

UF4-5 Introducción a Unity y C#



Contenido

■ Introducción	03
■ Ciclo de vida	05
■ Comentarios	06
■ Variables y tipos de datos	07
■ Operadores	12
■ Arrays	16
■ Estructuras de control	20
■ Funciones	25
■ Vectores	33
■ Más funcionalidades	44

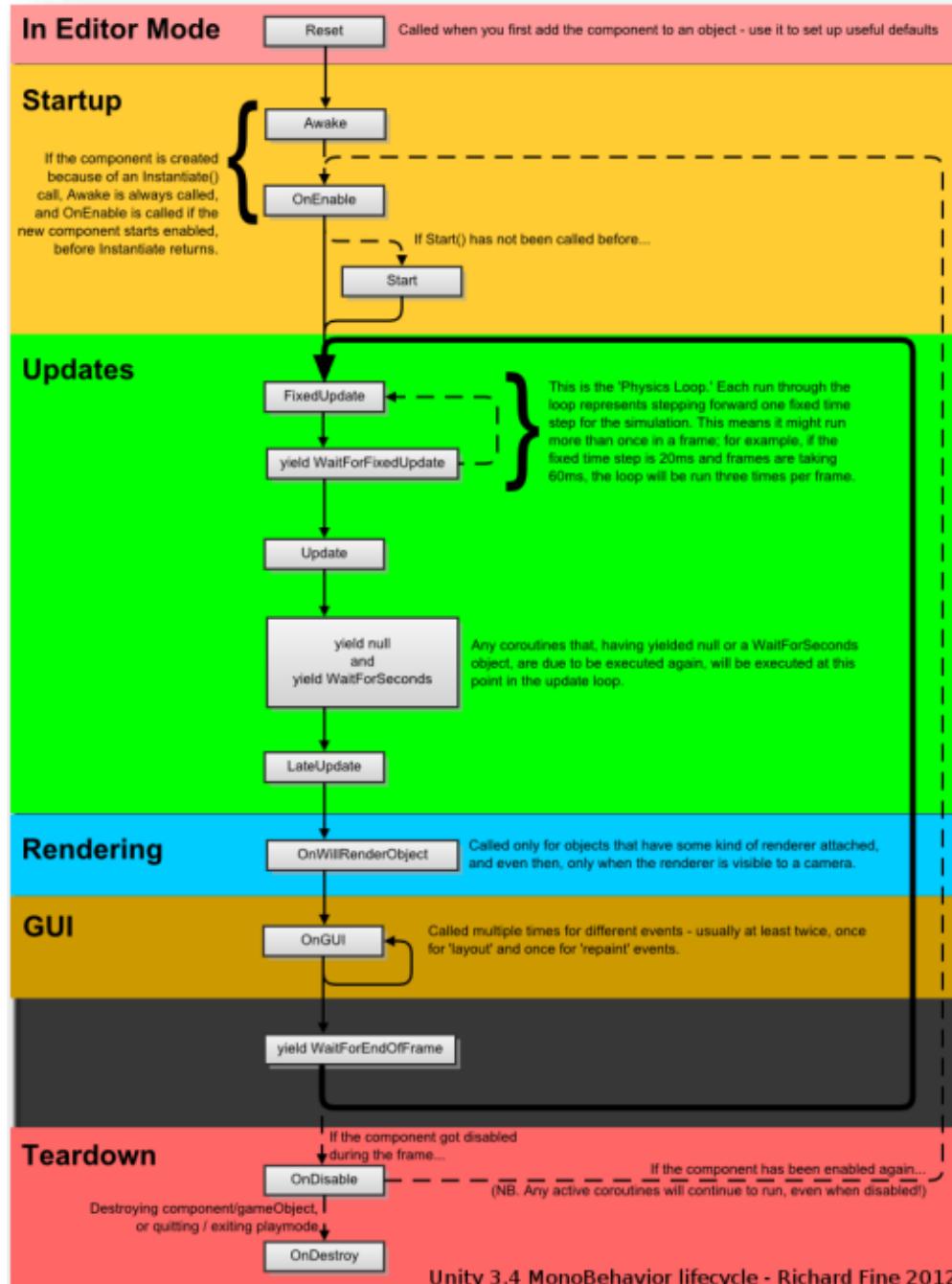
Introducción

■ Monobehaviour

- Encargada de dar lanzar los procesos en su orden
- Nos proporciona acceso a los GameObjets
- Gestión de eventos de los objetos
 - OnEnable
 - OnDisable
 - ...

■ Awake

- Instancia de un script
- El orden de llamada de Awake es aleatorio, no sabemos el orden de cada Script
- FixedUpdate: Para las físicas. Se llama x veces por segundo. Configurable en Edit->ProjectSettings->Time



Introducción

Al iniciar y crear un nuevo C# Script Unity te da los siguiente código:

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

Estas líneas al inicio del script son necesaria para que C# mande llamar directamente al Engine de Unity.

Al iniciar un nuevo documento de C# siempre le pondrá como nombre a la clase "NewBehaviourScript", este es el default que Unity le asigna como nombre, en este caso el nombre de **Script** y el de la **Clase**, **tienen que ser el mismo nombre**, debemos de crear una clase y dentro de esta deberá de ir todo el contenido que desarrollaremos para programar.

```
public class AI_C : MonoBehaviour {  
    // Use this for initialization  
    void Start () {
```

→ La forma en que se declarar función de Start (al iniciar el juego se ejecutara).

```
}
```

En C# TODA acción que se desee ejecutar debe de ir dentro de una function (void) forzadamente (Mejor performance).

```
    // Update is called once per frame
```

```
    void Update () {
```

→ La forma en la que se declarar función de Update (Constantemente se ejecuta).

```
}
```

```
}
```

Ciclo de vida

El método Awake

- ▶ Parte del bloque de inicialización
- ▶ Ejecutado cuando se crea el objeto
 - ▷ También cuando se añade el componente al objeto
 - ▷ Similar al constructor de la clase
- ▶ Uso más frecuente
 - ▷ Inicializar valores del componente
 - ▷ Obtener referencias del propio objeto

El método Start

- ▶ Se ejecuta una única vez
 - ▷ El primer frame tras activar el componente
- ▶ Usos más frecuentes
 - ▷ Inicializar valores
 - ▷ Obtener referencias
 - ▷ Obtener recursos de otros objetos
 - ▷ Inicializados en Awake

El método Update

- ▶ Se ejecuta en cada frame
 - ▷ Frame → Actualización completa
- ▶ Marca los FPS del juego
- ▶ Principal método de actualización
 - ▷ Lógica de juego
 - ▷ Métodos de entrada
 - ▷ Animaciones
 - ▷ Posiciones

El método FixedUpdate

- ▶ Se ejecuta a una frecuencia fija
 - ▷ Modificable en las preferencias
- ▶ Se ejecuta antes de los cálculos de física
- ▶ Si se aplican fuerzas debe hacerse aquí
 - ▷ Puede ejecutarse varias veces por frame!
 - ▷ Cuidado con aplicar fuerzas múltiples veces

El método LateUpdate

- ▶ Se ejecuta a la misma frecuencia que Update
- ▶ Pero siempre después de todos los Update
- ▶ Ideal para llevar a cabo acciones que requieren que todos los objetos de la escena se hayan actualizado

El método OnEnable

- ▶ Tras la creación de un objeto, debe activarse.
- ▶ Se ejecuta siempre que se active el objeto o componente
 - ▷ Por tanto, puede ejecutarse varias veces
- ▶ Uso más común:
 - ▷ Obtener referencias a componentes
 - ▷ Asignar eventos y delegados

El método OnDisable

- ▶ Se ejecuta cada vez que se desactiva el objeto o componente
- ▶ Útil para eliminar eventos y delegados

El método OnGUI

- ▶ Usa las clases *GUI* y *GUILayout*
 - ▷ *GUI.Color*: Cambia el color del GUI
 - ▷ *GUILayout.Label*: Muestra una etiqueta
 - ▷ *GUILayout.Button*: Muestra un botón
 - ▷ Y reacciona a su pulsación
 - ▷ *GUILayout.[Begin|End][Horizontal|Vertical]()*
- ▶ Ejemplo de herramienta sencilla
 - ▷ GUI para especificar dónde mover un objeto

Comentarios

- Comentar el código permite añadir notas a los scripts.
- Estas notas pueden ayudar a documentar el diseño y el funcionamiento de la aplicación.
- Se puede colocar un comentario en cualquier parte del código.

Comentarios de una línea

Los comentarios de una línea empiezan con dos barras inclinadas y afecta al resto de la línea:

```
// Use this for initialization
void Start () {

}
```

Comentarios de varias líneas

Los comentarios de varias líneas empiezan con una barra inclinada seguida de un asterisco y su efecto permanece hasta que encuentre un asterisco y una barra inclinada. Los comentarios regulares pueden extenderse por varias líneas:

```
/* esto es un comentario
que contiene varias
líneas de texto
*/
```

VARIABLES

- Es un espacio reservado en la memoria del ordenador,
- Este espacio está asociado a un nombre simbólico también conocido como identificador.
- En este espacio podemos almacenar un valor, que puede ser modificado en cualquier momento durante la ejecución del programa.

Declarar una variable:

```
string mensaje;  
string mensaje2 = "hola mundo";
```

- Se coloca el tipo de dato que esta variable almacenará.
- Se le da un identificador, es decir, el nombre de la variable,
- y por ultimo y opcional se le asigna un valor, en la misma sentencia o más tarde.

```
mensaje = "has colisionado con una roca";  
Debug.Log (mensaje);
```

Variables

```
public int numeroEntero;           // 1 2 3 4 5 6 7 8 9
public float numerosDecimales = 1f; // 1.1f 1.2f 1.13f 23.56f
public string textos;             // "Siempre entre comillas"
public bool puertaAbierta = true;  // tipo booleano
public bool puertaCerrada = false; // solo puede tomar dos valores: verdadero o falso

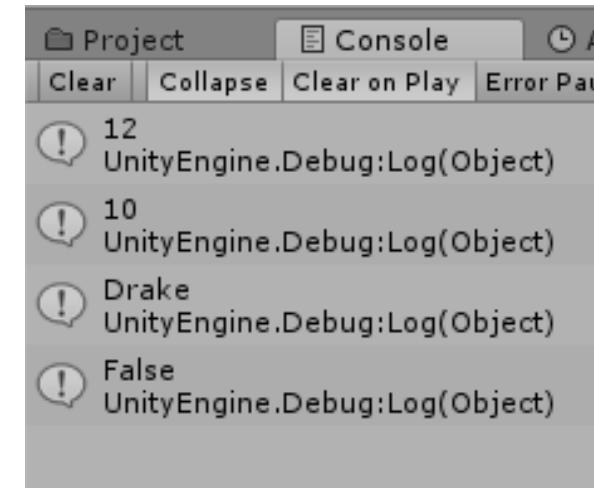
// variables mas usadas en Unity

public GameObject cubo;
public Transform miTransfrom;
public MeshFilter meshFilter;
public BoxCollider boxColliderRojo;
public MeshRenderer meshRender;
```

Tipo String (cadena de texto)

- Una cadena es una secuencia de caracteres encerrados entre comillas dobles.
- Cada carácter tiene una posición en la cadena, empezando por 0.
- Sobre una variable de tipo string se podrán realizar diferentes acciones:

```
string nombre = "Nathan Drake";  
  
void Start () {  
  
    Debug.Log(nombre.Length);  
    Debug.Log(nombre.IndexOf('k'));  
    Debug.Log(nombre.Substring(7,5));  
    Debug.Log(nombre.Equals("Geralt de Rivia"));  
}
```



Tipo int (número entero)

```
int a = 11;
int b = 2;

void Start () {
    Debug.Log (a + b + " esto es una suma");
    Debug.Log (a - b + " esto es una resta");
    Debug.Log (a / b + " esto es una división");
    Debug.Log (a * b + " esto es una multiplicación");
    Debug.Log (a % b + " esto es el módulo");
    Debug.Log (++a + "esto es un incremento en 1");
    Debug.Log (--b + "estos es un decremento en 1");
}
```

Concatenaciones

La *concatenación* es el proceso de anexar una cadena al final de otra cadena. Al concatenar literales o constantes de cadena mediante el operador `+`, el compilador crea una única cadena:

```
string mensaje1 = "Buenos días ";
string mensaje2 = "a tod@s";

void Start() {
    Debug.Log(mensaje1 + mensaje2);
}
```

Operadores de asignación

Los operadores de asignación almacenan un valor en el objeto designado por el operando izquierdo. Hay dos clases de operaciones de asignación: **asignación simple**, en la que el valor del segundo operando se almacena en el objeto especificado por el primer operando, y **asignación compuesta**, en la que se realiza una operación aritmética antes de almacenar el resultado.

Operador	Significado
=	Almacena el valor del segundo operando en el objeto especificado por el primer operando (asignación simple).
*=	Multiplica el valor del primer operando por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando.
/=	Divide el valor del primer operando por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando.
%=	Toma el módulo del primer operando especificado por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando.
+ =	Suma el valor del segundo operando al valor del primer operando; almacena el resultado en el objeto especificado por el primer operando.
--	Resta el valor del segundo operando del valor del primer operando; almacena el resultado en el objeto especificado por el primer operando.

Operadores relacionales

Los operadores relacionales son símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa.

Operador	nombre	ejemplo	significado
<	menor que	$a < b$	a es menor que b
>	mayor que	$a > b$	a es mayor que b
==	igual a	$a == b$	a es igual a b
!=	no igual a	$a != b$	a no es igual a b
<=	menor que o igual a	$a <= b$	a es menor que o igual a b
>=	mayor que o igual a	$a >= b$	a es mayor que o igual a b

Operadores lógicos

Permiten realizar las operaciones lógicas:

- “and” → `&&`
- “or” → `||`
- “not” → `!`

a	b	!a	a && b	a b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

para concatenar o añadir condiciones, tanto en los if como en cualquiera de las instrucciones que lleven una condición.

Ejemplo:

Aquí preguntamos

“si a es igual a 1 y b es igual a 3”.

```
int a = 1;  
int b = 3;  
  
if (a == 1 && b == 3)  
{  
    ...  
}
```

Operadores lógicos

Además podemos **combinar** estos operadores y añadir todas las condiciones que queramos, lo cual lo hará más complejo.

Veamos un **ejemplo** más complejo.

```
int a = 1;
int b = 3;
int c = 0;
if (a == 1 && b == 3 || c == 1)
{
    Console.WriteLine("a vale 1 y b vale 3 o c vale 1");
    Console.ReadKey();
}
```

```
graph TD
    Root((Sí)) --> A1((Sí))
    Root --> A2((No))
    A1 --> A3((Sí))
    A1 --> A4((Sí))
    A2 --> A5((No))
```

Tipos de datos complejos: arrays

Un array es la agrupación de varios valores de un mismo tipo de dato en una sola variable.

1. Tipo de dato seguido de corchetes []
2. Nombre de la variable
3. Operador de asignación
4. Palabra “new” para crear el array
5. Tipo de dato
6. Número de elementos que contendrá entre corchetes (tamaño del array)

```
string[] letras = new string[5]{"a","b","c","d","e"};  
  
public int[] numeros = new int[6]{0,1,2,3,4,5};  
  
void Start () {  
  
    Debug.Log(letras[0]);  
    Debug.Log(letras[1]);  
    Debug.Log(letras[2]);  
    Debug.Log(letras[3]);  
    Debug.Log(letras[4]);  
    letras[0] = "z";  
    Debug.Log(letras[0]);  
}
```

Tipos de datos complejos: arrays

- En programación, siempre la primera posición de un array se indica con un índice 0.
- Por eso el primer valor, hay que asignarlo al índice 0 del array.
- Por ello hay que tener en cuenta que el último índice en un array es su longitud menos 1.

```
int tamanoArray = letras.Length;  
string ultimaLetra = letras [tamanoArray - 1];
```

- Para acceder a los datos se hace igual que para asignar valores, hay que poner el nombre del array con el índice al que queremos acceder entre corchetes.

```
letras [0] = "z";
```

- También podemos darle los valores al array en el momento de crearlo. Es lo ideal para cuando conocemos qué valores tendrá, y se hace colocando los valores entre llaves {}.

Tipos de datos complejos: arrays

Para [recorrer un array](#), ya sea para asignarle los valores o como para obtenerlos, se utiliza el bucle for:

```
string[] letras = new string[5]{"a","b","c","d","e",};  
  
void Start () {  
  
    for (int i = 0; i < letras.Length; i++) {  
        Debug.Log (letras [i]);  
    }  
  
}
```

Tipos de datos complejos: enumerados

Un enumerado es un conjunto de constantes lógicamente relacionadas. Estas constantes se agrupan en un tipo con un identificador.

```
public enum Brujula {Norte, Sur, Este, Oeste};  
  
Brujula miDireccion;  
  
void Start () {  
  
    miDireccion = Brujula.Norte;  
    Debug.Log("Mi direccion actual es :" + miDireccion);  
  
}
```

Estructuras de control

La estructura "if-else"

```
if (condición) {  
    sentencias  
} [else {  
    sentencias  
}]
```

```
public int velocidadCoche = 0;  
  
void Start () {  
  
}  
  
void Update () {  
  
    if(velocidadCoche > 40){  
        Debug.Log ("Vas muy rápido");  
    }else if(velocidadCoche < 38){  
        Debug.Log ("Vas muy lento");  
    }else{  
        Debug.Log (velocidadCoche);  
    }  
    velocidadCoche++;  
}
```

Estructuras de control

La estructura "switch-case"

```
switch (x) {  
    case valor1:  
        sentencias  
        break;  
    case valor2:  
        sentencias  
        break;  
    ...  
    default:  
        sentencias  
}
```

```
public int vidas = 1;  
  
void Start () {  
    switch (vidas) {  
    case 0:  
        Debug.Log("No te quedan vidas");  
        break;  
    case 1:  
        Debug.Log("Te queda 1 vida");  
        break;  
    case 2:  
        Debug.Log("Te quedan 2 vidas");  
        break;  
    default:  
        Debug.Log("puedes conseguir más...");  
        break;  
    }  
}
```

Estructuras de control

Repite un bloque de código mientras se cumpla una o varias condiciones.

La estructura "while"

```
while (condición) {  
    sentencias  
    ...  
}
```

```
int vueltas = 5;  
  
void Start () {  
  
    while (vueltas > 0) {  
        Debug.Log("Te quedan: " + vueltas + " vueltas");  
        vueltas--;  
    }  
    Debug.Log("Ya no te quedan vueltas");  
}
```

Estructuras de control

La estructura "for"

```
for (contador = valor inicial; condición; expresión de incremento) {  
    sentencias;  
    ...  
}
```

```
void Start () {  
  
    for (int i = 1; i <= 5 ; i++) {  
        Debug.Log(i);  
    }  
}
```

Estructuras de control

EJEMPLO: IF

```
int variable_A = 1;  
int variable_B = 2;
```

Ejemplo de Condiciones:

```
void Start () {  
    if (variable_A > variable_B){  
        Debug.Log ("A es mayor que B");  
    }else if (variable_A == variable_B ){  
        Debug.Log ("A y B son iguales");  
    }else {  
        Debug.Log ("B es mayor que A");  
    }  
}
```

EJEMPLO: WHILE

```
int Valor = 0;  
  
void Start () {  
    // Si es igual a 10 o mayor se cumpla el loop.  
    while (Valor <= 10) {  
        // se hace aquí la acción que queremos repetir.  
        print("Cantidad: " + Valor);  
        // Agregamos el valor de 1, cada q se hace un loop.  
        Valor++;  
    }  
}
```



EJEMPLO: FOR

```
int Valor = 0;  
  
// Se define la variable, su límite y el incremento.  
void Start () {  
    for (Valor = 0; Valor <= 10; Valor++) {  
        // Se hace aquí la acción que queremos repetir.  
        print ("Cantidad: " + Valor);  
    }  
}
```

EJEMPLO: FOR IN

```
String Lista;  
  
function Start () {  
    foreach (Lista in nombres){  
        // Imprimir información  
        print (Lista);  
    }  
}
```

Funciones

Un problema complejo se puede dividir en pequeños subproblemas más sencillos. Estos subproblemas se conocen como funciones. Una función realiza una acción u operación específica.

Son trozos de código que permiten modularizar el programa, permitiendo dividir funcionalidades para poder usarlos en otros programas.

Todas las instrucciones deben estar incluidas en un procedimiento o función, a las que llamaremos mediante su identificador.

La utilización de funciones en programación ayuda mucho. Es altamente recomendable hacer uso de ellas por estos dos motivos principales:

- Ahorramos líneas de código.
- Facilitan mucho el entendimiento del programa y del código.

Funciones. Sintáxis

Las declaraciones de las funciones generalmente son especificadas por:

- **Un nombre único.** Nombre de la función con el que se identifica y se distingue de otras.
- **Un tipo de dato de retorno.** Tipo de dato del valor que la función devolverá al terminar su ejecución.
- **Una lista de parámetros.** Especificación del conjunto de argumentos (pueden ser cero uno o más) que la función debe recibir para realizar su tarea.

```
tipo_retorno nombreFunc ( parámetros) {  
    <sentencias>  
    [ return dato]  
}
```

Funciones sin retorno

Realizan una tarea, pero no devuelven ningún dato:

- El tipo de dato de retorno es "void".
- No tiene instrucción "return".

```
void Start () {  
    saludar ();  
}  
  
void saludar(){  
    Debug.Log ("hola");  
}
```

Llamada a la función para ejecutarla

Funciones sin retorno

```
void Start () {
    Saludo();
}

void Saludo() {
    Debug.Log("Hola");
    Nombre();
}

void Nombre() {
    Debug.Log("Tony");
    Debug.Log("Stark");
}
```

Funciones con retorno

Realizan una tarea y devuelven algún dato:

- El tipo de dato de retorno se especifica en la firma de la función.
- Necesitan la instrucción "return". Con un tipo de dato que concuerde con lo especificado en la firma

```
void Start () {  
    string nombre = obtenerNombre();  
    Debug.Log(nombre);  
}
```

Llamada a la función para ejecutarla.
Se sustituye por el valor que devuelva

```
string obtenerNombre() {  
    return "Tony Stark";  
}
```

El tipo de dato que se declara en la firma y lo que realmente se devuelve deben ser del mismo tipo

Funciones con parámetros

```
void Start () {  
    saludar ("Tony Stark");  
    string nombre = "Capitán América";  
    saludar (nombre);  
}  
  
void saludar (string nombre) {  
    Debug.Log ("hola " + nombre);  
}
```

Llamada a la función para ejecutarla pasándole el número y tipo de argumentos que necesita

Esta función necesita recibir un valor de tipo string y lo guarda en la variable que actúa como parámetro

Funciones con parámetros y return

```
void Start () {  
    int numero1 = 5;  
    int numero2 = 3;  
  
    int resultado = sumar (numero1, numero1);  
}  
  
int sumar (int num1, int num2) {  
    return num1 + num2;  
}
```

Scripts. Funciones principales

EVENTOS ESCENAS:

```
void Update ()  
void LateUpdate ()  
void FixedUpdate ()  
void Awake()  
void Start ()  
void Reset ()
```

- Es llamado cada frame.
- Es llamado cada frame. Si el "Behaviour" esta "activado" - [Usar siempre en camaras].
- Es llamado cada Fixed frame (Físicas).
- Es llamado cuando una instancia de script se está cargado (antes).
- Es llamado al inicio de la escena cuando esta se haya cargado (después).
- Reinicia los valores por default.



EVENTOS PARA MOUSE:

```
void OnMouseEnter ()  
void OnMouseOver ()  
void OnMouseExit ()  
void OnMouseDown ()  
void OnMouseUp ()  
void OnMouseDrag ()
```

- Es llamado cuando el mouse Entra en el GUIElement o un Collider.
- Es llamado cada frame cuando el mouse esta Encima del GUIElement o Collider.
- Es llamado cuando el mouse ya no está más sobre GUIElement o Collider.
- Es llamado cuando el mouse Presiono botón sobre un GUIElement o Collider.
- Es llamado cuando el mouse Soltó el botón sobre un GUIElement o Collider.
- Es llamado cuando el mouse Presiono botón sobre un GUIElement o Collider y aun continua presio.

EVENTOS TRIGGERS:

```
void OnTriggerEnter ()  
void OnTriggerExit()  
void OnTriggerStay ()
```

- Es llamado cuando el Collider u otros entrar en el Trigger.
- Es llamado cuando el Collider u otros han parado de tocar en el Trigger.
- Es llamado 1 vez por Frame por cada Collider u otros que están tocando al Trigger.

EVENTOS COLISIONADORES:

```
void OnCollisionEnter ()  
void OnCollisionExit()  
void OnCollisionStay()
```

- Es llamado cuando este Collider/rigidbody a comenzado a tocar otro rigidbody/Collider.
- Es llamado cuando este Collider/rigidbody ha dejado de tocar a otro Collider/rigidbody.
- Es llamado 1 vez por frame cada que este Collider/rigidbody está tocando otro Collider/rigidbody.

EVENTOS VISIBLES:

```
void OnBecameVisible ()  
void OnBecameInvisible()
```

- Es llamado cuando el Render se ha cambiado a Visible por cualquier cámara.
- Es llamado cuando el Render se ha cambiado a Invisible por cualquier cámara.

¿Qué es Vector3?

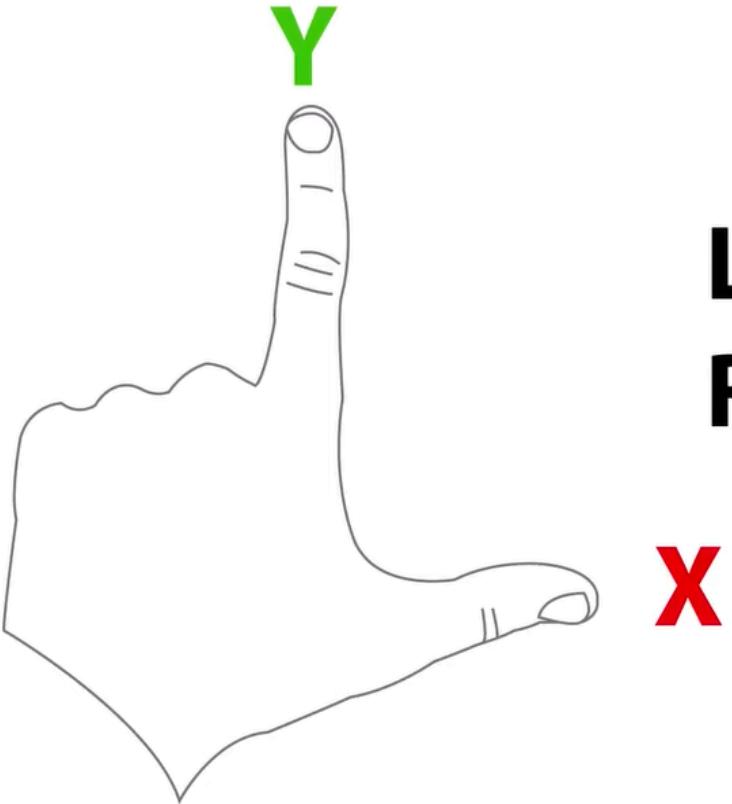
- ▶ Estructura con 3 atributos principales
 - ▷ x, y, z
- ▶ Métodos de manipulación
 - ▷ Distance, Angle, Dot, Lerp...
- ▶ Campos estáticos auxiliares
 - ▷ zero, one, up, down, left, right, forward, back...
- ▶ Movimiento suave e independiente de fps:
 - ▷ Distancia = velocidad * tiempo
 - ▷ Distancia = velocidad * Time.deltaTime

¿Asignar X, Y o Z directamente?

- ▶ Lo siguiente no es posible

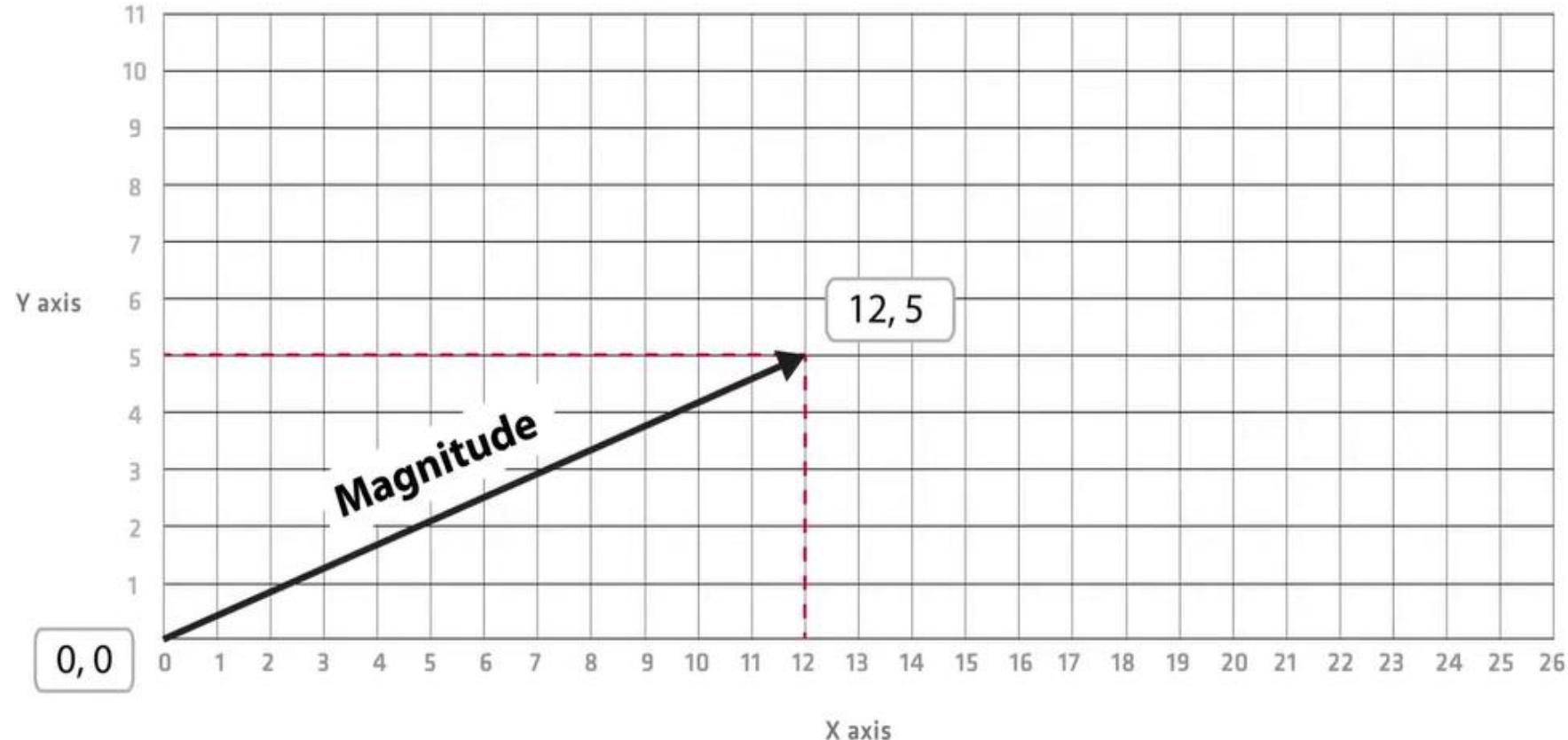
```
transform.position.y = 2f;
```
- ▶ *position* es una propiedad
 - ▷ Por tanto devuelve una copia.
 - ▷ Si hicéramos lo anterior, se modificaría el valor de la copia.
 - ▷ ¡No afectaría al transform!

Transform: vectores 2D

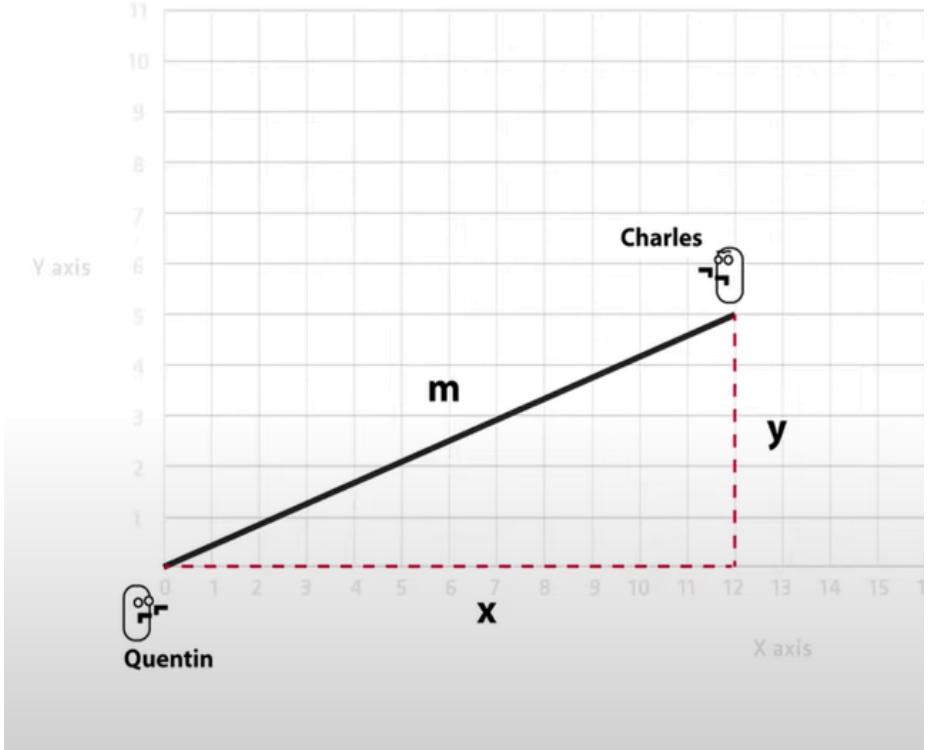


**Left Hand
Rule Coordinates**

Transform: vectores 2D



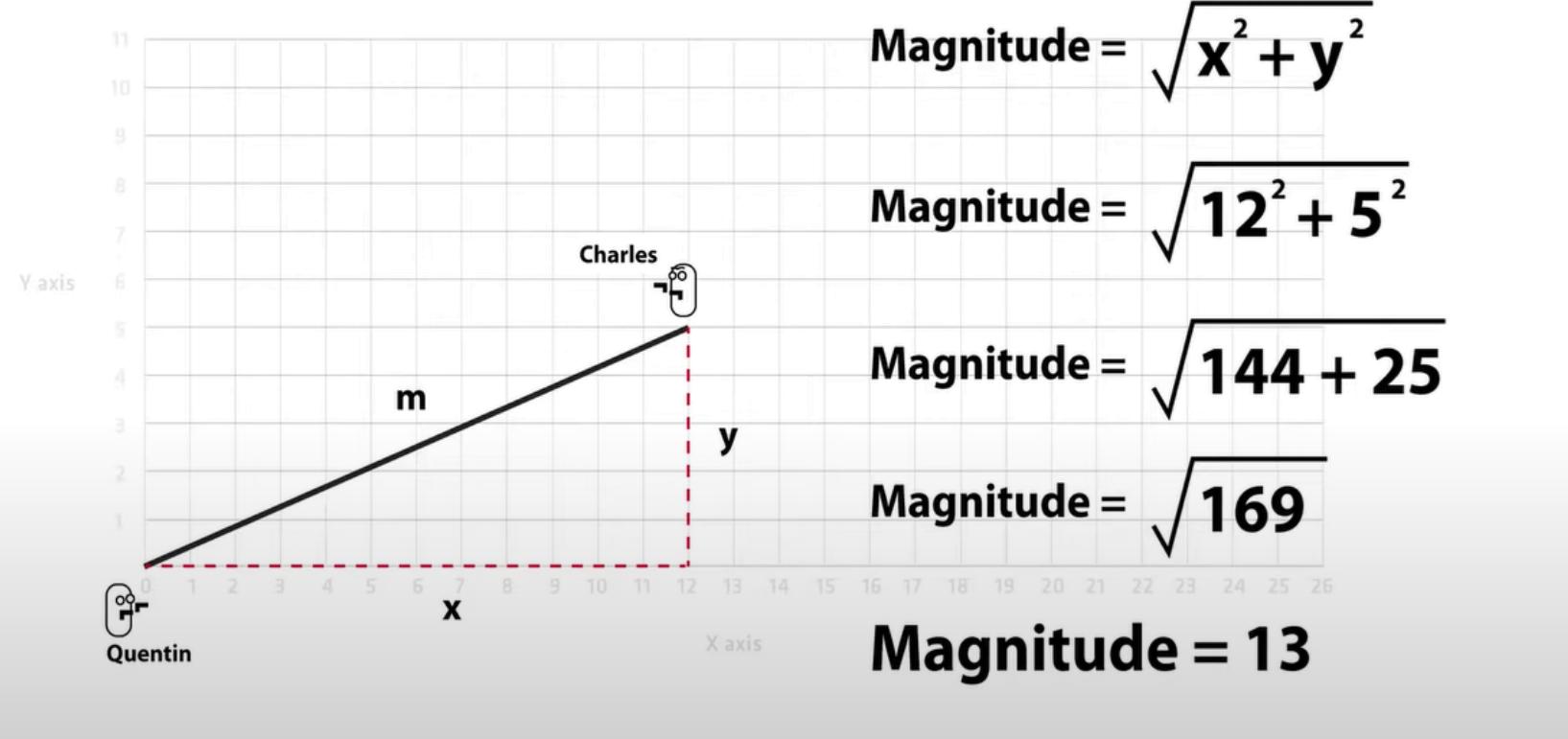
Transform: vectores 2D



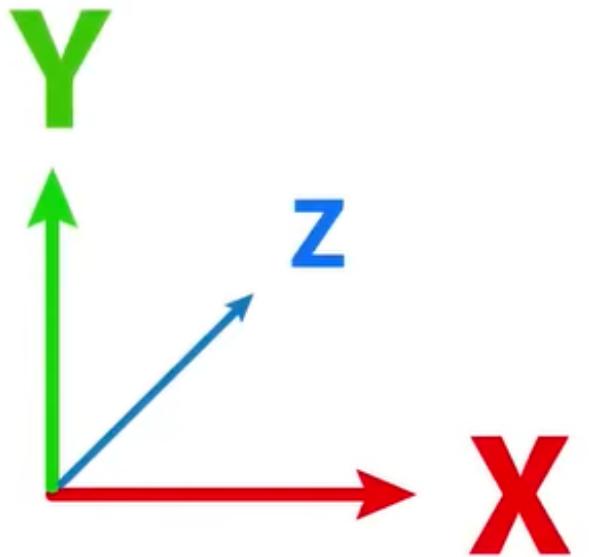
Si las armas tienen un alcance de 12, ¿se matarán el uno al otro?

Transform: vectores 2D

$$x^2 + y^2 = m^2$$

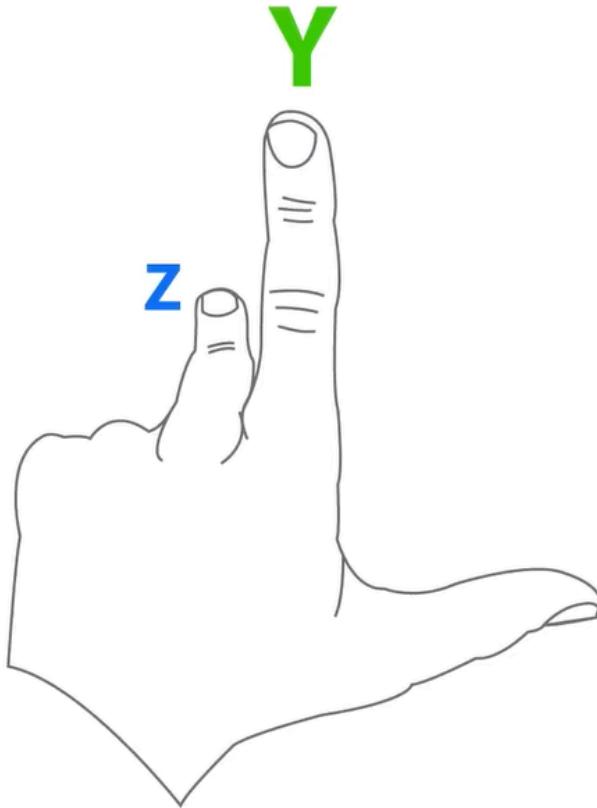


Transform: vectores



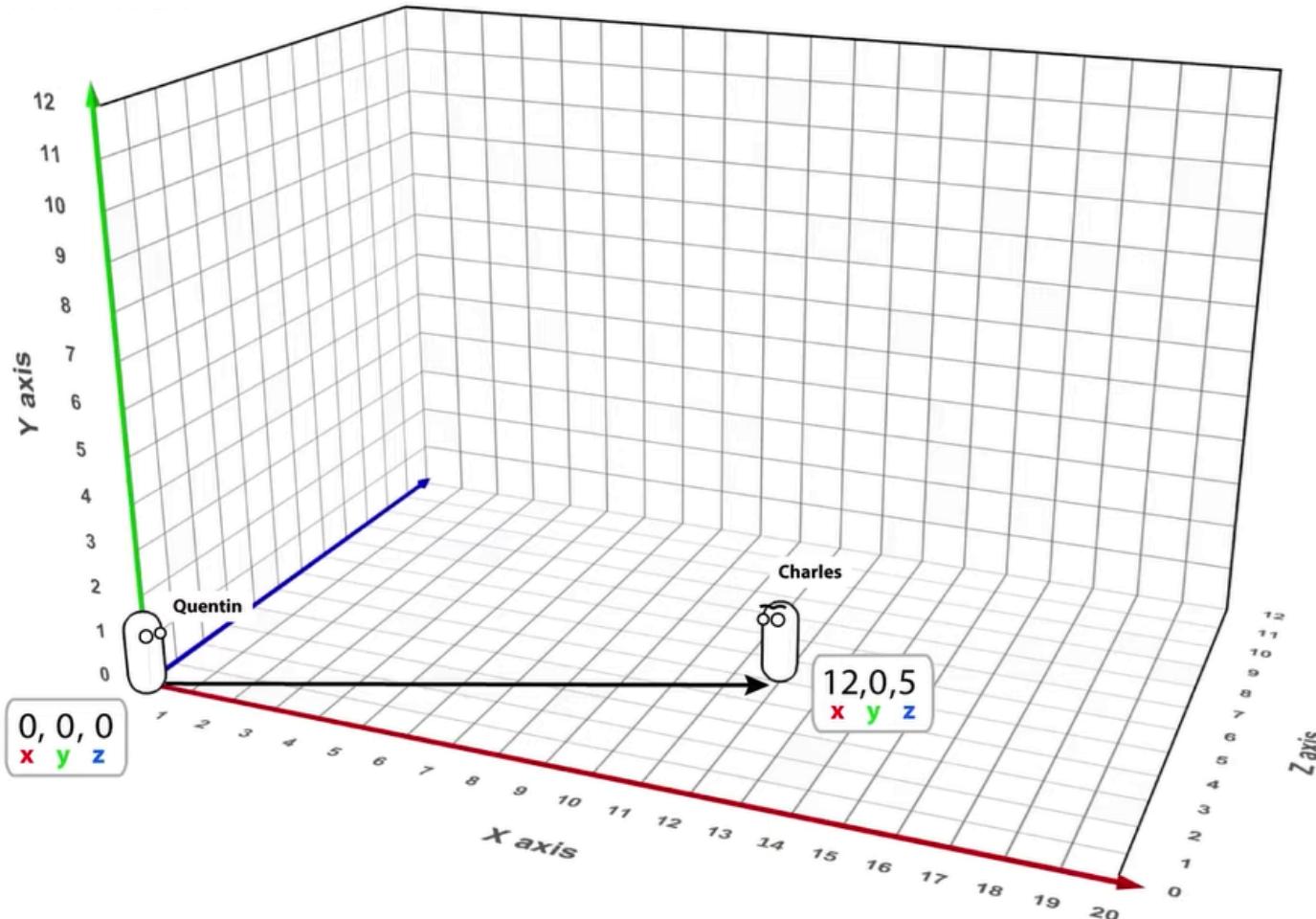
2D vs 3D

Transform: vectores 3D

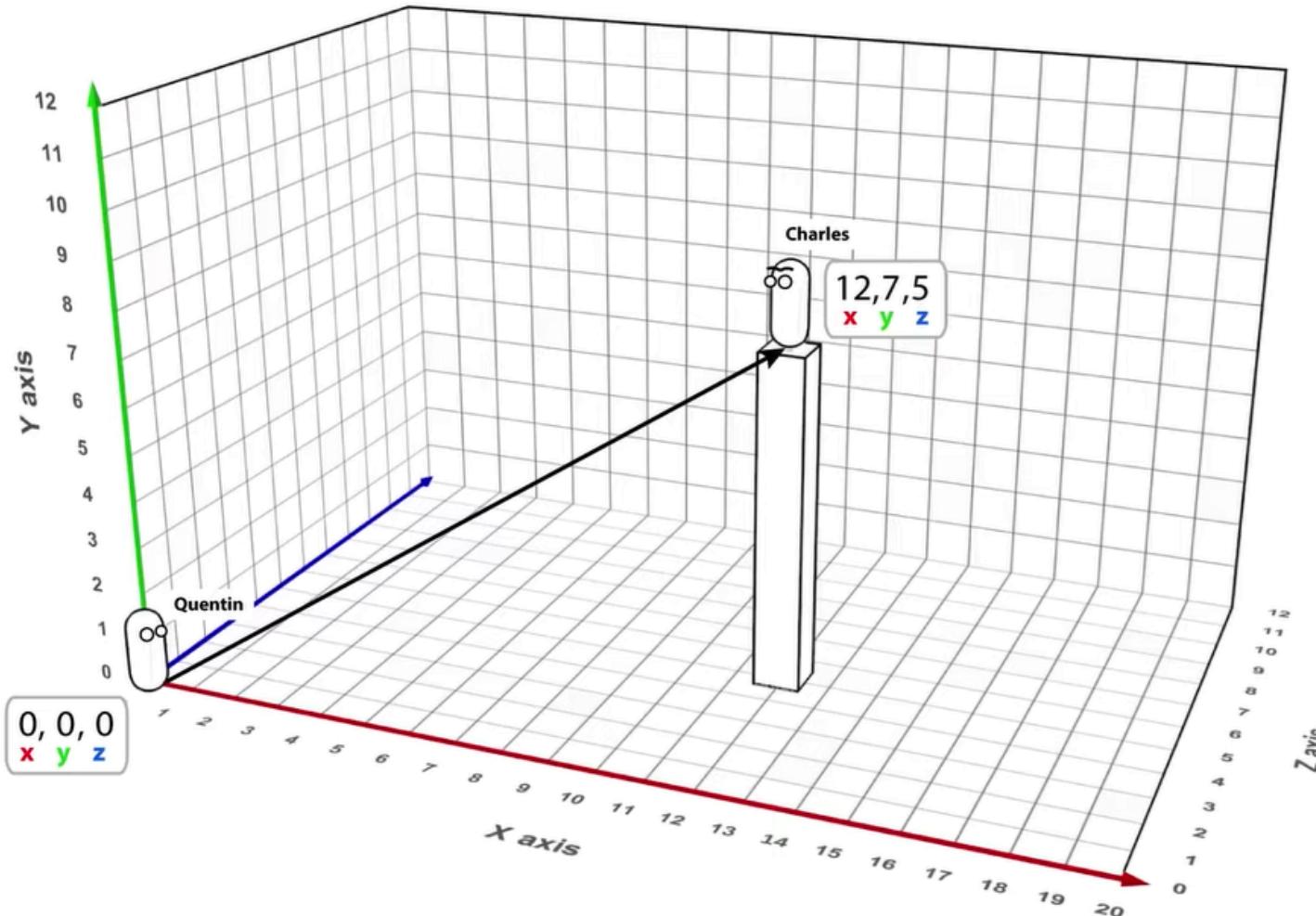


**Left Hand
Rule Coordinates**

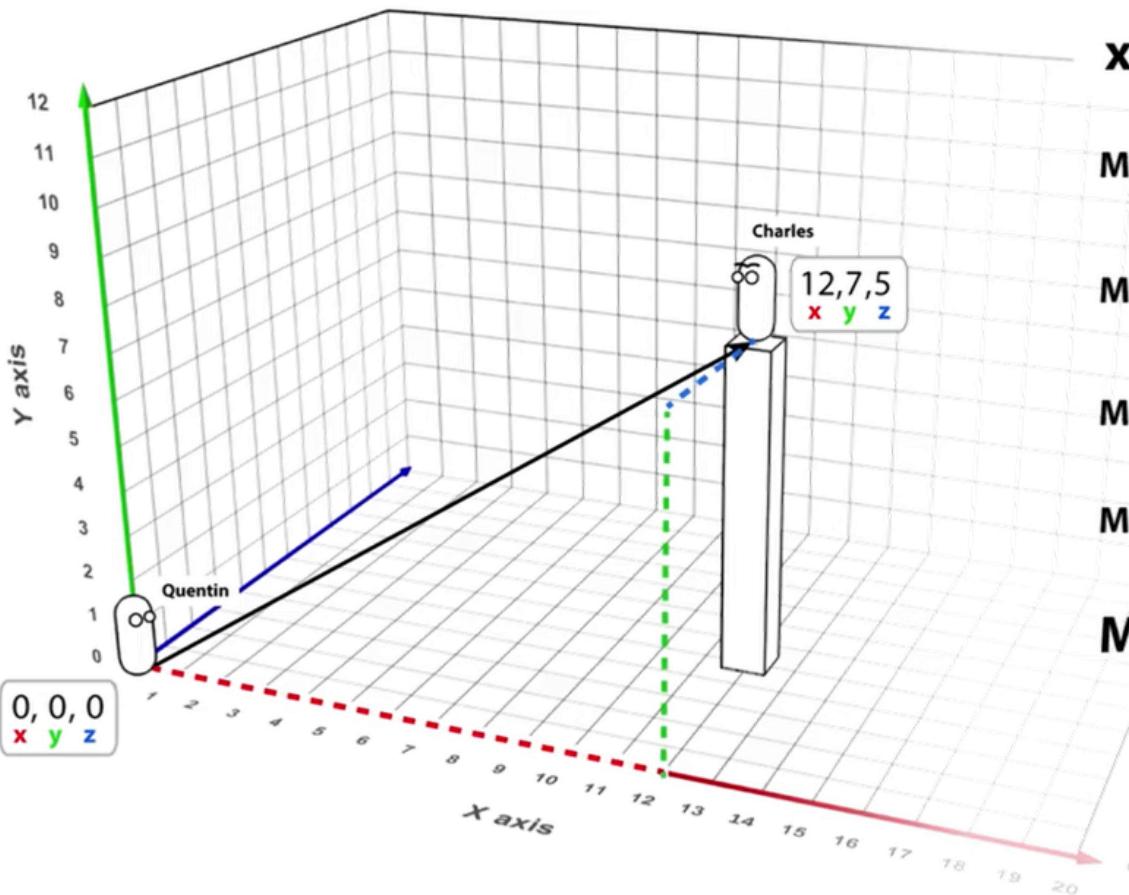
Transform: vectores 3D



Transform: vectores 3D



Transform: vectores 3D



$$x^2 + y^2 + z^2 = m^2$$

$$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

$$\text{Magnitude} = \sqrt{12^2 + 7^2 + 5^2}$$

$$\text{Magnitude} = \sqrt{144 + 49 + 25}$$

$$\text{Magnitude} = \sqrt{218}$$

$$\text{Magnitude} = 14.76$$

Vector3.magnitude

Transform

Transform.position -- Es para: "obtener/mover" la posición que un objeto con en el método de world space.
`Vector3 position`



Time.deltaTime -- Se usa para hacer que la velocidad del juego sea independiente en frames/s, para que sea en relación al tiempo y no a frames.
`float deltaTime`

Ejemplos: `Vector3.zero` → **(0,0,0)** | `Vector3.one` → **(1,1,1)** | `Vector3.up` → **(0,1,0)** | `Vector3.forward` → **(0,0,1)** | `Vector3.right` → **(1,0,0,)**.
Ejemplos: `transform.zero` → **(0,0,0)** | `transform.one` → **(1,1,1)** | `transform.up` → **(0,1,0)** | `transform.forward` → **(0,0,1)** | `transform.right` → **(1,0,0,)**.

CAMBIO DE POSICIÓN SOBRE LOS GAMEOBJECTS.

```
// Cambiar la posición de un objeto en X,Y,Z.  
transform.position = new Vector3 (0, 0, 0); | transform.position = Vector3.zero;  
  
// Cambiar la posición de un objeto en un solo canal de X,Y o Z.  
transform.position= new Vector3(10,0,0); | transform.position= new Vector3(0,10,0); | transform.position= new Vector3(0,0,10);  
ó  
transform.position= transform.right*10; | transform.position= transform.up*10; | transform.position= transform.forward*10;
```

MOVER OBJETOS CONTANTEMENTE DE ACUERDO AL TIEMPO/SEGUNDOS Y NO AL FRAMERATE.

Una manera de usar el Vector3 = **forward** (eje azul), **up** (eje verde), **right** (eje rojo).

```
void Update / LateUpdate () { // Update para personajes y LateUpdate para cámaras.  
    transform.position += transform.forward * 0.2f * Time.deltaTime;  
}  
  
void Update / LateUpdate () { // Update para personajes y LateUpdate para cámaras.  
    transform.position += transform.up * 0.2f * Time.deltaTime;  
}  
  
void Update / LateUpdate () { // Update para personajes y LateUpdate para cámaras.  
    transform.position += transform.right * 0.2f * Time.deltaTime;  
}
```

Entradas

La clase Input

- ▶ Gestiona todos los métodos de entrada
 - ▷ Teclado
 - ▷ Ratón
 - ▷ Gamepad/Joystick
 - ▷ Pantallas táctiles
 - ▷ Acelerómetro
 - ▷ Giroscopio
 - ▷ ...

La clase Input - Teclado

- ▶ Métodos para acceder al teclado:
 - ▷ `bool Input.GetKeyDown`: Estado actual de la tecla
 - ▷ `bool Input.GetKeyDown`: Acaba de pulsarse
 - ▷ `bool Input.GetKeyUp`: Acaba de soltarse
- ▶ Parámetro común de tipo KeyCode
 - ▷ `KeyCode.A`
 - ▷ `KeyCode.Space`
 - ▷ `KeyCode.LeftArrow`
- ▶ `Input.anyKey[Down]`

La clase Input - Ejes y botones virtuales

- ▶ Usa ejes y botones virtuales
 - ▷ Definidos en `ProjectSettings/Input`
- ▶ Mismo principio que `Input.GetKey*`
- ▶ Métodos
 - ▷ `bool Input.GetButton[Down|Up](string)`
 - ▷ Estado del botón virtual
- ▶ Toman como parámetro el nombre del botón o eje virtual

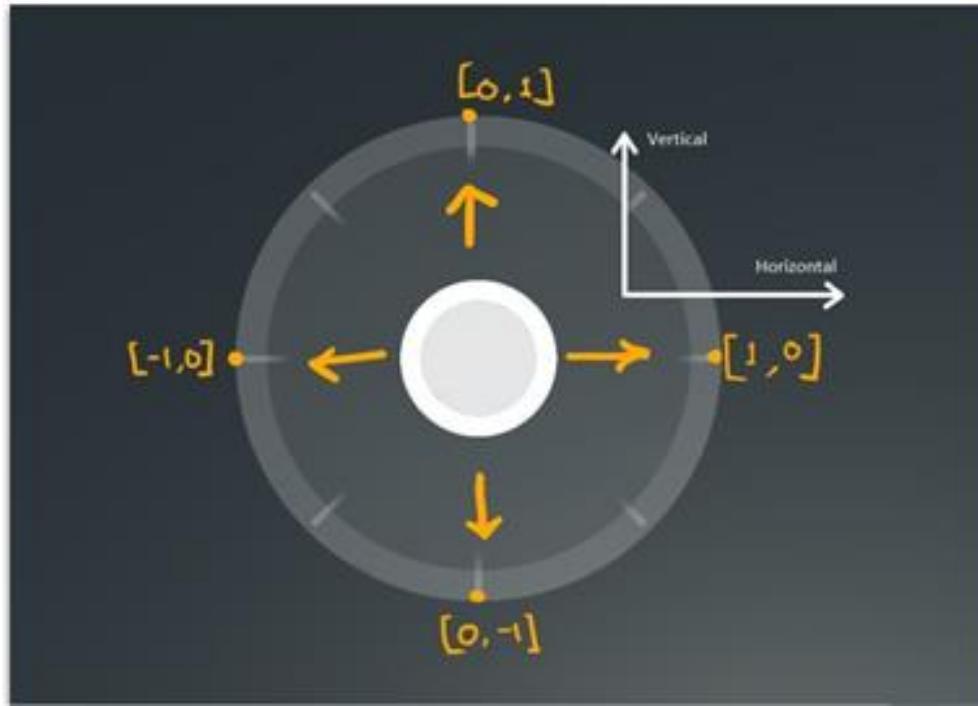
Entradas

La clase Input - Ejes y botones virtuales

- ▶ `float Input.GetAxis(string)`
 - ▷ Posición del eje virtual
 - ▷ Valores entre -1 y 1

La clase Input - Ratón

- ▶ Obtener estado de los botones
 - ▷ `Input.GetMouseButton[Down | Up]`
- ▶ Toma como parámetro el botón a comprobar:
 - ▷ 0: Botón principal del ratón (¡No necesariamente el izquierdo!)
 - ▷ 1: Botón secundario
 - ▷ 2: Botón terciario (central)
- ▶ Posición del ratón en pantalla
 - ▷ `Input.mousePosition`



Acceso a componentes

Componentes propios: GetComponent

- ▶ Obtiene un componente del propio objeto
- ▶ Familia amplia
 - ▷ GetComponent(s)
 - ▷ GetComponent(s)InChildren
 - ▷ GetComponent(s)InParent

GetComponent(s)

- ▶ Obtiene componente(s) del propio objeto
- ▶ Método por cadena, tipo y genérico
 - ▷ Component GetComponent("nombre")
 - ▷ Component GetComponent(System.Type)
 - ▷ T GetComponent<T>()

GetComponent(s)InChildren

- ▶ Busca un componente hacia abajo en la jerarquía
 - ▷ En los objetos hijo
- ▶ Similar a la versión "simple"
 - ▷ Component GetComponentInChildren(System.Type)
 - ▷ T GetComponentInChildren<T>()

GetComponent(s)InParent

- ▶ Busca un componente hacia arriba en la jerarquía
 - ▷ En los objetos padre
- ▶ Similar a GetComponentInChildren
 - ▷ Component GetComponentInParent(System.Type)
 - ▷ T GetComponentInParent<T>()

Instanciar objetos

- ▶ Instanciar → Crear copia de un objeto
 - ▷ Prefabs
 - ▷ Objetos de la escena
- ▶ Instantiate(GameObject)
 - ▷ Instantiate(GameObject, Vector3, Quaternion)

```
if (Input.GetMouseButtonDown(0))
{
    Vector3 mousePos = Input.mousePosition;
    mousePos.z = 10f;
    Vector3 worldPos = Camera.main.ScreenToWorldPoint(mousePos);

    Instantiate(prefab, worldPos, Quaternion.identity);
}
```

Destruir Objetos y Componentes

- ▶ Método *Destroy(Object)*
 - ▷ Funciona con GameObjects completos
 - ▷ Funciona con componentes
 - ▷ Se puede retrasar
 - ▷ *Destroy(Object, float delay)*



La clase Random

```
public GameObject[] prefabs;
public float maxRange = 10f;

0 referencias
void Update()
{
    if (Input.GetKeyDown(KeyCode.R))
    {
        Debug.Log("Valor aleatorio [0-1]: " + Random.value);
        Debug.Log("Valor entre 5 y 10: " + Random.Range(5f, 10f));
        Debug.Log("Valor entre 5 y 10 (entero): " + Random.Range(5, 10));
    }
    if (Input.GetKeyDown(KeyCode.I))
    {
        int idx = Random.Range(0, prefabs.Length);
        Vector3 newPosition = Random.insideUnitSphere * maxRange;
        Instantiate(prefabs[idx], newPosition, Quaternion.identity);
    }
}
```

Ejecución retrasada

Ejecución retrasada con Invoke

- ▶ El método Invoke permite esto
 - ▷ Retrasa la ejecución de un método específico el tiempo indicado
- ▶ Ejecutar el método “Método” a los 5 segundos
Invoke(“Metodo”, 5f)

Ejecución retrasada con InvokeRepeating

- ▶ Retrasa la ejecución de un método específico el tiempo indicado
 - ▷ Y repite la invocación cada cierto tiempo
- ▶ Ejecutar el método “Método” a los 5 segundos y repite la llamada cada 2 segundos
InvokeRepeating(“Metodo”, 5f, 2f)

Corrutinas

¿Qué son las corutinas?

- ▶ Métodos que pueden
 - ▷ Detener su ejecución
 - ▷ Devolver el control al programa
 - ▷ Continuar donde se quedaron más adelante

Creación de corutinas

- ▶ Métodos que devuelven un IEnumarator
- ▶ Contiene al menos una instrucción yield

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        StartCoroutine(CounterCoroutine());
    }
}

1 referencia
IEnumerator CounterCoroutine()
{
    counter = 0f;
    while(counter < 1f)
    {
        counter += .1f;
        Debug.Log("Valor de Counter: " + counter);
        yield return new WaitForSeconds(1f);
    }
}
```

La instrucción yield

- ▶ Se usa la instrucción *yield* para devolver un valor
- ▶ Usado en iteradores
 - ▷ Las Corrutinas son iteradores
- ▶ Toma dos formas:
 - `yield break`
 - `yield return expresión`
- ▶ Expresiones que se pueden devolver
 - ▷ `WaitForSeconds(float)`
 - ▷ `WaitForSecondsRealtime(float)`
 - ▷ `WaitForFixedUpdate()`
 - ▷ `WaitUntil(predicado)`
 - ▷ `WaitWhile(predicado)`
 - ▷ `AsyncOperation`
 - ▷ Carga de Escenas
 - ▷ Carga de assets
 - ▷ WWW

unir LA UNIVERSIDAD
EN INTERNET