PROYECTO FINAL DE GRADO

Desarrollo de un videojuego con Unity

Alicia Herrera Sánchez
2ºDAM, 2025.

ÍNDICE

1. Introducción

Contextualización

Desde hace unas décadas y cada vez más el entretenimiento de jóvenes (y ahora también no tan jóvenes) han sido y suelen ser los videojuegos. Ya sean en PC, consolas, consolas portátiles o dispositivos móviles, estos videojuegos han tenido la capacidad de engancharnos durante horas alejándonos del mundo real. Esto, desde 2020, ha sido de vital importancia para sobrellevar el confinamiento casi a nivel mundial que la pandemia del coronavirus obligó a instaurar.

Este hecho provocó que hubiera un cambio en la manera de socializar de las personas, por ello, muchos optaron por socializar con sus amigos mediante los videojuegos. A través de los videojuegos se abrió una vía de escape del encierro que estábamos viviendo, así como un entretenimiento para pasar las horas más rápido.

La Asociación Española de Videojuegos afirma que esta industria equivale a un 0'11% del PIB español, además añade, que la industria del videojuego da empleo a unas 9.000 personas, creando impacto sobre la economía de 3.577 millones de euros

El tipo de videojuegos que más ha crecido ha sido el multijugador, ya que a través de esta opción se puede entablar conversación con otras personas, ya seas amigos o desconocidos.

Desde la plataforma de venta digital por excelencia, Steam, se anunció un nuevo récord con más de 20 millones de usuarios que accedieron a ella en el inicio de la pandemia. De estos 20 millones, más de 6'5 millones jugaron de forma simultánea

Además, según la compañía Telefónica, entre el 13 y el 15 de marzo de 2020, el tráfico de gaming aumentó un 271% con respecto a la semana anterior, procediendo principalmente de PlayStation y Xbox, además de las descargas de juegos en móviles y otros dispositivos.

Unity

La empresa Unity Technologies fue fundada en 1988 por David Helgason (CEO), Nicholas Francis (CCO), y Joachim Ante (CTO) en Copenhague, Dinamarca después de su primer juego,

GooBall, que no obtuvo éxito. Los tres reconocieron el valor del motor y las herramientas de desarrollo que habían creado y se dispusieron a hacer un motor que cualquiera pudiera usar a un precio asequible.

El éxito de Unity se debe a su enfoque en cubrir las necesidades de los desarrolladores independientes, los cuales normalmente no pueden permitirse crear su propio motor de juegos, ni pagar el acceso a las herramientas necesarias. La compañía busca "democratizar" el desarrollo de videojuegos y hacer que el desarrollo de contenido interactivo 2D y 3D sea lo más accesible a tantas personas en todo el mundo como sea posible.

En 2008, ya contaba con una base de 1 millón de programadores y actualmente, según

datos de su propia página, ya cuenta con 3.000 millones de usuarios.

Hoy en día, Unity3D es uno de los motores de creación de juegos más usados en la actualidad. Más del 50% de juegos tanto para móvil, ordenador y otros dispositivos están realizados con Unity. Dentro de las funcionalidades típicas que tiene, podemos encontrar:

- Motor gráfico para renderizar gráficos en 2D y 3D
- Motor físico que simula las leyes de la física
- Animaciones
- Sonidos
- IA
- Scripting.

Monobehaviour, GameObjects y Prefabs

Monobehaviour: Es una serie serializable, padre de todos los scripts que se vayan añadir como componentes y da acceso a los métodos y eventos de la aplicación (como los métodos Start(), Update(), eventos de teclado y ratón y métodos como onDestroy().

Estos métodos serán imprescindibles en los scripts que conforman este proyecto.

El **GameObject** es el concepto más importante en el editor de Unity. Cada objeto en su juego es un GameObject, desde personajes y objetos, hasta luces, cámaras y efectos especiales. Sin embargo, un GameObject no puede hacer nada por sí mismo. Se necesita darle propiedades antes de que pueda convertirse en un personaje, un entorno o un efecto especial.

Un GameObject tiene un Tag asignado que permite distinguirlo de otros GameObjects de la escena, listarlo utilizando ese Tag o realizar alguna función si el objeto tiene asignado un determinado Tag. En cuanto a este juego, esto es realmente útil para diferenciar enemigos móviles de estáticos, para poder asignarles comportamientos distintos sin tener que ir uno por uno, sino “agrupándolos”.

Por otro lado, los **prefabs** son objetos reutilizables, y creados con una serie de características dentro de la vista proyecto, que serán instanciados en el videojuego cada vez que sea conveniente y tantas veces como sea necesario. Es decir, nos permiten crear 'copias' fácilmente con una serie de ventajas:

- Es posible diferenciar qué objetos están conectados a un prefab desde la jerarquía de objetos (resaltados en color azul).
- Permite modificar los valores de un elemento del prefab instanciado, no afectando al resto de instancias implementadas, y sin romper la conexión con el prefab creado en los recursos del proyecto
- Si se modifican las propiedades del prefab creado, y este es instanciado en diferentes zonas de la escena, los cambios realizados afectarán a todas las instancias implementadas, sin necesidad de tener que modificar de manera individual cada instancia

2. Justificación y objetivos

El objetivo principal de este proyecto es conocer la herramienta de desarrollo Unity así como conseguir desarrollar un videojuego en 2D que sea programado y diseñado en su parte más esencial y básica. No se pretende reunir un gran conjunto o una gran variedad de escenarios o detalles que podría disponer un videojuego comercial completo ya que no es posible en tiempo y forma, ni tampoco en forma de recursos ya que este trabajo lo estoy llevando a cabo de manera unipersonal.

Respecto al alcance que podría llegar a tener en vistas de futuro y post proyecto o después de cumplido el tiempo de trabajo de este módulo proyecto es el de catalogado como juego "indie" cumpliendo con las premisas mínimas para ser considerado como videojuego completo y comercial, añadiendo variedad en forma y contenido tanto mayor número de complementos como variedad en diferentes tipos de juego y escenarios.

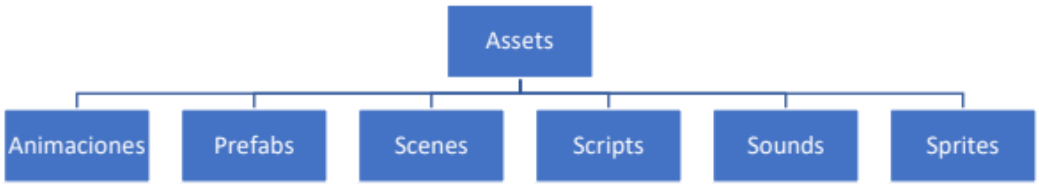
3. Aspectos generales

Metodología

Uno de los puntos más importantes para la realización de este pequeño trabajo, ha sido la fijación de pequeños objetivos diarios a fin de avanzar poco a poco y de manera más segura. Esto lo apliqué al comienzo del desarrollo, empezando por la realización de un par de cursos de desarrollo de videojuegos con Unity3D para poder comenzar con el desarrollo de una pequeña demo de videojuego.

Estructura

En el siguiente mapa conceptual se detalla cómo está dividido este trabajo. Cada elemento corresponde a una carpeta del proyecto.



En los **Assets** (recursos) se encuentran el resto de las carpetas que contienen los elementos que dan funcionalidad al trabajo. Así:

- Dentro de las animaciones se encuentran las animaciones del personaje principal.
- Dentro de los prefabs se encuentran los prefabs que la aplicación va generando como el suelo y los enemigos, en este caso obstáculos (piedra1, piedra2, enemigo móvil).
- La carpeta Scenes contiene las escenas de las que está compuesto el juego. Estas serán los niveles, el menú de selección de los mismos y la elección de perfil de jugador.

En la carpeta **Scripts** encontraremos los scripts es decir el código que da funcionalidad al juego. En este caso hay diez:

- GameManager: Probablemente sea el más importante. Es compartido por los 3 niveles, ya que, desde mi punto de vista, mejora mucho la eficiencia el hecho de tener el script compartido, y no código duplicado. La manera en la que lo he logrado compartir, es poniendo ciertas variables y métodos públicos, lo que permite su modificación en el inspector. De esta manera, en cada escena, las variables cambian según lo he considerado necesario, y lo guardo en la propia escena.

  Este script coordina todo el flujo de juego en el runner, desde la generación y desplazamiento del terreno y los obstáculos hasta la gestión de la interfaz y los récords:

Inicialización

- Lee el mejor récord de la base de datos (según la dificultad actual) y lo muestra en pantalla.
- Genera un tramo inicial de suelo (21 columnas o huecos, según la opción generarHuecos y los patrones “naranja”/“azul”).
- Instancia los primeros obstáculos estáticos y deja listos los móviles para que su propio script los mueva.

Arranque y fin de partida

- Espera un primer clic/tap/tecla para comenzar (start).
- Cuando el jugador choca o cae, activa el menú de “Game Over”, muestra la distancia recorrida, actualiza el récord en la base de datos y permite reiniciar la escena.

Bucle de juego (Update)

- Desplaza el fondo y el contador de tiempo.

```
0 references
void Update()
{
    if (!start)
    {
        if (Input.GetMouseButtonDown(0) || Input.GetKeyDown(KeyCode.X) ||
            (Input.touchCount > 0 && Input.touches[0].phase == TouchPhase.Began))
        {
            start = true;
        }
    }

    if (start && gameOver)
    {
        menuGameOver.SetActive(true);
        marcadorFinal.text = $"Has recorrido: {tiempo:F0} m";

        if (recordManager != null)
        {
            recordManager.GuardarRecord(nivelActual, tiempo);
            float mejor = recordManager.ObtenerRecord(nivelActual);
            recordTexto.text = $"Record: {mejor:F0} m";
        }

        if (Input.GetMouseButtonDown(0) || Input.GetKeyDown(KeyCode.X) ||
            (Input.touchCount > 0 && Input.touches[0].phase == TouchPhase.Began))
        {
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
    }
```

- Mueve cada columna de suelo a la izquierda; cuando sale de pantalla, la destruye y encola una nueva (o un hueco).

- Mueve y recicla los obstáculos estáticos con la misma lógica.

```
if (start && !gameOver)
{
    // 2) Mover obstáculos estáticos
    for (int i = obstaculos.Count - 1; i >= 0; i--)
    {
        var o = obstaculos[i];                                                    e.de
        if (o == null)
        {
            obstaculos.RemoveAt(i);
            continue;
        }

        if (o.CompareTag("Obstaculo"))
        {
            o.transform.position += Vector3.left * velocidad * Time.deltaTime;
            if (o.transform.position.x <= -10f)
            {
                Destroy(o);
                obstaculos.RemoveAt(i);
            }
        }
    }
            GameObject siguiente = columnasPendientes.Dequeue();
            if (siguiente != null)
                suelo[i] = Instantiate(siguiente, new Vector3(10f, -3f, 0f), Quaternio
            else
                suelo[i] = null;
        }
    }
```

- Regenera continuamente nuevos obstáculos mientras haya suelo bajo su posición y no choquen con otros ya existentes (evitando infinitos bucles de spawn).
- Verifica caída del jugador bajo cierto Y para dar Game Over.

Patrones de generación de suelo

- Modo Fácil (sin huecos): cada columna es elegida aleatoriamente de la lista columnaPrefabs.
- Modo Intermedio/Difícil (con huecos): alterna aleatoriamente bloques de un patrón “naranja” o “azul” (secuencias de índices fijos) y huecos de 1–2 columnas, evitando huecos consecutivos.

```
// Generar suelo inicial (21 columnas o huecos seguidos)
for (int i = 0; i < 21; i++)
{
    if (columnasPendientes.Count == 0)
        GenerarNuevoPatron();

    GameObject prefabCol = columnasPendientes.Dequeue();
    if (prefabCol != null)
    {
        GameObject nuevaCol = Instantiate(prefabCol, new Vector2(ultimaX, -3), Quatern
        suelo.Add(nuevaCol);
    }
    else
    {
        suelo.Add(null); // hueco
    }
    ultimaX++;
}
```

Obstáculos móviles y estáticos

- Los estáticos (tag="Obstaculo") se desplazan junto al suelo y se recrean cuando salen.
- Los móviles (tag="Enemigo") se instancian, se posicionan ajustados al suelo por AjustarAlturaSobreSuelo(), y su propio script les da oscilación además del desplazamiento general.

```
// 2) Mover obstáculos estáticos
for (int i = obstaculos.Count - 1; i >= 0; i--)
{
    var o = obstaculos[i];
    if (o == null)
    {
        obstaculos.RemoveAt(i);
        continue;
    }

    if (o.CompareTag("Obstaculo"))
    {
        o.transform.position += Vector3.left * velocidad * Time.deltaTime;
        if (o.transform.position.x <= -10f)
        {
            Destroy(o);
            obstaculos.RemoveAt(i);
        }
    }
}
```

- Jugador: gestiona la interacción del personaje con el mundo y los estados de salto, encapsulando la lógica de movimiento vertical y detección de colisiones:

Inicialización

- Obtiene referencias a su propio Rigidbody2D y Animator en Start().
- Mantiene un contador de saltos (saltosRestantes = 2) para habilitar un doble salto.

Entrada y salto

- En Update(), cuando el juego ha comenzado y no está en “Game Over”, detecta toques, clics o barra de espacio.

Si quedan saltos disponibles:

- Aplica una fuerza de impulso vertical: la **primera** con fuerzaSalto y la **segunda** (en el aire) con fuerzaSaltoAire, más suave.
- Resetea la velocidad vertical previa, lanza la animación de salto y decrementa saltosRestantes.

```
0 references
void Update()
{
    if (gameManager == null || gameManager.gameOver) return;

    if (gameManager.start)
    {
        bool inputSalto = (Input.touchCount > 0 && Input.touches[0].phase == TouchPhase.Beg
                           || Input.GetMouseButtonDown(0)
                           || Input.GetKeyDown(KeyCode.Space);

        if (inputSalto && saltosRestantes > 0)
        {
            float fuerza = (saltosRestantes == 2) ? fuerzaSalto : fuerzaSaltoAire;

            rb.velocity = new Vector2(rb.velocity.x, 0); // reset vertical
            rb.AddForce(new Vector2(0, fuerza), ForceMode2D.Impulse);
            animator1.SetBool("jumping", true);

            saltosRestantes--;
            grounded = false;
        }

        // Revisa si cae por fuera de pantalla
        if (transform.position.y < -6f)
        {
            gameManager.SetGameOver();
        }
    }
}
```

Detección de suelo y reinicio de saltos

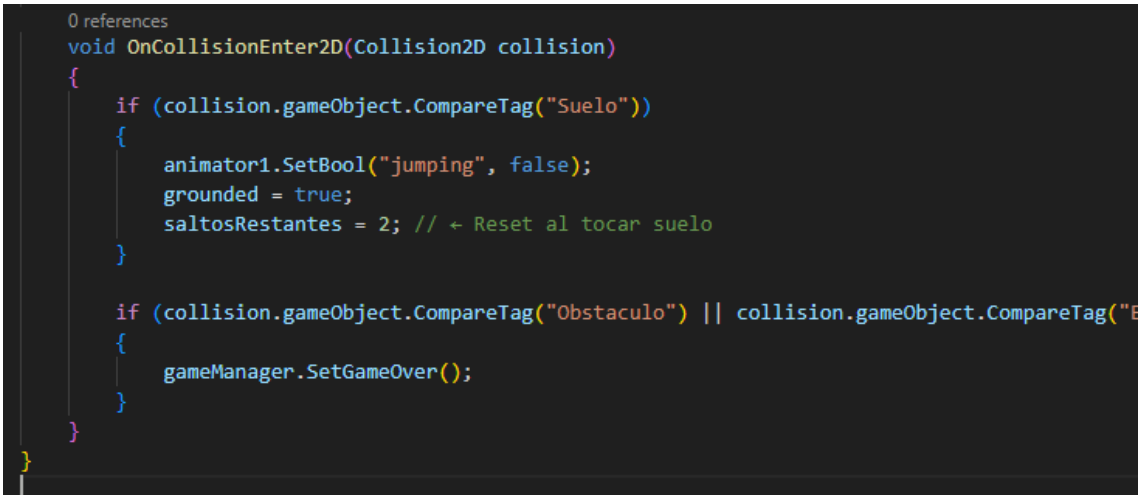
- Al colisionar con objetos taggeados como "Suelo", marca al jugador en tierra (grounded = true), detiene la animación de salto y restaura ambos saltos.

Colisiones peligrosas

- Si choca con un "Obstaculo" o un "Enemigo", notifica al GameManager para terminar la partida.

Caída fuera de pantalla

- Si el jugador cae por debajo de un umbral Y (por ejemplo, –6 f), considera que ha perdido y dispara también el "Game Over".
- En conjunto, permite un control intuitivo de doble salto con gravedad realista, detección de muerte por obstáculo o caída, y reinicio de partida coordinado con el GameManager.



- MenuManager: orquesta la navegación entre las escenas de nivel desde el menú principal:

- **CargarNivelFacil()**, **CargarNivelIntermedio()** y **CargarNivelDificil()** son métodos públicos vinculables a botones de UI.
- Cada uno invoca SceneManager.LoadScene() con el nombre correspondiente, cargando la escena deseada según la dificultad elegida.

- Al separar esta lógica en un controlador dedicado, el menú principal mantiene su código limpio y modular, facilitando añadir o reordenar niveles en el futuro.

- OpcionesMenu.cs: centraliza todas las funciones de la ventana de “Opciones” que aparece durante la partida:

Visibilidad del panel

- En Start() se asegura de ocultar (SetActive(false)) el panel de opciones al inicio.
- TogglePanel() alterna la visibilidad del panelOpciones y pausa/continúa el juego ajustando Time.timeScale.

Control de audio

- ToggleMute() silencia o restablece el volumen global (AudioListener.volume) manteniendo el estado interno estaMuteado.

```
0 references
void Start()
{
    panelOpciones.SetActive(false);
}


0 references
public void ToggleMute()
{
    estaMuteado = !estaMuteado;
    AudioListener.volume = estaMuteado ? 0f : 1f;
}
```

Navegación entre escenas

- CambiarDificultad() y CambiarJugador() reanudan el tiempo (Time.timeScale = 1f) y cargan las escenas "menu" o "Seleccion" respectivamente, permitiendo al usuario cambiar nivel o jugador activos.

```
0 references
public void CambiarDificultad()
{
    Time.timeScale = 1f;
    SceneManager.LoadScene("menu");
}

0 references
public void CambiarJugador()
{
    Time.timeScale = 1f;
    SceneManager.LoadScene("Seleccion");
}
```

Cancelar/reanudar

- Cancelar() cierra el panel y reanuda el juego restableciendo Time.timeScale y la bandera estaPausado.

Salir de la aplicación

- SalirDelJuego() invoca Application.Quit() para cerrar el ejecutable.

Al agrupar aquí la lógica de pausa, audio y cambio de contexto, este manager mantiene la UI de opciones modular, simple de enlazar a botones y fácil de mantener.

- ObstaculoMovil.cs implementa la lógica de un obstáculo “vivo” que combina un movimiento oscilatorio (patrón guardia) con el desplazamiento continuo del terreno:

Configuración inicial

- Guarda su posición de partida (posicionBase) y el estado de juego (referencia al GameManager).
- Elige al azar uno de los sprites provistos para darle variedad visual.

Chequeo de colisión y cambio de dirección

- En cada fotograma comprueba, según el modo (moverVertical), si debe invertir su trayectoria porque:
    - En **horizontal**: falta suelo justo "adelante" o hay un obstáculo detectado con un círculo de colisión.
    - En **vertical**: únicamente se basa en tener un obstáculo delante.
- Al invertir, reinicia la fase de la función seno y actualiza su posicionBase para suavizar el cambio.

Movimiento oscilatorio

Calcula un movimiento, en el eje X o Y según moverVertical.

Desplazamiento con el suelo

- Cada frame añade un vector hacia la izquierda (Vector3.left × velocidadAvance × deltaTime), sincronizándose con el scroll general del suelo definido en el GameManager.

Destrucción automática

- Cuando sale por la izquierda de la cámara (X ≤ –10), se autodestruye para liberar memoria.

En resumen, **ObstaculoMovil** ofrece un patrón de patrulla bidireccional configurable (horizontal o vertical) que a la vez participa en el movimiento continuo de un endless runner, reaccionando dinámicamente a huecos, al borde del terreno y a la presencia de otros obstáculos.

4. BBDD

Ahora pasamos de los apartados de funcionalidad, a los que usan la base de datos local. He utilizado SQLite, ya que se adapta a las necesidades de mi proyecto, y es sencillo de implementar.

Primero, tenemos la sección de Récords, que es la que se encarga de registrar las mejores marcas.

- **RecordData.cs** define la entidad que se almacena en la base de datos y representa un récord de distancia para cada nivel de dificultad:

- **Id**: clave primaria auto-incremental que identifica de forma única cada registro.
- **Dificultad**: cadena que indica el nivel ("Fácil", "Intermedio", "Difícil").
- **Distancia**: número flotante que guarda la mayor distancia recorrida en ese nivel.

Esta clase, junto con el atributo [PrimaryKey, AutoIncrement], permite que SQLite-Unity3d cree automáticamente la tabla correspondiente y trate cada instancia de **RecordData** como una fila en esa tabla.

```
using SQLite4Unity3d;

0 references
public class RecordData
{
    [PrimaryKey, AutoIncrement]
    0 references
    public int Id { get; set; }
    0 references
    public string Dificultad { get; set; }
    0 references
    public float Distancia { get; set; }
}
```

- **RecordManager.cs** se encarga de la persistencia de los récords en SQLite, ofreciendo dos operaciones principales:

Inicialización (Awake)

- Construye la ruta del fichero records.db en Application.persistentDataPath.
- Abre o crea la base de datos SQLite y garantiza que exista la tabla RecordData.

```
0 references
void Awake()
{
    string dbPath = Path.Combine(Application.persistentDataPath, "records.db");
    _connection = new SQLiteConnection(dbPath, SQLiteOpenFlags.ReadWrite | SQLiteOpenFlags
    _connection.CreateTable<RecordData>();
}
```

Guardar o actualizar un récord (GuardarRecord)

- Busca en la tabla un registro con la misma Dificultad.
- Si no existe, inserta uno nuevo con la distancia actual.
- Si existe y la nueva distancia es mayor, actualiza ese registro.

```
0 references
public void GuardarRecord(string dificultad, float distancia)
{
    var record = _connection.Table<RecordData>().FirstOrDefault(r => r.Dificultad == dific
    if (record == null)
    {
        _connection.Insert(new RecordData { Dificultad = dificultad, Distancia = distancia
    }
    else if (distancia > record.Distancia)
    {
        record.Distancia = distancia;
        _connection.Update(record);
    }
}
```

Recuperar un récord (ObtenerRecord)

- Busca el registro por Dificultad.
- Devuelve su Distancia o 0f si aún no hay ningún registro creado.

```
0 references
public float ObtenerRecord(string dificultad)
{
    var record = _connection.Table<RecordData>().FirstOrDefault(r => r.Dificultad == dific
    return record?.Distancia ?? 0f;
}
```

Así, mantiene centralizada la lógica de acceso a datos, permitiendo registrar y consultar fácilmente el mejor resultado de cada nivel.

➢ Dentro de la sección de BBDD, pasamos a los jugadores. Para ello, he creado tres script, siguiendo la misma estructura que con los récords.

JugadorData.cs define la estructura de la tabla que almacena los perfiles de jugador en SQLite:

- **Id**: clave primaria auto-incremental, identifica de forma única cada registro.
- **SlotId**: el número de slot (1–5) al que pertenece el jugador, permitiendo distinguir varios perfiles.

- **Nombre**: el nombre asignado al jugador en ese slot.

Al usar los atributos [PrimaryKey, AutoIncrement], SQLite crea automáticamente la tabla correspondiente y maneja la generación del Id, dejando el SlotId y Nombre listos para guardar o consultar las preferencias de cada usuario en la base de datos.

```
0 references
public class JugadorData
{
    [PrimaryKey, AutoIncrement]
    0 references
    public int Id { get; set; }

    0 references
    public int SlotId { get; set; }
    0 references
    public string Nombre { get; set; }
}
```

SeleccionJugador.cs controla la pantalla de elección y edición de perfiles de jugador, gestionando hasta N slots configurables:

- **Carga inicial**: en Start(), itera sobre cada botón de slot, consulta la base de datos (DBManager) para ver si ya existe un jugador y muestra su nombre o “Vacío”.

```
void Start()
{
    // 1) Configurar cada slot: Mostrar nombre o "Vacío", y asignar listener al botón de s
    for (int i = 0; i < botonesSlots.Length; i++)
    {
        int slot = i + 1;
        var jugador = DBManager.Instance.GetJugador(slot);

        textosSlots[i].text = jugador != null ? jugador.Nombre : "Vacío";

        // Listener para seleccionar slot (jugar o crear)
        botonesSlots[i].onClick.AddListener(() => SeleccionarSlot(slot));
    }
```

- **Selección de slot**:
    - Si no hay jugador asignado, abre un popup (panelEditar) para crear uno;

- Si ya existe, guarda el slot activo en PlayerPrefs y carga la escena de juego/menú.

- **Edición y borrado**: asigna listeners a botones “Editar” y “Borrar” para cada slot:
  - **Editar** muestra el nombre actual en el InputField y abre el popup;
  - **Borrar** elimina el registro de la BD y actualiza la UI a “Vacío”.

```
// 2) Configurar botones de Editar y Borrar para cada slot
for (int i = 0; i < botonesEditar.Length; i++)
{
    int slot = i + 1;
    botonesEditar[i].onClick.AddListener(() => MostrarPopupEditar(slot));
}

for (int i = 0; i < botonesBorrar.Length; i++)
{
    int slot = i + 1;
    botonesBorrar[i].onClick.AddListener(() => BorrarJugador(slot));
}
```

```
//  BorrarJugador: elimina de la base de datos y actualiza UI a "Vacío"
1 reference
void BorrarJugador(int slot)
{
    var jugador = DBManager.Instance.GetJugador(slot);
    if (jugador != null)
    {
        DBManager.Instance.BorrarJugador(slot);
    }
    textosSlots[slot - 1].text = "Vacío";
}
```

- **Popup de nombre**:
  - **Guardar** toma el texto ingresado, crea o actualiza el jugador en SQLite, refresca el texto del slot y cierra el panel;
  - **Cancelar** simplemente oculta el panel sin cambios.

```
// 3) Configurar botones del popup: Confirmar y Cancelar
botonConfirmar.onClick.AddListener(GuardarNombre);
botonCancelar.onClick.AddListener(CancelarEdicion);

// 4) Al inicio, ocultamos el panel de edición
panelEditar.SetActive(false);
```

Con esta lógica, el usuario puede crear, renombrar o eliminar perfiles de forma dinámica antes de comenzar una partida.

DBManager.cs es el controlador singleton que maneja el acceso a la base de datos SQLite de perfiles de jugador.

- **Inicialización** (Awake → InitDB): crea o copia el fichero "jugadores.db" en Application.persistentDataPath, luego abre la conexión con SQLite y garantiza la existencia de la tabla JugadorData.

```
1 reference
void InitDB()
{
    string fileName = "jugadores.db";

    // Ruta donde queremos la BD: siempre en persistentDataPath
    string persistentPath = Path.Combine(Application.persistentDataPath, fileName);

UNITY EDITOR
```

- **Operaciones CRUD**:
    - GetJugador(slotId) consulta y devuelve el perfil que coincide con ese slot.

- CrearJugador(slotId) o CrearJugador(slotId, nombre) insertan un nuevo registro con el nombre por defecto o uno personalizado.

```
2 references
public JugadorData GetJugador(int slotId)
{
    if (_db == null) return null;
    return _db.Table<JugadorData>()
      .Where(j => j.SlotId == slotId)
      .FirstOrDefault();

}

0 references
public void CrearJugador(int slotId)
{
    if (_db == null) return;
    var jugador = new JugadorData { SlotId = slotId, Nombre = "Jugador " + slotId };
    _db.Insert(jugador);
}
```

- ActualizarJugador(...) o ActualizarNombreJugador(...) permiten renombrar un perfil existente.

```
public void ActualizarNombreJugador(int slotId, string nuevoNombre)
{
    var jugador = GetJugador(slotId);
    if (jugador != null)
    {
        jugador.Nombre = nuevoNombre;
        _db.Update(jugador);
    }
}
0 references
public void ActualizarJugador(JugadorData jugador)
{
    _db.Update(jugador);
}
```

- BorrarJugador(slotId) elimina el perfil de la tabla.

- **Singleton persistente**: al usar DontDestroyOnLoad, mantiene viva la conexión y los datos incluso al cambiar de escena, centralizando toda la lógica de persistencia en un único componente.

```
0 references
void Awake()
{
    if (Instance == null)
    {
        DontDestroyOnLoad(gameObject);
        Instance = this;
        InitDB();
    }
    else
    {
        Destroy(gameObject);
    }
}
```

Con esto, concluimos la sección de Scripts que conforman el funcionamiento interno del videojuego.

5. Contenidos

En las siguientes imágenes se puede ver cómo es la interfaz que ve el usuario una vez que ha iniciado el juego.

Primero, aparece la escena de selección de jugador. No se puede iniciar el juego sin haber elegido un jugador. Por defecto, hay cinco slots de jugador, vacíos. Si hacemos clic en el slot y está vacío, abrirá el panel para introducir el nombre, al igual que si hacemos clic en el lápiz.

Podemos guardar el nombre en ese slot, o cancelar.

Con el lápiz podemos editar, y con el icono de la papelera podemos borrar el nombre, y vuelve a estar vacío. Si hacemos clic en un slot con nombre, podemos acceder al juego. Lo que nos lleva a la escena de selección de nivel:

Como se puede ver, hay 3 botones, uno por cada nivel (escenas).

Los niveles se diferencian por dificultad, enemigos y suelo, pero las funcionalidades son las mismas, así que vamos a escoger un nivel, y ver paso a paso cada una de las funcionalidades.

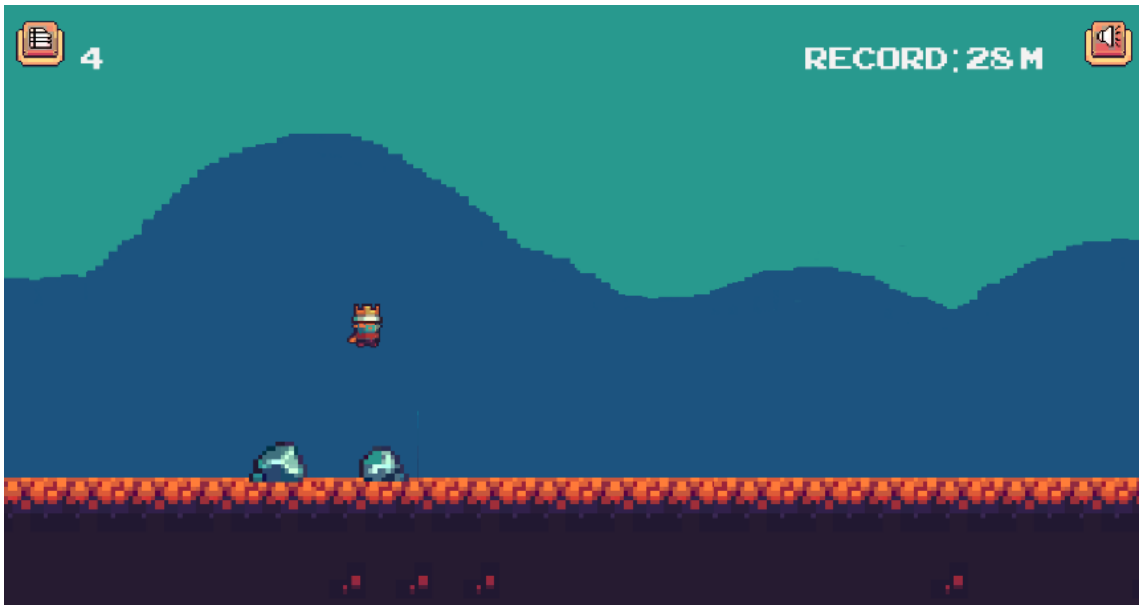
Tenemos el título del juego en el centro, y 'Pulsa para iniciar'. Se puede hacer clic con el ratón, o pulsando la barra espaciadora. La música de fondo ya suena, y la animación del rey comienza, pero se queda estático, para que no se mueva pero dé una sensación de dinamismo.

Arriba a la izquierda tenemos el botón del menú, ya que necesitaba una forma de moverme entre escenas. Este es el panel que se activa cuando se pulsa:

El primer botón es para volver al menú de selección de dificultad. El segundo, para cambiar el perfil del jugador. El tercero, para cerrar el juego. Por último, el cuarto botón sirve para cerrar el panel y reanudar el juego, ya que este panel pausa el juego.

Al lado del botón del menú, vemos los metros recorridos, y en la esquina superior izquierda, vemos el récord de este nivel. Cada nivel tiene un récord distinto.

A la derecha del récord, vemos el botón de la música. Pulsándolo, se silencia o vuelve a sonar, según su estado actual.

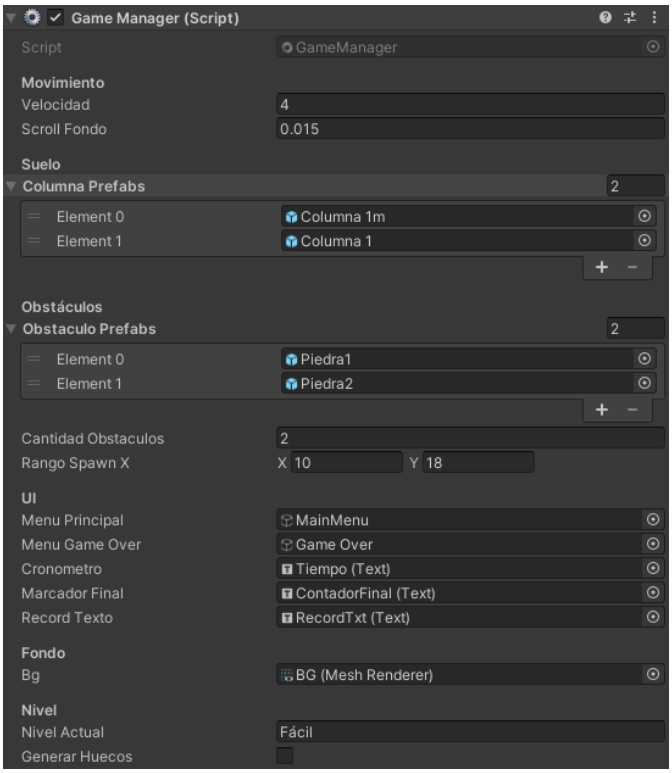


Así se ve el juego una vez iniciado, habiendo hecho clic con el ratón o con la barra espaciadora.

Por último, la sección del Game Over:

Se muestra cuantos metros se han recorrido en este intento, el texto 'Game Over', y 'Pulsa para reiniciar', que también con los mismos controles de antes, reinicia el juego.

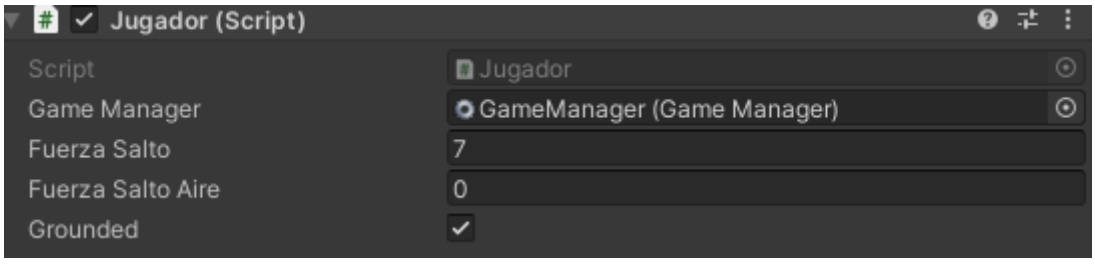
6. Implementación de las escenas



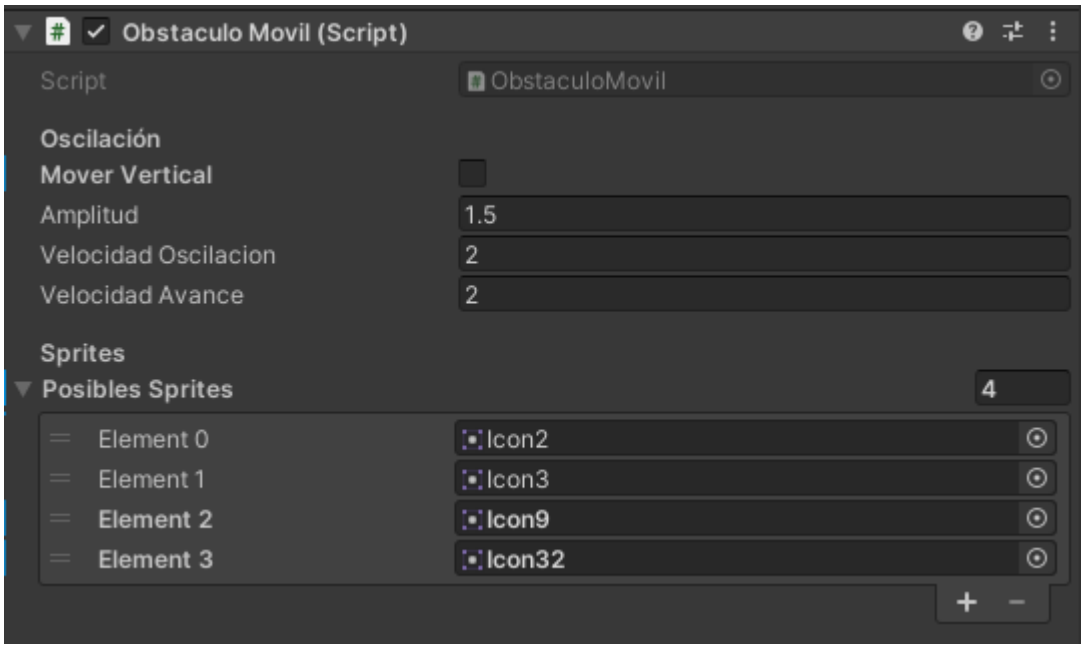
En el **GameManager** hemos expuesto en el Inspector dos parámetros clave para controlar el "nivel" sin tocar código:

1. **Nivel Actual** (string):
   Sirve principalmente para etiquetar el récord en base a la dificultad ("Fácil", "Intermedio" o "Difícil"), y así almacenarlo/recuperarlo con RecordManager.
2. **Generar Huecos** (bool):
   - **Desmarcado** (Modo Fácil):
     - **Suelo**: nunca se generan huecos, sólo se escoge cada columna al azar de la lista columnaPrefabs.
     - **Obstáculos**: siempre se colocan sobre suelo, sin preocuparse por saltos demasiado grandes ni huecos.
   - **Marcado** (Modos Intermedio/Difícil):
     - **Suelo**: se usa un sistema de patrones "naranja" y "azul" —bloques de 4 y 3 columnas— separados aleatoriamente por huecos de 1–2 columnas, evitando huecos consecutivos demasiado grandes.
     - **Obstáculos**: se alternan obstáculos estáticos y móviles colocados sólo si hay suelo bajo ellos, y estos respetan el patrón de huecos.

En la práctica, cambiando **Generar Huecos** en el Inspector pasa de un endless runner con suelo ininterrumpido (Fácil) a uno donde las plataformas vienen en tramos y huecos aleatorios (Intermedio/Difícil).

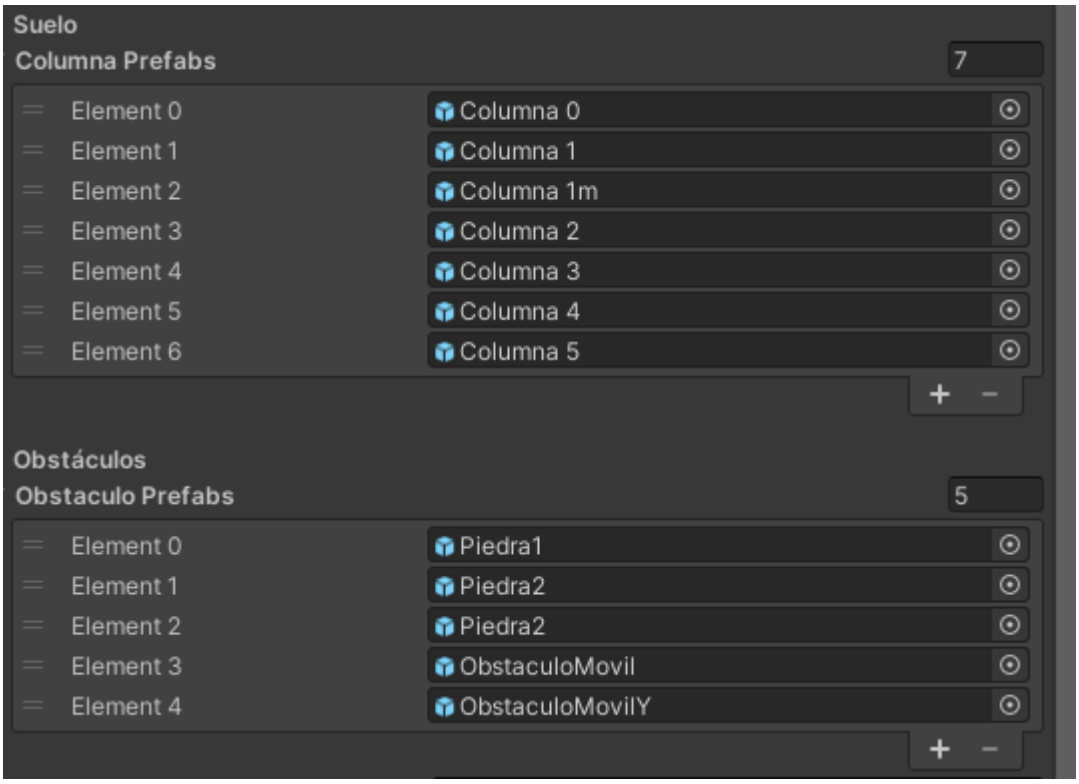


En cuanto al jugador, se cambia la fuerza de salto (ya que en el nivel difícil, el jugador salta con más fuerza, porque sino es demasiado complicado), y le he añadido un doble salto en el aire, que en el nivel fácil no existe, y por eso está a 0.



Este es el inspecto del enemigo móvil en el nivel intermedio. La opción de mover verticalmente está desmarcada, y solo se marca en el nivel difícil. Además, vemos los posibles enemigos que pueden aparecer. Se pueden añadir o quitar, sin afectar al funcionamiento. También podemos cambiar la velocidad de oscilación o movimiento del enemigo desde aquí.

Para contrastar, aquí esta el inspector del GameManager en el nivel difícil:



7. Incidencias

A lo largo del desarrollo han sido especialmente recurrentes estos bloques de problemas:

- **Instanciación de prefabs nulos o índices fuera de rango**
  - Errores ArgumentException: “Object you want to instantiate is null” y “ArgumentOutOfRangeException” al generar columnas cuando la cola de patrones estaba vacía o los arrays de patrones no coincidían con el tamaño de columnaPrefabs.
- **Huecos demasiado grandes para el jugador**
  - Se generaban saltos de más de 2 columnas, superando la capacidad de salto; se corrigió limitando el ancho de hueco a 1–2 y evitando dos huecos seguidos.
- **Obstáculos estáticos y móviles mal sincronizados**
  - Obstáculos “fijos” acababan moviéndose con el suelo.
  - Los móviles no respetaban el movimiento de fondo y a veces flotaban, atravesaban huecos o colisionaban mal (aparecían en dos sitios a la vez).
- **Gestión de colisiones y tags**
  - El jugador no detectaba correctamente choque con “Enemigo” vs. “Obstaculo”.
  - Fallos al usar CompareTag, se acabó usando tags separados y comprobaciones en OnCollisionEnter2D.
- **Persistencia con SQLite4Unity3d**
  - Conflicto de DLLs x86 vs x64 al hacer el build: Unity no encontraba la librería sqlite3 en build, y en el editor no abría la BD correctamente.
  - NullReferenceException en DBManager al no copiar/crear bien el fichero en persistentDataPath frente a StreamingAssets.

- **UI y flujo de escenas**
  - El panel de selección de jugador lanzaba NullReference por listeners mal asignados.
  - Botones de pausa no respondían porque la mecánica de "click" del juego interceptaba el evento; hubo que pausar el Time.timeScale y asegurarse de que el rectángulo del canvas esté correctamente configurado.
- **Build & Run fallido**
  - Errores al hacer el build (BuildMethodException) relacionados con plugins duplicados y referencias inválidas, que se solventaron limpiando la carpeta Plugins y ajustando las compatibilidades de plataforma.
- **Oscilación de enemigos**
  - La implementación inicial movía los sprites erráticamente (temblaban o giraban al colisionar).

Cada uno de estos problemas obligó a revisar de forma persistente el GameManager, los scripts de jugador y enemigo, y la configuración de assets y plugins en Unity hasta encontrar soluciones robustas para un endless runner fluido y estable.

8. Manual de usuario

He creado un script nuevo, que crea un manual de usuario en formato web y se puede abrir desde cualquier escena del videojuego con la tecla F11.

```
0 references
public class ManualLauncher : MonoBehaviour
{
    1 reference
    const string MANUAL_FILE = "manual.html";

    0 references
    void Update()
    {
        // Detecta pulsación de F11
        if (Input.GetKeyDown(KeyCode.F11))
        {
            OpenManual();
        }
    }

    1 reference
    void OpenManual()
    {
        // Construye la ruta completa
        #if UNITY_ANDROID && !UNITY_EDITOR
        string url = Path.Combine(Application.persistentDataPath, MANUAL_FILE);
        #else
        string url = Path.Combine(Application.streamingAssetsPath, MANUAL_FILE);
        #endif

        Application.OpenURL(url);
    }
}
```

9. Conclusiones

Este proyecto me ha dado una visión completa de cómo montar un juego en Unity desde cero. Diseño tres niveles con dificultades crecientes y aprendo a generar el suelo y los obstáculos para que siempre fuese jugable. Separar la lógica en componentes claros—un GameManager para el flujo del juego, un script para el jugador y otro para los obstáculos—me permitió iterar más rápido y mantener el código ordenado.

Integrar SQLite fue un reto: guardando récords y perfiles de jugador localmente, con un selector de hasta cinco usuarios. Tuve que ajustar la librería nativa para que funcionase en build. Además, Unity insiste en que se usen los Plugins de su propia tienda (de pago), por lo que lo tuve que implementar con terceros.

En cuanto a jugabilidad, añadir doble salto, obstáculos móviles que oscilan en X o Y, y un menú de pausa con manual HTML (accesible con F11) me hizo tener que investigar y aprender para hacer que diversas funcionalidades pudiesen ser compatibles entre sí.

En definitiva, este proyecto me ha enseñado a combinar mecánicas sencillas con generación automática, bases de datos y una UI funcional, todo en C# y Unity, sentando unas buenas bases para algún videojuego 2D que quiera construir a continuación.

10. Referencias

- Motor gráfico Unity: Unity
- Iconos de opciones: itch.io
- Sprites de jugador y mapa: antonymorsas.com
- Sprites de enemigos móviles: craftpix