«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 08

Buenos Aires Ciudad

Clase N° 8 | Interacción con UI en 3D I

Temario:

- Implementación First Person
- Canvas en el espacio 3D.
- Ejemplo Práctico (Data Panel inGame)

Objetivos de la clase

Implementar una perspectiva en primera persona (First Person) en Unity.

- Configurar una cámara en primera persona para interactuar con un entorno 3D.

Configurar un Canvas en el espacio 3D.

- Explorar cómo integrar elementos de interfaz de usuario (UI) directamente en el espacio 3D de la escena.
- Ajustar la posición y escala del Canvas para mantener la claridad y legibilidad.

First Person View

El cambio a una **perspectiva en primera persona** representa un giro importante en la forma en que el jugador percibe el mundo. A diferencia de una vista en tercera persona o isométrica, la cámara ahora se convierte en los ojos del personaje, lo que **aumenta la inmersión** y el vínculo emocional con el entorno.

Esta perspectiva es ideal para:

- Exploración detallada de escenarios.
- Interacción directa con objetos 3D (paneles, dispositivos, puertas).
- Experiencias subjetivas (como miedo, tensión o misterio).

¿Qué implica técnicamente?

- Controlar el eje Y para que el **cuerpo del personaje rote horizontalmente**.
- Controlar el eje X para que la **cámara (ojos) rote verticalmente**.
- Integrar el movimiento del jugador con la dirección de la cámara.
- Ocultar el cursor y bloquearlo en el centro de la pantalla.

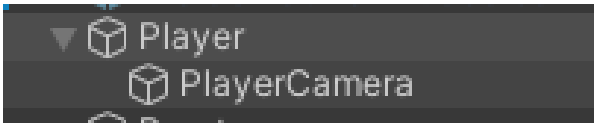
Implementación paso a paso

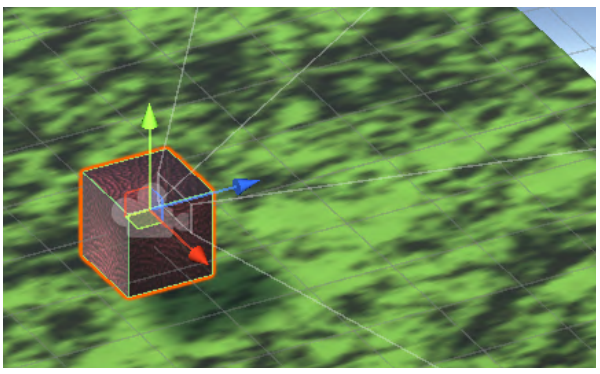
1. Estructura de GameObjects

Recomendamos la siguiente jerarquía para organizar el personaje:

```
Player (con Rigidbody + movimiento)
└── CameraHolder (vacio)
    └── Main Camera (posicionada como ojos)
```

Antes de iniciar, asegúrense de colocar una cámara bien alineada dentro del GameObject del player y desactivar la cámara que ya tenían.

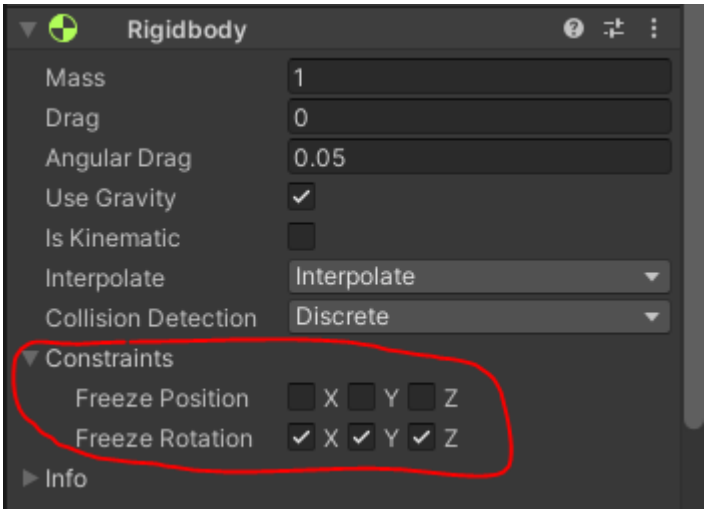




2. Bloquear rotaciones físicas en el Rigidbody

Para evitar que el jugador se incline por colisiones o movimientos bruscos, hacé lo siguiente:

- En el `Rigidbody`, activá las **Constraints**:
  Freeze Rotation: X, Y, Z ✅



Esto asegura que solo el código controle la orientación del personaje.

Script (Cámara)

```
public class CameraRotateScript : MonoBehaviour{
    private float x;
    private float y;
    public float sensitivity = -1f;
    private Vector3 rotate;
    // Start is called before the first frame update
    void Start() {
        Cursor.lockState = CursorLockMode.Locked;
    }
    // Update is called once per frame
    void Update(){
        y = Input.GetAxis("Mouse X");
        x = Input.GetAxis("Mouse Y");
        rotate = new Vector3(x, y * sensitivity, 0);
        transform.eulerAngles = transform.eulerAngles - rotate;
    }
}
```

Explicación paso a paso

En este código nos encontraremos con algunas cosas nuevas:

```
Cursor.lockState = CursorLockMode.Locked;
```

- **`Cursor`**: Es una clase estática en Unity que controla el comportamiento del cursor (mouse) en la ventana del juego.
- **`lockState`**: Es una propiedad de la clase `Cursor` que define cómo el cursor interactúa con la ventana del juego. Puede tomar valores de la enumeración `CursorLockMode`.
- **`CursorLockMode.Locked`**: Es un modo en el que el cursor está bloqueado en el centro de la pantalla y no es visible. Aunque el cursor está físicamente inmovilizado, el movimiento del mouse sigue capturándose, permitiendo al jugador, por ejemplo, rotar una cámara o un personaje en un juego en primera o tercera persona.

2. Movimiento del mouse

```
y = Input.GetAxis("Mouse X");
x = Input.GetAxis("Mouse Y");
```

- `Input.GetAxis("Mouse X")`: detecta el movimiento del mouse en el eje horizontal (izquierda/derecha).
- `Input.GetAxis("Mouse Y")`: detecta el movimiento vertical (arriba/abajo).

- El valor se guarda en las variables `x` y `y` respectivamente.

3. Cálculo del vector de rotación

```
rotate = new Vector3(x, y * sensitivity, 0);
```

- Se crea un vector de rotación con los movimientos del mouse.
- Se multiplica `y` (movimiento horizontal) por `sensitivity`.

🛠️ **Nota importante:** el valor de sensibilidad es negativo (`-1`) por defecto, lo que **invierte el sentido** de la rotación horizontal.

4. Aplicación de la rotación

```
transform.eulerAngles = transform.eulerAngles - rotate;
```

`transform.eulerAngles`:

- Representa las rotaciones del objeto en términos de ángulos de Euler (X, Y, Z) en grados.
- En este caso, toma los ángulos actuales del objeto (`transform.eulerAngles`) y les resta el valor calculado en `rotate`. Esto actualiza la rotación del objeto.

`transform.eulerAngles - rotate`:

- Resta el vector `rotate` para modificar la orientación del objeto:
    - **Eje X (`x`)**: Controla la inclinación (mirar hacia arriba o hacia abajo).
    - **Eje Y (`y * sensitivity`)**: Controla la rotación hacia los lados (girar hacia la izquierda o derecha).
    - **Eje Z (`0`)**: No se aplica ninguna rotación en este eje.

Con esto nuestra cámara estará lista para ser usada. Pero antes de dar Play, deberemos de chequear algo en nuestro Script del Player

Resultado

Con este script:

- La **cámara rota con el mouse** en primera persona.
- El movimiento es fluido y continuo.
- El jugador puede **mirar en todas las direcciones**, pero no mover aún su cuerpo o desplazarse (eso se trabaja aparte con el Rigidbody).

Script (Player)

En esta etapa conectamos el movimiento del personaje con la orientación de la cámara. Esto permite que **el jugador se desplace en la dirección en la que está mirando**, como es típico en juegos en primera persona.

Puede llegar a pasar que si probamos el juego, nuestro Player/cámara rote, pero no cambie su dirección de movimiento. Para solucionar esto, debemos modificar el Script de Movimiento de nuestro personaje haciendo que e Vector3 que era “movement” sea modificado por la función “TransformDirection”

```
float horizontalInput = Input.GetAxis("Horizontal");
float verticalInput = Input.GetAxis("Vertical");

Vector3 movement = new Vector3(horizontalInput, 0, verticalInput);
Vector3 newDirection = transform.TransformDirection(movement);

newDirection = newDirection.normalized;

rb.velocity = new Vector3(newDirection.x * moveSpeed,rb.velocity.y,
newDirection.z * moveSpeed);
```

Explicación paso a paso

1. Lectura de inputs de movimiento

```
float horizontalInput = Input.GetAxis("Horizontal");
float verticalInput = Input.GetAxis("Vertical");
```

- Estos métodos devuelven valores entre -1 y 1 según las teclas presionadas (por defecto: `A/D` y `W/S`).
- Esto representa **el movimiento local en el plano X-Z**: izquierda/derecha y adelante/atrás.

2. Vector de movimiento básico

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

- Se crea un vector con los valores de input, sin componente vertical (`Y = 0`), ya que en este sistema solo nos desplazamos en el plano horizontal.

3. Adaptar el vector a la rotación del jugador

```
Vector3 newDirection = transform.TransformDirection(movement);
```

- Este paso es clave: **transforma el vector de movimiento local en uno global**, según hacia dónde esté mirando el jugador.
- Gracias a esto, si el jugador gira la cámara a la derecha, el input de “adelante” lo moverá en esa dirección.

4. Normalizar la dirección

```
newDirection = newDirection.normalized;
```

- Esto asegura que el vector tenga una magnitud de 1.
  Evita que el jugador se mueva más rápido en diagonal (cuando se combinan dos direcciones).

5. Aplicar el movimiento al Rigidbody

```
rb.velocity = new Vector3(newDirection.x * moveSpeed,rb.velocity.y,
newDirection.z * moveSpeed);
```

- Multiplica la dirección por la velocidad deseada (`moveSpeed`).
- Se mantiene la componente vertical del `Rigidbody` (`rb.velocity.y`) para no interferir con efectos como saltos o gravedad.
- El resultado es un **movimiento suave, alineado con la dirección de la cámara**.

Resultado

Este sistema permite que el jugador:

- Se mueva en función del input WASD.
- Se desplace en la dirección en la que está mirando.
- Tenga un movimiento coherente con el entorno 3D y la rotación en primera persona

UI en el juego

Hasta ahora, las interfaces que hemos utilizado (HUD, score, vida, etc.) estaban siempre fijas en pantalla mediante un `Canvas` en modo **Screen Space**. En esta clase damos un paso más inmersivo: vamos a **incorporar elementos de UI directamente en el mundo 3D**, para que formen parte del entorno y puedan ser explorados o activados por el jugador.

¿Por qué usar UI en 3D?

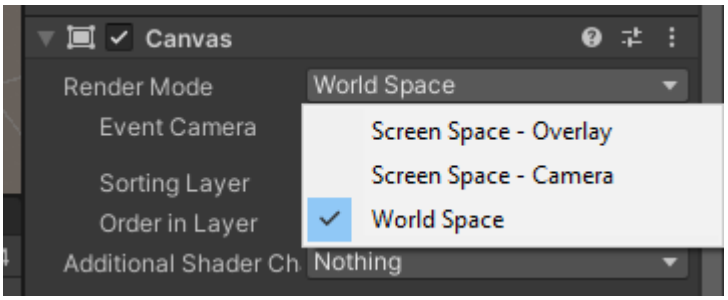
- Aumenta la **inmersión**: los elementos se perciben como objetos dentro del juego.
- Permite crear **interfaces contextuales**, como paneles informativos, indicadores físicos, botones flotantes, etc.
- Da lugar a nuevas formas de interacción: mirar, acercarse o activar elementos con teclas.

🎯*Ejemplo: un panel holográfico que muestra vida y energía en una estación médica dentro del escenario.*

Configuración de un Canvas en World Space

1. Crear el Canvas

- Ir a: `GameObject → UI → Canvas`.
- En el Inspector, cambiar el **Render Mode** de `Screen Space - Overlay` a `World Space`

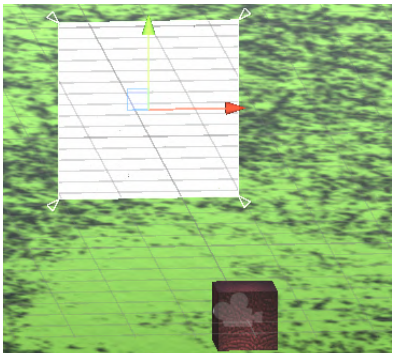


2. Ajustar su escala y posición

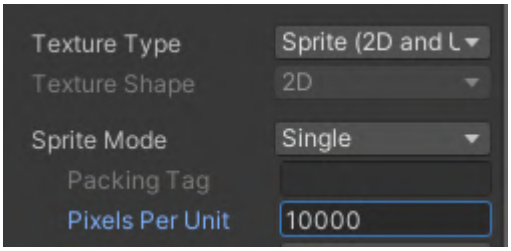
- Escalá el Canvas a valores adecuados (por ejemplo `0.01, 0.01, 0.01`) para que sea visible junto a objetos 3D.
- Posicionalo dentro del escenario, como si fuera parte del entorno (sobre una pared, mesa, etc.).

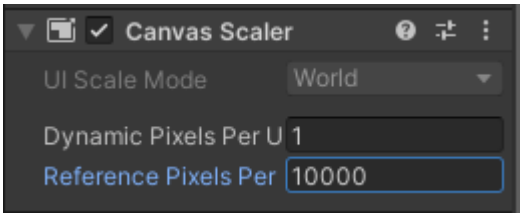
3. Añadir contenido al Canvas

- Crear una `Image` para que sirva de fondo (puede simular un cartel).
- Dentro de esa imagen, podés agregar texto, íconos, o elementos más complejos como barras de vida o botones.



- Si ven que su imagen se ve de una manera extraña prueben modificar la escala de píxeles tanto en la imagen como en el canvas que hicimos:





💡***Tip:*** *Si desean cambiar la escala, del canvas o la imagen, SIEMPRE asegúrense que sean la misma tanto el parent (canvas) como el child (imagen de fondo). Especialmente si desean rotarla.*

Data In-Game

Ahora que el Canvas está integrado dentro del mundo 3D, vamos a **convertirlo en un panel de datos** que el jugador pueda encontrar y consultar mientras explora el entorno.

La idea no es que el HUD esté siempre en pantalla, sino que el jugador **interactúe con paneles físicos en el juego** para acceder a información como vida, maná, energía, oro, etc.

> 🎯 *Este tipo de UI contextual no solo aporta información, sino que también construye narrativa, ambiente y realismo.*

Preparación visual del panel

1. Dentro del `Canvas`, agregá una `UI → Image` que actúe como fondo.
2. Dentro de esa imagen, creá 3 elementos `UI → Image` que simulen **barras** para:
    - Vida
    - Energía
    - Maná

3. Organizá cada barra como una sección separada.
4. Desactivá todas estas secciones al inicio. Solo una debe estar activa por vez.
5. Esto simulará un “panel digital” con varias pantallas que se pueden alternar.
6. Por ahora dejaremos las 3 deshabilitadas

Interacción: mostrar distintas pantallas con la tecla F

Vamos a crear un sistema en el que el jugador pueda cambiar entre las distintas secciones del panel presionando una tecla (F), pero solo si está cerca.

Script de navegación entre secciones

```
    private List<GameObject> children = new List<GameObject>();
    private int currentIndex = 0;
    private bool playerIn = false;
    private void Start(){
        // Llenar la lista con los hijos del panel
        foreach (Transform child in gameObject.transform){
            children.Add(child.gameObject);
        }

        // Asegurarse de que sólo el primer hijo esté activo al inicio
        UpdateVisibility();
    }

    private void Update(){
        if (Input.GetKeyDown(KeyCode.F) && playerIn){
            Debug.Log("Cambio");
            AdvanceSlide();
        }
    }

    private void AdvanceSlide(){
        // Ocultar el actual y avanzar al siguiente
    if (children.Count == 0) return; // Evitar errores si no hay hijos
       children[currentIndex].SetActive(false);
       currentIndex = (currentIndex + 1) % children.Count; // Ciclo
circular
        children[currentIndex].SetActive(true);
    }

    private void UpdateVisibility(){
        for (int i = 0; i < children.Count; i++){
            children[i].SetActive(i == currentIndex);
        }
    }
```

```
    private void OnTriggerEnter(Collider other){
        if (other.CompareTag("Player")){
            Debug.Log("Player Entro");
            playerIn = true;
        }
    }

    private void OnTriggerExit(Collider other){
        if (other.CompareTag("Player")){
            Debug.Log("Player salio");
            playerIn = false;
        }
    }
```

- **children:** Guarda los objetos hijos del panel (las pantallas de vida, maná, etc.).
- **currentIndex:** Indica qué sección está activa.
- **playerIn:** Detecta si el jugador está dentro del área de activación.
- **AdvanceSlide():** Cambia a la siguiente pantalla (circularmente).
- **OnTrigger…:** Habilita la interacción solo si el jugador está cerca.

Resultado

- Cuando el jugador entra en el área del panel, puede presionar F para alternar entre las pantallas.
- Solo se muestra una sección a la vez.
- Si se aleja, ya no puede interactuar.

Esto simula una interfaz futurista o contextual que podría encontrarse en una base, estación o edificio dentro del mundo de Nexus.

Materiales y recursos adicionales.

- Image:https://docs.unity3d.com/2022.3/Documentation/Manual/script-Image.html
- Canvas:https://docs.unity3d.com/es/2018.4/Manual/class-Canvas.html

Sumergido en 3D:

Con los cimientos del juego en su lugar, el cliente quiere que Nexus brinde una experiencia aún más inmersiva. Hasta ahora, el enfoque ha estado en mecánicas y sistemas generales, pero ahora es momento de situar al jugador dentro del mundo de Nexus de una manera más directa: *"Queremos que sientan que son parte del mundo, no solo observadores desde afuera."*

El equipo de TalentoLab recibe un nuevo desafío: implementar una perspectiva en primera persona y aprovechar el **Canvas en el espacio 3D** para crear elementos interactivos y visualmente integrados al entorno. Esto incluye la creación de un sistema que muestre información contextual al jugador en tiempo real, como un **Panel de Datos in-game**, que permita visualizar detalles importantes del entorno o del progreso de la partida.

Ejercicios prácticos:

La solicitud del cliente:
Después de revisar el primer prototipo del panel de datos en 3D, el equipo de Nexus Digital Studios envió una nueva solicitud.

Luigi comparte el siguiente mensaje del cliente:

"El panel de datos que han diseñado es un gran comienzo, pero necesitamos que sea más funcional e informativo. Queremos que refleje en tiempo real las estadísticas del jugador, como su vida, maná, energía y oro. Este detalle no solo añade inmersión, sino que también permite a los jugadores tomar decisiones estratégicas durante la partida."

Elizabeth, como coordinadora técnica, agrega una recomendación importante:

"El sistema debe ser dinámico y responder automáticamente a cualquier cambio en las estadísticas del jugador. No podemos depender de actualizaciones manuales o lentas."

Objetivo del ejercicio

Implementar un sistema de panel informativo que:

- Se visualice en el entorno 3D mediante un Canvas World Space.
- Se actualiza automáticamente en tiempo real.
- Esté conectado con las estadísticas del jugador.
- Utilice el sistema de EventManager para comunicar los cambios de estado.

Requisitos técnicos

- Conectar el panel a variables reales del jugador: vida, energía, maná, oro, etc.
- Escuchar eventos que reflejen cambios en esas variables.
- Actualizar el contenido del panel inmediatamente cuando:
    - El jugador pierde vida.
    - Recolecta monedas u oro.
    - Usa o recupera maná o energía.

🛑**Aclaración:** La información a mostrar en el panel es a gusto del estudiante

Consideraciones de diseño

- La presentación visual queda a elección del estudiante.
    - Puede usar barras, íconos, texto, gráficos, etc.
    - Puede mostrar todas las estadísticas en simultáneo o alternar entre pantallas.
- El sistema debe **funcionar sin necesidad de intervención directa**. El jugador no tiene que tocar nada para ver sus datos actualizados.

Tip para implementación

Podés reutilizar y adaptar:

- El `EventManager` genérico de la Clase 6.
- El `Canvas 3D` y sistema de cambio de pantalla de esta clase.
- Las estadísticas del jugador que ya estés manejando (por ejemplo, vida, oro o energía).

Consigna de Pre-Entrega de Proyecto

En esta clase nos encontramos con un momento clave en el desarrollo del proyecto: la **pre-entrega**. Será una oportunidad para mostrar todo lo que han avanzado hasta ahora, compartir su progreso y recibir devoluciones que les permitan ajustar detalles y afinar ideas antes de la entrega final.

Llegar a este punto implica reunir y organizar todo lo trabajado hasta la Clase 8. Es una excelente instancia para **consolidar el enfoque del proyecto**, asegurarse de que el camino recorrido es el adecuado, y preparar con solidez los próximos pasos. El objetivo es que puedan aprovechar la retroalimentación para **fortalecer el prototipo** y enfocarse en los aspectos esenciales de cara a la etapa final.

Requisitos de la entrega

La pre-entrega debe incluir todos los elementos desarrollados hasta ahora, organizados y funcionales. A continuación, se detallan los puntos obligatorios:

1. C# / Unity – Programación funcional

El proyecto debe demostrar aplicación concreta de los siguientes conceptos:

- Movimiento con `Input.GetAxis` y `Vector3`.
- Aplicación de físicas básicas (`Rigidbody`, `Force`, etc.).
- Implementación de **Eventos y Delegados** (`EventManager`, `Actions` o `Delegates`).
- Al menos una mecánica que utilice interacción con el entorno (UI 3D, Trigger, Pickup, etc.).

2. Modelos y Texturas

Debe estar presente la construcción básica del entorno inicial del juego:

- Modelos base (propios o descargados).
- Texturizado simple para distinguir superficies y objetos clave.
- Escenario jugable funcional (aunque no esté finalizado).

3. Mecánicas principales

El prototipo debe incluir al menos **dos** mecánicas core desarrolladas. Ejemplos:

- Plataformas móviles o que reaccionan.
- Dash o Double Jump.
- Teletransporte o cambio de zona.
- PickUps con lógica asociada.
- Invocaciones (`Summon/Invoke`) de objetos o enemigos.

4. Animaciones

- El personaje principal (jugador) debe tener implementadas **animaciones básicas** (idle, run, jump, etc.).
- Debe haber al menos un objeto extra con una animación loopeada (por ejemplo, una plataforma que sube y baja o un objeto flotante).

5. Game Design

Deben presentar un documento breve que explique las **decisiones de diseño** tomadas hasta el momento:

- ¿Cómo se plantea la curva de dificultad?
- ¿Qué feedback recibe el jugador?
- ¿Qué mecánicas están incluidas y por qué?

Este documento puede seguir el modelo trabajado en la Clase 7 e incluir recursos visuales si se desea (no obligatorio).

✅ Formato de entrega

- Carpeta del proyecto Unity (o build ejecutable si aplica).
- Documento de diseño (PDF o presentación).
- Video corto de demostración (opcional, pero recomendable).
- Archivo comprimido o link a carpeta compartida (Drive, OneDrive, etc.).

Plazo de entrega

Tendrán 7 días corridos para realizar la entrega desde la publicación de esta consigna.

Buenos Aires Ciudad