«Talento Tech»

Desarrollo de Videojuegos

# Unity 3D

Clase 10

Buenos Aires Ciudad

# Clase N° 10 | Data Persistence y PlayerPrefs (Script)

## Temario:

- Uso de PlayerPrefs para guardar datos simples.
- Corrutinas

---

## Objetivos de la clase

- Comprender el uso de PlayerPrefs para guardar y cargar datos simples.
- Implementar un sistema básico de guardado y carga de progreso.
- Introducir el concepto de Corrutinas para manejar procesos asíncronos en Unity.

# PlayerPrefbs

En esta clase veremos cómo guardar data de nuestro juego para volverla a usar más adelante. Una de las formas más sencillas es usando PlayerPrefbs.

`PlayerPrefs` es una clase en Unity que se utiliza para **almacenar datos simples de forma persistente** en el dispositivo del usuario. Los datos se guardan como pares clave-valor y permanecen disponibles incluso después de cerrar la aplicación. Es ideal para guardar configuraciones, progresos de juego o datos pequeños como el nivel alcanzado, la puntuación más alta o las preferencias de audio.

## ¿Para qué sirve PlayerPrefs?

1. **Guardar datos simples:**
    - Enteros (`int`), decimales (`float`) y cadenas de texto (`string`).
2. **Recuperar datos al iniciar el juego:**
    - Por ejemplo, cargar configuraciones de audio o el último nivel jugado.
3. **Reiniciar o borrar datos específicos:**
    - Útil para pruebas o si un jugador decide restablecer su progreso.

## Principales Métodos

**SetInt (guardar enteros):**

Guarda un valor entero asociado con una clave específica.

```
PlayerPrefs.SetInt("HighScore", 1000);
```

**Parámetros:**

- `"HighScore"`: Nombre de la clave que identifica el valor.
- `1000`: Valor entero a guardar.

**Ejemplo:** Guardar la puntuación más alta.

**GetInt (recuperar enteros):**

```
int highScore = PlayerPrefs.GetInt("HighScore", 0);
```

Parámetros:

- `"HighScore"`: Nombre de la clave.
- `0`: Valor por defecto si la clave no existe.

**Ejemplo:** Recuperar la puntuación más alta o usar `0` si no hay datos.

**SetFloat (guardar decimales):**

Guarda un valor decimal asociado con una clave.

```
PlayerPrefs.SetFloat("Volume", 0.75f);
```

Parámetros:

- `"Volume"`: Nombre de la clave.
- `0.75f`: Valor decimal a guardar.

**Ejemplo:** Guardar el nivel de volumen.

**GetFloat (recuperar decimales)**

Recupera un valor decimal asociado con una clave. Si la clave no existe, devuelve un valor por defecto.

```
float volume = PlayerPrefs.GetFloat("Volume", 1.0f);
```

**Parámetros:**

- `"Volume"`: Nombre de la clave.
- `1.0f`: Valor por defecto si la clave no existe.

**Ejemplo:** Recuperar el nivel de volumen guardado o usar `1.0f` como predeterminado.

**SetString (guardar cadenas de texto)**

Guarda una cadena de texto asociada con una clave.

```
PlayerPrefs.SetString("PlayerName", "JohnDoe");
```

**Parámetros:**

- `"PlayerName"`: Nombre de la clave.
- `"JohnDoe"`: Cadena de texto a guardar.

**Ejemplo:** Guardar el nombre del jugador.

**GetString (recuperar cadenas de texto)**

Recupera una cadena de texto asociada con una clave. Si la clave no existe, devuelve un valor por defecto.

```
string playerName = PlayerPrefs.GetString("PlayerName", "Guest");
```

**Parámetros:**

- `"PlayerName"`: Nombre de la clave.
- `"Guest"`: Valor por defecto si la clave no existe.

**Ejemplo:** Recuperar el nombre del jugador guardado o usar `"Guest"` como predeterminado.

**Save (guardar en disco)**

Asegura que todos los datos guardados con `PlayerPrefs` se escriban en el disco inmediatamente.

```
PlayerPrefs.Save();
```

- **Uso típico:** Llamar a `Save` después de usar `SetInt`, `SetFloat` o `SetString` para asegurarte de que los datos no se pierdan en caso de cierre inesperado.

**DeleteKey (borrar una clave específica)**

Elimina los datos asociados con una clave.

```
PlayerPrefs.DeleteKey("HighScore");
```

**Parámetro:**

- `"HighScore"`: Nombre de la clave a eliminar.

**Ejemplo:** Reiniciar la puntuación más alta.

**DeleteAll (borrar todos los datos)**

Elimina todos los datos guardados en `PlayerPrefs`.

```
PlayerPrefs.DeleteAll();
```

- **Uso típico:** Borrar todos los datos durante el desarrollo o al implementar una opción de "reinicio completo" en el juego.

**HasKey (verificar si existe una clave)**

Comprueba si existe una clave específica en `PlayerPrefs`.

```
if (PlayerPrefs.HasKey("HighScore")) {

    Debug.Log("High Score existe");

}
```

**Parámetro:**

- `"HighScore"`: Nombre de la clave.

**Retorna:** `true` si la clave existe, `false` de lo contrario.

**Ejemplo:** Comprobar si hay datos guardados antes de intentar cargarlos.

## Limitaciones de PlayerPrefs:

1. **Solo para datos simples:** No puede almacenar estructuras complejas como listas, arreglos o clases. Para eso, usa JSON, binarios o bases de datos.
2. **Tamaño limitado:** Dependiendo de la plataforma, el espacio para datos puede ser limitado.
3. **No seguro:** Los datos no están encriptados, lo que significa que se pueden manipular fácilmente.

# Ejemplo Práctico de PlayerPrefs

Crearemos un Script sencillo aplicado al concepto de Score para relacionarlo fácilmente con lo que veníamos trabajando

Si tenían creado un **Score Manager** de la **clase 6**, pueden añadirle las variables y funciones de este ejemplo:

```
    public class ScoreManager : MonoBehaviour{
    private string scoreKey = "HighScore";
    void Start()
    {
        // Cargar el High Score al inicio del juego
        int highScore = PlayerPrefs.GetInt(scoreKey, 0); // 0 por
defecto
        Debug.Log("High Score cargado: " + highScore);
    }

    public void UpdateScore(int currentScore){
        // Comparar el score actual con el High Score guardado
        int highScore = PlayerPrefs.GetInt(scoreKey, 0);
        if (currentScore > highScore)
        {
            // Guardar el nuevo High Score
            PlayerPrefs.SetInt(scoreKey, currentScore);
            PlayerPrefs.Save(); // Asegura que se guarde inmediatamente
            Debug.Log("Nuevo High Score guardado: " + currentScore);
        }
        else
        {
     Debug.Log("No se superó el High Score. High Score actual: " +
highScore);
        }
    }

    public void ResetScore(){
        // Reiniciar el High Score
        PlayerPrefs.DeleteKey(scoreKey);
        Debug.Log("High Score reiniciado.");
    }
}
```

Crea una función que actualiza el Score más alto y otra que la resetea. Si lo desglosamos seria algo asi:

**Definición de variables y clave:**

```
    private string scoreKey = "HighScore";
```

- **`scoreKey`**: Es una cadena que representa la clave utilizada en `PlayerPrefs` para guardar y recuperar el valor del **High Score**.
- `"HighScore"`: Es el nombre que identifica este dato específico.

**Método `Start`:**

```
    void Start() {
        // Cargar el High Score al inicio del juego
      int highScore = PlayerPrefs.GetInt(scoreKey, 0); // 0 por defecto
        Debug.Log("High Score cargado: " + highScore);
    }
```

- **Propósito:** Al iniciar el juego, este método carga el **High Score** previamente guardado para mostrarlo o usarlo.
- **`PlayerPrefs.GetInt(scoreKey, 0)`**: Busca el valor asociado con la clave `"HighScore"`. Si no existe, devuelve `0` como valor por defecto.
- **`Debug.Log(...)`**: Imprime el valor del **High Score** en la consola de Unity, útil para verificar que se cargó correctamente.

**Método `UpdateScore`:**

```
    public void UpdateScore(int currentScore) {
        // Comparar el score actual con el High Score guardado
        int highScore = PlayerPrefs.GetInt(scoreKey, 0);
        if (currentScore > highScore) {
            // Guardar el nuevo High Score
            PlayerPrefs.SetInt(scoreKey, currentScore);
            PlayerPrefs.Save(); // Asegura que se guarde inmediatamente
            Debug.Log("Nuevo High Score guardado: " + currentScore);
        }
        else {
        Debug.Log("No se superó el High Score. High Score actual: " +
highScore);
        }
    }
```

**Carga el High Score actual:**

- Recupera el valor guardado con la clave `"HighScore"`. Si no existe, devuelve `0`.

**Compara el puntaje actual (`currentScore`) con el High Score guardado:**

- Si el puntaje actual es mayor, el High Score se actualiza:
  - **`PlayerPrefs.SetInt(scoreKey, currentScore)`**: Guarda el nuevo High Score.
  - **`PlayerPrefs.Save()`**: Escribe los datos en disco para asegurarse de que persistan.
  - Imprime un mensaje en la consola indicando que el High Score ha sido actualizado.
- Si el puntaje actual no supera el High Score, muestra un mensaje indicando que no se superó.

**Método `ResetScore`:**

```
public void ResetScore()
{
    // Reiniciar el High Score
    PlayerPrefs.DeleteKey(scoreKey);
    Debug.Log("High Score reiniciado.");
}
```

**Eliminar la clave `scoreKey`:**

- **`PlayerPrefs.DeleteKey(scoreKey)`**: Borra el valor asociado con `"HighScore"`. Esto restablece el High Score al valor por defecto (en este caso, `0`).

**Imprime un mensaje en la consola:**

- Indica que el High Score ha sido reiniciado.

Si ahora generan alguna situación para llamar a la funcion “UpdateScore” notaran que cuando le dan “Play” y detienen el juego, en la siguiente “prueba” que hagan los valores anteriores se habrán guardado.

# Corrutinas

Para finalizar la clase vamos a ver una introducción a Corrutinas. Este es un método especial que permite ejecutar código de manera asíncrona o pausada a lo largo de varios frames sin detener el flujo principal del juego. Esto es útil para realizar tareas que deben ejecutarse con un retraso en el tiempo o en intervalos, como animaciones, temporizadores o ciclos con pausas.

En Unity, se define usando el tipo de retorno **IEnumerator** y se invoca con el método **StartCoroutine**.

## ¿Para qué sirve?

- **Realizar tareas que necesitan pausas** (esperar un tiempo antes de realizar algo).
- **Controlar flujos de tiempo** como temporizadores, cooldowns, o efectos por turnos.
- **Secuencias complejas** (encadenar acciones como animaciones o eventos).

## Ejemplo:

```
public class SimpleCoroutine : MonoBehaviour
{
    void Start()
    {
        // Inicia la corrutina cuando comienza el juego
        StartCoroutine(ShowMessagesWithDelay());
    }

    IEnumerator ShowMessagesWithDelay()
    {
        Debug.Log("Mensaje 1: Hola, esto es una corrutina.");

        // Espera 3 segundos
        yield return new WaitForSeconds(3);

        Debug.Log("Mensaje 2: Pasaron 3 segundos.");
    }
}
```

## Explicación paso a paso:

**Invocar la Corrutina**

```
StartCoroutine(ShowMessagesWithDelay());
```

- Llamamos a la corrutina desde `Start()`. Esto asegura que se ejecute cuando el juego comienza.
- `StartCoroutine` es obligatorio para que Unity maneje correctamente el comportamiento pausado.

**Definir la Corrutina**

```
IEnumerator ShowMessagesWithDelay()
```

- El método `ShowMessagesWithDelay` devuelve `IEnumerator`. Esto indica que es una corrutina y puede incluir pausas (`yield`).

**Primer mensaje**

```
Debug.Log("Mensaje 1: Hola, esto es una corrutina.");
```

- Imprime un mensaje en la consola para indicar que la corrutina ha comenzado.

**Esperar 3 segundos**

```
yield return new WaitForSeconds(3);
```

- Aquí la corrutina "se detiene" durante 3 segundos.
- Durante este tiempo, el juego sigue funcionando (no se congela).

**Segundo mensaje**

```
Debug.Log("Mensaje 2: Pasaron 3 segundos.");
```

- Una vez que pasa el tiempo de espera, se ejecuta el resto del código de la corrutina.

## Resultado en la consola:

```
Mensaje 1: Hola, esto es una corrutina.
(espera 3 segundos)
Mensaje 2: Pasaron 3 segundos.
```

# Ejemplo Corrutinas CoolDown

Esta herramienta que nos permite dar un tiempo de espera es perfecta para crear un sistema de CoolDowns.
Un código para eso, se veria de la siguiente manera:

```
public class CoolDownExample : MonoBehaviour
{
    public float coolDownTime = 3f; // Tiempo de cooldown (en segundos)
    private bool isCoolingDown = false;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space) && !isCoolingDown)
        {
            // Acción principal
            Debug.Log("¡Habilidad activada!");

            // Inicia el cooldown
            StartCoroutine(CoolDownCoroutine());
        }
    }

    IEnumerator CoolDownCoroutine()
    {
        isCoolingDown = true; // Inicia el estado de cooldown
        Debug.Log("Entrando en cooldown...");

        // Espera el tiempo de cooldown
        yield return new WaitForSeconds(coolDownTime);

        isCoolingDown = false; // Finaliza el cooldown
        Debug.Log("Cooldown terminado. Puedes usar la habilidad de
nuevo.");
    }
}
```

Donde, “**CoolDownTime**” es la variable que determinará el tiempo de espera para volver a usar la habilidad. Y usaremos un **bool** como “**isCoolingDown**” para que chequee/revise la disponibilidad para utilizarla.
Cuando se use la habilidad, llama a la corrutina que vuelve “**true**” al **bool**, evitando que pueda volver a usarla.  Al pasar el tiempo estimado en “**WaitForSecond(CoolDownTime)**” esta variable vuelve a ser **false** y la condición de uso vuelve a ser verdadera.

# Guardando el progreso:

Con el universo de **Nexus** cobrando forma, la dirección de Talento Lab destaca un nuevo aspecto crucial del desarrollo: la capacidad de **guardar progreso** y gestionar dinámicas en tiempo real que añadan fluidez a la jugabilidad.

En una reunión reciente, el cliente plantea un escenario común para los videojuegos:

> *"Nuestro objetivo no es solo crear un mundo inmersivo, sino también garantizar que los jugadores puedan regresar a él con su progreso intacto. Además, queremos integrar mecánicas temporales, como habilidades con tiempos de recarga, que añadan profundidad a la jugabilidad."*

Para lograr esto, el equipo de **TalentoLab** deberá dominar dos herramientas esenciales:

- **PlayerPrefs:** Un sistema para guardar y recuperar datos, como el puntaje del jugador o su progreso en el nivel.
- **Corrutinas:** Un mecanismo para manejar acciones que ocurren durante un período, como cooldowns para habilidades o animaciones secuenciales.

# Ejercicios prácticos:

El equipo de desarrolladores de TalentoLab, Roberta y Giuseppe, se reúne para trabajar en las dos nuevas funcionalidades planteadas por el cliente. "Para abordar estos aspectos, se les asignan dos tareas prácticas:

1. **Guardado de datos con PlayerPrefs:**
   Se solicita al equipo que implemente un sistema básico para almacenar información crítica del jugador, como su vida, puntaje, maná u oro.

   **El cliente destaca:**
   *"Imaginemos que el jugador termina una sesión después de recolectar mucho oro o alcanzar un récord de puntaje. La próxima vez que entre, debe sentir que su esfuerzo se valora al ver esos datos intactos."*

Esta tarea permitirá a los desarrolladores experimentar con la persistencia de datos y preparar la base para sistemas más complejos en el futuro.

2. **CoolDown con Corrutinas:**
   En paralelo, el equipo trabajará en integrar un **sistema de recarga temporal** para habilidades o movimientos específicos, como el Dash, el salto doble o incluso el movimiento de plataformas.

   **El cliente explica:**
   *"Queremos que las habilidades especiales tengan un impacto significativo, pero también que los jugadores tengan que gestionar su uso con cuidado. Los cooldowns añaden estrategia, tensión y ritmo al juego."*

   Los desarrolladores deberán utilizar corrutinas para crear estas dinámicas temporales de manera fluida, asegurándose de que sean funcionales y visualmente claras para el jugador.

---

# Materiales y recursos adicionales.

Corrutinas:
https://docs.unity3d.com/es/2021.1/Manual/Coroutines.html

PlayerPrefs:
https://docs.unity3d.com/2022.3/Documentation/ScriptReference/PlayerPrefs.html

---

# Preguntas para reflexionar.

1. ¿En qué situaciones podemos usar un CoolDown?
2. ¿Todos los juegos guardan los datos de los jugadores?
3. ¿En qué situaciones necesitamos guardar datos y cuales?

---

# Próximos pasos.

En la próxima clase avanzaremos con el GameDesign profundizando en algunos conceptos y descubriendo nuevos.


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad