



«Talento Tech»

Desarrollo de Videojuegos con Unity

CLASE 3



Clase N° 03 | ¡Empezando a Interactuar! (Movimiento)

Temario:

- If.
- Movimiento (AddForce, Velocity, Translate).
- Collider.
- Vector2.
- Rigidbody2D.
- FixedUpdate.

Condicionales.

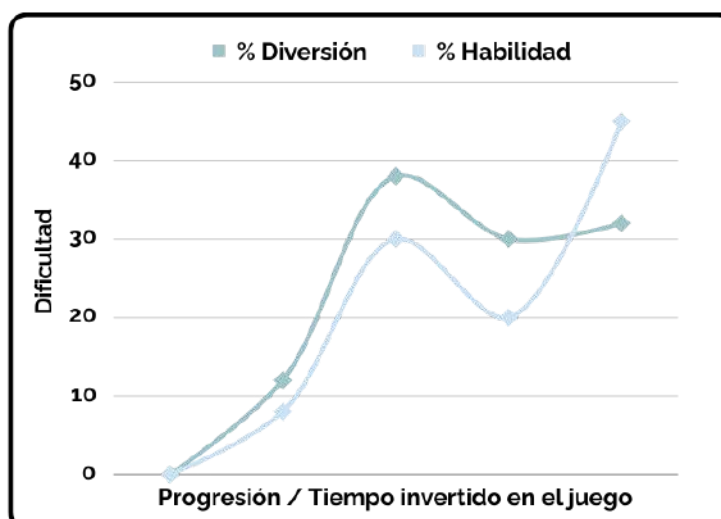
En programación utilizamos el término “instrucción” para describir una acción o comportamiento. Un código puede contener una o más instrucciones, pero ¿qué pasa cuando estas se ven afectadas por una condición?

Los condicionales son estructuras de control en el flujo de ejecución de un código que permite tomar decisiones dependiendo de si una condición es verdadera o falsa.

En los videojuegos la toma de decisiones le brinda al jugador un sentido de “control” sobre la narrativa y la continuidad del juego. Además estas pueden conducirlos a nuevos desafíos y consecuencias.

Los desafíos van acompañados con cambios progresivos en la curva de aprendizaje/dificultad.

CURVA DE APRENDIZAJE/DIFICULTAD



Desde nuestro lado como desarrolladores/as la idea es siempre brindarle a los/as jugadores/as una participación activa y hacer que experimenten con nuevas emociones.

Existen distintos tipos de estructuras. Vamos a ver las más utilizadas:

If (Si).

“If” es parte de las estructuras de control antes mencionadas y se usa para ejecutar un bloque de código si la condición dada es verdadera.

```
if (condicion)
{
    // Código a ejecutar si la condición es verdadera.
}
```

Else (Si no).

“Else” es otra de ellas y se usa en conjunto con “If” para ejecutar un bloque de código si la condición en “If” es falsa.

```
if (condicion)
{
    // Código a ejecutar si la condición es verdadera.
}
else
{
    // Código a ejecutar si la condición de if es falsa.
}
```

Else if (Si no, si).

“Else If” se usa para evaluar condiciones adicionales después de que las condiciones anteriores fueran falsas.

```
if (condicion1)
{
    // Código a ejecutar si la condición es verdadera.
}
else if (condicion2)
{
    // Código a ejecutar si la condición1 es falsa y la condición2 es verdadera.
}
else
{
    // Código a ejecutar si la condición de if es falsa.
}
```

&&(AND) y ||(OR).

Volviendo a nuestro código, ¿qué pasa cuando tengo más de una condición para poder continuar?

Te presentamos dos situaciones:

1. Para pasar al siguiente nivel tengo que recolectar 5 objetos **y** derrotar al jefe.
2. Para poder abrir el portal necesito 3 llaves **o** 2 ojos de gnomos.

Acá podemos ver el uso de operadores del tipo lógicos. Son los que nos ayudan a crear condiciones con un grado de complejidad adicional, porque los vamos a utilizar para combinar expresiones booleanas (**True** o **False**).

Vamos a verlas en detalle:

Operador	Situación	Resultado
&&	Condición 1 = ✓ && Condición 2 = ✓	True
&&	Condición 1 = ✓ && Condición 2 = ✗	False
&&	Condición 1 = ✗ && Condición 2 = ✗	False
	Condición 1 = ✓ && Condición 2 = ✓	True
	Condición 1 = ✓ && Condición 2 = ✗	True
	Condición 1 = ✗ && Condición 2 = ✗	False

Ahora traduzcamos en código las condiciones de pasar de nivel y apertura del portal.



¿Qué vamos a necesitar?

Principalmente variables para almacenar los datos que debemos evaluar. Y luego los ya vistos *If* y *Else*.

Situación 1:

Para pasar al siguiente nivel tengo que recolectar 5 objetos y derrotar al jefe.

```
int objetosRecolectados = 5;

bool jefeDerrotado = true;

void Start()

{

    if (objetosRecolectados >= 5 && jefeDerrotado)

    {

        Debug.Log("Pasaste de nivel");

    }

    else

    {

        Debug.Log("Todavía tenes que encontrar los
objetos y derrotar al jefe");

    }

}
```

Acá vamos a notar algo interesante, la variable “jefeDerrotado” está representada así pero podría estar representada de esta manera: “jefeDerrotado == True”. Lo escribimos así para resumir el código y optimizarlo para su lectura, ya que esa variable es el tipo booleano y a la vez verdadera por lo que debemos dejarla tal cual, si fuera falsa tendríamos que negarlo de esta manera: “!jefeDerrotado”.

Situación 2:

```
void Start()
{
    //Situacion 2: Abrir Portal

    int llaves = 3;

    int ojosDeGnomo = 2;

    if (llaves >= 3 || ojosDeGnomo >= 1)
    {
        Debug.Log("Podes abrir el portal!");
    }else
    {
        Debug.Log("Aun te faltan llaves u Ojos de Gnomo");
    }
}
```




Adicionalmente vamos a ver algunos operadores de comparación:

Igual a ($==$): Comprueba si dos valores son iguales.

No igual a ($!=$): Comprueba si dos valores no son iguales.

Mayor que ($>$): Comprueba si el valor de la izquierda es mayor que el de la derecha.

Menor que ($<$): Comprueba si el valor de la izquierda es menor que el de la derecha.

Mayor o igual que ($>=$): Comprueba si el valor de la izquierda es mayor o igual al de la derecha.

Menor o igual que ($<=$): Comprueba si el valor de la izquierda es menor o igual al de la derecha.

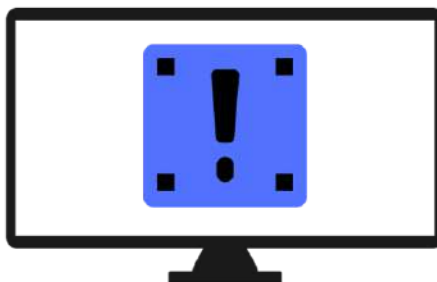
Input (GETKEY, GETKEYDOWN).

“Input” es una función que utilizan algunos lenguajes de programación para detectar datos de entrada.

Los **datos de entrada (inputs)** provienen de una fuente externa, generalmente de usuarios pero también pueden ser archivos o sensores, entre otras cosas.



Los **datos de salida (outputs)** son los resultados generados como consecuencia de procesar los datos de entrada.



Los inputs pueden incluir acciones del jugador/a (teclas presionadas, clics del mouse) y datos del juego, mientras que los outputs pueden ser cambios en la representación visual, modificaciones en el estado del juego, etc.

Inputs en Unity.

Input.GetKey() y ***Input.GetKeyDown()*** son **funciones** nativas de Unity, ambas sirven para detectar datos de entrada de un teclado en fotogramas de ejecución del juego. Además toman como parámetro un valor del tipo **KeyCode** seguido un punto y una tecla específica del teclado.

Veamos **Input.GetKey**: Verifica si una tecla específica está siendo presionada y funcionará mientras esté siendo presionada. Su resultado será **True** mientras la tecla esté siendo presionada y **False** cuando la tecla esté liberada.

Su sintaxis es la siguiente:

```
void Update()  
  
{  
  
    if (Input.GetKey(KeyCode.Space))  
  
    {  
  
        Debug.Log("Estas presionando la tecla espacio");  
  
    }  
  
}
```

Mientras la tecla espacio esté presionada va a imprimir el mensaje.


Para **Input.GetKeyDown** su resultado será **True** en el primer fotograma en el que la tecla es presionada y **False** en fotogramas subsiguientes hasta que la tecla es liberada y presionada nuevamente.

Su sintaxis es la siguiente:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("Estas presionando la tecla espacio");
    }
}
```

Solo va a imprimir el mensaje en el fotograma donde la tecla espacio es presionada por primera vez.



 **Datazo: Unity tiene 3 formas de mostrar información en consola. A través de Print, que funciona como Console WriteLine y Debug.Log. ¡Te invitamos a investigarlas!**

Movimiento de objetos

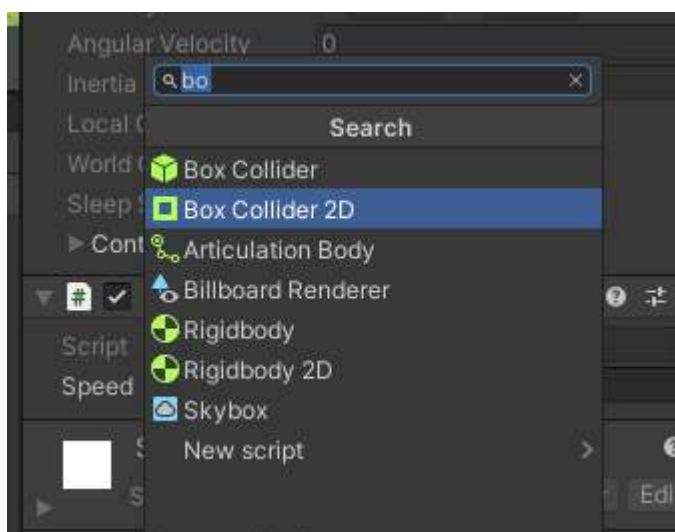
¿Qué pasa si combinamos condiciones con datos de entrada/salida?

Podemos lograr que nuestros objetos comiencen a “tomar vida”, generando movimiento.

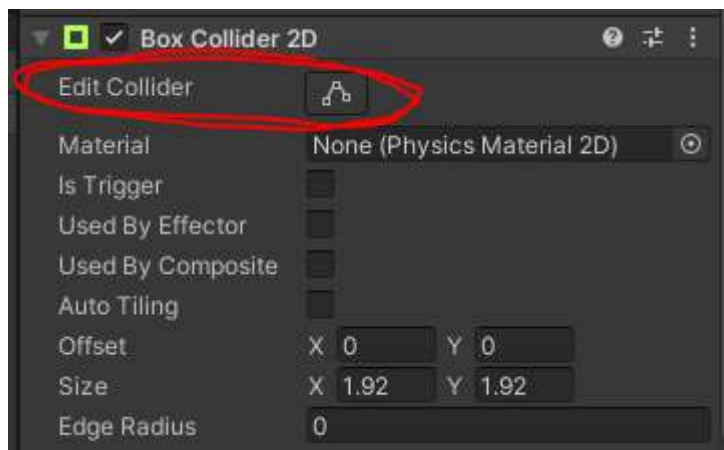
Tenemos varias formas de hacerlo, dos de ellas son a través de la función **AddForce()** y otra de **Translate()**.

Collider (Box Collider 2D)

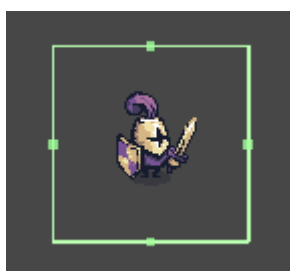
Para empezar a trabajar movimientos, tenemos que poner algunos componentes a nuestro personaje. El **Collider** o **HitBox** es fundamental para que luego podamos trabajar con las distintas consecuencias de colisiones o contactos.



Para colocarlo iremos al **Inspector**, seleccionaremos “Add Component” y buscaremos el “**Box Collider 2D**” y al elegirlo, veremos como se le forma un cuadrado verde a nuestro personaje, ese será los “límites de contacto” del objeto.

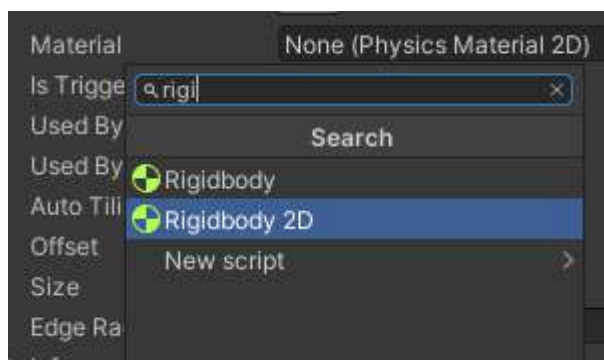


Si nos parece muy grande, siempre lo podremos modificar con “Edit Collider”.



Al estar puesto el Collider, nuestro personaje podrá “chocar” con cualquier objeto que posea uno.

Rigidbody2D



El **Rigidbody2D** es un componente que permite simular **físicas**. Esto nos dará la oportunidad de simular usos de fuerzas como la **gravedad** o empujes y genera mejores interacciones a la hora de **colisionar**. La forma de agregarlo es desde

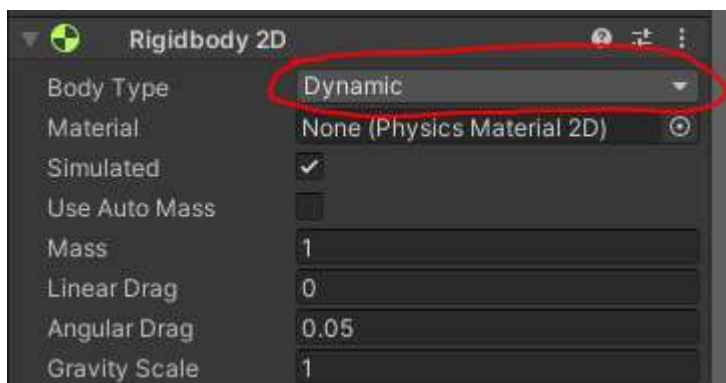


el *Inspector*. Seleccionamos nuestro **objeto**, y repetimos el proceso anterior.

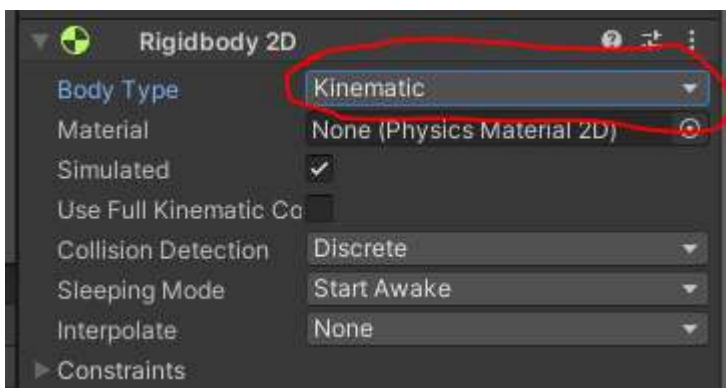
Una vez agregado, nuestro objeto adquiere propiedades físicas. Si probamos darle play al juego, vamos a notar que ahora el objeto es afectado por la gravedad.

Pero tenemos un problema, nuestro objeto se cae y no hay un suelo. Por eso, tenemos que configurar el Rigidbody2D.

Lo que podemos hacer es cambiar el “**Body Type**” de **Dynamic** a **Kinematic**.

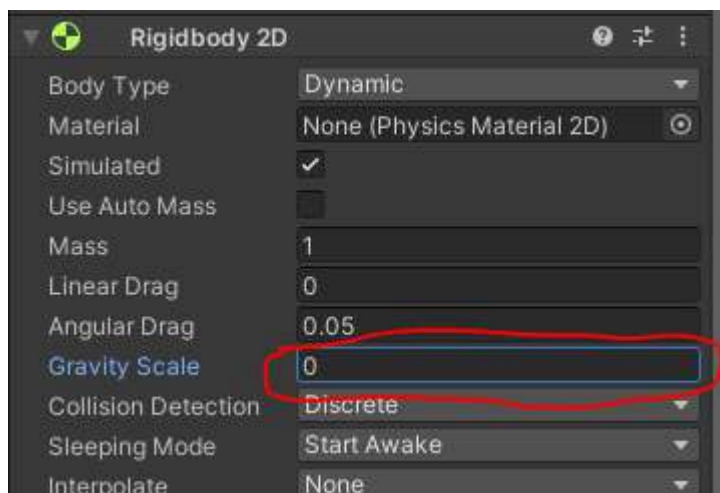


Dynamic: Es el estado por Default del Rigidbody2D, genera una simulación relativamente completa de las físicas y su interacción con el mundo.



Kinematic: Esta opción limita como afecta la física al GameObject. Podremos acceder a ellas mediante código pero las interacciones con el mundo estarán acortadas. Por ejemplo, no poseerá gravedad, ni será movido si colisiona con algún objeto.


Otra forma, es cambiar la variable "Gravity Scale" a 0(cero).



*Tengan en cuenta que esta variable SOLO aparece cuando el **Body Type** es **Dynamic**.*

AddForce()

En Unity se utiliza para aplicar fuerzas a un objeto con un componente Rigidbody. Lo utilizamos para simular movimiento en respuesta a eventos (Inputs), control de colisiones, entre otras. Va acompañado de Rigidbody, por lo tanto su sintaxis es: `Rigidbody.AddForce()`.

 **Super TIP importante:** Siempre asegurate de que el objeto al que querés aplicar la fuerza tenga un componente Rigidbody adjunto.

Crearemos un script, podés llamarlo como quieras siempre que guarde relación con el objeto/class/utilidad que le vayas a dar.

Paso 1. Nuestras variables, acá vamos a almacenar datos de movimiento, velocidad y nuestro Rigidbody

```
// Direccion

private int movimientoHorizontal = 0;

private int movimientoVertical = 0;

private Vector2 mov = new Vector2(0,0);


//Velocidad

[SerializeField] private float speed = 10;


//Variable que almacena el Rigidbody de nuestro objeto

//Para poder Acceder a sus propiedades

private Rigidbody2D rb;
```

Paso 2. Dentro de la función **Start()**, le asignaremos un valor a la **variable rb**. Utilizaremos la función “**GetComponent**” que nos permite obtener un componente del **gameObject** al que le hayamos puesto nuestro **Script**.

```
void Start()

{

    rb = GetComponent<Rigidbody2D>();

}
```

Paso 3. Aplicamos **if y else if** para verificar qué condiciones se deben cumplir para modificar el movimiento, tanto **horizontal** como **vertical**.

En este caso, el parámetro que les “pasamos” es la detección de los datos de entrada: **Input.GetKey()**.

```
void Update()

{

//Movimiento Vertical

    if (Input.GetKey(KeyCode.W))

    { movimientoVertical = 1; }

    else if (Input.GetKey(KeyCode.S))

    { movimientoVertical = (-1); }

    else { movimientoVertical = 0; } //puesto en el else
```

```
//Movimiento Horizontal

    if (Input.GetKey(KeyCode.D))

        { movimientoHorizontal = 1; }

    else if (Input.GetKey(KeyCode.A))

        { movimientoHorizontal = (-1); }

    else { movimientoHorizontal = 0; }//puesto en el else

}
```

Tengan en cuenta que antes de nuestros **if**, reseteamos los valores de nuestra **variables**, para que no mantengan su valor constantemente.

Paso 4. Pasamos como parámetros de dirección horizontal y vertical. Estos son los que necesita Unity para calcular la fuerza de movimiento:

```
// Asignamos la variable que determina la direccion

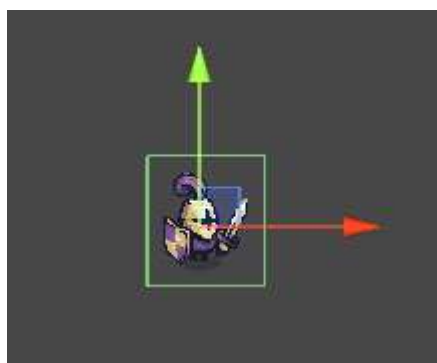
    mov = new Vector2(movimientoHorizontal, movimientoVertical);
```

El movimiento es en **vectores**.

¿Qué es un vector?

En física y matemáticas, un vector es un segmento de una línea recta, dotado de un sentido. Estos permiten representar magnitudes físicas dotadas no sólo de intensidad, sino de dirección.

Para nuestro código usamos el tipo de dato: Vector, en este caso Vector2 ya que nuestro juego es bidimensional



La X es el eje horizontal (derecha(+)) / izquierda(-))

La Y es el eje vertical (arriba(+)) / abajo(-))

Paso 5. La función Rigidbody.AddForce funciona con dos parámetros que son:

Vector2: El vector que representa la magnitud y dirección de la fuerza que se va a aplicar.

ForceMode: El modo de fuerza que especifica cómo se va a aplicar. Existen distintos tipos de modos de fuerza, pueden ser por impulso, por fuerza. “**Impulse**” funciona para simular cambios de velocidad repentinos, ya que aplica una fuerza instantánea. “**Force**” aplica una velocidad constante a lo largo de la ejecución y se ve afectado por la masa del objeto. En otras palabras **fuerza constante vs impulso**. Por default, si no se modifica, este aplica “**Force**”

Acá vamos a pasarle el vector (representado por la variable movimiento) y lo vamos a multiplicar por la fuerza (variable speed) y por “**Time.deltaTime**”, una función que calcula el tiempo en segundos que tardó en completarse el último frame, es decir el **tiempo entre Update() y Update()**. Otra forma de verlo es como una fracción del segundo que indica cuánto tiempo ha transcurrido entre el fotograma actual y el anterior. Se utiliza comúnmente para hacer que el movimiento y otras operaciones sean independientes de la velocidad de los fotogramas, es decir para evitar que este cálculo dependa del **FPS** y pase a trabajar en función del **tiempo**.

```

// Asignamos la variable que determina la direccion

        mov = new Vector2(movimientoHorizontal,
movimientoVertical);

        // Normalizamos el vector

        mov = mov.normalized;

        //Empujamos nuestro objeto

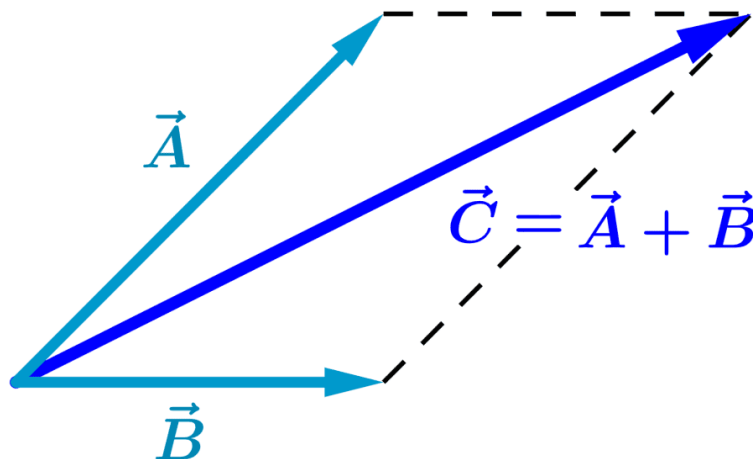
        rb.AddForce(mov * speed * Time.fixedDeltaTime);
    
```

Notaran que tenemos un “**movimiento.Normalize()**”. La función **Normalize**, reduce al vector a su magnitud mínima(1), conservando la dirección.

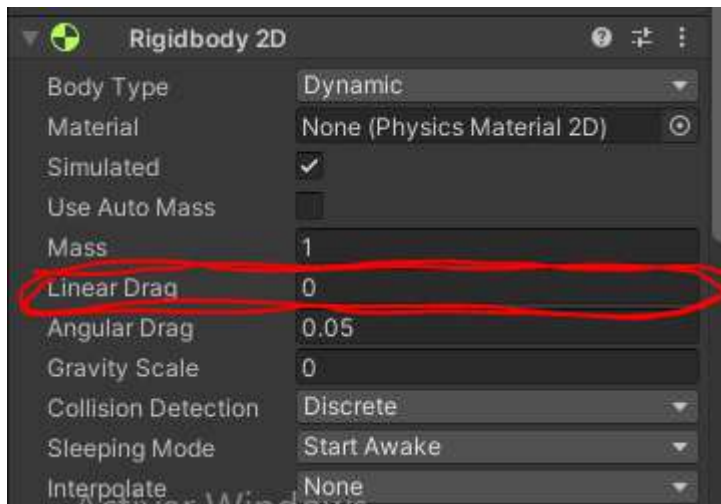
¿Por qué hacemos esto?

Cuando se normaliza, un vector mantiene la misma dirección pero su longitud es 1,0.

Al enviar sólo un eje, la magnitud del vector valdrá 1, pero si apretamos dos botones, estaremos enviando 2 valores, por ejemplo (1,1). Esto produce que para generar la dirección (Diagonal), el sistema suma los 2 vectores, haciendo que la velocidad incremente. Finalmente produciendo que, al ir en diagonal, nuestro personaje vaya más rápido. Para evitar esto, si es que deseamos evitarlo y NO transformarlo en un “Feature”(característica) de nuestro juego, usamos **Normalize**.



Tip: Verán que al aplicarle una fuerza a su objeto este se empezará a deslizar y no parara al momento de soltar los botones. Recordemos que estamos simulando las físicas y si aplicamos fuerza a un objeto y NO existe algo que lo detenga, como sucede en el espacio, este estaría en perpetuo movimiento. En nuestro caso y en el de Unity, tenemos acceso a las fuerzas de rozamiento, que desde Unity son simuladas con las variables “Linear Drag”(Rozamiento lineal) y “Angular Drag” (Rozamiento para la rotación).



Vayan probando modificar el **Linear Drag**, para generar distintos efectos de deslizamiento.

El código completo se vería de esta manera:

Apertura del Class + Definición de Variables + Start

```

public class Clase3 : MonoBehaviour
{
    // Direccion

    private int movimientoHorizontal = 0;

    private int movimientoVertical = 0;

    private Vector2 mov = new Vector2(0,0);

    //Velocidad

    [SerializeField] private float speed = 10;
    
```



```
//Variable que almacena el Rigidbody de nuestro objeto

//Para poder Acceder a sus propiedades

private Rigidbody2D rb;

void Start()

{

    rb = GetComponent<Rigidbody2D>();

}
```

Update + cierre del Class

```
void Update() {

//Movimiento Vertical

    if (Input.GetKey(KeyCode.W))

    { movimientoVertical = 1; }

    else if (Input.GetKey(KeyCode.S))

    { movimientoVertical = (-1); }

    else { movimientoVertical = 0; } //puesto en el else

}
```



```
//Movimiento Horizontal

    if (Input.GetKey(KeyCode.D))

    { movimientoHorizontal = 1; }

    else if (Input.GetKey(KeyCode.A))

    { movimientoHorizontal = (-1); }

    else { movimientoHorizontal = 0; } //puesto en el else

}

// Asignamos la variable que determina la direccion

mov = new Vector2(movimientoHorizontal, movimientoVertical);

    // Normalizamos el vector

    mov = mov.normalized;

    //Empujamos nuestro objeto

    rb.AddForce(mov * speed * Time.fixedDeltaTime);

}

}
```



Velocity()

Otra forma de mover a nuestro objeto utilizando Físicas es indicarle directamente la velocidad y dirección de nuestro objeto, sin producir el “empuje” del addforce.

Para esto modificaremos el “**rb.velocity**”, reemplazando al “AddForce()” previo.

```
// Asignamos la variable que determina la dirección
mov = new Vector2(movimientoHorizontal, movimientoVertical);
// Normalizamos el vector
mov = mov.normalized;
//Asignamos la velocidad a nuestro objeto
rb.velocity = mov * speed * Time.fixedDeltaTime;
```

¡ATENCIÓN! Si hicieron la prueba es posible que noten como el personaje no se mueve adecuadamente. ¿Por qué? por cómo se están procesando las físicas.

La solución es una función llamada “**FixedUpdate**”, esta es una función similar al Update, pero específica para procesar/leer/trabajar con físicas.

Pasaremos las líneas de código que trabajen con el procesamiento de físicas dentro de nuestra nueva función y tendremos que cambiar el “**Time.deltaTime**” por “**Time.fixedDeltaTime**”, siendo el comando debido para trabajar el tiempo en el FixedUpdate

```
private void FixedUpdate()  
  
    {  
  
        //Le asignamos la velocidad a nuestro objeto  
  
        rb.velocity = mov * speed * Time.fixedDeltaTime;  
  
    }
```

Translate().

La función Translate es otra alternativa al movimiento, la usaremos de forma similar a las anteriores. Pasaremos un **Vector2** para direccionar el movimiento y lo multiplicaremos por **speed** y ahora sí, podremos utilizar `time.deltaTime` de forma adecuada, ya que el translate, **NO utiliza físicas**, sino que accede al **transform** del objeto, trasladando el `GameObject` directamente.

¿Qué es el **transform**? Si vamos a Unity, seleccionamos cualquier objeto en la **escena** y vamos al inspector, notaremos que su primer componente siempre será “Transform”, este elemento es una característica universal de los objetos, que contiene 3 variables:

Position: La posición del objeto en los respectivos ejes.

Rotation: La rotación del objeto

Scale: El tamaño del objeto con respecto a su escala inicial (1,1,1)



En este caso, para mover el gameObject, usaremos la función **translate**, cuya utilidad es la mencionada. Recuerden acomodarlo dentro del **Update** y siendo una alternativa distinta a las 2 anteriores, borrar o comentar(//) el **rb.AddForce** o **rb.velocity**

```
// Asignamos la variable que determina la direccion
mov = new Vector2(movimientoHorizontal, movimientoVertical);

// Normalizamos el vector
mov = mov.normalized;

// Movemos al objeto
transform.Translate(mov * speed * Time.deltaTime);
```

Obviamente, como el **Translate()** no usa Físicas, lo colocaremos dentro del Update()

¡Listo! Ya podés hacer que tus objetos se muevan.

Con la ayuda de estructuras fundamentales, como "if," "else," y "else if," que nos brindan la flexibilidad necesaria para gestionar múltiples escenarios, pudimos integrar la toma de decisiones con el desarrollo de un código, explorando los inputs y los distintos movimientos que podemos aplicar a nuestros objetos.



Ejercicio práctico #2:

¡Sprint!

Una vez armado el movimiento, en el mismo script, creá una función que te permita cambiar la velocidad cuando apretamos la tecla “shift”.

Pista: Lo mejor en estos casos es tener tres variables. Una para la velocidad actual, y otras dos donde vamos a guardar dos velocidad distintas. Cuando apretamos “shift” deberíamos cambiar la velocidad actual por alguna de las dos velocidades.

No te olvides de **guardar** todos los cambios.

Buenos Aires
aprende

Agencia de Habilidades para el Futuro

