



MÓDULO PROYECTO

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

CLON DE FLAPPY BIRD CON GODOT 4.4

Tutor individual: Rodrigo Iglesias Gorron

Tutor colectivo: Cristina Silvan Pardo

Año: 2024/2025

Fecha de presentación: 23/5/2025 – 11:45

Nombre y Apellidos: Hugo del Rey
Holgueras

Email: hugo.reyhol@educa.jcyl.es

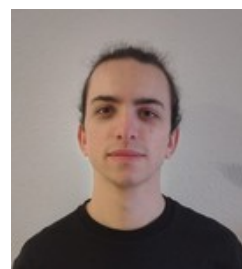


Tabla de contenido

1	Identificación proyecto.....	4
2	Organización de la memoria.....	4
3	Descripción general del proyecto.....	4
3.1	Objetivos.....	4
3.2	Cuestiones metodológicas.....	4
3.3	Entorno de trabajo (tecnologías de desarrollo y herramientas).....	5
4	Descripción general del producto.....	5
4.1	Visión general del sistema: límites del sistema, funcionalidades básicas, usuarios y/o otros sistemas con los que pueda interactuar.....	5
4.2	Descripción breve de métodos, técnicas o arquitecturas(m/t/a) utilizadas.....	6
4.3	Despliegue de la aplicación indicando plataforma tecnológica, instalación de la aplicación y puesta en marcha.....	7
5	Planificación y presupuesto.....	7
5.1	Planificación (GDD).....	7
5.1.1	Información.....	8
5.1.2	Mecánicas.....	8
5.1.3	Arte y audio.....	9
5.1.4	Tecnologías.....	9
5.1.5	Interfaz de Usuario.....	10
5.2	Presupuesto.....	12
6	Documentación Técnica: análisis, diseño, implementación y pruebas.....	13
6.1	Especificación de requisitos.....	13
6.2	Análisis del sistema.....	13
6.3	Diseño del sistema:.....	13
6.3.1	Diseño de la Base de Datos.....	13

6.3.2	Diseño de la Interfaz de usuario.....	14
6.3.3	Diseño de la Aplicación.....	19
6.4	Implementación:.....	19
6.4.1	Entorno de desarrollo.....	19
6.4.2	Estructura del código.....	20
6.4.3	Cuestiones de diseño e implementación reseñables.....	53
6.5	Pruebas.....	53
7	Manuales de usuario.....	54
7.1	Manual de usuario.....	54
7.2	Manual de instalación.....	56
8	Conclusiones y posibles ampliaciones.....	57
9	Bibliografía.....	58
10	Anexos.....	58

1 Identificación proyecto

El proyecto trata de crear un clon del videojuego **Flappy Bird** usando la herramienta Godot 4.4.

2 Organización de la memoria

La memoria se ha organizado en distintos apartados en los que se hablará de puntos como los objetivos, el entorno de trabajo, descripción general del producto, planificación, documentación, manuales y conclusiones finales con posibles ampliaciones.

3 Descripción general del proyecto

3.1 Objetivos

Mis principales objetivos al desarrollar este proyecto son:

- Aprender a organizar una aplicación compleja de principio a fin
- Aprender a usar el motor de videojuegos Godot y su lenguaje propio GDScript

3.2 Cuestiones metodológicas

Para desarrollar este proyecto se ha usado la metodología en cascada o waterfall en la cual se siguen unos pasos detallados que deberán ser completados antes de pasar al siguiente.

Los pasos que se seguido han sido los siguientes:

1. **Análisis de requisitos:** Listar todo lo necesario para que el proyecto sea aceptado como válido, en mi caso las normas de proyecto y de la memoria listadas en GitHub y relacionar apartados de la aplicación con cada una de las materias vista en clase
2. **Diseño:** Se planea todo lo que va a contener la aplicación, para más detalles ver [Planificación](#) y [Diseño](#)
3. **Implementación:** Desarrollo de la aplicación y pruebas unitarias, para más detalles ver [Implementación](#)

4. **Pruebas:** Se realizan distintos test para comprobar que la aplicación es funcional y cumple los requisitos solicitados, para más detalles ver [Pruebas](#)
5. **Despliegue:** Se distribuye la aplicación entre los usuarios finales, para más detalles ver [Despliegue](#)

Entre las ventajas que ofrece la metodología en cascada, la que más destaca es la facilidad de desarrollo en el aparatado de implementación debido al análisis realizado previamente, también hay que tener en cuenta sus puntos negativos como la poca flexibilidad que ofrece ante cambios.

3.3 Entorno de trabajo (tecnologías de desarrollo y herramientas)

Para desarrollar el proyecto uso las siguientes herramientas:

Godot 4.4: Es el motor en el cual se crea el videojuego, en él se programa todo lo relacionado con el comportamiento del juego y las llamadas a la API, también es el programa usado para exportarlo a las distintas plataformas.

Aseprite 1.3.10.1: Es el programa con el que se crean y editan las imágenes que se usan dentro del videojuego, esta especializado en *pixel art*.

GDScript: Es el lenguaje de programación usado para la lógica del videojuego, es propio de Godot y está optimizado para su uso en el desarrollo de videojuegos.

Supabase: Es la plataforma en la cual se alojará la base de datos y la API que usará el juego.

Git y GitHub: Es el software y la plataforma usados para el control de versiones del proyecto.

4 Descripción general del producto

4.1 Visión general del sistema: límites del sistema, funcionalidades básicas, usuarios y/o otros sistemas con los que pueda interactuar.

Límites del sistema:

Permitir al usuario (jugador) realizar varias partidas y guardar su estadísticas en la base de datos solo en caso de estar registrado.

Funcionalidades básicas:

- Permitir a los usuario registrarse e iniciar sesión
- Obtener información de la API para crear una tabla de clasificación
- Escuchar los *inputs* del jugador, teclas en pc, pantalla en móviles...
- Mover al personaje en función de los inputs
- Detectar colisiones del personaje con elementos de juego
- Enviar información a la API sobre la última partida
- Reiniciar la partida para poder jugar varias

Usuarios y/o otros sistemas con los que puede interactuar:

Todos los jugadores pueden realizar las acciones necesarias para jugar partidas, la principal diferencia es que los no registrados podrán registrarse o iniciar sesión pero no podrán guardar sus partidas en la base de datos y los jugadores que si estén registrados podrán cerrar sesión y guardar sus partidas en la base de datos.

4.2 Descripción breve de métodos, técnicas o arquitecturas(m/t/a) utilizadas.

Se ha usado la metodología en cascada como desarrollaba en [Cuestiones Metodológicas](#).

Algunas técnicas usadas son:

- **Comunicación mediante señales:** Comunicaciones entre nodos de forma desacoplada que permite reaccionar a eventos
- **State machines:** Permite manejar distinta lógica dependiendo de un estado interno del programa

- **UI adaptativa:** Mediante anclajes permite que la interfaz se ajuste a distintas resoluciones
- **Delta timing:** Mide el tiempo entre fotogramas para que las animaciones e interacciones físicas sean iguales en equipos con distinta potencia
- **Guardado de datos:** Se utilizan estructuras JSON y ConfigFile para almacenar datos de la aplicación que se mantienen al cerrarla
- **Recursos personalizados:** Permite definir características reutilizables de nodos sin implementar su lógica

Algunas arquitecturas utilizadas:

- **Composición:** Los nodos usados se forman a partir de varios nodos con funcionalidades más básicas
- **MVC ligera:** Se separa la lógica y datos de los nodos de la interfaz
- **Singletons:** Se crean scripts que solo se instancian una vez por ejecución y que se usan para tener acceso global a varias funciones

4.3 Despliegue de la aplicación indicando plataforma tecnológica, instalación de la aplicación y puesta en marcha

La aplicación está compilada en un binario por lo que solo será necesario seguir los pasos en el [Manual de instalación](#) para obtenerla, en cuanto a la base de datos y su API, esta está desplegada en Supabase por lo que mientras tengas acceso a internet y se esté usando un binario precompilado de GitHub siempre se podrá usar.

5 Planificación y presupuesto

5.1 Planificación (GDD)

En el mundo de los videojuegos a la hora de planear un nuevo proyecto siempre se crea un GDD (Game Design Document) el cual es un documento que recopila todos los detalles relacionados con el juego, estos pueden ir desde las mecánicas que se tienen que programar, el estilo artístico que tienen que usar los artistas gráficos y de sonido o la ambientación de este para los guionistas.

En el GDD a continuación se van a omitir los puntos *Mundo, Historia y Narrativa, Monetización y Producción* ya que al estar haciendo un clon de juego ya existente no son necesarios.

5.1.1 Información

En este apartado se darán detalles generales del videojuego.

- **Título:** Clon de Flappy Bird
- **Género:** Acción y Scroll Infinito
- **Plataformas:** Windows, Linux y Android
- **Público objetivo:** Jugadores de todas las edades
- **Descripción:** Es un juego en el que los jugadores tendrán que poner a prueba su habilidad para esquivar obstáculos para alcanzar una mayor puntuación usando la mecánica principal del salto

5.1.2 Mecánicas

En este apartado se define como interactuará el jugador con nuestro videojuego.

- **Gameplay Loop:** El jugador evitará chocarse con obstáculos que aparecerán de forma aleatoria, al colisionar con uno o salirse del mapa perderán la partida y empezarán de nuevo.
- **Controles y Cámara:** Al jugar en PC podremos saltar usando la tecla de espacio o con un clic izquierdo, mientras que en móviles pulsaremos la pantalla, en cuanto a la cámara esta estará fija en la misma posición junto con el personaje del jugador y lo único que se desplazará serán los obstáculos.
- **Interacciones del jugador:** El jugador estará fijo en su coordenada horizontal y se podrá desplazar verticalmente saltando, al saltar aumentará su altura y con el paso del tiempo caerá al suelo por acción de la gravedad.
- **Sistema de Obstáculos:** Los obstáculos se componen por dos tuberías alineadas verticalmente que estarán separadas entre sí por una distancia aleatoria, en el mapa se podrán generar varios de estos obstáculos que avanzarán de derecha a izquierda, el jugador aumentará su puntuación al cruzar entre las tuberías. Para aumentar la

dificultad del juego los obstáculos se desplazarán más rápido dependiendo de cuantos puntos tenga el jugador.

- **Condiciones de Victoria y Derrota:** Una partida es infinita por lo que un jugador no puede ganar una partida, una partida se dará por finalizada cuando el jugador choque con un obstáculo, el suelo o el techo.

5.1.3 Arte y audio

En este apartado se definiría el apartado audio-visual del juego para que los artistas pudiesen crear estos recursos, pero al ser un clon voy a usar unos similares a los originales que he encontrado en internet.

En el juego original el estilo visual es tipo *pixel art*^[1], un estilo artístico que consiste en representar imágenes usando resoluciones muy pequeñas imitando la estética de los videojuegos de los años 80 y 90.

El apartado sonoro se compone de sonidos de corta duración y muy distinguibles entre sí que ayudan al jugador a entender que está pasando en su partida.

5.1.4 Tecnologías

En este apartado se seleccionan las herramientas que se usarán para crear todas las partes necesarias para el videojuego.

Motor: Usaré Godot por su curva de aprendizaje accesible y no tener ningún tipo de coste para el desarrollador.

Lenguaje de Programación: Se usará GDScript ya que Godot está diseñado para usar este.

Plataforma de Desarrollo: Todo desarrollo y las pruebas unitarias se realizarán en un PC con Linux.

Arte: En caso de necesitar algún recurso extra o tener que modificar uno ya existente se usará Aseprite, un software especializado en la creación de imágenes con estilo pixel-art.

Base de Datos y API: Todos los datos se almacenarán en Supabase por su plan gratuito y facilidad de uso.

5.1.5 Interfaz de Usuario

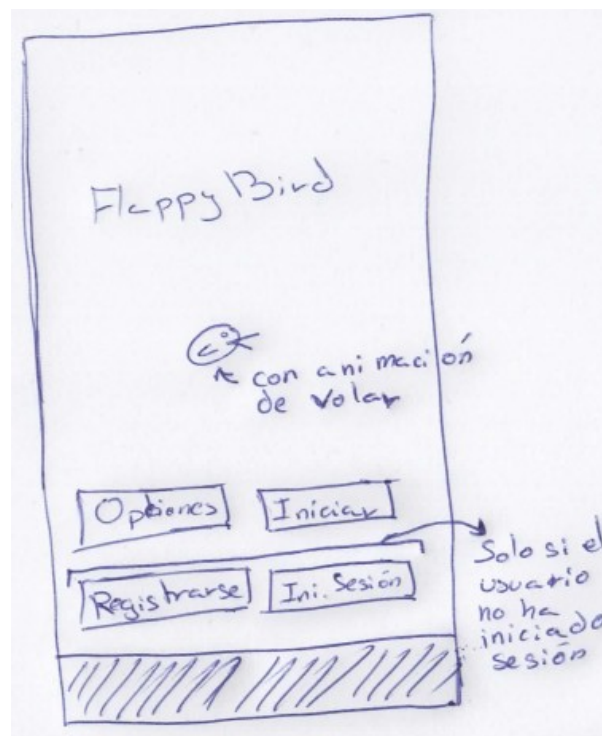
En este apartado se especificará a grandes rasgos como se dispondrán los distintos elemento del videojuego en la pantalla, para empezar tanto en pc como en móviles el juego tendrá un disposición vertical, es decir, el alto de la cámara será mayor que su ancho.

Menú Inicio:

Este es el menú que aparece cuando se abre el juego, esta se compone por el título con el nombre del juego en la parte superior, en la parte central aparece el personaje con el que va a jugar el usuario y en la parte inferior hay cuatro botones, el primero permite abrir el menú de opciones, el segundo cambia a la pantalla **Partida**, y los dos últimos permiten registrarse e iniciar sesión respectivamente y solo se mostrarán en caso de que el usuario no haya iniciado sesión.

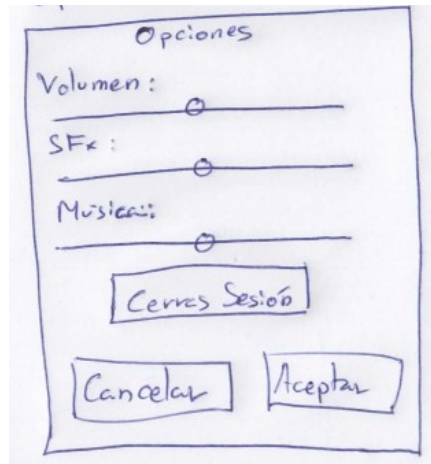
Al abrir el menú se ocultaran los botones y se superpondrá sobre la pantalla de inicio.

La transición entre **Menú Inicio** y **Partida** ocultará el título del juego y los botones, también posicionará el jugador del centro al primer cuarto de la pantalla y una vez ahí se hará el cambio real entre escenas.



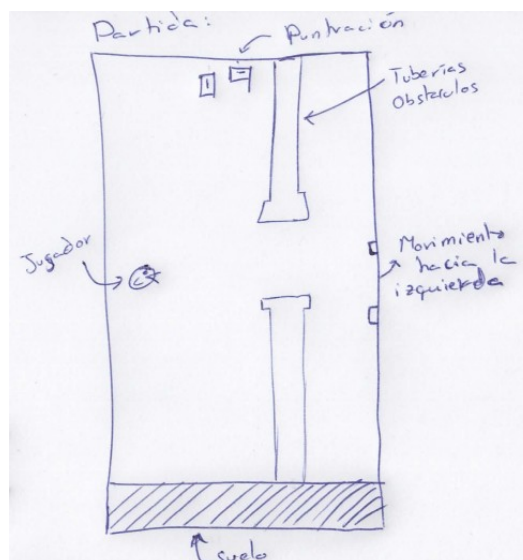
Opciones:

Al abrir las opciones estás se mostrarán encima de la ventana actual, en estas se podrá cambiar el volumen maestro, los efectos de sonido y la música, una vez hechos estos cambios podremos pulsar Aceptar o Cancelar para conservarlos o no. También se podrá cerrar la sesión actual desde este menú.



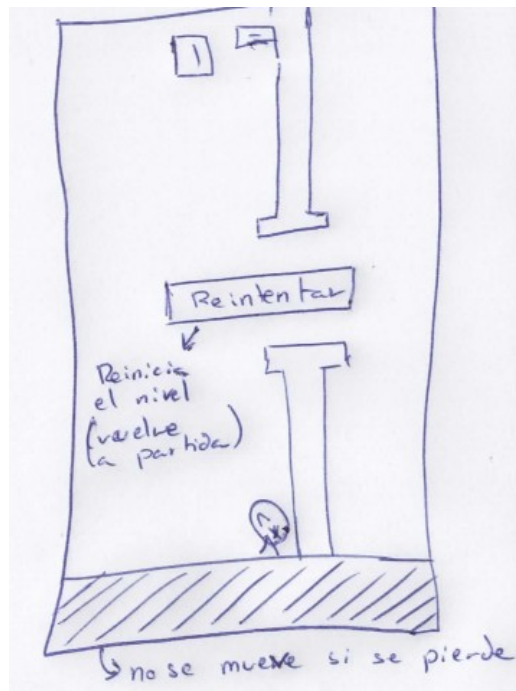
Partida:

El jugador se mantendrá siempre en el primer cuarto de la pantalla y no se desplazará horizontalmente, en la parte superior se mostrará la puntuación actual de la partida, el fondo y el suelo se moverán hacia la izquierda para dar sensación de movimiento, por último los obstáculos aparecerán por la parte derecha de la pantalla y desaparecerán por la izquierda.



Fin Partida:

Al colisionar con un obstáculo se modificará el aspecto de la pantalla **Partida**, todos los elementos en movimiento se detendrán y el jugador caerá hasta tocar el suelo, una vez ahí se mostrará el botón *Reintentar* con el cual se reiniciará la escena de la partida volviendo al estado de **Partida**.



5.2 Presupuesto

Para calcular el coste de un desarrollo se tienen que tener en cuenta múltiples apartados como los siguientes:

- **Tipo de Videojuego:** Será un juego 2D de tipo scroll infinito con una mecánica principal, por lo que su desarrollo será sencillo.
- **Arte y Audio:** Ambos se han conseguido gratuitamente de internet.
- **Herramientas y Licencias:** Godot es completamente gratuito y Aseprite también en caso de que lo compiles tu mismo desde su GitHub oficial.
- **Marketing y Distribución:** No se realizará y el juego se distribuirá en la pestaña releases de su repositorio de GitHub.

Teniendo en cuenta todos los apartados anteriores en lo único que habría que gastar dinero sería en el sueldo del programador y con la estimación de 50 horas de desarrollo

(incluyendo la memoria y presentación) y el SMI por hora de España, unos 9,26€/h, estimo que los costes rondarían los 463€ en total.

6 Documentación Técnica: análisis, diseño, implementación y pruebas.

6.1 Especificación de requisitos

Para este proyecto se ha pedido que la aplicación sea funcional, tenga comentarios útiles, documentación y manual de usuarios, tenga una buena estructura, etc...

6.2 Análisis del sistema

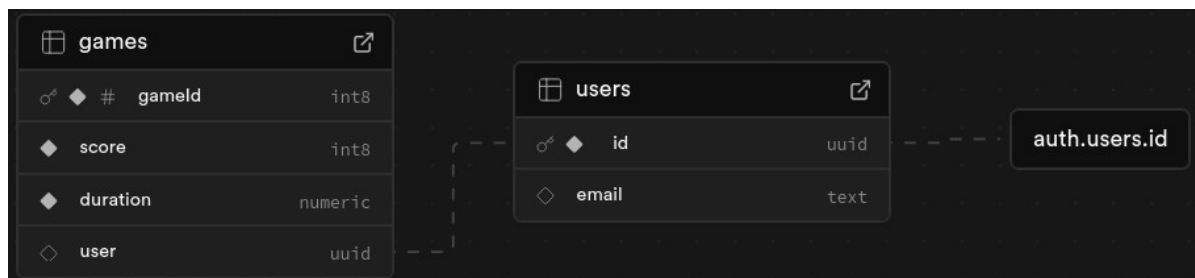
Se da una explicación detallada del sistema en el apartado [Visión general del sistema](#).

6.3 Diseño del sistema:

6.3.1 Diseño de la Base de Datos

La estructura de la base de datos es simple ya que la complejidad de la aplicación no reside en sus datos, contará con dos tablas, una para los usuarios llamada **users** en la que almacenaremos sus correos electrónicos y sus uuids de la tabla auth, la otra guardará las partidas llamada **games** realizadas con los puntos obtenidos en esta y el tiempo que duró esta en milisegundos.

Las tablas se relacionarán entre sí con una relación *Many to One*, en la cual un usuario podrá registrar varias partidas.



Supabase se encarga de gestionar los registros e inicios de sesión en una tabla privada llamada **auth**, a esta tabla solo se puede acceder desde la interfaz de configuración de la página web, por ello para poder relacionarla con la tabla **games** hay que crear un trigger cada vez que se añade un usuario que haga un clon con parte de la información del nuevo usuario en la tabla **users**.

```
CREATE TABLE USERS (  
  id uuid references auth.users not null primary key,  
  email text  
);  
create or replace function public.handle_new_user()  
returns trigger as $$  
begin  
  insert into public.users (id, email)  
  values (new.id, new.email);  
  return new;  
end;  
$$ language plpgsql security definer;  
  
create trigger on_new_user  
after insert on auth.users for each row  
execute procedure public.handle_new_user ();
```

6.3.2 Diseño de la Interfaz de usuario.

La interfaz de usuario es parecida a la mostrada en el apartado [Interfaz de Usuario del GDD](#), los cambios que se han realizado son los siguientes.

Menú de inicio:

En el menú inicial se pueden ver varios elementos, en orden son:

- **Tabla de puntuaciones:** En este se pueden ver las tres mejores partidas almacenadas en la base de datos y el nombre del jugador que las realizó, no aparece en pantalla hasta que la consulta a la API se resuelve y al iniciar una partida se oculta desplazándose hacia arriba. La consulta solo se realiza una vez al inicio del menú de inicio y solo se vuelve a realizar si se vuelve a abrir el juego o al perder una partida y volver al menú.
- **Título:** Muestra el nombre del juego, entra y sale de pantalla por el borde superior de la ventana al entrar en el menú inicio y al iniciar un juego respectivamente.
- **Jugador:** Una versión del jugador que reproduce una animación en el centro de la pantalla, al iniciar una partida se desplaza al punto de juego mientras se carga la pantalla de partida.

- **Botón de inicio:** Cambia la pantalla de inicio al de partida, se puede activar pulsándolo o usando la barra de espacio.
- **Botón de opciones:** Abre el menú de opciones en el que el jugador puede configurar la aplicación.
- **Botones de sesión:** Los botones usados para abrir el menú de registro y de inicio de sesión, estos botones no se muestran cuando el jugador tiene una sesión iniciada.
- **Botón de salida:** El botón con el que se puede cerrar el juego, este botón no aparece en la versión de móvil ya que estos dispositivos suelen tener métodos propios para cerrar aplicaciones.

En la izquierda se puede ver el menú antes de iniciar sesión y la derecha después de hacerlo.



Menú de opciones:

En este menú el jugador puede cambiar ajustes del juego como el audio e idioma, sus elementos son:

- **Volumen:** Sección para configurar los niveles de audio con *sliders* y botones para silenciar completamente las pistas, se divide en tres opciones:
 - **General:** Se encarga de todo el sonido en global.
 - **Música:** Modifica solo el volumen de las pistas de música.
 - **SFX:** Usado para cambiar el volumen de los efectos de sonido.
- **Usuario:** Sección para configurar preferencias del usuario, se divide en dos opciones:
 - **Idioma:** Permite intercambiar el idioma del juego entre todas las traducciones realizadas, actualmente español e inglés.
 - **Sesión:** Permite cerrar la sesión actual, por limitaciones de Supabase cierra sesión en todos los dispositivos conectados. Solo aparece cuando el usuario ha iniciado sesión.
- **Botón de cancelar:** Revierte los ajustes al estado en el que se encontraban al abrir las opciones a excepción de cerrar la sesión ya que depende de Supabase.
- **Botón de guardar:** Almacena todos los cambios en un archivo de configuración para que se carguen la siguiente vez que se abra el juego.

A la izquierda se puede ver una foto del menú sin la sesión iniciada y a la derecha otra que si la tiene.

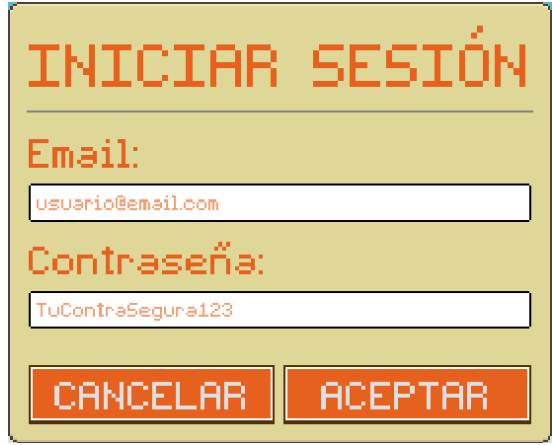


Menú de inicio/registro:

En este menú el jugador puede registrarse en el servidor de Supabase con su correo electrónico y una contraseña o iniciar sesión si ya dispone de una cuenta, cuenta con:

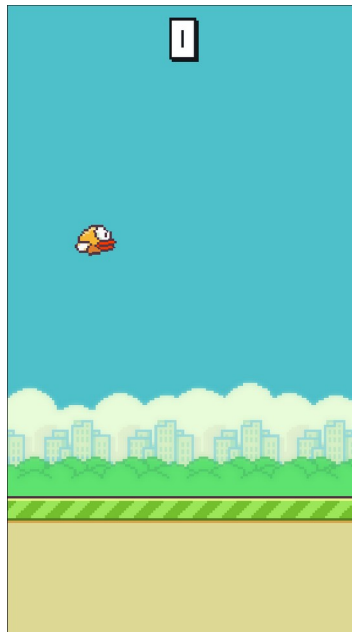
- **Campo email:** Parte del formulario donde se introduce el correo electrónico del jugador, tiene un ejemplo de email.
- **Campo contraseña:** Parte donde se introduce la contraseña.
- **Botones cancelar/aceptar:** Cancelan y empiezan el inicio de sesión.

A la izquierda se puede ver el formulario de registro y a la derecha el de inicio de sesión, son el mismo cambiando el título dependiendo de la acción a iniciar.



Jugando una partida:

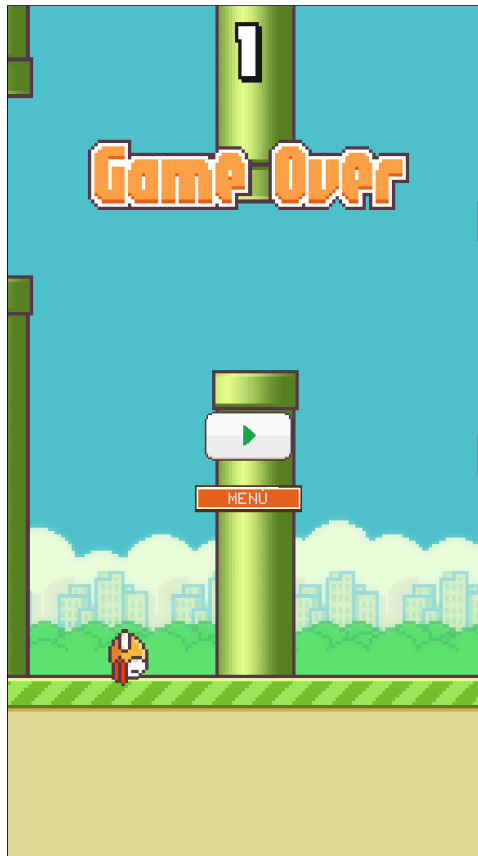
Para no entorpecer la visibilidad del jugador el único elemento de la interfaz gráfica visible durante las partidas es la puntuación actual.



Al perder una partida:

Es la interfaz que aparece en pantalla cuando el jugador choca con un obstáculo, además de la puntuación mencionada en el apartado anterior consta de los siguientes elementos:

- **Mensaje perdida:** Un gran cartel que le indica al jugador que ha perdido.
- **Botón de inicio:** Reinicia el juego para que el jugador pueda jugar otra partida inmediatamente.
- **Botón de menú:** Vuelve a la pantalla de inicio.



6.3.3 Diseño de la Aplicación.

Para la lógica de diseño he seguido el [GDD del apartado 5.1](#) y los únicos cambios destacables han sido algunas de las interfaces, las cuales ya han sido analizadas en el [apartado 6.3.2](#).

6.4 Implementación:

6.4.1 Entorno de desarrollo.

El entorno de desarrollo utilizado en este proyecto ha sido el que viene incluido con el motor **Godot 4.4**, además para facilitar su uso he instalado tres plugins, el primero es [Script IDE](#) que convierte la interfaz gráfica a una más cercana a la usada por otros IDEs como los de JetBrains, el segundo es [Kanban Tasks - Todo Manager](#) el cual añade una pestaña al IDE para gestionar las tareas necesarias en el proyecto al estilo Kanban, el último es [Godot Engine - Supabase \(4.x\)](#) encargado de gestionar las conexiones a Supabase facilitando la comunicación con la API.

6.4.2 Estructura del código.

Para poder explicar adecuadamente la estructura de mi código y por qué decidí dividirla de esa forma primero necesito explicar cómo se programa en Godot.

Introducción:

En Godot los elementos con los que el jugador interactúa se llaman *escenas*, estas escenas pueden ir desde un nivel, un menú, una simple pared e incluso el personaje que controla el jugador. Estas escenas están formadas a su vez por otras escenas y/o *nodos*, este es el nombre que reciben los objetos con su propia lógica nativos del motor.

Los nodos siguen el paradigma POO, heredando características de sus padres, además están precompilados en el propio motor por lo que hacer uso de ellos en la medida de lo posible para implementar la lógica propia de nuestro juego aumentará el rendimiento, consistencia y experiencia programando, depurando y jugando nuestro videojuego.

Por otra parte los componentes que forman las escenas se comunican entre si haciendo uso de *scripts*, fragmentos de código en GDScript que extienden la lógica de un nodo de forma similar a crear una clase que herede de él, en los que escribamos la lógica necesaria para hacer funcionar nuestro juego.

Las escenas siempre se componen por un único nodo/escena padre el cual podrá o no tener otros nodos/escenas hijos, cuando queremos añadirle a estos nodos comportamientos personalizados se le asigna al padre de la escena un script el cual generalmente se comunicará con los nodos hijos haciendo llamadas a sus métodos y recibirá información de estos mediante señales.

Estas señales se usan para que los nodos puedan avisar y compartir información con sus nodos padres y hermanos sobre eventos que hayan ocurrido dentro de su lógica, pero para que esto suceda primero estos nodos deben conectarse a la señal y asignarle un método a ejecutar cuando esta se emita, asegurando de esta forma que los nodos interesados en ciertos eventos siempre estén al tanto de ellos sin realizar acoplamientos con otros nodos y manteniendo el código flexible entre nodos padres e hijos.

Organización de carpetas:

En el mundo de los videojuegos hay dos practicas para organizar los proyectos dependiendo de la magnitud de estos, en las empresas encargadas de desarrollar grandes productos se divide los archivos por tipos, por ejemplo colocando así todos los

scripts en la misma carpeta y haciendo subdivisiones dependiendo de a que objetos afecte.

En mi caso me he decantado por la otra alternativa, ordenar los archivos por la entidad que construyen, este enfoque me parece especialmente correcto para Godot ya que está fuertemente orientado a la composición de todos sus componentes sobre su extensión o su función.

A partir de la raíz de mi proyecto podemos ver varias subcarpetas para las siguientes categorías:

Addons: Es la carpeta por defecto donde se instalan los plugins así que la ignoraremos.

Android: Contiene información para el compilador de Android.

Exports: La ruta de destino de los binarios finales.

Locations: Los fondos del juego, en este caso solo existe la ciudad.

Obstacles: Los obstáculos que se encontrará el jugador y que deberá esquivar.

Player: El personaje que controla el jugadores.

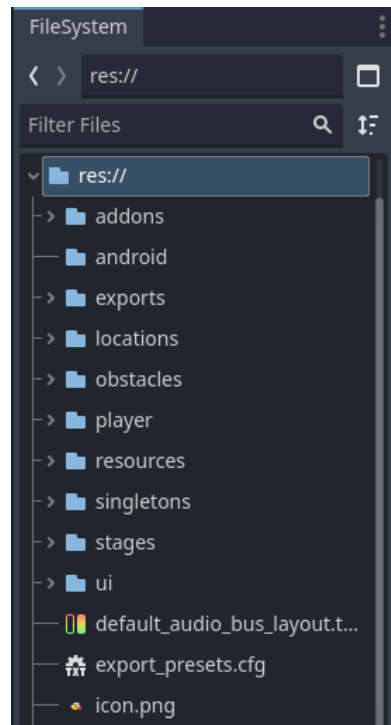
Resources: Recursos compartidos por todo el proyectos.

Singletons: Los singletons del proyecto.

Stages: Las pantallas que existen dentro del juego.

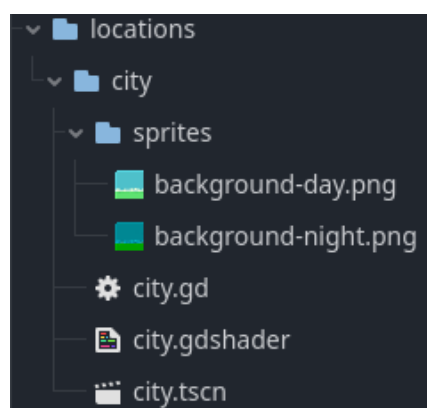
UI: Los elementos de la interfaz de usuario.

Dentro de la carpeta también podemos ver archivos importantes del proyecto como los buses de audio, el icono de la aplicación y la configuración de los compiladores.

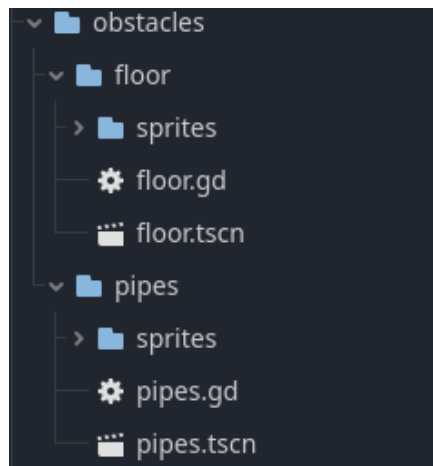


Ahora explicaré cómo he dividido cada una de estas carpetas y los parecidos que hay entre todas ellas antes de entrar en detalles sobre su comportamiento interno.

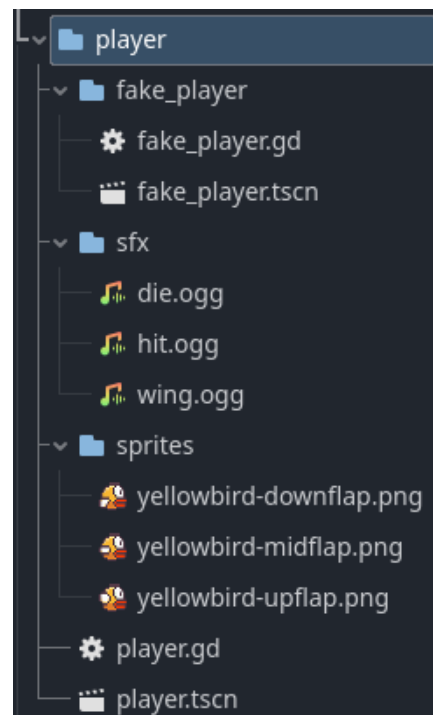
Empezando por la carpeta **locations**, en esta se guardan los fondos que ambientan el juego, en el caso de este proyecto solo existe city, la ciudad, que contiene su escena (acabada en .tscn), su script (acabado en .gd), las imágenes que usa en la carpeta *sprites* y un tipo de archivo que solo usé en este fondo, los shaders (acabado en .gdshader).



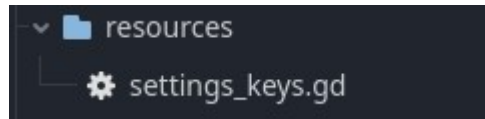
La siguiente carpeta es **obstacles**, donde se almacenan los obstáculos, esta contiene las tuberías y el suelo, ambos elementos con los que puede colisionar el jugador, como en el caso anterior cada una de ellas contiene una escena, un script y las imágenes que usan.



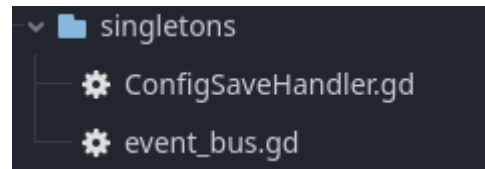
A continuación la carpeta **player** que contiene los archivos del personaje jugable, en esta podemos ver su escena, script y sus imágenes al igual que en las anteriores, además de esto tiene las subcarpetas *sfx* para los efectos de sonido que hace el jugador y *fake_player*, un clon del player sin ningún tipo de lógica usado como decoración en la pantalla de inicio.



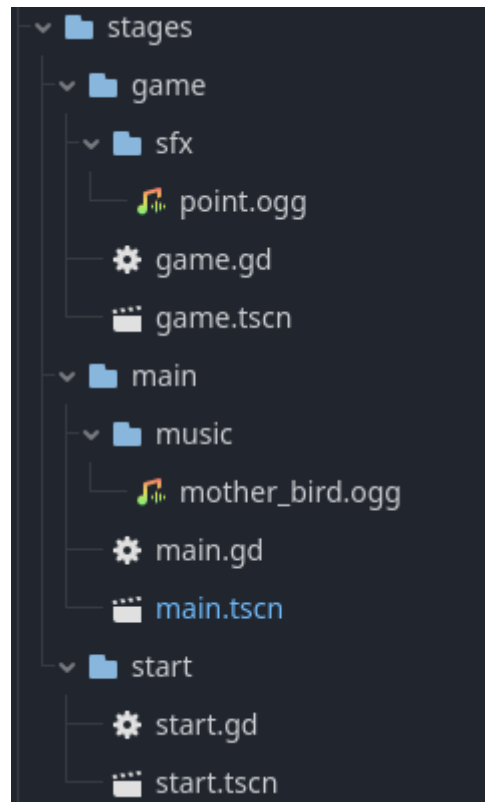
La carpeta **resources** contiene recursos que se van a usar globalmente en el proyecto, en este caso el único existente es el de las *keys* usadas para acceder al archivo de guardado de la configuración.



La siguiente carpeta contiene los dos **singletons** que usa el proyecto.

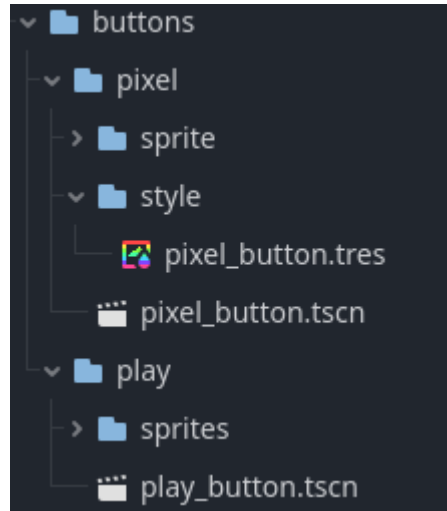


La carpeta **stages** contiene las escenas que hacen de pantallas en el juego, *game* es la que se usa durante una partida, *start* el menú de inicio y *main* es la pantalla que se carga al abrir el juego y sobre la que se instancian las otras cuando es necesario cambiar de escena, además esta última también contiene la música del juego.

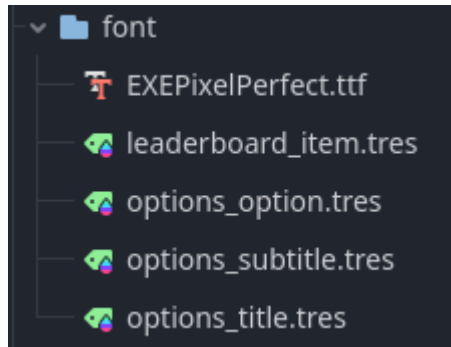


Dentro de la carpeta **ui** se guardan todos los elementos de la interfaz gráfica, son los siguientes:

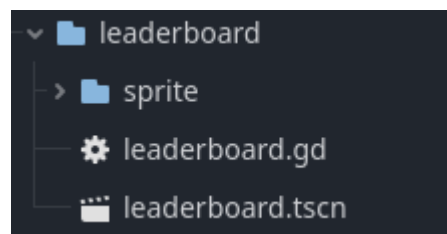
- **Buttons:** Contiene el botón usado para iniciar partidas y otro personalizado para que tenga la estética del juego original, este último tiene un recurso (acabado en .tres) encargado de modificar su estilo.



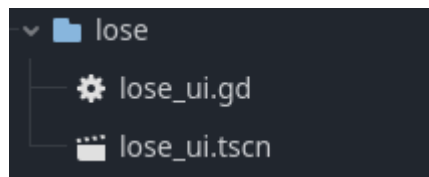
- **Font:** Donde se guarda la fuente con estética pixel art y recursos de nodos etiquetas para representar texto con esa misma estética.



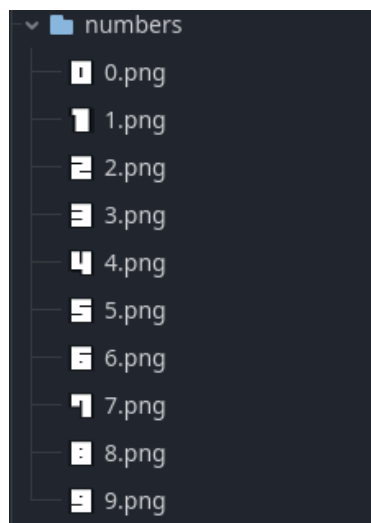
- **Leaderboard:** Contiene la escena usada para mostrar la tabla de puntuaciones.



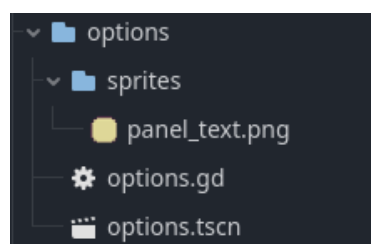
- **Lose:** Es el menú y título que aparecen cuando un jugador pierde una partida.



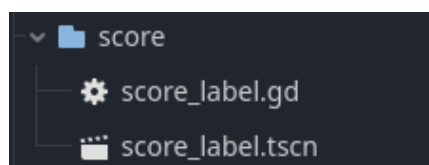
- **Numbers:** Contiene todos los números usados para mostrar la puntuación, deberían estar junto a la escena que se encarga de mostrar los puntos, pero como al principio no sabía si también los usaría en la tabla de puntuaciones los dejé en una carpeta a parte.



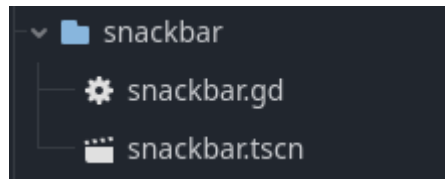
- **Options:** Es donde se guarda el menú de opciones.



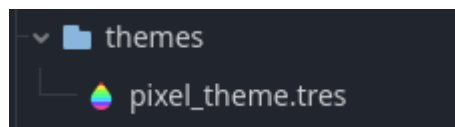
- **Score:** Es la escena encargada de mostrar la puntuación actual en la partida.



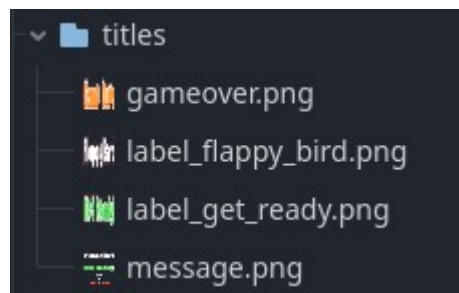
- **Snackbar:** Donde se guarda la escena para mostrar las notificaciones dentro del juego.



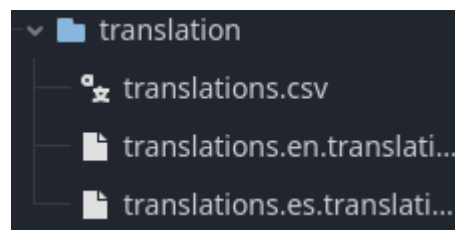
- **Themes:** El tema con estética pixel art que se usa en los elementos de la interfaz gráfica.



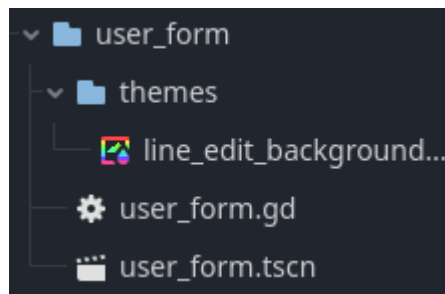
- **Titles:** Los mensajes importantes que se ven siempre en pantalla, como el título del juego y el mensaje de partida perdida.



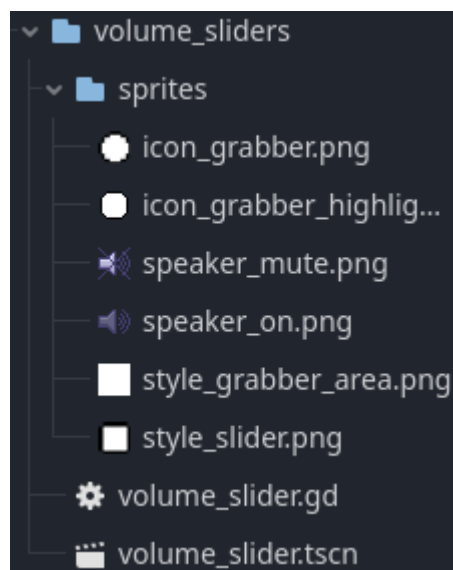
- **Translation:** El csv con todas las traducciones del juego y los archivos que genera el motor para su uso.



- **User Form:** Donde se guarda la escena del formulario para que los jugadores puedan registrarse e iniciar sesión, además también tiene un tema personalizado para los campos de texto que puede rellenar el jugador.



- **Volume Sliders:** Es la escena que se usa para que los jugadores puedan cambiar el nivel del audio.

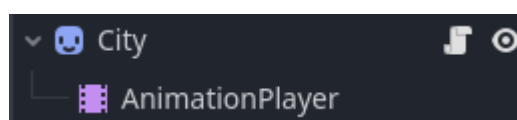


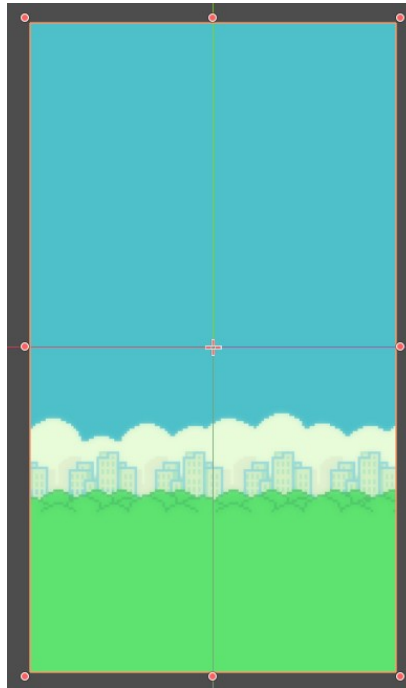
Composición de escenas y sus scripts:

En este apartado explicaré la composición de las escenas y de sus scripts en el mismo orden en el que se mostraron las carpetas y sus contenidos.

City:

El fondo de ciudad se compone por una imagen estática y un nodo de animación encargado de crear transiciones en las propiedades de otros nodos. Como podemos ver el primer nodo tiene un símbolo de un papel a su derecha, lo que indica que tiene un script que modifica su comportamiento base.





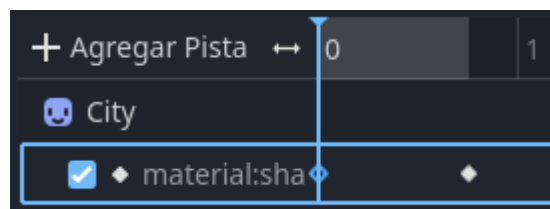
Al pulsar el icono del script se abre el editor de código y podremos ver su código, lo primero que podemos ver es la clase de la cual hereda su comportamiento marcado por la palabra ***extends***, seguido de dos variables, una booleana para controlar el estado de día/noche y una referencia al nodo de animación ambas precedidas por la palabra ***var*** para crear variables y la última también por ***@onready***, indicándole al motor que debe esperar a que todos los nodos hijos del actual y él mismo se carguen completamente antes de asignar el valor.

En GDScript las funciones se declaran usando ***func***, la primera que vemos es ***_ready***, una función de la clase ***Node*** de la cual derivan todos los nodos como los usados para construir escenas, esta función se ejecuta cuando el nodo y todos sus hijos entran en el árbol de nodos, es decir, se crean exitosamente, y solo se llama una vez en la vida del nodo, siendo similar a ***@onready***. Dentro de esta función se suele escribir el código necesario para configurar el nodo correctamente antes de usarlo, en este caso se llama a otra función que restablece los valores por defecto de las variables. Aunque para este nodo no sería necesario hacer este ajuste ya que las variables se inicializan con los valores adecuados la función ***reset*** es usada por otros nodos para reiniciar el juego más adelante.

La última función ***change_time*** activa una animación en el nodo de animaciones encargado de la transición entre el estado de día y noche.

```
city.gd
1 extends Sprite2D
2
3
4 var _day: bool = true
5
6 @onready var anim_player: AnimationPlayer = $AnimationPlayer
7
8
9 # Por defecto inicia con el fondo de día
10 func _ready() -> void:
11     reset()
12
13
14 func reset() -> void:
15     _day = true
16     anim_player.play("RESET")
17
18 # Cambia el fondo por el contrario
19 func change_time() -> void:
20     if _day:
21         anim_player.play("change_time")
22     else:
23         anim_player.play_backwards("change_time")
24
25
26     _day = !_day
```

Con el nodo de animaciones podremos seleccionar nodos y propiedades de estos en una línea de tiempo y asignarles distintos valores fijos a lo largo de esta, lo que la hace una buena herramienta para animaciones fijas que no requieran calcular los valores iniciales o finales en tiempo de ejecución, por ejemplo en la animación de **city** la propiedad de un **shader** cambia de 0 a 1 progresivamente hasta que acaba la animación, la cual podemos ver que dura unos 0.75 segundos. En las siguientes escenas no voy a hablar tanto de sus animaciones ya que son algo puramente estético y no requieren conocimientos de programación para realizarlas.



En el apartado anterior hablé de cómo cambiaba el valor de un **shader**, estos son unos programas que se ejecutan en la GPU y se utilizan para controlar la forma en la que se representan los gráficos, estos se ejecutan por cada pixel generado y son una forma muy precisa de generar detalles visuales que normalmente no serían posibles con la programación a la que estamos acostumbrados.

En este proyecto solo he tenido que usar este ya que quería crear una transición entre día y noche en la que se viese como cambia poco a poco, al no ser esto posible con el nodo de animación tuve que hacer el siguiente **shader** para fundir los píxeles de una imagen de día y otra de noche asignándoles distintas prioridades a uno u otro dependiendo del estado de la animación.

Para explicar cómo funciona al inicio del script cargo las dos imágenes y un tercer parámetro para controlar la cantidad de mezcla que necesito, estos parámetros se pueden editar desde fuera de este código y en tiempo de ejecución.

En la función **fragment** le indico que quiero ejecutar esa función para cada pixel de la imagen, similar a un bucle for, dentro de este obtengo el pixel actual de las imágenes haciendo uso de las coordenadas **UV**, similares a XY, y le asigno a **COLOR**, la variable del pixel por el que se llega el bucle, una mezcla de los píxeles obtenidos previamente con el valor de mezcla. Esta función se actualiza cada vez que en el nodo de animación cambia el valor de mezcla, por lo que está sincronizada con la animación que enseñe anteriormente.

```
shader_type canvas_item;

uniform sampler2D background_day;
uniform sampler2D background_night;
uniform float mix_amount = 0.0;

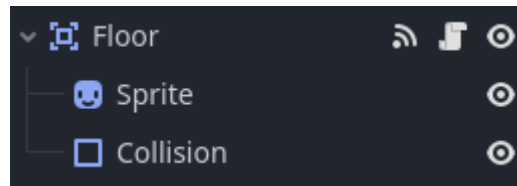
// Funcion para convinar ambas imagenes, usado para alternar entre ellas
void fragment() {
    >1 vec4 day = texture(background_day, UV);
    >1 vec4 night = texture(background_night, UV);
    >1 COLOR = mix(day, night, mix_amount);
}
```

Floor:

Esta es la escena que hace de suelo, además de dar la sensación de movimiento al desplazarse de derecha a izquierda también detecta colisiones con el jugador y emite señales cuando esto ocurre.

La raíz es un nodo de tipo **Area2D**, especializado en detectar la entrada y salida de otros objetos en una zona definida, en este caso esa zona es el nodo collision, de tipo **CollisionShape2D**, el cual puede crear formar áreas de diversas formas y tamaños que pueden utilizar otros nodos. El último nodo es otro **Sprite** para mostrar la imagen del suelo.

A la derecha del nodo **Floor** se puede ver que tiene un símbolo de ondas, esto indica que tiene emitida una señal que ha sido conectada con una función.



En la imagen inferior se puede ver una zona azul, esta es el área que crea el nodo collision.



En la primera parte del script podemos ver como se definen las variables y las señales, estas últimas se pueden crear usando la palabra **signal** y emitir usando el método **emit**, también hay variables que tienen la palabra **@export**, esta sirve para poder cambiar el valor de la variable desde el editor de escenas sin necesidad de abrir el código, es perfecto para hacer pruebas rápidas o para que divida el trabajo con alguien que no sepa programar.

En las funciones nuevas podemos ver **_process**, otra función de la clase **Node** la cual se ejecuta una vez por **frame** o fotograma que genera el motor, es ideal para todas las funciones que se tengan que ejecutar de forma constante a lo largo del programa y que no requieran cálculos de físicas. Esta función tiene la variable **delta** la cual nos dice cuanto tiempo ha pasado desde el último **frame** que se generó, esto es muy útil ya que cada procesador tiene capacidades de cálculo distintas y mediante esta variable nos podemos asegurar que los cambios que queramos realizar sean proporcionales con el tiempo que ha pasado, ralentizándolos en los ordenadores con componentes más potentes al ser delta un número menor y a la inversa en ordenadores más humildes.

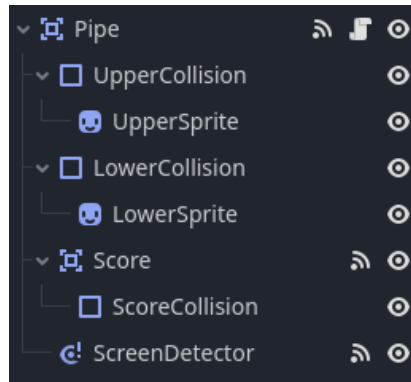
En el caso de este script se desplaza ligeramente la imagen del suelo hacia la izquierda con una velocidad marcada multiplicada por delta.

Finalmente la detección del nodo **Area2D** se conecta con su propio script para emitir la señal de jugador golpeado, de esta forma el código que deba ejecutarse cuando el jugador pierda es totalmente independiente del objeto que detectó la colisión.

```
floor.gd
1  extends Area2D
2
3  signal player_hitted
4
5  const MIN_OFFSET: float = 0
6  const MAX_OFFSET: float = -48
7
8  @export var sprite: Sprite2D
9  @export var speed: float = 2.0
10
11  var move: bool = true
12
13
14  # Al cargar la escena empieza en offset 0
15  func _ready() -> void:
16      >| sprite.offset.x = MIN_OFFSET
17
18
19  # Si el juego continua se mueve infinitamente el sprite de der. a izq.
20  func _process(delta: float) -> void:
21      >| if not move:
22          >| >| return
23      >|
24      >| sprite.offset.x -= speed * delta
25      >| sprite.offset.x = fmod(sprite.offset.x, MAX_OFFSET)
26
27
28  # Al detectar una colision con el player se emite fin del juego
29  func _on_body_entered(_body: Node2D) -> void:
30      >| player_hitted.emit()
```

Pipes:

Las pipes o tuberías son otro obstáculo con el que el jugador puede colisionar y perder la partida, estas se componen por un **Area2D** que cuenta con dos **CollisionShape2D** y un **Sprite2D** en cada una de ellas para hacer las colisiones de las tuberías y mostrar sus imágenes, también cuentan con una segunda área con su propia zona de colisión para detectar cuando el jugador ha conseguido pasar entre las tuberías, por último tiene un nodo **VisibleOnScreenNotifier2D** el cual emite señales cuando las tuberías entran y salen de pantalla.



En el código de **Pipes** podemos ver nuevas palabras reservadas, **class_name** nos permite guardar la escena como un nodo más haciendo que el motor la reconozca y nos genere una documentación sencilla de nuestras clases, esto es útil cuando una clase se va a usar mucho desde otra, la otra es **@export_category** la cual crea agrupaciones en las variables con **@export** para organizarlas mejor en el editor.

```
1 class_name Pipe
2 extends Node2D
3
4
5 signal player_hitted
6 signal player_scored
7 signal pipe_entered
8 signal pipe_exited()
9
10 @export var speed: float = 20.0
11 @export_category("Pipe height")
12 @export var min_height: int = -40
13 @export var max_height: int = 40
14 @export_category("Pipe space")
15 @export var min_space: int = 30
16 @export var max_space: int = 70
17
18 var move: bool = true
19 var spawn: int = 400
20 var upper_pipe_y: int
21 var lower_pipe_y: int
22
23 @onready var upper_pipe: CollisionShape2D = $UpperCollision
24 @onready var lower_pipe: CollisionShape2D = $LowerCollision
```

La siguiente parte del código es la lógica de movimiento y configuración de las tuberías, al entrar al árbol de nodos se guarda su posición original y se llama a la función de configuración, en esta se separan las tuberías entre si y se colocan a una altura aleatoriamente, asegurando de esta forma que cada tubería sea un obstáculo ligeramente distinto.

```

pipes.gd
27 # Configura los valores por defecto cuando cargan todos sus nodos
28 func _ready() -> void:
29     upper_pipe_y = upper_pipe.position.y
30     lower_pipe_y = lower_pipe.position.y
31     _set_pipe()
32
33
34 # Mueve las tuberías de der. a izq.
35 func _process(delta: float) -> void:
36     if not move:
37         return
38
39     position.x -= speed * delta
40
41
42 # Separa las tuberías entre sí y cambia su altura
43 func _set_pipe() -> void:
44     position = Vector2i(spawn, 200)
45     upper_pipe.position.y = upper_pipe_y
46     lower_pipe.position.y = lower_pipe_y
47
48     var space: int = randi_range(min_space, max_space)
49     upper_pipe.position.y -= space
50     lower_pipe.position.y += space
51
52     var height: int = randi_range(min_height, max_height)
53     position.y += height

```

La parte final del script son detecciones y emisiones de señales de distintos nodos.

```

56 # Emite la señal cuando el jugador cruza las tuberías
57 func _on_score_body_exited(_body: Node2D) -> void:
58     player_scored.emit()
59
60
61 # Emite la señal cuando el jugador choca con las tuberías
62 func _on_pipe_body_entered(_body: Node2D) -> void:
63     player_hitted.emit()
64
65
66 # Detecta cuando una tubería ha salido de pantalla
67 func _on_screen_exit_detected() -> void:
68     if position.x <= 0:
69         pipe_exited.emit()
70
71
72 # Detecta cuando una tubería ha entrado en pantalla
73 func _on_screen_entered_detected() -> void:
74     pipe_entered.emit()

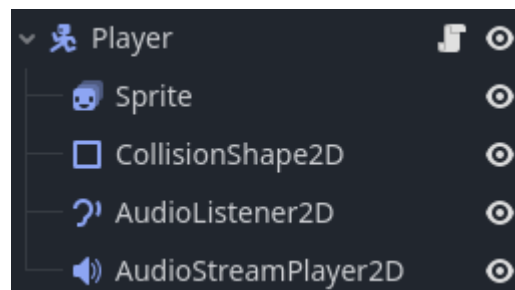
```

Player:

A partir de esta clase ignoraré los nodos de los que ya haya hablado.

Esta es la escena que controla el jugador, esta se compone por un **CharacterBody2D**, clase diseñada para responder a la entrada del jugador, especialmente para realizar

procesos que involucren físicas como el movimiento del jugador, también contiene un **AudioListener2D** y un **AudioStreamPlayer2D** encargados de detectar y reproducir sonidos respectivamente. En el **AudioPlayer** se pueden crear múltiples configuraciones de audio como sonidos individuales, listas de reproducción, transiciones y música reactiva con distintas pistas de audio, en este caso he creado una lista de reproducción con todos los efectos de sonido las transiciones entre ellos para poder activarlas desde código. El **AudioListener** permite interpretaciones complejas del sonido que se está reproduciendo en el juego, ya que por defecto sin este las configuraciones basadas en la posición entre el objeto que emite el sonido y el que lo recibe realizadas en el **AudioPlayer** no se pueden calcular, como el sonido de este juego es muy simple este nodo no sería necesario, pero lo he incluido para seguir las buenas prácticas.



El **Player** se caracteriza por la interacción que realiza el jugador con el juego a través de él, pero estas son complicadas de programar y cada nuevo estado e interacción suman una gran complejidad al código, por ello para evitar errores y facilitar la lectura del código suele ser una práctica común realizar máquinas de estados, he optado por una sencilla usando un **enum** con todos los estados posibles y un switch, que en GDScript se denominan **match**, esta es una de las formas más sencillas de realizar una máquina de estados.

La función **physics_process** pertenece a **Node** y es similar a **process**, su principal diferencia es que esta sirve para hacer cálculos que modifiquen las físicas de un nodo, en este se comprueba cual es el estado actual y se le aplican unas fuerzas u otras al jugador dependiendo de esto.

```
5  enum State {
6      FALL,
7      JUMP,
8      RISE,
9      DEAD,
10 }
```

```
25  # Calcula el movimiento del jugador en cada frame
26  func _physics_process(delta: float) -> void:
27      match actual_state:
28          # Rota el personaje cuando cae
29          State.FALL:
30              _handle_gravity(delta)
31              rotation_degrees = lerp(rotation_degrees, 35.0, 3.5*delta)
32          # Realiza el salto
33          State.JUMP:
34              sprite.play("jump")
35              velocity.y = jump_speed
36              actual_state = State.RISE
37              move_and_slide()
38          # Rota el personaje cuando vuela
39          State.RISE:
40              _handle_gravity(delta)
41              rotation_degrees = lerp(rotation_degrees, -45.0, 10.0*delta)
42              if velocity.y >= 0:
43                  actual_state = State.FALL
44          # Mueve al jugador al suelo y lo rota
45          State.DEAD:
46              if position.y != 390:
47                  position.y = move_toward(position.y, 390, 350*delta)
48                  rotation_degrees = lerp(rotation_degrees, 90.0, 8.0*delta)
```

La función **input** también es heredada de **Node**, está nos permite detectar eventos de entrada y reaccionar a ellos, en esta función compruebo que la tecla de **Jump** o salto, la cual he configurado previamente en el editor, haya sido pulsada y que el jugador se encuentre en un estado que le permita saltar.

```
54  # Detecta el evento de salto
55  func _input(event: InputEvent) -> void:
56      if actual_state == State.DEAD:
57          return
58      if event.is_action_pressed("Jump") and actual_state != State.JUMP:
59          actual_state = State.JUMP
60      # Cambia el tono ligeramente para que el sonido no sea repetitivo
61      audio_stream.pitch_scale = randf_range(0.8, 1.2)
62      audio_stream["parameters/switch_to_clip"] = "Jump"
```

Fake Player:

Es una versión más sencilla del jugador la cual no contiene su lógica, su función es decorativa y evita mezclar el código necesario para decorar con el usado para jugar. Tiene una estructura de un solo nodo **AnimatedSprite2D** el cual reproduce una animación en bucle y un **process** en el cual se mueve verticalmente para dar la sensación de vuelo.



Settings Keys:

Este es un recurso accesible por todos los scripts del juego con las constantes usadas para acceder y configurar el script de carga y guardado de la configuración del juego. Su principal función es evitar errores de escritura en las **keys** o llaves que se usaran para acceder a la configuración ya que esta es un mapa y el motor no detecta automáticamente sus posibles valores y llaves.

```
settings_keys.gd X
1  class_name SettingsKeys
2  extends Resource
3
4
5  const supported_locales: Array[String] = ["es", "en"]
6  const supported_languages: Array[String] = ["Español", "English"]
7  const default_locale: String = "en"
8
9  const volume: String = "volume"
10 const user: String = "user"
11
12 const master_vol: String = "master_volume"
13 const music_vol: String = "music_volume"
14 const sfx_vol: String = "sfx_volume"
15
16 const master_mute: String = "master_muted"
17 const music_mute: String = "music_muted"
18 const sfx_mute: String = "sfx_muted"
19
20 const jwt: String = "refresh_token"
21 const locale: String = "locale"
```

ConfigSaveHandler:

Este es un **singleton** que he creado para poder cargar, guardar y acceder a la configuración, en un proyecto de Godot se pueden modificar los singletons activos y su orden de ejecución desde las opciones del proyecto.

Lo primero que destaca es la constante para la ruta de guardado del archivo, el prefijo *user://* es interpretado por Godot como la carpeta de archivos de usuario, usa este prefijo genérico ya que el motor permite exportar a distintas plataformas con estructuras de carpetas diferentes, además la extensión del archivo es indiferente, pero las más usadas son *.ini* y *.cfg*.

```
8  const SETTINGS_FILE_PATH: String = "user://settings.ini"
```

Al inicio del **singleton** se conectan varias señales de control de sesión de la base de datos, después de esto se intenta leer el archivo, en caso de no encontrarse se crea uno nuevo con los valores por defecto y se guarda, en caso de existir se lee y cargan sus contenidos, además si contiene un **token** de sesión se conecta a la base de datos.


```

19 func _ready() -> void:
20     Supabase.auth.error.connect(_on_error)
21     Supabase.auth.token_refreshed.connect(_on_user_log)
22     Supabase.auth.signed_out.connect(_on_log_out)
23     Supabase.auth.signed_in.connect(_on_user_log)
24     Supabase.auth.signed_up.connect(_on_user_log)
25
26     if !FileAccess.file_exists(SETTINGS_FILE_PATH):
27         config.set_value(SettingsKeys.volume, SettingsKeys.master_vol, 0.5)
28         config.set_value(SettingsKeys.volume, SettingsKeys.master_mute, false)
29         config.set_value(SettingsKeys.volume, SettingsKeys.music_vol, 0.5)
30         config.set_value(SettingsKeys.volume, SettingsKeys.music_mute, false)
31         config.set_value(SettingsKeys.volume, SettingsKeys.sfx_vol, 0.5)
32         config.set_value(SettingsKeys.volume, SettingsKeys.sfx_mute, false)
33         config.set_value(SettingsKeys.user, SettingsKeys.jwt, "")
34         var user_locale: String = OS.get_locale_language()
35         config.set_value(
36             SettingsKeys.user,
37             SettingsKeys.locale,
38             user_locale if
39                 user_locale in SettingsKeys.supported_locales
40                 else SettingsKeys.default_locale
41         )
42         config.save(SETTINGS_FILE_PATH)
43         loading_state = LoadingState.LOADED
44         config_ended.emit()
45
46     else:
47         load_config()
48         # Inicia sesion a partir del jwt almacenado
49         var jwt := str(config.get_value(SettingsKeys.user, SettingsKeys.jwt))
50         if not jwt.is_empty():
51             loading_state = LoadingState.LOADING
52             Supabase.auth.refresh_token(jwt, 10.5)
53         else:
54             loading_state = LoadingState.LOADED
55         config_ended.emit()

```

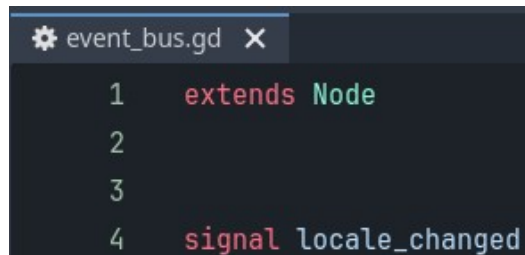
Al ser un diccionario y ser su asignación y lectura de valores bastante repetitiva voy a centrarme en su maquina de estados, esta se compone por tres, Leyendo, Cargando y Cargado, a lo largo de las configuraciones el estado del **singleton** se actualiza y al finalizar emite una señal indicándole a los demás nodos que ya pueden pedir o modificar los datos que contiene.

```
6 enum LoadingState {READING, LOADING, LOADED}
```

Event Bus:

El siguiente **singleton** es mucho más sencillo pero igual de útil, este es un bus de eventos accesibles desde cualquier parte del proyecto, contiene señales a las que cualquier nodo se puede conectar y también emitir. En la mayoría de casos no es

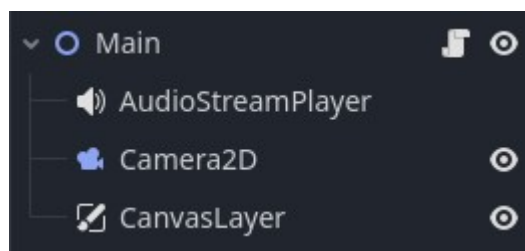
recomendable usar este tipo de buses ya que cada nodo debería contener sus propias señales y conectarse solo a las señales de sus hijos y hermanos, pero estas soluciones destacan en cambios globales como en el caso de cambio de idioma del juego completo, ya que permite conectar limpiamente todas las interfaces que contengan texto a esta señal.



```
1 extends Node
2
3
4 signal locale_changed
```

Main:

Esta es la escena principal y es la primera que se carga cuando se abre el juego, sirve como contenedor para el menú de inicio y la pantalla de juego además de encargarse de gestionar los cambios de escena y las notificaciones. Como se puede ver en la captura inferior todos lo que contiene es un **Node2D** simple junto con una **Camera2D** para definir el espacio visible y un **CanvasLayer**, un nodo especializado en representar la interfaz gráfica, este ignora la restricción de resolución y siempre se representa a la resolución nativa de la pantalla, mejorando la legibilidad de las tipografías cuando se juega a bajas resoluciones.



Sus variables contienen las escenas que se van a instanciar, en este caso la inicial es el menú de inicio, también almacena la escena actual para poder eliminarla cuando sea necesario realizar el cambio de escenas.

```
4  const INITIAL_STAGE: String = "res://stages/start/start.tscn"
5  const SNACK_BAR: PackedScene = preload("res://ui/snackbar/snackbar.tscn")
6  const SNACK_BAR_POS = Vector2(144, 516)
7
8  @export var canvas_layer: CanvasLayer
9
10 var stage: Node2D
11 var options_open: bool = false
12 var snack_bar: Snackbar
```

Dentro de **ready** se instancia la escena inicial y se carga en el árbol de nodos, además se busca y conecta una señal utilizada para realizar cambios de escena dentro de las hijas.

```
16 func _ready() -> void:
17     Supabase.auth.error.connect(_on_error)
18
19     var stage_res: PackedScene = load(INITIAL_STAGE)
20     stage = stage_res.instantiate()
21
22     if stage.has_signal("change_scene_requested"):
23         stage.connect("change_scene_requested", _change_scene)
24
25     add_child(stage)
```

Esta señal se conecta con el siguiente método el cual elimina la escena existente y la reemplaza por la nueva.

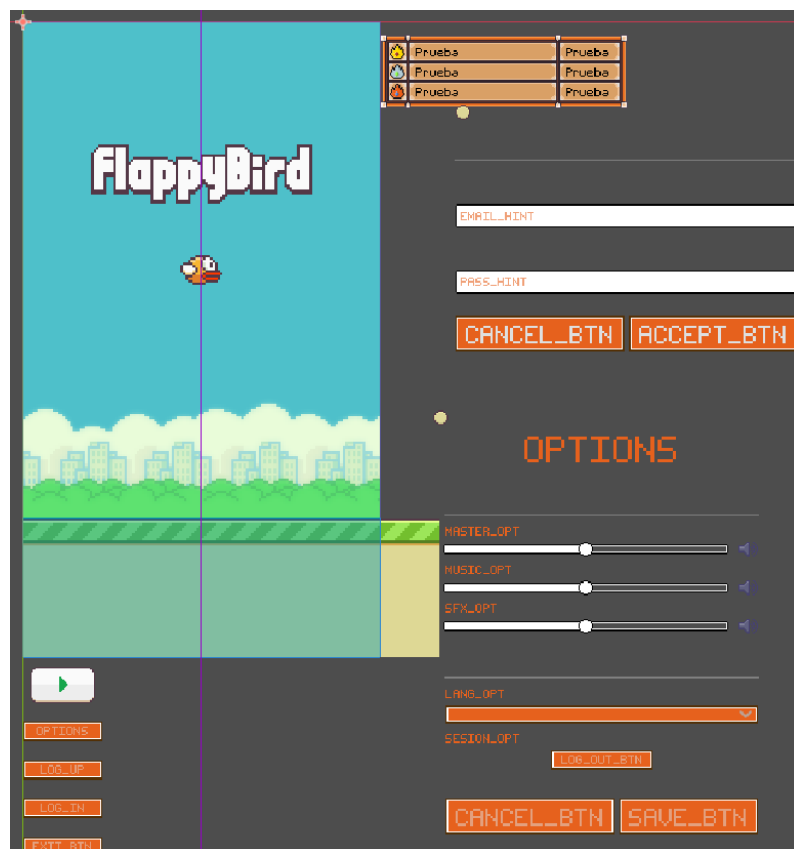
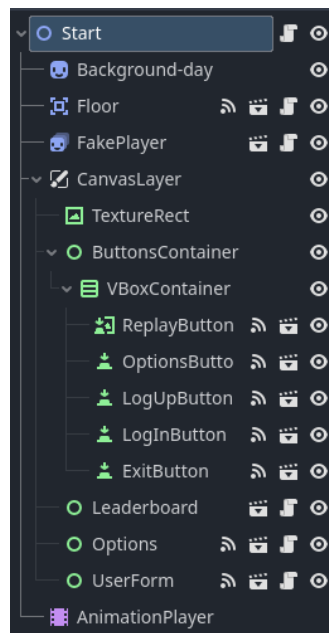
```
34 # Cambia la escena actual por la pedida en su señal de cambio
35 func _change_scene(new_scene: String) -> void:
36     var new_scene_res: PackedScene = load(new_scene)
37
38     # Elimina la escena existente
39     stage.disconnect("change_scene_requested", _change_scene)
40     remove_child(stage)
41     stage.queue_free()
42
43     # Carga la nueva en el arbol de nodos
44     stage = new_scene_res.instantiate()
45     if stage.has_signal("change_scene_requested"):
46         stage.connect("change_scene_requested", _change_scene)
47     add_child(stage)
```

Por último el método para las notificaciones, este está conectado a la señal de error de la API, obtiene le mensaje de este y comprueba si tiene una traducción antes de crear una nueva notificación.

```
50 # Muestra el mensaje de error en el snackbar
51 func _on_error(error: SupabaseAuthError) -> void:
52     print(error.code)
53     print(error.message)
54
55     var msg: String
56     match error.message:
57         "missing email or phone":
58             msg = "Missing email or phone"
59         "Invalid login credentials":
60             msg = "Invalid login credentials"
61         "Anonymous sign-ins are disabled":
62             msg = "Anonymous sign-ins are disabled"
63         "Signup requires a valid password":
64             msg = "Signup requires a valid password"
65         "Password should be at least 6 characters.":
66             msg = "Password should be at least 6 characters"
67         "User already registered":
68             msg = "User already registered"
69         _:
70             msg = "Unknown error"
71
72     if snack_bar != null:
73         snack_bar.queue_free()
74
75     snack_bar = SNACK_BAR.instantiate()
76     canvas_layer.add_child(snack_bar)
77
78     snack_bar.position = SNACK_BAR_POS
79     snack_bar.set_anchors_preset(Control.PRESET_CENTER_BOTTOM)
80     snack_bar.show_msg(msg)
```

Start:

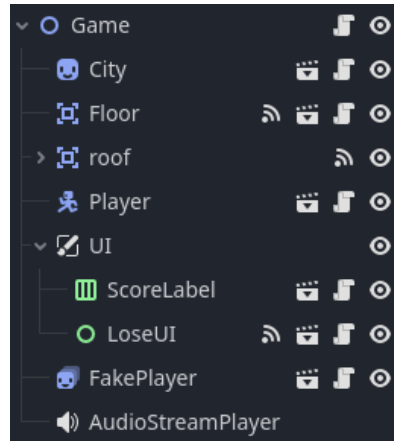
Esta escena contiene todos los menús de configuración y escenas que hemos visto anteriormente como el suelo y el jugador falso. Su lógica no es nada compleja por eso no entraré muy en detalle en ella, pero principalmente son funciones conectadas a las señales de todas las interfaces.



Game:

Esta escena contiene la lógica para generar los obstáculos que debe esquivar el jugador y los estados por los que pasa una partida.

La estructura de esta escena incluye varias de las escenas anteriores como los jugadores, el suelo y el fondo, además incluye un techo para que el jugador no pueda salir por la parte superior de la pantalla.



En el script las variables más importantes son los distintos estados en los que puede estar la escena, Esperando, Jugando y Muerto, la escena de las tuberías se precarga para poder instanciarla varias veces en tiempo de ejecución sin que haya tirones, también está el `initial_tick` el cual almacena el milisegundo en el que el jugador ha empezado a jugar para calcular el tiempo de partida.

```

4  signal change_scene_requested(new_scene)
5
6  enum GameState {
7      > WAIT,
8      > PLAY,
9      > DEAD,
10 }
11
12 const PIPES = preload("res://obstacles/pipes/pipes.tscn")
13
14 @export var audio_player: AudioStreamPlayer
15 @export var initial_speed: int = 60
16 @export var speed_add: int = 10
17 @export var pipe_spawn: int = 320
18 @export var score_diff_add: int = 15
19
20 var speed: int = initial_speed
21 var pipes: Array[Pipe] = []
22 var score: int = 0
23 var game_state: GameState = GameState.WAIT
24 var initial_tick: int

```

En la función **input** se detecta cuando el jugador pulsa el botón de saltar por primera vez, cuando esto pasa se cambia el estado de Espera a Jugando, el jugador falso se vuelve invisible y el autentico se vuelve visible y se coloca en la posición en la que estaba

el falso, también se guarda el milisegundo de la función antes de instanciar la primera tubería.

```
43 func _input(event: InputEvent) -> void:
44     if event.is_action_pressed("Jump") and game_state == GameState.WAIT:
45         # Muestra el player
46         player.position = fake_player.position
47         player.set_physics_process(true)
48         player.visible = true
49         player.audio_stream.play()
50
51         # Oculta el fake player y para sus procesos
52         fake_player.visible = false
53         fake_player.set_process(false)
54         fake_player.stop()
55
56         # Registra el tick de inicio
57         initial_tick = Time.get_ticks_msec()
58
59         # Cambia en estado de la escena
60         game_state = GameState.PLAY
61         _spawn_pipe()
```

La función **spawn_pipe** añade tuberías a la partida, para hacer esto instancia la escena que estaba definida en las variables, conecta las señales de esta a las funciones que tiene **Game**, ajusta su posición inicial, se añade al array que contiene todas las tuberías y finalmente se añade al árbol de nodos.

```
100 func _spawn_pipe():
101     var pipe: Pipe = PIPES.instantiate()
102
103     pipe.player_scored.connect(_on_point_scored)
104     pipe.player_hitted.connect(_on_player_hitted)
105     pipe.pipe_entered.connect(_on_pipe_screen_entered)
106     pipe.pipe_exited.connect(_on_pipe_screen_exited)
107
108     pipe.spawn = pipe_spawn
109     pipe.speed = speed
110     pipes.append(pipe)
111     add_child(pipe)
```

La función **on_point_scored** está conectada a la señal que envían las tuberías cuando el jugador las atraviesa, suma un punto a la puntuación total, actualiza la etiqueta de puntuación y reproduce un sonido. Después de esto comprueba si la puntuación es módulo de 15 ya que cada esta cantidad de puntos aumenta la dificultad del juego haciendo que las tuberías se desplacen más rápido, además es aquí donde se llama a la función de la ciudad que cambia el fondo entre día y noche.

```
129  ▾ func _on_point_scored() -> void:
130  ▾  >|  if game_state != GameState.PLAY:
131  >|  >|  return
132  >|
133  >|  score += 1
134  >|  score_label.set_numbers(score)
135  >|  audio_player.play()
136  >|
137  ▾  >|  if score % score_diff_add == 0:
138  >|  >|  speed += speed_add
139  >|  >|
140  ▾  >|  >|  for pipe in pipes:
141  ▾  >|  >|  >|  if pipe != null:
142  >|  >|  >|  >|  pipe.speed = speed
143  >|  >|
144  >|  >|  game_floor.speed = speed
145  >|  >|  city.change_time()
```

La función **on_pipe_screen_entered** está conectada a la señal que emiten las tuberías cuando entran en pantalla, llama a la función **spawn_pipe** para que siempre haya una tubería fuera de pantalla en dirección hacia el jugador.

```
166  ▾ func _on_pipe_screen_entered() -> void:
167  >|  _spawn_pipe()
```

La función **on_pipe_screen_exited** está conectada a la señal que emiten las tuberías cuando salen de pantalla, cuando esto pasa elimina la tubería del juego y su referencia del array.

```
171  ▾ func _on_pipe_screen_exited() -> void:
172  >|  pipes[0].queue_free()
173  >|  pipes.remove_at(0)
```

La función **_on_player_hitted** está conectada a las señales del techo, suelo y tuberías que se emiten cuando el jugador se choca con ellos, cuando esto pasa estado cambia a Muerto, el suelo y las tuberías dejan de moverse, se llama a la función del jugador para dejar de registrar inputs y que reproduzca la animación de caer al suelo y el sonido de

choque, también la del menú de pérdida para poder reiniciar la partida o salir al menú de inicio, por último envía el registro de la partida al servidor con las siguiente función.

```
149 func _on_player_hitted() -> void:
150     if game_state != GameState.PLAY:
151         return
152
153     game_state = GameState.DEAD
154
155     game_floor.move = false
156     for pipe in pipes:
157         if pipe != null:
158             pipe.move = false
159
160     _send_game()
161     player.kill()
162     lose_ui.show_ui()
```

La función **send_game** se encarga de hacer un insert en la base de datos de Supabase, si el usuario no está registrado en la aplicación esta función se salta.

```
115 func _send_game() -> void:
116     if Supabase.auth.client == null:
117         return
118
119     Supabase.database.query(SupabaseQuery.new()
120         .from("games")
121         .insert([
122             {
123                 score = score,
124                 duration = Time.get_ticks_msec() - initial_tick
125             }
126         ]))
```

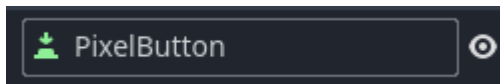
Si el usuario quiere jugar otra partida seguida se llama a la función **reset** la cual deja la escena en el mismo estado que estaba al entrar en ella.

UI:

En esta sección muchos de los controles son muy similares, por ello agruparé los que sean similares en estructura y lógica.

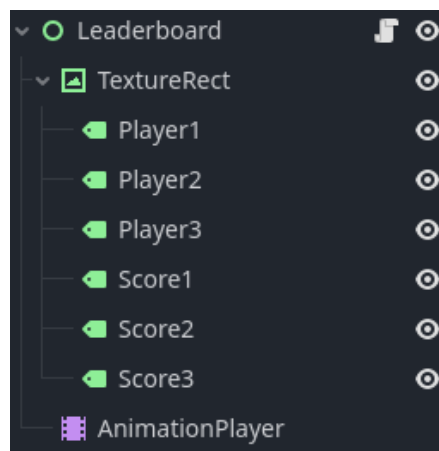
Botones:

Los botones son escenas de un simple nodo de tipo **Button** para los botones con texto y **TextureButton** para el que tiene la imagen de jugar, estos no tienen código ya que con las señales que tienen por defecto es suficiente para la interacción con el usuario. Los botones pueden detectar clics de ratón, pulsaciones en pantallas táctiles y además se les pueden asignar atajos de teclado.



Leaderboard:

La tabla de puntuaciones se compone de una textura que adorna su fondo y seis etiquetas, tres para los usuario y las otras para sus puntuaciones, además tiene un nodo de animaciones para hacer que aparezca en pantalla cuando termine la consulta a la API.



El código de **ready** es una consulta a la API en la cual se pide las tres primeras partidas ordenadas por puntuación y duración, cuando la consulta se realiza correctamente se emite una señal la cual conecta con la función que escribe los datos en las etiquetas de la tabla de puntuaciones.

La función **update_leaderboard** escribe los datos obtenidos en los nodos, para hacer esto usa un bucle for para iterar por el array devuelto, este es un diccionario con el correo electrónico y la puntuación, al acabar se reproduce la animación para mostrar las puntuaciones.

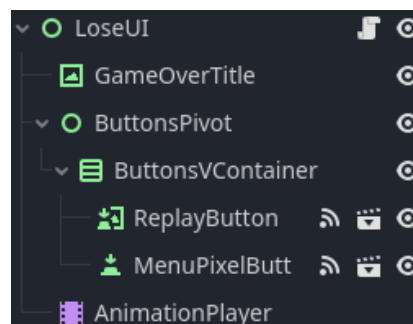
```

10 func _ready() -> void:
11     Supabase.database.selected.connect(_update_leaderboard)
12     Supabase.database.query(SupabaseQuery.new()
13         .from("games")
14         .select(["score", "user(email)"])
15         .order("score", SupabaseQuery.Directions.Descending)
16         .order("duration")
17         .range(0, MAX_USER - 1)
18     )
19
20
21 # Escribe los datos obtenidos en las etiquetas
22 func _update_leaderboard(result: Array) -> void:
23     for index in range(MAX_USER):
24         var player_lbl: Label = get_node("TextureRect/Player" + str(index+1))
25         var score_lbl: Label = get_node("TextureRect/Score" + str(index+1))
26
27         if result.size() > index:
28             var score = result[index]
29             player_lbl.text = score["user"]["email"].split("@")[0]
30             score_lbl.text = str(score["score"] as int)
31         else:
32             player_lbl.text = "-----"
33             score_lbl.text = "-----"
34
35     anim_player.play("show_leaderboard")
36

```

Lose UI, Options y User Form:

En las estructuras de estas tres escenas se usan multiples **VBoxContainer** y **HBoxContainer**, estos sirven para alinear nodos hijos en vertical y horizontal respectivamente, todos los elementos del menú están organizados con estos nodos.



En cuanto al código la mayoría de funciones son conexiones con las señales de los elementos de la interfaz, cuando se recibe un cambio estas funciones se encargan de guardar el nuevo valor en la configuración y en el juego, estos cambios se pueden fijar usando el botón de *guardar* o revertir con el de *cancelar*.

```
156 func _on_master_slider_muted_pressed(muted: bool) -> void:
157     ConfigSaveHandler.set_setting(
158         SettingsKeys.volume,
159         SettingsKeys.master_mute,
160         muted
161     )
162
163     AudioServer.set_bus_mute(master_bus, muted)
```

Otro método único de estas interfaces es el ajustar el tamaño de su fondo dinámicamente, para hacer esto se usa un nodo de tipo **NinePatchRect** en el cual se puede definir una textura con sus bordes, esquinas y centro de forma que puede adoptar varios tamaños repitiendo la textura necesaria hasta ocupar todo el espacio, este tipo de nodos es ideal para darles texturas personalizadas a fondos que pueden variar en tamaño. Cuando el método de ajuste detecta el cambio de tamaño del contenedor hijo mediante una señal se le calcula al fondo un nuevo tamaño añadiéndole márgenes, evitando así que al haber cambios en la interfaz se salgan elementos del menú.

```
233 # Cambia el tamaño del fondo cuando VBox cambia de tamaño
234 func _on_options_container_resized() -> void:
235     patch_rect.size = v_box.size + Vector2.ONE * MARGIN_AREA * 2
236
237     center_pos = (get_viewport().get_visible_rect().size - patch_rect.size) / 2.0
238     hidden_pos.x = center_pos.x
239
240     position = center_pos if vis else hidden_pos
```

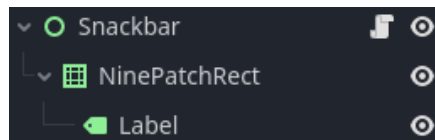
Score:

La escena usada para mostrar la puntuación está formada por un **HBoxContainer** cuyo contenido se genera de forma dinámica dependiendo de la cantidad de cifras que tengan los puntos, para hacer esto instancia y sobrescribe múltiples nodos **TextureRect** los cuales pueden mostrar en pantalla imágenes en 2D, les asigna su imagen correspondiente al convertir el número en una cadena e iterar sobre sus caracteres, usando estos mismo para identificar las referencias a las imágenes almacenadas en un diccionario.

```
30 func set_numbers(score: int) -> void:
31     var value: String = str(score)
32
33     for i in range(len(value)):
34         if len(numbers) > i:
35             var number: TextureRect = numbers[i]
36             number.texture = NUMBERS_TEXTURES[value[i]]
37         else:
38             var number: TextureRect = TextureRect.new()
39             number.texture = NUMBERS_TEXTURES[value[i]]
40             numbers.append(number)
41             add_child(number)
```

Snackbar:

Esta es la escena usada para mostrar las notificaciones del juego, se compone de una etiqueta con el error y su fondo se ajusta de forma dinámica de la misma forma que en los menús.



Al usar tamaños dinámicos no se puede crear su animación con el **AnimationPlayer**, por ello se usa un **Tween**, un objeto capaz de intercalar valores en una variable a partir de su valor inicial y el final que se les asigna, la ventaja que ofrecen es que este último puede calcularse en ejecución. Para crear el orden de ejecución de la animación se añaden propiedades a animar usando la función **tween_property** del tween, a esta se le asigna el nodo objetivo, su propiedad a modificar, el valor final que debe tener y el tiempo que debe durar la transición, también se le pueden añadir tiempos de espera con **set_delay** para que espere esa cantidad de segundos antes de ejecutar la transición que le precede. En caso de querer ejecutar código extra cuando la animación acaba se le puede pasar la referencia a una función o una lambda con **tween_callback**, en mi caso la uso para eliminar la notificación de memoria una vez se ha vuelto a ocultar. Para que la animación empiece a reproducirse hay que llamar al método **play**.

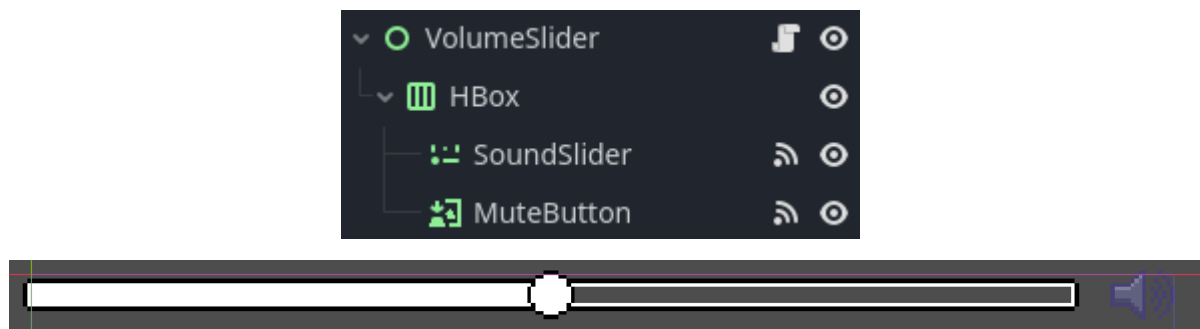
```

33 func _anim_play() -> void:
34     var tween: Tween = create_tween()
35
36     var original_pos: Vector2 = position
37
38     tween.tween_property(self, "position", Vector2(position.x, position.y - show_margin), 0.3 )
39     tween.tween_property(self, "position", original_pos, 0.3).set_delay(snackbar_duration)
40     tween.tween_callback(queue_free)
41
42     tween.play()

```

Volume Slider:

Es el control usado para modificar los decibelios de los buses de sonido, se compone por un botón que silencia el bus y por un **HSlider**, un nodo que permite seleccionar un valor dentro de un rango a partir de un mínimo y máximo en una disposición horizontal. El *slider* tiene señales las cuales emite cuando su valor cambia y con estas también envía el nuevo valor, es gracias a esta señal que en el menú de opciones se puede guardar el valor en la configuración.



6.4.3 Cuestiones de diseño e implementación reseñables.

Algo a destacar es que el plugin de Supabase, aunque muy útil, carece de documentación completa que explique detalladamente su uso, por lo que lo poco que pude aprender de ella es la instalación y las clases existentes, para poder hacer uso del plugin completo e implementar todas funcionalidades que tenía en mente he tenido que leer todo el código para entender su funcionamiento y poder aplicarlo en el juego.

6.5 Pruebas.

Las pruebas que se han realizado han sido en su mayoría probar la última funcionalidad añadida interactuando manualmente con el juego.

Al acabar el juego también he probado que todas las versiones compiladas se ejecutasen correctamente y realice varias pruebas visuales con la resolución y el escalado de pantalla del juego en distintos dispositivos, ya que Godot permite usar escalados enteros^[2] y fraccionarios^[3], optando por esta última.

7 Manuales de usuario

7.1 *Manual de usuario*

Lo primero que vemos al abrir el juego es el menú de inicio con varios botones, uno por uno su uso es el siguiente:

- **Jugar:** Es el botón sin texto que tiene un triángulo verde, este se usa para empezar una partida la cual se detallará más adelante.
- **Opciones:** Abre el menú de opciones en el cual podremos configurar varios aspectos de la aplicación.

La primera sección que se ve es el control de volumen, se puede alterar el volumen de las pistas de audio General, Música y SFX (Efectos Especiales) desde aquí, se usan las barras horizontales para cambiar el volumen entre el 0% y el 100%, también incluyen un botón para silenciar y reactivar completamente la pista.

La siguiente es la sección de usuario en la cual se puede cambiar el idioma de la aplicación con el desplegable de idioma y cerrar la sesión de usuario en caso de haber iniciado una.

Al final aparecen dos botones ambas cierran el menú pero Cancelar revierte los cambios realizados mientras que Guardar los almacena para las siguientes veces que se abra la aplicación.



- **Registrarse / Iniciar Sesión:** Abre el formulario con el cual los usuarios pueden registrarse en la base de datos, ambos son iguales y se componen por un campo email en el cual el usuario puede introducir su correo electrónico al clicar o pulsar encima de él y escribiendo después con el teclado, el segundo campo para contraseña funciona igual pero al ser datos sensible tapa el contenido usando asteriscos.

Abajo tiene dos botones, Cancelar cierra el formulario sin iniciar sesión mientras que Aceptar se conecta con la base de datos para hacerlo.

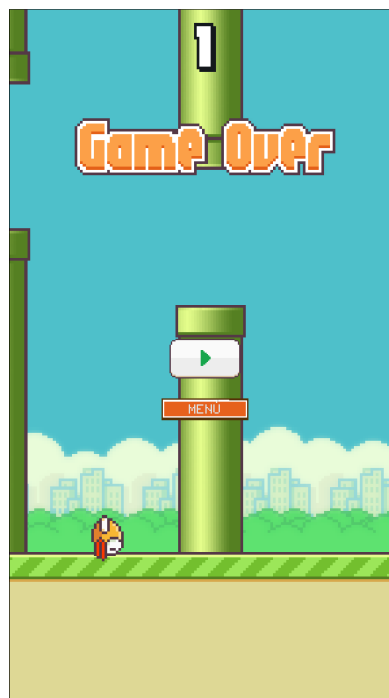
Cuando hay algún error en el inicio como usar una contraseña poco segura o registrarse con un correo ya registrado aparecerá en la parte inferior de la pantalla una notificación informando del error.



- **Salir:** Cierra el juego, este botón no está disponibles en Android ya que estos dispositivos suelen contar con formas propias de cerrar aplicaciones.

Si hemos pulsado el botón de Jugar empezara una animación que ocultará los elementos de la interfaz y moverá el personaje a la izquierda, cuando deje de moverse horizontalmente podremos pulsar la tecla de espacio, clic izquierdo o tocar la pantalla en caso de estar en móvil para comenzar el juego, tendremos que repetirlo para evitar que el personaje toque el suelo controlando que no llegue al techo o que choque con una tubería, el juego seguirá hasta que esto pase.

Cuando un jugador se choca aparecen dos botones, el primero reinicia la pantalla de juego y nos deja en el mismo momento que al pulsar Jugar en el menú de inicio, el segundo botón con el texto Menú nos devuelve al menú de inicio.



7.2 Manual de instalación

Los pasos para instalar la aplicación son los siguientes:

1. Abrir la pestaña de lanzamientos o releases del repositorio en GitHub ([enlace a las releases](#))
2. Seleccionar la primera versión que aparezca
3. Dirigirse al apartado de *Assets* donde veremos los distintos ejecutables
4. Seleccionar tipo de archivo dependiendo de nuestro sistema operativo y procesador y clicar sobre él para comenzar la descarga
 - **.exe** para la versión de Windows con amd64

- **.apk** para Android con armx64
- **comp.apk** para dispositivos Android antiguos incompatibles con vulkan
- **x86_64** para Linux con amd64
- **debug.exe** para la versión de Windows con amd64 modificada con cambios de dificultad más rápidos
- **debug.x86_64** para la versión de Linux con amd64 modificada con cambios de dificultad más rápidos

Una vez finalizada podremos abrir la aplicación con el dispositivo que hayamos elegido y se abrirá o instalará dependiendo de las necesidades de sistema.

8 Conclusiones y posibles ampliaciones

He aprendido mucho sobre el motor Godot, la lógica necesaria para unir las distintas piezas del juego y el cómo organizar mis proyectos de ahora en adelante realizando este trabajo, ha sido una buena experiencia con la que me he divertido aprendiendo y que definitivamente usaré a futuro.

Algunas ampliaciones que me gustaría haber realizado son las siguiente:

- Añadir movimiento a la cámara cuando el jugador choca con un obstáculo
- Traducir el juego a más idiomas
- Configurar Supabase para que me permita desconectar sesiones de dispositivos individuales en vez de toda la cuenta
- Configurar Supabase para que me permita añadir usuario con un nombre de jugador en vez de extraerlo del correo electrónico
- Añadir más tipos de obstáculos para los aumentos de dificultad
- Usar imágenes más grandes para que el escalado fraccionario no sea necesario
- Un apartado adicional en el que un usuario pueda ver todas sus puntuaciones
- Refactorizar el código para hacer mejor uso de la composición característica de Godot

9 Bibliografía

Documentación Godot: <https://docs.godotengine.org/en/stable/>

Documentación plugin Supabase: <https://github.com/supabase-community/godot-engine.supabase/wiki>

Metodología: <https://asana.com/es/resources/waterfall-project-management-methodology>

Crear trigger en Supabase: <https://stackoverflow.com/questions/77751173/getting-auth-users-table-data-in-supabase>

10 Anexos

Pixel art: Forma de arte digital en la que las imágenes se crean y editan a nivel de píxel individual, generalmente con una resolución baja y una paleta de colores limitada, lo que da como resultado un estilo visual distintivo y retro.

Escalado entero: En juego pixel art con pantallas con mayor resolución que la interna cada píxel se multiplica un número entero de veces hasta llenar la pantalla logrando ampliar la imagen manteniendo su nitidez y píxeles perfectos.

Escalado fraccionario: En un juego pixel art con pantallas con mayor resolución que la interna cada píxel se multiplica por el número que consiga rellenar la pantalla completamente pudiendo no ser un entero, consigue que el juego ocupe todo el espacio de pantalla a costo de posibles desperfectos visuales.