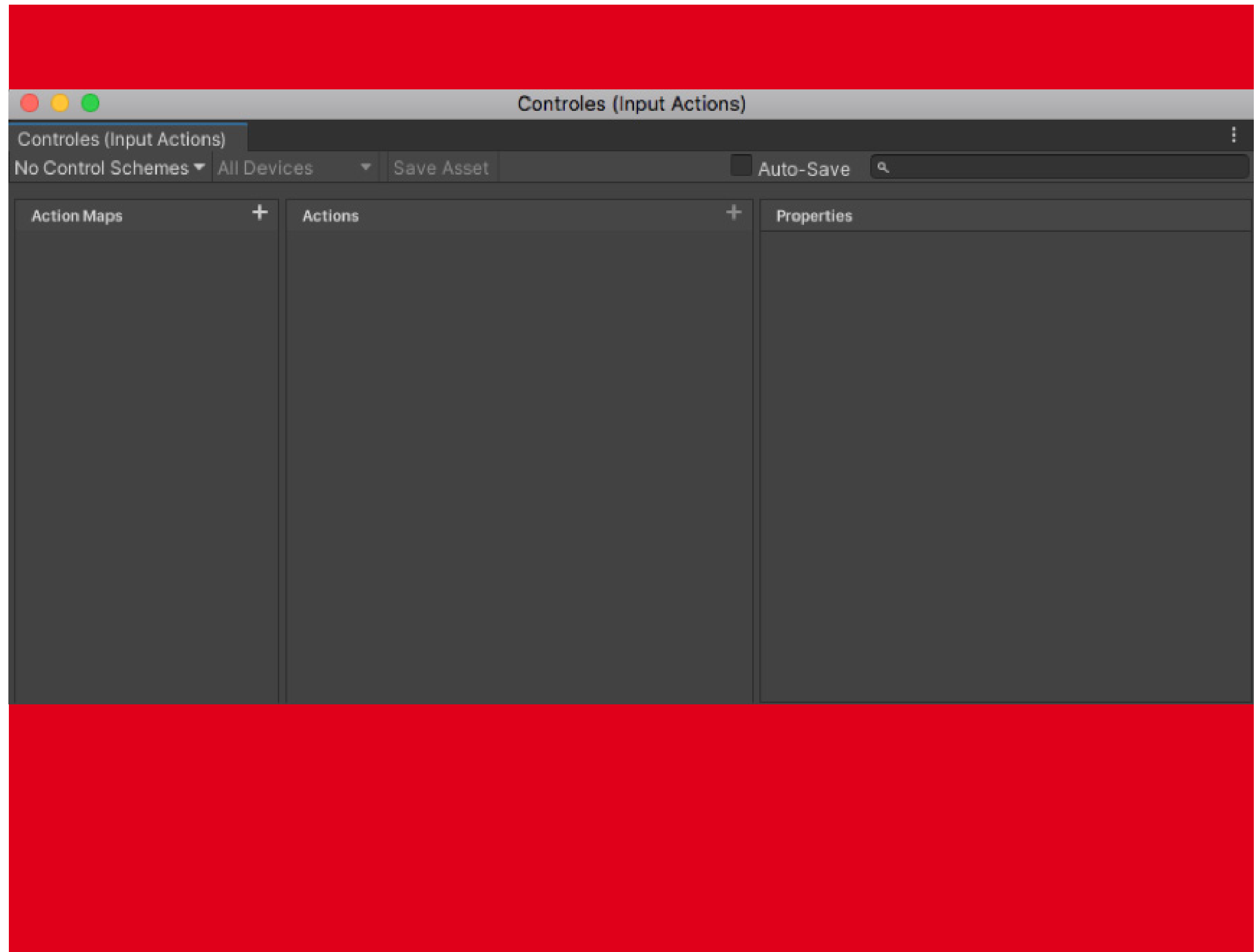
Veamos brevemente sus apartados:

1. **Esquemas de control**: Si vamos a realizar un juego complejo, pensado para múltiples dispositivos de entrada, es bueno crear esquemas de control para luego poder filtrarlos. No es obligatorio.

2. **Action maps**: nos permite organizar las acciones, por ejemplo podemos agrupar las que tienen que ver con el jugador por un lado, y las que tienen que ver con el control de cámara o la User Interface por otro.

3. **Actions**: cada acción que queramos mapear en nuestro juego se corresponderá con un elemento aquí. Un ejemplo "saltar". Una vez creado, le podemos ir añadiendo los llamados "bindings", es decir, mapeos desde un dispositivo de entrada a la acción. De esta forma, el "salto" puede estar unido a un botón del GamePad y a una tecla del teclado, es decir, 2 bindings.

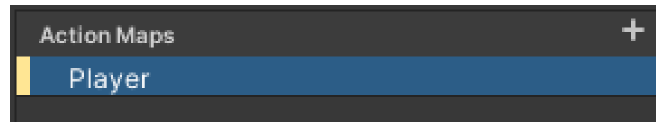
4. **Properties**: cada "binding" tendrá sus propiedades, no solo en lo que se refiere al hardware de entrada sino a su comportamiento.



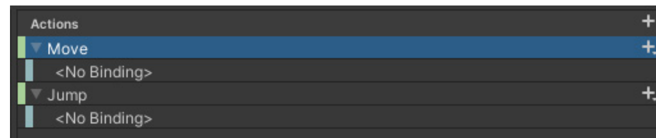
EJEMPLO GUIADO

Veamos un ejemplo guiado para "moverse" y "saltar":

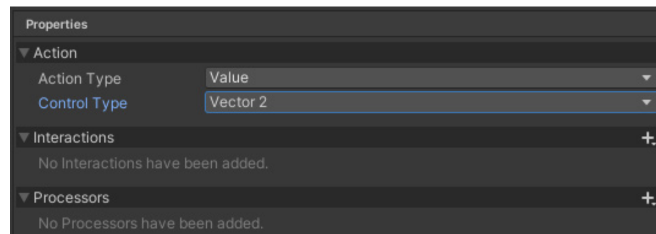
Primero creamos un mapa de acciones que contendrás las acciones de nuestro jugador. Le he llamado "Player"



Creo dos acciones, de momento sin bindings:



Al seleccionar cada acción, en la ventana de propiedades deberé configurar varias cosas:



• "Action Type". El dato más importante, que debemos decir si esa acción se ejecutará como un botón (por ejemplo el salto), o devolverá un valor. En este caso, hay que decirle qué tipo de valor devolverá. Los más habituales:

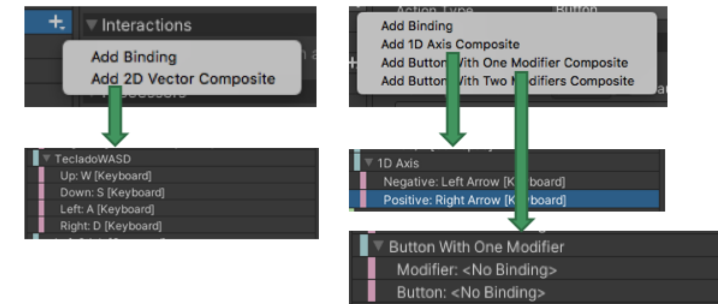
Axis: similar a los ejes del anterior sistema, que van de -1 a 1. Si fuese a coger por ejemplo el valor de un solo eje del joystick, elegiría este valor.

Vector2: nos permite coger en una misma variable dos valores independientes, en este caso vinculados a los 2 ejes del joystick: el eje X y el eje Y.

• "Interactions": permite indicar cuándo se ejecuta esa acción (por defecto al hacer click, pero puede ser al mantener pulsado (Hold), al pulsar o liberar el botón, Tap y Slow Tap, al hacer múltiple click (si selecciona, nos dejará elegir el nº de clicks, y la separación en tiempo). En Project Settings > Input System Package podemos confugrar otros valores pore defecto, como la presión de los botones.

• "Processors": Tenemos algunas opciones útiles, como puede ser invertir los controles, normalizarlos, indicar valores de los ejes que no se tienen en cuenta, etc.

NOTA: Dependiendo del tipo de acción que hayamos elegido, podremos añadir unos tipos u otros de Bindings. En la siguiente imagen verás dos ejemplos:



1. En una acción de tipo "2D Vector", además de un Binding normal que nos proporcionará por ejemplo un joystick de 2 ejes, podemos elgir un "2D Vector Composite", que nos permite elegir qué Input actuará sobre cada eje. En este caso, un sistema WASD.

2. En una acción de tipo botón, además de un Binding normal podemos añadir un "1D Axis Composite", donde no solo tenemos un botón sino dos entradas para indicar cuándo se activa y cuándo se desactiva. Incluso podemos añadir un botón con modificador (por ejemplo, Shift. + Espacio)

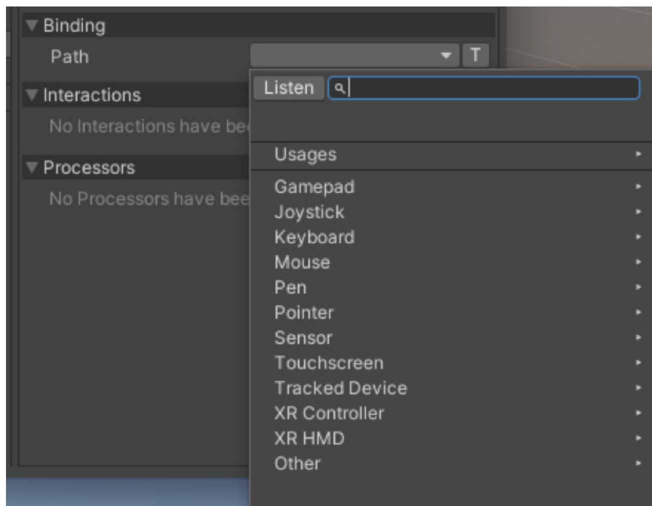
Las posibilidades son ilimitadas y, sobre todo, que podemos añadir a una acción tantas entradas (Bindings) como queramos).

(CONTIÚA)

Continuación

Al seleccionar los "bindings" creados, debemos indicar de qué dispositivo recoge la entrada (path), mediante la lista de dispositivos disponibles, y dentro de cada dispositivo la tecla ó función específica.

TRUCO: Si activamos el botón "Listen" podemos pulsar la tecla que queramos y la asignará

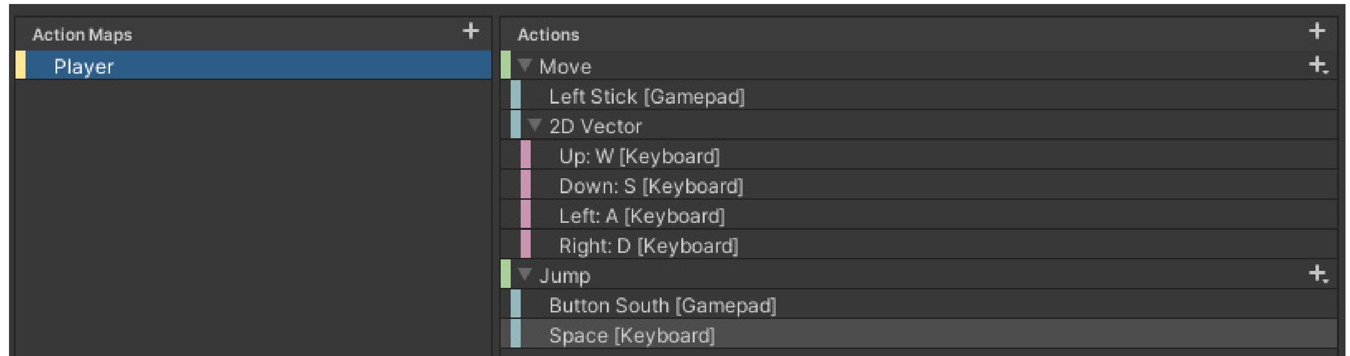


Si es un GamePad, pueden aparecer 2 opciones, por ejemplo "ButtonSouth" para cualquier GamePad, o "buttonA del X-Box" si queremos que sea solo para ese modelo de GamePad.

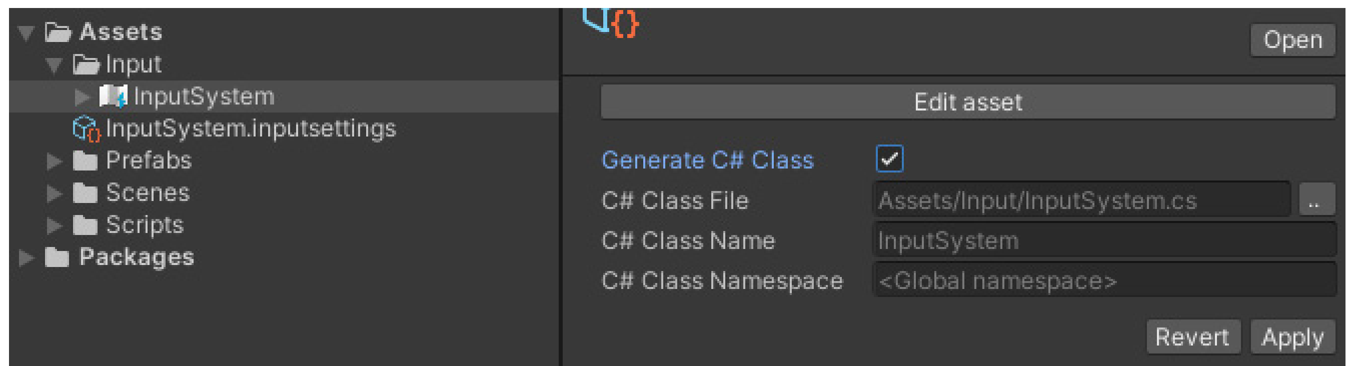
IMPORTANTE: todos los cambios que hagamos deben ser guardados antes de salir, pulsando el botón de "Save Asset" (se puede activar también el autoguardado).

Trasladar a código (1ª Parte)

Vamos a ver cómo llevar nuestra configuración de acciones al juego. Para ello, usaremos el ejemplo anterior, con una acción de salto y otra para moverse. Así ha quedado:



Lo primero que debemos hacer, es seleccionar nuestro Asset creado con las acciones (veremos que si la desplegamos aparecerán los mapas y las acciones), y crear una clase activando la casilla que aparece en la ventana del inspector. En este caso, llamaremos a la clase "InputSystem", aunque podemos llamarla como queramos (recuerda que al ser una clase debe escribirse con mayúscula). En la siguiente imagen se ve el panel de proyecto a la izquierda y el de inspector a la derecha:



Al aplicarlo, se creará un script con el nombre y la ubicación indicados, con todas las acciones indicadas (no tendremos que hacer nada con ese script)

CONTROLANDO LAS ENTRADAS

Ahora podemos crear un script propio para gestionar las entradas como hacíamos con el sistema clásico. Pero debemos seguir estos pasos:

1.- Crear una referencia a nuestro mapa de acciones. Usaremos la clase que hemos creado (en mi caso, "InputSystem"), y por convención, con el mismo nombre, pero en minúscula:

2.- Ahora crearemos una instancia de esa clase, pero lo haremos en el método Awake, para asegurarnos de que se lanza al lanzarse la escena. Quedaría así:

```
InputSystem inputSystem;

private void Awake()
{
    inputSystem = new InputSystem();
}
```

3.- A continuación, en el método heredado de MonoBehaviour OnEnable, activamos los controles, y lo propio al desactivarse. Para ello usaremos los métodos Enable() y Disable() disponibles en el objeto "inputSystem". Esto nos asegura que los controles se activan cuando el script está activado. Este es el ejemplo (lo podemos poner donde queramos del archivo):

```
private void OnEnable()
{
    inputSystem.Enable();
}

private void OnDisable()
{
    inputSystem.Disable();
}
```

TRUCO: verás que VisualStudio es capaz de autocompletar todo el código de la función.

4.- De nuevo en el método Awake, podemos detectar cuándo se ha producido una de estas tres acciones en nuestras entradas, llamados "callbacks":

- **Started**: se detecta la interacción, solo una vez, cuando se inicia.
- **Performed**: se detecta en todo momento, una vez iniciada la interacción y en cada fotograma.
- **Canceled**: se ha cancelado la interacción (por ejemplo, se ha dejado de pulsar el botón)

Basándonos en la estructura de nuestro mapa de acciones debemos hacer la llamada siguiendo esta sintaxis:

```
Instancia.ActionMap.Action.Callback +=
context => MiMetodo();
```

Al ser una acción tipo "callback" debemos asignar el valor devuelto a una variable (en este caso llamada "context", aunque se puede abreviar a "ctx") y ejecutar un método (en este caso "MiMetodo")

Si queremos pasar el parámetro devuelto por la entrada (por ejemplo, un Vector2 o un Eje), debemos usar esta sintaxis:

```
Instancia.ActionMap.Action.
Callback += ctx => myVar = ctx.
ReadValue<tipoDeValor>();
```

En este caso, la variable que ha devuelto el callback se llama "ctx", y usando el método "ReadValue" obtenemos el dato que queremos, en el formato que queremos (por ejemplo, Vector2), que se lo pasamos a la variable previamente declarada "myVar";

Es necesario emplear el mismo sistema para poner a cero el valor cuando la acción ha sido cancelada, o se producirán efectos no deseados.

Otra opción, es ejecutar directamente código usando llaves, ya sea llamar a un método o crear un script entero. En este caso como no queremos "callback" escribimos directamente "_":

```
Instancia.ActionMap.Action.Callback +=
_ => {print("hola");};
```

Resultado final

Parece un poco lioso, pero veremos cómo queda en nuestro ejemplo:

```
InputSystem inputSystem;

//Variable Vector2 que contendrá lo obtenido del joystick
Vector2 moveData;

//Usaos el método Awake en lugar del Start
private void Awake()
{
    inputSystem = new InputSystem();

    //Cada vez que se ejecuta la acción "Move" del mapa de acciones "Payer"
    inputSystem.Player.Move.performed += context => moveData = context.ReadValue<Vector2>();
    //Al volver al joystick a su posición, reseteamos el valor de ese vector
    inputSystem.Player.Move.canceled += _ => moveData = Vector2.zero;

    //Detectamos que se aprieta el botón de saltar, y ejecutamos un método
    inputSystem.Player.Jump.started += _ => { Saltar(); };
}

void Update()
{
    //Mostrar en pantalla en todo momomento el valor del Vector2 creado
    print(moveData);
}

//Método creado para saltar

void Saltar()
{
    print("estoy saltando");
}
```

Lanza el juego y verás cómo en todo momento la variable "moveData" tiene en sus atributos X e Y los valores de movimiento. Y si pulsamos el botón elegido para saltar, se ejecutará el método "Saltar".

IMPORTANTE: Para detectar los dispositivos de entrada, debemos estar con el panel de Juego activa. Si hacemos click en otro panel, dejará de detectar las entradas de datos.

PARA NOTA: A menudo necesitamos registrar el movimiento a partir de un elemento de la pantalla, al que controlamos mediante el ratón o los dedos en una pantalla táctil (muy útil para juegos para móviles). El nuevo Input System incluye un componente llamado On-ScreenStick y On-Screen-Button que permite hacerlo de forma sencilla.

EJERCICIO

Ya estás en disposición de crear un movimiento básico a tu objeto en pantalla usando el nuevo Input System.

Instrucciones

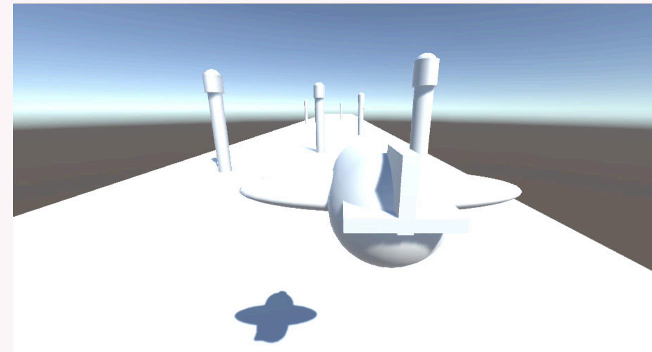
Usa el script del ejercicio anterior y adáptalo a este sistema, para que la nave responda a tus controles.

Crea una acción nueva que sea "Fire" para disparar, y asócialo al botón que quieras. ¿Serías capaz de evitar un doble disparo, o un autodisparo?

Obstáculos

Vamos empezar a dotar de cierta jugabilidad al proyecto:

- Crea obstáculos que se desplacen en todo momento hacia la nave, creando sensación de movimiento. Deberás crear un prefab y usar un instanciador que los añada a la escena a una distancia prudencial (puedes usar un Empty Object a modo de instanciador). Para el desplazamiento, haz que el script del prefab lo haga moverse hacia la nave en una velocidad constante.
- No todos los obstáculos deberían instanciarse en el mismo sitio. ¿Serías capaz de averiguar cómo se generan números aleatorios en Unity, y aplicarlo a unos de los parámetros del Vector de instanciación?
- Deberás destruir aquellos obstáculos que han superado cierta línea del terreno. Para ello puedes usar el método Destroy.GameObject().



NOTA: es importante tener en cuenta cosas para el futuro, por ejemplo, que todos los obstáculos tengan la misma etiqueta (tag), o que la velocidad a la que se mueven no dependa de ellos, sino de la nave. Para ello, crea una variable pública (para poder acceder a ella) en un script que tenga la nave o un EmptyObject que maneje estas cosas (prueba a llamarlo GameManager, verás qué pasa), y accede a esa variable como un atributo de un componente de un GameObject.

MOVIMIENTO SUAVIZADO

MoveTowards / smooth / lerp

Las clases Vector2 y Vector3 incluyen unos métodos que son muy útiles para desplazar un objeto modificando su posición en esa dirección y a una velocidad determinada.

Ejemplos como hemos visto (Vector3.up, Vectoor3.ritght, etc.) son útiles pero son vectores normalizados, de magnitud 1. Unity nos permite crear vectores que "crecen" y "decrecen" a medida que nos acercamos al destino, lo que permite crear movimientos y rotaciones suavizados.

Básicamente son dos:

o **Lerp y Slerp**: Básicamente son iguales, aunque Slerp ofrece un suavizado mayor. En este hilo se ven las diferencias:

https://twitter.com/FreyaHolmer/status/1176137498323501058

o **SmoothDamp**: más usado para posiciones que para direcciones. Cambia gradualmente el valor de un vector hasta el valor indicado y en el tiempo indicado. Es aplicable a Vectores2, Vectores2 y floats (en este caso, mediante Mathf.SmoothDamp)

En ambos casos, los valores necesarios son:

1. Valor actual. Vector de posición o de rotación actual, que iremos actualizando en cada frame.

2. Valor de destino.

3. Velocidad actual. La podemos referenciar directamente, de forma que no es necesario pasarle el valor

4. Velocidad del suavizado: básicamente el tiempo que tardará en llegar al destino.

5. Opcionalmente se pueden dar velocidades máximas y el tiempo desde la última llamada (por defecto, Time.DeltaTime)

La sintaxis sería así, para un desplazamiento:

```
currentPos = Vector3.SmoothDamp(currentPos, targetPos, ref smoothMoveVelocity, MoveVelocity);

transform.Translate(currentPos * Time.deltaTime);
```

En este script, currentPos es un Vector3 que indica dónde estamos, targetPos el Vector3 de desplazamiento (normalmente marcado por las entradas de datos), smoothMoveVelocity un Vector3 creado al inicio, pero sin valor, y MoveVelocity un valor decimal que podemos ir variando para cambiar el suavizado.

Para la rotación es algo más complicado:

```
currentRot = Vector3.SmoothDamp(currentRot, vectorRot, ref smoothRotateVelocity, RotateVelocity);

transform.eulerAngles = currentRot;
```

CurrentRot es un Vector3 que contiene lo que queramos girar, siempre teniendo en cuenta que se mide en grados, y vinculado normalmente a la entrada de datos.

EJERCICIO: aunque es complicado, trata de crear una cámara que siga a nuestra nave en todo momento. Para ello deberás conocer la posición de nuestra nave, y en base a eso mover la cámara, pero solo en los ejes X e Y, y de forma suave.

ENLACES A VÍDEOS

Aquí podrás acceder a los vídeos que explican y/o profundizan sobre los conceptos vistos, por si te son de utilidad.

Movimiento

Mover objetos

https://www.youtube.com/watch?v=3ez-gBMroH4

Instanciando prefabs

https://www.youtube.com/watch?v=nqgsnGjqYCU

Interactividad

Input System clásico

https://www.youtube.com/watch?v=U5YFh_WzDuw

Nuevo Input System Package

https://www.youtube.com/watch?v=QePfKu_Wf9M

Corrutinas

Creando corrutinas

https://www.youtube.com/watch?v=3QaBlvtdpYA