«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 02

Buenos Aires Ciudad

Clase N° 2 | Movimientos en 3D

Temario:

- Implementación de controles de personaje con Input.
- Física en 3D: Rigidbody y Force.
- Raycast:Salto y detección de suelo.
- Configuración de la cámara en juegos 3D: Follow.

Objetivos de la clase

En esta clase vas a desarrollar el sistema de control del personaje en el entorno 3D. El objetivo será lograr un movimiento fluido y preciso utilizando el sistema de Input, integrando físicas realistas con Rigidbody, y sumando mecánicas clave como el salto, el doble salto y el dash. También vas a configurar una cámara dinámica que siga al jugador y aporte inmersión. Cada herramienta que apliques tendrá un propósito claro dentro del juego: mejorar la experiencia del jugador y mostrar el potencial del prototipo al cliente.

Implementación de controles de personaje con Input

Diseñar un sistema de control que permita mover al personaje en un entorno 3D usando entradas del teclado o un joystick, de manera fluida y sensible al contexto del juego.

¿Por qué es importante?

El control del personaje es el primer contacto entre el jugador y el mundo del juego. Si el personaje no se mueve de forma intuitiva, rápida y fluida, toda la experiencia pierde impacto. En Unity, esto se logra a través del sistema de **Input**, que permite leer entradas del usuario (teclas, botones o sticks) y traducirlas en movimiento dentro de la escena.

Input System

Este es el sistema clásico que Unity incluye por defecto. Empezaremos a trabajar utilizando un sistema de Inputs distinto del que veníamos haciendo:

```
// Obtener los valores de los ejes horizontales y verticales.

float horizontalInput = Input.GetAxis("Horizontal"); // Movimiento en el eje X.

float verticalInput = Input.GetAxis("Vertical");     // Movimiento en el eje Z.
```

Este sistema se basa en nombres de ejes y botones predefinidos en el menú:

`Edit > Project Settings > Input Manager`

Es fácil de usar y rápido para prototipos, pero limitado cuando necesitás manejar:

- Varios jugadores o dispositivos al mismo tiempo
- Reasignación dinámica de controles
- Entrada compleja (ej. gamepad + teclado + UI simultáneamente)

`Input.GetAxis("Horizontal")` y `Input.GetAxis("Vertical")` devuelven valores entre -1 y 1, lo que permite crear movimientos **suaves y continuos** en lugar de respuestas bruscas. Estos ejes están preconfigurados para responder a teclas como **WASD** o las **flechas del teclado**, y también a joysticks si están disponibles.

Este método es muy útil para manejar movimiento o rotación en un juego, ya que proporciona un valor suave y continuo, ideal para una experiencia más natural. Los ejes

más comunes predefinidos en Unity son **Horizontal y Vertical**, que suelen estar configurados para las teclas de dirección (flechas), las teclas W, A, S, D o los sticks de un gamepad.

De esta manera obtendremos un resultado más fluido y versátil que el que veníamos haciendo. Un código sencillo de movimiento en 3D se veria asi:

```
public float moveSpeed = 5f; // Velocidad de movimiento del jugador.
private Rigidbody rb; // Referencia al Rigidbody.
 void Start(){
rb = GetComponent<Rigidbody>();  // Obtener la referencia al Rigidbody.
 }
 void FixedUpdate(){
   // Obtener los valores de los ejes horizontales y verticales.
   float horizontalInput = Input.GetAxis("Horizontal"); // Movimiento en el eje X.
  float verticalInput = Input.GetAxis("Vertical");     // Movimiento en el eje Z.
   // Crear un vector de movimiento basado en los valores de los ejes.
   Vector3 movement = new Vector3(horizontalInput, 0, verticalInput);
   movement = movement.normalized;
  // Aplicar movimiento al Rigidbody.
   rb.velocity = new Vector3(movement.x * moveSpeed, rb.velocity.y, movement.z*moveSpeed);
}
```

Como verán, no cambia mucho más que la disponibilidad en la que se colocan los datos en los que sería el “Vector3”. La principal diferencia, como se mencionó en la clase anterior, es que ahora manejaremos los ejes X,Y,Z. Siendo X,Y para movernos a los alrededores (horizontal y profundidad) y el eje Y, será para las alturas.

Es por eso que en este caso dejamos el valor de Y en 0:

```
Vector3 movement = new Vector3(horizontalInput, 0, verticalInput);
```

Rigidbody y Force

En un entorno 3D, el movimiento debe sentirse realista y con peso. Unity tiene un motor de físicas incorporado que nos permite simular gravedad, colisiones y fuerzas. Para que un objeto sea afectado por esas reglas, debe tener un componente Rigidbody.

Esto permite que el personaje reaccione como un cuerpo físico: puede caer, ser empujado, rebotar o resistir impactos, dependiendo de cómo lo programemos.

El **Rigidbody** es un componente que le da al objeto la capacidad de:

- Ser afectado por la gravedad
- Recibir fuerzas (con AddForce)
- Detectar colisiones con otros objetos físicos
- Controlar su masa, arrastre (drag) y rotación

Se puede agregar desde:
Inspector > Add Component > Rigidbody

Aplicar movimiento con física

En lugar de cambiar directamente la posición o la velocidad del objeto, podés aplicar una **fuerza** para que se mueva de forma natural:

```
public float jumpForce = 5f;
void Jump() {
    rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
}
```

- `Vector3.up` equivale a `(0,1,0)`, es decir, una fuerza hacia arriba en el eje Y.
- `ForceMode.Impulse` aplica una fuerza repentina, ideal para saltos.

¿Cuándo usar `velocity` vs. `AddForce`?

Método	Comportamiento	Uso típico
`rb.velocity`	Cambia la velocidad de forma directa	Movimiento controlado
`rb.AddForce`	Aplica fuerza como en el mundo real	Saltos,empujes, explosiones

Ambos métodos pueden convivir: por ejemplo, podés usar `velocity` para moverte y `AddForce` para saltar.

Controlar el eje Y

Como vimos antes, para mantener el control de la **altura** del personaje (eje Y), es común que el movimiento horizontal **preserve la velocidad vertical**, para no interferir con la gravedad ni el salto:

```
rb.velocity = new Vector3(movement.x * speed, rb.velocity.y, movement.z * speed);
```

Esto permite que el personaje siga cayendo (por gravedad) o mantenga el impulso de un salto, aunque se mueva lateralmente.

Salto (Raycast)

Para seguir interactuando con nuestro nuevo espacio crearemos un salto. Para esto, deberemos buscar maneras para que reconozca cuando está en el suelo. Como venimos acostumbrados, hay varias maneras de hacerlo. Por ejemplo, podríamos estar chequeando la velocidad en Y de nuestro personaje. Si es = a 0 saltaría. Pero en este caso, utilizaremos el **Raycast**, una forma sencilla de detección que nos avisará cuando estemos “con los pies en la tierra”.

Un Raycast en Unity es una técnica que permite proyectar un rayo invisible desde un punto en el espacio hacia una dirección y detectar si colisiona con algún objeto dentro de un rango especificado. En este caso, lo usamos para **detectar si hay suelo debajo del personaje:**

.

```
bool isGrounded = Physics.Raycast(transform.position, Vector3.down, rayLength);
```

Esto es útil para interacciones en el juego, como detección de objetos, colisiones o validación de posiciones.

```
public float jumpForce = 5f; // Fuerza del salto
public float rayLength = 1.1f; // Longitud del Raycast
private Rigidbody rb; // Referencia al Rigidbody del jugador
void Update(){
 // Verificar si el jugador está en el suelo usando un Raycast
      bool isGrounded = Physics.Raycast(transform.position, Vector3.down, rayLength);
 //Dibujar el rayo en la vista de la escena para depuración
 Debug.DrawRay(transform.position, Vector3.down*rayLength, Color.red);
//Permitir el salto si el jugador está en el suelo y presiona la barra espaciadora
 if (isGrounded && Input.GetKeyDown(KeyCode.Space)){
   Jump(); } }
void Jump(){
 // Aplicar fuerza hacia arriba para saltar
    rb.AddForce(0, jumpForce, 0); }
```

Aplicación del Raycast:

```
    bool isGrounded = Physics.Raycast(transform.position, Vector3.down, rayLength);
```

Este método devuelve un valor booleano (true o false), indicando si el rayo chocó con algo dentro del rango especificado.

¿Cómo funciona?

- **transform.position:** Representa el punto de origen del rayo. En este caso, el rayo se lanza desde la posición actual del objeto al que está asociado este script.
- **Vector3.down:** Es un vector que apunta hacia abajo en el eje Y global. Esto significa que el rayo se proyecta hacia abajo desde el origen (transform.position), ideal para detectar si hay suelo debajo del objeto.
- **rayLength:** Es un valor flotante (float) que define la distancia máxima que recorrerá el rayo desde su origen. Si no encuentra un objeto dentro de esta distancia, el método devolverá false.
- **bool isGrounded:** Es una variable booleana que almacena el resultado de Physics.Raycast.
    - Si el rayo colisiona con algo dentro del rango de rayLength, su valor será true.
    - Si no encuentra ningún objeto, será false.

🔴 Si el rayo **colisiona** con algo dentro de ese rango (por ejemplo, el suelo), `isGrounded` será **true**.

⚫ Si no detecta nada (por estar en el aire), será **false**.

Si sumamos esta linea de codigo con un condicional sencillo, podemos decirle que solo salte, “**si estoy tocando el piso**”:

```
    if (isGrounded && Input.GetKeyDown(KeyCode.Space)) {
        Jump();
    }
```

Con todo lo que armamos, tendremos un codigo de movimiento basico para nuestro primer juego 3D.

Cámara.

En juegos 3D, el jugador necesita mantener al personaje en pantalla **desde un ángulo adecuado**. A diferencia del 2D (donde muchas veces la cámara es fija o lateral), en 3D el entorno tiene profundidad, altura y movimiento en múltiples direcciones.

Una **cámara que siga bien al personaje mejora la inmersión y la jugabilidad**. Una mal configurada puede dificultar el control, marear o incluso ocultar información clave del entorno.

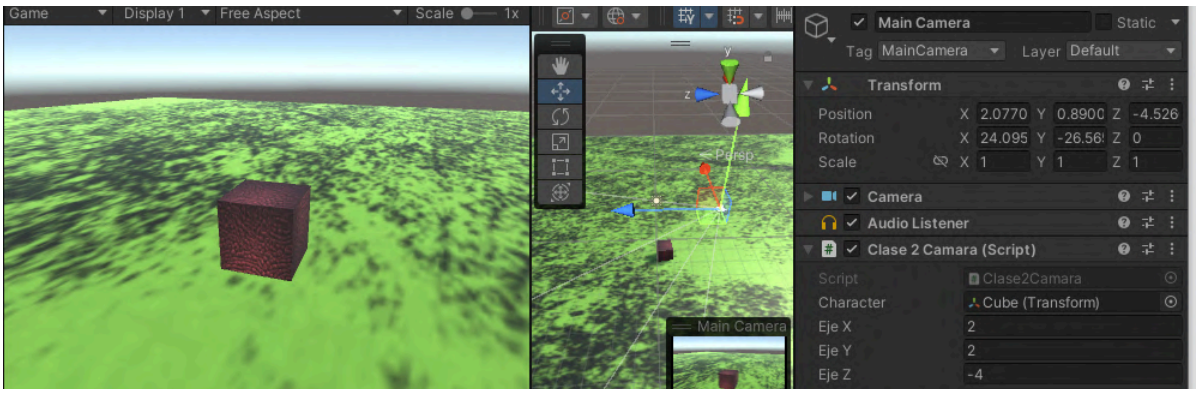
Por ahora usaremos un código muy sencillo:

```
[SerializeField] private Transform character;

    [SerializeField] private float ejeX = 0f;

    [SerializeField] private float ejeY = 0f;

    [SerializeField] private float ejeZ = 0f;

    void Update() {

    transform.position = character.position + new Vector3(ejeX, ejeY, ejeZ);

        transform.LookAt(character);

}
```

Este script permite que una cámara **siga al personaje desde una posición relativa ajustable**, y que siempre lo mantenga en el centro de la vista.

Desglose paso a paso

- `character`: es una referencia al objeto (generalmente el jugador) que la cámara debe seguir.
- `ejeX`, `ejeY`, `ejeZ`: definen el **desplazamiento de la cámara respecto al personaje**. Son modificables desde el **Inspector de Unity**, lo que permite experimentar fácilmente con diferentes ángulos.
- `transform.position = character.position + new Vector3(ejeX, ejeY, ejeZ);`:
  Esta línea ubica la cámara en una posición **relativa** al personaje. Por ejemplo:
  - Un valor alto en `ejeY` coloca la cámara más arriba.
  - Un valor negativo en `ejeZ` la aleja hacia atrás.
- `transform.LookAt(character);`:
  Esta línea **hace que la cámara mire siempre al personaje**, manteniéndolo visible al centro de la escena.



¡Nuevo reto! Double Jump y Dash

Ya desarrollaste el movimiento básico del personaje: puede desplazarse por el entorno 3D con fluidez, saltar solo cuando está en el suelo y es seguido por una cámara funcional.

Ahora que dominás los fundamentos, es momento de **llevar el sistema de control un paso más allá**, incorporando **mecánicas avanzadas** que aparecen en muchos videojuegos actuales. Estas mecánicas no solo mejoran la jugabilidad, sino que también **te permiten practicar la integración de lógica condicional, variables de control, raycasts y física en tiempo real**.

En este nuevo bloque vas a trabajar dos desafíos:

Doble salto (Double Jump): una mecánica que permite al personaje realizar un segundo salto mientras aún está en el aire.
Dash: un desplazamiento rápido en la dirección actual del movimiento, útil para esquivar obstáculos o recorrer distancias cortas de forma dinámica.

Ambos desafíos te permitirán reforzar conceptos como:

- Control de estado del personaje (¿está en el aire?, ¿cuántos saltos le quedan?)
- Uso de variables que se actualizan según el entorno y las acciones del jugador
- Aplicación de fuerzas y manipulación de la posición del objeto

Mecánica: Double Jump

Empecemos a construir nuestras mecánicas. Crearemos un doble salto o “Double Jump”. Para esto tendremos que modificar nuestro código de salto, haciendo que

```
void Update(){
 if (saltosRestantes <= 0){
  // Verificar si el jugador está en el suelo usando un Raycast
 isGrounded = Physics.Raycast(transform.position, Vector3.down, rayLength);
  if (isGrounded){
    saltosRestantes = 2;
          } }

    // Permitir el salto si el jugador está en el suelo y presiona la barra espaciadora y tiene saltos
restantes
   if (isGrounded && Input.GetKeyDown(KeyCode.Space) && saltosRestantes > 0){
    saltosRestantes--;
    Jump();
       } }
 void Jump(){
   // Aplicar fuerza hacia arriba para saltar
    rb.velocity = new Vector3(rb.velocity.x, 0 , rb.velocity.z);
    rb.AddForce(0, jumpForce, 0);
    }
```

Explicación del código:

1) Verificación de si el jugador está en el suelo

```
   if (saltosRestantes <= 0)
{
   // Verificar si el jugador está en el suelo usando un Raycast
   isGrounded = Physics.Raycast(transform.position, Vector3.down, rayLength);
   if (isGrounded)
   {
      saltosRestantes = 2;
   }
}
```

- **saltosRestantes:**
  - Controla cuántos saltos adicionales puede realizar el jugador.
  - Se reinicia a 2 cuando el jugador vuelve al suelo.

- **Raycast (Physics.Raycast):**
    - Lanza un rayo invisible desde la posición del jugador (transform.position) hacia abajo (Vector3.down) con una longitud (rayLength) para detectar si está tocando el suelo.
    - Si el rayo detecta el suelo (isGrounded es true), el contador de saltos (saltosRestantes) se reinicia a 2.

Esto asegura que los saltos solo se recargan cuando el jugador está en contacto con el suelo.

2) Salto del jugador:

```
if (isGrounded && Input.GetKeyDown(KeyCode.Space) && saltosRestantes > 0){

   saltosRestantes--;

   Jump();

}
```

Condiciones para saltar:

- El jugador debe estar en el suelo (isGrounded).
- La tecla de salto (Space) debe ser presionada.
- Debe tener al menos un salto disponible (saltosRestantes > 0).

Reducción de saltos restantes:

- Cada vez que salta, saltosRestantes se reduce en 1. Esto permite un máximo de dos saltos consecutivos.

Llamada al método Jump:

- Ejecuta la lógica del salto para aplicar una fuerza hacia arriba.

3) Lógica del salto

```
void Jump()
{
    // Aplicar fuerza hacia arriba para saltar
    rb.velocity = new Vector3(rb.velocity.x, 0, rb.velocity.z);
    rb.AddForce(0, jumpForce, 0);
}
```

rb.velocity:

- Restablece la velocidad vertical (y) a 0 para evitar acumulaciones de fuerzas de salto si el jugador salta rápidamente varias veces seguidas. Las velocidades en los ejes x y z se mantienen iguales para no alterar el movimiento horizontal.

rb.AddForce:

- Aplica una fuerza hacia arriba (y) usando la magnitud definida en jumpForce. Esto hace que el jugador salte.

Mecánica: Dash

Para terminar crearemos un “Dash” sencillo para agregarle más habilidades a nuestro personaje. Esto lo haremos creando una suerte de teletransporte o “Teleport” corto, siendo un método muy común que se acompaña con alguna animación o estela de particular para disimular.

```
    private Vector3 previousPosition; // Almacena la posición del objeto en el frame anterior
     public Vector3 currentDirection; // Dirección actual del movimiento
     void Start()  {
        // Inicializar la posición previa con la posición inicial del objeto
        previousPosition = transform.position;
     }
     void Update()  {
        CalculateDirection();
        Teleport();
     }

    void Teleport() {
      if (Input.GetKeyDown(KeyCode.E))  {
            Debug.Log("Dash");
            transform.position += currentDirection * 5;
            Debug.Log(currentDirection);
            }
     }
void CalculateDirection()  {
     // Calcular la dirección del movimiento
    Vector3 movement = transform.position - previousPosition;
     // Normalizar la dirección para obtener un vector unitario
     movement = new Vector3(movement.x, 0, movement.z);
     if (movement != Vector3.zero)   {
        currentDirection = movement.normalized;
     }
        // Actualizar la posición previa
        previousPosition = transform.position;
     }
```

Explicación del código

Este código implementa un sistema que calcula la dirección de movimiento de un objeto en Unity y permite al jugador realizar un **teletransporte instantáneo (dash)** en esa dirección al presionar la tecla **E**. A continuación, explicamos en detalle cada parte del código.

Componentes:

- **previousPosition**: Almacena la posición del objeto en el frame anterior. Es usada para calcular la dirección del movimiento.
- **currentDirection**: Representa la dirección normalizada del movimiento del objeto. Es el vector unitario que indica hacia dónde se mueve el objeto.

Inicialización (Start)

```
    void Start()
{
  previousPosition = transform.position;
}
```

Al iniciar el juego, se guarda la posición inicial del objeto en previousPosition para compararla en cada frame.

Cálculo de Dirección (CalculateDirection)

```
void CalculateDirection(){
  Vector3 movement = transform.position - previousPosition;
  movement = new Vector3(movement.x, 0, movement.z);
  if (movement != Vector3.zero)  {
    currentDirection = movement.normalized;
  }
  previousPosition = transform.position;}
```

Cálculo de Movimiento:

- La dirección del movimiento se calcula como la diferencia entre la posición actual (transform.position) y la posición anterior (previousPosition).
- La componente vertical (y) del movimiento se ignora, ya que solo se calculan movimientos en el plano horizontal (x y z).

Normalización:

- Si el objeto se mueve (movement != Vector3.zero), el vector de movimiento se normaliza. Esto asegura que currentDirection sea un vector unitario (magnitud de 1), que representa únicamente la dirección.

Actualización de previousPosition:

- Se guarda la posición actual como la nueva posición previa para usarla en el siguiente frame.

Teletransporte (Teleport)

```
void Teleport(){
  if (Input.GetKeyDown(KeyCode.E)){
    Debug.Log("Dash");
    transform.position += currentDirection * 5;
    Debug.Log(currentDirection); }}
```

Activación del Teletransporte:

- Si el jugador presiona la tecla E, se activa el teletransporte.

Cálculo de la Nueva Posición:

- El objeto se mueve en la dirección almacenada en currentDirection multiplicada por una distancia fija de 5 unidades.

Depuración:

- Usa Debug.Log para imprimir un mensaje en la consola y verificar la dirección en la que se realiza el teletransporte.

De esta forma, terminaremos con un personaje con un sistema de movimiento bastante completo.

Situación inicial en TechLab.

¡Bienvenido de nuevo al equipo de TalentoLab! Tras el primer día de introducción, ya se tiene listo un pedido urgente de un cliente. Se trata de un prototipo que servirá para demostrar la jugabilidad de un personaje en un entorno 3D.

El cliente, un inversionista exigente pero visionario, quiere asegurarse de que la base de control del personaje sea perfecta antes de aprobar el presupuesto para el resto del desarrollo. Por eso, debemos crear un personaje que pueda moverse con fluidez, saltar y detectar el suelo con precisión. Además, este personaje deberá ser compatible con un sistema de cámara dinámico que permita a los jugadores seguirlo mientras exploran el entorno 3D.

Ejercicios prácticos:

¡Es tu momento para brillar! Uno de nuestros desarrolladores, Giuseppe, desea que lo asistas con un demo. Si logramos impresionar con esto, podríamos desbloquear más recursos para el proyecto Nexus. Pero tené cuidado, porque cualquier detalle fuera de lugar podría costarnos su confianza.

Comencemos por darle vida a nuestro personaje en el mundo 3D. ¡Adelante!

Requerimientos:

1. Implementar controles para que el personaje pueda moverse en el mundo 3D. El cliente ha pedido que utilicemos el **Input System** más moderno para garantizar la mejor experiencia.
2. Simular movimiento realista mediante las leyes de la física, integrando componentes como el **Rigidbody** y aplicando **fuerzas**.
3. Diseñar un sistema de detección con **Raycast** para que el personaje pueda saltar únicamente cuando esté en el suelo.
4. Configurar una cámara que siga al personaje, asegurando que la perspectiva sea cómoda y funcional para el jugador.
5. Pensar, buscar y agregar 1 mecánica más a elección. Posibles ejemplos:
    a. Invocación de objetos como plataformas temporales, disparos, entre otros.
    b. Agacharse (bajando la Y del objeto al mantener apretado un botón).
    c. Sprint

Materiales y recursos adicionales.

Input.GetAxis
https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Input.GetAxis.html

RigidBody:
https://docs.unity3d.com/ScriptReference/Rigidbody.html

AddForce:
https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Rigidbody.AddForce.html

Raycast:
https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Physics.Raycast.html

Preguntas para reflexionar.

- ¿Qué otras aplicaciones puede tener el Raycast?
- ¿Qué mecánicas podríamos crear que se basen en juegos de plataforma?
- ¿Qué importancia tienen las mecánicas en el juego y su historia?

Próximos pasos.

En la siguiente clase profundizaremos en las plataformas para nuestro juego. Creamos diferentes tipos de ellas para poder mezclar y colocar en el proyecto según nos sea conveniente.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad