

«Talento Tech»

Videojuegos

Clase 03





Clase N° 3 | Conceptos básicos

Temario:

- Condicionales (&&, ||)
- Inputs
- Mover objetos con AddForce() y Translate()

Condicionales

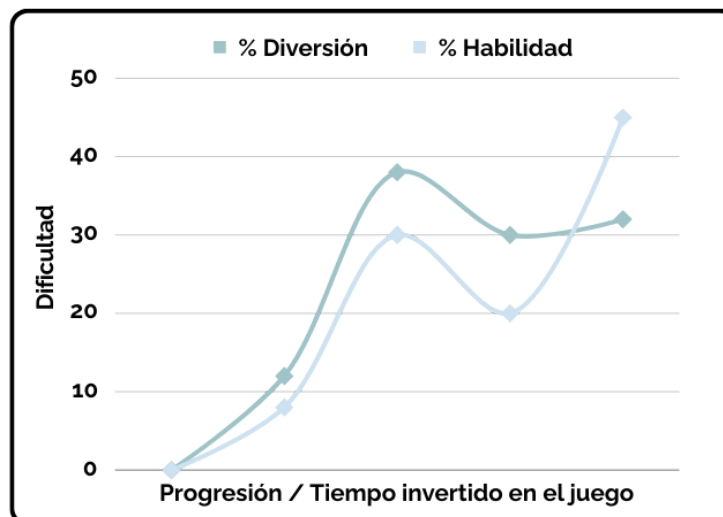
En programación utilizamos el término “instrucción” para describir una acción o comportamiento. Un código puede contener una o más instrucciones, pero, ¿Qué pasa cuando estas se ven afectadas por una condición?

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 la idea es siempre brindarle a los jugadores una participación activa y hacer que experimenten con nuevas emociones.

Existen distintos tipos de estructuras pero 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 presento dos situaciones:

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

Para poder abrir el portal necesito 3 llaves **o** 2 ojos de gnomo.

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**).

Veamoslas 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 |

Y 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.

```

void Start()
{
    // Situación 1: Pasar de nivel

    int objetosRecolectados = 5;
    bool jefeDerrotado = true;

    if (objetosRecolectados >= 5 && jefeDerrotado)
    {
        Console.WriteLine("¡Cumplís los requisitos para pasar de nivel!");
    }
    else
    {
        Console.WriteLine("Todavía tenés 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 la dejamos tal cual, si fuera falsa tendríamos que negarlo de esta manera: "¡jefeDerrotado".

Situación 2:

```
void Start()
{
    // Situación 2: Abrir el portal

    int llaves = 3;
    int ojosDeGnomos = 2;

    if (llaves >= 3 || ojosDeGnomos >= 2)
    {
        Console.WriteLine("¡Podés abrir el portal y continuar!");
    }
    else
    {
        Console.WriteLine("No tenés suficientes llaves ni ojos de gnomos para abrir el portal.");
    }
}
```

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 (GET KEY, GET KEY DOWN)

En la clase 2 vimos funciones (completar según datos clase 2). “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 (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()
{
    // Verifica si la tecla "Espacio" está siendo presionada.
    if (Input.GetKey(KeyCode.Space))
    {
        Debug.Log("Estás 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:

```
void Update()
{
    // Verifica si la tecla "Espacio" ha sido presionada en este fotograma.
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("La tecla Espacio ha sido presionada.");
    }
}
```

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 dos formas de hacerlo, una a través de la función **AddForce()** y otra de **Translate()**.

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

Vamos a usar el cubo que hicimos la clase pasada. Al que le añadimos el componente Rigidbody.

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

Punto 1. Nuestras variables, acá vamos a almacenar datos de movimiento y velocidad. Vamos a borrar nuestra función `Start()`, ya que generalmente, la llamada a `Rigidbody.AddForce()` se realiza en la función `Update()` para aplicar la fuerza en cada fotograma.

```

//Velocidad que luego vamos a multiplicar:
public float velocidad = 5f;

Mensaje de Unity | 0 referencias
void Update()
{
    //Acá almacenamos los movimientos:

    float movimientoHorizontal = 0f;
    float movimientoProfundidad = 0f;
    
```

Punto 2. 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()*.

```
// Usando GetKey y condicionales,
// ejecutaremos distintos movimientos dependiendo de la tecla presionada:

// Movimiento horizontal
if (Input.GetKey(KeyCode.D))
    movimientoHorizontal = 1f;
else if (Input.GetKey(KeyCode.A))
    movimientoHorizontal = -1f;

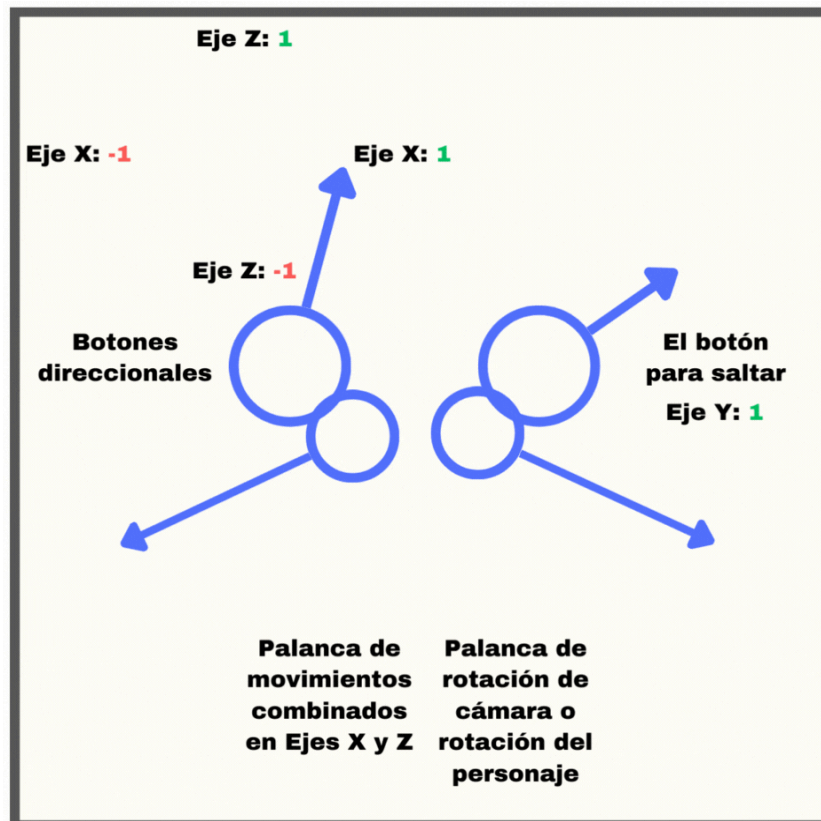
// Movimiento vertical
if (Input.GetKey(KeyCode.W))
    movimientoProfundidad = 1f;
else if (Input.GetKey(KeyCode.S))
    movimientoProfundidad = -1f;
```

Punto 3. Pasamos como parámetros el movimiento horizontal, dejamos el vertical como está porque no vamos a aplicar salto, y pasamos el movimiento en profundidad. Estos son los que necesita Unity para calcular la fuerza de movimiento:

```
// Calculamos la fuerza de movimiento:

Vector3 movimiento = new Vector3(movimientoHorizontal, 0f, movimientoVertical);
```

Para entender esto tenemos que ver cómo funcionan los ejes cartesianos en Unity:

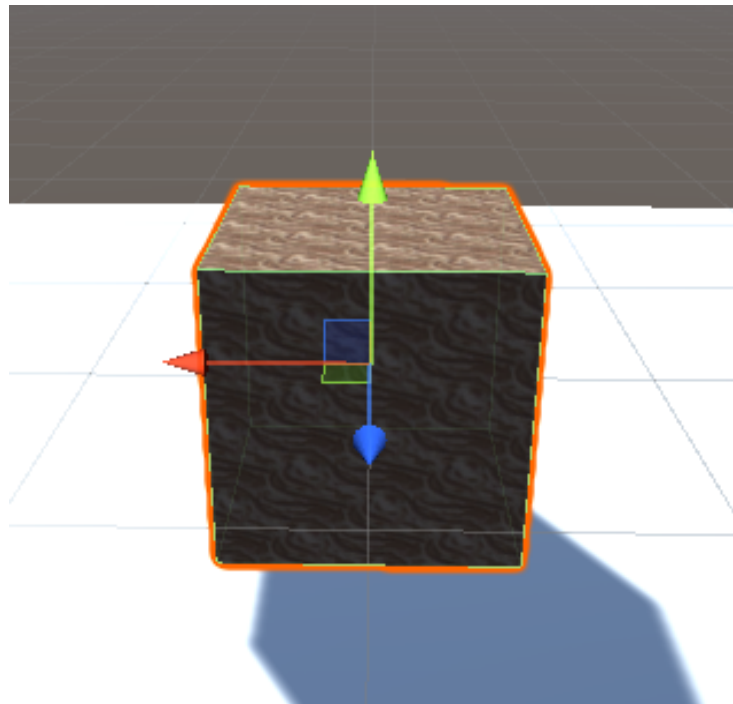


El movimiento es en vectores. Para nuestro código usamos el tipo de dato: Vector, en este caso Vector3 ya que nuestro juego es tridimensional:

La X es el eje horizontal (derecha - izquierda)

La Y es el eje vertical (arriba - abajo)

La Z es el eje de profundidad (adelante - atrás)



Punto 4. La función `Rigidbody.AddForce` funciona con dos parámetros:

Uno es `Vector3 force`: El vector que representa la magnitud y dirección de la fuerza que se va a aplicar.

`ForceMode mode`: El modo de fuerza que especifica cómo se va a aplicar. Existen distintos tipos de modos de fuerza, pueden ser por impulso, por fuerza. Impulso 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.

Acá vamos a pasarle el vector (representado por la variable `movimiento`) y lo vamos a multiplicar por la fuerza (variable `velocidad`):

```
// Aplicamos esa fuerza al Rigidbody que acompañe al objeto:
GetComponent<Rigidbody>().AddForce(movimiento * velocidad);
```

Translate()

La función Translate es otra alternativa al movimiento, la vamos a utilizar para hacer programar un salto.

Salto: añadimos dos variables a nuestro código, **altura** y **velocidad**.

Utilizando **condicionales** e *Input()* vamos a movernos en el eje vertical.

Vector3.up representa el vector (0, 1, 0)

Time.deltaTime es: Es 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.

```
float alturaSalto = 5f;
float velocidadSalto = 2f;

// Verifica si el jugador presiona la tecla de salto:
if (Input.GetKeyDown(KeyCode.Space))
{
    // Realiza el salto en el eje Y.
    transform.Translate(Vector3.up * alturaSalto * velocidadSalto * Time.deltaTime);
}
```

¡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.



Desafío N° 3:

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.

Tomá una **captura de pantalla** del código.

Subilas al espacio correspondiente del Desafío 3.



Buenos Aires
aprende
Agencia de Actividades para el Futuro

BA Buenos
Aires
Ciudad