ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Graduado en Ingeniería Informática

Implementación de un videojuego serio enfocado en desarrollar el oído musical

Implementation of a serious video game focused on training musical ear

Realizado por
Francisco José Aragonés de la Rosa

Tutorizado por
Antonio José Fernández Leiva
Isabel Barbancho Pérez

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, mayo de 2023

Resumen

La meta de este proyecto es el desarrollo de un videojuego serio que se use como recurso para ayudar a mejorar la capacidad auditiva en el ámbito musical. El juego está diseñado para ser una herramienta útil para músicos y estudiantes de música, así como para cualquier persona interesada en mejorar su capacidad auditiva en el ámbito musical.

Para conseguir el objetivo, el juego se centra en un entrenamiento auditivo para mejorar la percepción musical del usuario, que incluye el reconocimiento de notas y el desarrollo de la memoria musical. Para ello, el juego cuenta con dos fases:

En la primera el jugador usa el reconocimiento de notas para memorizar las notas que suenan en un patrón.

En la segunda fase, el jugador introduce el patrón en el orden correcto pulsando las notas. Esta metodología ayuda al jugador a reconocer las diferencias entre las distintas notas.

Palabras clave: videojuego, juego serio, oído musical, ámbito musical.

Abstract

The goal of this project is the development of a serious video game that serves as a resource to help improve auditory skills in the musical field. The game is designed to be a useful tool for musicians and music students, as well as anyone interested in improving their auditory skills in the musical realm.

To achieve this objective, the game focuses on auditory training to improve the user's musical perception, including note recognition and the development of musical memory. To accomplish this, the game has two phases:

In the first phase, the player uses note recognition to memorize the notes that are played in a pattern.

In the second phase, the player inputs the pattern in the correct order by pressing the notes. This methodology helps the player to recognize the differences between the different notes.

Keywords: video game, serious game, musical ear, musical realm.

Índice

Índice de figuras

Índice de tablas

1. Introducción

En este apartado presentaremos al lector las motivaciones, objetivos y la estructura del presente documento.

1.1 Motivación

Nuestra motivación para este trabajo fin de grado es abordar la necesidad de desarrollar una herramienta efectiva y accesible que permita mejorar la capacidad auditiva en el ámbito musical. El oído musical es una habilidad esencial para cualquier músico o estudiante de música, y es una capacidad que puede ser difícil de desarrollar sin una guía adecuada.

1.2 Objetivos

Nuestro objetivo para este proyecto es abordar la necesidad de mejorar la percepción musical mediante el desarrollo de un videojuego serio. Para ello, nos enfocamos en combinar técnicas de entrenamiento auditivo con elementos lúdicos para crear una experiencia atractiva y efectiva para el usuario. Creemos que esta combinación puede ser una herramienta valiosa para músicos y estudiantes de música, así como para cualquier persona interesada en mejorar su capacidad auditiva en el ámbito musical.

1.3 Estructura

- Apartado 2 **Antecedentes**: En este apartado abordaremos el concepto de juego serio y como se relaciona con nuestro proyecto, explicaremos los conceptos necesarios para comprenderlo, hablaremos sobre el entorno de trabajo y la metodología seleccionada, así como las tecnologías usadas, además de los recursos disponibles usados este proyecto.
- Apartado 3 **Diseño:** En esta fase detallaremos el diseño del videojuego mediante el documento de diseño que incluye todas las mecánicas de juego, los elementos de la interfaz de usuario, el diseño de niveles y otros aspectos fundamentales para la jugabilidad del juego. Este documento será nuestra guía para llevar a cabo la implementación de manera coherente con la visión general del videojuego. Además explicaremos detalles sobre estructuras de datos utilizadas, así como la planificación seguida.

- Apartado 4 **Implementación:** Después de haber finalizado la fase de diseño del videojuego, nos enfocaremos en la implementación de todas las mecánicas e interfaces que hemos descrito en detalle en el documento de diseño.
- Apartado 5 **Conclusiones:** Compartiremos nuestra experiencia de aprendizaje personal adquirida durante el proyecto, así como los desafíos técnicos que enfrentamos y las soluciones que encontramos para superarlos. También destacaremos las mejoras futuras que consideramos para el videojuego.

2. Antecedentes

En este apartado desarrollaremos los conceptos que necesita el lector para comprender completamente nuestro proyecto.

En primer lugar, hablaremos del concepto de juego serio y demás conceptos necesarios para el lector, describiremos la metodología que hemos seguido, así como los diferentes roles que han participado en él. Además, proporcionaremos información sobre las tecnologías que hemos utilizado y los recursos que hemos tenido disponibles durante su desarrollo.

2.1 Juegos serios

Los juegos serios [1] son juegos diseñados con el propósito principal de educar, entrenar o informar a los usuarios, en lugar de ser simplemente entretenimiento. Se utilizan en una amplia variedad de campos, desde la educación y la formación hasta la salud y el bienestar, y se han vuelto cada vez más populares en las últimas décadas. Estos juegos serios utilizan la tecnología para involucrar a los usuarios en experiencias interactivas que les permiten aprender y practicar habilidades y conocimientos relevantes para su campo.

Nuestro proyecto se enmarca en el campo del aprendizaje, más concretamente en el ámbito de la educación musical. Esta herramienta tiene como objetivo el entrenamiento del oído a través de elementos visuales que estimulan la cognición, siendo de interés tanto para estudiantes de música como para cualquier persona interesada en mejorar su habilidad auditiva.

2.2 Conceptos

En este subapartado, recopilaremos algunos de los términos y siglas relacionados con los videojuegos y su desarrollo.

Concepto	**Definición**
GDD	Siglas de *Game Design Document*. Se trata del documento que describe todo el diseño del juego.
Wireframe	Representación visual simplificada de la estructura y diseño de la interfaz de usuario.
Sprite	Elemento gráfico en un videojuego que representa un personaje, objeto u otro elemento interactivo.

HUD	Siglas de *Heads-up display. En* una interfaz gráfica que muestra información relevante al jugador en tiempo real.
SCRUM	Metodología de trabajo ágil para la gestión y desarrollo de proyectos de software.
GameObject	elemento básico que representar cualquier entidad dentro del juego.
FBX	Formato de archivo utilizado para el intercambio de datos 3D.
Visión ortográfica	Técnica utilizada para representar objetos en una vista bidimensional sin perspectiva, manteniendo las proporciones y la escala correctas.
API	Siglas de *Application Programming Interface. E*s una interfaz que permite a los desarrolladores acceder y utilizar las funciones y datos de una aplicación.
UI	Siglas de *User Interface*. Se refiere a todos los elementos que un usuario ve y con los que interactúa.
Sharepoint	Es una plataforma de colaboración y gestión de intercambio de contenidos.

Tabla 2.1: Tabla de conceptos

2.3 Metodología

Como comentamos anteriormente, vamos a seguir una metodología ágil de desarrollo de software, en concreto SCRUM. Esto no permite poner especial atención a todas las etapas del desarrollo cubriendo todas las fases de diseño y los cambios que surjan durante el proyecto.

2.3.1 SCRUM

La metodología SCRUM [2] es un marco de trabajo ágil para la gestión y desarrollo de proyectos, especialmente en el ámbito del software. Se basa en la idea de dividir el trabajo en iteraciones cortas y fijas de tiempo, llamadas *sprints*, en las que se planifican, desarrollan, prueban y entregan pequeñas porciones de trabajo.

2.3.2 Conceptos

- **Sprint**
  Iteraciones cortas y fijas de tiempo, en concreto dos semanas.

- **Product backlog**
  Lista priorizada de todas las funcionalidades, características, mejoras y arreglos de errores que se deben implementar en el proyecto. Es la fuente única de requisitos para cualquier cambio en el producto.
- **Sprint backlog**
  Es una lista de tareas seleccionadas del *Product Backlog* para ser completados durante un Sprint.
- **Increment**
  Se refiere al resultado tangible y potencialmente entregable de un Sprint. Cada incremento se suma a los incrementos anteriores, lo que crea una versión cada vez más completa y avanzada del producto final.

2.3.3 Roles

- **Product owner**
  Es uno de los roles principales en la metodología Scrum. Este actúa como cliente con la autoridad sobre todo el proyecto. Es quien comunica su visión del producto al equipo y define los requisitos, lo que lo define como el encargado del *Product Backlog*.
- **Scrum master**
  El *Scrum Master* encabeza el equipo de desarrollo y los orienta en cuanto a las reglas y la metodología de trabajo. Asimismo, maneja los inconvenientes que surjan en el proyecto y actúa como nexo principal con el cliente, que en este caso es el *Product Owner.*
- **Team**
  También conocido como equipo de desarrollo, conformado por expertos liderados por el *Scrum Master*. Son los encargados de crear el producto final y su trabajo es fundamental para el éxito del proyecto.

En nuestro proyecto los tutores harán de *Product Owner*, disponiendo los requisitos necesarios. Mi persona tomará el rol de *Scrum Master*, quien será el encargado de agendar reuniones cada dos semanas y de presentar los avances y requisitos implementados. A su vez, mi persona también formará parte del *Team* como lead programmer*,* junto con el artista 3D y 2D.

2.4 Tecnologías

El enfoque de esta subsección es proporcionar una descripción concisa de las tecnologías utilizadas en este proyecto.

2.4.1 Unity

Se ha optado por utilizar *Unity* [3] como motor gráfico para el desarrollo del videojuego, gracias a su capacidad de exportación a diferentes plataformas y su facilidad para adaptarse a ordenadores de sobremesa, consolas y dispositivos móviles. Este motor cuenta con amplios recursos, como su plataforma de tutoriales *Unity Learn* [4] y su extensa documentación, tanto del motor [5], como de su *API* [6].

2.4.2 Lenguaje de programación C#

Unity hace uso del lenguaje orientado a objectos C# [7] para implementar sus *scripts.* Además, *Unity* implementa de manera nativa librerías en C# que permiten el control de los *GameObjects.*

2.4.3 Control de versiones

En el desarrollo de proyectos de software, es fundamental trabajar de manera colaborativa y proteger la información para evitar problemas y retrasos en las entregas previstas. Para ello, se ha optado por implementar un sistema de control de versiones en este proyecto mediante *Git* [8], una herramienta gratuita y de código abierto.

Este sistema ofrece dos interfaces: una por consola de comandos y otra mediante el uso de *GitHub Desktop* [9]. Hemos decidido usar esta última ya que ofrece una interfaz más amigable y una curva de aprendizaje menor respecto al sistema por consola.

Crear el repositorio es tan sencillo con indicar la ruta del proyecto de *Unity* e incluir el archivo *.gitignore*, que incluye la descripción de archivos (como metadatos) y directorios (que pueden ser librerías, logs) a ignorar y no subir a *Git*.

2.5 Recursos disponibles

A continuación, presentaremos los recursos físicos y artísticos disponibles, además de los recursos tecnológicos que tenemos a nuestra disposición. Esto es fundamental para poder trabajar con las herramientas dadas y lograr los objetivos del proyecto.

- Recursos Hardware:

    - Ordenador de sobremesa AMD Ryzen 3900X – 1080Ti con 64Gb de RAM DDR4.
- Recursos Software:
    - *JetBrain Rider* [10] con licencia educativa.
    - *Microsoft Teams* como *sharepoint*.

Los recursos artísticos serán producidos por el artista 3D y 2D.

3. Diseño

En este apartado describimos en detalle el diseño del videojuego, abarcando desde las mecánicas de juego hasta la interfaz de usuario, pasando por el diseño de niveles y otros aspectos clave que influirán en la jugabilidad.

3.1 GDD

Documento de diseño del juego que registra todos los aspectos del videojuego. Es un documento vivo, es decir, aunque definimos este documento al principio del proyecto, vamos adaptándolo según los cambios en los requisitos y las opiniones del cliente.

3.1.1 Descripción

Athena's Song es un juego de memoria y habilidad que consiste en repetir patrones de colores y sonidos mediante un piano representado por columnas. El juego presenta secuencias cada vez más complejas a medida que se avanza de nivel, que deben ser recordadas y repetidas con precisión.

3.1.2 Características técnicas

Athena's Song está orientado inicialmente a ordenadores de sobremesa con sistema operativo Windows. Hemos elegido usar el motor de videojuegos *Unity,* debido a su gran comunidad y versatilidad.

3.1.3 Experiencia de juego

A continuación, definiremos el público objetivo, indicando los antecedentes de este público antes de conocer nuestro producto, cómo pueden reaccionar a él y que sensaciones buscamos trasmitirles.

Nuestro público objetivo son jóvenes entre 10 y 20 años, con interés en la música y el deseo de mejorar sus habilidades musicales. Muchos de ellos pueden tener antecedentes en la formación musical, ya sea a través de clases de instrumentos o educación musical en la escuela. Otros pueden haber descubierto su pasión por la música a través de plataformas digitales o de sus artistas favoritos en las redes sociales.

Esperamos que nuestro producto, un videojuego serio para entrenar el oído musical, les resulte atractivo y divertido a la vez que útil para mejorar sus habilidades musicales. Buscamos transmitirles la sensación de que pueden mejorar y progresar en su formación musical de una manera emocionante y accesible, a través de una experiencia interactiva y

dinámica. Queremos que se sientan motivados y entusiasmados al jugar nuestro videojuego, mientras aprenden a reconocer notas, y entrenan su oído para detectar diferentes patrones y sonidos musicales.

3.1.4 Jugabilidad

En este apartado describiremos los elementos con los que interactúa el jugador y que reacción tienen estos.

1. Acciones y mecánicas

A continuación, hablamos de las acciones y mecánicas de las que dispone el jugador. Una acción es algo que hace el jugador dentro del juego (entorno virtual) y la mecánica es cómo el jugador llega a realizar dicha acción en el entorno real.

La acción principal trata de seleccionar una nota mediante la mecánica de *point and click,* conocida con apuntar a un objeto interactivo y pulsar sobre él, mediante la pulsación de una tecla, en nuestro caso, usamos el botón izquierdo del ratón como mecánica.

Tras haberse reproducido el patrón a recordar, el jugador puede interactuar con las notas. Si la nota es correcta, el jugador puede continuar interactuando hasta completar la secuencia. Si es incorrecto, se acaba el nivel.

2. Opciones de juego

En este apartado comentaremos las opciones gráficas y sonoras de las que dispone el usuario.

En primer lugar, para las opciones gráficas el juego cuenta con la opción de ayuda visual. Esta consiste en si las notas se iluminan al sonar o no. Esta opción supone una gran ayuda para el jugador, sobre inicialmente. Al desactivar esta opción, además se cambia la puntuación que recibe el jugador, que varía entre estrellas si está activa y liras si está desactivada.

También se cuenta con una opción de mostrar colores al iluminarse las notas. Esta opción cambia el color en el que se iluminan las notas. Si está activa, las notas se iluminan de distintos colores, lo que ayuda al jugador a recordar el patrón. Si está desactivada, todas las columnas se iluminan del mismo color.

A mayores se cuenta con opciones que son puramente visuales y no cambian la jugabilidad, para que el jugador elige la opción que prefiera. El jugador puede elegir entre usar una notación clásica (Do, Re, Mi, Sol, La, Si) o usar la notación anglosajona (C, D, E, F, G, A, B). También está la opción de elegir entre usar bemoles o sostenidos para las notas intermedias.

Para las opciones sonoras, el jugador dispone de una barra de sonido para controlar el nivel de audio. Además, existe una opción donde se incluye una lista con los distintos instrumentos que pueden sonar y que el jugador puede elegir.

El idioma es una opción que, aunque no está presente en el menú de opciones, sí está implementada. Es decir, el idioma de la aplicación depende del idioma del sistema, variando entre español para sistemas con idioma en español e inglés para cualquier otro sistema.

3. Tutorial

A mayores, se cuenta con un tutorial que explica todo lo necesario para entender cómo funciona el juego y las diferentes opciones. Este tutorial se activa la primera vez que se instala el juego y se muestra al entrar en cada pantalla que incluye un tutorial. Posteriormente, tras haber visto todos los tutoriales, en el menú principal se activa la opción para poder volver a activar y desactivar el tutorial.

A continuación, en las diferentes tablas se muestran los diálogos del tutorial en cada pantalla. Las referencias *<imagen=nombre>* se usan para indicar que se insertar una imagen en el texto. Esta referencia posteriormente es sustituida durante el desarrollo por *<sprite=número>*, donde número hace referencia a la posición en un documento que ocupa la imagen que queremos usar.

Orden	Diálogos menú opciones
1	Este es el panel de opciones.
2	En la velocidad de notas modificas la velocidad entre notas y la duración de las mismas.
3	Si se activa esta opción, las teclas se iluminan. Al desactivar esta opción se consiguen <imagen=liras> en lugar de <imagen=corona>.
4	El activar esta opción las teclas se iluminan de distintos colores. Si esta opción esta desactivada todas se iluminan del mismo color.
5	Elige entre la notación anglosajona y la notación clásica.
6	Elige entre <imagen=símbolo bemol *b*> bemoles o <imagen=símbolo sostenido #> sostenidos para las teclas negras.
7	Volumen que controla todo.
8	Elige el instrumento que suena.

Tabla 3.1: Tabla con diálogos del tutorial para el menú de opciones

Orden	Diálogos menú niveles
1	Este es el panel de niveles.
2	Cada nivel cuenta con unas notas que sonaran en un orden aleatorio.
3	Para conseguir <imagen=corona> debes acertar 5-6 notas. Para conseguir <imagen=corona> <imagen=corona> debes acertar 7-8 notas. Para conseguir <imagen=corona> <imagen=corona> <imagen=corona> debes acertar 9-10 notas.
4	Para conseguir <imagen=liras> debes desactivar la ayuda visual en opciones.
5	Para conseguir <imagen=liras> debes acertar 5-6 notas. Para conseguir <imagen=liras> <imagen=liras> debes acertar 7-8 notas. Para conseguir <imagen=liras> <imagen=liras> <imagen=liras>debes acertar 9-10 notas.
6	Pulsa <imagen=botonEmpezar> para empezar.

Tabla 3.2: Tabla con diálogos del tutorial para el menú de niveles

Orden	Diálogos HUD
1	Esta es la vista de juego.
2	En la barra superior se muestra el número de rondas. En total son 8 rondas. Se empieza con tres notas por ronda y se suma una al avanzar de ronda.
3	En la parte inferior se muestran las notas totales a acertar. También se muestra las notas acertadas hasta el momento.
4	Fíjate en las notas que suenan y acuérdate del orden.
5	¡Ahora es tu turno!

Tabla 3.3: Tabla con diálogos del tutorial para el HUD

Orden	Diálogos menú principal
1	El tutorial se ha activado de nuevo. Pulsa en mí para desactivarlo.
2	El tutorial se ha desactivado. Pulsa en mí para activarlo.

Tabla 3.4: Tabla con diálogos del tutorial para el menú principal

Todos los diálogos se reproducen por el orden indicado, salvo en el caso de la Tabla 3.4. Para estos no hay orden, si no que se van intercambiando según se usen. Es decir, primero se reproduce el orden 1. En la siguiente vez que se interactúe con él, se reproduce el orden 2 y así sucesivamente.

3.1.5 Flujo de juego

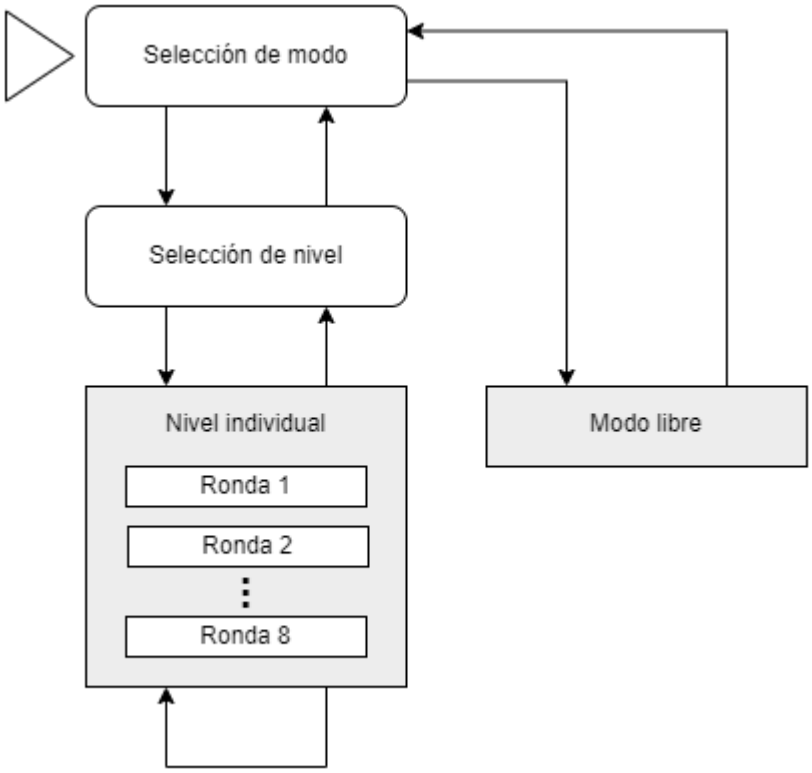

Figura 3.1: Diagrama flujo de juego

Como podemos ver en la Figura 3.1, el flujo de juego comienza en la selección de modo. Los recuadros sombreados indican unidades de juego. En nuestro juego podemos encontrar dos unidades de juego o modos distintos: el modo libre y el modo por niveles.

En el modo libre se presenta el escenario con las notas y el jugador pulsa las mismas libremente sin restricción.

En el modo por niveles, el jugador primero debe seleccionar el nivel que quiere realizar. Tras elegirlo, aparece el escenario y se empieza a reproducir la primera ronda que incluye el primer patrón. Este primer patrón cuenta con tres notas generadas aleatoriamente entre las notas disponibles en ese nivel. Si el jugador acierta todas las notas se pasa a la siguiente ronda, donde se añade una nueva nota al final de las ya existentes y se reproduce el patrón de nuevo. El total de rondas es ocho, lo que hace un total de diez notas a recordar.

Al acabar todas las rondas o fallar una nota, el jugador conoce la puntuación que ha obtenido. Esta puntuación pueden ser estrellas o liras, dependiendo de si la opción de ayuda visual está activada o desactivada. La puntuación funciona de la siguiente manera:

- Si se aciertan menos de cinco notas, se reciben cero estrellas/liras.

- Si se aciertan entre cinco y seis notas, se recibe una estrella/lira.
- Si se aciertan entre siete y ocho notas, se reciben dos estrellas/liras.
- Si se aciertan entre nueve y diez, se reciben tres estrellas/liras.

Tras esto el jugador tiene la opción de repetir el nivel para aumentar la puntuación o volver a la lista de niveles para empezar una nuevo.

Si el nivel es infinito, la puntuación se cuenta en rondas completadas, es decir, cada vez que se completa un patrón, se suma uno a la puntuación, diferenciando también entre estrellas y liras, dependiendo si la ayuda visual se usa.

1. Diálogos finales

Los diálogos finales se reproducen al acabar un nivel e informan al jugador de qué tal ha realizado el nivel. Estos diálogos dependen de la puntuación y se eligen de manera aleatoria entre tres distintos. En las tablas que se muestran a continuación se indican los diálogos para las distintas puntuaciones:

Diálogos puntuación igual a cero
Inténtalo mejor la próxima.
Tu memoria puede mejorar.
Sigue intentándolo.

Tabla 3.5: Tabla con diálogos finales para puntuación cero

Diálogos puntuación igual a uno
Te queda por mejorar.
Todo esfuerzo tiene recompensa.
No está mal para empezar.

Tabla 3.6: Tabla con diálogos finales para puntuación uno

Diálogos puntuación igual a dos
Sigue así.
Has mejorado.
A la próxima seguro que lo consigues.

Tabla 3.7: Tabla con diálogos finales para puntuación dos

Diálogos puntuación igual a tres
Enhorabuena, puntuación máxima.
Puntuación máxima, gran trabajo.
Felicidades, sigue así.

Tabla 3.8: Tabla con diálogos finales para puntuación tres

3.1.6 Diseño de niveles

En el juego se cuenta en total con once niveles. Cada nivel tiene asignado unas notas que pueden sonar en un orden aleatorio.

- **Nivel 1:** notas C, E, G (Do, Mi, Sol).
- **Nivel 2:** notas C, D, E (Do, Re, Mi).
- **Nivel 3:** notas C, F, A (Do, Fa, La).
- **Nivel 4:** notas G, A, B (Sol, La, Si).
- **Nivel 5:** notas C, D, E, G, A (Do, Re, Mi, Sol, La).
- **Nivel 6:** notas C, E, E, F, G, A (Do, Re, Mi, Fa, Sol, La).
- **Nivel 7:** notas C, D, E, F, G, A, B (Do, Re, Mi, Fa, Sol, La, Si).
- **Nivel 8:** notas C, C#, D, E (Do, $Do_{\#}$/Re_{b}, Re, Mi).
- **Nivel 9:** notas C, E, F, F#, G, G#, A (Do, Mi, Fa, $Fa_{\#}$/Sol_{b} , Sol, $Sol_{\#}$/La_{b}, La).
- **Nivel 10:** todas las notas.
- **Nivel Infinito:** todas las notas y no hay límite de notas.

3.1.7 Diseño de cámara

Inicialmente disponemos de una cámara con visión ortográfica mostrando todo el escenario como podemos observar en la Figura 3.2.

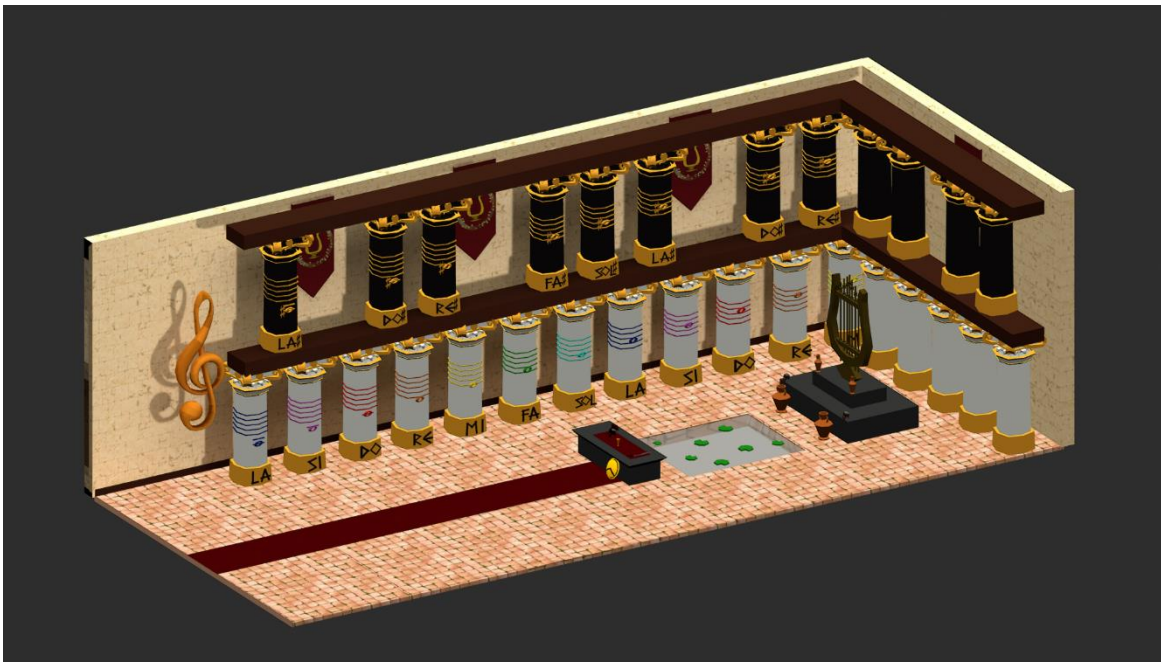

Figura 3.2: Visual cámara en posición inicial

De esta posición la cámara se mueve a la posición de juego, vista que se muestra en la Figura 3.3.

Figura 3.3: Visual cámara en posición de juego

3.1.8 Diseño de interfaz

En esta sección comentaremos y mostraremos los distintos elementos de interfaz con los que interactúa el usuario.

1. Flujo de pantalla

Inicialmente, en la Figura 3.4 mostramos el flujo que puede seguir el usuario a través de la interfaz, visualizando las distintas pantallas.

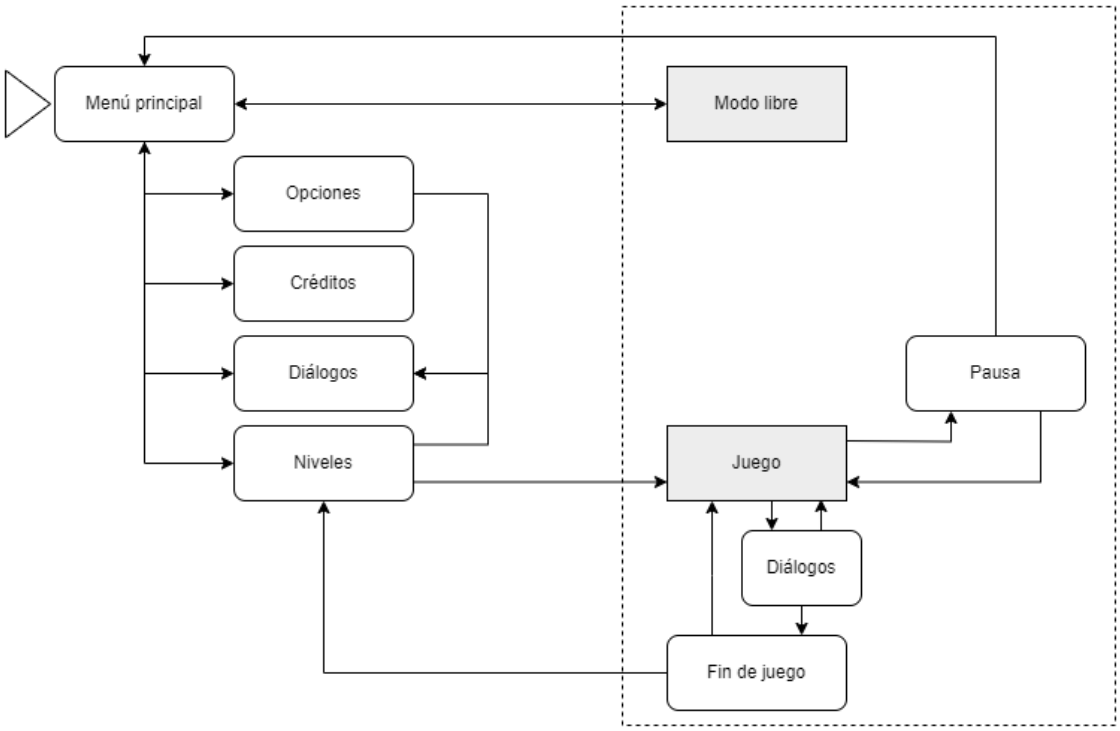

Figura 3.4: Visual cámara en posición de juego

En primer lugar, explicaremos los distintos elementos presentes en el flujo de pantalla.

- La flecha que apunta a menú principal indica el comienzo de todo el flujo.
- El recuadro con bordes en punta y fondo sombreado indican que se trata de un HUD.
- Los recuadros con bordes redondeados indican que se trata de interfaz que muestra información estática, es decir, la información que muestran no se modifica mientras el jugador la está visualizando.
- El cuadro con borde de punto discontinuos indica que estamos en el entorno de juego.

Al comenzar el juego, el jugador se encuentra en la interfaz de menú principal. Desde aquí puede acceder al menú de opciones, a los créditos y al de niveles. También puede

acceder directamente al juego mediante el modo libre y volver al menú principal directamente desde este modo.

El menú de diálogos que conecta con el menú principal es especial, ya que este más que un menú, es una interfaz superpuesta a las demás interfaces con las que se comunica y que solo se muestra si el tutorial está activado.

Desde el menú de niveles podemos empezar un nivel de juego. Al entrar en el nivel, se muestra el HUD. Desde aquí podemos pausar el juego, mostrando el menú de pausa, desde el cual podemos volver al menú principal.

Continuando con el nivel, ya sea si el jugador se equivoca o completa el nivel, se muestra el menú de diálogos, que hace uso de los diálogos finales y posteriormente el menú de fin de juego. Desde este último podemos repetir el nivel o volver al menú de niveles. Cabe mencionar que el menú de diálogos también es el usado por el tutorial, por lo cual de ahí la relación bidireccional entre juego y diálogos.

2. Wireframe

A continuación, describiremos los *wireframes* de cada interfaz y los elementos que los componen.

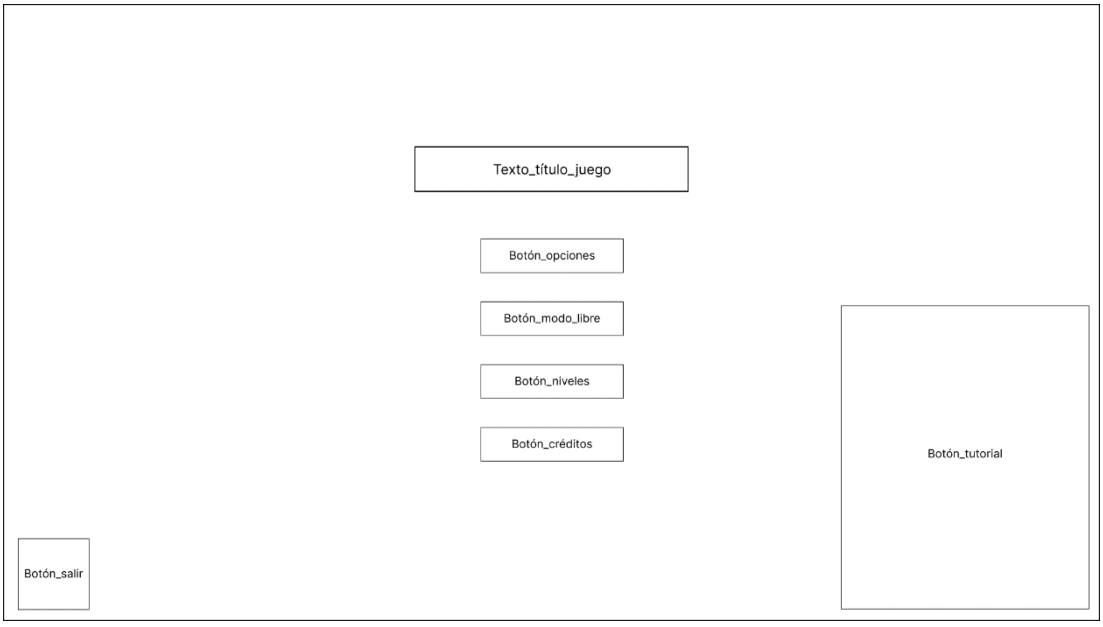

Figura 3.5: *Wireframe* - Menú principal

Como podemos observar en la Figura 3.5, el menú principal incluye toda la navegación a los distintos menús a través de los botones situados en la parte central.

El botón para el tutorial tiene una funcionalidad especial, ya que este no aparece hasta que el jugador no ha visualizado todo el tutorial en las diferentes interfaces. Este

botón además no es una navegación a otro menú, si no que activa y desactiva el tutorial al pulsarlo, mostrando un diálogo informativo. Por último, el botón salir cierra el juego.

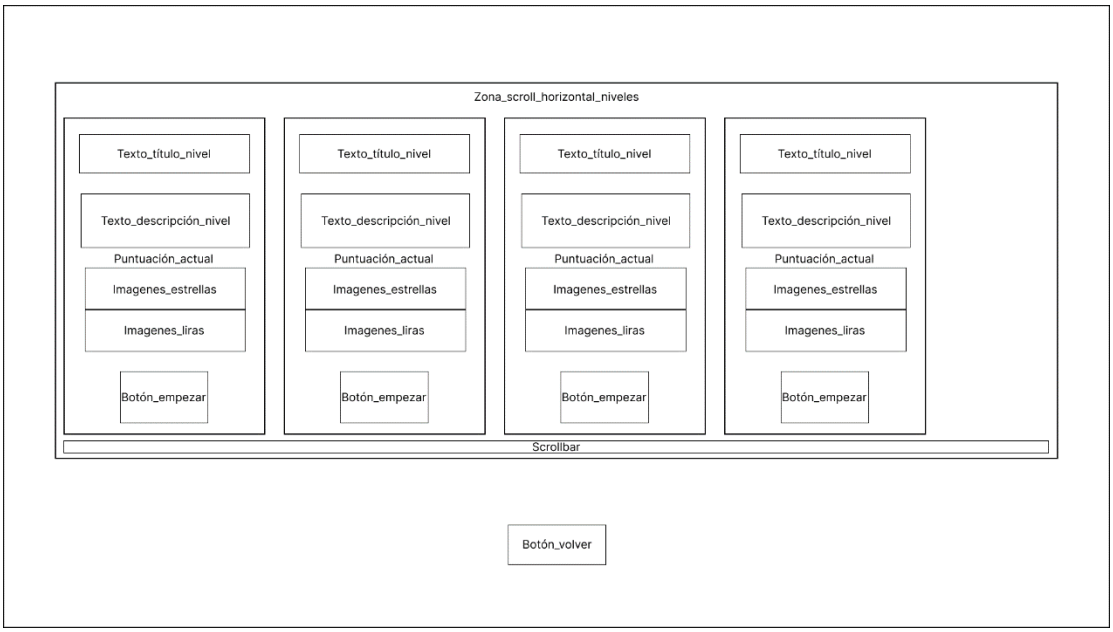

Figura 3.6: *Wireframe* - Menú niveles

Como se muestra en la Figura 3.6, para la selección de los niveles hemos incluido una zona de desplazamiento horizontal, controlada por la barra de desplazamiento (*scrollbar*) situada en la zona inferior, donde se incluyen todos los niveles.

Cada nivel cuenta con un título para diferenciarlos y una descripción, donde se indican que notas está disponibles en el nivel. También se muestra la zona de la puntuación, que indica la máxima puntuación obtenida en el nivel, diferenciando entre estrellas, si se usó ayuda visual y liras si esta ayuda no se usó.

Los botones empezar sirven de inicio y navegación para el HUD. El botón volver se conecta al menú principal.

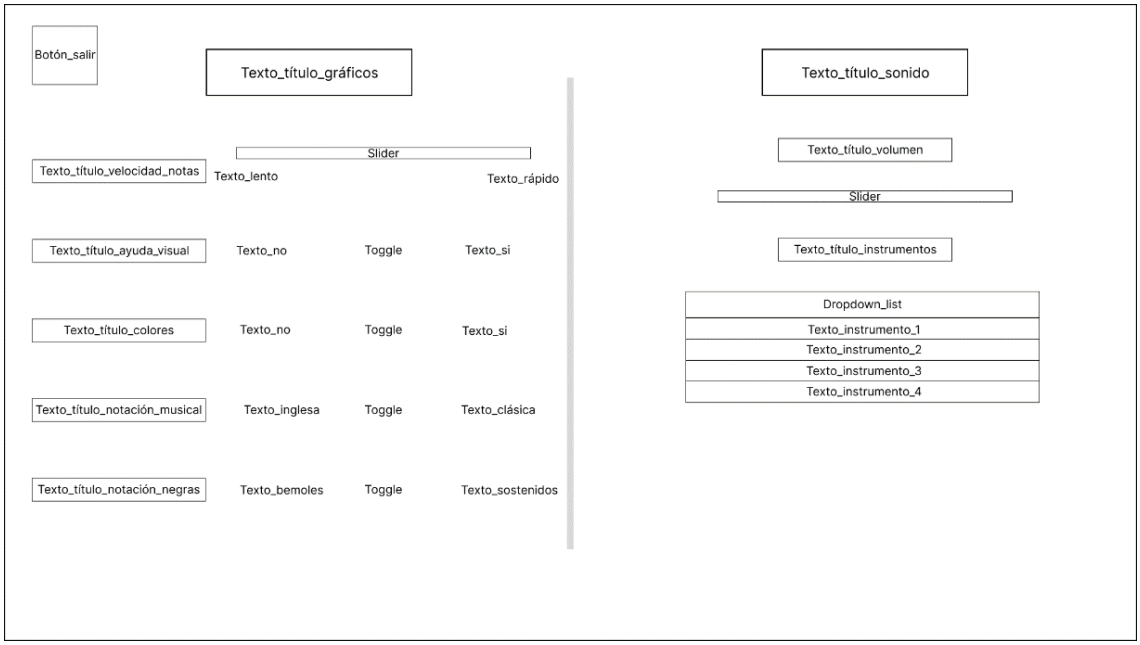

Figura 3.7: *Wireframe* - Menú opciones

En el menú de opciones, como se indica en la Figura 3.7, nos encontramos dos apartados disponibles, la zona de gráficos y la zona de sonido.

En la zona de gráficos están incluidas aquellas opciones que tienen algún impacto en el apartado visual. En primer lugar, podemos configurar la velocidad de las notas mediante una barra de desplazamiento o deslizador (*slider*), donde disponemos de 5 niveles de más lento a más rápido. A continuación, encontramos las opciones de ayuda visual y mostrar colores que hacen uso de un interruptor (*toggle*), eligiendo entre sí y no.

Las dos últimas opciones también hacen uso de un interruptor, pero para la primera opción, donde podemos configurar la notación que se muestra en las distintas columnas, las elecciones disponibles son la notación clásica y la notación anglosajona o inglesa. Para la última opción, que configura la notación que se muestra en las columnas que representan las teclas negras de un piano, podemos elegir entre bemoles o sostenidos.

Para la zona sonora, contamos con una barra de desplazamiento o deslizador que controla el volumen de todo el juego y una lista desplegable (*dropdown list*) para elegir el instrumento que suena.

Figura 3.8: *Wireframe* - Menú créditos

En el apartado de créditos, que observamos en la Figura 3.8, contamos con una zona para escribir el rol que ha tomado el participante y su nombre en la zona superior y en la zona inferior, en el apartado de agradecimiento, mostramos los nombres de las personas que han hecho posible este proyecto.

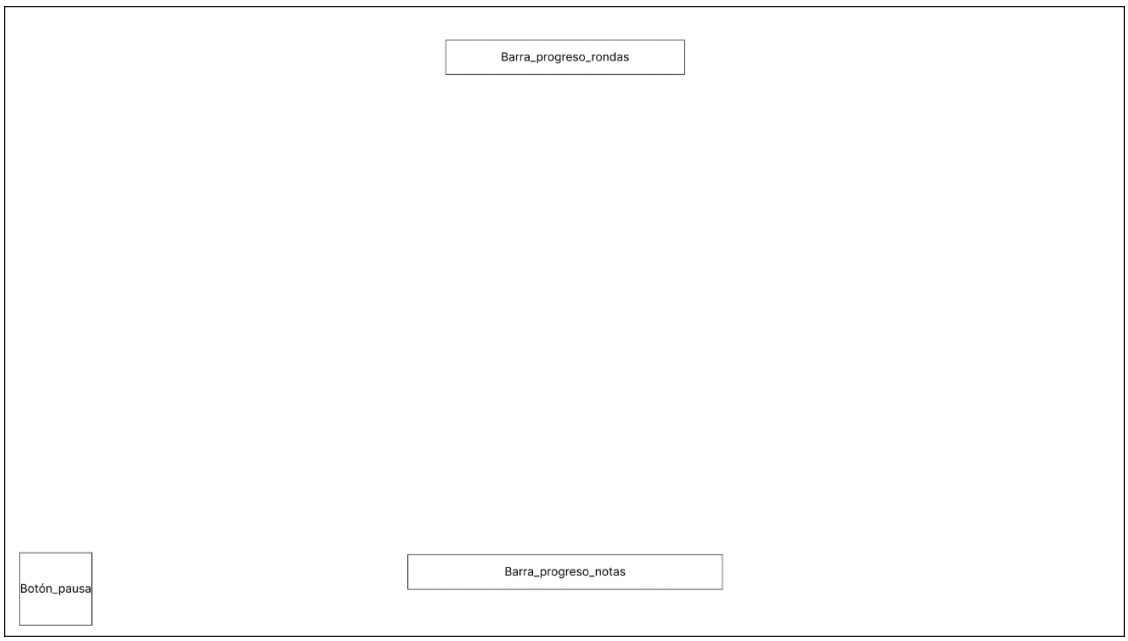

Figura 3.9: *Wireframe* - HUD

En la Figura 3.9 encontramos el HUD o interfaz de juego. En esta se informa al usuario de su progreso en el juego.

En nuestro caso, en la zona superior localizamos la barra de progreso de rondas, barra que se va llenando a medida que el jugador avanza de ronda, indicando las rondas restantes para completar el nivel. En el caso del nivel infinito, esta barra desaparece, y en su lugar se indica la ronda que ha completado el jugador mediante un número. Por ejemplo, si ha completado la ronda 7, se muestra un 7 mientras está jugando la ronda 8.

En la zona inferior ubicamos la barra de progreso de notas, que indica al jugador las notas que tiene que acertar en esa ronda, que se iluminan a medida que acierta notas.

El botón de pausa, situado en la zona inferior izquierda, pone en pausa todo el juego. Si estamos en el modo libre, ocultamos las rondas y notas, mostrando solo el botón de pausa que, en este caso, devuelve al jugador al menú principal directamente como indicamos en el flujo de pantalla.

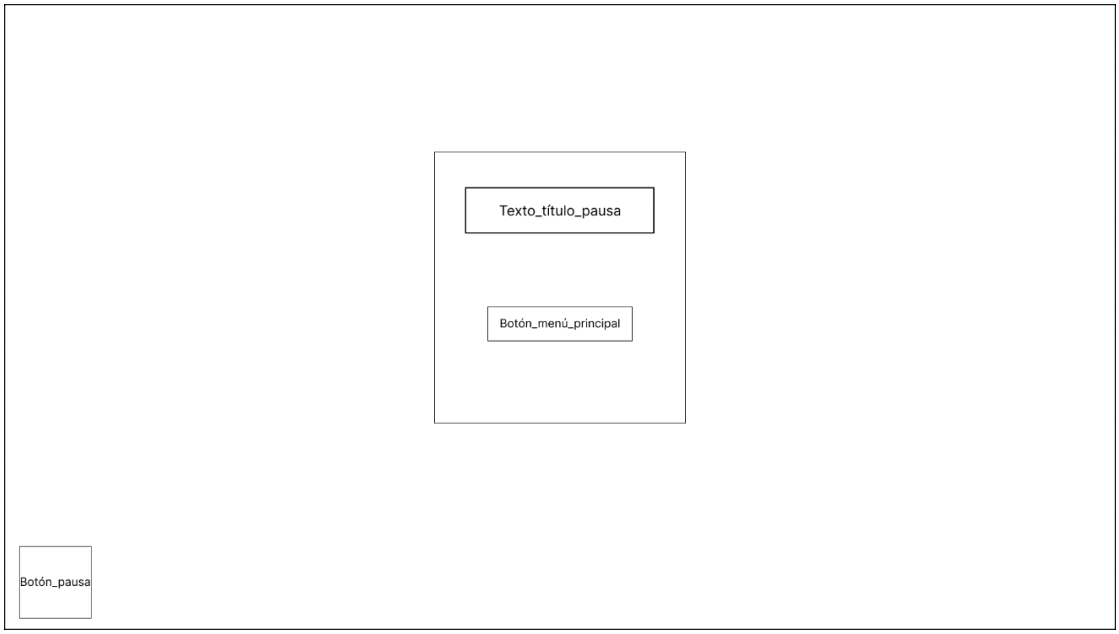

Figura 3.10: *Wireframe* - Menú pausa

En el menú de pausa, como observamos en la Figura 3.10, encontramos solamente dos botones.

El primero, en la esquina inferior izquierda, reanudar todo el juego desde donde el jugador lo pausó.

El segundo, en el centro, sirve de navegación hacia el menú principal, perdiendo todo el progreso que el jugador haya realizado en ese nivel.

Por último, el título en la parte superior informa al usuario del estado del juego.

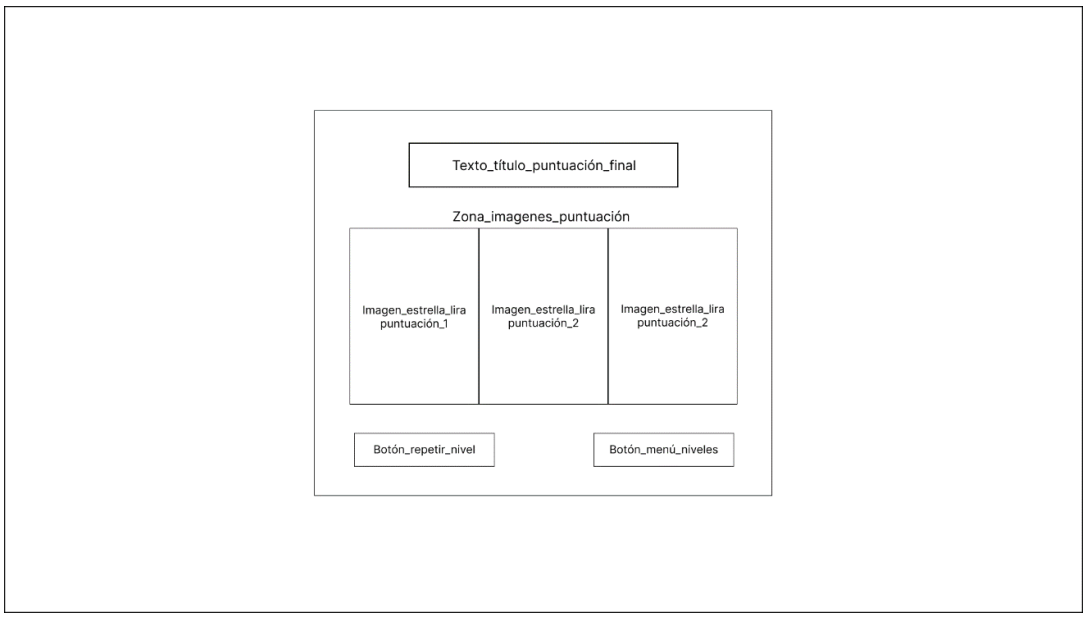

Figura 3.11: *Wireframe* - Pantalla fin de juego

En la pantalla final del juego, se muestra la puntuación obtenida al finalizar un nivel, como podemos ver en la Figura 3.11. Dependiendo si la opción de ayuda visual está activada o desactivada, en la puntuación se mostrarán estrellas o liras. Para indicar la puntuación de izquierda a derecha se muestran las imágenes llenas, es decir, para una puntuación de dos, las dos primeras imágenes serán figuras con relleno y la última es una figura formada solo con contorno y sin relleno.

Además, en la parte inferior, encontramos el botón para repetir el nivel y así aumentar la puntuación y el botón para volver al menú de niveles y empezar un nuevo nivel.

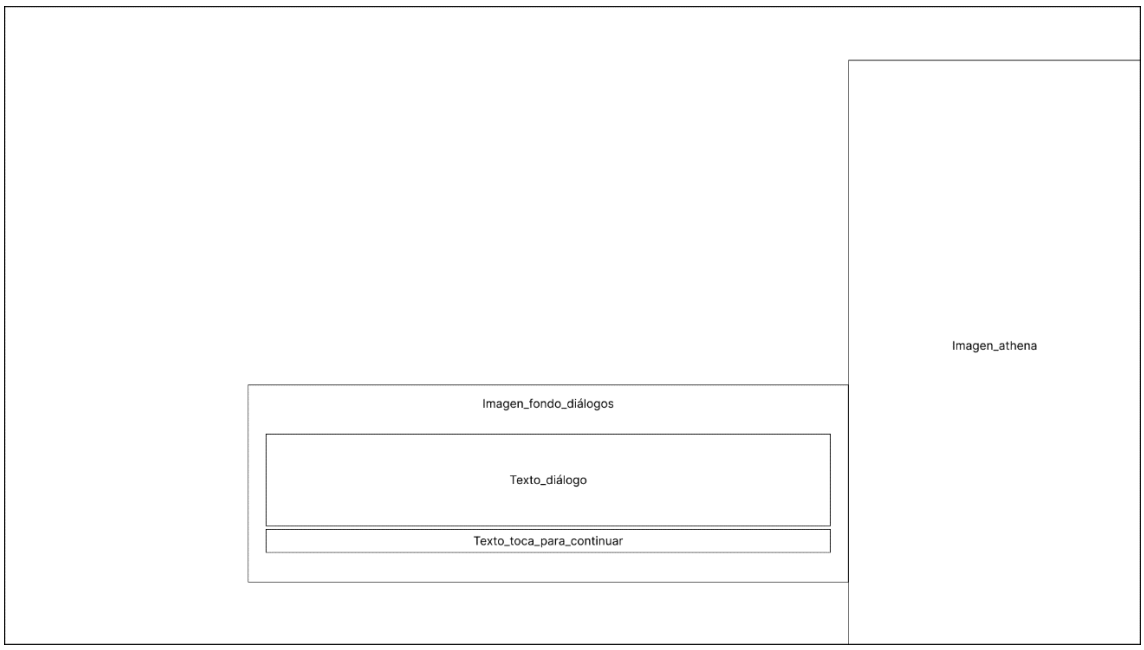

Figura 3.12: *Wireframe* - Diálogos

La interfaz de diálogos es usada en gran parte de la interfaz, tanto para mostrar el tutorial, como para mostrar mensajes tras finalizar un nivel.

Por ello, como podemos ver en la Figura 3.12, disponemos de una imagen de fondo sobre la que colocamos el texto que queremos mostrar. A la derecha de esta imagen está Atenea, personificación que realiza estos diálogos al jugador.

3.1.9 Ambientación

El juego recrea un entorno de la antigua Grecia con toques de mitología. Todo gira en torno al templo de Atenea, quien aparece en los tutoriales para informar de todo al jugador. En cuanto al templo, también encontramos objetos típicos como liras y vasijas.

3.1.10 Diseño de arte

En este apartado comentaremos las distintas decisiones de diseño artístico que se han tomado.

1. Diseño gráfico

En primer lugar, expondremos las decisiones sobre el entorno visual.

Para el entorno de juego se dispone de un templo formado por columnas que representan las notas de un piano, como observamos en la Figura 3.2 y Figura 3.3.

Encontramos una clave se sol en la izquierda, indicando la colocación de las notas. Para facilitar al jugador que notas son, sobre cada columna se encuentra un pentagrama indicando la nota, así como su nombre en la parte inferior.

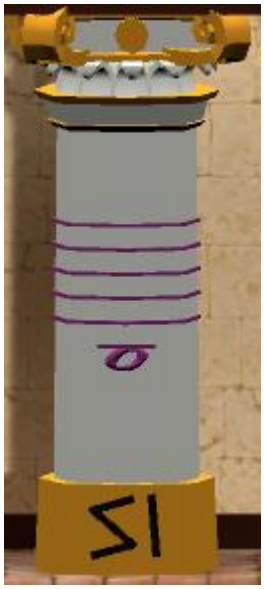

Figura 3.13: Columna blanca con color nota SI

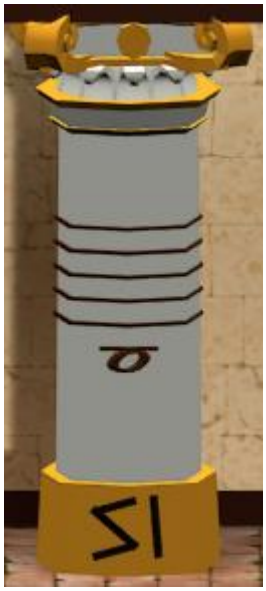

Figura 3.14: Columna blanca sin color nota SI

Como observamos en la Figura 3.13. también contamos con un color para diferenciar la nota de manera más sencilla. En concreto los colores son los siguientes:

- Nota La en color azul
- Nota Si en color morado
- Nota Do en color rojo
- Nota Re en color marrón
- Nota Mi en color amarillo
- Nota Fa en color verde
- Nota Sol en color azul verdoso

Estos colores se repiten para las mismas notas, independientemente de la escala en la que se encuentre la nota. Como apunte, el color solo afecta a las columnas blancas, dado que las columnas negras siempre se iluminan del mismo color, denominado color por defecto, el cual es un tono marrón, diferente al usado por la nota Re.

En la Figura 3.14, podemos ver la misma columna sin color, que usa el mismo color que las columnas negras para iluminarse.

El jugado puede configurar si la nota que se muestra es en notación clásica, como en la Figura 3.13 o si es en notación anglosajona como el la Figura 3.15.

Figura 3.15: columna blanca con color nota B

Las columnas negras tienen la opción de mostrar bemoles y sostenidos, ya que en un piano una tecla negra comparte ambas. En la Figura 3.16 observamos un La sostenido, columna que es compartida por el Si bemol, como se muestra en la Figura 3.17. Ambas figuras hacen uso de la notación clásica, aunque también es posible usar la notación anglosajona.

Figura 3.16: Columna negra LA sostenido

Figura 3.17: Columna negra SI bemol

2. Diseño de sonido

En segundo lugar, hablaremos sobre las decisiones que se han tomado en el apartado sonoro.

En cuanto a sonido que se usa al hacer sonar un patrón de notas, se ha decidido implementar la opción de elegir el instrumento que suena. Por ello, las diferentes opciones son:

- Piano
- Niña cantado notas
- Clave
- Vibráfono
- Órgano
- Harmónica
- Guitarra overdrive
- Violín
- Coro
- Oboe
- Piccolo

Otra de las decisiones de diseño de sonido es implementar un *feedback* al pulsar sobre cualquier botón de la interfaz. Para ello disponemos de un audio para esta funcionalidad.

También necesitamos un sonido para indicar al jugador que la nota que ha introducido ha sido incorrecta (si es correcta, suena la nota que pulsó).

Por último, si el jugador ha completado el nivel al completo, usamos un sonido para indicar que ha terminado el nivel satisfactoriamente.

3.2 Estructura de datos

En este apartado explicaremos el diseño que vamos a seguir para estructurar los datos, diferenciando entre los datos de guardado y los datos que usamos para el tutorial.

3.2.1 Datos de guardado

Como hemos indicado en el documento de diseño, necesitamos guardar la puntuación que consigue el jugador en cada nivel, en concreto, las estrellas y liras. Para ello, cada nivel dispone de una dupla que registra los valores conseguidos, oscilando entre 0 y 3 para los niveles del 1 al 10. Para el nivel 11, el nivel infinito, no hay límite de valor. Una vez todos los niveles disponen de su información, agrupamos las duplas en el orden de los niveles para posteriormente guardarlas y consultarlas.

3.2.2 Scriptable Objects

En este apartado describiremos las estructuras que hemos usado para guardar texto que posteriormente consultaremos para mostrarlo. Para ello hemos decidido usar los *scriptable objects* [11]. Estos son contenedores de datos que podemos configurar de múltiples formas. Para nuestro proyecto los vamos a configurar para guardar texto y el orden de estos. En concreto, usamos estos para los tutoriales, los diálogos finales y para el idioma.

Empezaremos explicando su uso para el apartado de tutoriales. Como indicamos en el apartado de diseño, el tutorial se compone de una sucesión de textos con un orden. Por ello, en el s*criptable object* configuramos una lista que guarda todos estos textos. Como además queremos guardar el orden y una lista generalmente solo puede definirse sobre un tipo único, necesitamos una estructura que incluya el orden y el texto. Para el orden nos basta con guardar un número y para el texto guardamos una sucesión de caracteres. Como funcionalidad adicional, incluimos en la estructura un valor que nos indica si tenemos que cambiar la máscara que usamos o no. Esta máscara nos ayuda en la explicación del tutorial, resaltando la información a la que hace referencia el tutorial.

Para los diálogos finales como describimos en el apartado de diseño, estos no tienen orden, por lo tanto, podemos hacer uso de una estructura que guarda únicamente texto mediante una lista.

Respecto al idioma, necesitamos una estructura que guarde las traducciones en orden para posteriormente asignárselas a los elementos correspondientes. Para ello, usamos una lista en el s*criptable object* que guarde texto.

3.3 Planificación

En la metodología SCRUM definimos *sprints* para establecer los plazos que se seguirán en cada uno. La planificación es crucial para organizar las reuniones, evitar retrasos en la implementación y garantizar que todo el proceso funcione de manera ordenada. Por ello, hemos definido la planificación en un diagrama de Gantt, que observamos en la Figura 3.18.

	SPRINT 1	SPRINT 2	SPRINT 3	SPRINT 4	SPRINT 5	SPRINT 6	SPRINT 7	SPRINT 8
High concept Anteproyecto								
GDD								
Mecánica principal								
Control de cámara								
Niveles								
Sistema de sonido								
Interfaz								
Sistema de guardado								
Sistema de tutorial								
Sistema de idioma								
Testing								

Figura 3.18: Planificación - Diagrama de Gantt

4. Implementación

En este apartado, nos sumergiremos en un análisis detallado de las implementaciones realizadas, las cuales se fundamentan en la etapa de diseño. Aquí exploraremos en detalle las soluciones que hemos desarrollado y los desafíos que hemos enfrentado en el proceso.

4.1 Importación de FBX

El primer paso para trabajar en un entorno 3D es tener un escenario. Para ello, el artista 3D nos proporciona a través del *sharepoint* los archivos .FBX que usaremos.

Para usar estos archivos en *Unity* es tan fácil como guardarlos en la carpeta *Assets* del proyecto. Estos archivos no necesitan una configuración adicional, para usarlos solo lo añadimos a la escena en la posición que queramos.

4.2 Estructura *scripts*

Antes de pasar a comentar la implementación de los *scripts*, explicaremos la estructura que siguen todos los *scripts*.

```
namespace Proyecto.StructureExample
{
    public class StructureExample
    {
        #region Inspector Variables

        #endregion

        #region Public Variables

        #endregion

        #region Private Variables

        #endregion

        #region Unity Methods

        #endregion

        #region Public Methods

        #endregion

        #region Private Methods

        #endregion
    }
}
```

Figura 4.1: *Script* - Estructura

Lo primero que introducimos en un *script* es a que *namespace* pertenece, como podemos observar en la Figura 4.1. Esto nos permite estructurar todos los scripts para que no haya posibles conflictos con los nombres.

Lo siguiente es introducir diferentes regiones para tener ordenado el código. La primera región son las variables que se muestran por inspector. Estos son variables privadas en las que usamos la etiqueta *[SerializeField]* para mostrarla por inspector. Además, estas variables al ser privadas empiezan por barra baja seguido por una letra minúscula.

La siguiente región es para las variables públicas, donde generalmente usamos *properties* [12] para hacer *get* y *set* de las variables privadas. Estas variables públicas siempre empiezan por mayúscula.

Las variables privadas son la siguiente región. Estas siguen la misma regla para los nombres que las variables para el inspector, empezando por barra baja seguido por una letra minúscula.

En la región para los métodos de *Unity* incluimos los métodos que heredan de la clase *MonoBehaviour* [13]. Esta clase incluye métodos como:

- Awake() : Se ejecuta cuando se carga el *script* en escena, aunque el *script* o *gameObject* esté deshabilitado.
- Start() : Se ejecuta una vez antes del primer *frame* y solo si está activo el *script*.
- Update() : Se ejecuta una vez por *frame*.

Por último, incluimos las regiones para los métodos públicos y privados. Para diferenciarlos, los públicos empiezan por letra mayúscula y los privados por letra minúscula.

4.3 *Singleton*

A continuación, explicaremos el script más útil del que hacemos uso. Generalmente para acceder a otro *script* se hace uso del método *getComponent<T>,* pero este método es poco eficiente.

La estructura básica del **singleton** se basa en el uso de una variable estática que guarda el *script* sobre el que se ejecuta. Como vemos en la Figura 4.2, tenemos una variable pública llamada *Instance* que se rellena con el *script* del tipo indicado mediante la cabecera *<T>* cuando hacemos la primera llamada a ese *script.*

Para usar el **singleton** usamos la herencia como se muestra en la Figura 4.3.

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
    protected static T _instance;

    194 usages  Francisco José Aragonés de la Rosa
    public static T Instance
    {
        Frequently called
        get
        {
            if(_instance == null) _instance = (T) FindObjectOfType(typeof(T));
            return _instance;
        }
    }
}
```

Figura 4.2: *Script - Singleton*

```
public class StructureExample : Singleton<StructureExample>
```

Figura 4.3: *Script* - Herencia *singleton*

4.4 Mecánica principal

Tras importar los archivos .FBX podemos ponernos a implementar la mecánica principal, el *point and click.*

Para ello vamos a usar un *raycast* [14] lanzado desde la cámara. Esto es un rayo invisible para el jugador que impacta con los objetos que tengan un componente *collider*. Esta funcionalidad se controla desde la clase **PointAndClickManager**.

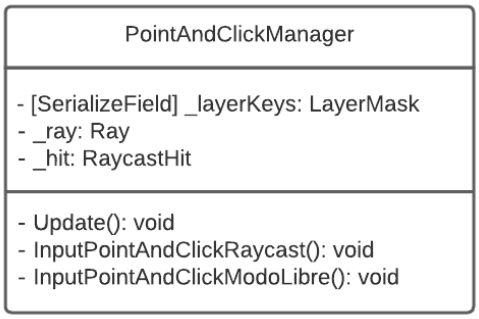

Figura 4.4: Clase *PointAndClickManager*

Como vemos en la Figura 4.4, para usar un *raycast* necesitamos un *Ray,* que es el rayo en sí, teniendo origen y dirección, un *RaycasyHit,* que guarda la información sobre que

objecto se ha colisionado y un *LayerMask*, que usamos como filtro para solo colisionar con los objectos que queremos.

Mediante el método *Update(),* llamamos constantemente a los métodos *InputPointAndClickRaycast()* y *InputPointAndClickModoLibre(),* controlando si el jugador está en modo libre, para llamar al último o al primero.

Antes de que pasemos a comentar ambos métodos, primero explicaremos como se configuran las columnas para controlar las colisiones del *raycast.*

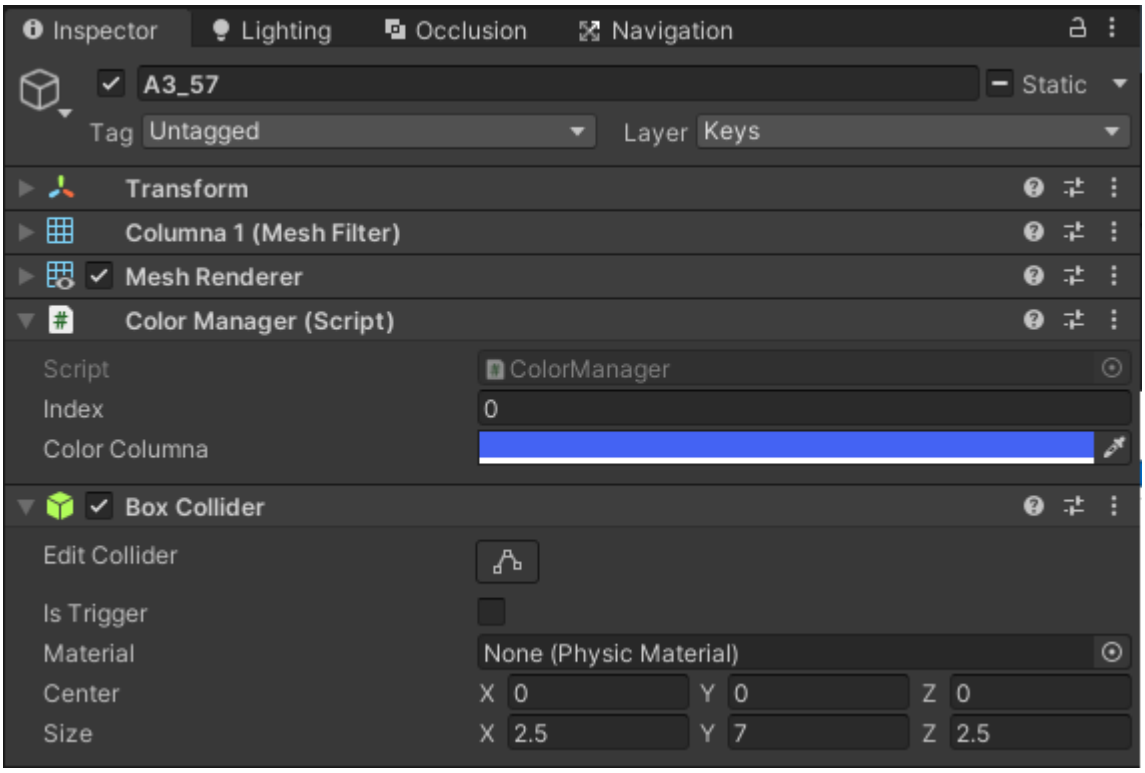

Figura 4.5: Configuración columna

Lo primero es incluir un componente *Box Collider* para poder colisionar con la columna. En la Figura 4.5, también observamos que se ha configurado la *Layer Keys* para filtrar las columnas. El nombre de la columna, en este caso A3_57, es importante, pues lo usaremos posteriormente para comprobar que columna se ha pulsado. La clase ***ColorManager*** la incluimos en cada columna para indicar mediante el *index* la posición de la columna, que va desde 0 a 20, y el color en el que se ilumina esa columna, mediante el parámetro *Color Columna.*

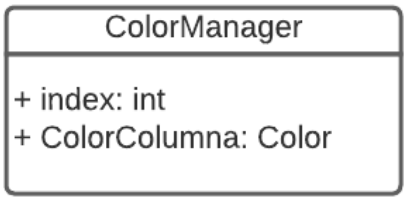

Figura 4.6: Clase *ColorManager*

Configurada cada columna, podemos pasar a explicar el método *InputPointAndClickRaycast().* En este método hacemos uso de la clase *Physics* incluida en *UnityEngine*[] para lanzar un *raycast*. La llamada en concreto es *Physics.Raycast(Ray, out RaycastHit, LayerMask*). Esta llamada devuelve true si colisiona con algún objecto con *collider* de la capa *LayerMask*. Para configurar *Ray*, usamos el método de la clase *Camera*, también de *UnityEngine, ScreenPointToRay(Vector3)*, que devuelve un *Ray*, con origen en la cámara y dirección al punto *Vector3.* El punto lo obtenemos mediante la posición del puntero del ratón en la pantalla, usando *Input.mousePosition*.

Con esto tenemos un rayo que está constantemente impactando sobre objectos del mundo. Ahora vamos a cambiar esto para que sea el jugador el que controla cuando se impacta con los objectos usando el click izquierdo del ratón. Para ello usamos *Input.GetMouseButtonDown(0),* que devuelve true una vez por pulsación.

Hasta este punto, los métodos *InputPointAndClickModoLibre()* y InputPointAndClickRaycast*()* son idénticos. A continuación, comentaremos las diferencias entre ellos.

En el caso del primero, la columna sobre la que impacta el rayo es iluminada, haciendo sonar la nota correspondiente, mediante del método *ShowHit()* de la clase **VisualKeyManager**. Para el segundo, tras pulsar el jugador se comprueba que la columna impactada es la correcta, para iluminarla y hacer sonar la nota, llamando al método *CheckPattern()* de la clase **GameplayManager**.

4.5 Control de cámara

Tras tener la jugabilidad configurada, pasamos a mover la cámara entre las distintas posiciones comentadas en el apartado de diseño. Esta funcionalidad la realiza la clase **CameraController,** mostrada en la Figura 4.7.

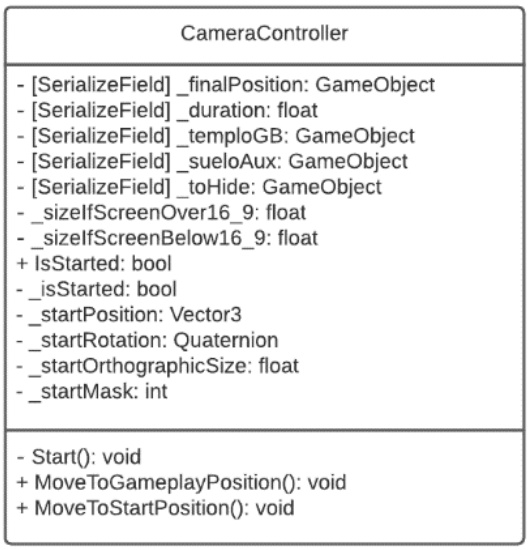

Figura 4.7: Clase *CameraController*

Inicialmente la cámara está en la posición que visualiza todo el entorno desde el lateral. En *_finalPosition* indicamos la posición que visualiza el entorno desde una vista frontal.

Mediante el método *MoveToGameplayPosition()* movemos la cámara desde la posición inicial a la posición final mediante una secuencia de la librería *Dotween* [15].

En *MoveToStartPosition()* movemos la cámara a la posición inicial instantáneamente para posteriormente poder usar *MoveToGameplayPosition()*.

4.6 Niveles

Una vez el jugador puede ver e interactuar con el entorno, pasamos a implementar los niveles y su funcionalidad.

4.6.1 Construcción de secuencia

Para la secuencia hemos usado herencia mediante una clase *abstract* [16] como observamos en la Figura 4.8, donde **Level** es la clase abstracta y las clases *Level_1*, *Level_2* hasta *Level_10* heredan de esta. El método *StartLevel()* es un método público que heredan todos los hijos, donde llenamos la lista *Sucesion* con las tres primeras notas mediante el método *AddNuevaNota()*.

Este último método es un método público abstracto que implementa cada hijo. En este se indican las notas que pueden sonar en cada nivel y cada vez que es llamado, devuelve una nota aleatoria entre las posibles.

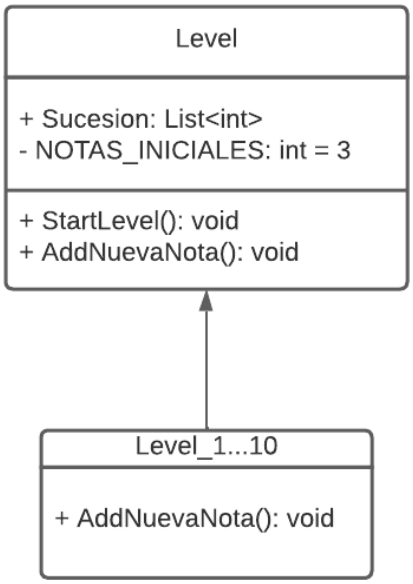

Figura 4.8: Diagrama UML niveles

Para crear un nivel usamos la clase **LevelsManager,** mostrada en la Figura 4.9. En concreto, usamos *StartLevel()* indicando qué nivel queremos empezar. Cuando llamamos a este método, se guarda el número del nivel por si el jugador posteriormente quiere volver a intentar el nivel, donde llamamos a *RestartLevel().* Esta clase hace un *StartCoroutine()* del método *restartLevelCoroutine()*. Como nota el tipo *IEnumerator* se usa para indicar que se trata de una corutina [17], métodos que se llaman usando StartCoroutine(corutina). Ambos métodos (*StartLevel() y restartLevelCoroutine()*) hacen uso de *StarNewLevel()* de la clase **GameplayManager**.

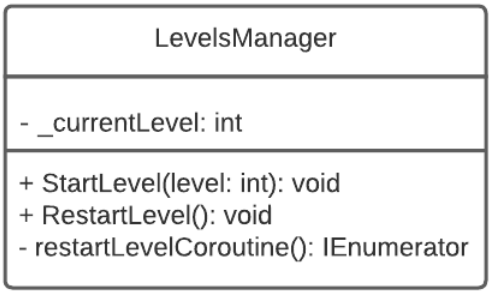

Figura 4.9: Clase *LevelsManager*

A continuación, explicaremos la clase que se encarga de toda la funcionalidad de jugabilidad y de guardar los estados de juego. En concreto, se trata de la clase **GameplayManager**, mostrada en la Figura 4.10.

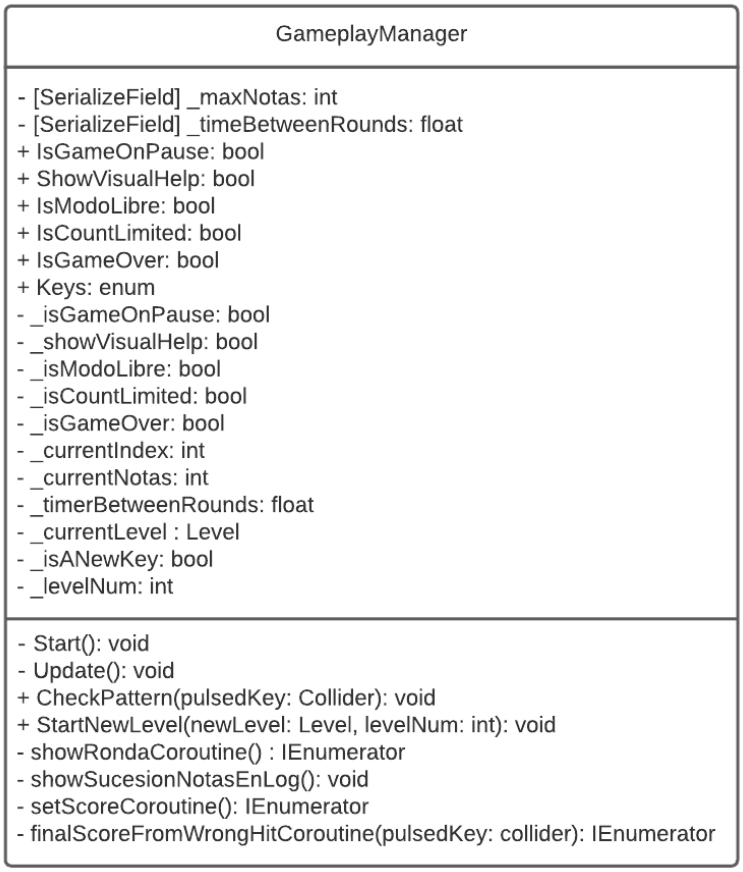

Figura 4.10: Clase *GameplayManager*

Primero comentaremos las variables que controlan los distintos estados de juego que podemos tener:

- IsGameOnPause: controla la pausa de juego.
- ShowVisualHelp: controla cuando se usa ayuda visual.
- IsModoLibre: controla cuando el jugador está en el modo libre.
- IsCountLimited: controla si hay límite de notas. Usado para el nivel infinito.
- IsGameOver: controla cuando el jugador pierde un nivel.

Como inciso, las variables públicas usan *properties* sobre las variables privadas para controlar los *get* y *set*, como podemos ver en la Figura 4.11. Esta implementación es interesante ya que podemos controlar las variables públicas y como se acceden a ellas.

```
public bool IsGameOnPause
{
    Frequently called
    get =>_isGameOnPause;
    set => _isGameOnPause = value;
}
```

Figura 4.11: Ejemplo *properties* sobre *_isGameOnPause*

Seguidamente, pasamos a exponer los métodos de **GameplayManager**. Como comentario general para todas las clases, en los métodos *Start()* y *Awake()* se inicializan las variables y referencias necesarias en esa clase. La utilización de uno u otro método obedece al orden de ejecución, que explicaremos posteriormente de manera más detallada.

En *Update()* comprobamos si estamos en juego (usando *_isGameOnPause*). Una vez comprobado esto, comprobamos la variable *_isANewKey*. Si está a *true* significa que una secuencia está disponible para ser reproducida. Para ello usamos la clase **VisualKeyManager***,* clase que explicaremos en el apartado 4.6.2.

En *CheckPattern()* se compara la nota pasada por parámetro con la nota siguiente en la secuencia. Si es incorrecta se acaba el nivel llamando a *finalScoreFromWrongHitCoroutine()* que usa *setScoreCoroutine()* para guardar la puntuación y mostrar el resultado. Si es correcta, pueden ocurrir varios sucesos:

- Quedan notas en la sucesión: se pasa a la siguiente nota de la sucesión esperando la siguiente llamada.
- No quedan notas en la sucesión y hay más rondas disponibles: se llama a *AddNuevaNota()* del nivel actual y se pasa a la siguiente ronda. Cabe mencionar que en *AddNuevaNota()* se usa el enumerador *Keys* para elegir las notas. Este enumerado contiene el nombre de las columnas en el orden en el que están dispuestas en el entorno.
- No quedan notas en la sucesión y no hay más rondas disponibles: el jugador ha completado el nivel, se muestra y guarda el resultado.

En *StartNewLevel()* se configuran todas las variables y se llama al método *StartLevel()* del **Level** pasado por parámetro.

4.6.2 Visualización de secuencia

Tras tener construida la secuencia, podemos pasar a mostrarla en el entorno. Para ello hacemos uso de la clase **VisualKeyManager**, mostrada en la Figura 4.12.

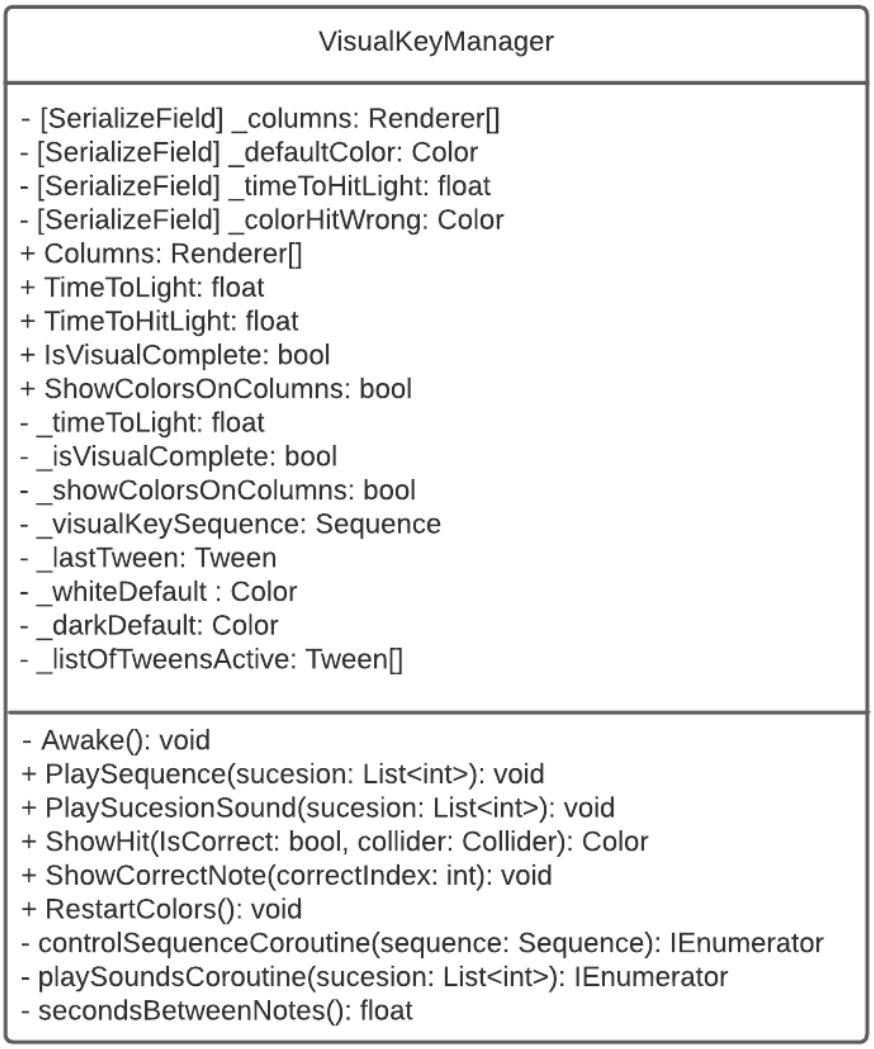

Figura 4.12: Clase *VisualKeyManager*

Comenzaremos mencionando las variables que hacen uso de *SerializeField*. Estas se muestran por inspector como observamos en la Figura 4.13, donde hemos configurado todas las columnas, así como el color por defecto y el color que se usa para mostrar el error.

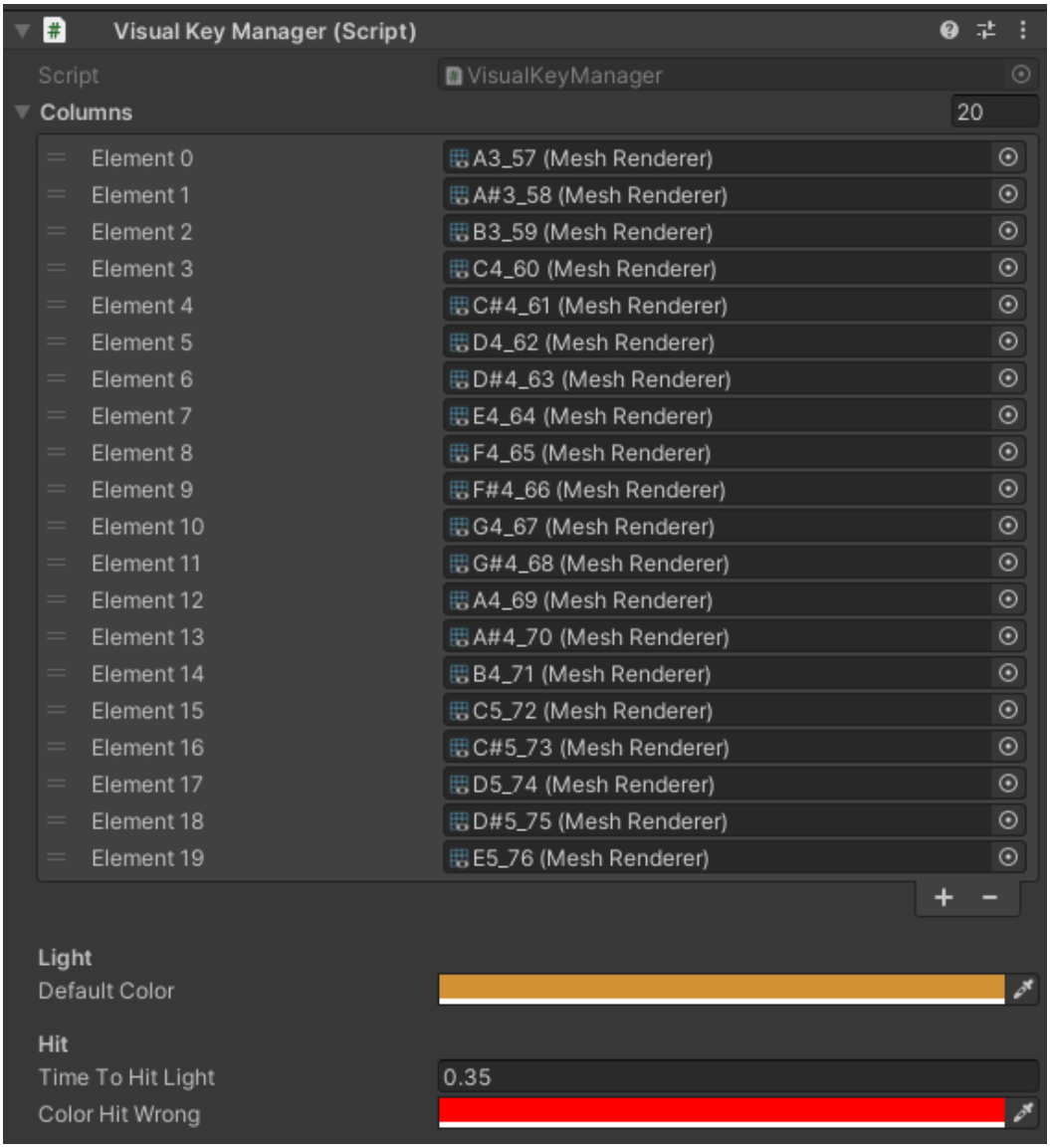

Figura 4.13: Vista inspector clase *VisualKeyManager*

En *PlaySequence()* mediante el uso de la librería *Dotween*, construimos una secuencia para mostrar las notas en orden. Esta secuencia incluye tanto el color como el sonido.

En el caso de *PlaySucesionSound(),* aquí se llama a *playSoundsCoroutine()*, que mediante una corutina muestra la sucesión en orden, pero solo el sonido, iluminando solo la primera columna de la sucesión para tener una referencia sonora.

En ShowHit() se ilumina la columna pasada por parámetro mediante el *collider*. El color depende de si es un acierto o un fallo, además de la opción de mostrar color, que determinad si se usa el color por defecto(*_defaultColor*) o el color de la columna (**ColorManager**.*ColorColumna*). Se devuelve el color del que se ilumina para actualizar la UI, que veremos cómo se implementa posteriormente.

El método *ShowCorrectNote()* se llama cuando el jugador se equivoca y muestra la columna correcta que el jugador debería haber pulsado.

Los dos últimos métodos controlSequenceCoroutine() y secondsBetweenNotes() se usan el primero para tener un control sobre la secuencia que se está reproduciendo y pararla en caso de ser necesario, y el último para calcular los segundos entre las notas.

4.7 Sonido

Con lo comentado en el episodio anterior ya tenemos el apartado visual, pero para controlar el apartado sonoro usamos la clase **AudioManager**, mostrada en la Figura 4.14.

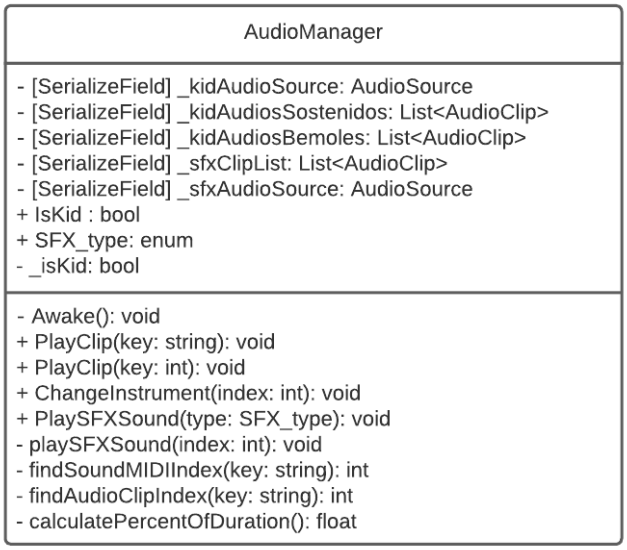

Figura 4.14: Clase *AudioManager*

Lo primero para poder usar audio es tener un componente *AudioListener* en la cámara. Seguidamente usamos el componente *AudioSource* para hacer sonar el audio. Este componente necesita archivos de audio que hace sonar mediante el método *PlayOneShot().* Estos archivos los introducimos por inspector diferenciando entre los archivos para las notas (*_kidAudiosSostenidos* y *_kidAudioBemoles*) y los archivos para los efectos especiales (*_sfxClipList*).

Así ya tenemos audio, pero solo de la niña cantando las notas, ahora nos faltan los demás instrumentos. Para ello vamos a hacer uso de un *asset* disponible en la *asset store* llamado Maestro [18]. Este *asset* usa archivos MIDI [19] para hacer sonar los instrumentos. Antes de comentar como se ha implementado Maestro, comentaremos los métodos que se incluyen en **AudioManager**.

En PlayClip() hacemos sonar una nota pasado el nombre de la columna en el primer caso o pasado el índice de la columna en el segundo caso.

En ChangeInstrument() cambiamos el instrumento que estamos usando pasado un índice. Este índice lo obtenemos por UI, apartado que veremos posteriormente.

En PlaySFXSound() hacemos sonar un efecto especial pasándolo por parámetro mediante el enumerador, que puede tomar los valores:

- Fail: sonido usado al fallar
- Click: sonido usado al pulsar sobre un elemento de la interfaz.
- Victory: sonido usado al completar todas las rondas de un nivel.

Los métodos privados son clases auxiliares usadas por los métodos anteriores. Los dos métodos que empiezan por *find* devuelven el índice correspondiente al nombra de la columna, devolviendo el primero un número entre 57 y 76, y el segundo un número entre 0 y 19.

Para usar Maestro necesitamos una clase comunicadora con él, la clase **MidiPlayer**, implementada por David Báez, colaborador del proyecto.

4.8 Sistema de guardado

Una vez tenemos toda la jugabilidad, podemos pasar a guardar la información sobre el avance en los niveles y las opciones.

En ambos casos hacemos uso de la clase **SaveManager**, mostrada en la Figura 4.15.

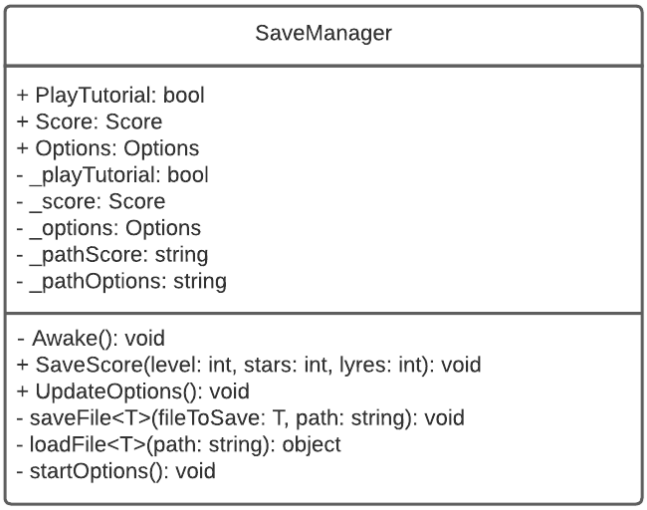

Figura 4.15: Clase *SaveManager*

En esta clase es importante mencionar que en *Awake()* se cargan los datos desde los archivos en caso de existir estos, o que se crean en caso de no existir.

Los campos más interesantes son las variables que guardan la puntuación (*_score*) y las opciones (*_options*). Antes de pasar a explicar estas clases, cabe mencionar como se

configuran las rutas *_pathScore* y *_pathOptions*. Para estás hacemos uso de *Application.persistentDataPath* [20], directorio persistente que depende de cada dispositivo, más el nombre del archivo, *score.json* para la puntuación y *options.json* para las opciones.

Para guardar los archivos hemos elegido el formato *JSON.* Para guardar hacemos uso de *saveFile()* y para cargar el archivo hacemos uso de *loadFile().* Ambos métodos son polimórficos, esto quiere decir que se adaptan al tipo de clase que los llaman, mediante el uso de *<T>*.

En *SaveScore()* guardamos la puntuación obtenida al acabar un nivel, solo si esta puntuación es mejor a la anterior.

En *UpdateOptions()* guardamos las opciones. Esta clase es llamada al salir del menú de opciones.

4.8.1 Puntuación

A continuación, explicaremos la clase **Score**, mostrada en la Figura 4.16. Seguimos la estructura de datos que describimos en el apartado de diseño, donde cada nivel tiene una dupla donde se guardan las estrellas y liras conseguidas, en concreto, **StarsAndLyres** mostrada en la Figura 4.17**.** Posteriormente la dupla se guarda en una lista de tamaño igual a los niveles disponibles cuyo orden hace referencia al orden de los niveles.

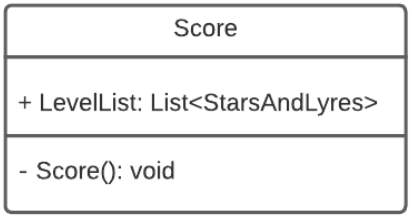

Figura 4.16: Clase *Score*

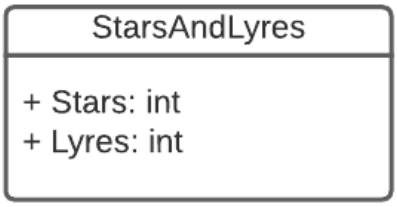

Figura 4.17: Clase *StarsAndLyres*

En la clase **Score** contamos con un constructor que inicializa la lista al tamaño necesario incluyendo la clase **StarsAndLyres** en cada posición.

4.8.2 Opciones

Por último, comentaremos la clase **Options**, mostrada en la Figura 4.18.

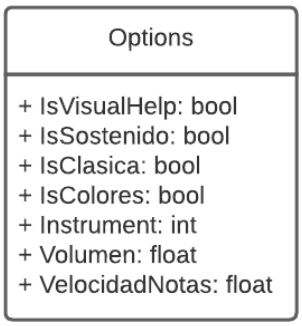

Figura 4.18: Clase *Options*

En esta clase se guardan todas las opciones que usaremos posteriormente para configurar todos los apartados necesarios, tanto del apartado gráfico como sonora.

Como nota general, ambas clases, **Score** y **Options**, necesitan la cabecera *[Serializable]* para poder serializarlas y guardarlas.

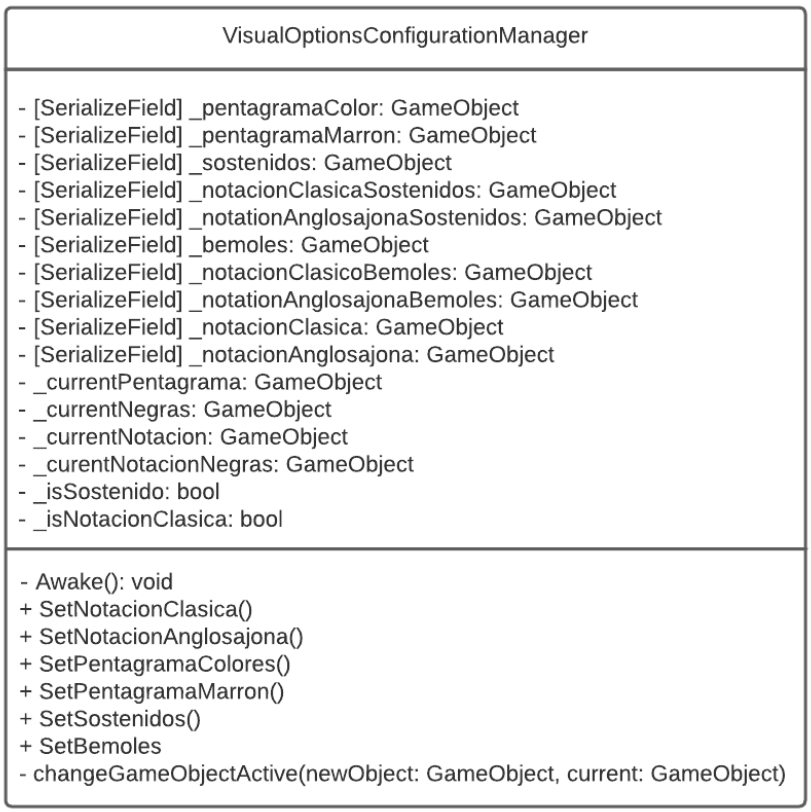

Figura 4.19: Clase *VisualOptionsConfigurationManager*

Para configurar las diferentes opciones gráficas mencionadas en el apartado de diseño gráfico usamos la clase **VisualOptionsConfigurationManager**, mostrada en la Figura 4.19.

Todos los métodos públicos hacen uso del método privado *changeGameObjectActive()* para intercambiar el nuevo *GameObject* con el antiguo. En concreto se desactiva el antiguo, se activa el nuevo y se devuelve para guardarlo en la variable *_current* correspondiente.

4.9 UI

Es hora de mostrar por pantalla todo lo implementado anteriormente.

4.9.1 Importación de sprites

En primer lugar, necesitaremos los *sprites* creados por el artista 2D que usaremos para la interfaz. Cada archivo .png que usamos hay que configurarlo al importarlo a *Unity,* como se muestra en la Figura 4.120.

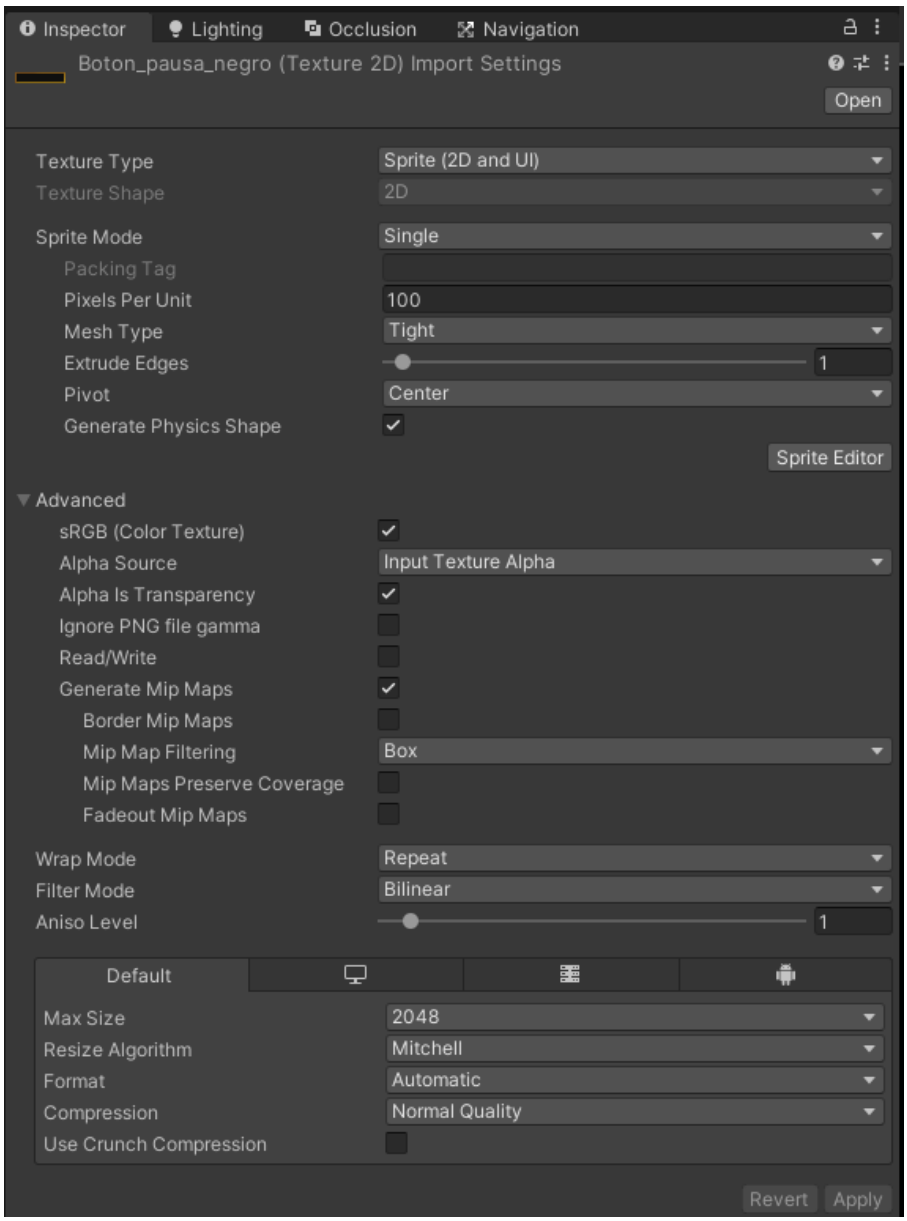

Figura 4.20: Vista inspector *sprite*

Es importante marca el *Texture Type* como ***Sprite***, poner el *Sprite Mode* a ***Single*** ya que todos los archivos son individuales y marcar que alpha en transparente.

4.9.2. Montaje y funcionalidad interfaz

Con todos los archivos importados podemos pasar a montar la interfaz siguiendo el *wireframe* del apartado de diseño. Además, añadiremos la funcionalidad a todos los componentes necesarios.

En primer lugar, comentaremos la clase **ButtonFunctionlity,** que usamos para dar funcionalidad a los botones e imágenes, mostrada en la Figura 4.21.

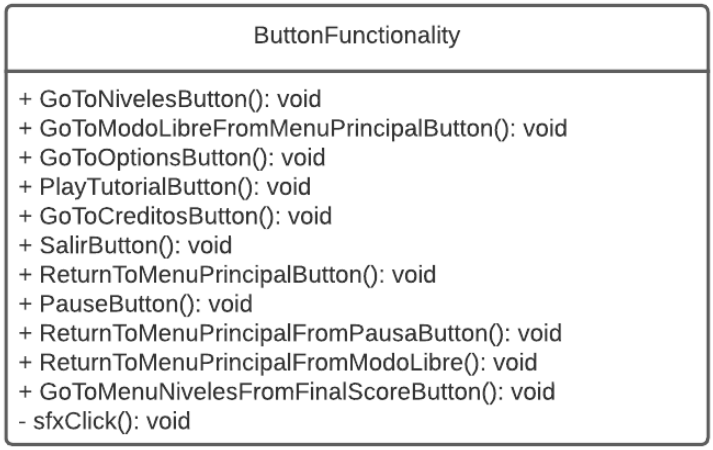

Figura 4.21: Clase *ButtonFunctionlity*

Todos los métodos públicos hacen uso de *sfxClick()* para hacer sonar el efecto de click seguido de un método de la clase **UIManager,** que realiza la navegación entre interfaces, además de otros aspectos de la interfaz que veremos a continuación.

Figura 4.22: Clase *UIManager*

Como observamos en la Figura 4.22 , esta es una clase con gran extensión, por lo que vamos a agrupar los métodos que realizan funcionalidades parecidas.

En primer lugar, en las variables *SerializeField* podemos observar algunos tipos que necesitamos mencionar:

- **TMP_Text**: clase padre de *TextMeshPro*, encargada de mostrar texto por interfaz.
- **TMP_Dropdown**: clase que hereda de TextMeshPro, implementa una lista desplegable.
- **Level_Info**: clase, mostrada en la Figura 4.23, que guarda las referencias a las imágenes de cada nivel en el menú de niveles para mostrar la puntuación.

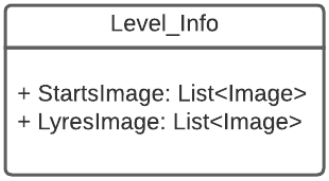

Figura 4.23: Clase *Level_Info*

Seguidamente agruparemos los distintos métodos públicos. Los dos primeros quitando *Awake()*, los usamos para mostrar y actualizar el panel de fin de juego.

Desde *ShowNivelesFromMenuPrincipal()* hasta *SetMenuPrincipalWithTutorial()* son los métodos encargados de la navegación entre las distintas interfaces, mostrando y ocultando según sea necesario.

Desde *UpdateProgresoNotas()* hasta *RestartRondaActual()* son los métodos encargados de actualizar el HUD según se progresa en el nivel.

Por último, mencionamos el método privado *showLevelPuntuacion()* que se encarga de actualizar la puntuación de los niveles en el menú de niveles con la información guardada anteriormente. Este método es el que hace uso de **Level_Info** para acceder a las imágenes a actualizar.

A continuación, mostraremos las interfaces finales y señalares los aspectos más relevantes de su implementación.

- **Menú principal**

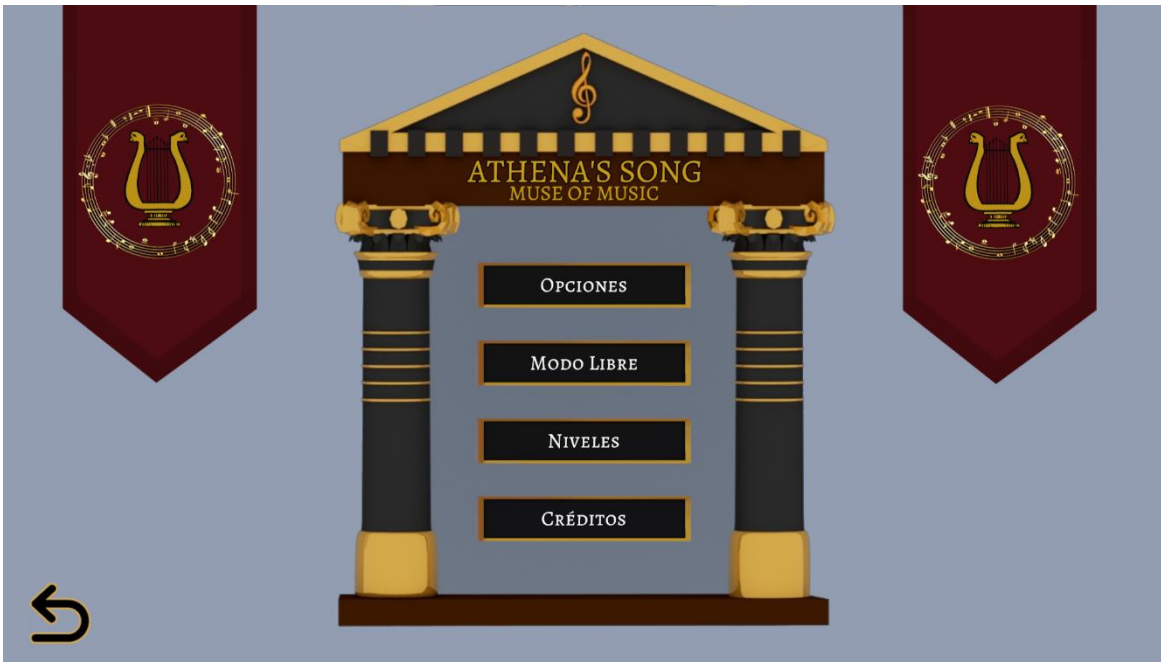

Figura 4.24: *UI* - Menú principal

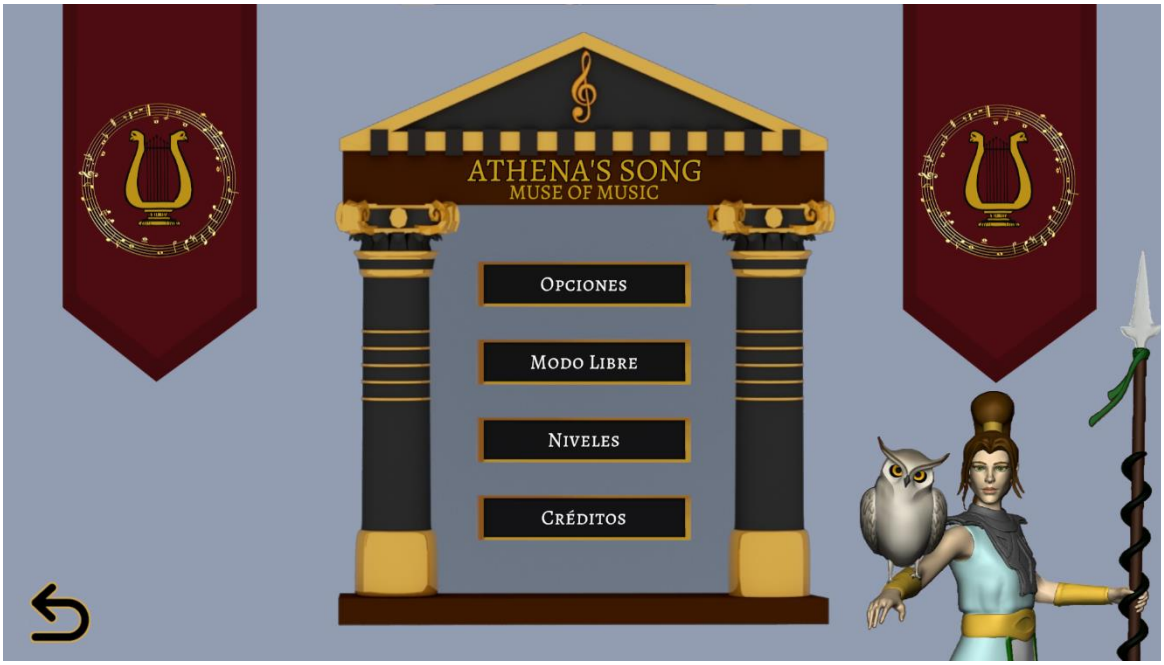

Figura 4.25: *UI* - Menú principal con tutorial

Para los botones de la Figura 4.24 hacemos uso del componente *Button* que incorpora *Unity*, configurándolo como se muestra en la Figura 4.26. En concreto, usamos el evento OnClick(), para llamar a **ButtonFunctionlity.**

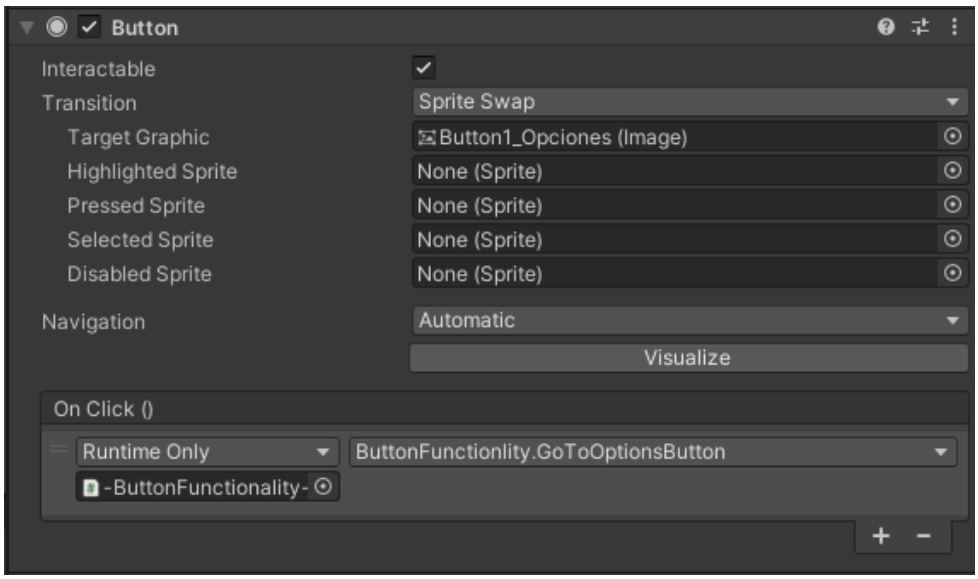

Figura 4.26: Configuración *Button*

Para la imagen que añadimos para el tutorial, que podemos observar en la Figura 4.25, al tratarse de una imagen usamos un *Event* Trigger [21], configurándolo como visualizamos en la Figura 4.27.

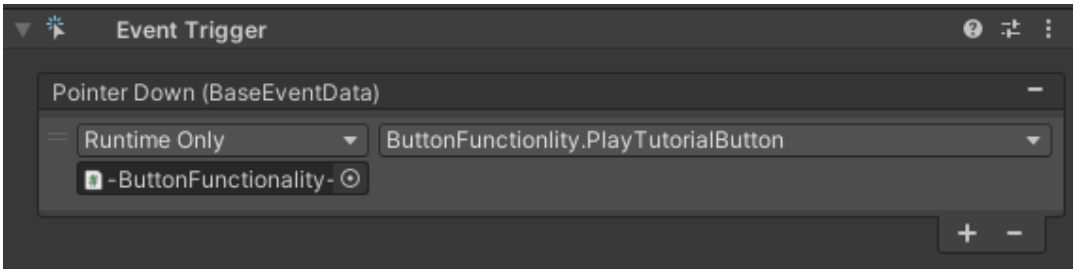

Figura 4.27: Configuración *Event Trigger*

En concreto, añadimos el evento PointerDown, que es llamado cuando se realiza un click y el puntero está sobre la imagen, asignándole de nuevo el método correspondiente de **ButtonFunctionlity.**

Para que ambos eventos funcionen en la escena tiene que existir un GameObject llamado *EventSystem*. Este GameObject se añade automáticamente al añadir un botón o *event trigger* a la escena, pero podemos eliminarlo accidentalmente, así que hay que tenerlo en cuenta.

- **Menú niveles**

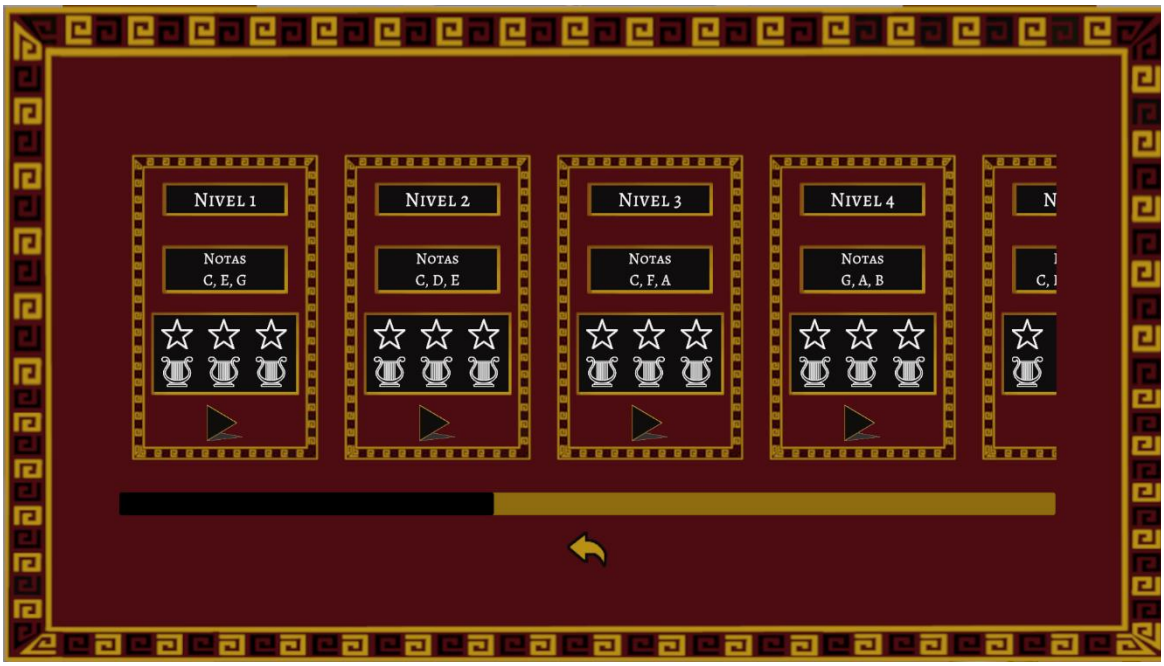

Figura 4.28: *UI* - Menú niveles

Hemos usado un *Scroll Rect* configurado como en la Figura 4.29 para tener una zona de desplazamiento horizontal.

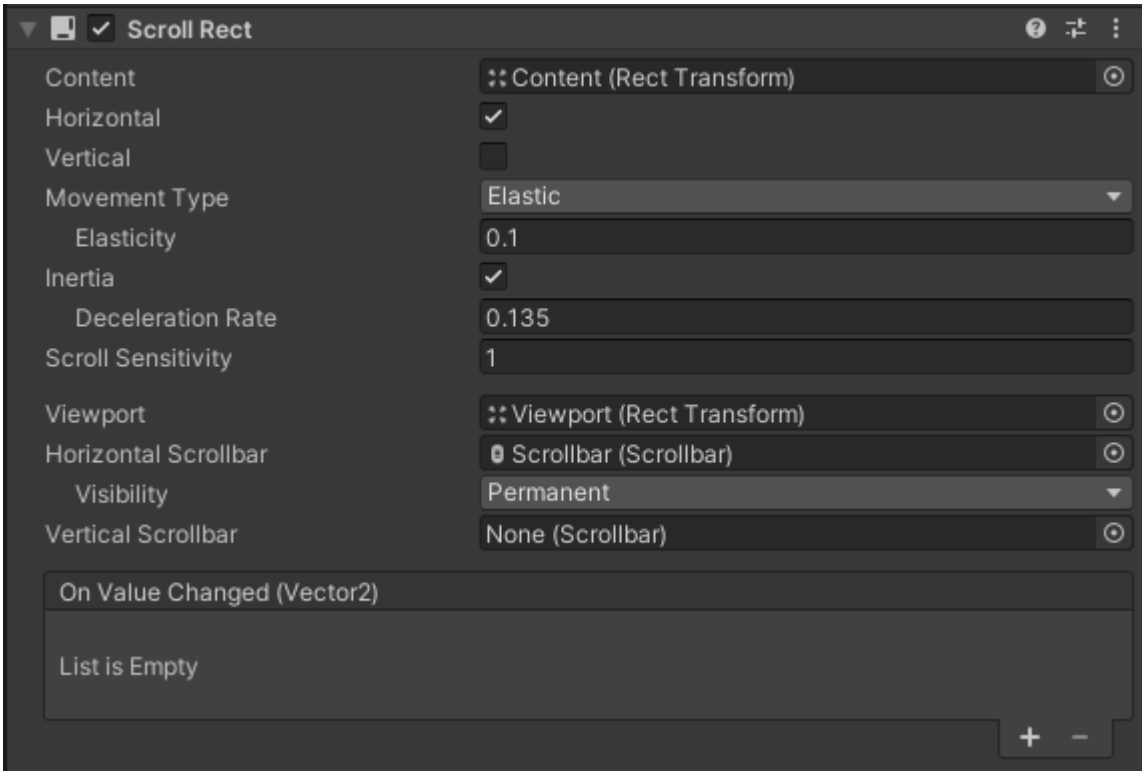

Figura 4.29: Configuración *Scroll Rect*

Para empezar cada nivel, usamos la clase **LevelsManager** como se muestra en Figura 4.30, indicando que nivel es el que empezamos.

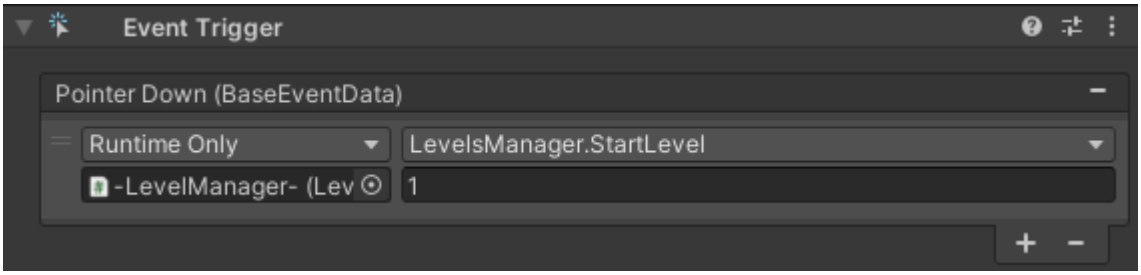

Figura 4.30: Configuración botón empezar

En la puntuación, como indicamos anteriormente usamos la clase **Level_Info** sobre cada nivel, configurando cada elemento como se observa en la Figura 4.31.

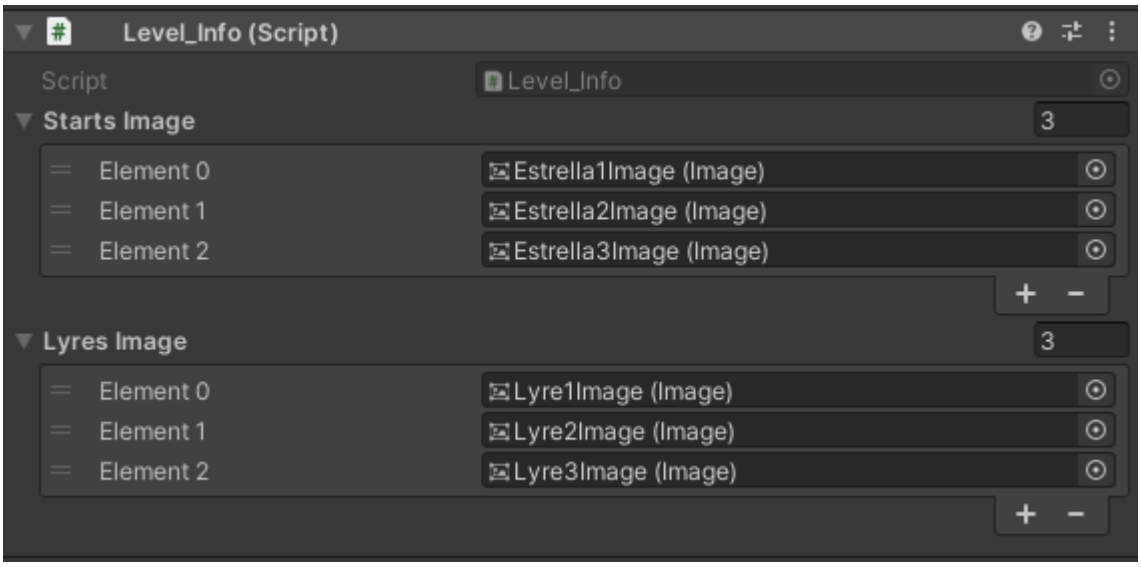

Figura 4.31: Configuración *Level_Info*

- **Menú opciones**

Figura 4.32: *UI* - Menú opciones

Vamos a comenzar comentando el apartado de gráficos. En primer lugar, para la velocidad de las notas usamos un *Slider* configurado como en la Figura 4.33.

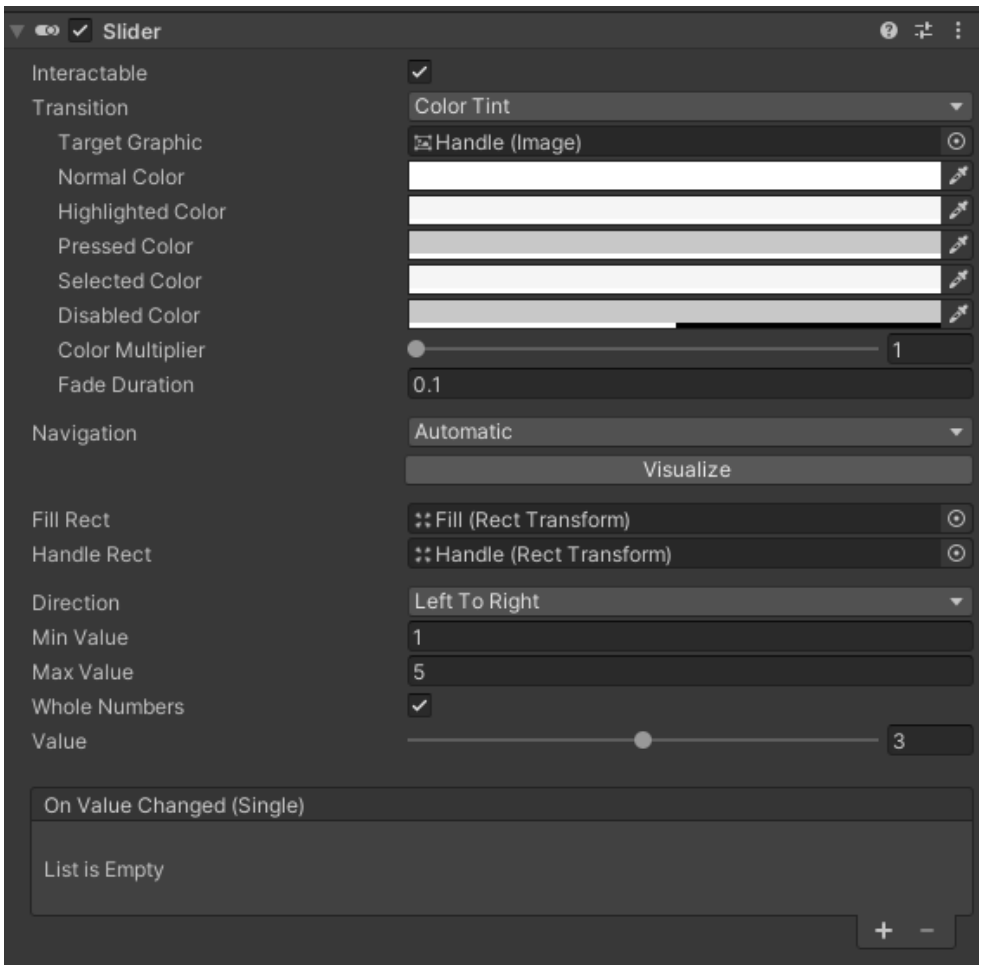

Figura 4.33: Configuración *Slider*

Además, para controlar la funcionalidad, empleamos la clase **VelocidadManager,** clase mostrada en la Figura 4.34.

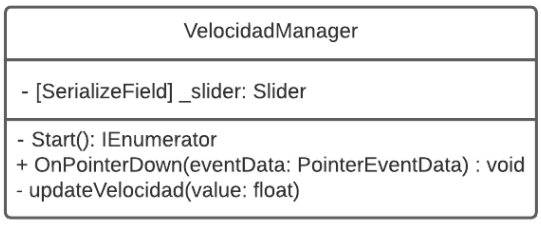

Figura 4.34: Clase *VelocidadManager*

Esta clase además de heredar de *Singleton<VelocidadManager>* también hereda de la interfaz **IPointerDownHandle**. Esta interfaz implementa el método *OnPointerDown(),* que es llamado igual que el evento *PointerDown* de un *Event Trigger*. En este caso, usamos este evento para hacer sonar un click. Para la funcionalidad empleamos el parámetro *_slider,* indicando que en el evento onValueChanged, evento que es llamado cuando se modifica el valor del *slider*, llame al método updateVelocidad. Como inciso, *Start()* tiene el tipo

IEnumerator, lo que indica que es una corutina. Esta corutina la llamamos desde **UIManager** para inicializar las variables y así tener un control más preciso del flujo de ejecución.

Continuamos implementando los interruptores o *toggles.* Para cada uno hacemos uso de la clase **ToggleFunctionality**, mostrada en la Figura 4.35.

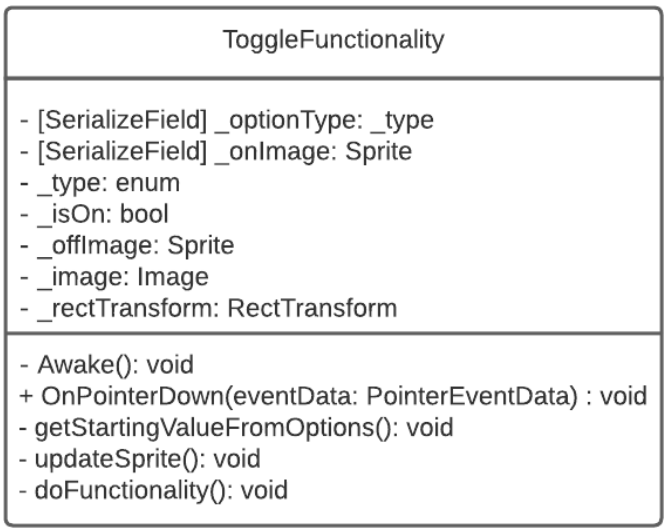

Figura 4.35: Clase *ToggleFunctionality*

En esta clase también hacemos uso de **IPointerDownHandle,** implementado *OnPointerDown().* En este caso, cada vez que se llama a este método, cambiamos el valor de *_isOn* al contrario y seguidamente llamamos a *updateSprite(),* que cambia el *sprite* al contrario (varía entre *_onImage* y *_offImage*), y finalmente a doFunctionality(), que realiza la opción indicada en el enumerado *_type* (este puede tomar valor igual a AyudaVisual, Colores, Notacion y NotacionNegras) llamando al método que corresponda de **VisualOptionsConfigurationManager.**

Seguidamente pasamos a explicar el apartado del sonido. En primer lugar, para el volumen usamos un *slider,* parecido al usado en el *slider* para la velocidad de las notas, pero este nuevo toma valores entre 0 y 1, desmarcando la opción de usar valores enteros (*Whole Numbers*). Para configurar su funcionalidad usamos la clase **VolumenManager**, mostrada en la Figura 4.36.

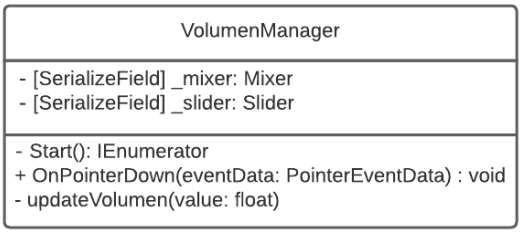

Figura 4.36: Clase *VolumenManager*

Esta clase es muy parecida a **VelocidadManager**, añadiendo que en el volumen usamos un *Mixer* [22] para configurar un parámetro expuesto anteriormente mediante *SetFloat()*.

Por último, para implementar la lista desplegable para los instrumentos empleamos el componente *Dropdown*, configurado como se muestra en la Figura 4.37.

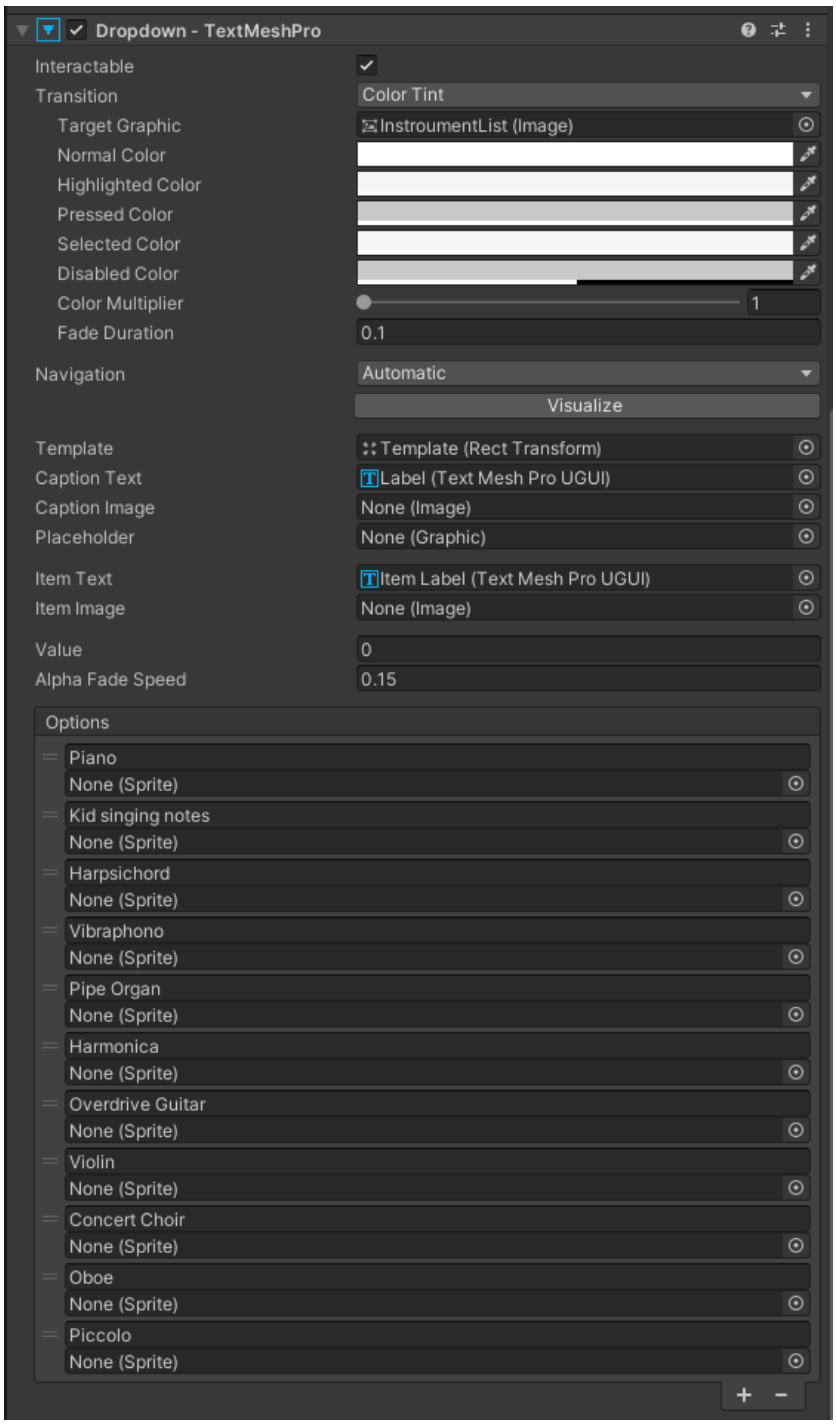

Figura 4.37: Configuración *Dropdown*

Para controlar el cambio de valor de esta lista usamos el atributo *onValueChanged* y el método *ChangeInstrument()* de la clase **UIManager.**

- **Menú créditos**

Figura 4.38: *UI* - Menú créditos

- **HUD**

Figura 4.39: *UI* - HUD

En la implementación de la barra de progreso de las rondas usamos el tipo *Filled* para la imagen, como podemos ver en la Figura 4.40, de modo que podemos cambiar el valor según se avanza de ronda mediante el *Fill Amount*.

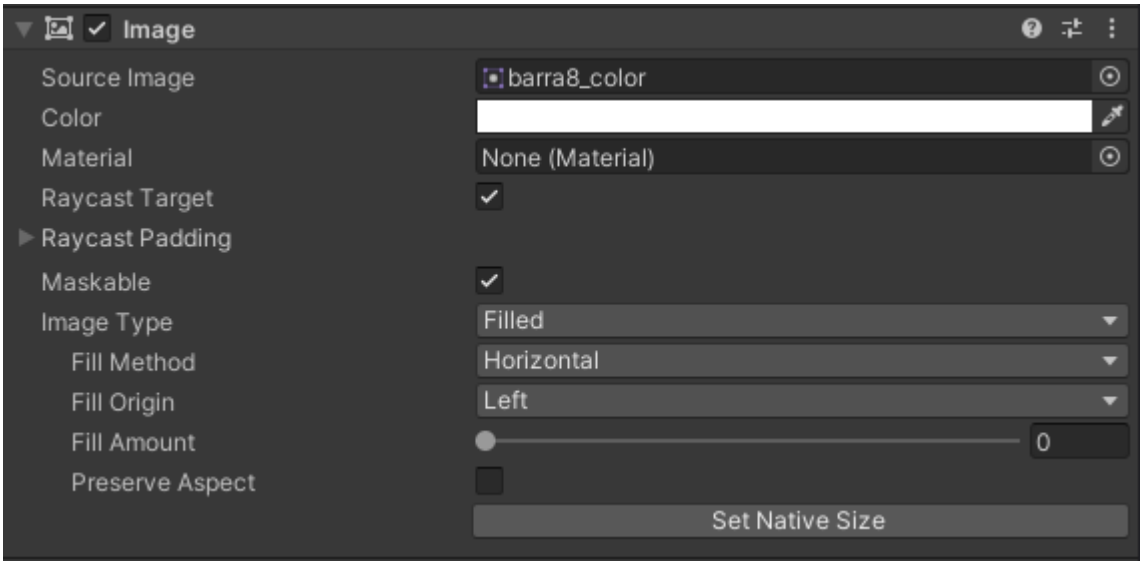

Figura 4.40: Configuración *Image*

Para implementar el progreso de las notas hacemos uso de la clase **NotasUtility**, mostrada en la Figura 4.41.

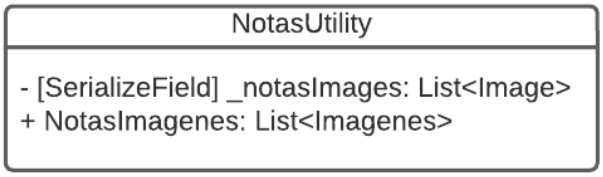

Figura 4.41: Clase *NotasUtility*

Esta clase guarda la imagen de cada nota para posteriormente ir activándolas según avanza el jugador de ronda. Inicialmente hay 3 notas activadas, sumándose una hasta un total de 10. Además de añadir la nota se mueve el conjunto para centrarlo en pantalla, todo esto controlado en **UIManager**. Así mismo, si estamos en el nivel infinito, una vez completado diez notas, se genera una nueva secuencia de notas en la interfaz, pero esta vez hay una sola nota activa inicialmente. Para mostrar esta nueva secuencia, se oculta la anterior y se muestra la nueva según sea necesario, navegando entre las distintas secuencias.

- **HUD Modo libre**

Figura 4.42: *UI* - Menú créditos

- **Menú Pausa**

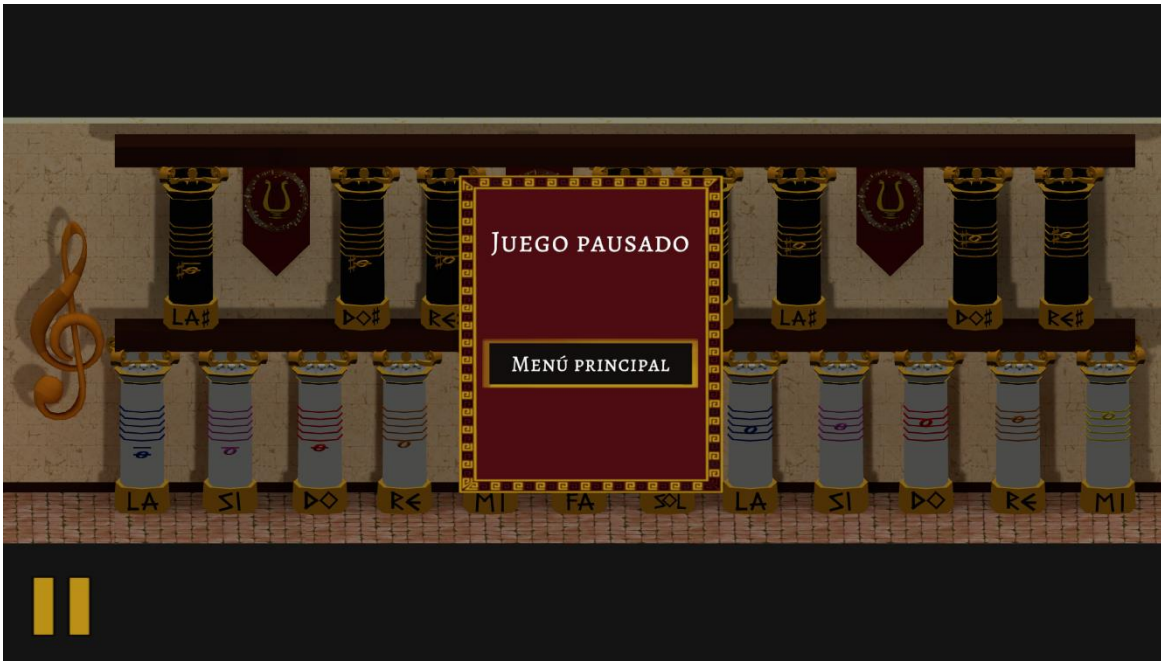

Figura 4.43: *UI* - Menú pausa

- **Panel fin de juego**

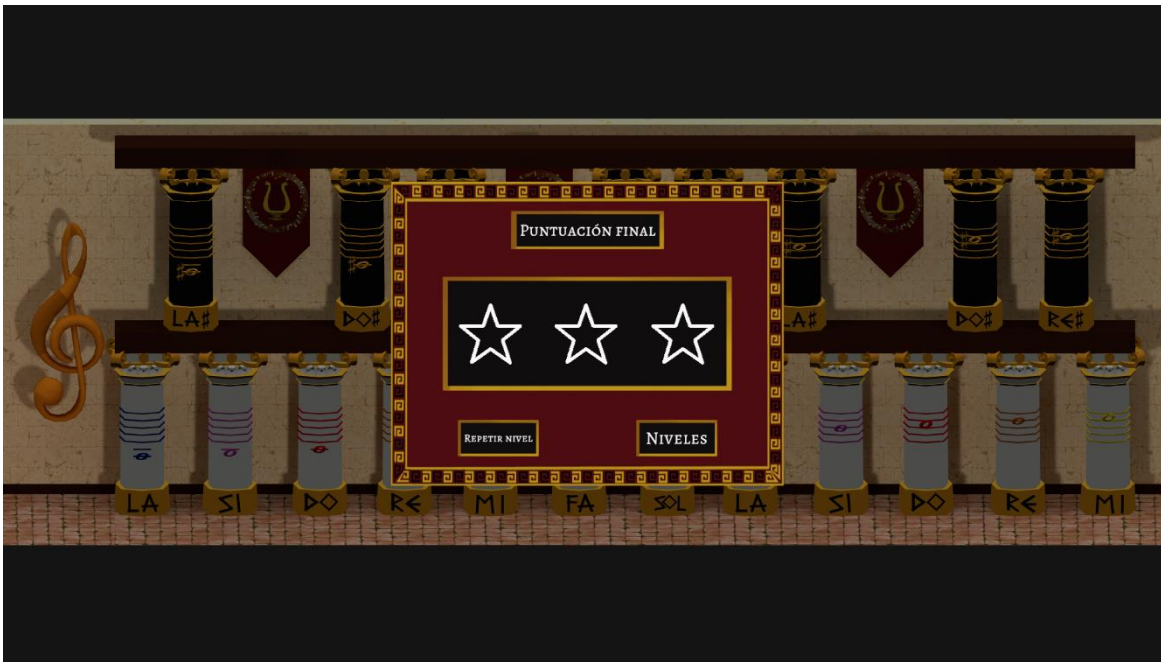

Figura 4.44: *UI* - Menú fin de juego

Como observamos en la Figura 4.44, la puntuación se muestra mediante tres imágenes, en este caso tres estrellas vacías. Si hay puntuación se cambian las imágenes vacías por imágenes desde la izquierda hasta la derecha.

- **Diálogos**

Figura 4.45: *UI* - Diálogos

A continuación, en el siguiente apartado explicaremos como se ha implementado el tutorial usando como interfaz la Figura 4.44.

4.9 Tutorial

En este apartado detallaremos qué implementación se ha seguido para los tutoriales explicados en el apartado de diseño.

4.9.1 Implementación mediante Scriptable Object

Para la implementación de los tutoriales vamos a seguir la estructura de datos que explicamos en el apartado de diseño usando *Scriptable Object*. Primero necesitamos una clase que guarde todos los datos, en concreto la clase **TutorialText,** mostrada en la Figura 4.46. Posteriormente usamos esta clase en **TutorialScriptableObject**, como se observa en la Figura 4.47.

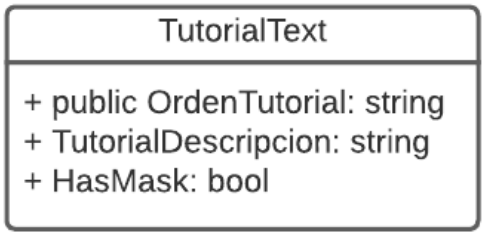

Figura 4.46: Clase *TutorialText*

Figura 4.47: Clase TutorialScriptableObject

Antes de continuar vamos a comentar las variables de **TutorialText** y sus usos. En primer lugar, *OrdenTutorial* indica el orden de los diálogos, usado para saber que diálogo estamos reproduciendo y hacer alguna acción especial según el caso. En *TutorialDescripcion* introducimos el texto del diálogo en concreto. Es importante mencionar que este texto no tenga una extensión mayor de 4 líneas, ya que es espacio es del bocadillo que usamos es limitado. Por último, en *HasMask* marcamos si usamos una máscara distinta a la por defecto. La implementación de estas máscaras la comentaremos posteriormente. Asimismo, esta clase necesita la cabecera *[Serializable]* para funcionar.

Una vez tenemos los datos que queremos guardar, montamos una lista con ellos. En esta lista el orden será el orden en el que se muestren los textos en la interfaz de diálogos. Es importante mencionar que para que todo funcione la clase **TutorialScriptableObject** tiene que heredar de la clase **ScriptableObject,** implementada por *UnityEngine*. Además,

para poder crear los objectos añadimos la siguiente cabecera a la clase: *[CreateAssetMenu(fileName = "TutorialData", menuName = "Scriptable/Tutorial")].*

Esta cabecera añade al menú de *assets* un apartado para *scriptable*, como observamos en la figura 4.48. Tras montar todo lo necesario, podemos ver un ejemplo de *scriptable object* en la Figura 4.49.

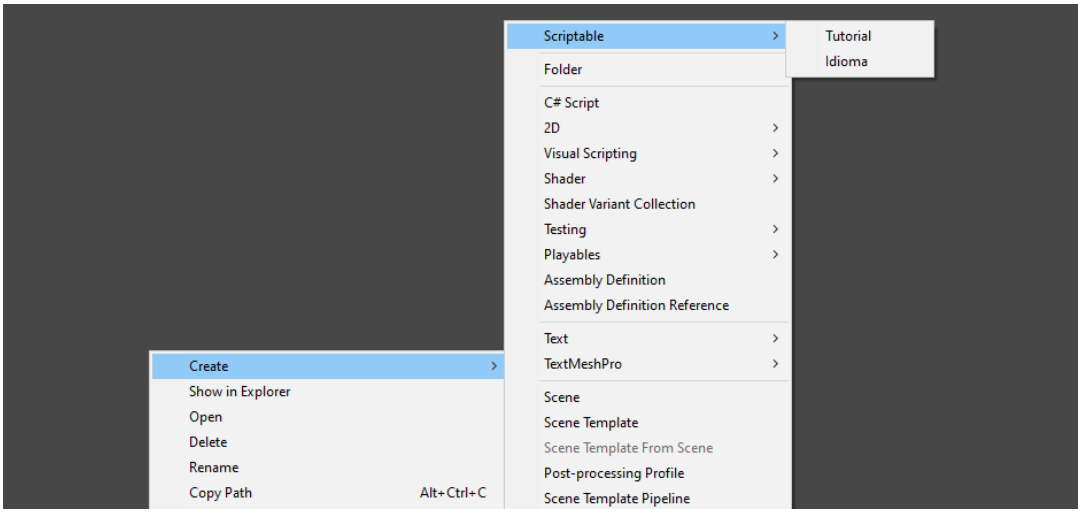

Figura 4.48: Menú *assets* con apartado para *scriptable*

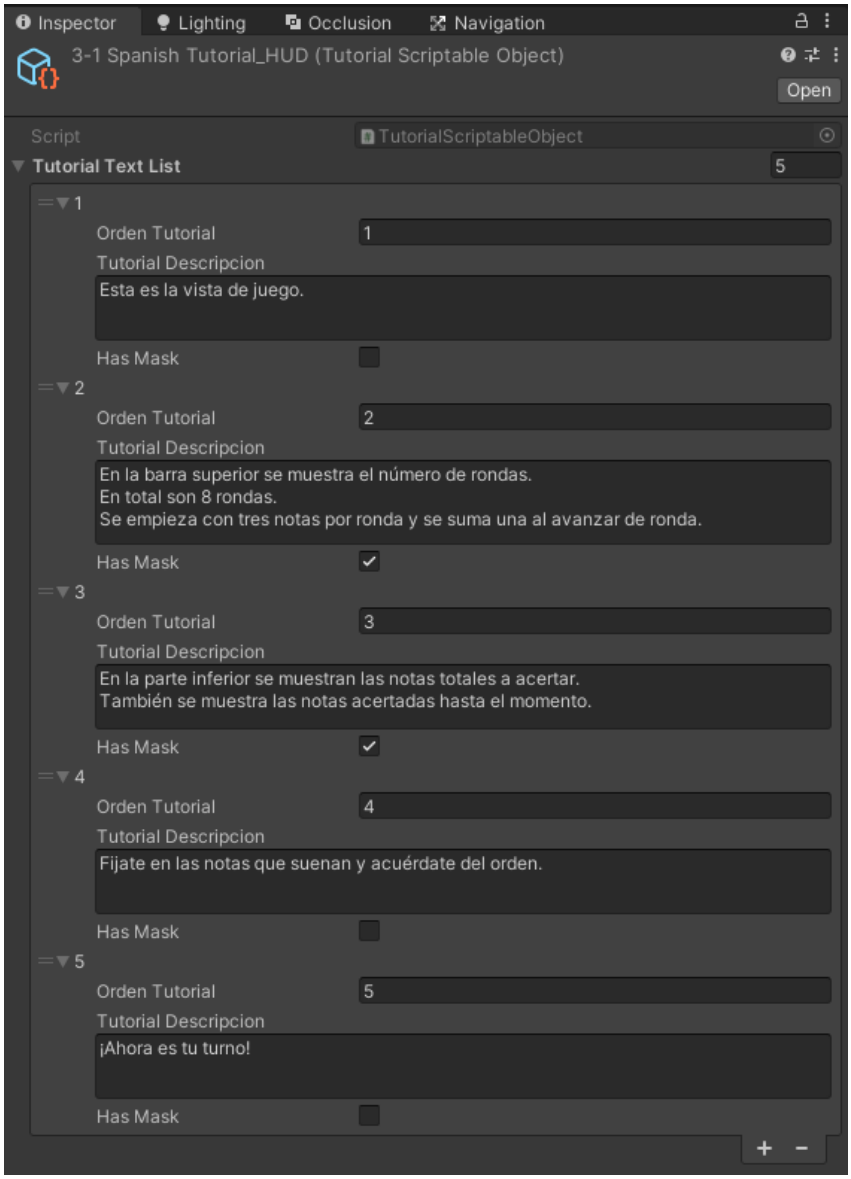

Figura 4.49: Ejemplo *scriptable object* para tutorial

Los scriptable objects comentados anteriormente los usamos en la clase **TutorialManager**, mostrada en la Figura 4.50.

Figura 4.50: Clase *TutorialManager*

En *PlayTutorial()* dependiendo de que índice pasemos por parámetros, llamamos a un *StartCoroutine()* distintos:

- Para 1 llamamos a *playMenuPrincipalTutorialCorutine()*
- Para 2 llamamos a *playNivelesTutorialCoroutine()*
- Para 3 llamamos a *playHUDTutorialCoroutine()*
- Para 4 llamamos a *playOpcionesTutorialCoroutine()*

Cada uno de estos métodos controla el tutorial correspondiente a cada menú. Estos usan los *scriptable objects* mencionados anteriormente.

En *PlayFinalText()* mostramos los diálogos finales al mostrar la pantalla de fin de juego. Para elegir que diálogo final se elige hacemos uso de la clase **DialogosFinalesManager**, mostrada en la Figura 4.51.

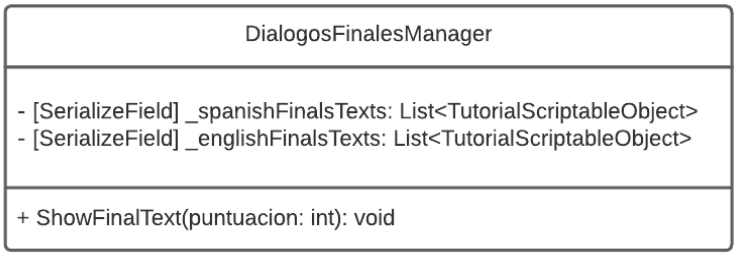

Figura 4.51: Clase *DialogosFinalesManager*

Esta clase hace uso de la misma estructura que los tutoriales, es decir, de **TutorialScriptableObject,** pero en este caso, el orden no importa, ya que se elige aleatoriamente entre los textos introducidos.

En *ShowFinalText()* dependiendo de la puntuación, se elige aleatoriamente entre las opciones posibles y posteriormente se llama a *PlayFinalText()* con el texto elegido.

Los métodos privados situados al final de **TutorialManager** son métodos auxiliares usados por los demás métodos para controlar todo el tutorial.

4.9.2 Máscaras

Seguidamente explicaremos como hemos implementado las máscaras para el tutorial, usadas para marcar sobre qué se está hablando.

Las máscaras por defecto en *Unity* ocultan la figura sobre la que se aplica y muestra lo demás. Esta funcionalidad no es la que queremos, pues necesitamos mostrar la figura sobre la que se aplica la máscara y ocultar el resto. Para ello vamos a modificar la clase *Image* proporcionada por *Unity* creando la clase **CutoutMaskUI.**

```
public class CutoutMaskUI : Image
{
    public override Material materialForRendering
    {
        get
        {
            Material material = new Material(base.materialForRendering);
            material.SetInt( name: "_StencilComp", value: (int)CompareFunction.NotEqual);
            return material;
        }
    }
}
```

Figura 4.52: Script *CutoutMaskUI*

En concreto, como observamos en la Figura 4.52, esta clase hereda de *Image.* Lo que necesitamos modificar es como se renderiza el material, para ello sobrescribimos la variable *Material* modificando el *get* del mismo. Para invertir el renderizado accedemos al componente que lo controlar mediante *SetInt*, en concreto *_StencilComp* [23], y cambiamos su valor a la función *NotEqual,* proporcionada por *UnityEngine*. Por defecto este parámetro está configurado como *Equal.*

4.9.3 Fuente

Para los diálogos vamos a modificar la fuente que usamos en otros apartados de la interfaz, añadiendo una línea exterior, además de montar un *TMP_Sprite Asset* para poder mostrar imágenes mediante texto.

- **Línea exterior**

  Esta funcionalidad la conseguimos alterando el material de la fuente como se muestra en la Figura 4.53.

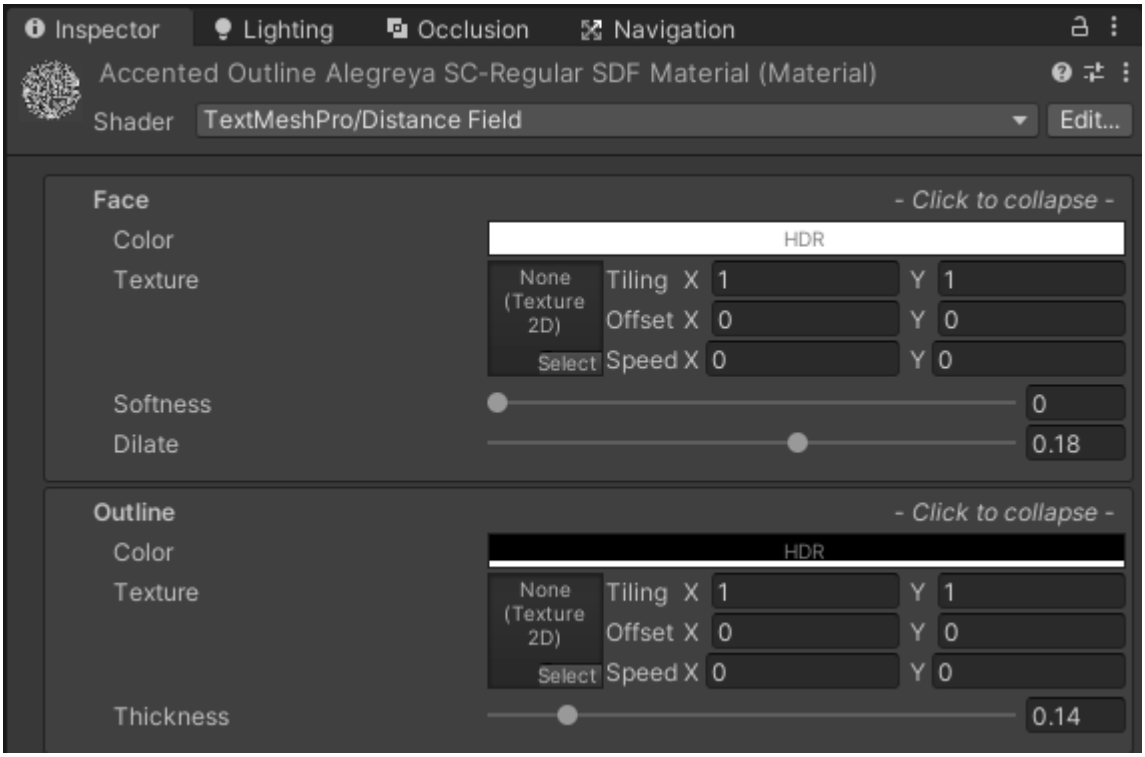

Figura 4.53: Configuración material fuente

- **TMP_Sprite Asset**

  Este componente hace uso de un *sprite* múltiple que tiene las imágenes que queremos incluir. Como vemos el Figura 4.54, al importar la imagen tenemos que

indicar en *Sprite Mode* que se trata de un archivo *Multiple*. Posteriormente, en el *Sprite Editor* cortamos cada imagen individualmente usando la función *Slice* automática, y creamos el *TMP_Sprite Asset*.

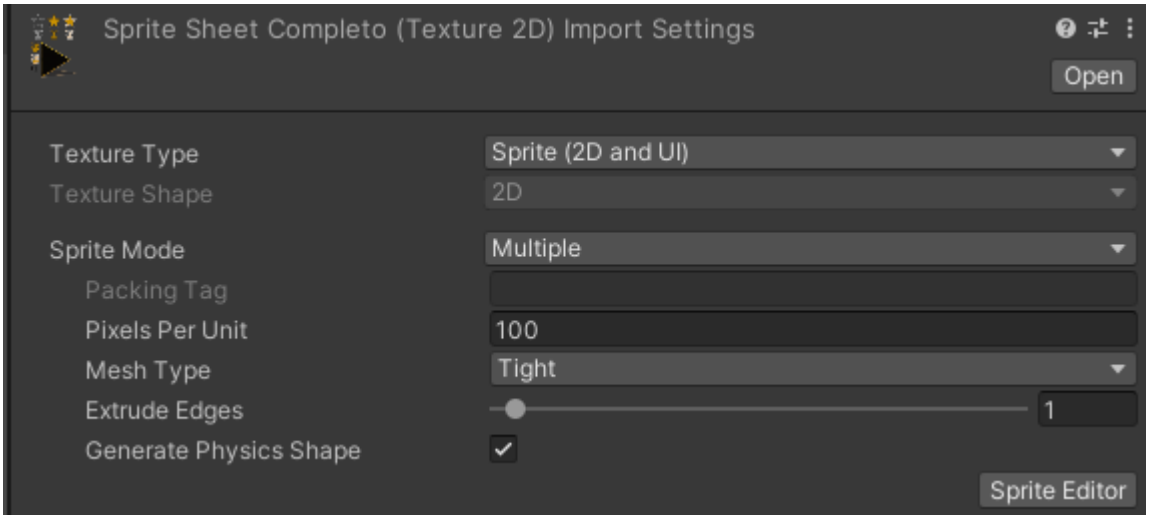

Figura 4.54: Importación *sprite* múltiple

Una vez creado el *TMP_Sprite Asset,* como observamos en la Figura 4.55, a cada imagen se le asigna un *Index*, que usamos en el texto mediante *<sprite=Index>* para incluir la imagen que queramos. Importante, para poder usar este archivo, tenemos que añadirlo al componente *TextMeshPro - Text* en el apartado *Sprite Asset* en la sección *Extra Settings*, como indicamos en la Figura 4.55.

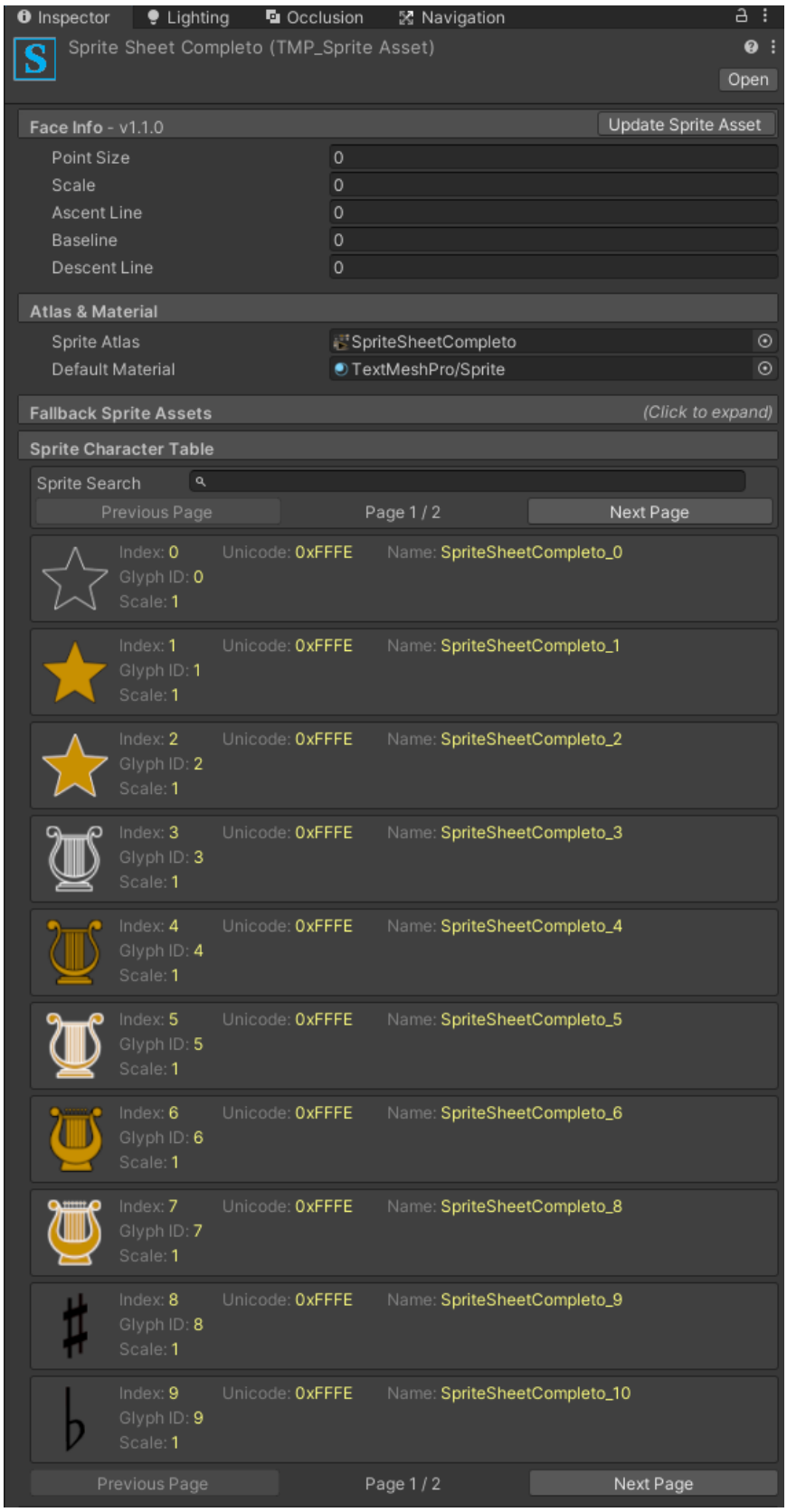

Figura 4.55: Configuración *TMP_Sprite Asset*

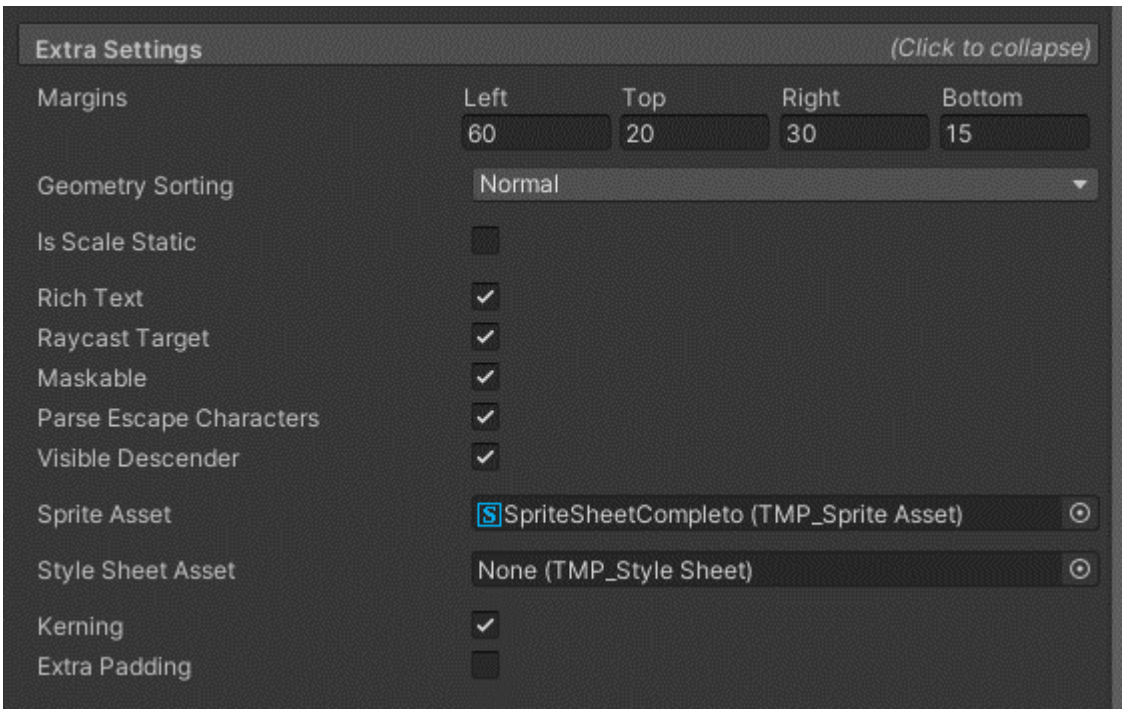

Figura 4.56: Configuración *TextMeshPro - Text*

4.10 Idioma

En este apartado expondremos como hemos configurado el idioma y qué implementación hemo seguido.

Primeramente, comentaremos como hemos implementado la estructura de datos explicada en el apartado de diseño. En concreto, usamos una estructura similar a la usada en el apartado de tutorial mediante *Scripable Object*.

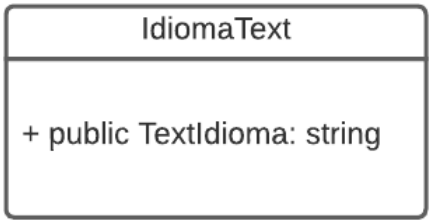

Figura 4.57: Clase *IdiomaText*

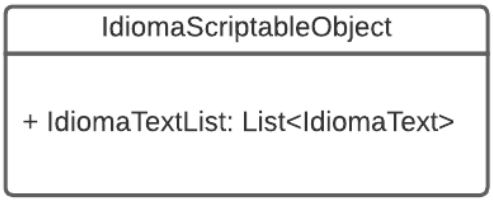

Figura 4.58: Clase Idioma*ScriptableObject*

En este caso, en lugar de necesitar guardar el orden y la máscara, solo guardamos el texto en la Clase **IdiomaText,** mostrada en la Figura 4.57. Seguidamente usamos el **IdiomaText** en la clase **IdiomaScriptableObject,** mostrada en la Figura 4.58 y creamos el *scriptable object* de la misma manera que para el tutorial. En la Figura 4.59 observamos una configuración para la traducción del menú principal para inglés.

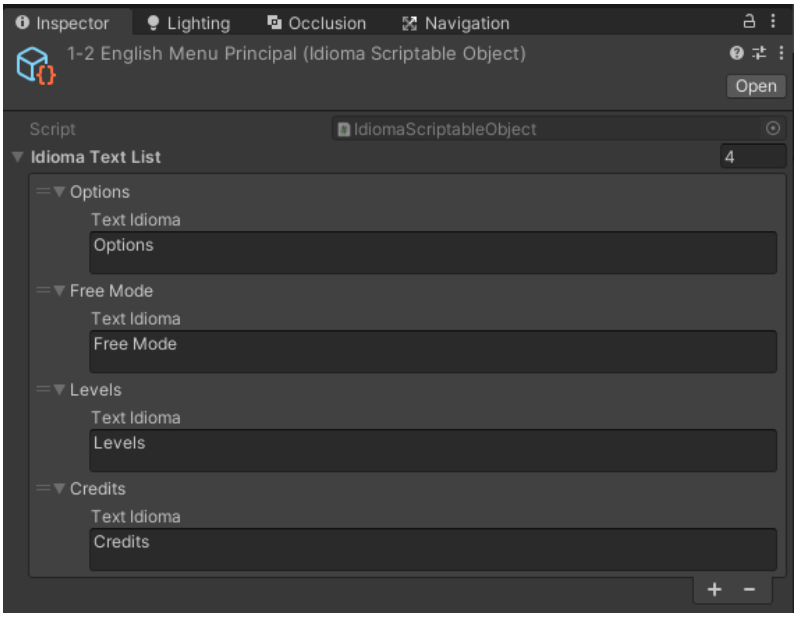

Figura 4.59: Ejemplo *scriptable object* para idioma

Para configurar el idioma necesitamos una clase con referencia a todos los textos de la interfaz para cambiarlos según el idioma y una referencia a la estructura de datos que guarda los textos a introducir. Esta clase es **IdiomaManager,** mostrada en la Figura 4.60.

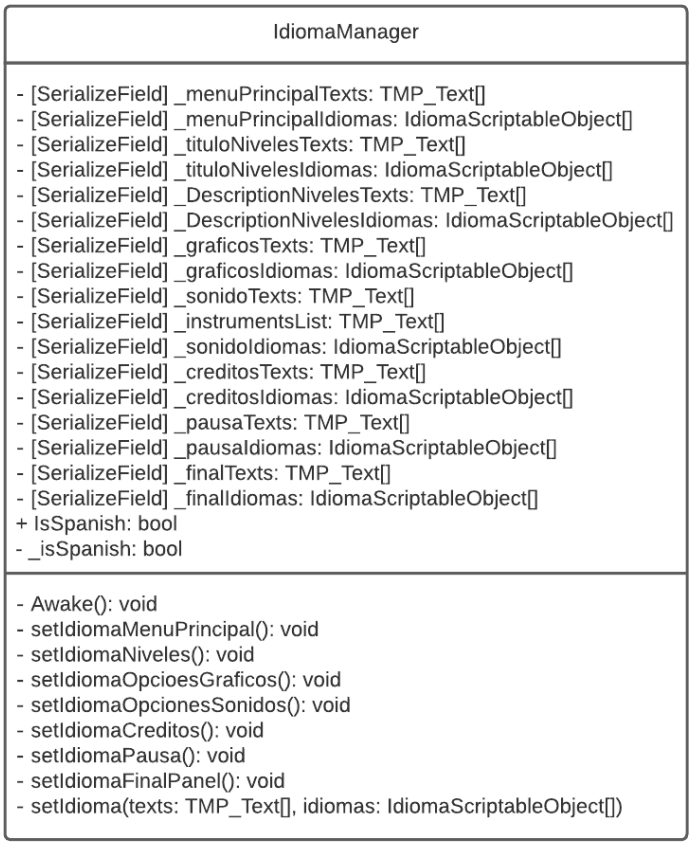

Figura 4.60: Clase *IdiomaManager*

Toda la configuración se realiza en *Awake().* Como indicamos en el apartado de diseño si el idioma del sistema es español, ponemos el idioma a español, si no es español, ponemos el idioma a inglés. Para ello, mediante *Application.systemLanguage* obtenemos el idioma del sistema. Para cambiar el idioma, a continuación, llamamos a los métodos desde *setIdiomaMenuPrincipal()* hasta *setIdiomaFinalPanel().* Estos métodos hacen uso de *setIdioma()* que usan las referencias correspondientes a la interfaz a cambiar y el idioma a usar.

4.11 Orden de ejecución

Con esto tenemos todos los scripts implementados, pero ahora hay que indicar el orden correcto de ejecución, ya que Unity realiza este de manera aleatoria. Aunque podemos controlar la ejecución con los métodos *Awake()* y *Start(),* con estos solo tenemos dos niveles de ejecución y para nuestro proyecto necesitamos más. Por ello hacemos uso de la opción *Script Execution Order* [24] que nos ofrece *Unity* en *Project Settings.*

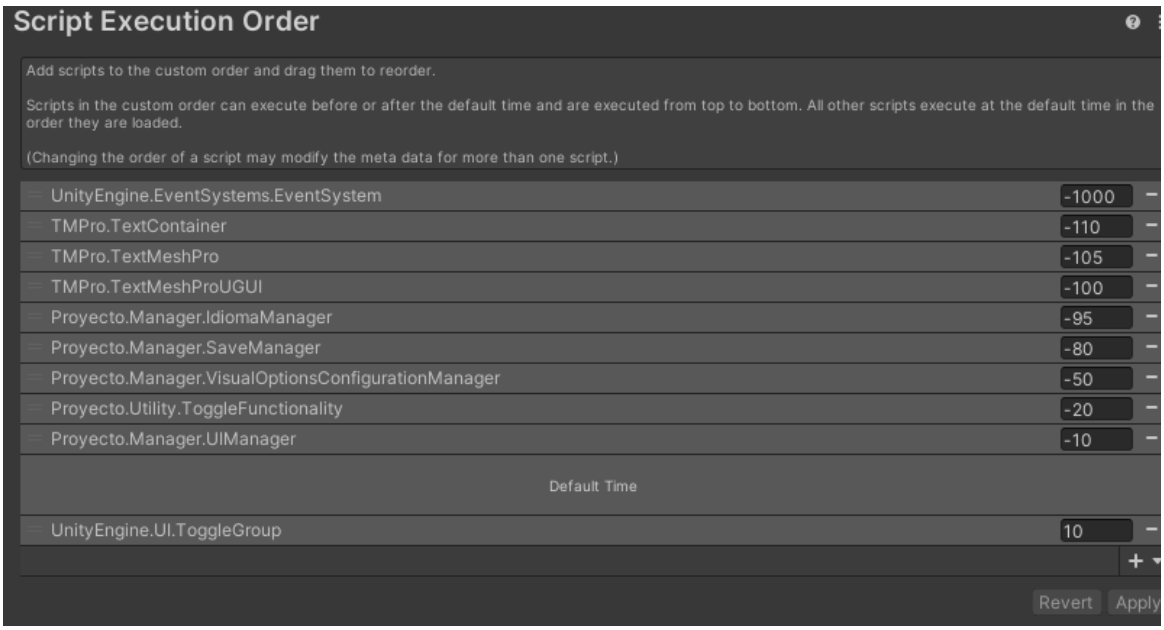

Figura 4.61: Configuración Script Execution Order

Como se muestra en la Figura 4.61, el *Default Time* es donde Unity ejecuta los *scripts* por defecto, en orden aleatorio. Para ejecutar un *script* antes que el tiempo por defecto, introducimos el script antes del *Default Time*.

A continuación, explicamos el orden que hemos configurado y por qué:

- **IdiomaManager**: lo primero de todo es configurar el idioma.

- **SaveManager:** carga los archivos guardados para su posterior uso.
- **VisualOptionsConfigurationManager:** necesario porque inicializa unos parámetros que usaremos posteriormente en **ToggleFunctionality**.
- **ToggleFunctionality:** Aplica las opciones desde los archivos cargados por **SaveManager** y las aplica mediante **VisualOptionsConfigurationManager.**
- **UIManager:** Carga y configura toda la interfaz. También carga las opciones de sonido guardadas.

Todos los *scripts* no mencionados se ejecutan en Default Time.

4.12 Iluminación

Al tratarse de un entorno 3D necesitamos iluminación para visualizar todos los componentes. Nuestra iluminación proviene de dos fuentes: una luz frontal sobre el templo y una luz ambiental, configurada como se muestra en la Figura 4.62.

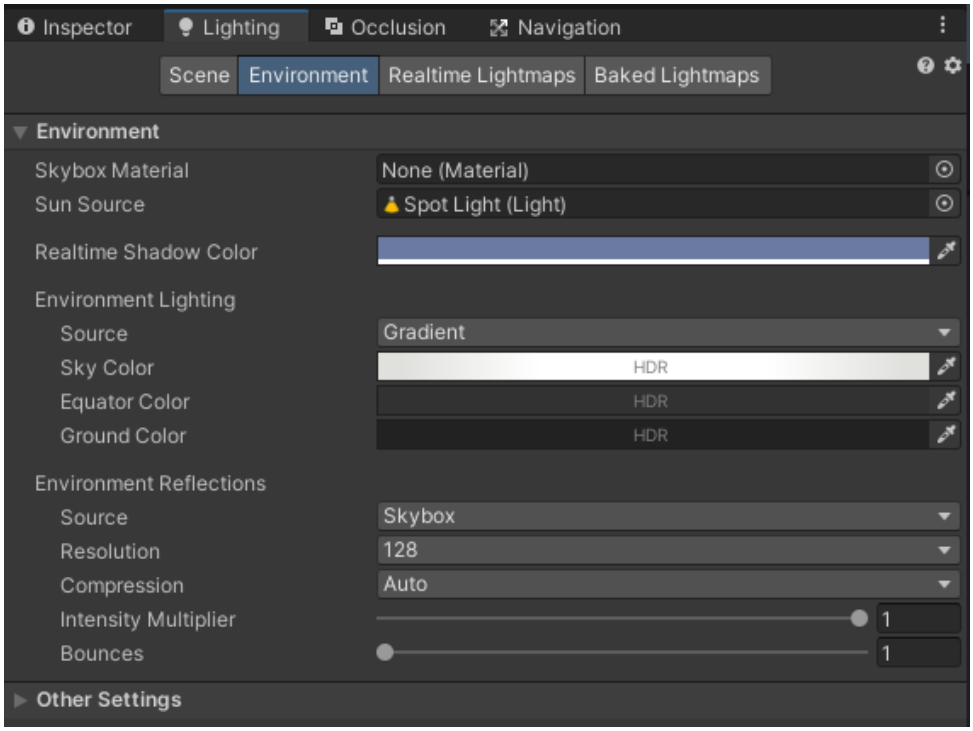

Figura 4.62: Configuración luz ambiental

Dado que la iluminación es uno de los apartados que más recursos consumen, para aumentar el rendimiento recalcularemos la iluminación con la configuración que se vemos en la Figura 4.63.

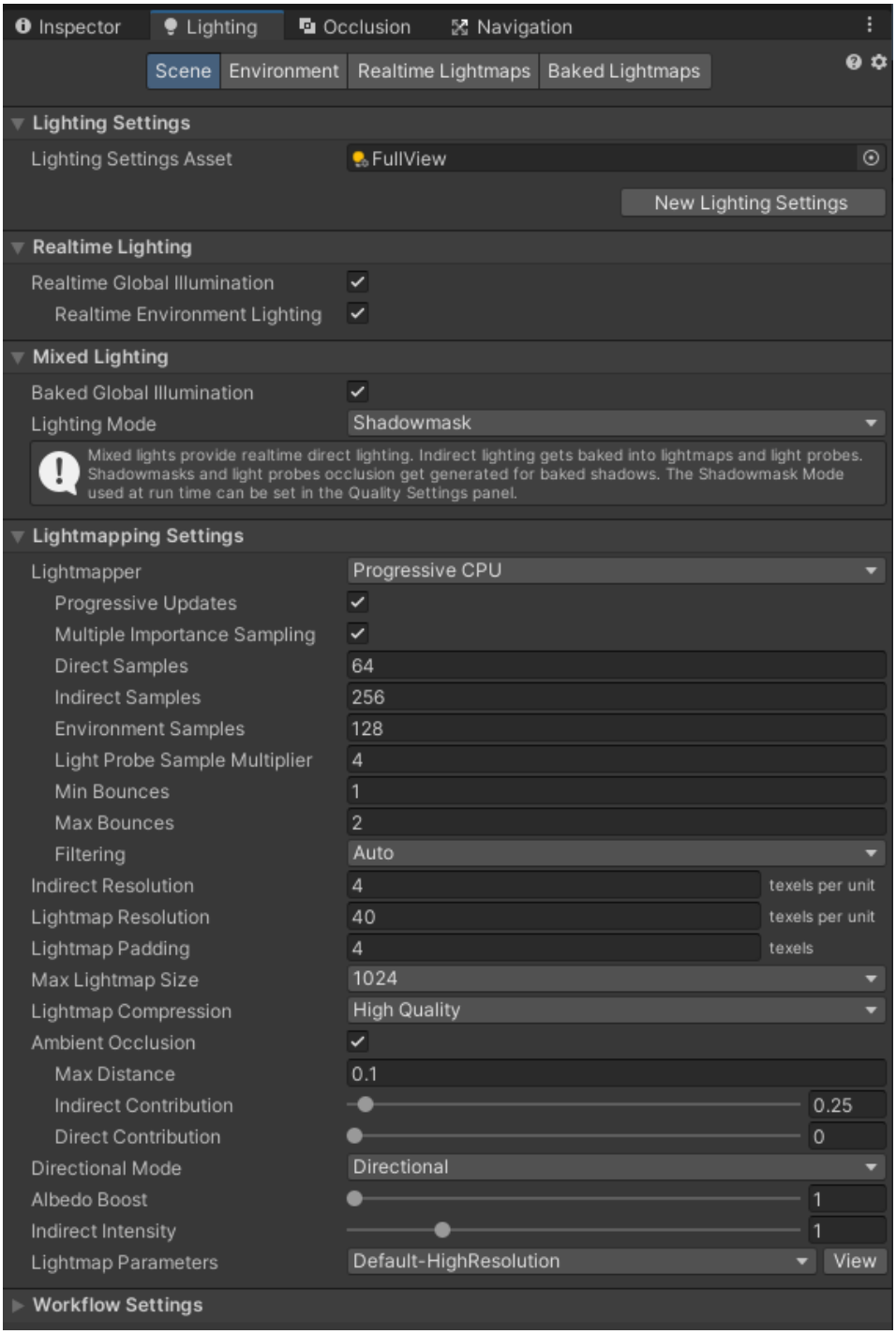

Figura 4.63: Configuración precálculo iluminación

5. Conclusiones

En este apartado expondremos nuestra experiencia de aprendizaje personal, junto con los problemas que hemos enfrentado y qué solución hemos aplicados. También comentaremos las posibles mejoras sobre el producto y cambios a futuro.

5.1 Aprendizaje personal

Nuestro aprendizaje lo podemos dividir en distintos apartados:

- **Comunicación**

Durante la duración del proyecto nos hemos comunicado tanto con el artista como con el *Product Owner*. Primeramente, para comunicarse con el artista hemos aprendido que archivos y referencia necesita para completar su trabajo. Así mismo, necesitamos una comunicación fluida para completar los plazos establecidos.

En la comunicación con el *Product Owner* realizada mediante las reuniones , hemos aprendido a informar sobre los avances del producto, así como entender los cambios que sugiere.

- **Entorno Unity**

Para llevar a cabo este proyecto hemos aprendido como usar el motor gráfico. Hemos obtenido conocimientos sobre los distintos apartados como la iluminación, la creación de objectos, ya sea para la interfaz, el sonido, etc. Hemos aprendido a crear un comportamiento mediante la programación en lenguaje C#. En este lenguaje hemos aprendido a usar una parte de la API que nos proporciona el motor, como el uso de *Raycasts*, la serialización de datos, el sistema de entrada para el jugador o la comunicación entre *scripts*.

5.2 Problemas técnicos

A continuación, nombraremos los problemas que hemos encontrado junto con la solución implementada:

- Error al guardar un archivo JSON: cuando implementamos el sistema de guardado de puntuación, al serializar el archivo de puntuación a un documento JSON, este no se creaba correctamente. Para solucionar este error, cambiamos el modo en el que se crea el archivo JSON. Este problema nos llevó más tiempo del esperado, ya que en primera instancia no pensamos que sería en modo de serializar el documento el que daba error, sino como creamos el archivo a serializar.

- Problemas al visualizar el entorno: a la hora de usar la cámara frontal, el modelo 3D tiene pequeños detalles que no se visualizaban correctamente. Para solucionarlo, en lugar de tener la vista totalmente frontal, hemos subido la cámara de altura y hemos añadido un poco de ángulo en dirección inferior. Este problema también se podría haber solucionado cambiando el modelo 3D, pero el artista 3D no estaba disponible.

5.3 Trabajo futuro y mejoras

Para finalizar comentaremos el futuro del producto y las mejoras posibles a implementar.

El principal trabajo a futuro es la expansión a nuevos dispositivos, en concreto a dispositivos móviles. Al poder usar dispositivos móviles tenemos un nuevo sistema de entrada, con múltiples posibilidades, es decir, más de una entrada a la vez. Esto nos da pie a la inclusión de acordes, es decir, varias notas a la vez, con lo que serían necesarios nuevos niveles que usasen estos acordes.

Así mismo, este producto se enmarca en el desarrollo de otra serie de videojuegos serios enfocados en el ámbito musical llevados a cabo por el grupo de investigación ATIC. Un desarrollo a futuro es la unión de todos estos videojuegos en uno mismo, creando un centro de unión para todos.

Bibliografía

[1] “Serious games: qué son, ejemplos y tipos” https://gestionet.net/serious-games-blog/ (Acceso en marzo de 2023).

[2] “¿Qué es SCRUM? Definición y términos de la metodología ágil” https://donetonic.com/es/que-es-scrum/ (Acceso en marzo de 2023).

[3] "Página oficial de Unity" https://unity.com/es (Acceso en marzo de 2023).

[4] "Página oficial para el aprendizaje de Unity" https://learn.unity.com/ (Acceso en marzo de 2023).

[5] "Documentación del motor de Unity" https://docs.unity3d.com/es/530/Manual/UnityManual.html (Acceso en marzo de 2023).

[6] "Documentación de la API de Unity" https://docs.unity3d.com/ScriptReference/ (Acceso en marzo de 2023).

[7] "Documentación oficial de C#" https://learn.microsoft.com/en-us/dotnet/csharp/ (Acceso en marzo de 2023).

[8] "Página oficial de Git" https://git-scm.com/ (Acceso en marzo de 2023).

[9] "Página oficial de GitHub Desktop" https://desktop.github.com/ (Acceso en marzo de 2023).

[10] "Página oficial de JetBrain Rider" https://www.jetbrains.com/es-es/rider/ (Acceso en marzo de 2023).

[11] "Documentación sobre Scriptable Object" https://docs.unity3d.com/Manual/class-ScriptableObject.html (Acceso en abril de 2023).

[12] "Documentación sobre properties" https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties (Acceso en marzo de 2023).

[13] "Documentación sobre MonoBehaviour" https://docs.unity3d.com/ScriptReference/MonoBehaviour.html (Acceso en marzo de 2023).

[14] "Documentación sobre Physics.Raycast" https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (Acceso en marzo de 2023).

[15] "Página oficial Dotween" http://dotween.demigiant.com/ (Acceso en marzo de 2023).

[16] "Documentación sobre abstract" https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract (Acceso en marzo de 2023).

[17] "Documentación sobre corutinas en Unity" https://docs.unity3d.com/Manual/Coroutines.html (Acceso en marzo de 2023).

[18] "Página de Maestro en la Asset Store" https://assetstore.unity.com/packages/tools/audio/maestro-midi-player-tool-kit-free-107994 (Acceso en abril de 2023).

[19] “MIDI” https://es.wikipedia.org/wiki/MIDI (Acceso en abril de 2023).

[20] "Documentación sobre persistentDataPath" https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html (Acceso en abril de 2023).

[21] "Documentación sobre EventTrigger" https://docs.unity3d.com/2018.4/Documentation/Manual/script-EventTrigger.html (Acceso en abril de 2023).

[22] "Documentación sobre Mixer" https://docs.unity3d.com/Manual/AudioMixer.html (Acceso en abril de 2023).

[23] "Documentación sobre Stencil" https://docs.unity3d.com/Manual/SL-Stencil.html (Acceso en abril de 2023).

[24] "Documentación sobre Script Execution Order" https://docs.unity3d.com/Manual/class-MonoManager.html (Acceso en abril de 2023).

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga