

«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 10



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

Temario:

- Uso de PlayerPrefs para guardar datos simples.
- Corrutinas


Objetivos de la clase

En esta clase vamos a explorar dos herramientas fundamentales para enriquecer la experiencia del jugador: el **guardado de datos persistentes** y el **control del tiempo con corrutinas**. Aprenderemos a utilizar **PlayerPrefs** para conservar información clave como el puntaje o el estado del jugador, incluso después de cerrar el juego. Además, implementaremos **corrutinas**, que nos permitirán controlar secuencias temporales dentro del gameplay, como enfriar habilidades o mostrar mensajes en pantalla. Estas herramientas no solo mejoran la funcionalidad del proyecto, sino que aportan cohesión y profundidad a la jugabilidad en *Nexus*.

Guardado de Datos con **PlayerPrefs**

PlayerPrefs es una herramienta de Unity que permite **guardar datos simples de forma persistente**, permaneciendo disponibles incluso después de cerrar el juego, lo que lo hace Ideal para almacenar:

- Puntuaciones
- Configuraciones
- Preferencias del usuario
- Progreso de juego básico
- Datos de nivel o puntuaciones

 Imaginemos que el jugador recolecta oro o alcanza un récord: cuando vuelva a entrar, debe encontrar esos datos intactos.

Entonces, ¿Cómo funciona PlayerPrefs?

1. **Guarda datos simples:**
 - Enteros (int), decimales (float) y cadenas de texto (string).
2. **Recupera datos al iniciar el juego:**
 - Por ejemplo, cargar configuraciones de audio o el último nivel jugado.
3. **Reinicia o borra datos específicos:**
 - Útil para pruebas o si un jugador decide restablecer su progreso.
4. **Sintaxis clave-valor:**
 - PlayerPrefs.metodo("clave",valor);

Dónde **método**, **clave** y **valor** son datos a modificar según corresponda.

Principales Métodos

SetInt

Guarda un valor entero asociado con una clave específica.

Por ejemplo:

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

Parámetros:

- Clave que identifica al valor: "HighScore".
- Valor entero a guardar: 1000.

Ejemplo de uso: Guardar la puntuación más alta.

GetInt

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

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

Parámetros:

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

Ejemplo de uso: Recuperar la puntuación más alta o usar 0 si no hay datos.

SetFloat

Guarda un valor decimal asociado con una clave.

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

Parámetros:

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

Ejemplo de uso: Guardar el nivel de volumen.

GetFloat

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:

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

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

SetString

Guarda una cadena de texto asociada con una clave.

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

Parámetros:

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

Ejemplo de uso: Guardar el nombre del jugador.

GetString

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:

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

Ejemplo de uso: Recuperar el nombre del jugador guardado o usar "Guest" como predeterminado.

Save

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

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

Ejemplo de uso: 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

Elimina los datos asociados con una clave.

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

Parámetros:

- Nombre de la clave a eliminar: "HighScore"

Ejemplo de uso: Reiniciar la puntuación más alta.

DeleteAll

Elimina todos los datos guardados en PlayerPrefs.

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

Ejemplo de uso: Borrar todos los datos durante el desarrollo o al implementar una opción de "reinicio completo" en el juego.

HasKey

Comprueba si existe una clave específica en PlayerPrefs.

```
if (PlayerPrefs.HasKey("HighScore")) {  
    Debug.Log("High Score existe");  
}
```

Parámetro:

- Nombre de la clave: "HighScore"

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

Ejemplo de uso: 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.

Vamos a implementar PlayerPrefs

En este caso crearemos un sistema sencillo de guardado de puntaje, utilizando **PlayerPrefs**, que les permitirá guardar el **mayor puntaje alcanzado** y mantenerlo incluso después de cerrar el juego.

Si ya tenías un ScoreManager desde la clase 6, podés reutilizarlo y agregarle estas variables y funciones.

```
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.");
}
}

```

¿Qué hace este script?

- **Start():** Carga el récord anterior desde memoria persistente.
- **UpdateScore():** Suma puntos y verifica si es un nuevo récord.
- **ResetScore():** Borra el puntaje guardado para empezar de cero.

Explicación paso a paso:

Definición de variables y clave:

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

- **scoreKey:** Es una variable 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);
}

```

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);
    }
}
}

```

Este método recupera el valor guardado con la clave "HighScore" pero 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.

Probando

Ahora asignamos el script a nuestro Game Manager persistente, (si aún no lo tenemos lo creamos) y generamos alguna situación para llamar a la función "UpdateScore": En nuestro caso ganaremos o perderemos puntaje. Notaremos que cuando presionamos "Play" y se

detiene el juego, en la siguiente prueba que se haga los valores anteriores se habrán guardado.

Corrutinas

Para finalizar la clase vamos a introducirnos en las Corrutinas.

Una **corrutina** en Unity es un método especial que permite ejecutar acciones **con pausas o demoras en el tiempo**, sin congelar todo el juego. Es ideal para:

- Temporizadores y cooldowns (período de tiempo que debe transcurrir después de usar una habilidad, acción o ítem antes de que pueda ser utilizado nuevamente)
- Esperas entre eventos (como mostrar mensajes secuenciales)
- Efectos visuales en cadena
- Activación progresiva de objetos o acciones
- Secuencias complejas como encadenar acciones, animaciones o eventos.

En *Nexus*, por ejemplo, una corrutina podría controlar el **cooldown de un dash (movimiento rápido y breve)**, el **tiempo de espera entre invocaciones** o el **retraso de aparición de plataformas**.

En Unity, para utilizar una corrutina, se define el método con el tipo de retorno **IEnumerator** y se invoca utilizando **StartCoroutine**.

Sintaxis de su definición básica:

```
IEnumerator MiCorrutina() {  
    Debug.Log("Inicio");  
    yield return new WaitForSeconds(3f); // Espera 3 segundos  
    Debug.Log("Fin");  
}
```

Explicacion:

- **IEnumerator**: tipo que permite que la función se comporte como una secuencia que se puede pausar y reanudar.
- **yield return**: indica un punto de espera.
- **WaitForSeconds(3f)**: pausa la ejecución durante 3 segundos

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.");
    yield return new WaitForSeconds(3); // Espera 3 segundos
    Debug.Log("Mensaje 2: Pasaron 3 segundos.");
}
}
```

Explicación paso a paso:

● Invocar la Corrutina

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

- Esta línea se ejecuta dentro del método Start(), lo que significa que la corrutina comenzará apenas inicie el juego.
- El método StartCoroutine es necesario para que Unity pueda ejecutar una función IEnumerator de manera especial, permitiendo pausas sin frenar el juego.

🔧 Definir la Corrutina

```
IEnumerator ShowMessagesWithDelay()
```

- El método *ShowMessagesWithDelay* método devuelve IEnumerator, lo que le permite incluir instrucciones como yield return que pausan temporalmente su ejecución.
- Unity reconocerá esta función como una corrutina gracias a ese tipo de retorno.

🧠 Mostrar el primer mensaje

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

- Se imprime un mensaje en la consola.
- Sirve para indicar que la corrutina comenzó a ejecutarse correctamente.

⌚ Esperar 3 segundos

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

- Aquí es donde ocurre la "magia": la corrutina se pausa durante 3 segundos.
- A diferencia de un **sleep** tradicional, el resto del juego sigue corriendo normalmente.

Mostrar el segundo mensaje

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

- Una vez que se cumple el tiempo de espera, la corrutina **retoma la ejecución** desde donde la dejó.
- Se imprime otro mensaje para confirmar que pasaron los 3 segundos.

Resultado en la consola:

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

Ejemplo práctico: cooldown de habilidad

Una de las formas más útiles y divertidas de aplicar corrutinas es para crear **sistemas de recarga de habilidades**, conocidos como *cooldowns*. Esta técnica es fundamental en muchos géneros, como shooters, RPGs o plataformas, donde hay que **esperar un tiempo antes de volver a usar una acción especial**.

Imaginá que querés implementar un **dash** (un movimiento rápido hacia adelante), pero no querés que el jugador lo use sin parar. Querés que pueda hacerlo, sí... ¡pero solo cada 3 segundos! Ahí es donde entra en juego nuestra corrutina.

Vamos a crear una lógica que:

1. Detecta cuándo el jugador presiona la barra espaciadora.
2. Ejecuta la acción (el dash).
3. Activa un cooldown durante el cual la habilidad no se puede volver a usar.
4. Y pasado ese tiempo, la reactiva automáticamente.

```
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.");
    }
}

```

✖ Explicación paso a paso:

1. Variables clave:

- **coolDownTime:** determina cuántos segundos debe esperar el jugador antes de volver a usar la habilidad.
- **isCoolingDown:** indica si estamos en medio del cooldown. Si es true, la habilidad no se puede usar.

2. En el método Update():

- Se detecta si el jugador presionó la barra espaciadora.
- Si la habilidad está disponible (!isCoolingDown), se activa y se lanza la corrutina.

3. La corrutina CoolDownCoroutine():

- Pone isCoolingDown en true para bloquear temporalmente la acción.
- Usa yield return new WaitForSeconds(coolDownTime) para **esperar sin congelar el juego**.
- Al terminar el tiempo de espera, vuelve a poner isCoolingDown en false y la habilidad queda disponible otra vez.

✨ ¡Y listo! Este patrón se puede adaptar fácilmente para cualquier habilidad con recarga: disparos, saltos especiales, poderes mágicos. Es ideal para mantener el juego balanceado y darle un ritmo más dinámico.

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>

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 **Talento Lab** 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.



Buenos Aires
aprende
Agencia de Habilidades para el Futuro

BA Buenos
Aires
Ciudad