Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

REMAKE DE UN VIDEOJUEGO CLÁSICO 2D DE PLATAFORMAS ESTILO SUPER MARIO BROS

Alumno

Alfonso José Arroyo Loaisa

Tutor

Carlos Javier Ogayar Anguita
(Departamento de Informática)

Septiembre, 2021

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Carlos Javier Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: '**REMAKE DE UN VIDEOJUEGO CLÁSICO 2D DE PLATAFORMAS ESTILO SUPER MARIO BROS**', que presenta Don Alfonso José Arroyo Loaisa, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2021

El alumno:

ARROYO LOAISA ALFONSO JOSE - 26522589R

Firmado digitalmente por ARROYO LOAISA ALFONSO JOSE - 26522589R
Fecha: 2021.09.06 11:32:19 +02'00'

Alfonso José Arroyo Loaisa

El tutor:

OGAYAR ANGUITA CARLOS JAVIER - 26039170L

Firmado digitalmente por OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-26039170L, givenName=CARLOS JAVIER, sn=OGAYAR ANGUITA, cn=OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Fecha: 2021.09.06 11:34:56 +02'00'

Carlos Javier Ogayar Anguita

(Página intencionalmente en blanco)

Agradecimientos

A mis padres, Alfonso y Pilar, y mi hermano, Alejandro, por haberme ayudado siempre en todo lo que han podido y más, y haber dado todo para que pueda llegar aquí. Por apoyarme siempre en cada una de mis decisiones, sin ellos nada de esto habría sido posible. Sois la mejor familia que se pueda tener.

A Marina, por soportarme, por darme ánimos y fuerzas para seguir de donde no había, siempre con una sonrisa. Por creer en mi cuando ni yo mismo creía, y por todo el tiempo que hemos pasado y que pasaremos juntos. Siempre has hecho que hasta el peor de los días pudiera tener algo bueno.

A Jaime, el mejor compañero de piso y, sobre todo, amigo que se pueda tener. Por aguantarme todos estos años y compartir tantas cosas juntos, tanto alegrías como tristezas. Estos años de carrera no habrían sido lo mismo sin ti.

A mi tutor de TFG, Carlos Javier, por guiarme y ayudarme en todo el proceso del TFG.

Y por último a todos los profesores de la Universidad de Jaén que he tenido a lo largo de estos años. Gracias por todo lo que me habeis enseñado. Aunque a veces haya tenido días interminables de estudio, he aprendido muchísimo de todos vosotros.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Sin especialidad
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Alfonso José Arroyo Loaisa
Fecha de asignación	10/03/2021
Descripción corta	Remake del videojuego clásico Super Mario Bros con un motor de videojuegos actual, incluyendo un generador aleatorio de niveles.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

Contenido

Índice de ilustraciones

Índice de tablas

1 INTRODUCCIÓN

En este documento se detallan las etapas de planificación, diseño y desarrollo de nuestro proyecto. El desarrollo de este tiene como objetivo crear un *remake* de un videojuego clásico de plataformas de Super Mario Bros.

De los apartados uno a cinco se presentarán las tareas de planificación, incluyendo los objetivos del proyecto y el software empleado.

De los apartados seis a ocho está comprendida la etapa de diseño, profundizando en el diseño de las mecánicas, la interfaz de usuario y los niveles.

El apartado nueve comprende el desarrollo del proyecto, dividido por iteraciones.

El apartado diez se extraerán las conclusiones del proyecto.

Finalmente, de los apartados once a trece se incluyen los apéndices, definiciones y abreviaturas, y la bibliografía utilizada.

2 OBJETIVOS

El objetivo final de este TFG es el desarrollo de un *remake* de un videojuego clásico como Super Mario Bros con un motor de desarrollo de videojuegos actual, reinventando todos sus niveles e incluyendo nuevas mecánicas de los juegos más actuales de Super Mario. Podríamos definir una serie de objetivos generales:

- Identificar el juego del que se realizará el *remake*, las mecánicas del juego elegido y las posibles nuevas mecánicas y modificaciones que se incluirán.
- Concretar y definir las mecánicas a desarrollar.
- Seleccionar el motor de videojuegos más adecuado, así como las herramientas auxiliares que se requerirán para el desarrollo.
- Análisis y diseño de la interacción con el usuario y de la interfaz.
- Análisis y diseño del mundo en el que se desarrolla el videojuego, los personajes, las animaciones, efectos visuales y efectos de sonido.

- Evaluación del proyecto desarrollado.

3 VISIÓN GENERAL DEL JUEGO Y CLASIFICACIÓN

Clasificación del juego:

- Título: Super Mario Bros Remake.
- Género: plataformas, casual.
- Modo de juego: plataformas con desplazamiento lateral.
- Plataforma: PC.
- Idioma: textos en inglés, no hay voces.
- Clasificación PEGI: PEGI 3, ya que es un juego adecuado para todas las edades, sin contenido ofensivo o violento que pueda asustar a los más pequeños.

El objetivo principal del juego es superar cada nivel en el menor tiempo posible, recogiendo por el nivel el mayor número de estrellas que podamos encontrar en él. Para ello, cada nivel tendrá un tiempo límite y tres estrellas. Estos se completan llegando hasta la meta. Si el jugador cae o es eliminado por alguno de los enemigos, se reiniciará el nivel. En este reinicio se perderá una de las vidas, de las tres que teníamos inicialmente. Si se pierden todas las vidas, el juego acabará y se reiniciará desde el primer nivel. Se conseguirán vidas a lo largo de los niveles, recogiendo distintos objetos.

La mecánica principal del juego consiste en moverse por el nivel de manera lateral, izquierda a derecha. A medida que avancen los niveles, estos serán más complicados. Estos estarán agrupados por mundos. Cada mundo incluirá nuevas mecánicas y enemigos.

Este *remake* busca rejuvenecer el juego con un motor más actual y gráficos mejorados.

4 SOFTWARE UTILIZADO

4.1 Elección del motor de videojuegos

Para poder desarrollar un videojuego, debemos de tener en cuenta en primer lugar el motor con el que se procede a trabajar.

Los motores de videojuegos ofrecen un motor para renderizar gráficos 3D y 2D, motor de sonido y simulaciones de leyes físicas como la gravedad o colisiones. También aportan mecanismos para la creación de animaciones, scripts y gestión de las entradas del usuario, entre otros.

En la actualidad, se encuentran muchos motores de videojuegos. Estos motores pueden ser gratuitos o de pago, y tener distintos tipos de licencias dependiendo; si el motor será usado con fines personales, educativos o con fines profesionales. Algunos de los motores de videojuegos más populares de la actualidad son Unity, Unreal Engine o Cry Engine [1-3].

Unity es un motor creado por Unity Technologies, el cual es una buena alternativa debido a la gran comunidad de usuarios que tiene detrás, pudiendo encontrar una gran cantidad de tutoriales y documentación. De esta manera, se dará respuesta a todos los problemas que surjan durante el desarrollo de una forma sencilla.

Unity tiene soporte para la mayoría de las plataformas actuales: Windows, Mac, Linux, PS4, iOS, Android o WebGL.

Además, se puede encontrar dentro de Unity la Asset Store [4], una tienda de recursos que contiene una gran cantidad de funciones, utilidades, así como modelos, animaciones y demás.

Existen varios tipos de licencia en Unity: las licencias personales, las cuales son gratuitas si nuestros ingresos son inferiores a 100.000 $; las licencias de pago, divididas en “Plus”, “Pro” y “Empresa” con un precio de 399 $, 1800 $ y 2000 $, respectivamente.

La segunda opción valorada es Unreal Engine, un motor desarrollado por Epic Games, que al igual que Unity, tiene una gran cantidad de documentación y tutoriales además de una gran cantidad de desarrolladores que lo usan.

Unreal Engine tiene también soporte para la mayoría de las plataformas actuales incluyendo Windows, Mac, Linux, Xbox One, PS4 y Nintendo Switch.

El motor y todas sus herramientas son gratuitas a cambio del 5% de los ingresos que genere el proyecto una vez se haya desarrollado y comercializado.

Como última opción, se encuentra Cry Engine, motor desarrollado por Crytek, y que podemos descargar de forma gratuita si cuando este sea comercializado no supera los 5.000 $ en ingresos. Si se superan, tendremos que pagar un 5% de los ingresos de nuestro juego.

Al igual que los dos motores anteriores, tiene soporte para la mayoría de las plataformas actuales y cuenta con una gran cantidad de documentación y tutoriales.

Para este proyecto se usará Unity ya que es totalmente gratuito y tiene una gran cantidad de tutoriales y documentación, así como actualizaciones frecuentes que mejoran el software constantemente. Además, este motor ha sido elegido frente al resto por experiencias previas en desarrollo de videojuegos con Unity. Por lo que será mucho más fácil comenzar con el desarrollo en este software. En concreto, se utilizará la versión 2020.2.2f1.

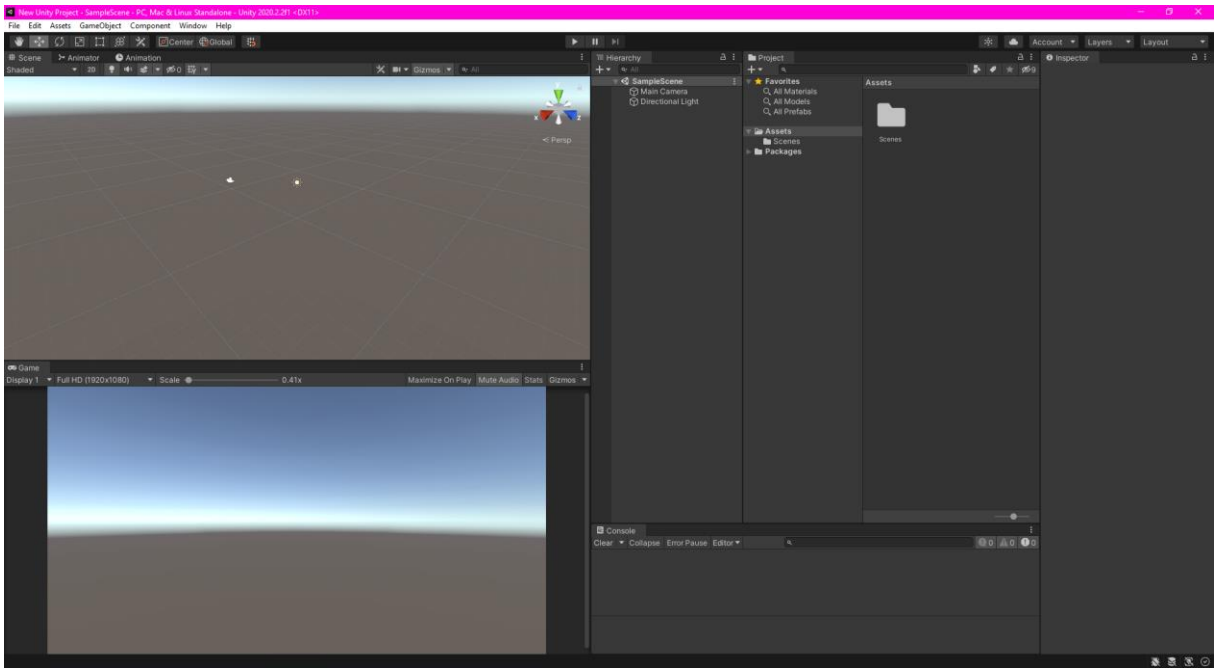


Ilustración 1 Unity 2020.2.2f1

4.2 Microsoft Visual Studio

Microsoft Visual Studio es un entorno de desarrollo integrado para Windows, Mac y Linux que es compatible con múltiples lenguajes de programación [5]. Es gratuito y, en nuestro caso, Microsoft Visual Studio viene junto con Unity y se integra

con él en el momento de escribir código para nuestro proyecto. Se ha usado la versión Community 2019 v16.8.4.

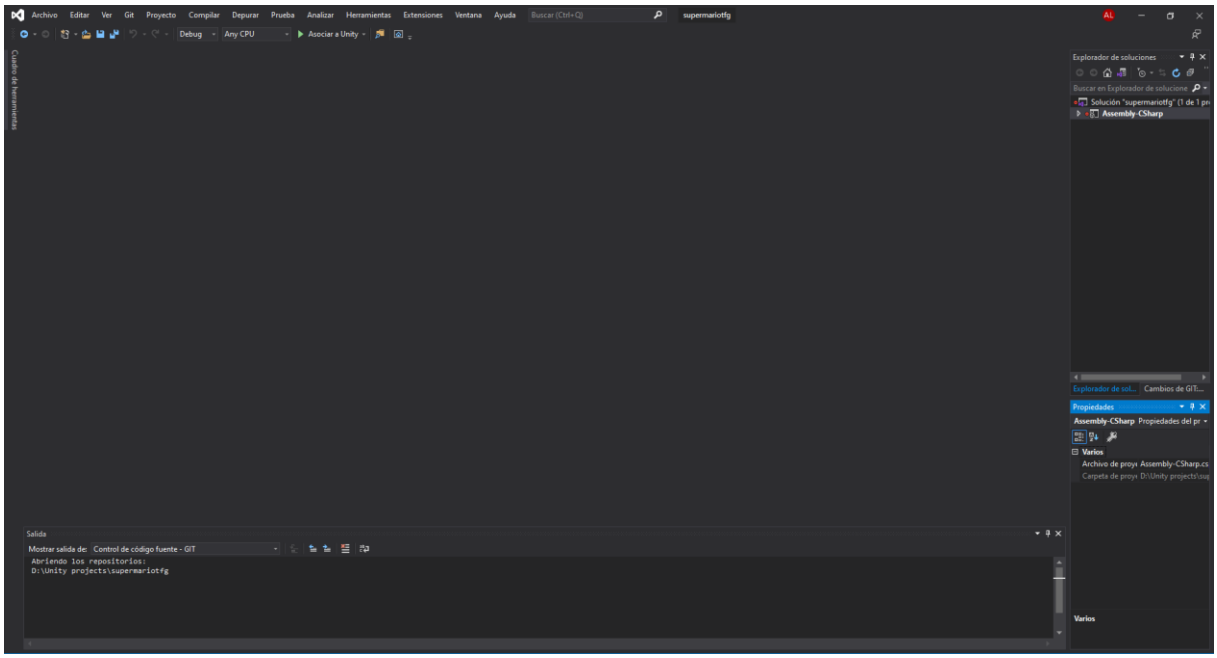


Ilustración 2 Visual Studio Community 2019 v16.8.4.

4.3 Microsoft Office Word

Microsoft Office Word es un procesador de textos desarrollado por Microsoft que pertenece al paquete de herramientas ofimáticas Microsoft Office, incluido en Microsoft 365 [6]. Será utilizado para la redacción de esta memoria.

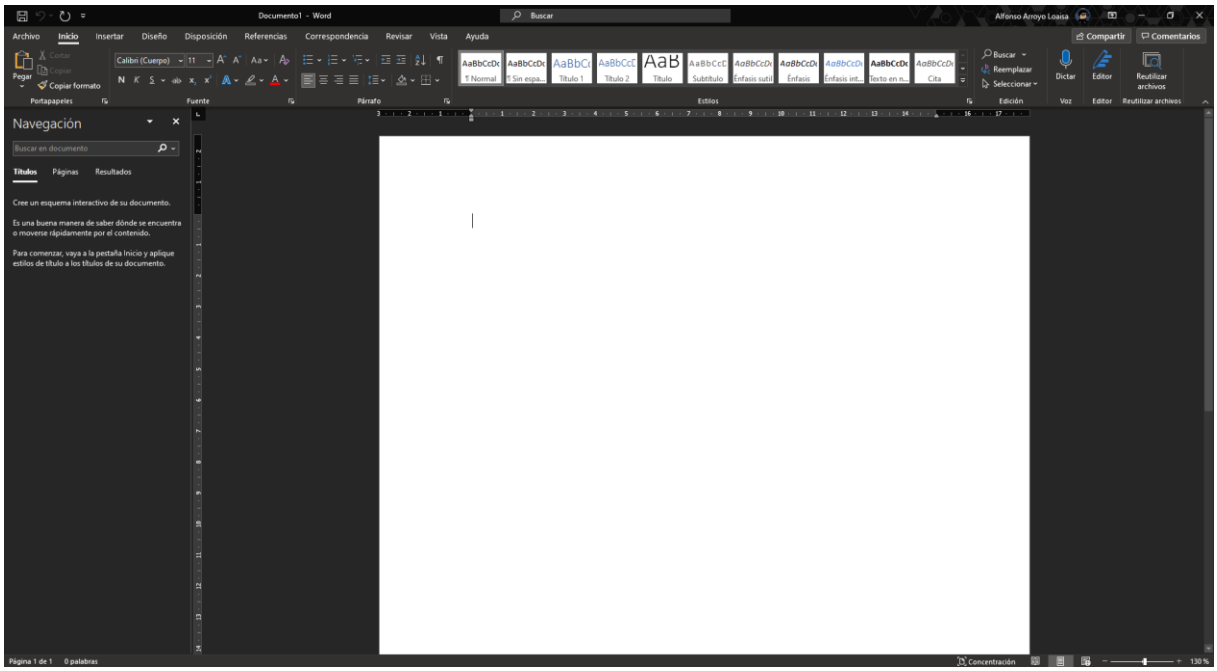


Ilustración 3 Microsoft Office Word

4.4 Adobe Photoshop

Adobe Photoshop es un software de edición de imágenes y fotografías desarrollado por Adobe [7]. Este software será utilizado para crear las imágenes necesarias para el proyecto. Para el proyecto se usará la versión CS5.

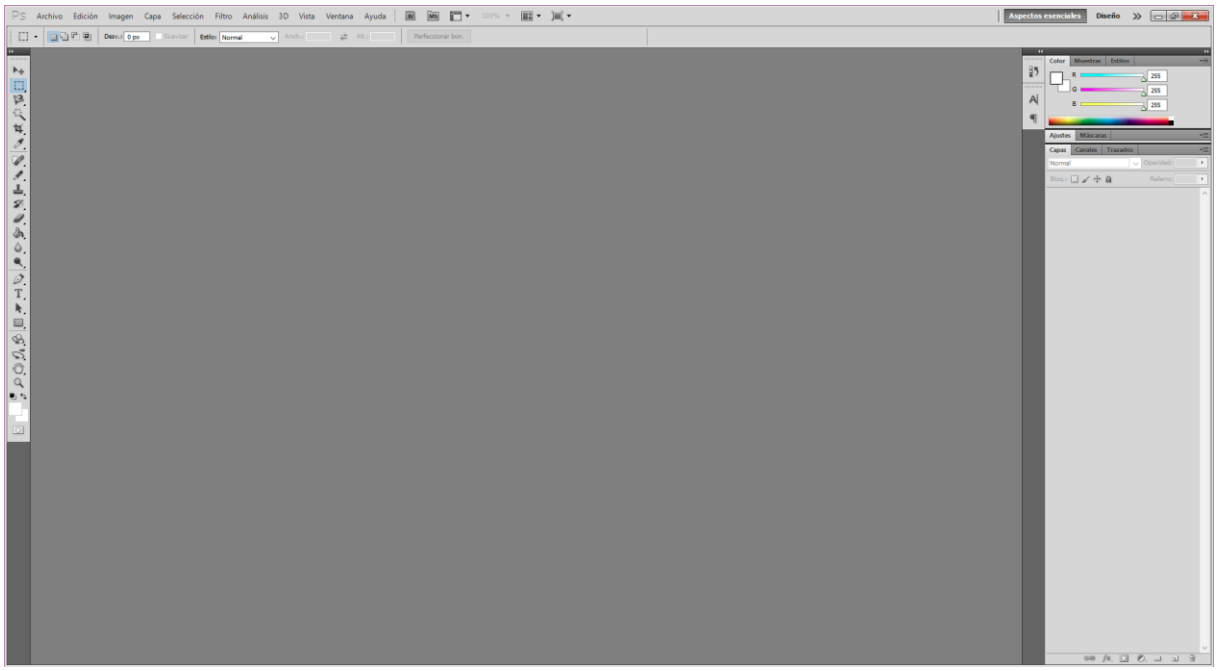


Ilustración 4 Adobe Photoshop CS5

5 PLANIFICACIÓN

5.1 Metodología de desarrollo y planificación temporal

Para el desarrollo de este proyecto se ha utilizado una metodología de tipo incremental, ya que en cada iteración del proyecto se irán incluyendo nuevas funcionalidades, niveles y objetos. Además, se irán mejorando aspectos del juego en base de pruebas que se realizarán durante el desarrollo del proyecto. Se ha usado contenido del libro de Pressman, R. (2010) para todos los procesos de desarrollo y planificación. [18]

Las primeras iteraciones serán las que incluyan el desarrollo las funcionalidades básicas del juego y objetos comunes a todos los niveles; mientras que, las últimas contendrán el desarrollo de los niveles y objetos específicos del nivel, así como el generador aleatorio de niveles y las pruebas y posibles mejoras que se realicen en el proyecto.

El proyecto se ha dividido en diez iteraciones:

1. Desarrollo del nivel de prueba, del personaje principal y la cámara.
2. Desarrollo del controlador del juego, que incluirá todas las funcionalidades del juego, así como los objetos básicos iniciales. También incluirá el desarrollo de la interfaz.
3. Desarrollo del resto de objetos básicos y comunes a todos los niveles. Inicio de la documentación.
4. Desarrollo del primer y segundo nivel. Además, se desarrollará el cambio entre escenas y la escena de *game over*.
5. Desarrollo del tercer y cuarto nivel, y los objetos necesarios para completarlos.
6. Desarrollo del quinto y sexto nivel, y los objetos necesarios para completarlos, junto con el sistema de sonido.
7. Desarrollo del séptimo y octavo nivel, y los objetos necesarios para completarlos.
8. Desarrollo del menú principal, pantalla de inicio y menú de pausa.

9. Desarrollo del generador aleatorio de niveles
10. Mejoras, pruebas y resolución de errores. Finalización de la documentación.

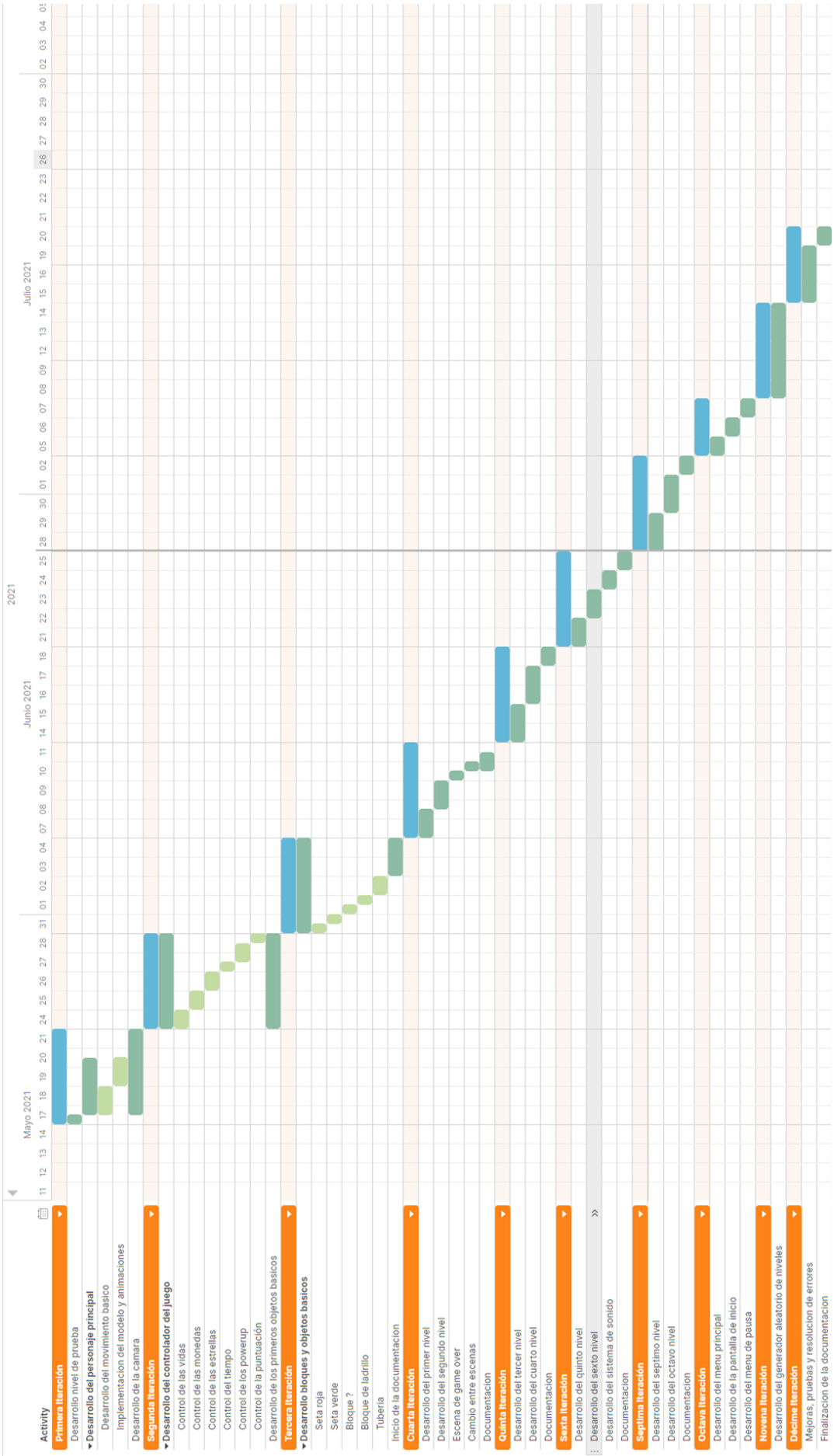

Activity
Primera Iteración
Desarrollo nivel de prueba
Desarrollo del personaje principal
Desarrollo del movimiento basico
Implementacion del modelo y animaciones
Desarrollo de la camara
Segunda Iteración
Desarrollo del controlador del juego
Control de las vidas
Control de las monedas
Control de las estrellas
Control del tiempo
Control de los powerup
Control de la puntuación
Desarrollo de los primeros objetos basicos
Tercera Iteración
Desarrollo bloques y objetos basicos
Seta roja
Seta verde
Bloque ?
Bloque de ladrillo
Tuberia
Inicio de la documentacion
Cuarta Iteración
Desarrollo del primer nivel
Desarrollo del segundo nivel
Escena de game over
Cambio entre escenas
Documentacion
Quinta Iteración
Desarrollo del tercer nivel
Desarrollo del cuarto nivel
Documentacion
Sexta Iteración
Desarrollo del quinto nivel
Desarrollo del sexto nivel
Desarrollo del sistema de sonido
Documentacion
Septima Iteración
Desarrollo del septimo nivel
Desarrollo del octavo nivel
Documentacion
Octava Iteración
Desarrollo del menu principal
Desarrollo de la pantalla de inicio
Desarrollo del menu de pausa
Novena Iteración
Desarrollo del generador aleatorio de niveles
Décima Iteración
Mejoras, pruebas y resolucion de errores
Finalizacion de la documentacion
Mayo 2021
Junio 2021
Julio 2021
2021

Ilustración 5 Diagrama de Gantt

5.2 Estimación de recursos y costes

Para hacer una estimación de los costes del desarrollo se tendrán en cuenta todos los recursos empleados. En estos se incluyen los recursos software, recursos humanos y recursos materiales.

Dentro de los costes de los recursos software encontramos Microsoft Office Word, el cual debe adquirirse como parte del paquete Microsoft 365. Este tiene un coste de 7 € al mes en su modalidad personal, por lo que durante el desarrollo tendrá un coste de 21 €, ya que el proyecto será realizado por una sola persona.

Adobe Photoshop puede ser adquirido por un precio de 24,19 € al mes en su modalidad individual, por lo que tendrá un coste total de 72,57 €.

Unity no tendrá ningún coste hasta que no comercialicemos el juego y superemos los 100.000 $ en ingresos.

El resto de software utilizado es gratuito y no tiene ningún coste.

Software	Coste
Unity	0,00 €
Microsoft Visual Studio	0,00 €
Microsoft Office Word	21,00 €
Adobe Photoshop	72,57 €
Coste total:	93,57 €

Tabla 1 Costes software

Se tienen en cuenta, también, los recursos materiales que en este caso serán el ordenador y periféricos usados para completar el proyecto. Estos tienen un coste aproximado de 1.500 €, amortizados durante cinco años.

Materiales	Coste
Ordenador	70,00 €
Periféricos	5,00 €
Coste total:	75,00 €

Tabla 2 Coste de recursos materiales

Por último, se tendrá en cuenta los recursos humanos empleados. En este caso, el proyecto estará desarrollado por una única persona, que asumirá todos los roles existentes en el proyecto.

Software	Salario anual	Salario semanal	N.º semanas	Total
Jefe de proyecto	16.342,06 €	314,27 €	10	3.142,70 €
Programador	15.442,56 €	296,97 €	10	2.969,70 €
Diseñador	10.965,64 €	210,87 €	5	1054,35 €
			Coste total:	7.166,75 €

Tabla 3 Coste de recursos humanos

Para el cálculo de los salarios se ha tenido en cuenta que un año este compuesto por 52 semanas. Los salarios anuales han sido extraídos del Boletín Oficial del Estado del XVII Convenio colectivo estatal de empresas de consultoría, y estudios de mercados y de la opinión pública, del 22 de febrero de 2018 [12].

Recurso	Coste
Recursos software	93,57 €
Recursos materiales	75,00 €
Recursos humanos	7.166,75 €
Coste total:	7.335,32 €

Tabla 4 Coste total del proyecto

El presupuesto inicial necesario para desarrollar este proyecto asciende a 7.335,32 €.

6 DISEÑO INICIAL

6.1 Análisis y diseño del sistema

6.1.1 Diseño de la interfaz de usuario

Una de las partes más importantes a la hora de desarrollar un videojuego es el diseño de su interfaz, ya que es el medio por el cual el usuario puede comunicarse con el juego y obtener información de él.

Una buena interfaz debe ser simple e intuitiva, y ser consistente con el conocimiento previo del usuario. También deberá encajar correctamente con el resto del juego y su estética.

Además, debemos tener en cuenta el dispositivo donde será visualizada la interfaz, que en este caso será en un ordenador. Por defecto, consideraremos que la interfaz se visualizará en una pantalla de 16:9.

Se han creado los bocetos correspondientes a cada una de las vistas de la interfaz para tener una referencia a la hora de empezar con su creación final. Tanto los bocetos como las vistas finales serán expuestas en este documento.

6.1.1.1 Workflow

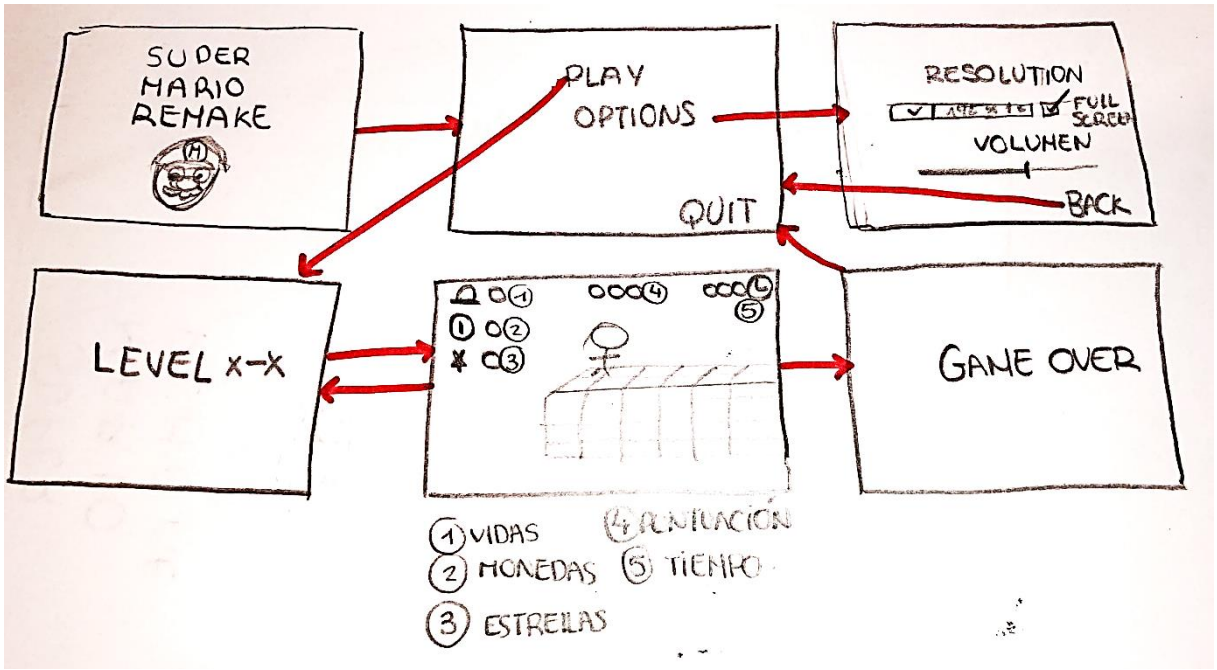


Ilustración 6 Workflow

6.1.1.2 Pantalla de inicio

La pantalla de inicio será la primera pantalla que vea el usuario nada más abrir el juego. Contendrá el título del videojuego y una imagen del juego.

Ilustración 7 Boceto de la pantalla de inicio

6.1.1.3 Menú principal

El menú principal se presentará después de la pantalla de inicio. En él habrá tres botones los cuales permitirán empezar el juego, ir a la pantalla de opciones o salir del juego.

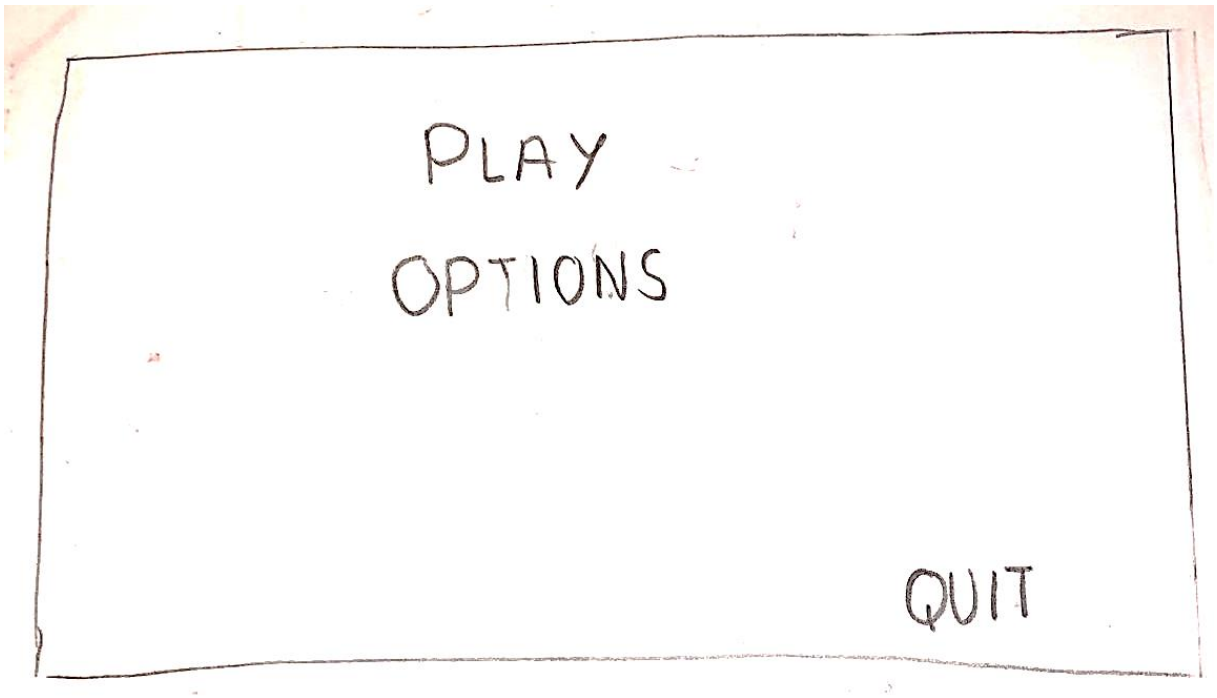


Ilustración 8 Boceto del menú principal

6.1.1.4 Menú de opciones

En el menú de opciones se tendrá la posibilidad de cambiar la resolución del juego y de cambiar entre pantalla completa o ventana.

También se podrá ajustar el volumen del juego.

Por último, también se presentará un botón para volver al menú principal una vez se haya terminado de ajustar las opciones.

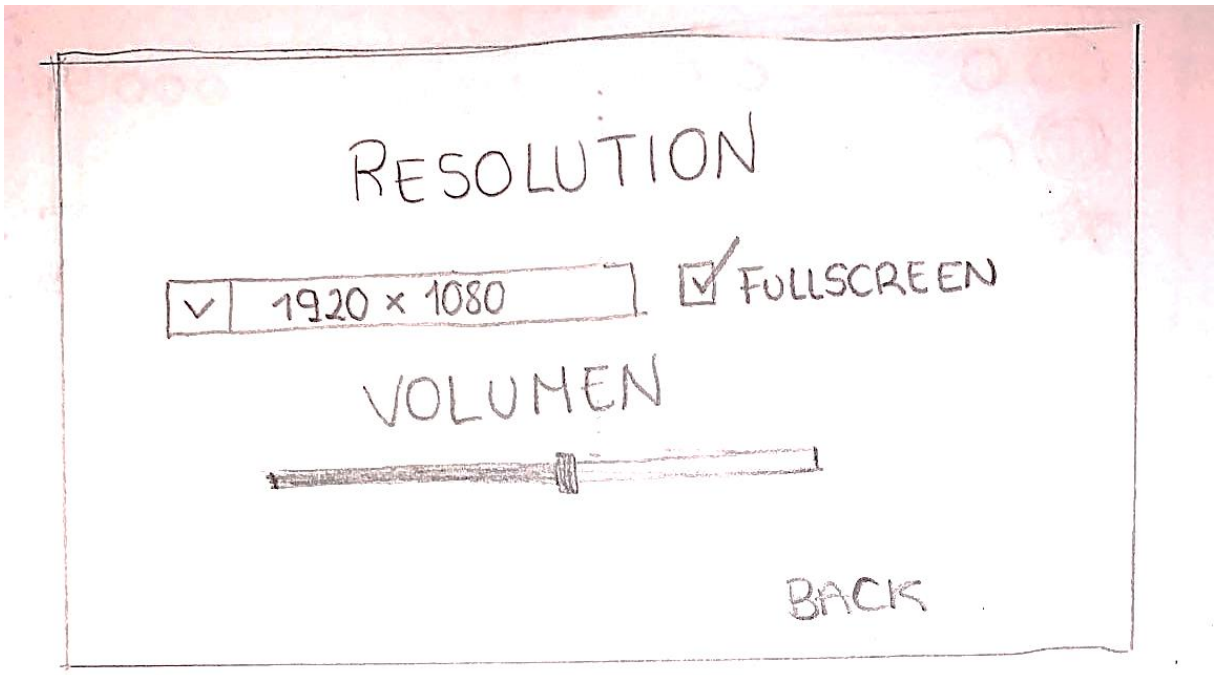


Ilustración 9 Boceto de pantalla de opciones

6.1.1.5 Pantalla de carga del nivel

Esta pantalla se mostrará cada vez que se cambie de nivel o cada vez que este se reinicie. Se sustituirán las dos X del boceto por el mundo y nivel respectivamente.

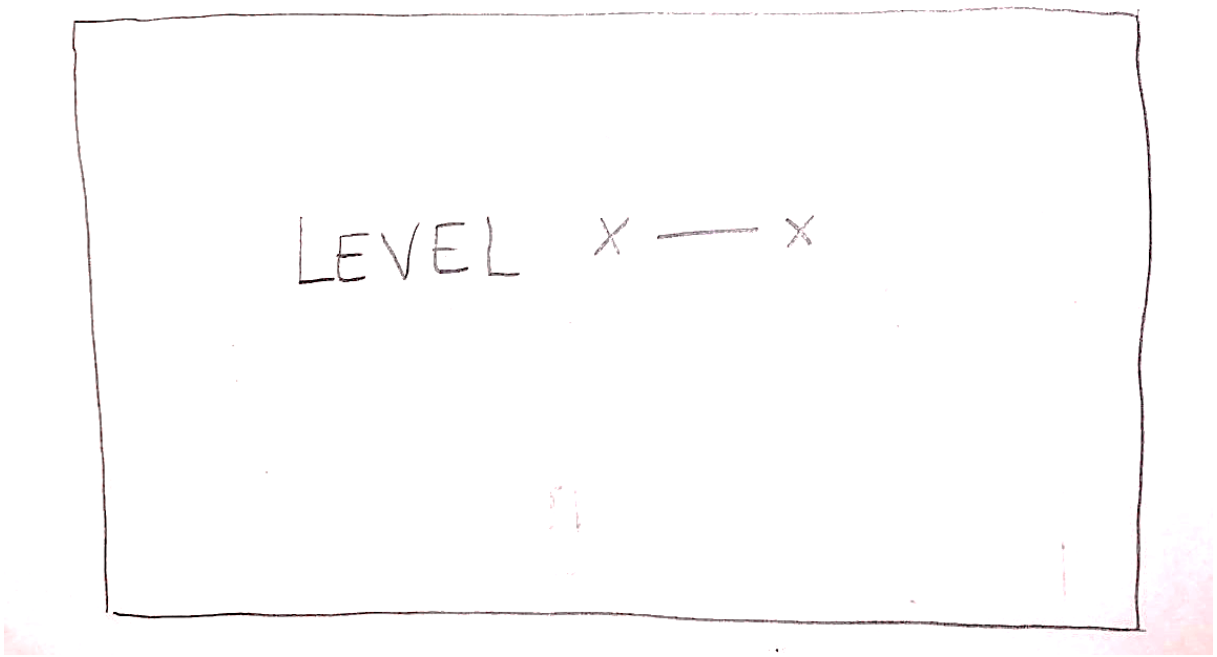


Ilustración 10 Boceto de la pantalla de carga

6.1.1.6 Interfaz dentro del juego

La interfaz dentro del juego mostrará al usuario toda la información sobre el estado actual de su partida. En la parte superior izquierda, de arriba hacia abajo, se mostrarán las vidas, las monedas y las estrellas recogidas por el jugador.

En el centro de la pantalla, en la parte superior, se mostrará el contador de puntos del jugador.

Por último, en la parte superior derecha se mostrará el tiempo restante en el nivel.

Cada uno de los contadores, excepto el de puntuación, se mostrarán acompañados de una imagen o icono que represente lo que está indicando el contador.

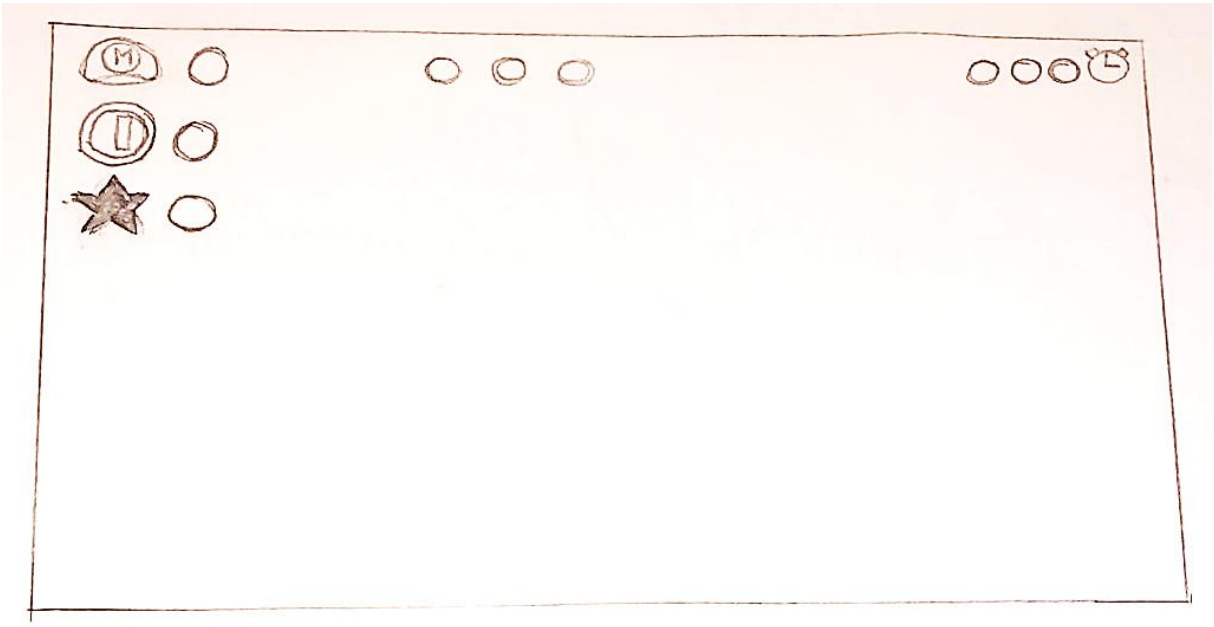


Ilustración 11 Boceto de la interfaz durante el juego

6.1.1.7 Pantalla de juego terminado

Esta será la pantalla que se muestre al jugador una vez que se le han terminado todas las vidas. Será una pantalla con un texto sencillo indicando que la partida ha acabado.

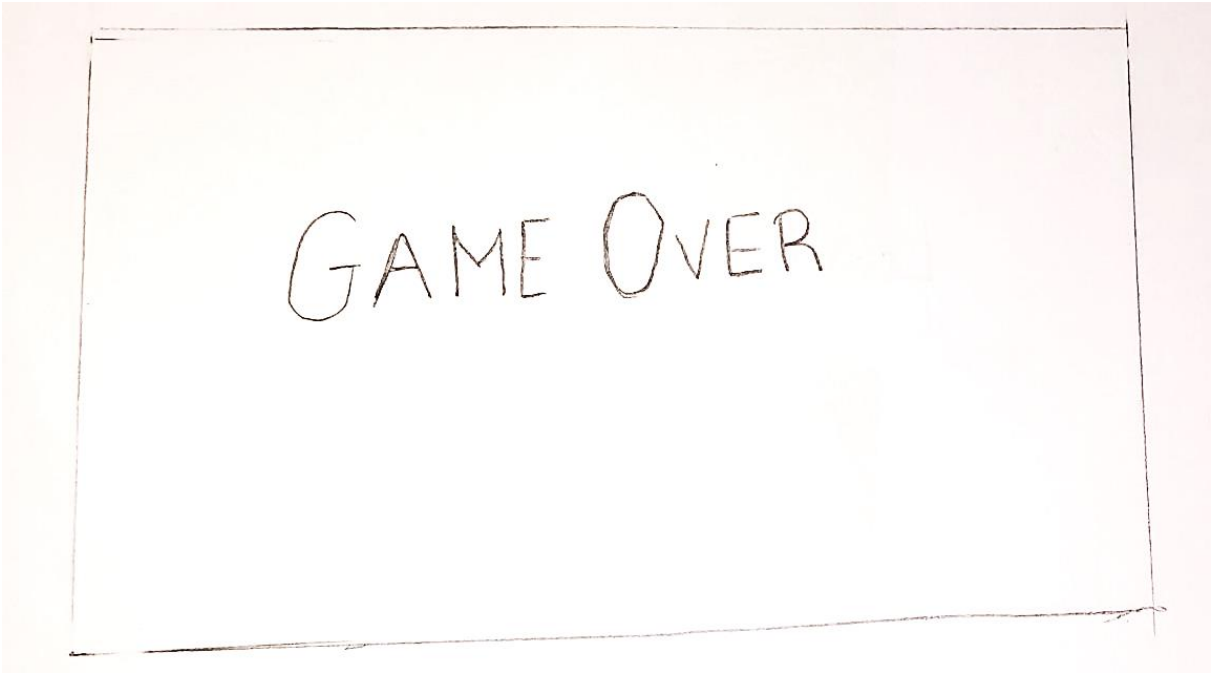


Ilustración 12 ***Boceto de la pantalla de fin de juego***

6.2 Mecánicas de juego

6.2.1 Objetivos del jugador

El objetivo del jugador será obtener la mayor cantidad de puntos, realizando para ello distintas acciones, las cuales aumentarán su marcador. Estas serán realizadas a lo largo de cada uno de los niveles del juego. Todos ellos contarán con distintos enemigos que derrotar, y monedas y estrellas que conseguir que sumará puntos al jugador. Además, este recibirá puntos extra por el tiempo restante en el nivel, por lo que también será premiada la rapidez en la que se supera el nivel.

Los niveles serán jugados en un orden predefinido, excepto para el modo de juego aleatorio, y el jugador tendrá un número de vidas para superarlos. A lo largo de los niveles podrá obtener nuevas vidas.

6.2.2 Criterios de éxito y fracaso

El jugador deberá llegar hasta la meta para completar el nivel, derrotando o esquivando a los enemigos, y evitando caer fuera del nivel. Si el jugador cae fuera del nivel o es tocado por un enemigo, pierde una de sus vidas y deberá comenzar el nivel de nuevo. Si el jugador pierde todas sus vidas, deberá comenzar el juego de nuevo.

6.2.3 Mecánica principal

La mecánica principal del juego será el movimiento lateral del jugador sobre las plataformas, de izquierda a derecha. Además, el jugador también podrá realizar un salto en cualquier momento, así como correr, lo que hará que este se desplace mucho más rápido. Todos estos movimientos se realizarán mediante los botones asignados. Haciendo uso de esta mecánica, el jugador tendrá que ir completando los distintos niveles que, a medida que se progrese en el juego, incluirán nuevas mecánicas que mantengan atento al jugador.

6.2.4 Mundo y personajes

El mundo está ambientando en un reino ficticio, a lo largo del cual encontraremos distintos personajes y enemigos que lo habitan. En este mundo encontraremos también bloques mediante los cuales podremos obtener powerups y distintos objetos que nos ayudarán a lo largo del juego.

6.2.4.1 Personaje principal. Mario.

Mario es el personaje principal y será el que el jugador controle. Este podrá desplazarse por los niveles lateralmente, saltar y correr. Mediante los powerups podrá “crecer”, obteniendo un punto de vida extra antes de morir. Este tendrá como objetivo llegar al último nivel, intentando rescatar a la princesa que ha sido secuestrada por el villano del reino.

Ilustración 13 Mario

6.2.4.2 Goomba

Los Goomba son uno de los enemigos más básicos del juego. Estos se desplazarán a lo largo del nivel e intentarán acabar con Mario. Para derrotarlos habrá que saltar sobre ellos.

Ilustración 14 Goomba

6.2.4.3 Planta piraña

Las plantas piraña son un enemigo de Mario que podremos encontrar dentro de algunas tuberías a lo largo del juego. Estas intentarán morder a Mario cuando este pase salte sobre ellas. No podrán ser derrotadas de ninguna manera, solo podrán ser esquivadas.

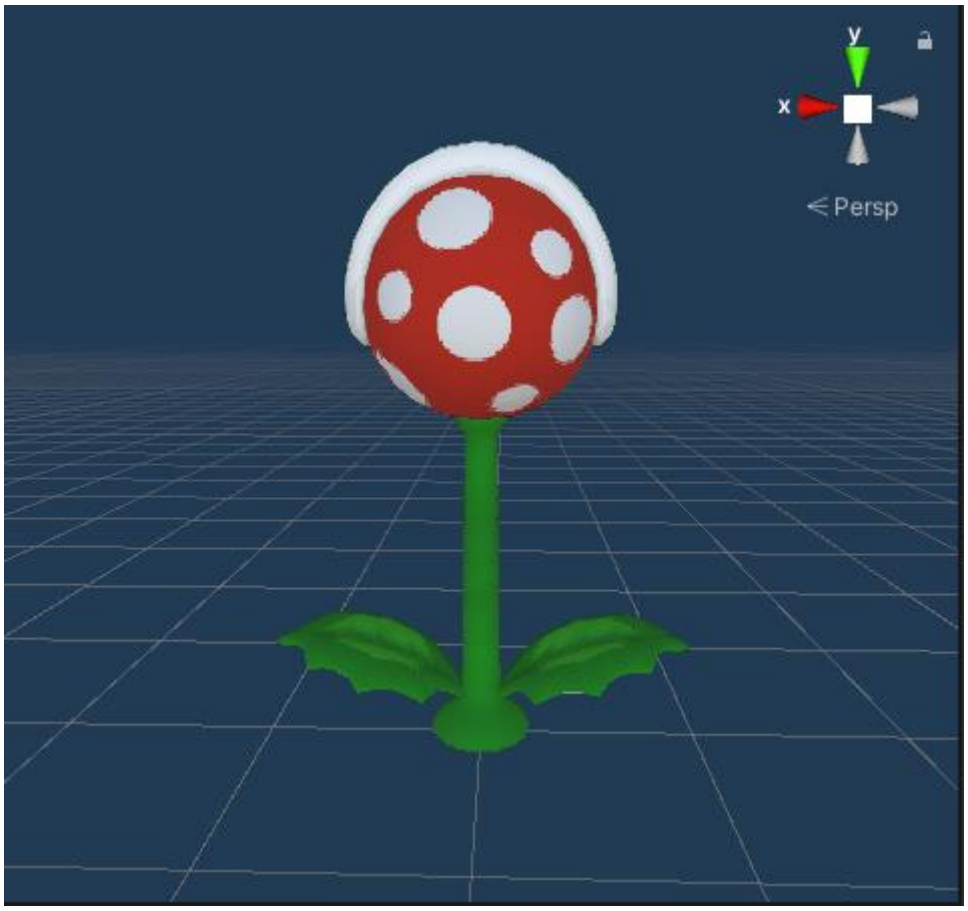


Ilustración 15 Planta piraña

6.2.4.4 Koopa

Los Koopa son otra de las criaturas presentes en este mundo. Al igual que las dos anteriores, intentarán acabar con Mario. Estas se desplazarán lateralmente por todo el nivel, y podrán ser derrotadas si Mario salta sobre ellas. Una vez que sean derrotadas su caparazón quedará sobre el suelo, y podrá ser lanzado por Mario en cualquier dirección.

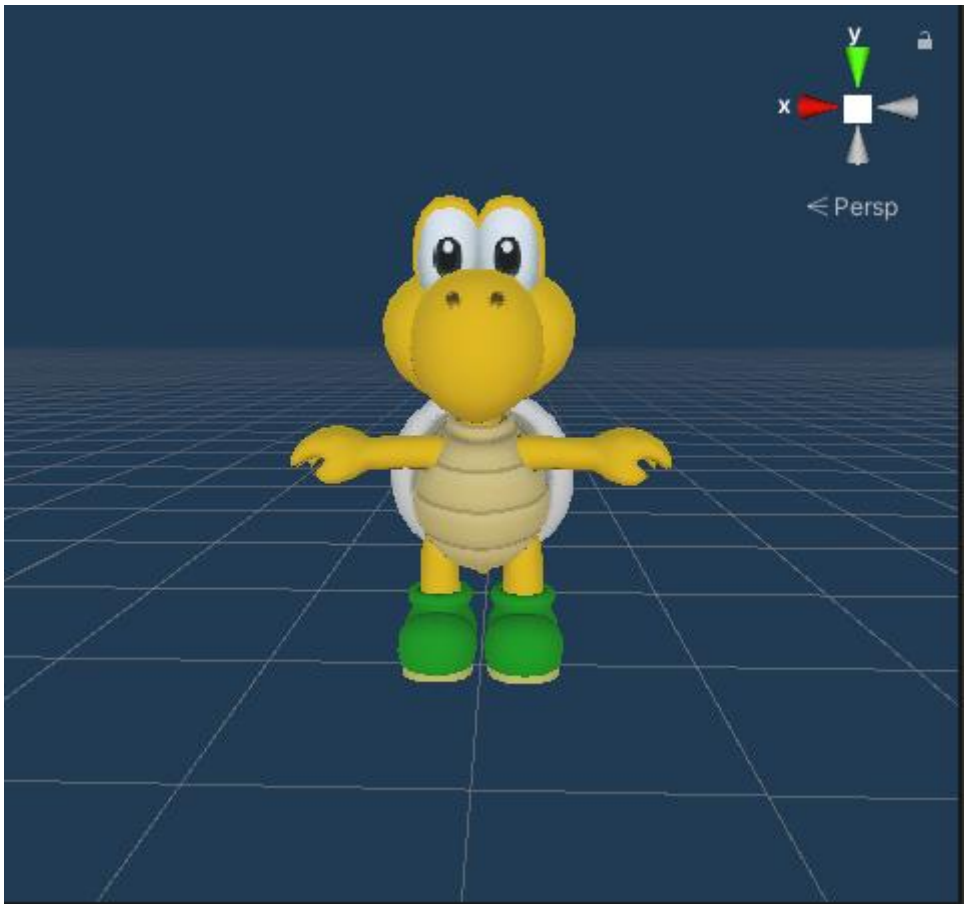


Ilustración 16 Koopa

6.2.4.5 Spiny

Los Spiny serán otro de los enemigos que encontraremos a lo largo del juego. Estos podrán desplazarse por el nivel, y Mario no podrá acabar con ellos de ninguna manera, a no ser que sean golpeados por un caparazón de un Koopa.

Ilustración 17 Spiny

6.2.4.6 Cheep Cheep

Los Cheep Cheep son unos enemigos que habitan en el agua. Estos saltarán fuera de ella, realizando un salto en parábola, para intentar acabar con Mario. Podremos encontrarlos en los niveles en los que haya agua bajo el nivel.

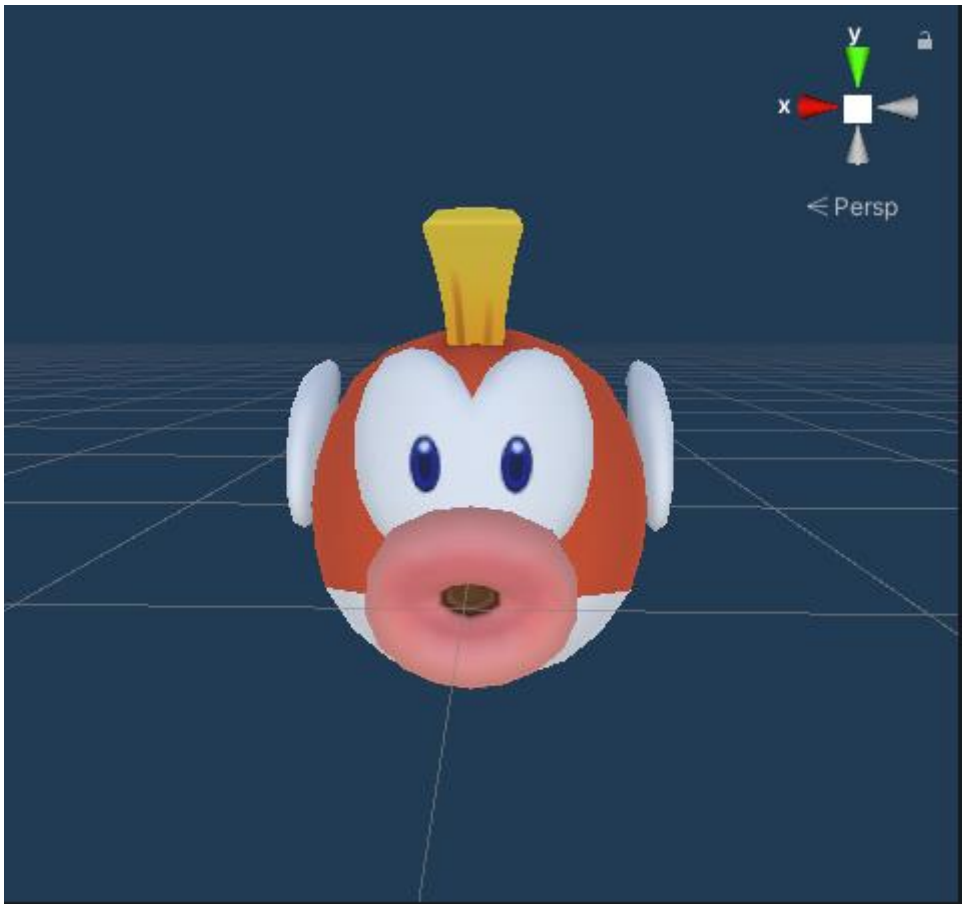


Ilustración 18 Cheep Cheep

6.2.4.7 Bowser Jr

Bowser Jr es el villano del reino y este estará situado en cada uno de los niveles castillo que encontremos a lo largo del juego, y será el jefe final de ellos. Este podrá saltar y acabar con nosotros. Para derrotarlo el jugador deberá esquivarlo y acabar con el mediante un botón que estará tras él, el cual abrirá el puente sobre el que está situado.

Ilustración 19 Bowser Jr

6.2.4.8 Toad

Los Toad son los sirvientes de la princesa que ha sido secuestrada, y los cuales nos esperarán al final de los niveles castillo, y solo nos informarán de que aun tenemos que seguir buscando en los próximos niveles.

Ilustración** 20 **Toad

6.3 Diseño de niveles

Para el diseño de los niveles del juego se han utilizado los diseños originales del juego Super Mario Bros original, para la plataforma Nintendo Entertainment System, NES, como base. A estos se le han introducido algunos cambios respecto de los originales. Los diseños originales de los niveles han sido obtenidos de NES Maps [8].

6.3.1 Nivel 1 (1-1)

El nivel 1 será el nivel inicial y más sencillo de todos, fácil de superar, para que el jugador se familiarice con los controles y funcionamiento del juego. En este aparecen por primera vez los Goomba y objetos básicos como los bloques de ladrillo y “?”, y setas rojas. Estará ambientado en el exterior.

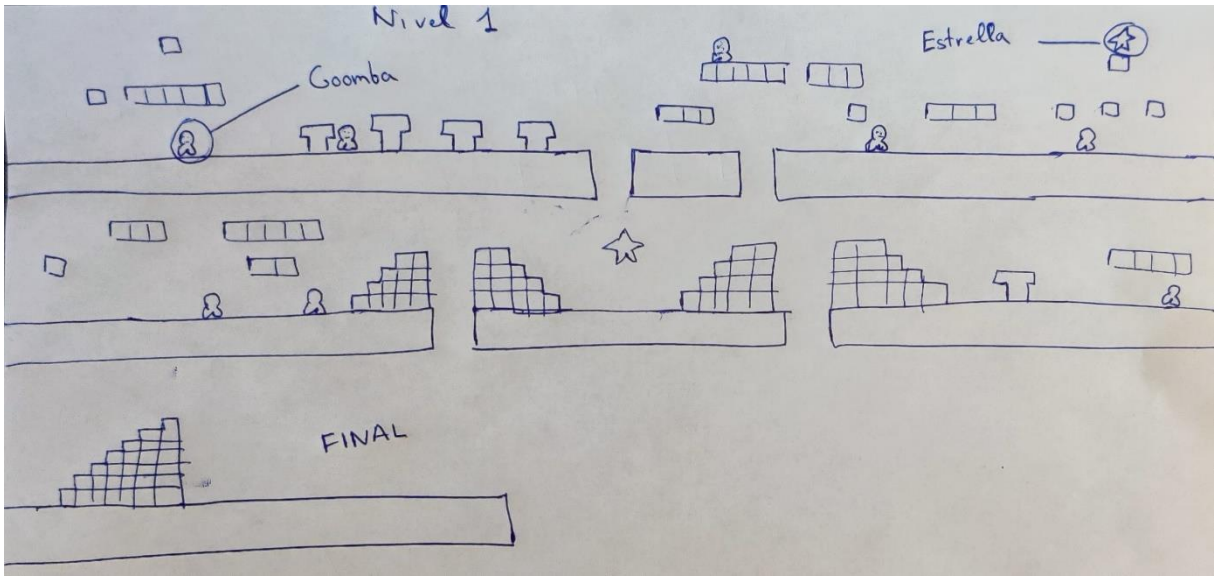


Ilustración 21 Boceto del nivel 1

Este nivel está basado en el nivel 1-1 del videojuego original.

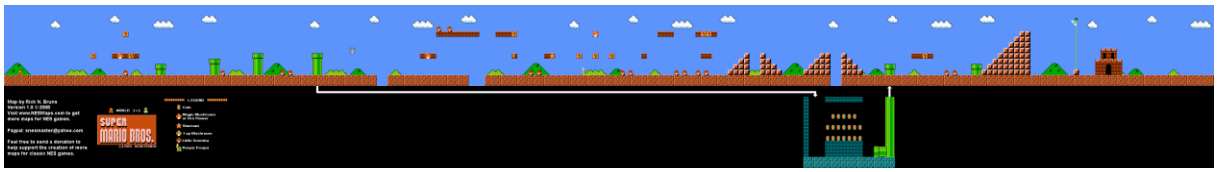


Ilustración 22 Nivel 1-1 original

6.3.2 Nivel 2 (1-2)

El nivel 2, además de incluir los Goomba, incluirá también plantas piranas, que estarán en el interior de las tuberías. En este nivel se aumenta la dificultad respecto del primero. También será la primera vez que haya monedas a lo largo del nivel.

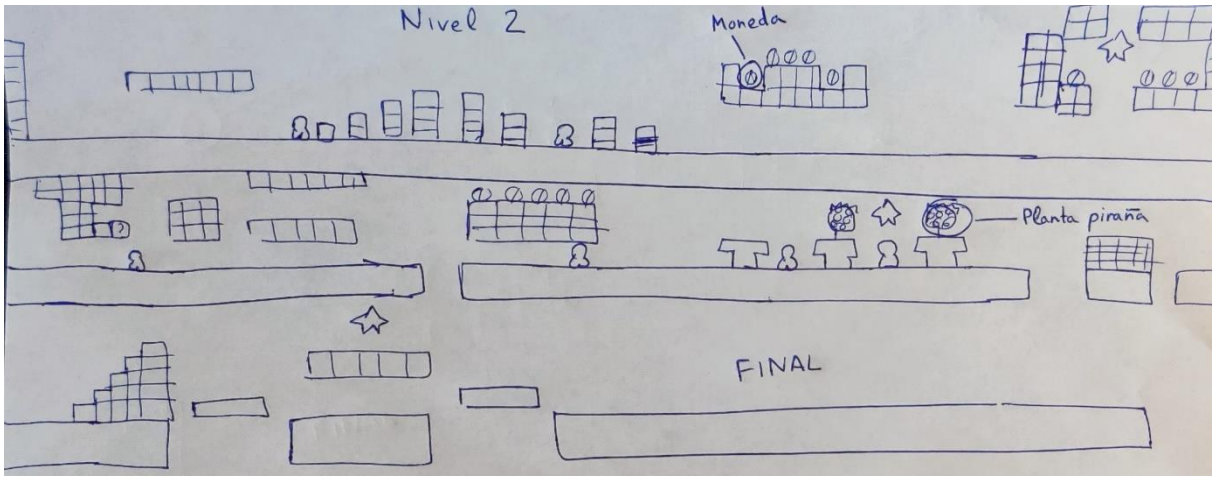


Ilustración 23 Boceto del nivel 2

Este nivel está basado en el nivel 1-2 del videojuego original.

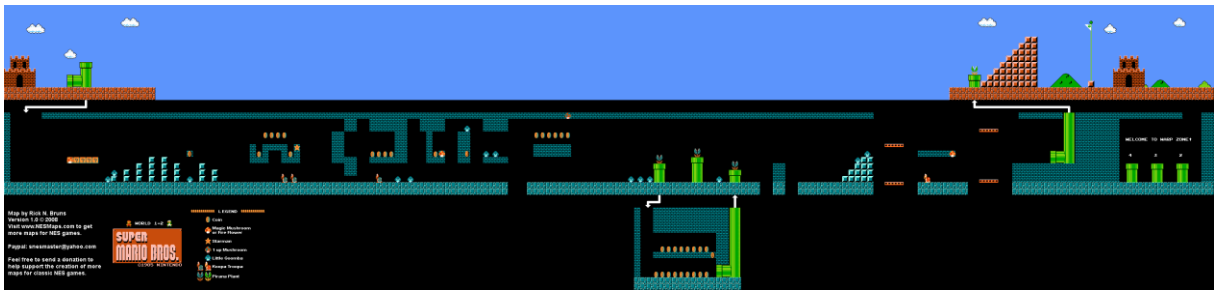


Ilustración 24 Nivel 1-2 original

6.3.3 Nivel 3 (1-3)

El nivel 3 será un nivel que aumente la dificultad respecto de los anteriores al tener muchos más sitios por los que el jugador pueda precipitarse al vacío. Además, en este nivel se introducen los Koopa, un nuevo enemigo.

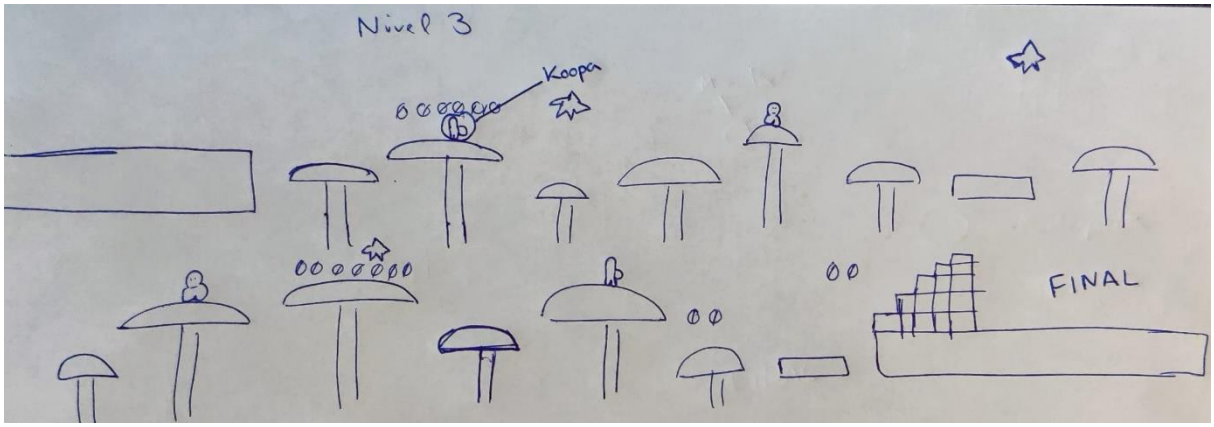


Ilustración 25 Boceto del nivel 3

Este nivel está basado en el nivel 1-3 del videojuego original.

Ilustración 26 Nivel 1-3 original

6.3.4 Nivel 4 (1-4)

Este nivel será el primer nivel final que encontremos en el juego. En el se introduce un nuevo tipo de obstáculo, las barras de fuego giratorias, las cuales podrán eliminar el jugador. Además, al final de este encontraremos a Bowser Jr como jefe final, al cual podremos derrotar mediante un botón. Finalmente, encontraremos un Toad en el final, una vez superado el nivel.

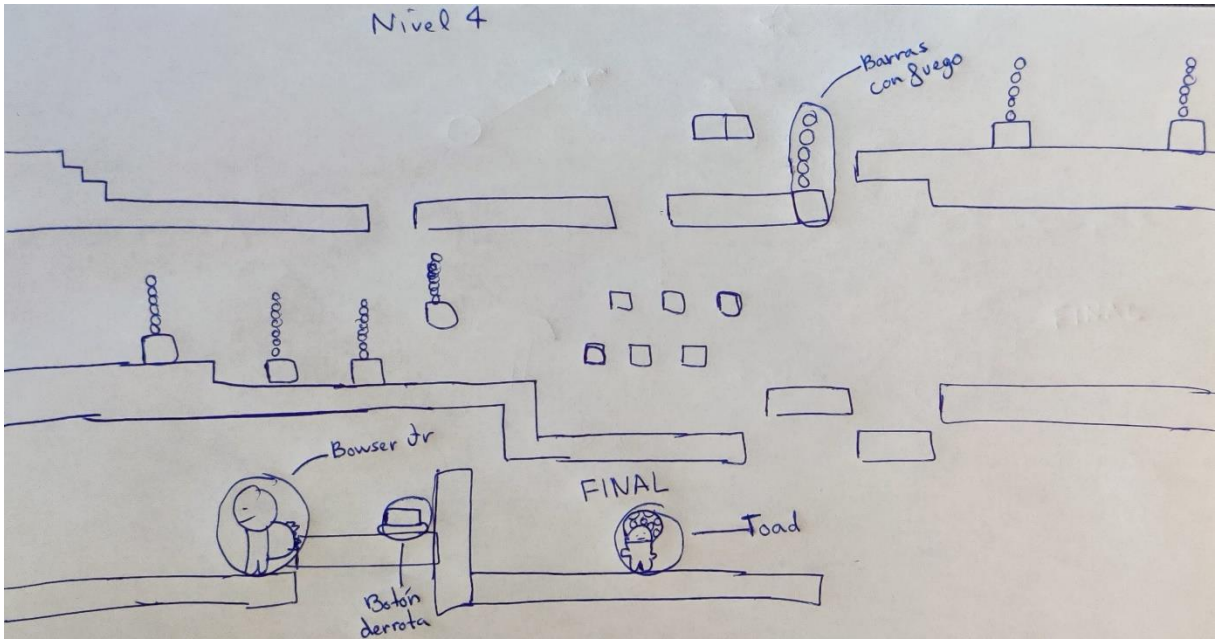


Ilustración 27 Boceto del nivel 4

Este nivel está basado en el nivel 1-4 del videojuego original.

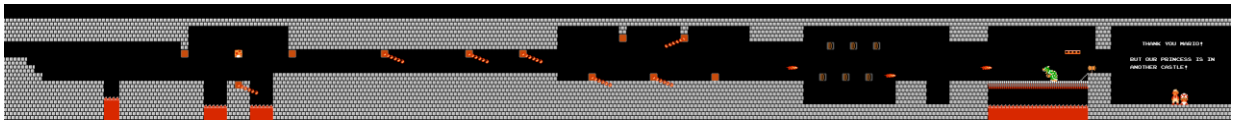


Ilustración 28 Nivel 1-4 original

6.3.5 Nivel 5 (2-1)

El nivel 5 será el nivel con el que inicia el mundo 2. En este encontraremos elementos de los niveles pasados y, como novedad, presenta una zona en la parte

superior de la segunda mitad del nivel, a la cual el jugador podrá acceder con un poco de habilidad y exploracion, y en la que podrá obtener monedas extra.

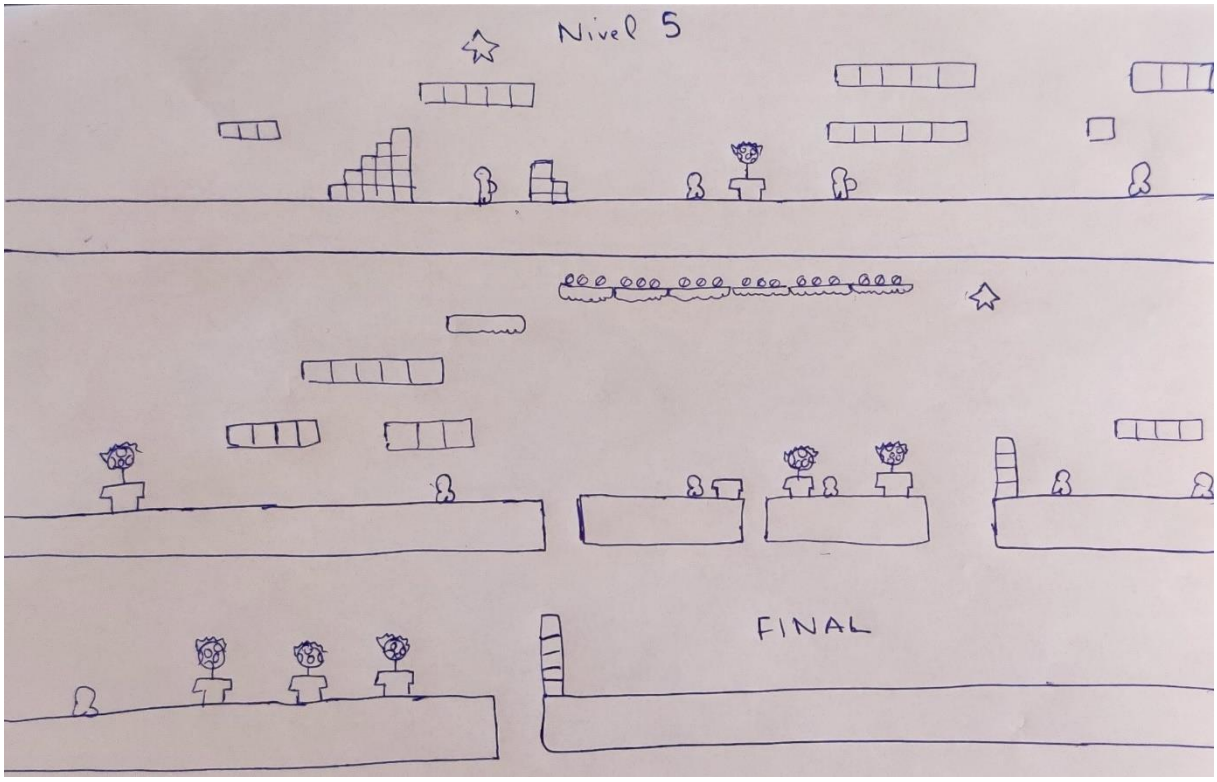


Ilustración 29 Boceto del nivel 5

Este nivel está basado en el nivel 2-1 del videojuego original.

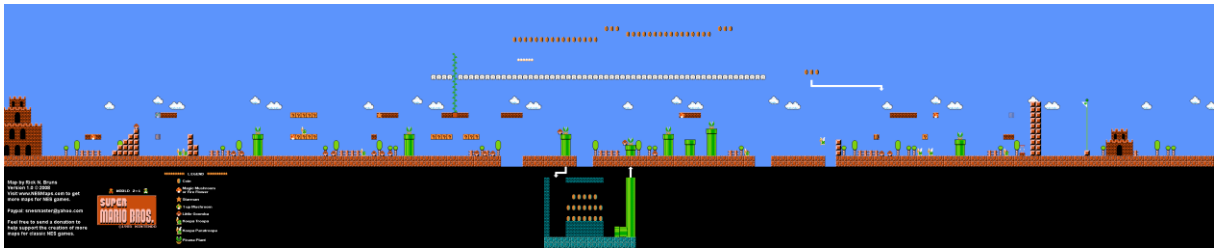


Ilustración 30 Nivel 2-1 original

6.3.6 Nivel 6 (2-2)

En el nivel 6 se introduce un enemigo nuevo, los Spiny, que estarán a lo largo de todo el nivel. En este continúan usándose los recursos de los niveles anteriores.

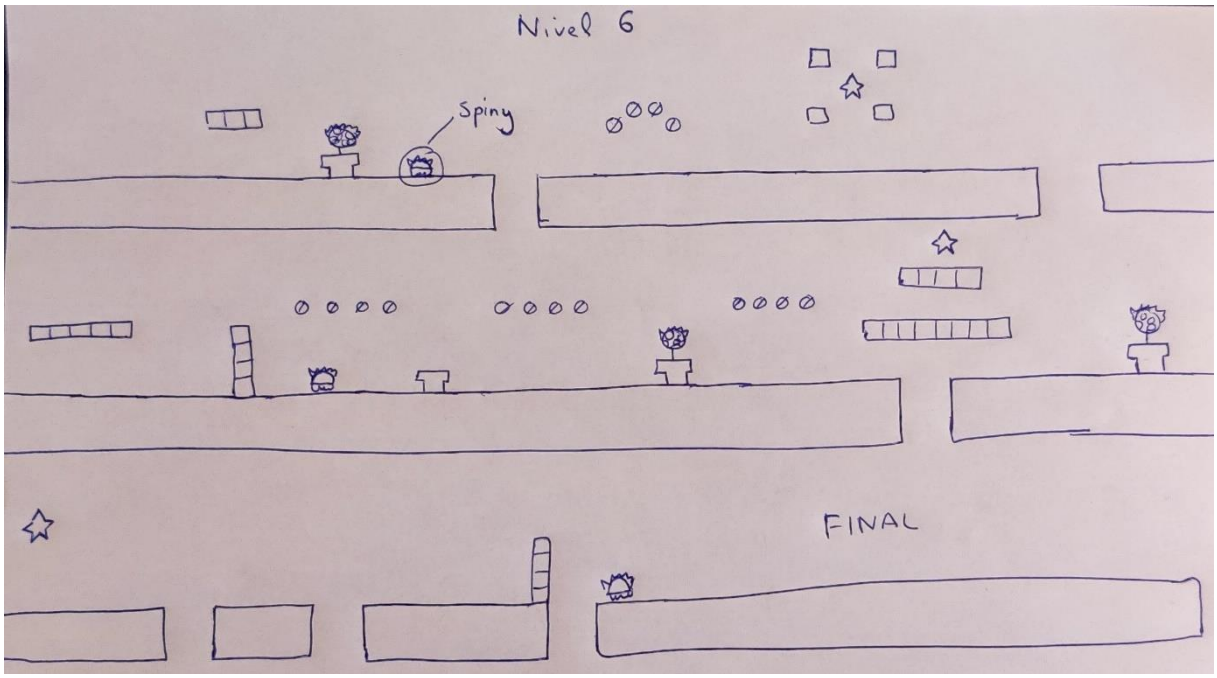


Ilustración 31 Boceto del nivel 6

Este nivel está basado en el nivel 4-1 del videojuego original.

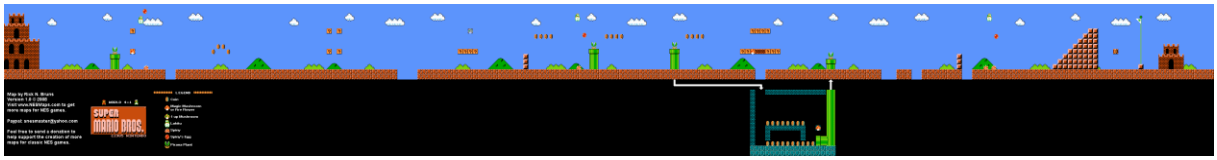


Ilustración 32 Nivel 4-1 original

6.3.7 Nivel 7 (2-3)

El nivel 7 estará formado por una serie de plataformas que nos ayudaran a superar el nivel. Además, en la segunda mitad del nivel se incluyen unas plataformas las cuales tendrán un movimiento hacia arriba y hacia abajo. También, en este nivel aparecen los Cheep Cheep.

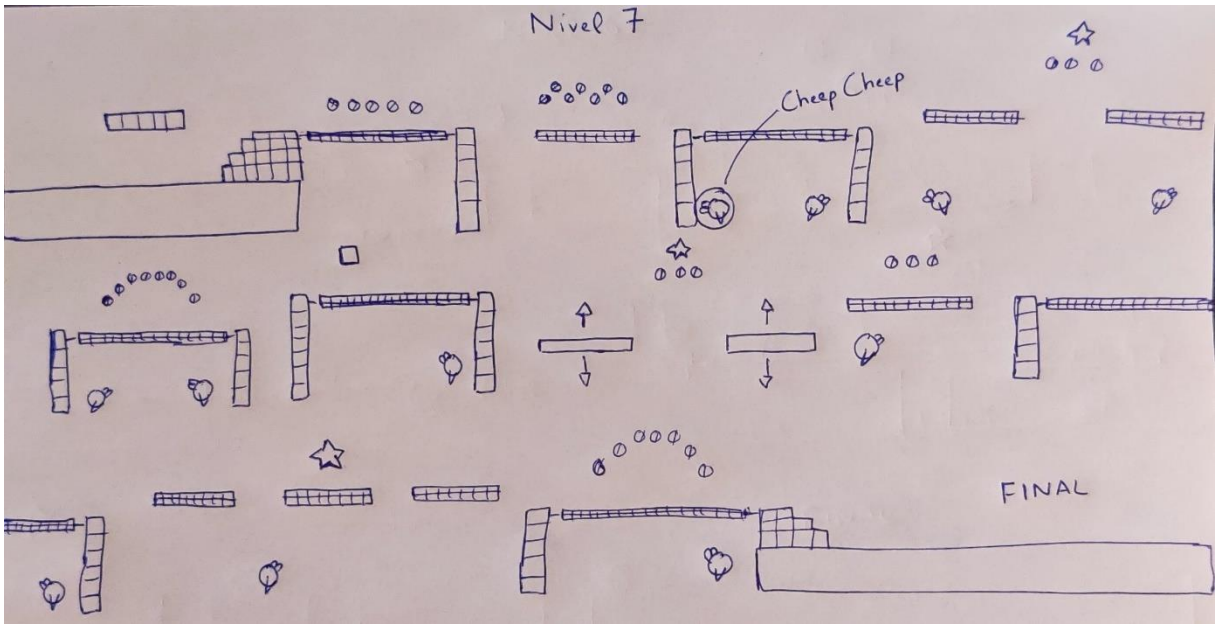


Ilustración 33 Boceto del nivel 7

Este nivel está basado en el nivel 2-3 del videojuego original.

Ilustración 34 Nivel 2-3 original

6.3.8 Nivel 8 (2-4)

El nivel 8 y último incluirá como novedad unas bolas de fuego al inicio del nivel las cuales se elevarán cuando el jugador pase por encima de ellas. Además, en la segunda mitad del nivel se incluyen unas plataformas las cuales bajaran o subirán y, una vez lleguen a su punto máximo o mínimo, se reiniciarán para volver a su punto inicial.

Al final, al igual que en el nivel 4, se encuentra como jefe final Bowser Jr, el cual podrá ser derrotado mediante un botón. Una vez pulsado accederemos a una sala donde estará Toad.

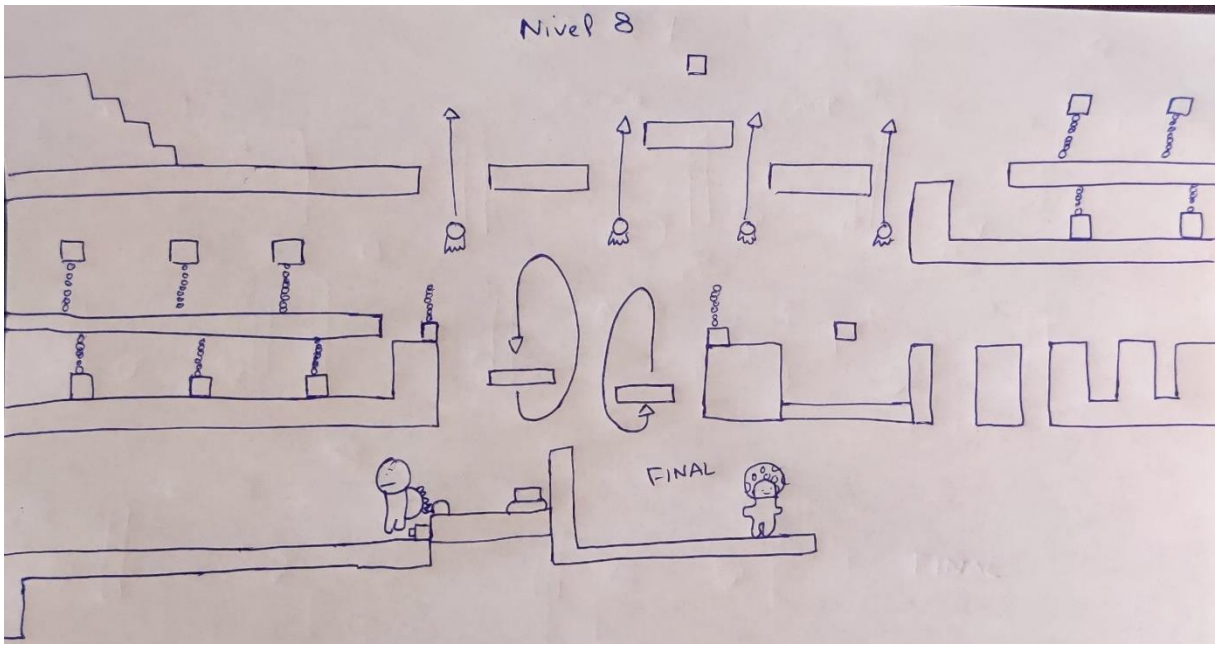


Ilustración 35 Boceto del nivel 8

Este nivel está basado en el nivel 2-4 del videojuego original.

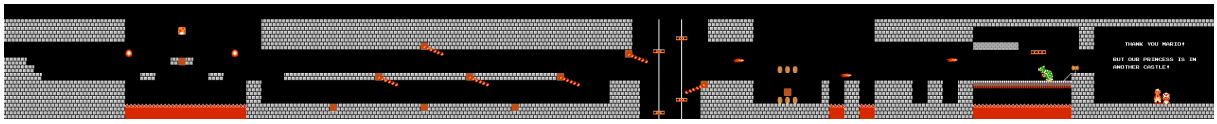


Ilustración 36 *Nivel 2-4 original*

7 DESARROLLO DEL PROYECTO

7.1 Primera iteración

Durante esta iteración se ha llevado a cabo el desarrollo de un nivel de prueba básico para pruebas posteriores. También se ha llevado a cabo el desarrollo del personaje principal y de la cámara.

7.1.1 Nivel de pruebas

El nivel de prueba será una de las escenas más importantes durante el desarrollo, ya que en ella se desarrollarán y probarán todas las funcionalidades antes de ser implementadas en el nivel correspondiente.

En esta iteración, el nivel solo incluirá algunas estructuras básicas donde probar el movimiento del jugador durante su desarrollo. Para ello se han colocado una serie

de cubos simulando una escalera y una rampa. También se podrá encontrar una serie de plataformas elevadas, todo ello sobre un plano.

Para las estructuras se ha creado un material de color blanco, mientras que, para el suelo se ha creado y aplicado un material en color verde.

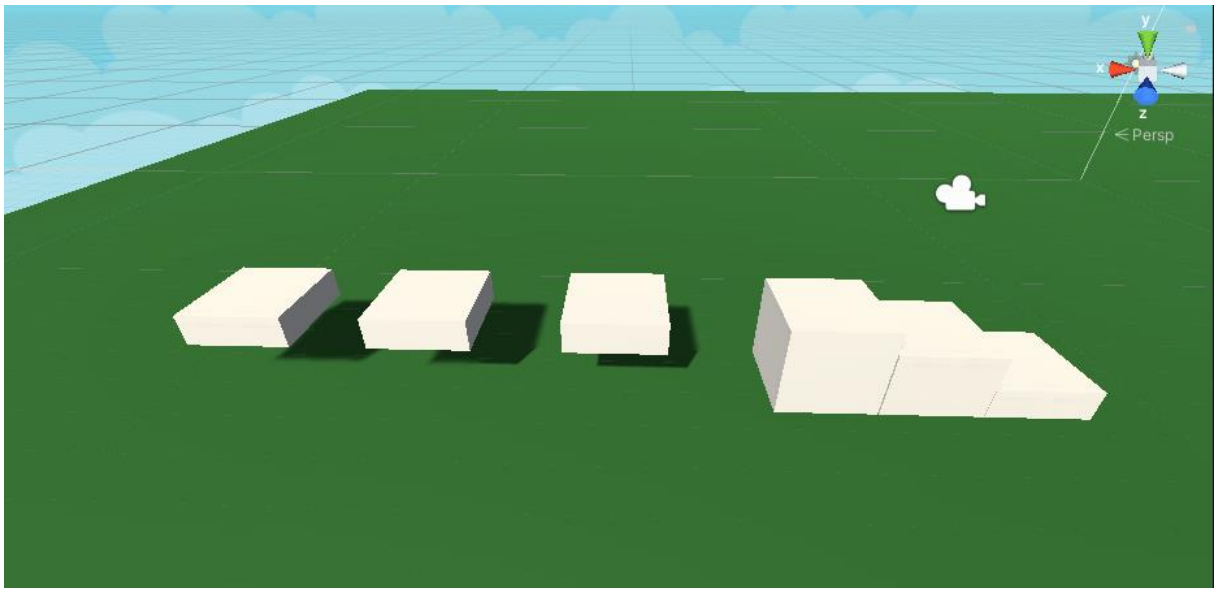


Ilustración 37 Estructuras básicas en la escena de prueba

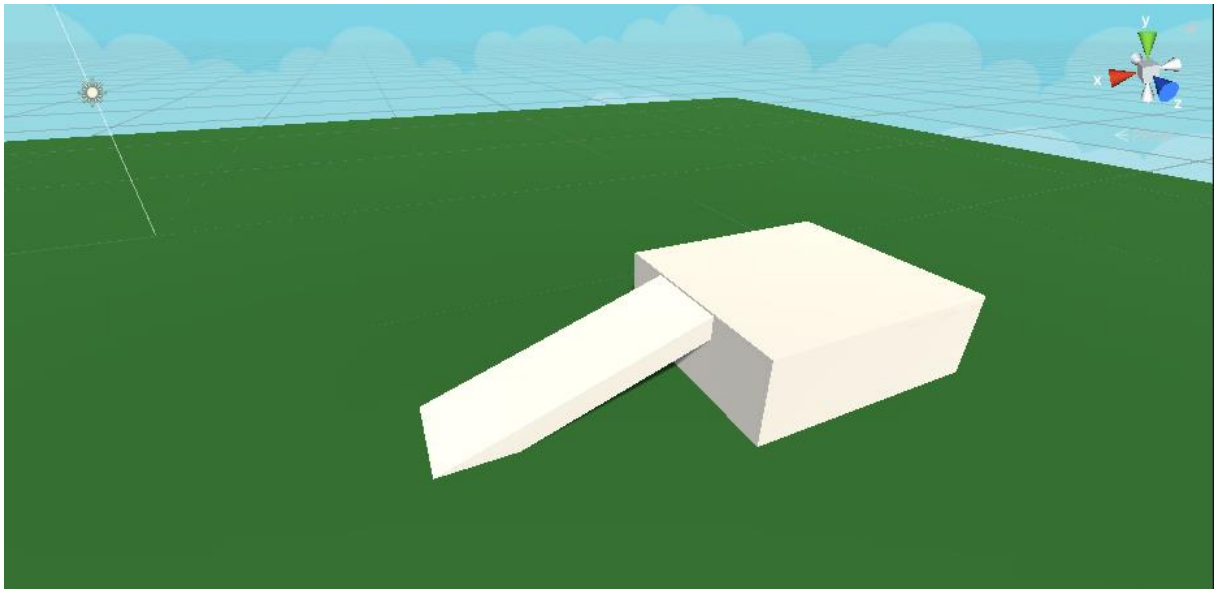


Ilustración 38 Estructuras básicas en la escena de prueba

Mediante estas se probará el controlador del jugador simulando posibles estructuras que podrían estar en los futuros niveles que se desarrollen.

A esta escena también se le ha añadido un material al *skybox* obtenido del paquete Farland Skies - Cloudy Crown [9], descargado desde la Asset Store.

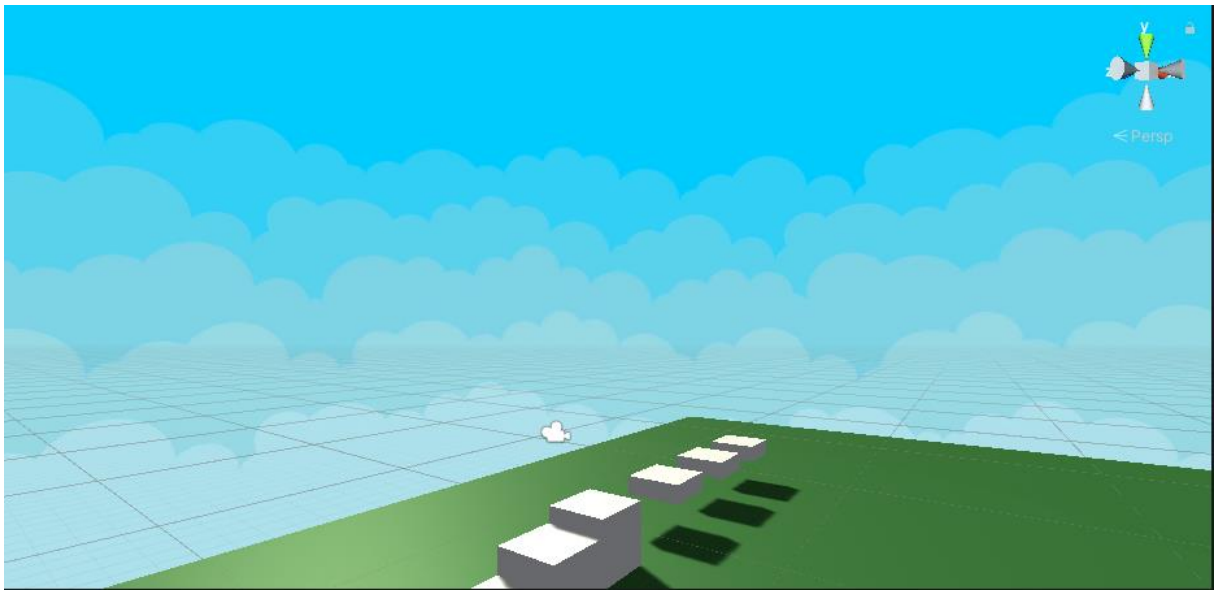


Ilustración 39 Skybox

7.1.2 Personaje principal

El personaje principal y su controlador será unas de las partes más importantes del proyecto, ya que el jugador debe sentir en todo momento que tiene el control del personaje. Durante el desarrollo del personaje, se ha realizado de forma paralela el desarrollo de la cámara, que será explicado en el siguiente apartado.

Al comienzo del desarrollo se ha creado un modelo básico para realizar las pruebas de movimiento sobre él. Este estará formado por una cápsula a la que se le ha aplicado un material en color morado y un cubo que representará la dirección en la que mira el personaje.

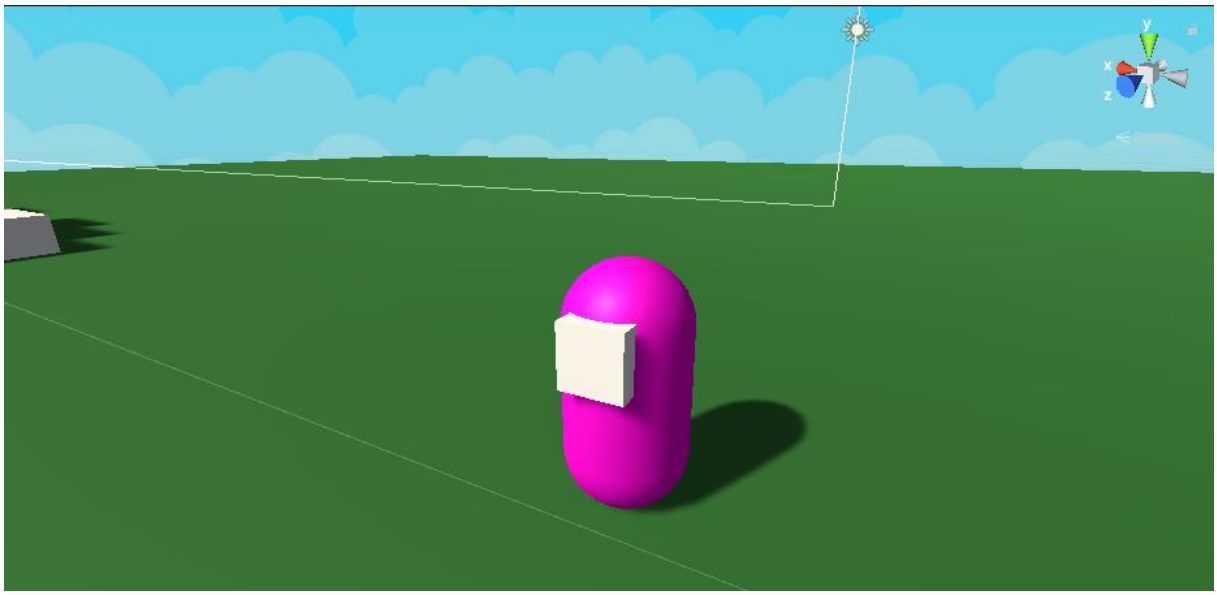


Ilustración 40 Modelo inicial del personaje

Continuando con el desarrollo del jugador, se realizaron dos implementaciones distintas para el movimiento de este; la primera, basada en el componente *Rigidbody*; la segunda, basada en un *Character Controller*. Estas se fueron desarrollando de forma paralela para poder comparar las dos opciones, de esta manera, ver sus ventajas y desventajas. Para el desarrollo de los próximos scripts, en esta y futuras iteraciones, se han tomado de referencia los libros de Borromeo, NA. (2020) y Ferrone, H. (2020) que tratan sobre el desarrollo de juegos con el motor Unity. [16–17]

Durante el desarrollo del personaje mediante un *Rigidbody* se ha desarrollado un *script* llamado *PlayerControllerRigidbody*, el cual contiene la funcionalidad básica para el movimiento del personaje mediante este componente. Esta funcionalidad mínima permite el movimiento del personaje en las direcciones izquierda y derecha, y también permite saltar. Este *script* será añadido al modelo creado anteriormente, junto con el componente *Rigidbody*, al cual se le han modificado las *Constraints* para bloquear la rotación en cualquiera de los ejes.

```
public class PlayerControllerRigidbody : MonoBehaviour
{
    public float moveSpeed;
    public float jumpSpeed;
    private Rigidbody rb;
    // Start is called before the first frame update
    Mensaje de Unity | – referencias
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    Mensaje de Unity | – referencias
    void Update()
    {
        rb.velocity = new Vector3(Input.GetAxis("Horizontal") * moveSpeed, rb.velocity.y, Input.GetAxis("Vertical") * moveSpeed);
        if (Input.GetButtonDown("Jump"))
        {
            rb.velocity = new Vector3(rb.velocity.x, jumpSpeed, rb.velocity.z);
        }
    }
}
```

Ilustración 41 Script PlayerControllerRigidbody

Paralelamente, se implementó esta misma funcionalidad mediante un *Character Controller*, desarrollando un *script* llamado *PlayerControllerCharacterController*. Este fue añadido de nuevo al modelo anterior junto con un componente *CharacterController*, con los parámetros por defecto.

En las pruebas y comparaciones entre ambas implementaciones no se observaron demasiadas diferencias. El jugador en todo momento tenía un control óptimo sobre el personaje.

Tras las pruebas se optó por usar un *Character Controller* por su sencillez y por experiencias anteriores en otros proyectos, por lo que se descartó el uso de un *Rigidbody*.

Siguiendo con el desarrollo del *Character Controller*, se añadieron mejoras a la implementación básica. Primero se añadió una comprobación a la hora del salto para que el jugador no pueda realizar un salto cuando el personaje esté en el aire. A continuación, se implementó la gravedad del personaje para que se aplicara cada vez que esté en el aire, y se añadieron las variables para controlar la velocidad de movimiento y salto. Por último, se añadieron mejoras al personaje relacionadas con la rotación al cambiar de dirección, para que esta sea más suave, interpolando la actual y la que se quiere obtener.

En este punto del desarrollo del personaje, también se inició el desarrollo de la cámara que se usará en el proyecto, esta será explicada en el siguiente punto.

Para terminar con el desarrollo del personaje, se eliminó el modelo creado para las pruebas y se pasó a usarse un modelo de Mario descargado desde Hello Mario

Assets, una colección de modelos y animaciones de Super Mario Bros muy completa, que nos resultará útil durante todo el desarrollo del proyecto [9].

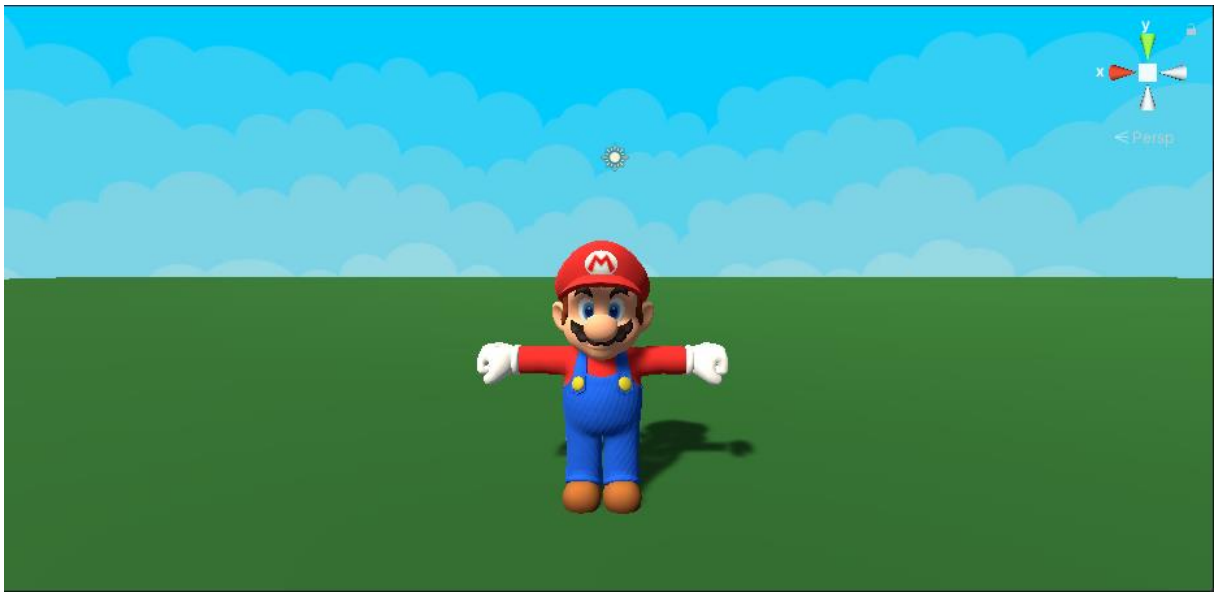


Ilustración 42 Modelo de Mario

A este modelo se le ha añadido un componente *Animator* con su correspondiente *Animator Controller* llamado *PlayerAnimatorController* al cual se le han añadido tres animaciones: la animación de espera, correr y saltar. Estas serán controladas desde el script desarrollado con anterioridad a través de dos variables establecidas en el *Animator Controller*: *speed*, de tipo real, y *isGrounded*, de tipo booleano, que nos permitirán cambiar entre las distintas animaciones.

```
public class PlayerControllerCharacterController : MonoBehaviour
{
    public float moveSpeed = 5;
    public float runSpeed = 10;
    public float jumpSpeed = 8.5f;
    private CharacterController characterController;

    private Vector3 moveDirection;
    public float gravityScale = 2;

    private Animator animator;

    public float rotationSpeed;

    public GameObject playerModel;

    // Start is called before the first frame update

    void Start()
    {
        characterController = GetComponent<CharacterController>();
        animator = GetComponentInChildren<Animator>();
    }

    // Update is called once per frame

    void Update()
    {
        moveWith3DCamera();
    }


    private void moveWith3DCamera()
    {
        if (Input.GetButton("Run"))
        {
            moveDirection = ((transform.right * -Input.GetAxis("Horizontal"))) * runSpeed + transform.up * moveDirection.y;
        }
        else
        {
            moveDirection = new Vector3(-Input.GetAxis("Horizontal") * moveSpeed, moveDirection.y, 0f);
        }

        if (characterController.isGrounded)
        {
            moveDirection.y = 0f;
            if (Input.GetButton("Jump"))
            {
                moveDirection.y = jumpSpeed;
            }
        }

        moveDirection.y = moveDirection.y + (Physics.gravity.y * gravityScale * Time.deltaTime);
        characterController.Move(moveDirection * Time.deltaTime);


        if (Input.GetAxisRaw("Horizontal") != 0)
        {
            Quaternion playerNewRotation = Quaternion.LookRotation(new Vector3(moveDirection.x, 0f, moveDirection.z));
            playerModel.transform.rotation = Quaternion.Slerp(playerModel.transform.rotation, playerNewRotation, rotationSpeed * Time.deltaTime); //Para realizar
        }

        animator.SetBool("isGrounded", characterController.isGrounded);
        animator.SetFloat("Speed", Mathf.Abs(Input.GetAxis("Horizontal")) + Mathf.Abs(Input.GetAxis("Vertical")));
    }
}
```

Ilustración 43 Script final PlayerControllerCharacterController

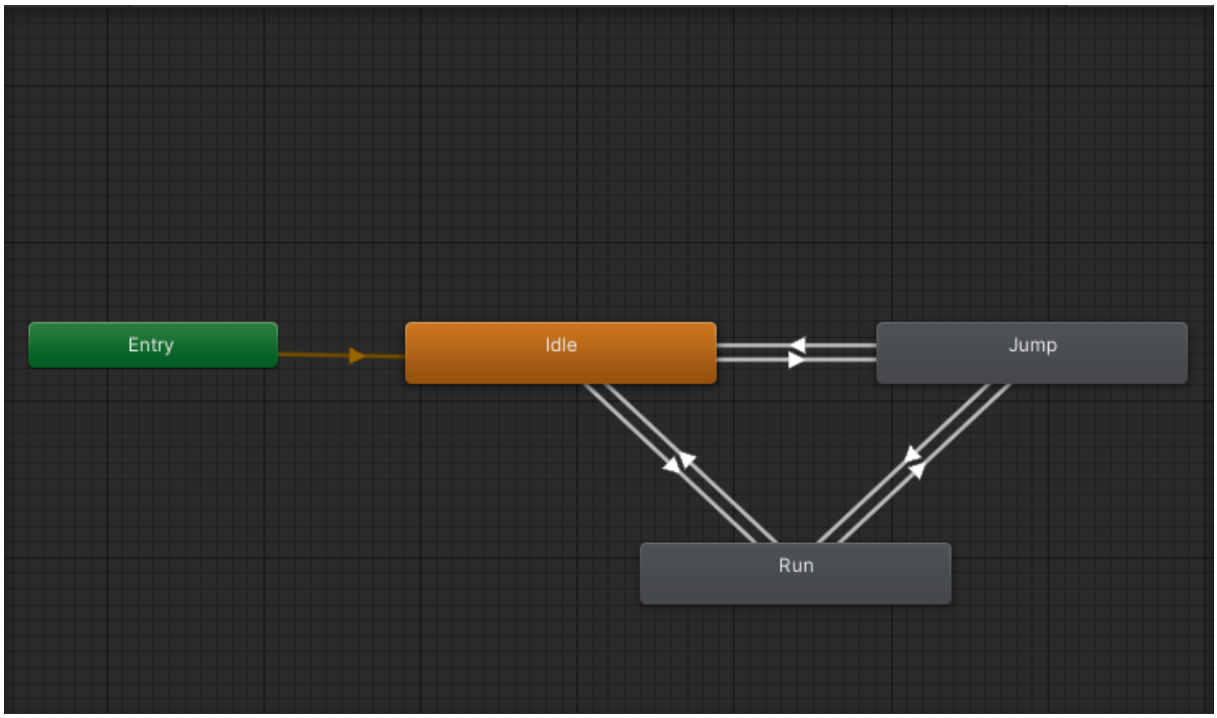


Ilustración 44 Animator Controller de Mario

Por último, se le ha añadido el tag *"Player"* para que pueda ser encontrado fácilmente en cualquier momento.

7.1.3 Cámara 3D

La cámara en 3D dará una vista en 3D de la escena y servirá para poder superar los niveles. Esta mostrará el nivel desde la parte frontal.

Primero se ha creado un objeto Cámara y se le ha añadido un nuevo *script* llamado *CameraController3D*. En este *script*, en el método *Start* se ha buscado el objeto jugador para poder obtener su posición en todo momento. En el método *LateUpdate*, se actualiza la posición de la cámara respecto del jugador y se podrá aplicar un *offset* para desplazar la cámara. Finalmente, se conseguirá que la cámara siempre este apuntando al jugador.

```
public class CameraController3D : MonoBehaviour
{
    private GameObject player;

    public float offsetX = 0;
    public float offsetY = 0;
    public float offsetZ = 0;


    // Start is called before the first frame update

    void Start()
    {
        player = GameObject.Find("PlayerCC").gameObject;
        transform.position = new Vector3(player.transform.position.x - offsetX, player.transform.position.y + offsetY, player.transform.position.z - offsetZ);
    }

    // Update is called once per frame

    void LateUpdate()
    {
        transform.position = new Vector3(player.transform.position.x - offsetX, player.transform.position.y + offsetY, player.transform.position.z - offsetZ);
    }
}
```

Ilustración 45 Script CameraController3D

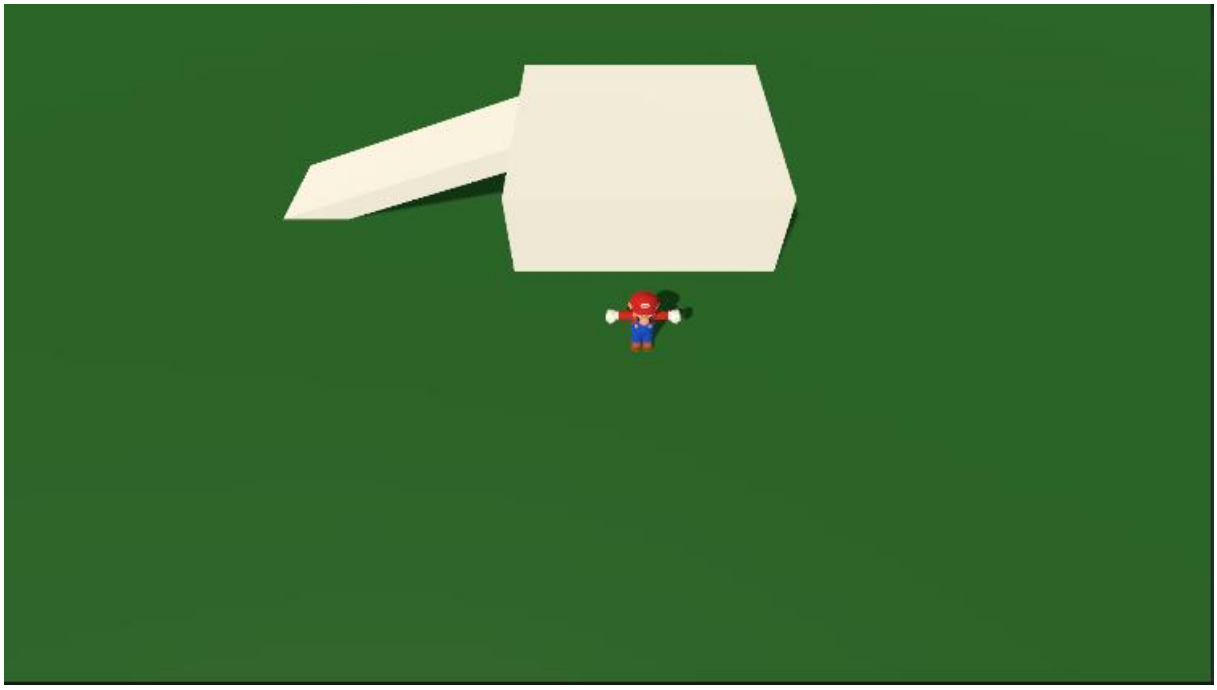


Ilustración 46 Cámara 3D

7.2 Segunda iteración

En la segunda iteración se ha desarrollado el objeto *GameController*, el cual controlará el número de vidas restantes, monedas recogidas a lo largo del nivel y estrellas. También controlará el tiempo restante del nivel y los *powerups*, todo ello desde el *script GameControl*, que estará contenido en el objeto. Junto con el script, se ha desarrollado la interfaz de usuario.

7.2.1 Control de las vidas

Para llevar control de las vidas del personaje se han creado dentro del *script GameControl* las variables necesarias para inicializar las vidas y llevar su control: una variable en la que podremos ajustar el número de vidas iniciales, y otra variable que llevará la cuenta interna de las vidas. Además, se han creado dos métodos públicos estáticos para incrementar y decrementar en uno las vidas, que podrán ser llamados desde otros *scripts* que desarrollemos. El método de decrementar será terminado en iteraciones siguientes para que cuando las vidas lleguen a cero, la partida termine. El código completo del *script GameControl* podrá ser consultado en iteraciones posteriores, cuando este sea terminado.

Para probar el correcto funcionamiento del *script*, se ha creado un objeto *Canvas* al que se le ha añadido un objeto de tipo Texto en la esquina superior izquierda que será actualizado desde el *script*. En el método *Start* se buscará el objeto del *canvas* correspondiente y en el método *Update* se llamará al método *updateDisplays*, el cual realizará la actualización del canvas.

Por otra parte, se ha creado un objeto esfera que simule un enemigo. Hemos aplicado un material rojo y se le ha añadido un nuevo *script* llamado *EnemyMakeDamage* el cual mediante el método *OnTriggerEnter* esperará a un objeto. Cuando un objeto lo active, se llamará al *script GameControl* y a su función para decrementar vidas. Para que no sea activado por cualquier otro objeto de la escena, se le ha añadido la comprobación de que el objeto que lo active debe de tener el *tag "Player".*

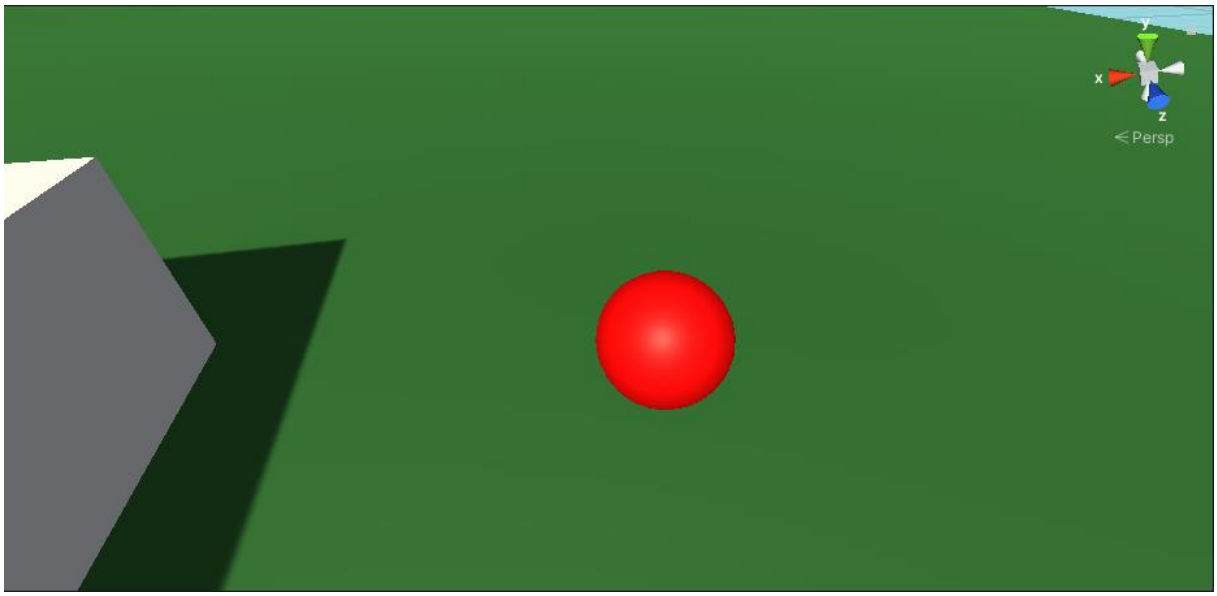


Ilustración 47 Modelo inicial enemigo

Al terminar la implementación se han realizado pruebas y comprobado que el control de vidas funciona correctamente. Al pasar por encima del objeto, este nos restará una vida. Posteriormente, sobre este objeto se implementarán los distintos enemigos del juego.

7.2.2 Control de las monedas

Para llevar control de las monedas recogidas a lo largo del juego se ha creado dentro del *script GameControl*, una variable que almacenará el número de monedas que vayamos recogiendo a lo largo de los niveles. Además, se han creado dos métodos públicos estáticos, para incrementar y decrementar en uno las monedas, que podrán ser llamados desde otros *scripts*.

Para probar el correcto funcionamiento del *script*, se ha creado un objeto *Canvas* al que se le ha añadido un objeto de tipo Texto en la esquina superior izquierda, debajo del anterior, que será actualizado desde el *script*. En el método *Start* se buscará el objeto del *canvas* correspondiente y en el método *Update* se llamará a la función *updateDisplays*, el cual realizará la actualización del *canvas*.

Del mismo modo, se ha creado un objeto esfera que simule una moneda. Hemos aplicado un material amarillo y se le ha añadido un *script* llamado *CoinPickUp* el cual mediante el método *OnTriggerEnter* esperará a un objeto. Cuando un objeto lo

active, se llamará al *script GameControl* y a su función para incrementar las monedas. Para que no sea activado por cualquier objeto de la escena, se le ha añadido también a este método la comprobación de que el objeto que lo active debe de tener el *tag "Player"*.

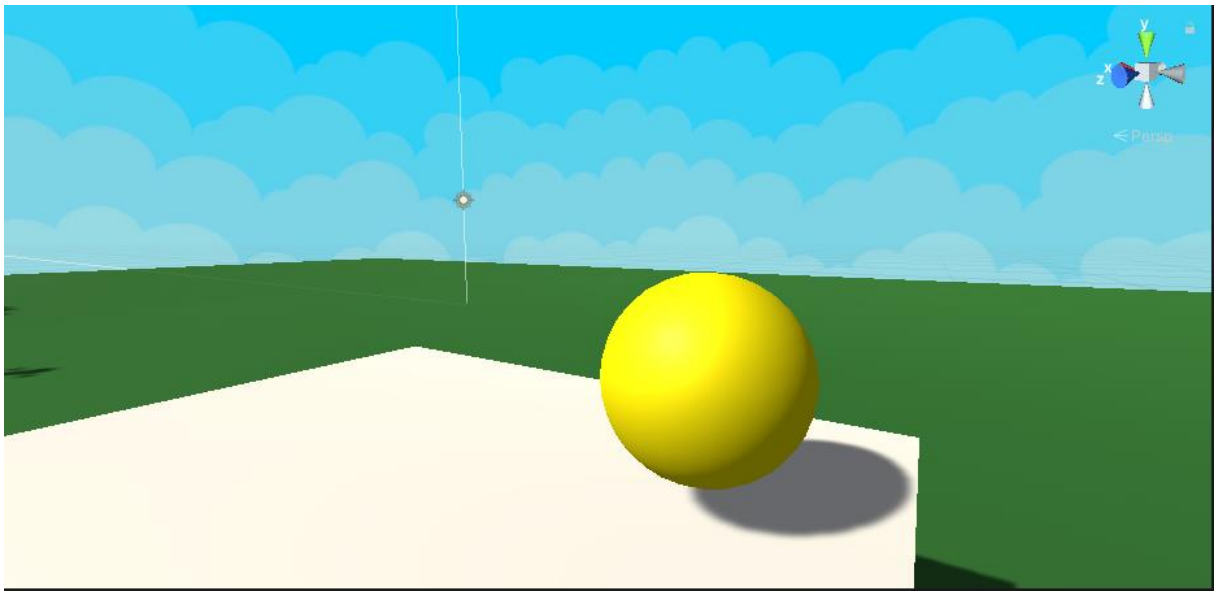


Ilustración 48 Modelo inicial de la moneda

Después de haber comprobado que funciona correctamente, se ha sustituido el modelo anterior por el de una moneda, incluido en el paquete Hello Mario Assets [10]. Además, se le ha añadido a este modelo un componente *Animator* y se le ha creado una animación para que esta rote sobre sí misma continuamente.

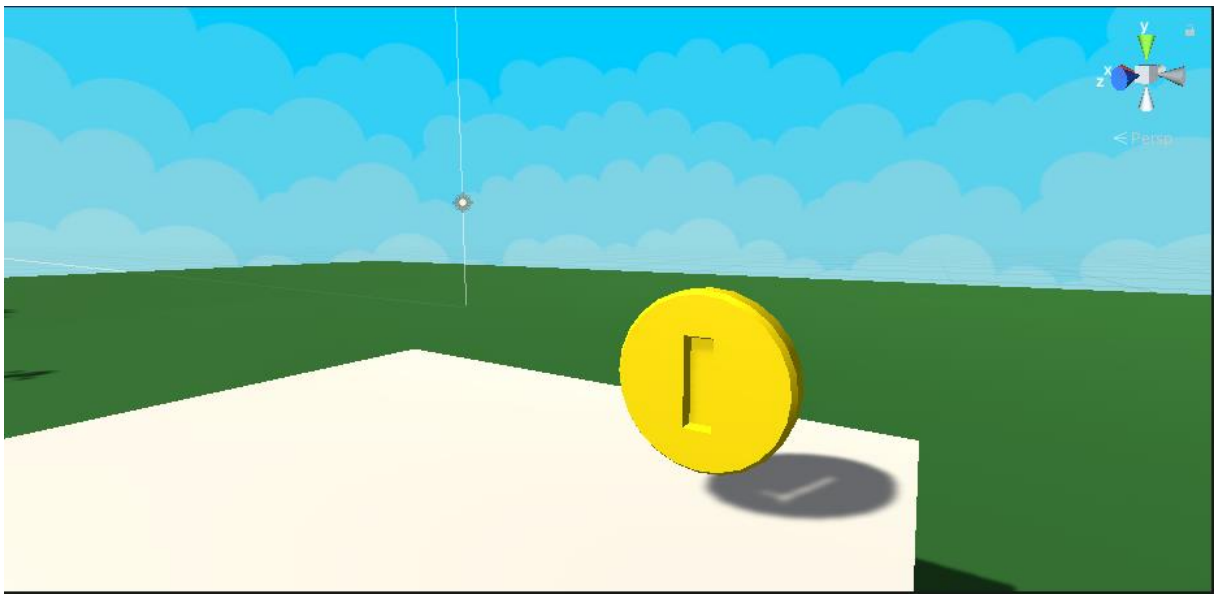


Ilustración 49 Modelo de la moneda

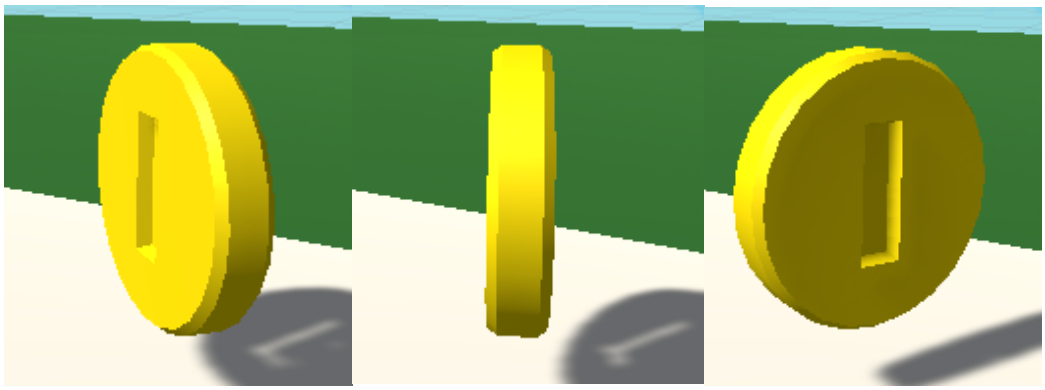


Ilustración 50 Animación de rotación de la moneda

Una vez finalizado estos pasos quedaría completado el desarrollo para el control de las monedas. Así como el objeto junto con su funcionalidad, modelo y animación.

7.2.3 Control de las estrellas

Para llevar control de las estrellas recogidas a lo largo del juego, se ha creado dentro del *script GameControl* una variable que almacenará el número de estrellas que vayamos recogiendo a lo largo de los niveles. También se han creado dos métodos públicos estáticos, para incrementar y decrementar en uno las estrellas, que podrán ser llamados desde otros *scripts*.

Para probar el correcto funcionamiento del script se ha creado un objeto *Canvas* al que se le ha añadido un objeto de tipo Texto en la esquina superior izquierda, debajo de los anteriores, que será modificado desde el *script*. En el método *Start* se buscará el objeto del *canvas* correspondiente y en el método *Update* se realizará la actualización del texto.

También se ha creado un objeto esfera que simule una estrella. Se le ha aplicado un material naranja y se le ha añadido un *script* llamado *StarPickUp* el cual mediante el método *OnTriggerEnter* esperará a un objeto que lo active. En ese momento se llamará al *script GameControl* y su función para incrementar las estrellas y el objeto desaparecerá. Al igual que en el script de las monedas, se le ha añadido también a este método la comprobación de que el objeto que lo active debe de tener el *tag "Player".*

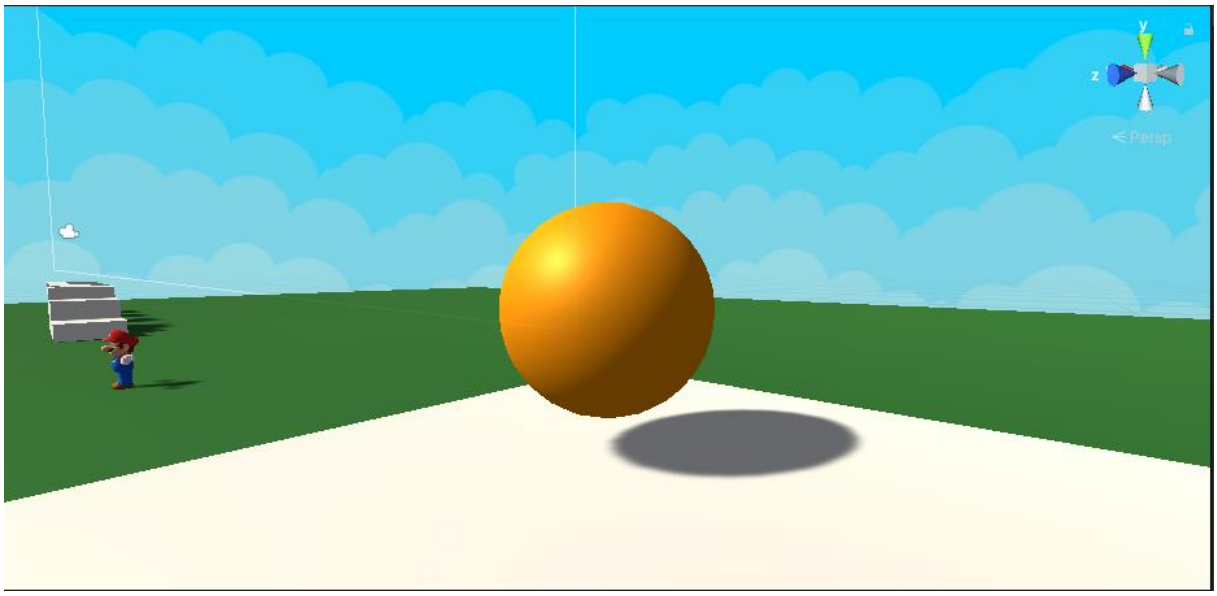


Ilustración 51 Modelo inicial de la estrella

Después de haber comprobado su funcionamiento, se ha sustituido el modelo por el de una estrella, incluido en el paquete Hello Mario Assets. También se le ha añadido a este modelo un componente *Animator* y se le ha añadido una animación que la hará rotar sobre sí misma, de igual manera que las monedas.

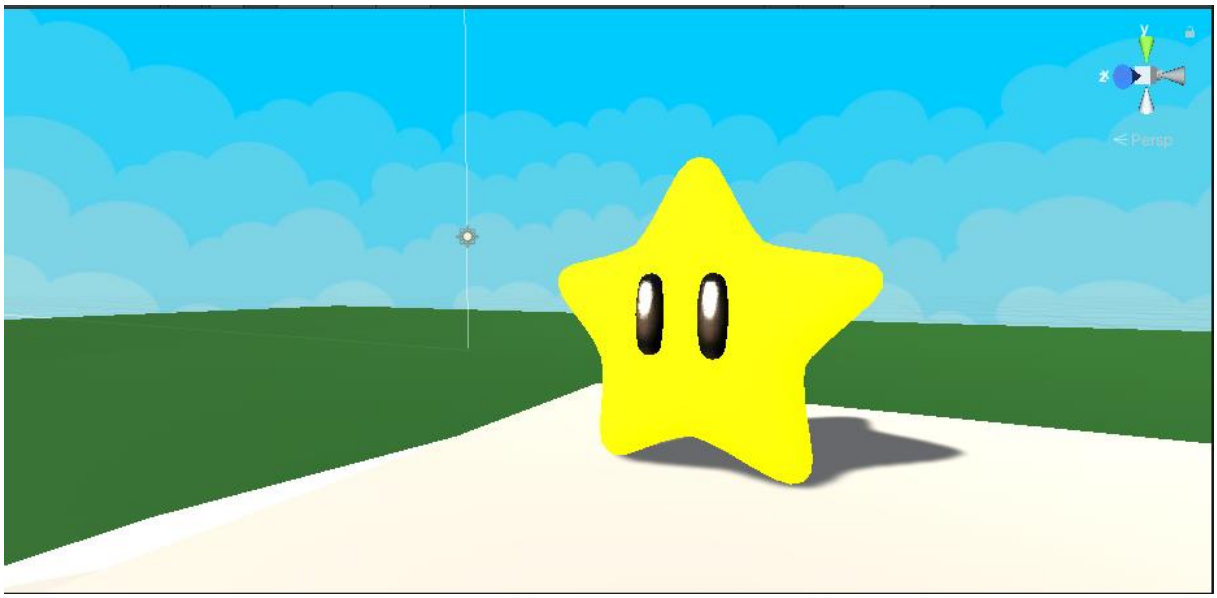


Ilustración 52 Modelo de la estrella

Una vez finalizado estos apartados quedaría completo el desarrollo para el control de las estrellas junto con el modelo y animación que se utilizara en los niveles.

7.2.4 Control del tiempo

Para controlar el tiempo restante en el nivel se ha creado dos nuevas variables en el *script GameControl*; una que almacenará el tiempo inicial del nivel, que por defecto serán trescientos segundos; y una segunda variable que almacenará el tiempo en un momento concreto. Además, se ha creado un objeto Texto en el canvas en la parte superior derecha, que se actualizará de la misma manera que lo anterior.

Para decrementar el tiempo mientras que estamos jugando, se ha creado un método llamado *updateTimer*, que será llamado desde el método *Update*, que restará a la variable donde almacenamos el tiempo la variable *deltaTime* de la clase *Time*. Esta contiene el tiempo entre *frames* en segundos y permitirá que nuestro personaje se mueva de manera constante independientemente del *framerate*. En este mismo método también se comprobará si el tiempo está por debajo de cero, para terminar la partida. Esto se implementará en iteraciones posteriores.

El canvas final quedaría de la siguiente manera:

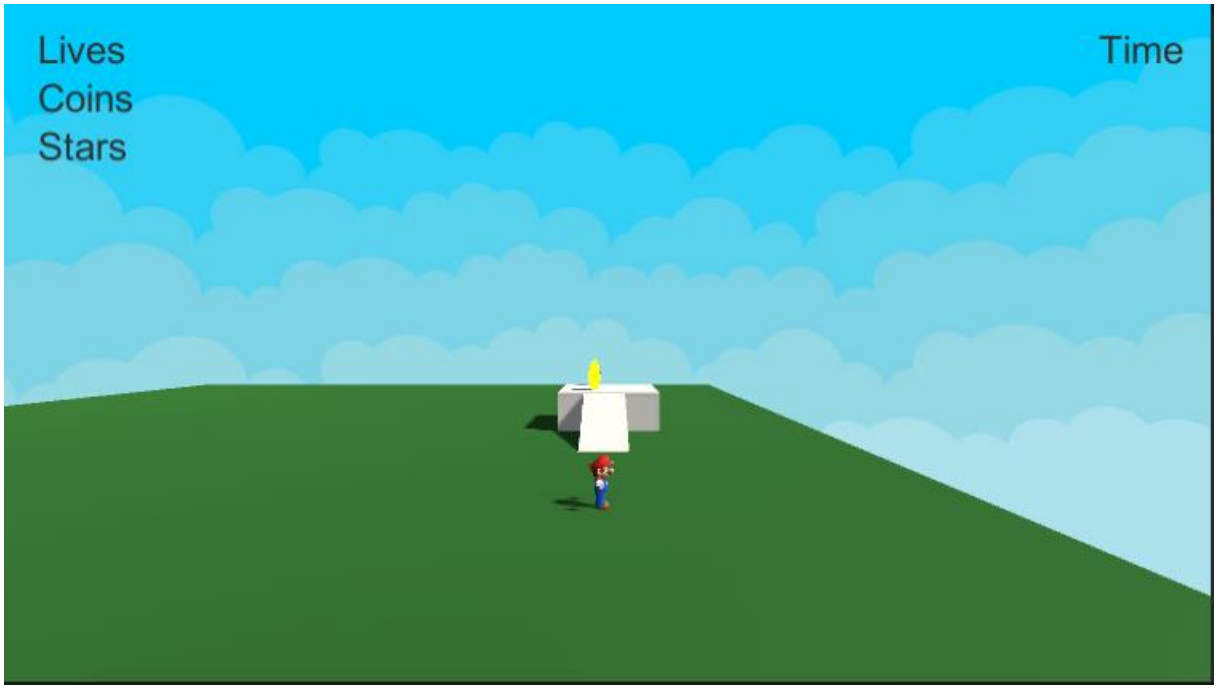


Ilustración 53 Diseño final del canvas

Por último, se han creado dos variables *booleanas* que controlarán si el tiempo se está acabando o, si hay que parar el tiempo en algún momento.

7.2.5 Control de los powerup

El objeto que nos otorgará el *powerup* será desarrollado posteriormente en la siguiente iteración. En esta iteración se han creado las variables y métodos necesarios para llevar el control de los *powerups* posteriormente.

En primer lugar, se han creado tres variables estáticas que controlarán si el jugador tiene un *powerup* activo o no, además, si debe actualizarse el modelo del personaje para reflejar el estado. Por otra parte, se han creado métodos estáticos que podrán ser llamados desde otros *scripts* y que actualizarán estas variables si recogemos un *powerup* o, recibimos daño y perdemos ese *powerup*.

Cuando un *powerup* sea recogido, el tamaño del modelo de nuestro jugador aumentará con una animación que aportará *feedback* al jugador sobre si hemos recogido un *powerup* o no. Si este tiene un *powerup* activo, se le añadirá una vida al jugador; si no, realizará la animación y aumentará su tamaño.

Cuando un enemigo nos haga daño y tengamos un *powerup* activo, nuestro jugador cambiará su escala, haciendo la animación contraria a la anterior.

Se ha creado un método *updateMarioScale* que comprobará si hay que cambiar esta escala. Esta será llamada desde el método *Update*.

7.2.6 Control de la puntuación

Por último, en esta segunda iteración, se ha creado un contador en el canvas mediante un Texto, en la parte superior central de la pantalla. Se ha añadido al *script GameControl* una variable que llevará el control de la puntuación.

Cada vez que el jugador realice una acción como matar enemigos, recoger monedas, *powerups*, etc. tendrán una recompensa en puntos. Estos puntos están reflejados en la siguiente tabla:

Acción	Puntuación
Eliminar enemigo	100
Recoger powerup	50
Recoger vida extra	150
Moneda	100
Terminar nivel	Tiempo restante del nivel

Tabla 5 Puntuaciones

Para que el contador funcione se ha implementado como una corutina que actualizará en todo momento el score, incrementándolo poco a poco hasta llegar a la cantidad de puntos que tenemos. Esto quiere decir que no realizará la suma de golpe. Para ello se han creado los métodos *updateScore*, que será iniciado como corutina en el método *Start* del *script GameControl*; y el método *addScore,* que podrá ser llamado desde cualquier otro script para incrementar los puntos del jugador.

7.2.7 Finalización de la interfaz

Para finalizar el desarrollo de la interfaz se han modificado las fuentes de todos los textos añadidos en el canvas. La nueva fuente se ha obtenido desde DaFont [13] y es una réplica de la fuente usada en el título de Super Mario. También, se han utilizado imágenes para que el jugador sepa que representa cada uno de los indicadores en pantalla.

Ilustración 54 Interfaz final del juego

7.3 Tercera iteración

En la tercera iteración se llevará a cabo el desarrollo de los objetos básicos del videojuego como lo son: la seta roja y verde, el bloque de ladrillo, el bloque “?” y las tuberías. De cada uno de estos objetos se obtendrá un *prefab* el cual podremos usar a lo largo del desarrollo de los niveles. Estos objetos serán comunes a la mayoría de

los niveles. También son objetos básicos los desarrollados en la segunda iteración para las pruebas del *script* desarrollado en esa iteración.

A todos estos nuevos objetos se les añadirá funcionalidad, modelo y animaciones, si fuese necesario.

7.3.1 Seta roja

Para comenzar el desarrollo de este objeto se ha creado un cubo que representará la seta roja, añadiremos su modelo más tarde. Se ha desactivado su *Box Collider* y se le ha añadido como hijo un objeto vacío al que llamaremos *Trigger*, que contendrá el *script* que le proporcionará la funcionalidad. Además, tendrá un *Box Collider* con la opción *isTrigger* activada, para que el jugador pueda interaccionar con ella.

Se ha creado un *script* llamado *RedMushroomPickUp* y se ha añadido el método *onTriggerEnter*. Dentro de esta, cada vez que un objeto con el *tag "Player"* active el *trigger*, que llamará al método de la clase *GameControl* llamada *powerUpCollect* que realizará el resto de las acciones. Se desactivará también el modelo y el *collider* de la seta, para que esta desaparezca y no pueda ser usada más de una vez.

Para finalizar, se ha añadido un modelo del paquete Hello Mario Assets y se ha desactivado el renderizador del cubo. Este modelo ha sido añadido como hijo del objeto, a la misma altura del *trigger*.

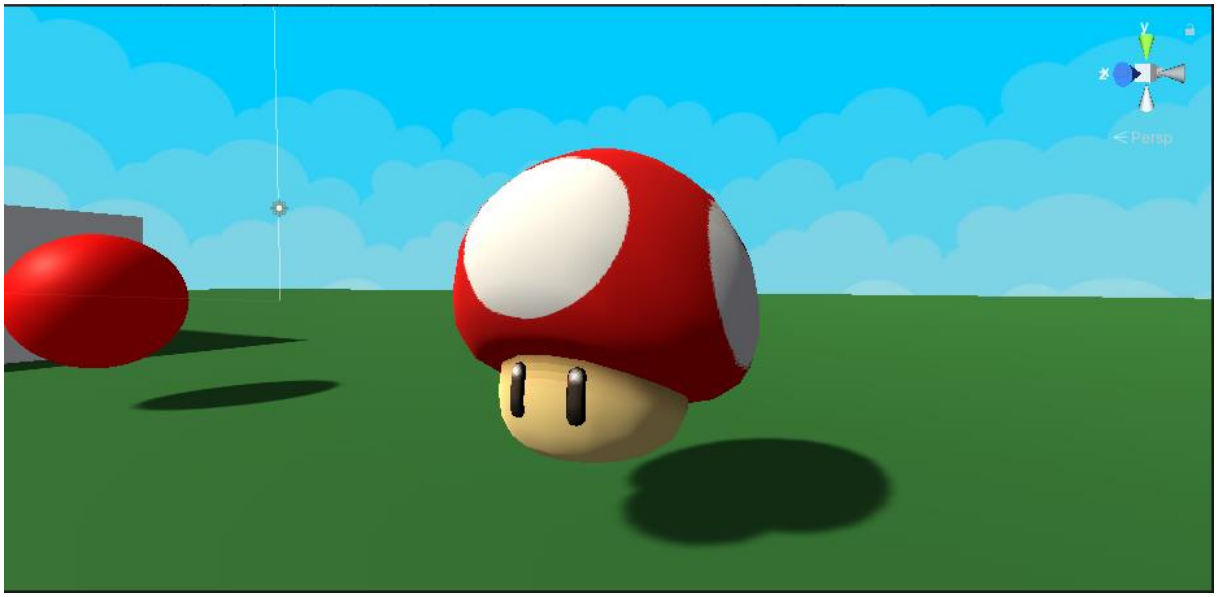


Ilustración 55 Modelo de seta roja

7.3.2 Seta verde

Para el desarrollo de la seta verde se ha realizado una implementación igual que la de la seta roja, pero cada vez que se llame al método *onTriggerEnter*, en el nuevo *script GreenMushroomPickUp*, se llamará al método *increaseLives* de *GameControl.*

Por último, se le ha añadido un modelo del paquete Hello Mario Assets que se corresponde con la seta verde.

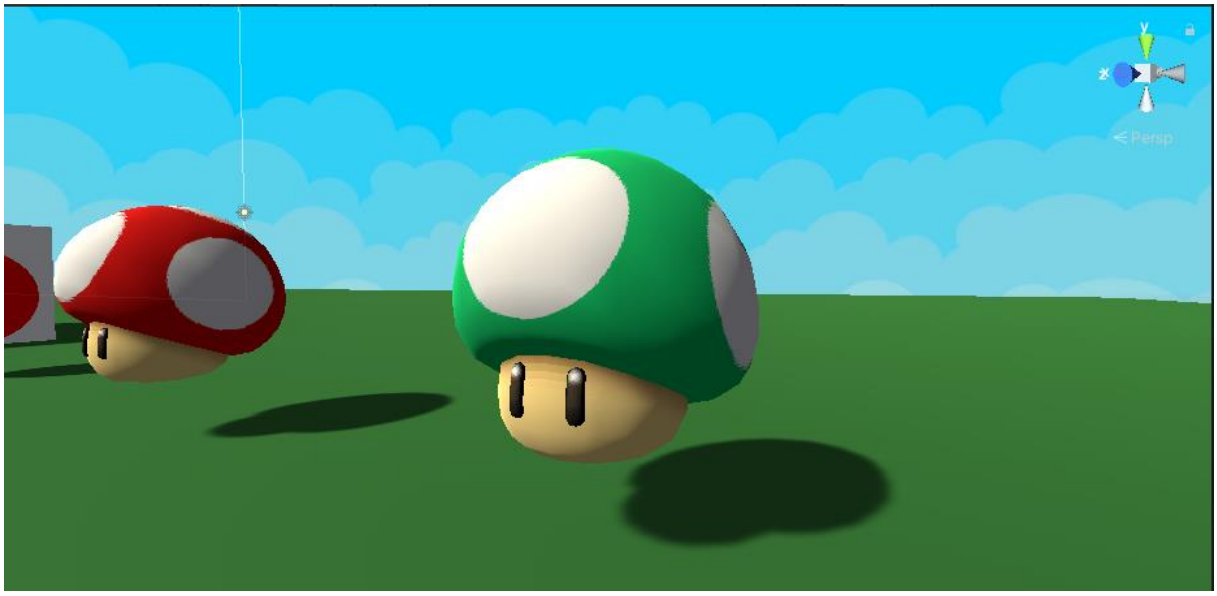


Ilustración 56 Modelo de seta verde

7.3.3 Bloque "?"

El bloque "?" es uno de los objetos básicos de Super Mario. Este podrá ser golpeado por la parte inferior y liberará un *powerup* por la parte superior. Este *powerup* será la misma seta roja que hemos desarrollado anteriormente.

Para comenzar con su desarrollo se agregó un objeto cubo a la escena junto con un *Box Collider*. Después se ha creado un objeto hijo vacío que será nuestro *trigger*, al que se le ha añadido un *Box Collider* con la opción *isTrigger* activada. A este mismo objeto se le ha añadido un nuevo script llamado *QuestionBlockActivate*.

En el *script* se declarará el método *OnTriggerEnter* y se comprobará que el *collider* contenga el *tag "Player"*. Si cumple la condición, nos activará el *powerup* y el bloque cambiará de forma.

Antes de continuar con el *script*, se han añadido los dos modelos necesarios para el bloque como hijos del primer objeto. El primero de los modelos será el bloque antes de ser activado; el segundo, después de ser activado.

Ilustración 57 Modelo de bloque ? sin abrir

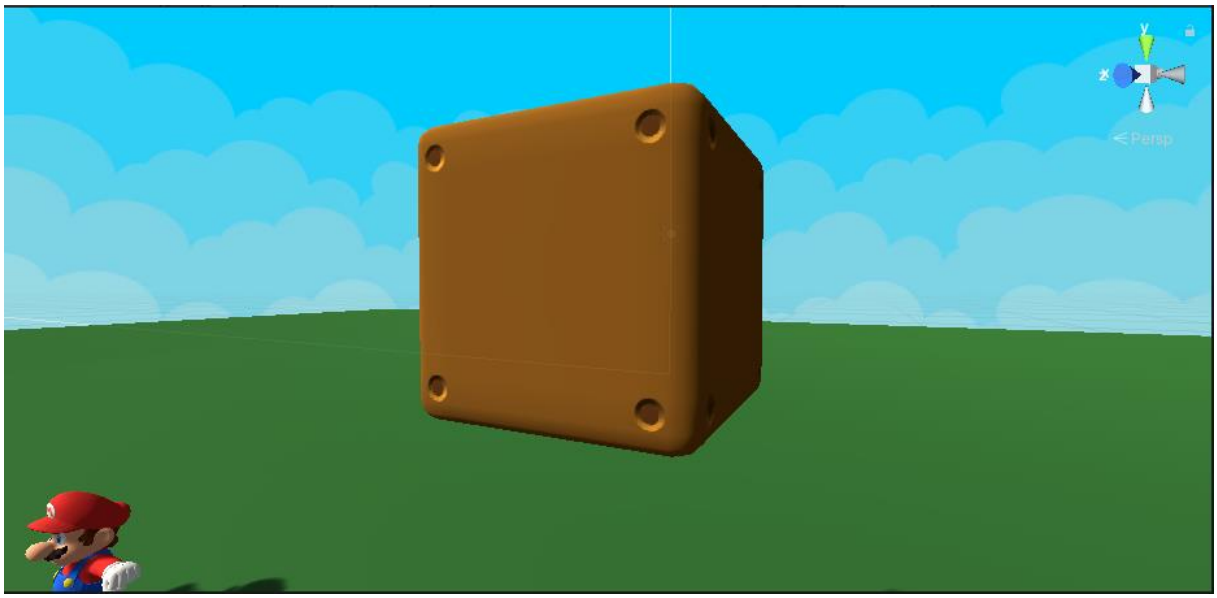


Ilustración 58 Modelo de bloque ? activado

Estos modelos han sido añadidos desde el paquete Hello Mario Assets.

Se ha añadido la seta roja desarrollada en apartados anteriores, que estará contenida en el interior del bloque y que, por medio de una animación, se hará que salga por la parte superior.

En el *script* tendremos referencias a estos tres bloques y cuando el *trigger* se active, se cambiará el modelo del bloque y se activará a su vez la animación de la seta, que saldrá por la parte superior y se quedará encima del bloque.

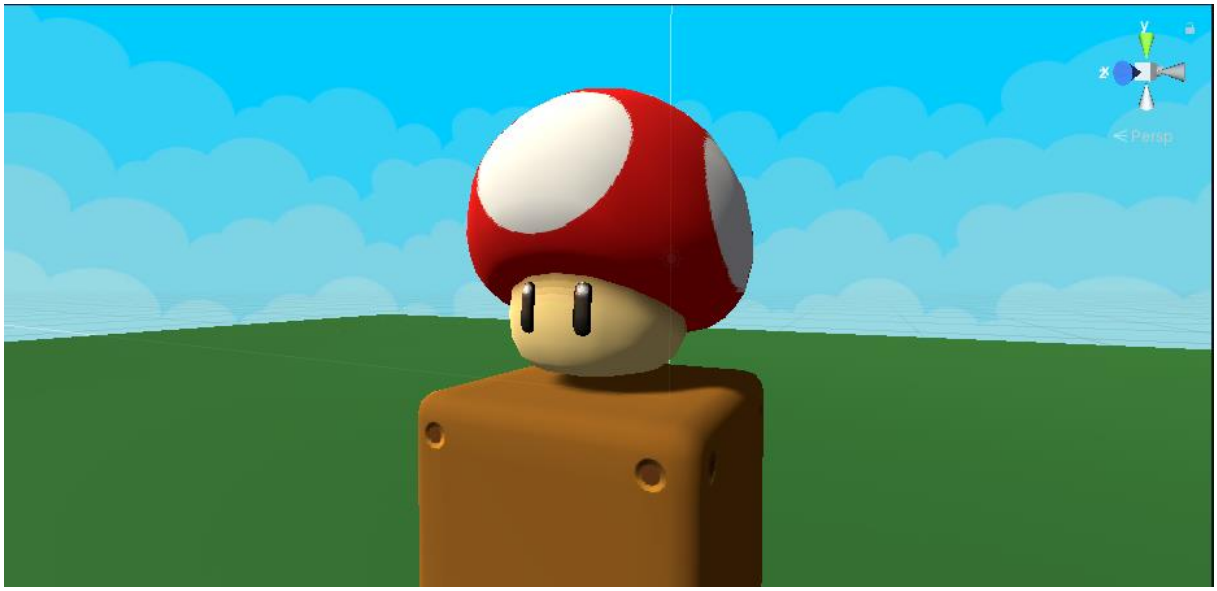


Ilustración 59 Estado final del bloque "?"

7.3.4 Bloque de ladrillo

El bloque de ladrillo podrá ser activado desde la parte inferior cuando el personaje tenga un *powerup* activo. Si no lo tiene, el bloque realizará una animación; si lo tiene activo, el bloque se romperá. Estos bloques tendrán dos variantes: con moneda o sin ella.

Para el desarrollo del bloque se comenzará añadiendo un objeto vacío, al que se añadirá el modelo del bloque antes de romperse y después de romperse, como en el bloque anterior. Más tarde se añadirá un objeto vacío que será el *trigger.* Este tendrá un *Box Collider* con la opción *isTrigger* activada. Este tendrá un nuevo script llamado *BrickBlockBreak* el cual mediante al método *OnTriggerEnter* esperará a ser activado por un objeto que tenga el *tag "Player".* Se tendrá en cuenta si el personaje tiene un *powerup* activo mediante el *getter* de la clase *GameControl*.

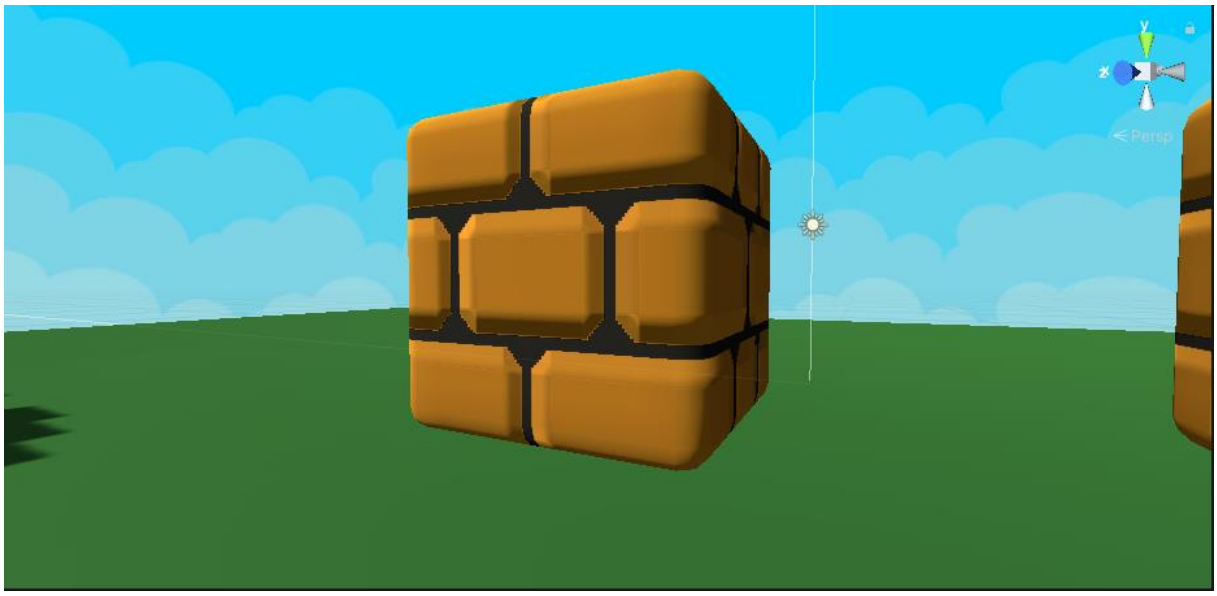


Ilustración 60 Modelo bloque de ladrillo

Si el personaje tiene un *powerup* activo, el modelo del bloque cambiará y después de esto se activará la animación de romperse. Cuando acabe, el bloque desaparecerá.

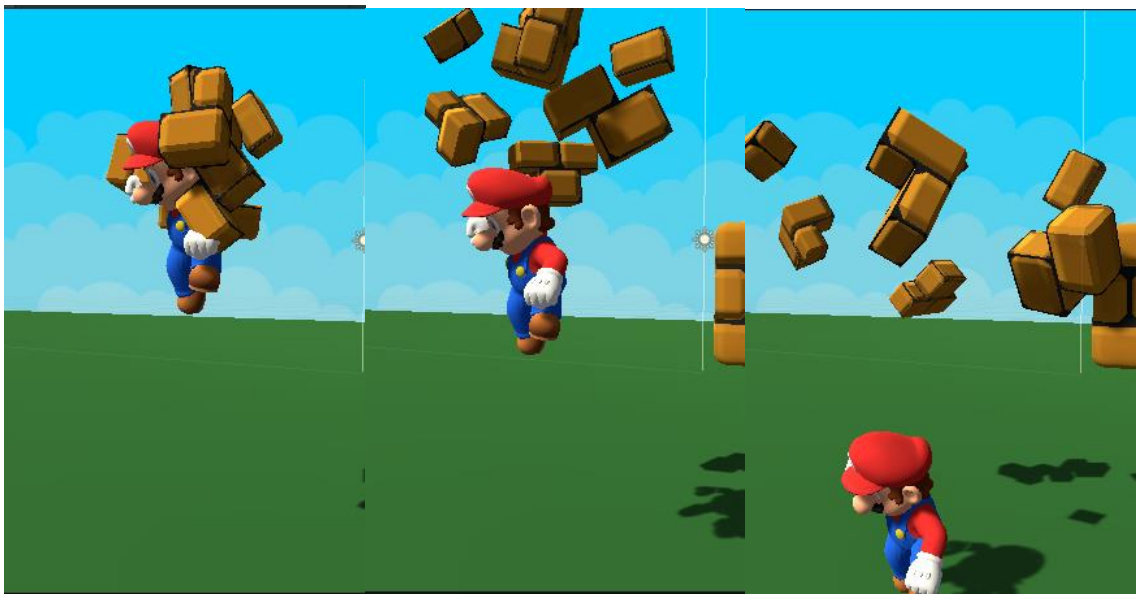


Ilustración 61 Animación de romperse bloque de ladrillo

En el caso de que no haya un *powerup* activo, se realizará la animación de botar. Esta acción indica que no tenemos ese *powerup*.

Por último, se añadirá al script una variable *booleana* que indicará si un bloque contiene una moneda o no. Si el bloque es activado y contiene una moneda, se llamará al método para incrementar las monedas de la clase *GameControl*.

7.3.5 Tubería

La tubería es el último de los objetos básicos que se creará. Esta permitirá trasladar el personaje a otra ubicación del mapa. Cuando el jugador se posicione encima podrá pulsar el botón Control Izquierdo. Esto activará la tubería y su animación, junto con otra de transición. Una vez termine esta animación, el jugador se encontrará en otra posición.

Para comenzar el desarrollo de las tuberías se obtuvo el modelo incluido en el paquete Hello Mario Assets de la tubería.

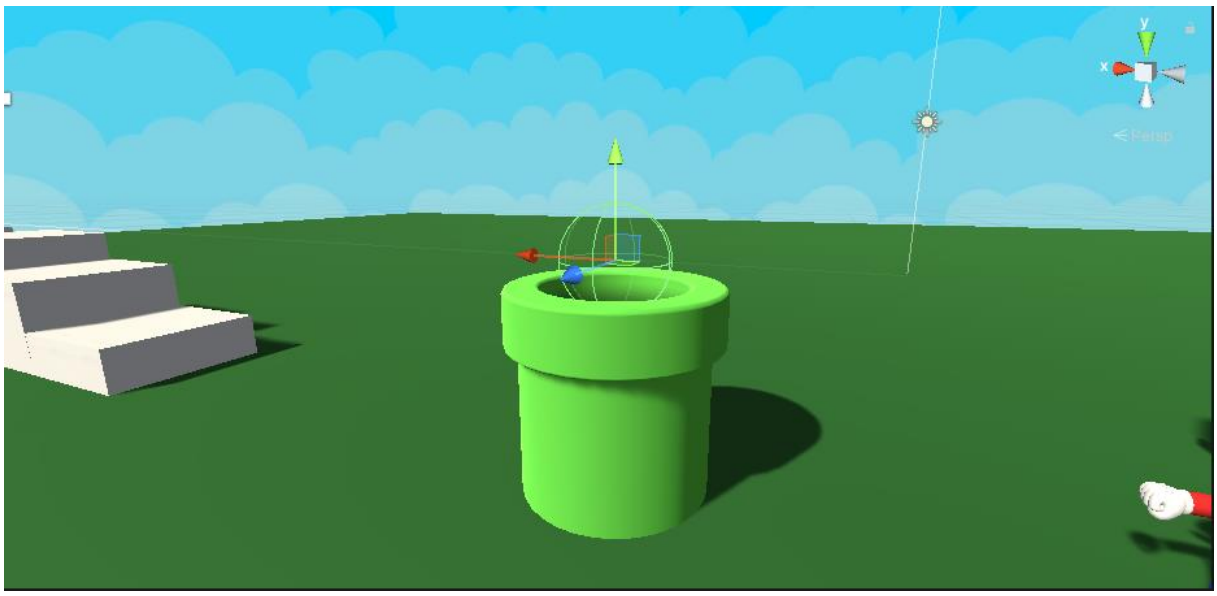


Ilustración 62 Modelo de la tubería

Se creó un objeto padre vacío y se le añadió el modelo como un nuevo objeto hijo. Se le ha añadido un objeto vacío que será nuestro *trigger* y un objeto que será el *collider* de la tubería.

El objeto *trigger* tendrá un componente *Sphere Collider* que será colocado en la parte superior. Este tendrá la opción *isTrigger* activada para que desde el nuevo script que tendrá este objeto.

El objeto *collider* servirá de *collider* para la tubería y a la vez realizará la animación que simulará el personaje entrando por la tubería.

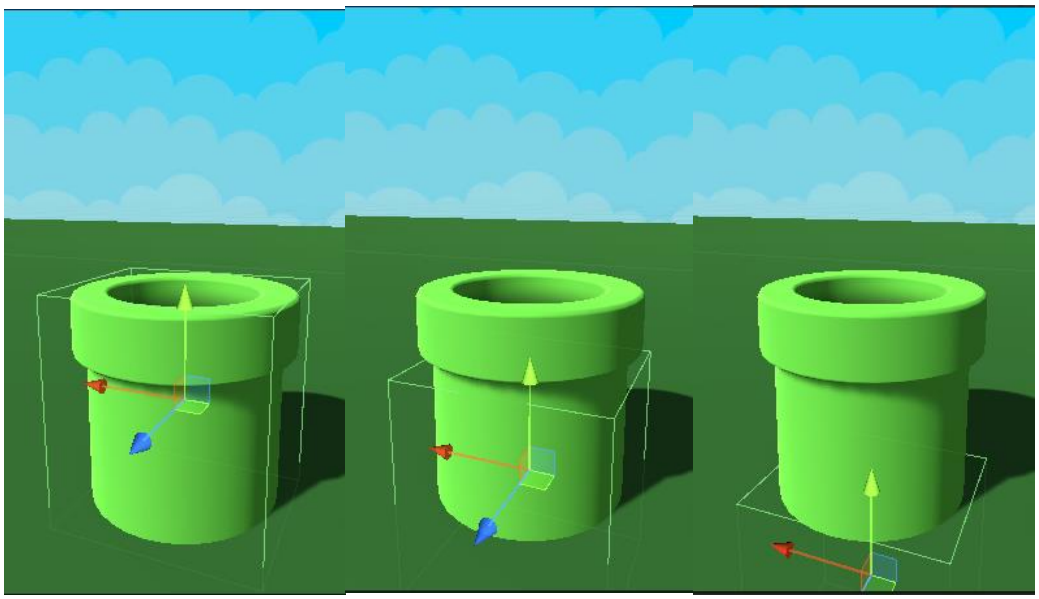


Ilustración 63 Animación del collider de la tubería

Se ha creado un *script* llamado *PipeUse*, que se añadirá al *trigger*, que comprobará mediante los métodos *OnTriggerEnter* y *OnTriggerExit* si el jugador está encima de la tubería, modificando una variable *booleana* llamada *playerAtTop*. En el método *Update*, se podrá comprobar si el jugador ha pulsado el botón Control y si esta encima, con la variable *playerAtTop*. Esto activará una corutina *usePipe* que colocará al jugador justo en el centro y desactivará el controlador del personaje. Activará la animación del *collider* y después trasladará al jugador a la posición final. Para la posición final se ha creado un objeto vacío que se colocará en la posición de salida de nuestro personaje.

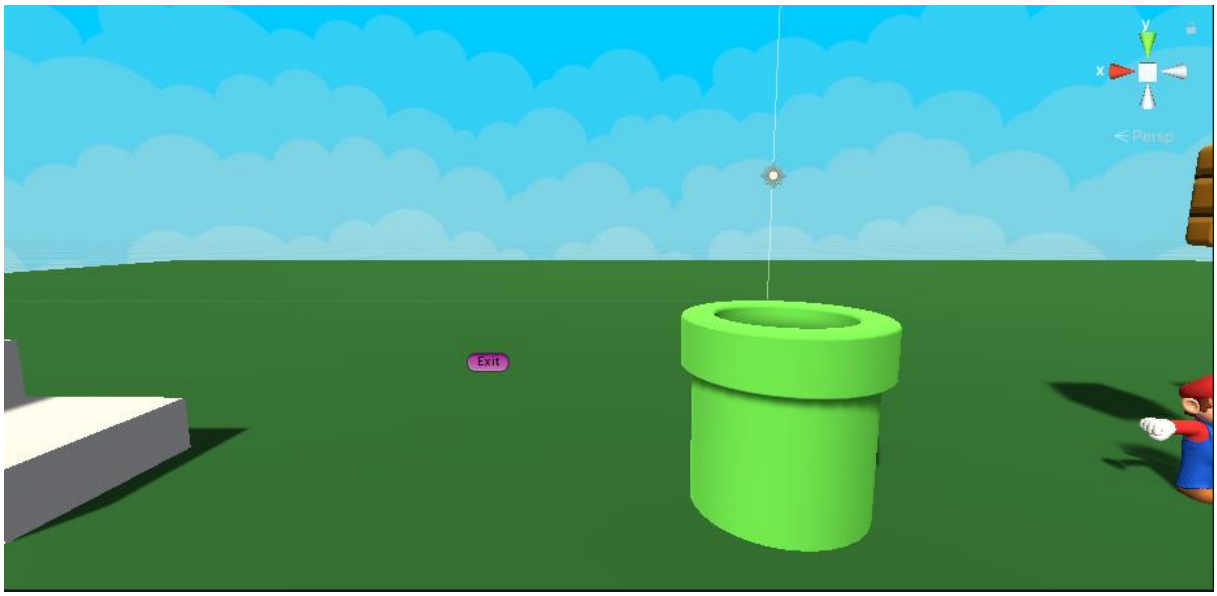


Ilustración 64 Objeto de salida y tubería final

Por último, para que la transición de una posición a otra sea menos brusca se ha creado una transición con un objeto Imagen añadido al *canvas*. Esta imagen estará completamente en negro. A través de una animación se podrá controlar su transparencia, realizando una transición a negro y después de vuelta.

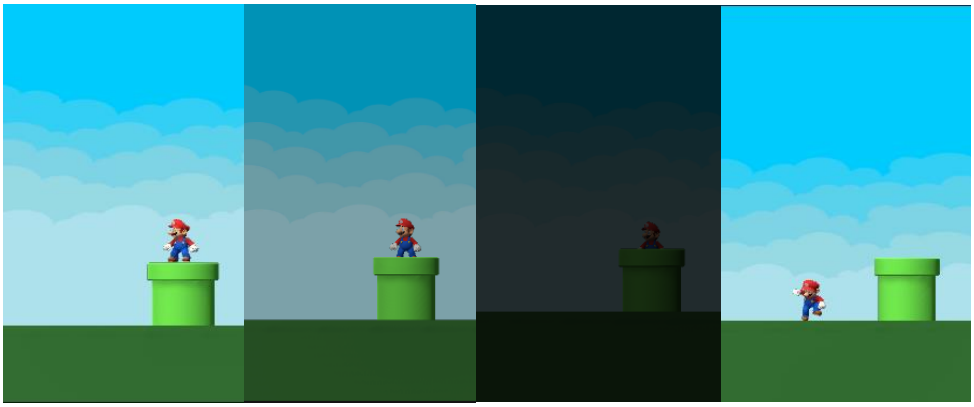


Ilustración 65 Animación de la pantalla de transición

7.4 Cuarta iteración

En esta iteración se llevará a cabo el desarrollo del primer nivel del juego, del primer enemigo del juego, su pantalla de carga, el cambio de escenas y la escena de *game over*. También se llevará a cabo el desarrollo del segundo nivel y los elementos necesarios para completarlo.

7.4.1 Nivel 1 (1-1)

Para la base y las estructuras que existen a lo largo del escenario se han creado dos cubos a los que se le han aplicado dos texturas. Ambos bloques tendrán un componente *BoxCollider*.

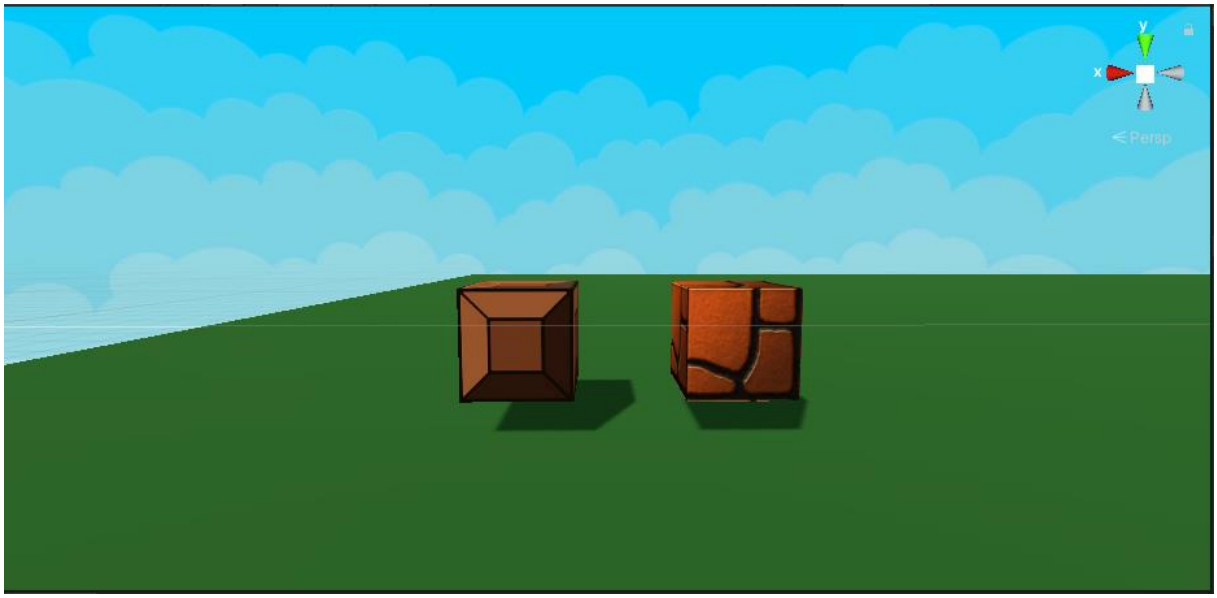


Ilustración 66 Bloques de las estructuras y suelo

Una vez recreada la estructura básica del diseño del nivel, se han añadido todos los objetos que contiene.

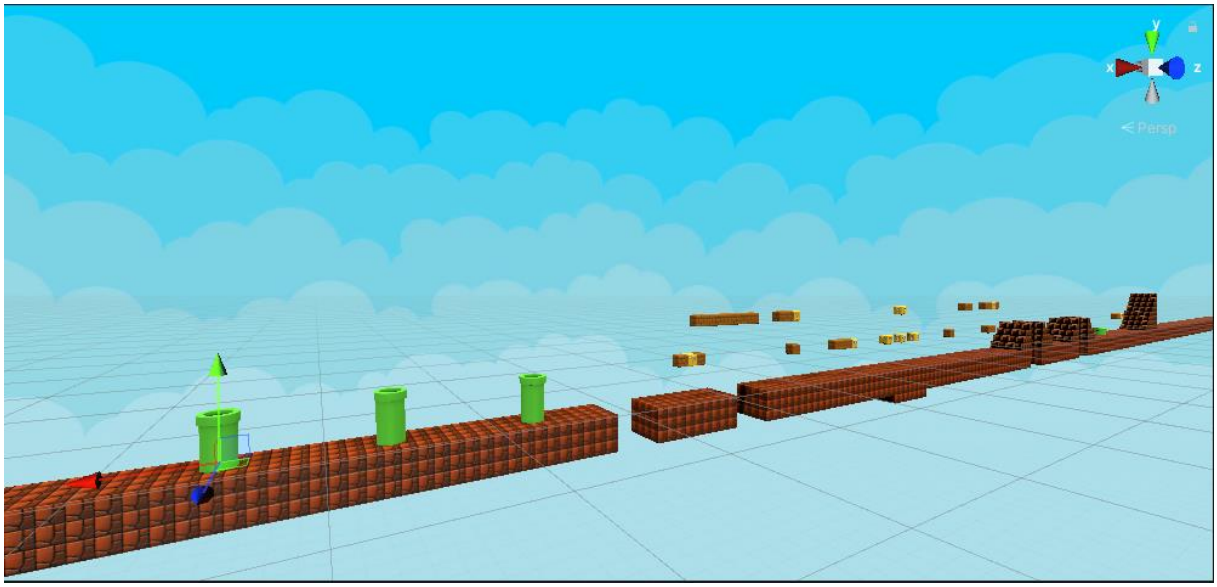


Ilustración 67 Nivel 1

También se han creado dos nuevos cubos para crear la zona a la que se accede a través de la tubería, que será una zona bonus que contendrá monedas.

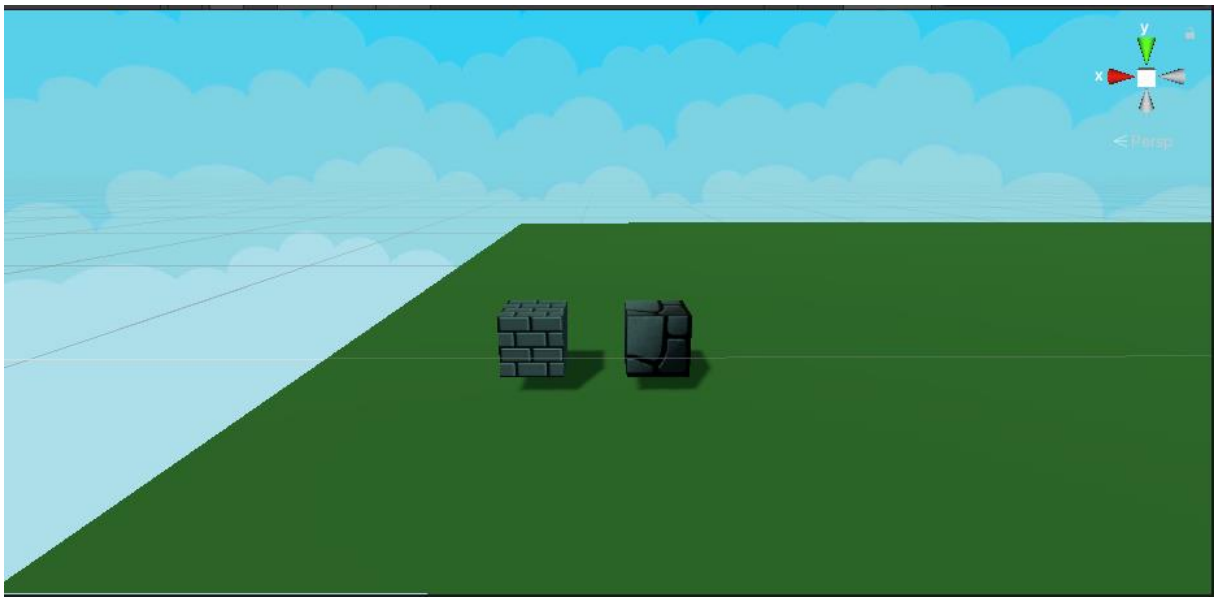


Ilustración 68 Bloques para la zona bonus

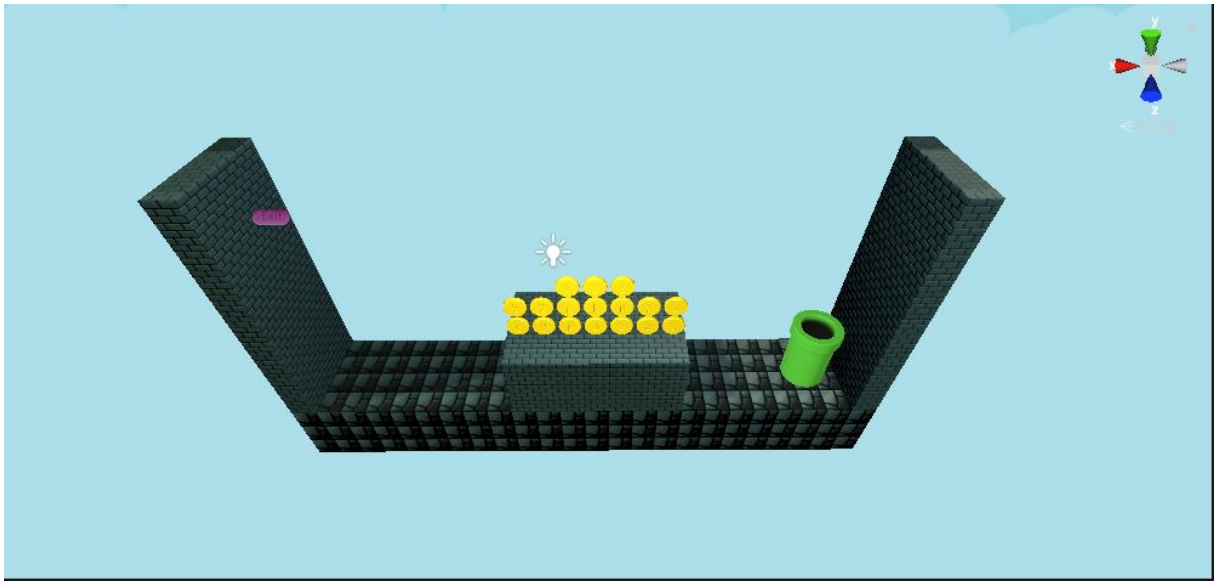


Ilustración 69 Zona bonus

Se ha desarrollado también un objeto plano que cubre todo el escenario en la parte inferior, y que contendrá un *BoxCollider* con la opción *isTrigger* activada para que, si el jugador sale del escenario, este pierda y se reinicie el nivel. Para ello, se ha creado el *script LevelDeath* en el cual cuando el jugador entre, llamará al método de decrementar las vidas del *script GameControl*, que realizará el reinicio del nivel. Por

último, se ha creado un *prefab* llamado *LevelDeathTrigger* para que pueda ser usado en los siguientes niveles.

Se ha usado un *asset* descargado desde la Assets Store llamado LowPoly Water [14] el cual contiene un *prefab* que simula un océano o mar con oleaje y que colocaremos justo debajo de nuestro nivel como decoración, al mismo nivel que el *LevelDeathTrigger.*

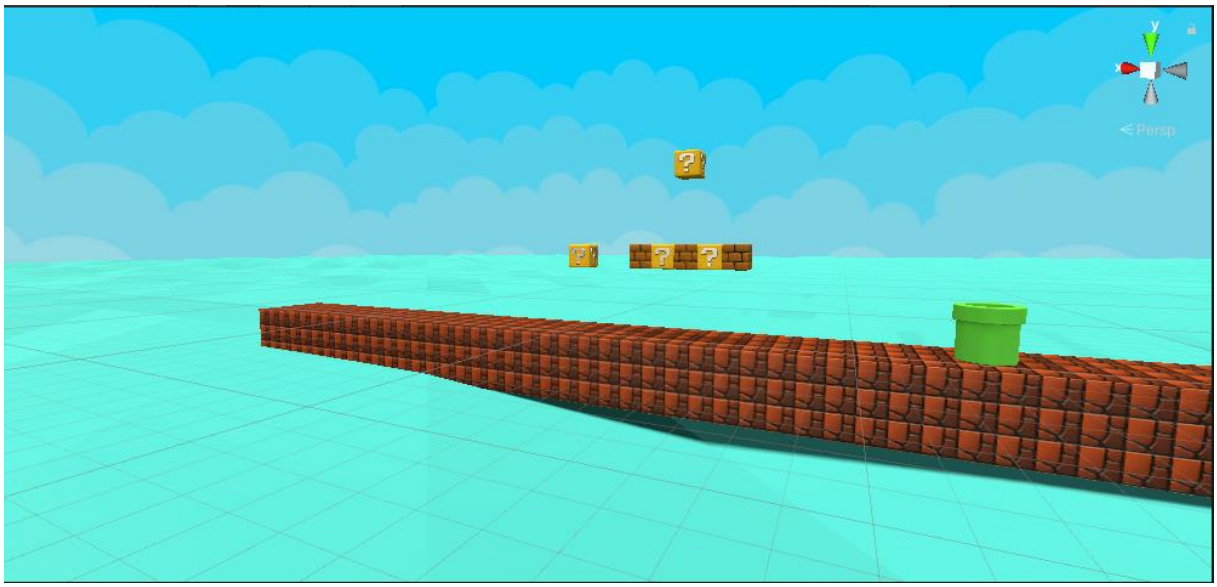


Ilustración 70 LowPolyWater

7.4.1.1 Goomba

Para la implementación de este enemigo se ha partido de la base desarrollada en la segunda iteración.

Se ha añadido el modelo del paquete Hello Mario Assets al enemigo base y se le ha creado un nuevo objeto que contendrá un componente *Box Collider* con la opción *isTrigger* activada. Este será colocado en la parte superior del modelo y contendrá el *script EnemyDead* en el cual el método *OnTriggerExit* esperará a un objeto con el *tag "Player"* y en ese momento desactivará sus animaciones y *scripts*, cambiará su forma a una forma "aplastada" y pasado un tiempo el objeto será desactivado.

Para el movimiento, se ha creado el *script EnemyMove2D* el cual hará que el enemigo se mueva entre dos puntos. Estos puntos serán dos objetos vacíos que se colocarán en las escenas y de los que se obtendrá su posición, y facilitarán el diseño del nivel al solo tener que situar los puntos a lo largo de este.

Ilustración 71 Prefab del Goomba

El Goomba se moverá entre ambos puntos, mirando en la dirección del movimiento y cambiando de dirección cuando llegue a uno de ellos. La velocidad de movimiento puede ajustarse desde el Inspector.

Por último, se creará un *prefab* y se colocará a lo largo del nivel.

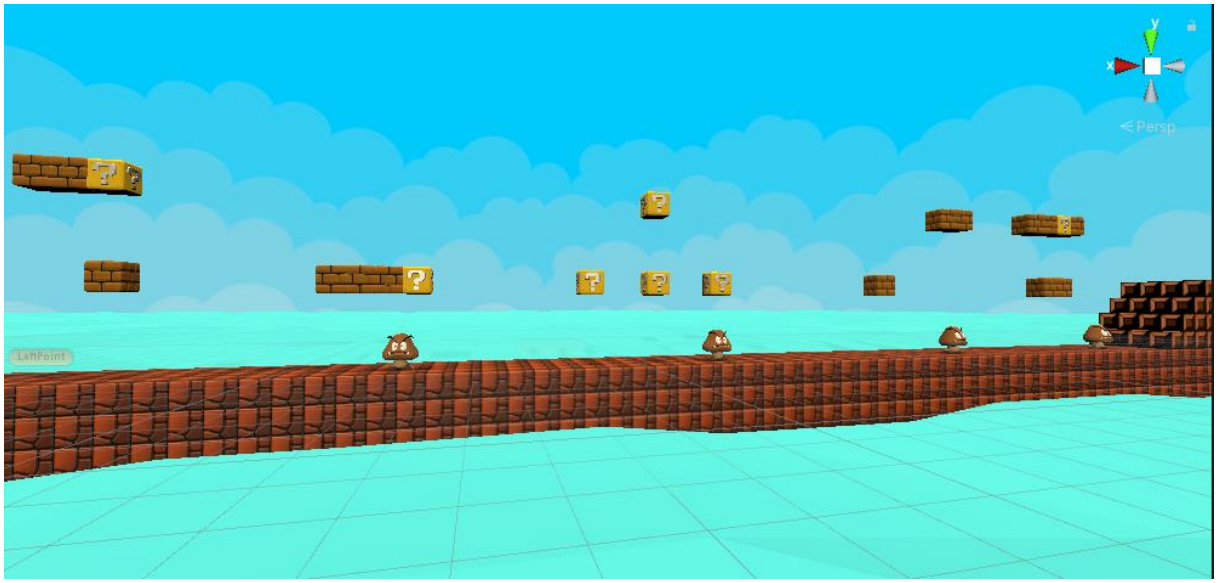


Ilustración 72 Goombas en el nivel 1

7.4.1.2 Pantalla de carga

Esta será la pantalla que el juego mostrará cada vez que se pase de nivel o que este se reinicie. Se ha creado una escena nueva que contiene un canvas con una

imagen en negro, que será el fondo, y un texto que indicará el nivel que se está cargando.

Ilustración 73 Pantalla de carga nivel 1

En esta escena también se ha añadido un objeto vacío que contiene el *script LevelLoad* en el cual el método *Start* incluirá una *corutina* que esperará durante dos segundos, mientras se muestra esta pantalla, para después cargar el nivel correspondiente.

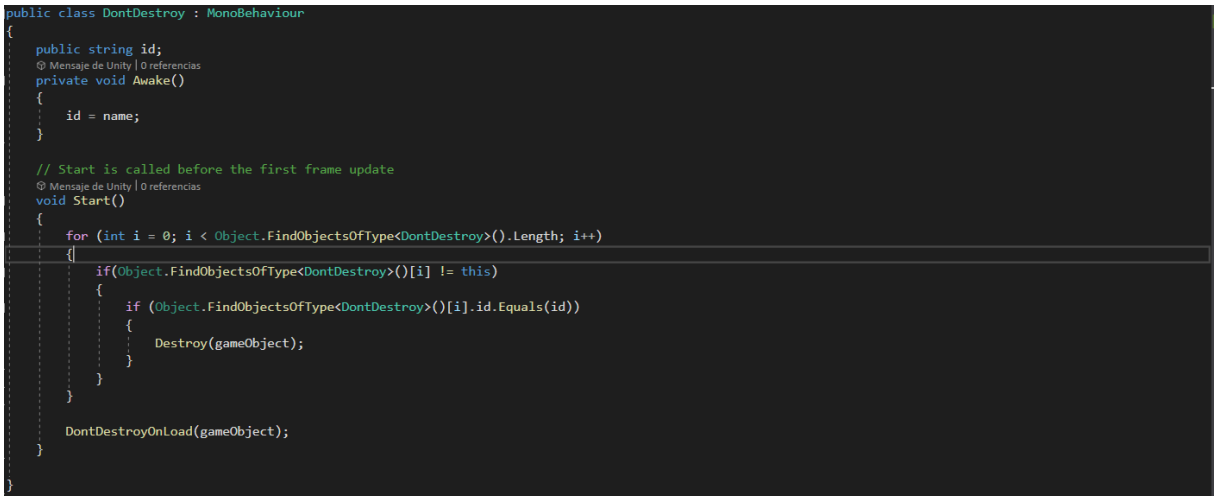
7.4.2 Escena de *game over*

Esta escena será mostrada cada vez que el jugador pierda todas sus vidas. En esta escena se ha creado un canvas que incluye un fondo en negro y un texto, que mostrará *GAME OVER*. Por otra parte, se ha creado el *script RestartGame* asignado a un objeto vacío, el cual mostrará la escena durante un tiempo y posteriormente nos llevará a la escena del menú principal, que se desarrollará en las iteraciones siguientes.

Ilustración 74 Escena de GAME OVER

7.4.3 Cambio entre escenas

Para el correcto cambio entre escenas se ha decidido que los objetos como la cámara, el objeto *GameController* con el *script GameControl*, el *canvas* y el jugador no se destruirán entre escenas. Para ello, se ha creado el *script DontDestroy* que será añadido a todos los objetos que tengan que persistir entre escenas. En el método *Start* se indicará que el objeto no debe ser destruido con el método *DontDestroyOnLoad.* En cada objeto será asignada una *id* y se comprobará que el mismo objeto no esté duplicado en la escena. En caso de que esté duplicado, se eliminará el más reciente de ellos. Esta misma comprobación también se ha añadido en el *script GameControl*.



```
public class DontDestroy : MonoBehaviour
{
    public string id;
    Mensaje de Unity | 0 referencias
    private void Awake()
    {
        id = name;
    }

    // Start is called before the first frame update
    Mensaje de Unity | 0 referencias
    void Start()
    {
        for (int i = 0; i < Object.FindObjectsOfType<DontDestroy>().Length; i++)
        {
            if(Object.FindObjectsOfType<DontDestroy>()[i] != this)
            {
                if (Object.FindObjectsOfType<DontDestroy>()[i].id.Equals(id))
                {
                    Destroy(gameObject);
                }
            }
        }

        DontDestroyOnLoad(gameObject);
    }
}
```

Ilustración 75 Script DontDestroy

Se ha modificado el *script LevelLoad* para que cada vez que se vaya a cargar de nuevo un nivel, se reinicie el contador de tiempo.

Además, se ha creado un nuevo objeto que se situará al final del nivel y contendrá un *BoxCollider* con la opción *isTrigger* activada. A continuación, se ha creado el *script LevelEnd* en el cual, cuando se llegue a este objeto, activará una corutina que fijará la cámara en la posición final. También mantendrá al personaje corriendo hacia delante y sumará a nuestra puntuación el tiempo restante que quede en el nivel. Por último, cargará la siguiente escena, que será una nueva pantalla de carga para el siguiente nivel. De este objeto se ha creado un *prefab LevelEndTrigger* que, podrá colocar en cualquier nivel que desarrollemos.

Al *script GameControl* se le han añadido los métodos *restartLevel* y *restartGame*. El primero será el encargado de reiniciar al nivel, cargando de nuevo la escena de carga; el segundo se encargará de llevar al jugador a la pantalla de *game over.*

Por último, se ha creado un objeto que será la posición inicial del personaje en el nivel. Cada vez que este sea cargado, el personaje será ubicado en la posición de ese objeto. De él se ha creado un *prefab StartPosition* que se podrá colocar en los siguientes niveles.

7.4.4 Nivel 2 (1-2)

Para el nivel 2 se han usado los bloques desarrollados anteriormente y se ha creado un nuevo bloque el cual será usado en algunas estructuras del nivel 2.

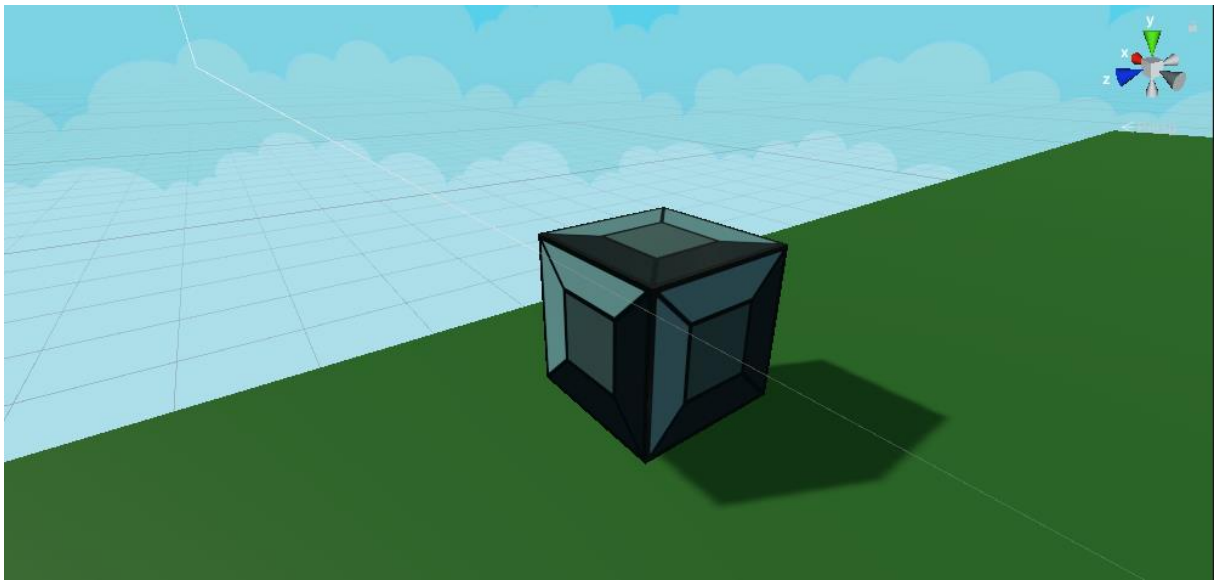


Ilustración 76 Bloque para las estructuras del nivel 2

También se ha creado una variación del bloque de ladrillo con un material distinto, para que encaje en la estética del nivel 2. Este bloque mantiene la funcionalidad del bloque de ladrillo original.

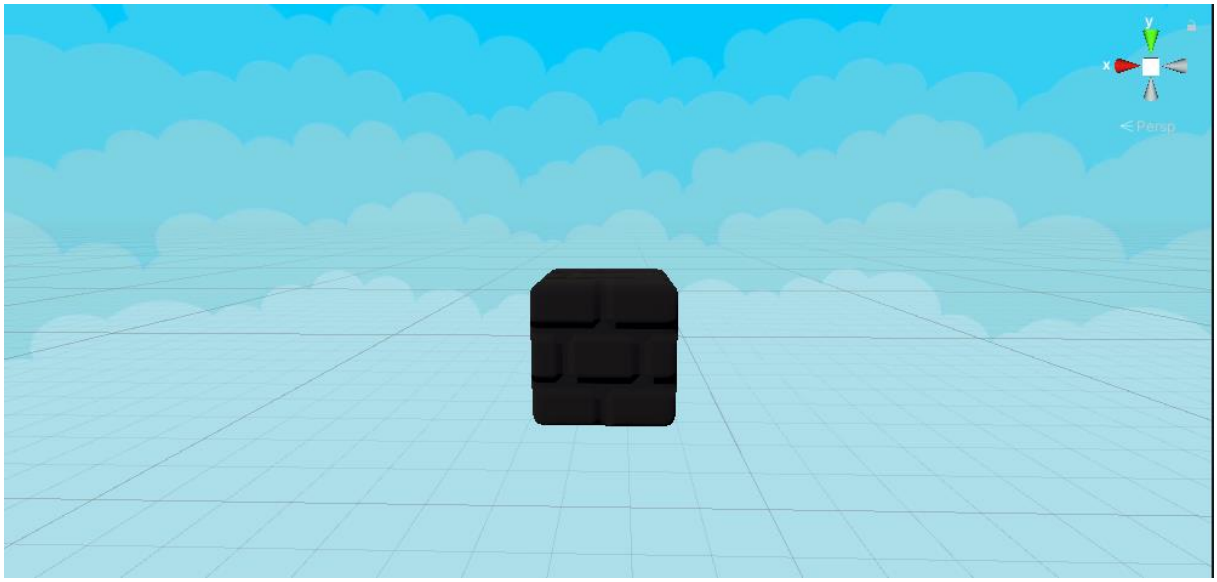


Ilustración 77 Variación del bloque de ladrillo para el nivel 2

Una vez se han creado ambos bloques y sus *prefabs* correspondientes, se podrá completar la estructura del nivel.

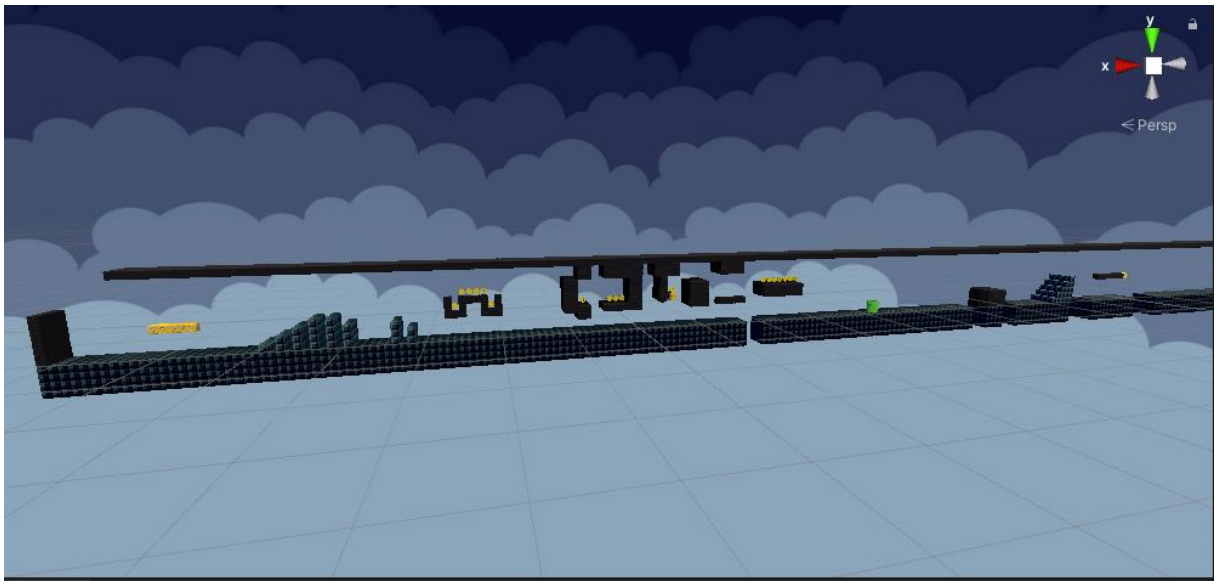


Ilustración 78 Estructura del nivel 2

Por último, se ha añadido el *prefab LevelEndTrigger y LevelDeathTrigger* desarrollados en la anterior iteración.

7.4.4.1 Planta piraña en tubería

La planta piraña en tubería es un nuevo enemigo que aparece en el nivel 2. Esta estará estática en un sitio concreto, y subirá y bajará dentro de la tubería en la que se encuentra contenida.

Para su desarrollo se ha empleado el mismo modelo que el de la tubería, usado en la tercera iteración, junto con un nuevo modelo del paquete usado anteriormente el cual contiene una planta piraña.

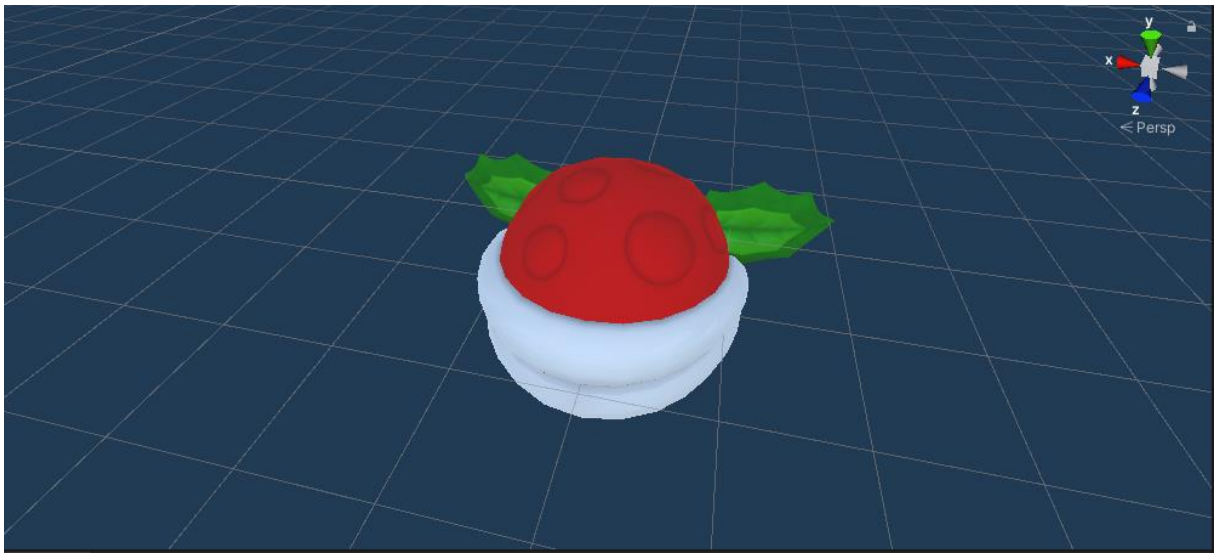


Ilustración 79 Modelo de la planta piraña

También se ha usado un modelo para el tronco de la planta piraña perteneciente al mismo paquete.

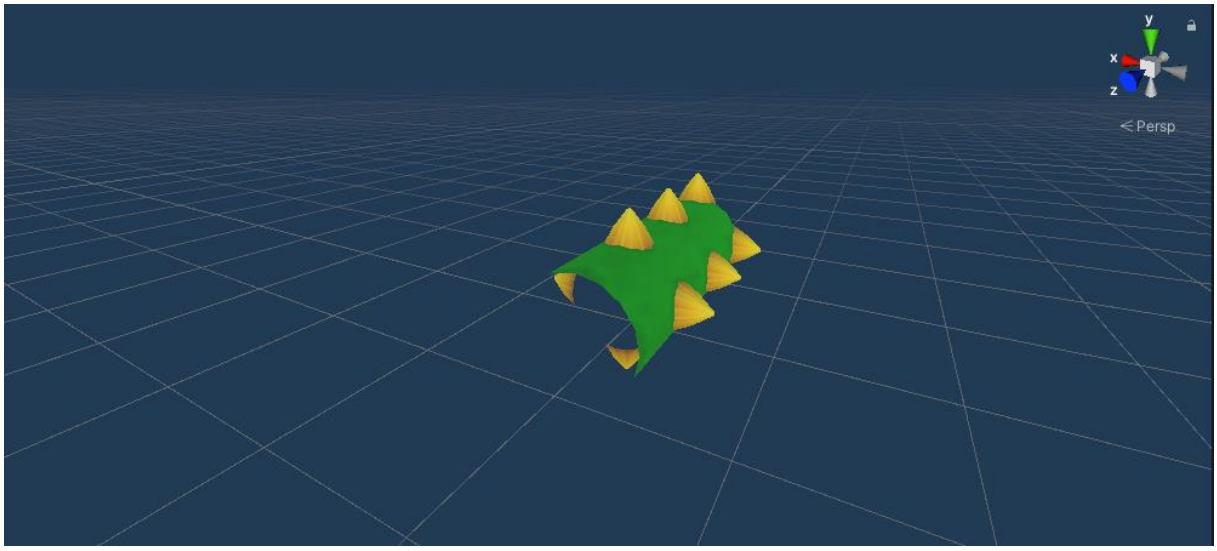


Ilustración 80 Modelo del tronco de la planta piraña

Se han colocado estos modelos junto con los de la tubería para completar el modelo final.

Ilustración 81 Objeto final de la planta piraña

Para que esta suba y baje dentro de la tubería, se ha añadido una animación a esta. También se ha añadido a la parte superior de la planta una animación para que abra y cierre la boca mientras realiza la animación anterior.

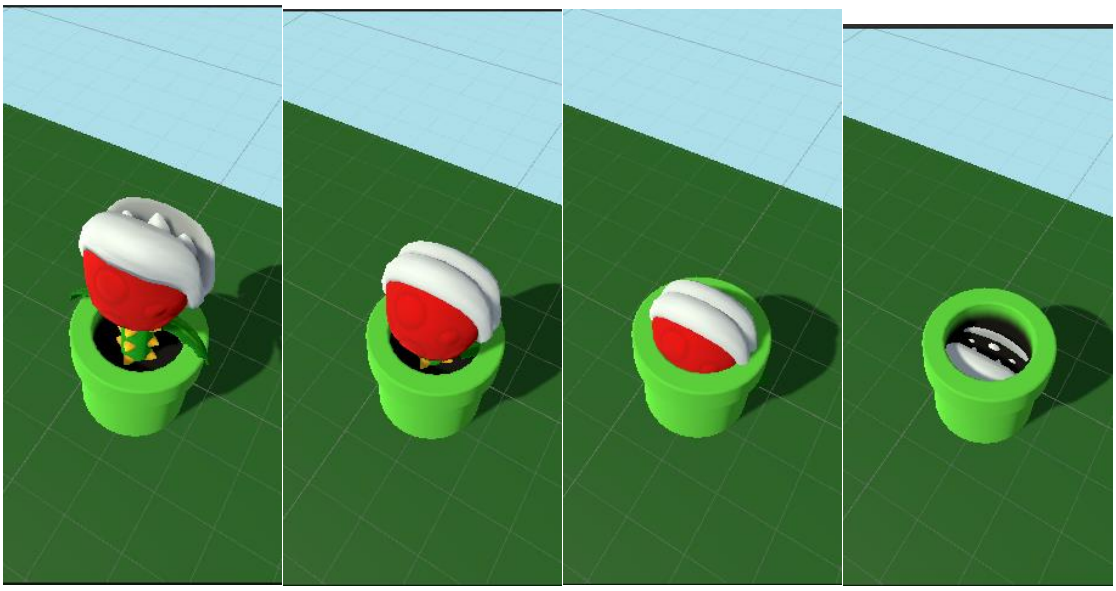


Ilustración 82 Animaciones de la planta piraña en la tubería

Se ha añadido el *script* desarrollado en iteraciones anteriores, *EnemyMakeDamage*, junto con un *Box Collider* con la opción *isTrigger* activada para que cuando el jugador entre en contacto, se decrementen sus vidas o pierda el *powerup.*

Por último, se han añadido este y el resto de los enemigos presentes en el nivel.

Ilustración 83 Enemigos en el nivel 2

7.4.4.2 Plataforma con desplazamiento vertical

Para completar el nivel 2 se ha desarrollado una plataforma. Esta estará ubicada al final del nivel 2 y se desplazará verticalmente entre dos puntos.

El modelo usado para la plataforma se ha obtenido del paquete Hello Mario Assets.

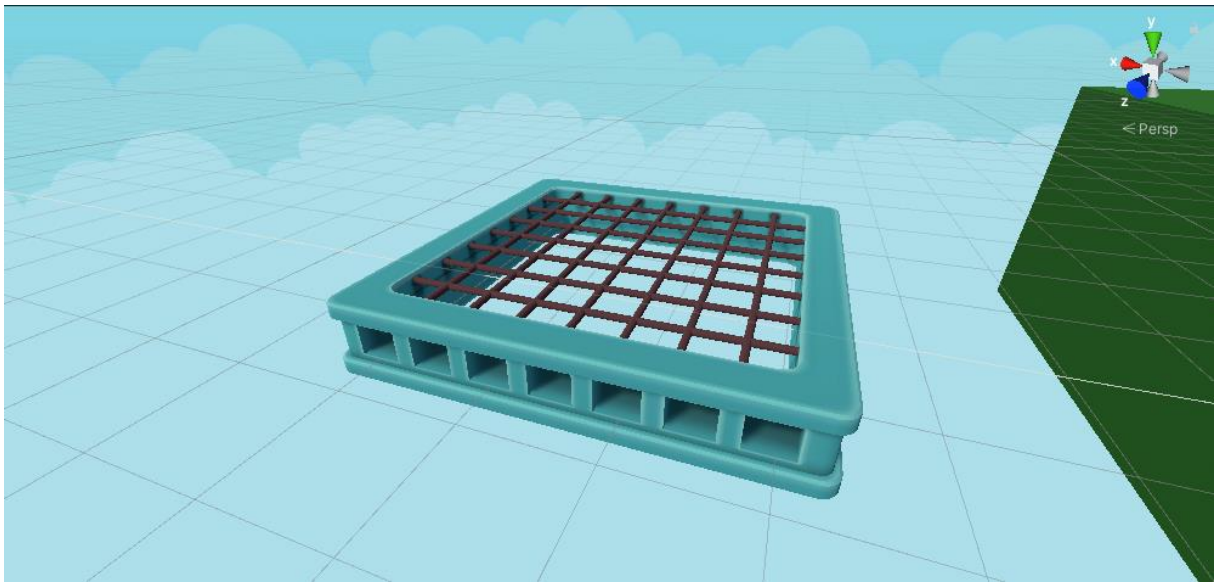


Ilustración 84 Modelo para la plataforma

Este modelo se ha añadido a un objeto, y a este mismo se le han añadido dos nuevos objetos vacíos, colocados por debajo y por encima, que representarán los puntos máximo y mínimo del movimiento de la plataforma.

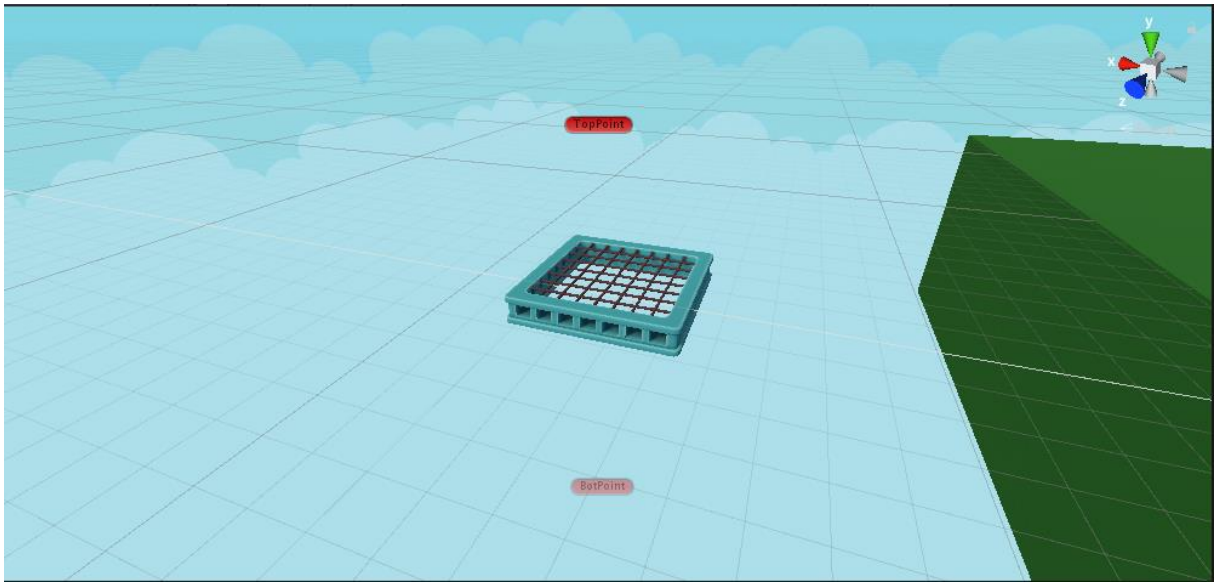


Ilustración 85 Objeto plataforma, y puntos máximo y mínimo

A continuación, se ha creado y añadido al objeto un nuevo *script* *LiftMoveVertical*. En el método *Start* se obtendrá la posición de los puntos máximo y mínimos. En el método *Update* se trasladará la plataforma hasta llegar a uno de los dos puntos y en ese momento se moverá en la otra dirección. La velocidad podrá ser ajustada mediante una variable pública que se podrá modificar desde el editor.

Por último, se han colocado los objetos en el nivel, ajustando los puntos máximos y mínimos de ambas plataformas.

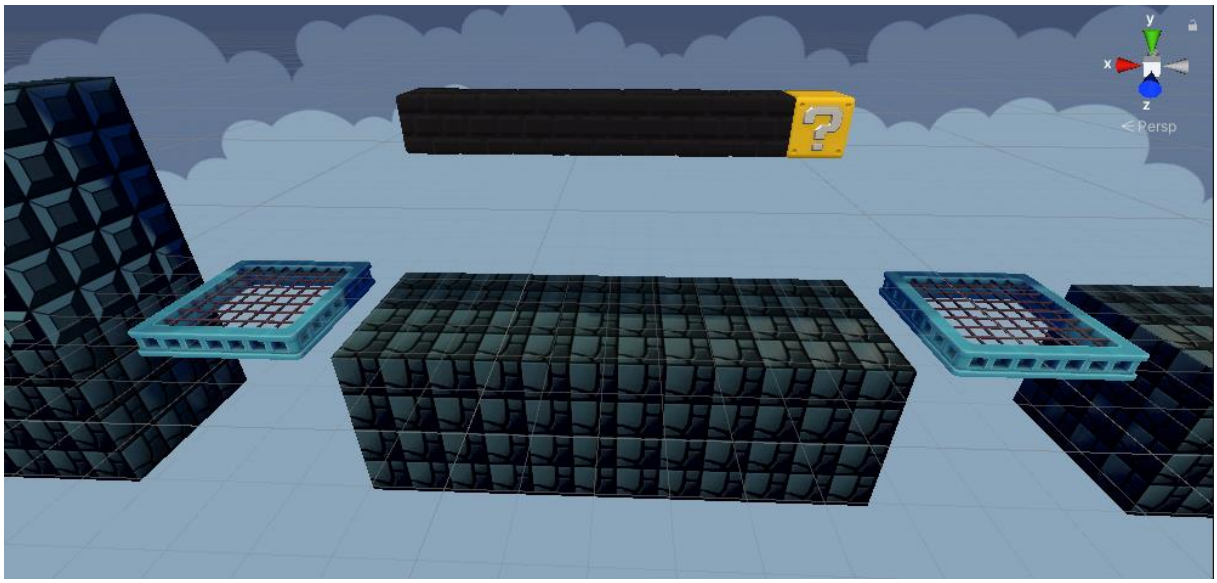


Ilustración 86 Plataformas en el nivel 2

7.4.4.3 Pantalla de carga

Se ha creado una nueva escena que será la pantalla de carga del nivel 2 y que contendrá la misma funcionalidad que la pantalla de carga desarrollada en la iteración anterior.

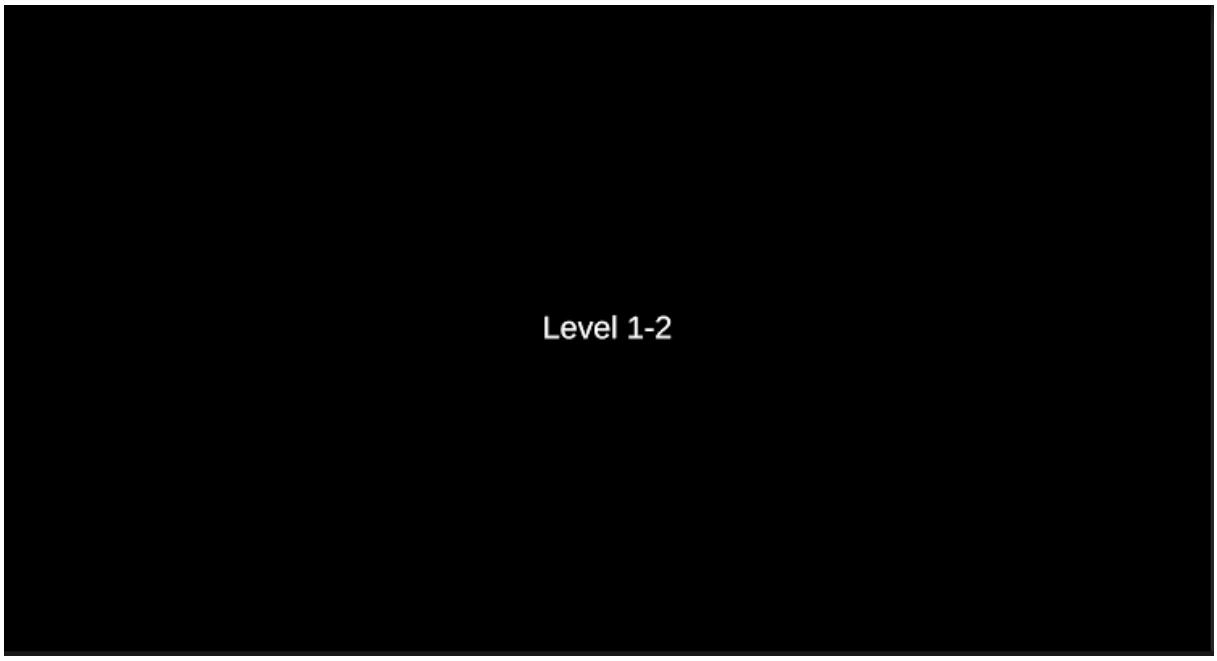


Ilustración 87 Pantalla de carga del nivel 2

En esta misma iteración se han desarrollado, además, todo el resto de las pantallas de cargas para los niveles 3 a 8. Para los niveles 4 y 8, las pantallas de carga serán diferentes al ser estos niveles jefes finales.

Ilustración 88 Pantalla de carga del nivel 4

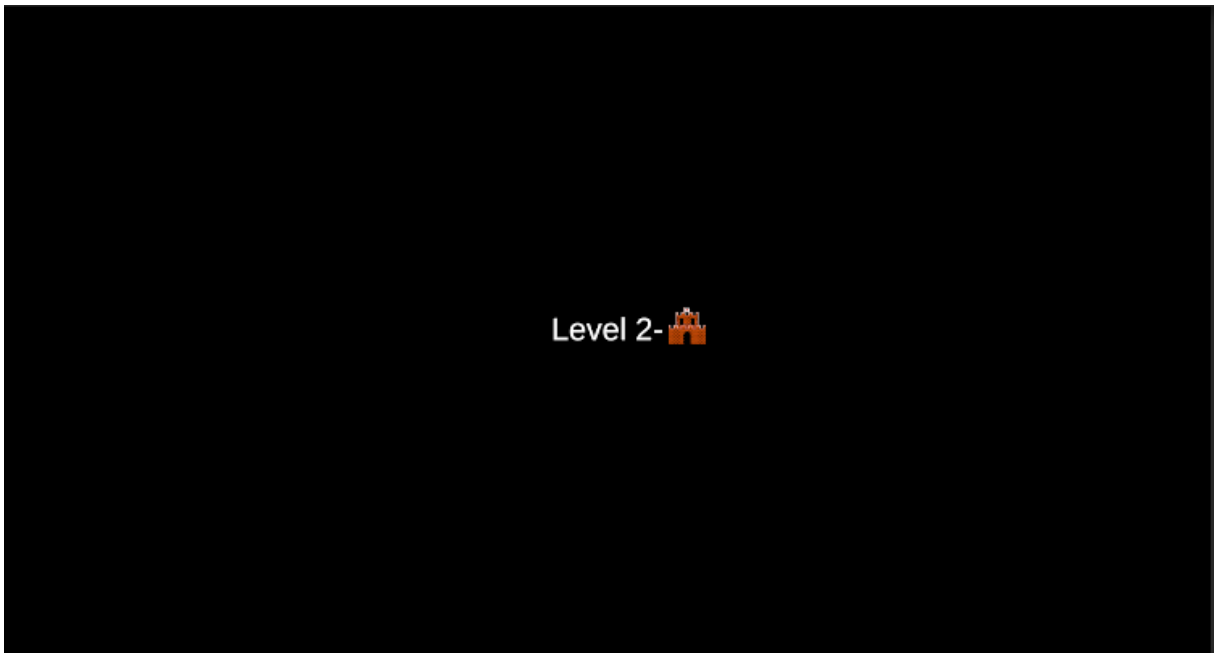


Ilustración 89 Pantalla de carga del nivel 8

7.5 Quinta iteración

En esta iteración se van a desarrollar los niveles 3 y 4, y todos los elementos necesarios para completarlos.

7.5.1 Nivel 3 (1-3)

Para el nivel 3 se han usado tres nuevos modelos de setas, obtenidos del paquete Hello Mario Assets.

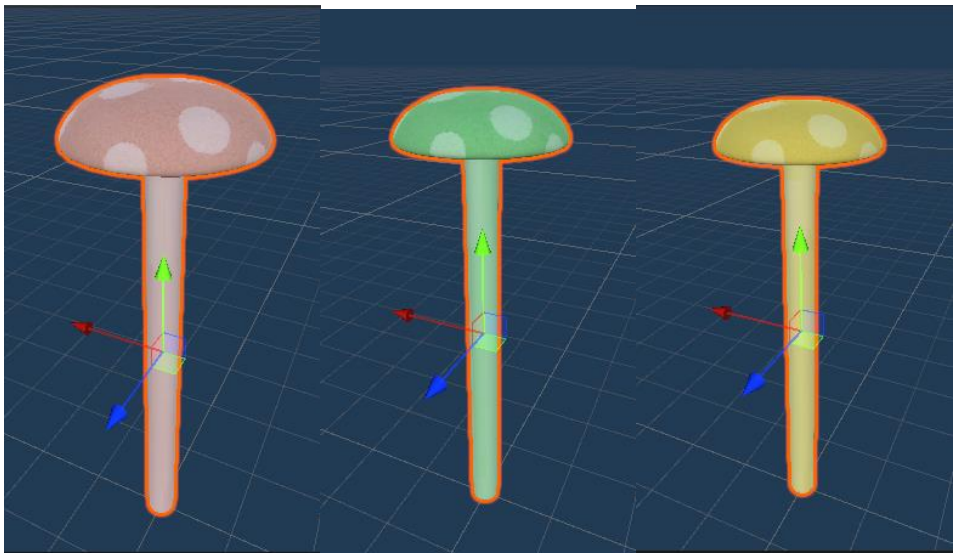


Ilustración 90 Modelos originales de las setas

Estos modelos han sido modificados para que sean planos en su superficie y el jugador pueda desplazarse por su superficie.

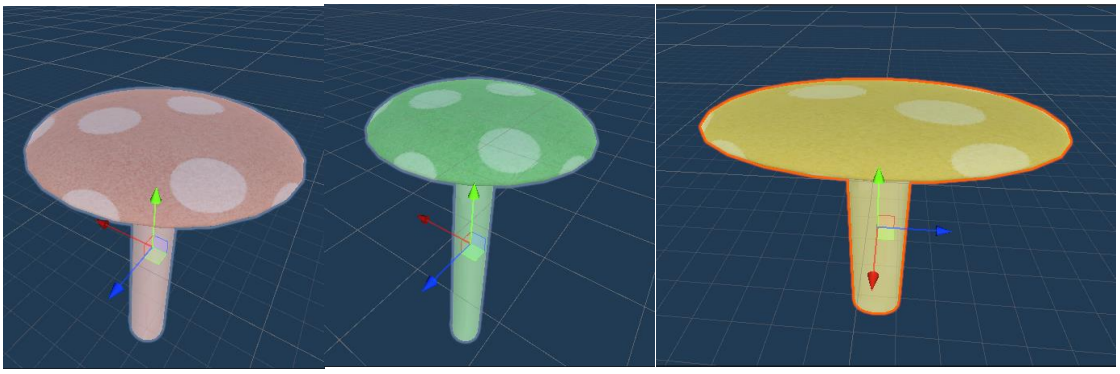


Ilustración 91 Modelos adaptados de las setas

Estas se han colocado por todo el nivel atendiendo al diseño. También se han añadido todos los bloques, enemigos y objetos al nivel.

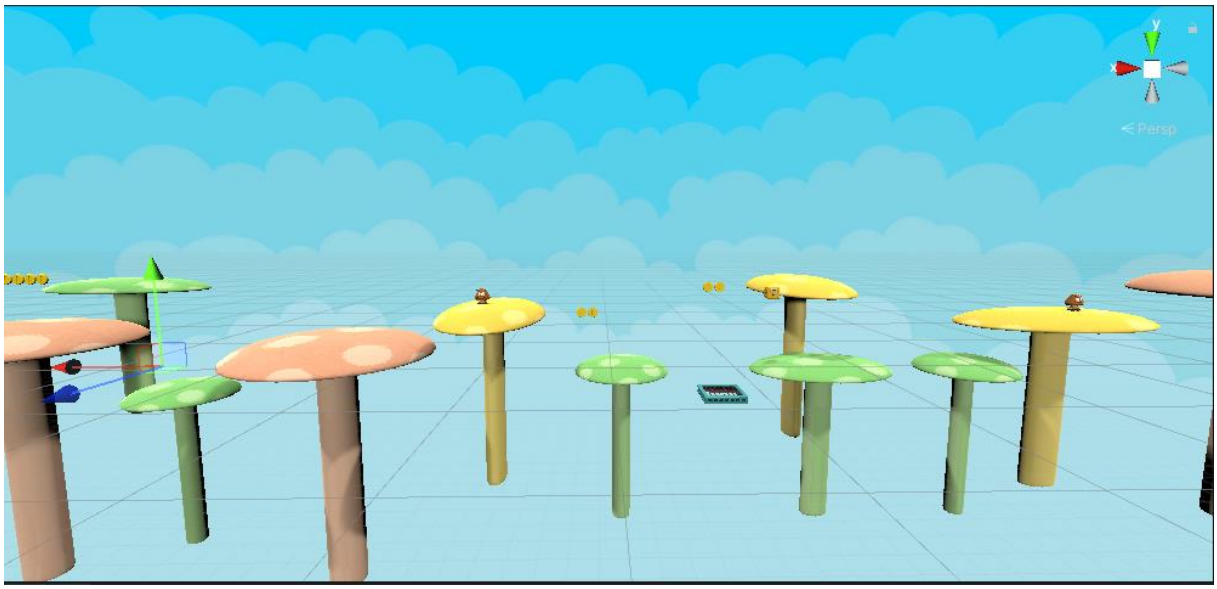


Ilustración 92 Estructura del nivel 3

Por último, se han colocado los objetos *LevelEndTrigger* y *LevelDeathTrigger*.

7.5.1.1 Koopa

El desarrollo de este enemigo se ha dividido en dos partes: la primera, cuando el enemigo aún está vivo y puede moverse libremente; la segunda, cuando este haya sido derrotado y su caparazón quede sobre el nivel.

Para la primera parte, se ha obtenido el modelo del Koopa del paquete Hello Mario Assets.

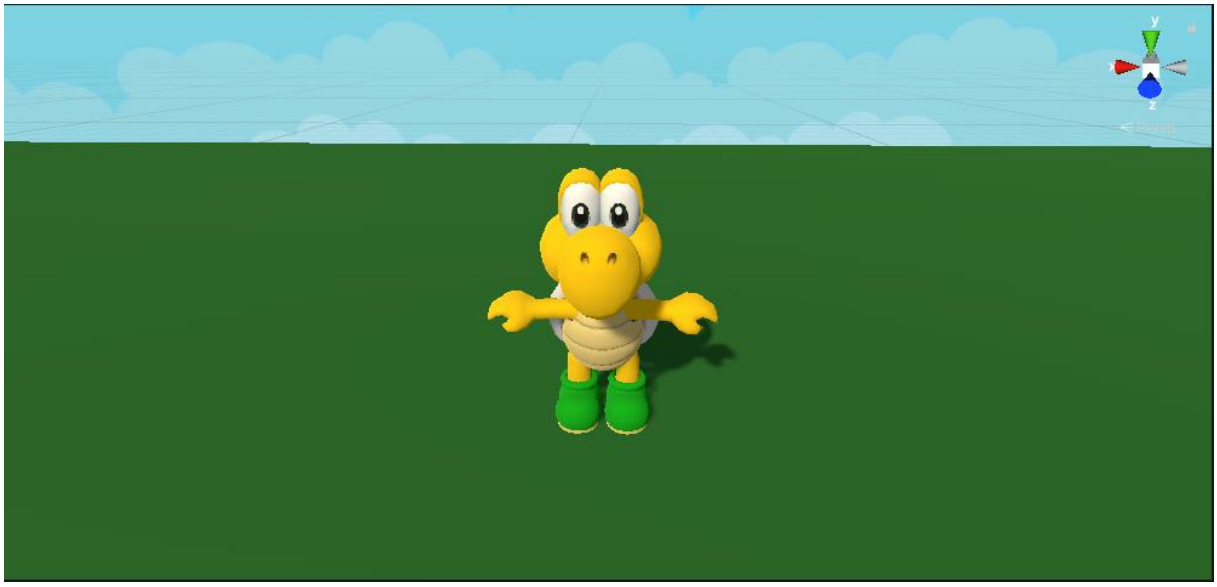


Ilustración 93 Modelo del Koopa

A continuación, se le han añadido dos *Box Collider* con la opción *isTrigger* activada.

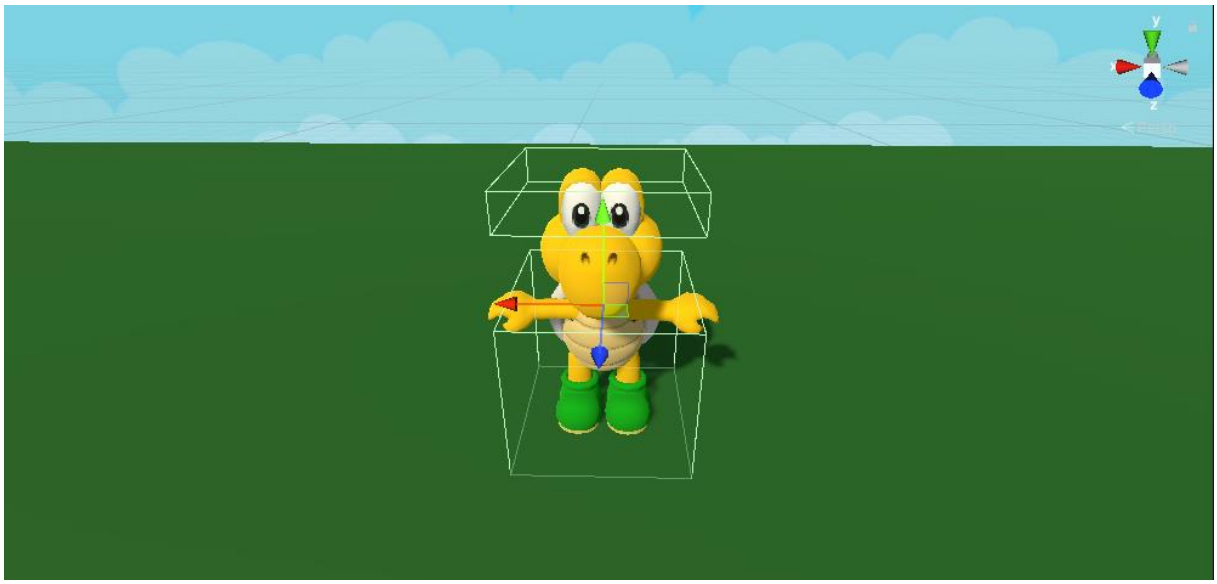


Ilustración 94 Colliders del Koopa

Al igual que el Goomba de la cuarta iteración, el *Box Collider* superior servirá para que nuestro enemigo sea derrotado; el *collider* de la parte inferior contendrá el *script EnemyMakeDamage* desarrollado anteriormente para que el enemigo pueda causar daño al jugador.

Para el *collider* superior se ha desarrollado un nuevo *script KoopaDead* el cual en el método *onTriggerEnter* esperará a un objeto con el *tag "Player".* Una vez que el jugador entre en contacto con el *collider*, se desactivaran las animaciones del enemigo, su c*ollider*, movimiento y el *script EnemyMakeDamage*. Además, se pondrá como inactivo el modelo del Koopa, y se activará el modelo que contendrá el caparazón, y que desarrollará a continuación. Por último, se le ha añadido la animación de caminar al modelo.

Para la segunda parte, se ha obtenido del paquete Hello Mario Assets el modelo de un caparazón vacío, y se le ha añadido un *Box Collider*.

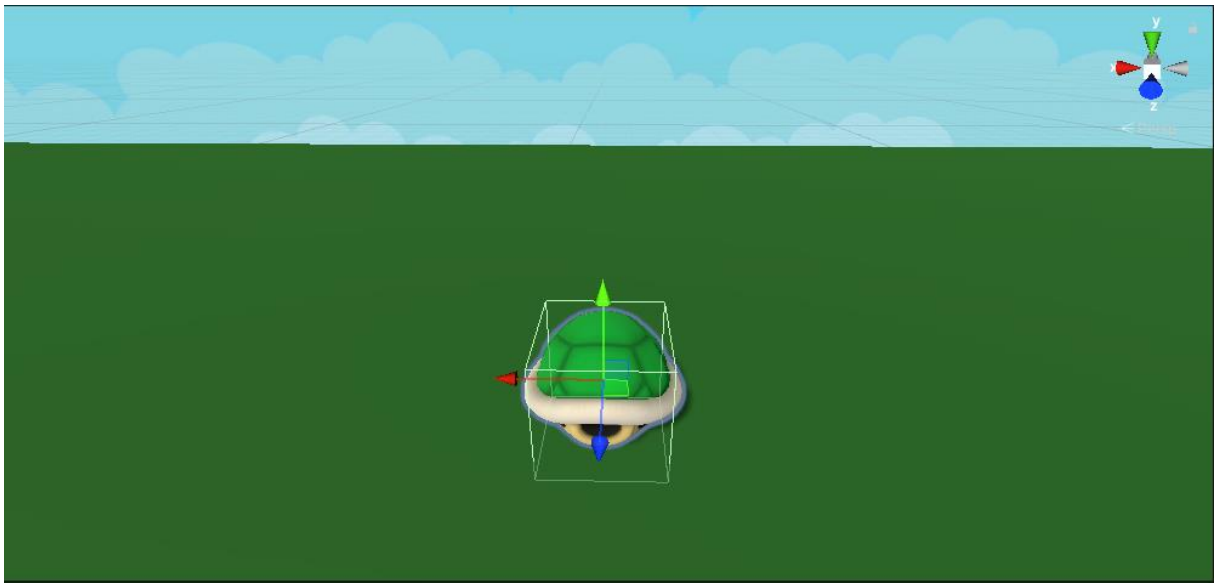


Ilustración 95 Modelo del caparazón y su collider

El caparazón deberá moverse hacia la izquierda o la derecha dependiendo de donde es golpeado. Para ello se han creado dos *Box Colliders* con la *isTrigger* activada.

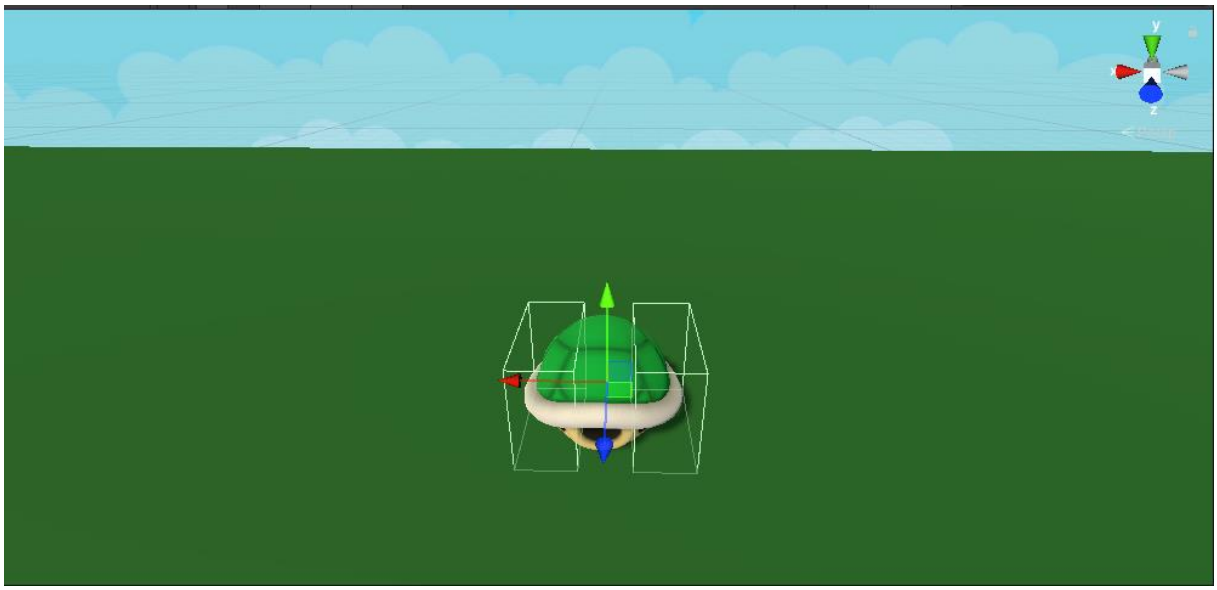


Ilustración 96 Colliders para el movimiento del caparazón

También, se ha añadido un *Rigidbody* con las *Constraints Freeze Rotation* en los ejes X, Y, Z.

A continuación, se han creado dos nuevos *scripts KoopaLeftTrigger* y *KoopaRightTrigger* que serán colocados a los dos objetos creados anteriormente. Para el *script KoopaLeftTrigger*, en el método *OnTriggerEnter* se encontrará un *switch*

el cual, dependiendo del *tag* que tenga el objeto con el que entre en el *trigger* realizará una acción u otra. Si el *tag* es *"Enemy",* destruirá el objeto; si es el jugador, el caparazón dará la vuelta y se dirigirá en la dirección contraria. Esto se realizará mediante una variable *booleana goRight* junto con una referencia a la variable *goLeft*, que estará en el script *KoopaRightTrigger*. La variable *goRight* será cambiada al valor *true*; la variable *goLeft* será cambiada al valor *false*. Por último, en el método *Update* se actualizará la velocidad del objeto en el eje X mediante el *Rigidbody* creado anteriormente. También se le aplicara gravedad en el eje Y, para que este tenga físicas y caiga al llegar al final de una superficie.

El *script KoopaRightTrigger* será exactamente igual que el descrito anteriormente, pero el caparazón se moverá en la dirección contraria.

Para terminar, se le ha añadido al Koopa el *script EnemyMove2D* desarrollado anteriormente y se han creado los dos objetos vacíos que serán los puntos entre los que se mueva el enemigo.

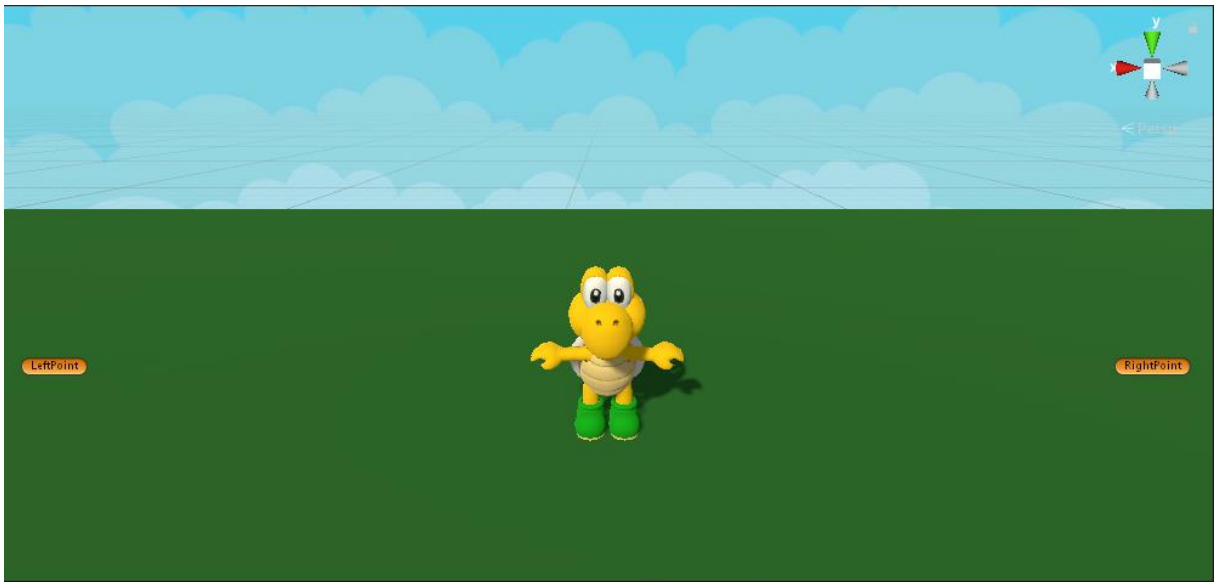


Ilustración 97 Objeto final del Koopa

Estos han sido añadidos al nivel desarrollado anteriormente, ajustando sus puntos de movimiento.

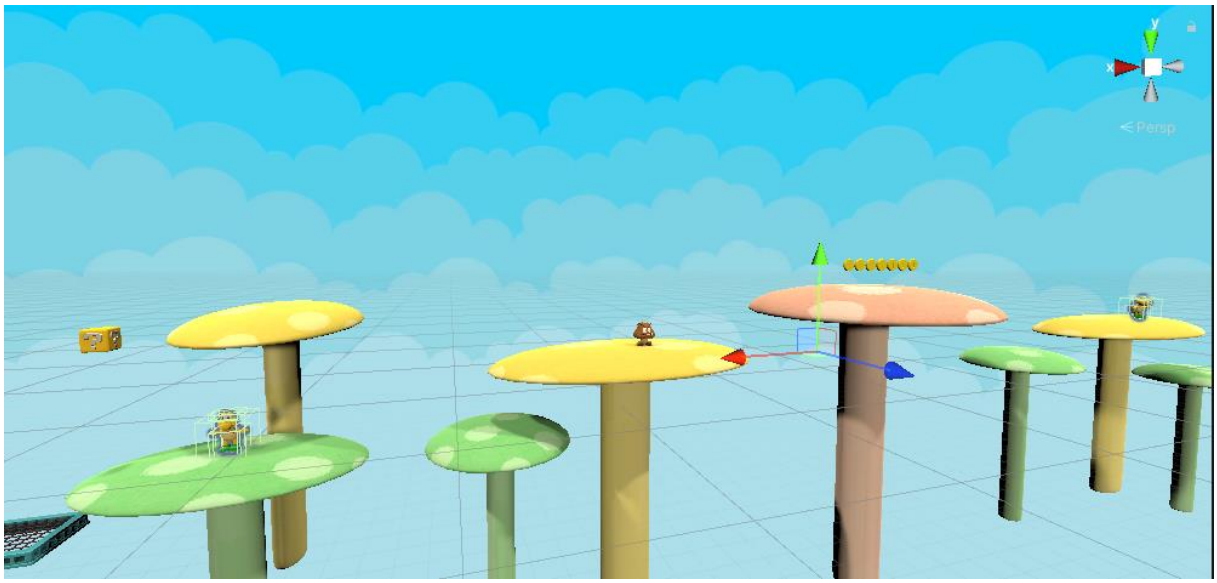


Ilustración 98 Nivel 3 con Koopa

7.5.2 Nivel 4 (1-4)

En este nivel será necesario desarrollar el jefe final y un mecanismo para su derrota, así como un final de nivel que será diferente al del resto, al ser este final del mundo. La estructura básica del nivel puede ser creada con objetos de iteraciones pasadas.

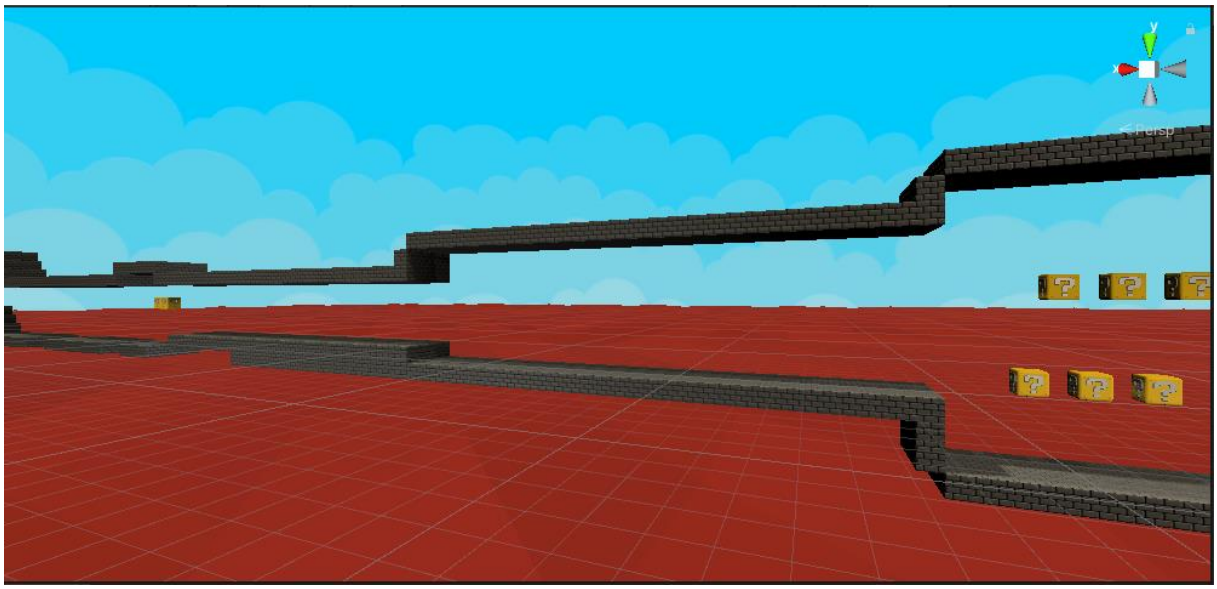


Ilustración 99 Estructura del nivel 4

Se ha modificado el prefab de océano obtenido del paquete LowPoly Water con un nuevo material color rojo para simular la lava.

7.5.2.1 Barras de fuego giratorias

Las barras de fuego giratorias son un obstáculo o enemigo que estará presente a lo largo del nivel 4. Consta de un cubo, que será la base, alrededor de la que girará una barra de fuego hecha por varias bolas de fuego.

Para el desarrollo de este objeto se ha obtenido de la Assets Store un asset llamado Epic Toon FX [15] el cual contiene sistemas de partículas estilo *cartoon*. Se ha seleccionado el sistema de partículas llamado *CartoonFireTorchRed* para usarlo como bola de fuego.

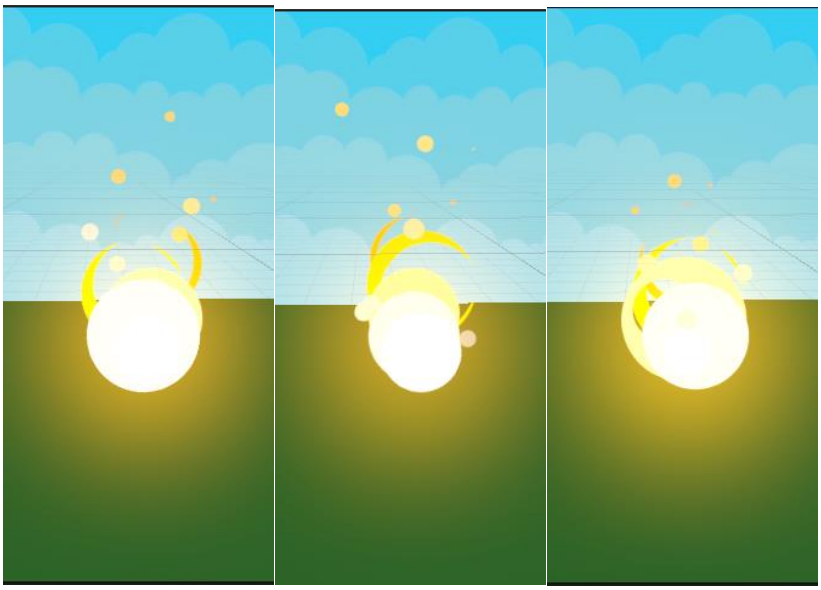


Ilustración 100 Sistema de partículas para la bola de fuego

Este sistema de particulas ha sido añadido a un objeto con un *Sphere Collider* con la opción *isTrigger* activada. También, se le ha añadido el *script* *EnemyMakeDamage* desarrollado anteriormente para que puedan dañar al jugador. Por último, se ha creado un *prefab*.

Se ha creado un objeto base sobre el que girarán las bolas de fuego. El modelo ha sido obtenido del paquete Hello Mario Assets, y este mismo modelo ha sido usado para el bloque “?”. Se le ha añadido un *Box Collider*.

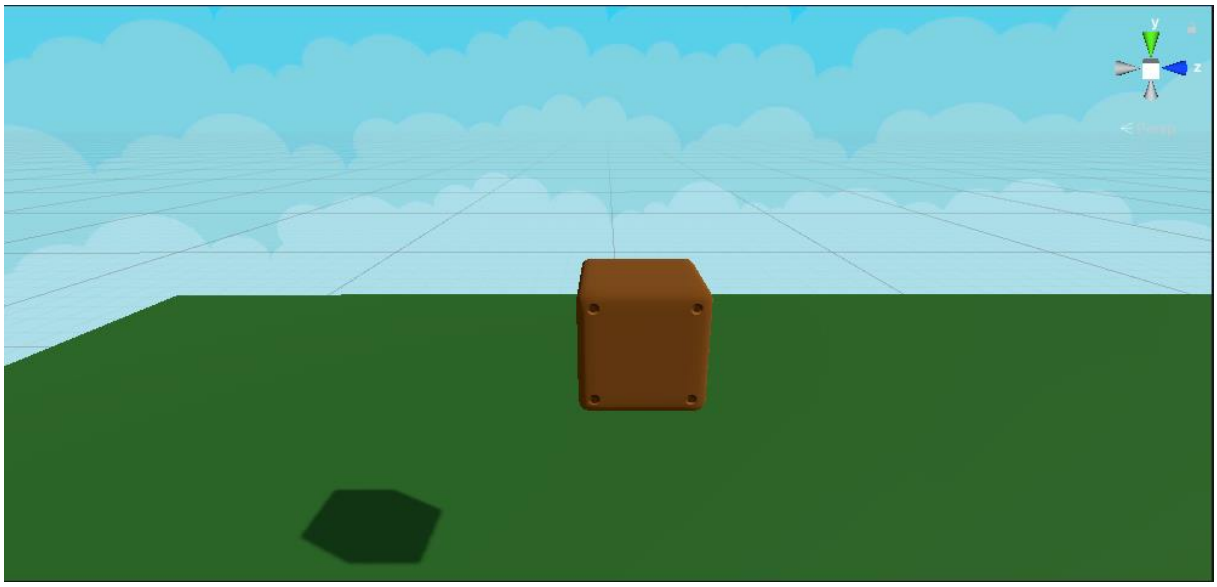


Ilustración 101 Objeto base para el objeto

A partir de este bloque se han colocado los sistemas de partículas formando una línea recta. Todas las bolas de fuego estarán contenidas en un mismo objeto.

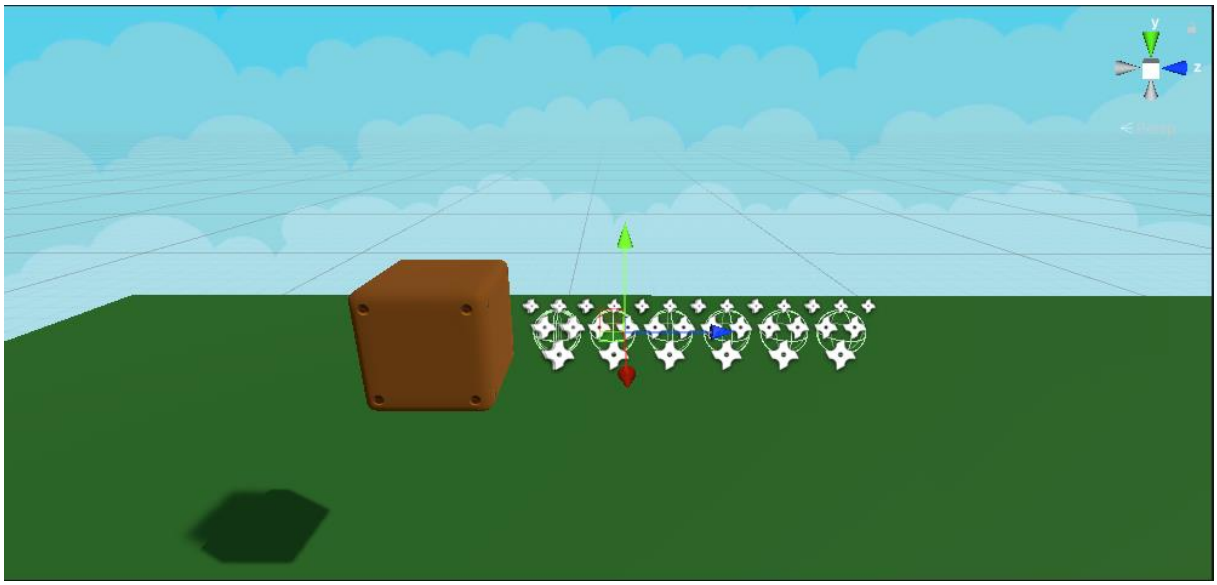


Ilustración 102 Colación de los sistemas de partículas

A este último objeto se le ha añadido un nuevo *script* llamado *RotatingFireBlockRotate* el cual en el método *Update* rotará el objeto en el eje X continuamente.

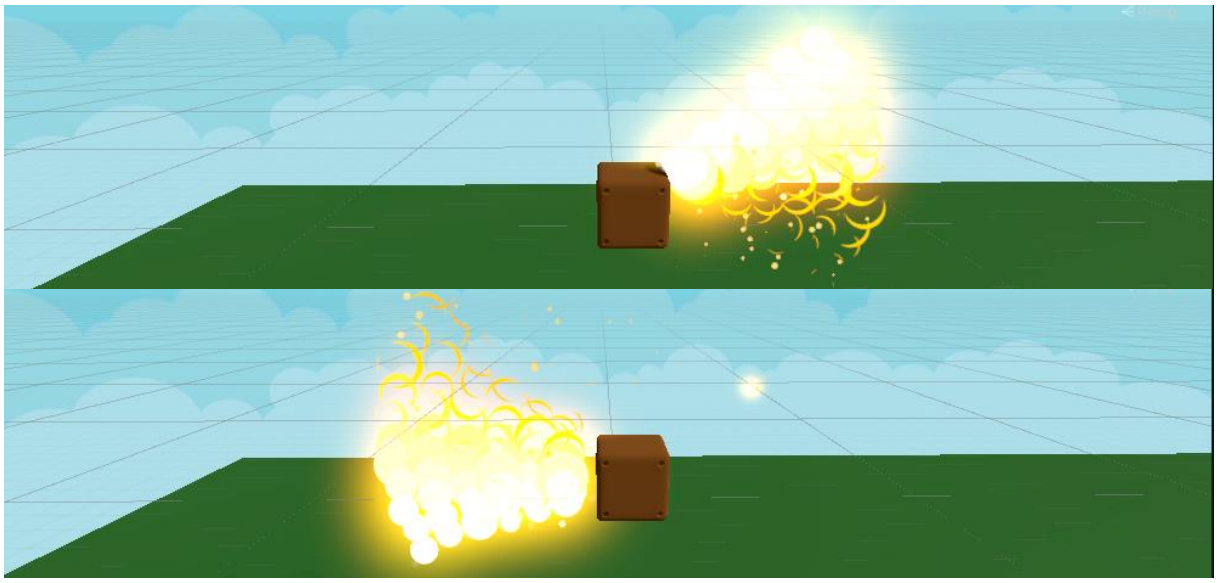


Ilustración 103 Objeto final barras de fuego giratorias

Por último, se han colocado a lo largo del nivel atendiendo al diseño.

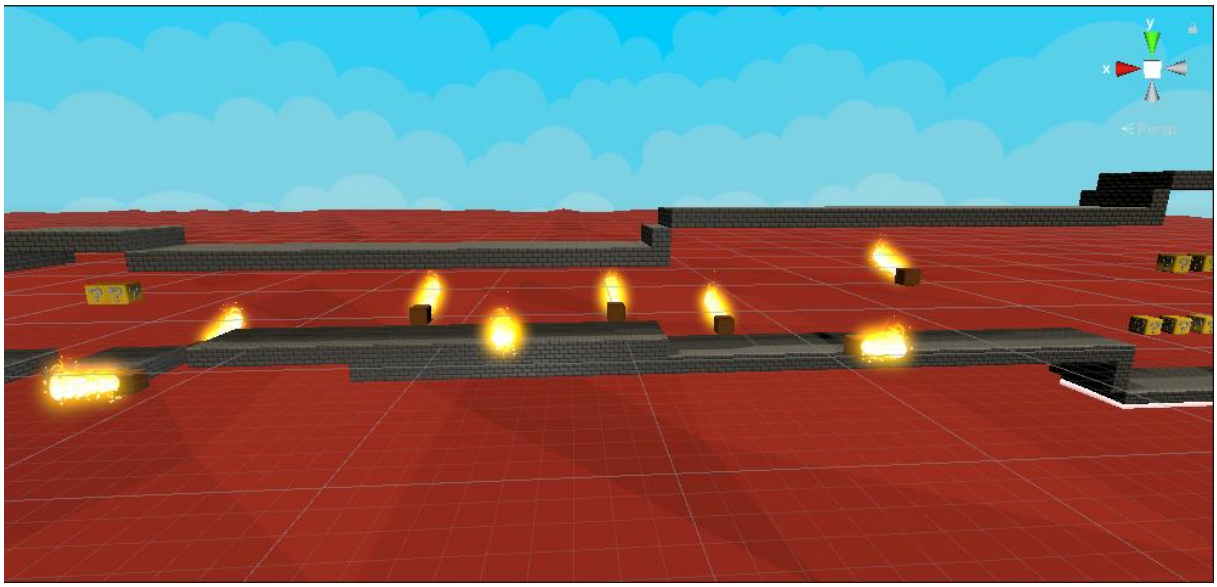


Ilustración 104 Nivel 4 con el nuevo objeto

7.5.2.2 Cañón de bolas de fuego

Este objeto estará ubicado al final del nivel, justo detrás del jefe final que se desarrollará en la última parte de esta iteración. El objeto estará formado por un cañón el cual lanzará bolas de fuego con un tiempo predefinido entre lanzamientos.

Para el cañón se ha usado un modelo del paquete Hello Mario Assets.

Ilustración 105 Modelo del cañón

Las bolas de fuego que serán lanzadas han sido desarrolladas en el apartado anterior y han sido adaptadas para que cuando sean instanciadas, estas se muevan hacia delante. Para ello se ha desarrollado el *script FireballMove* que en el método *Update* trasladará la bola de fuego hacia delante. La velocidad también podrá ser ajustada, aunque por defecto será ocho. Se ha creado un *prefab MovingFireball* de este objeto.

Se ha creado un *script FireballCannonLaunch* que en el método *Update* comprobará si hay una bola de fuego ya instanciada. Si no la hay, comenzará una corutina, la cual instanciará el *prefab* de la bola de fuego, desarrollada anteriormente, en la posición del cañón. Una vez instanciada, la bola de fuego se moverá hacia delante durante cinco segundos y, una vez transcurrido ese tiempo, se creará una nueva bola de fuego.

Por último, se han colocado en la parte final del nivel.

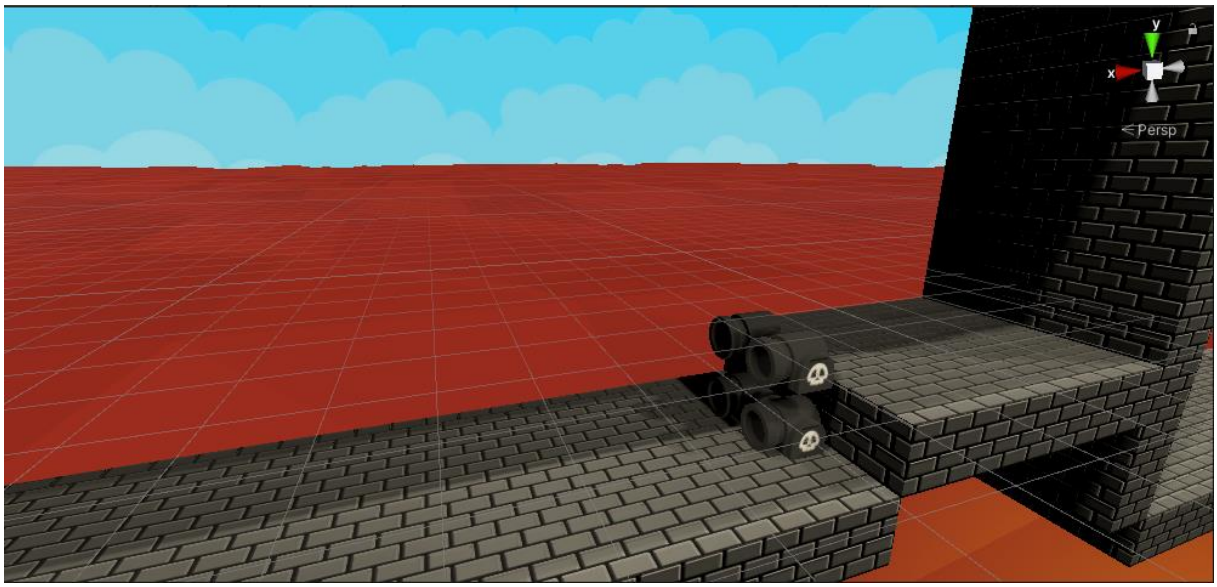


Ilustración 106 Cañones al final del nivel 4

7.5.2.3 Jefe final

Para el jefe final se ha obtenido un modelo del paquete Hello Mario Assets correspondiente a Bowsy.

Ilustración 107 Modelo de Bowsy

Al igual que en el juego original, este jefe final estará ubicado al final del nivel y su único movimiento será saltar. Para que este sea derrotado, habrá que pasar por debajo, o por encima de él, hasta llegar al botón ubicado detrás, que lo derrotará, abriendo el suelo bajo sus pies y desbloqueando el final.

El modelo del botón ha sido obtenido del paquete Hello Mario Assets.

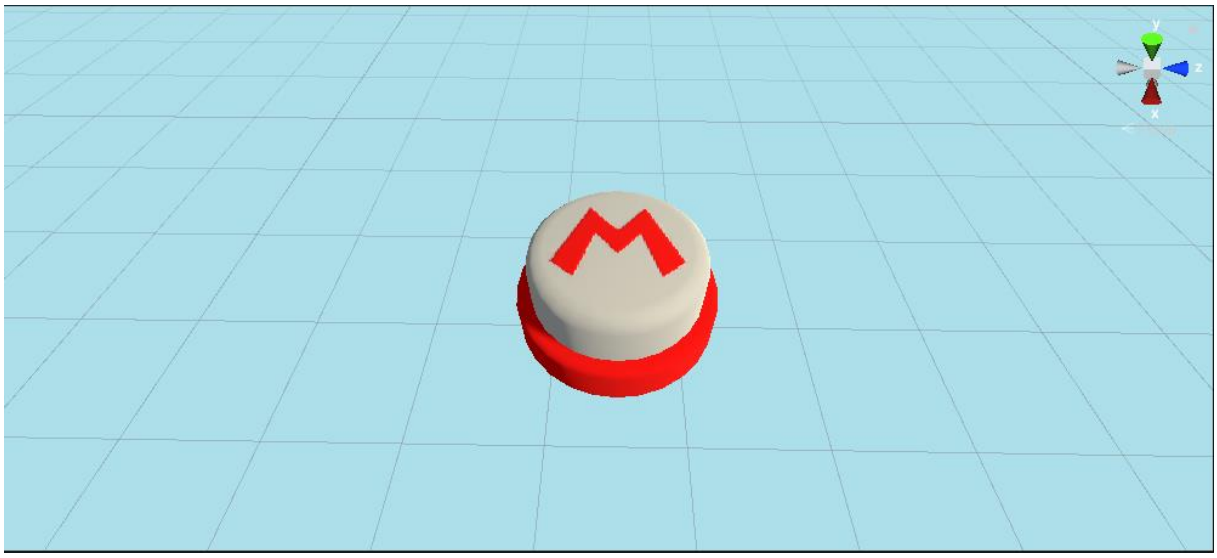


Ilustración 108 Modelo para el botón

Para iniciar el desarrollo se le ha añadido un *Animator* a Bowsy con varias animaciones: la primera, será la animación *Wait* que hará que este esté en espera; la segunda, *Jump* que realizará un salto, y será activada mediante el parámetro *booleano jump*; y la última, *Defeat* que será activada mediante la variable *booleana defeat*.

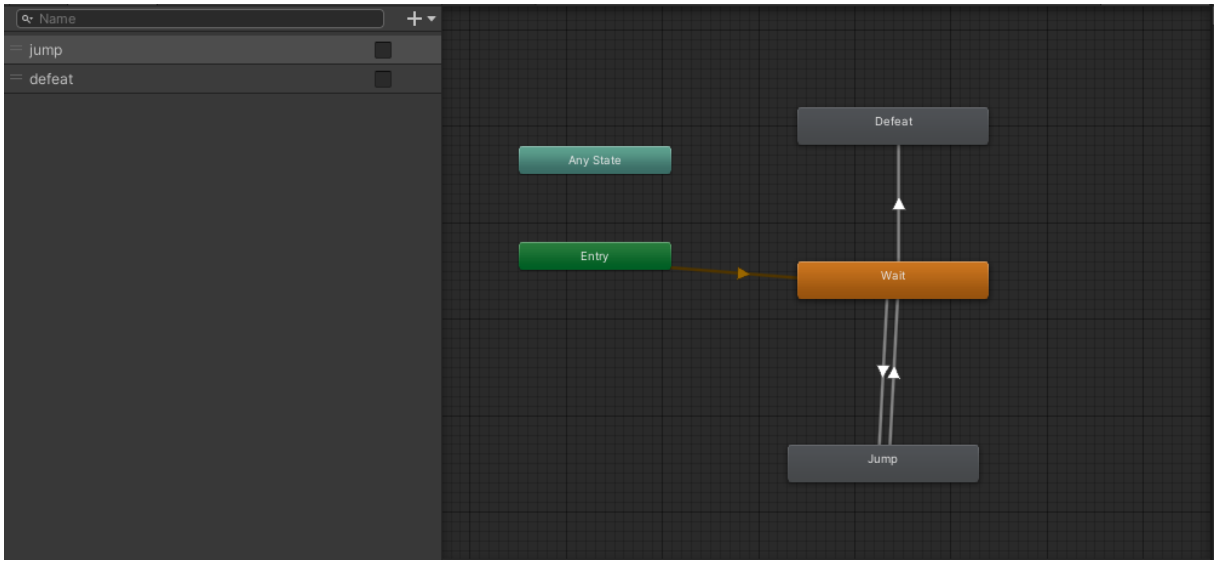


Ilustración 109 Animator de Bowsy

A continuación, se ha añadido un *Rigidbody* a Bowsy, con los parámetros por defecto, y un nuevo *script* llamado *BowserJrCharacterController*. Este *script*, en el método *Start*, buscará los componentes *Rigidbody* y *Animator*, y creará un nuevo *Vector3* con la dirección de movimiento, que aplicaremos al *Rigidbody*. También

tendremos una variable *booleana jumping* para controlar el movimiento y que nuestro enemigo salte. En el método *Update* comprobaremos si la variable *jumping* tiene el valor *false*. En ese caso, se iniciará una corutina en la cual la variable *jumping* adquiere el valor *true* y, a continuación, se activa la animación *Jump* creada anteriormente. También se aplicará al *Rigidbody* una velocidad en el eje Y. Se esperará por un tiempo predeterminado y se desactivará la animación, devolviendo también la variable *jumping* a su valor original, *false*. Por último, se le ha añadido un componente *Box Collider* con la opción *isTrigger* activada y el *script EnemyMakeDamage*.

```
public class BowserJrCharacterController : MonoBehaviour
{
    private Animator animator;
    private Rigidbody rb;
    private bool jumping = false;

    private Vector3 moveDirection;
    // Start is called before the first frame update

    void Start()
    {
        animator = GetComponentInChildren<Animator>();
        rb = GetComponent<Rigidbody>();
        moveDirection = new Vector3(0, 1, 0);
    }

    // Update is called once per frame

    void Update()
    {
        if (!jumping)
        {
            StartCoroutine(jump());
        }
    }


    private IEnumerator jump()
    {
        jumping = true;
        animator.SetBool("jump", true);
        yield return new WaitForSeconds(1f);
        rb.velocity = new Vector3(0, 10, 0);
        yield return new WaitForSeconds(2f);
        animator.SetBool("jump", false);
        jumping = false;
    }
}
```

Ilustración 110 Script BowserJrCharacterController

Para que Bowsy pueda ser derrotado, se ha creado y añadido al botón el *script BowserJrDeadTrigger* y un *Box Collider* con la opción *isTrigger* activada.

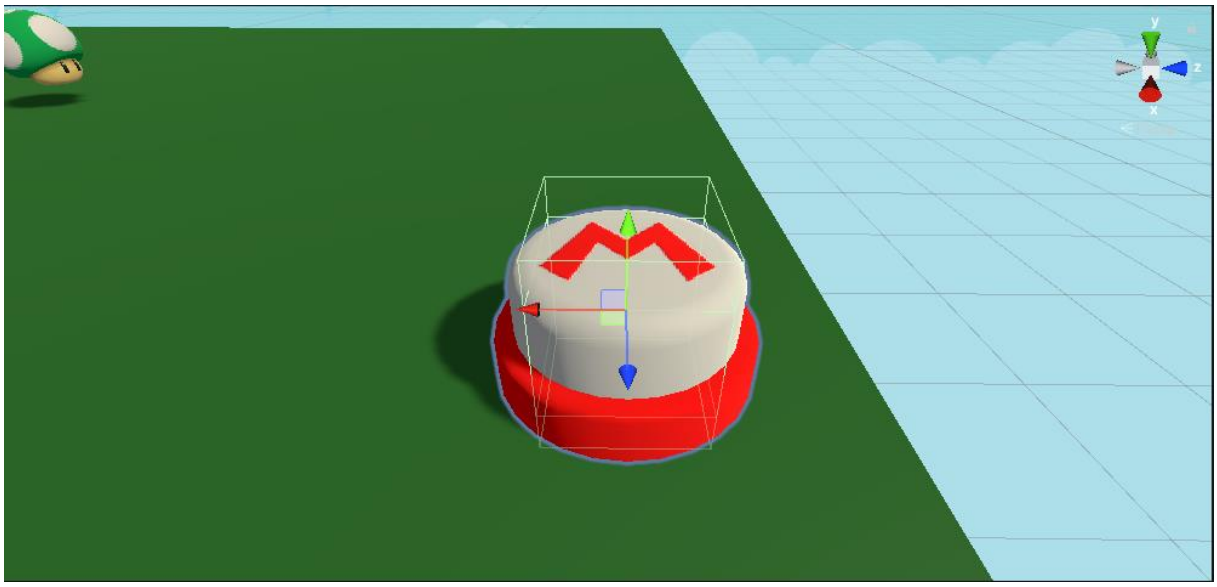


Ilustración 111 Botón junto con el trigger

También, se ha modificado el modelo del botón para que tenga dos posiciones: una que se corresponde a la posición original, y otra la posición cuando este es activado.

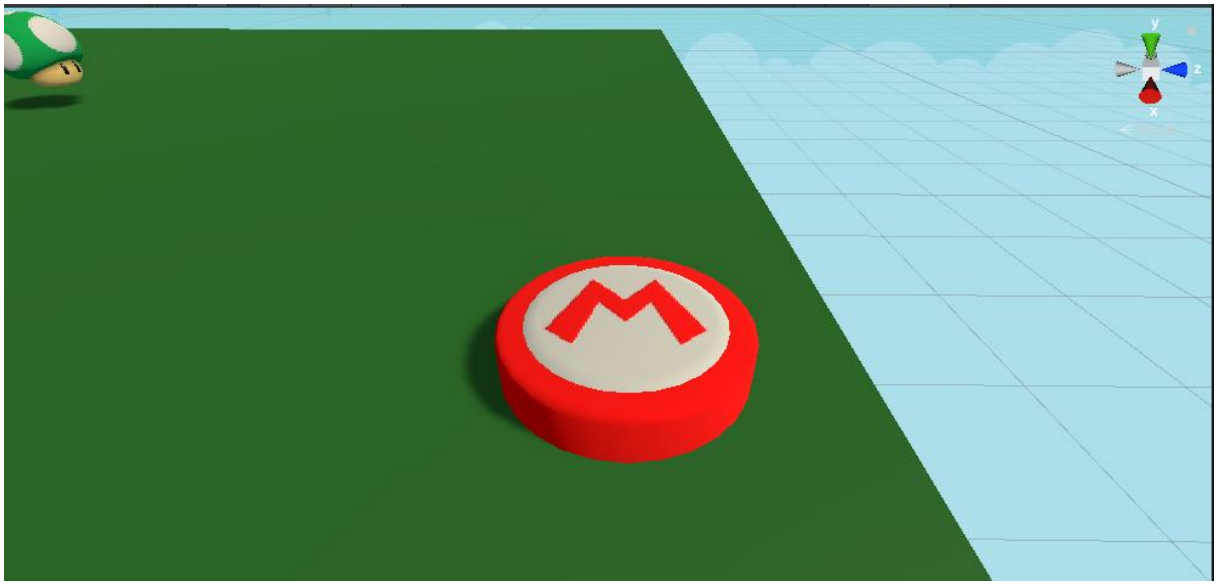


Ilustración 112 Modelo del botón activado

Continuando con el *script BowserJrDeadTrigger*, se ha implementado el método *OnTriggerEnter* en el cual esperará al objeto con el *tag "Player"* e iniciará la corutina en la que se cambiará al modelo del botón activado creado anteriormente. También, desactivará en intervalos de 0.5 segundos la plataforma sobre la que se ubique el jefe.

```
public class BowserJrDeadTrigger : MonoBehaviour
{
    public GameObject switchOn;
    public GameObject switchOff;

    public GameObject explosionPrefab;
    public GameObject bossModelObject;

    public GameObject[] ground;
    public GameObject endBarrier;

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag.Equals("Player"))
        {
            StartCoroutine(wait());
        }
    }

    private IEnumerator wait()
    {
        switchOn.SetActive(false);
        switchOff.SetActive(true);
        for (int i = 0; i < ground.Length; i++)
        {
            ground[i].SetActive(false);
            yield return new WaitForSeconds(0.5f);
        }

        transform.parent.GetComponentInChildren<BowserJrCharacterController>().enabled = false;
        transform.parent.GetComponentInChildren<Animator>().SetBool("jump", false);
        transform.parent.GetComponentInChildren<Animator>().SetBool("defeat", true);

        yield return new WaitForSeconds(2.4f);

        explosionPrefab.SetActive(true);
        bossModelObject.SetActive(false);

        endBarrier.SetActive(false);
    }
}
```

Ilustración 113 Script BowserJrDeadTrigger

Para probar el funcionamiento, ubicaremos a Bowsy y el botón desarrollado en el nivel 4.

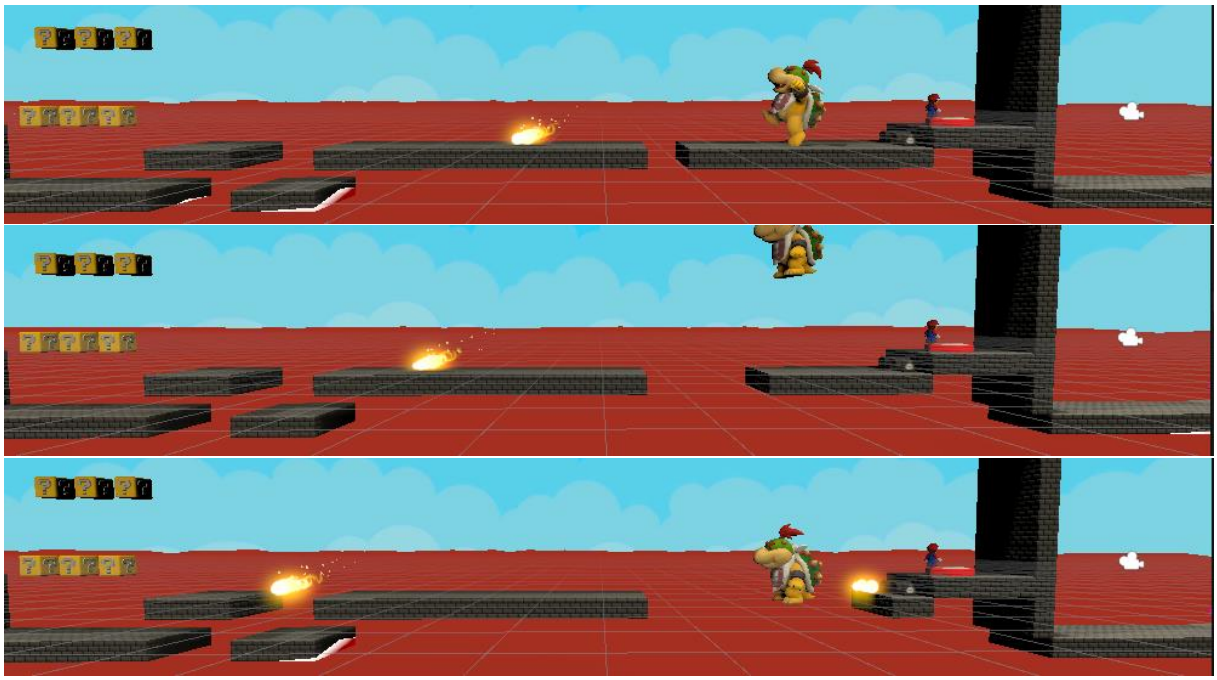


Ilustración 114 Animación para la plataforma final

A continuación, desactivará el *script BowserJrCharacterController* y activará la animación *Defeat*. Una vez que se complete la animación, se instanciará un nuevo sistema de partículas, obtenido del paquete EpicToon FX.

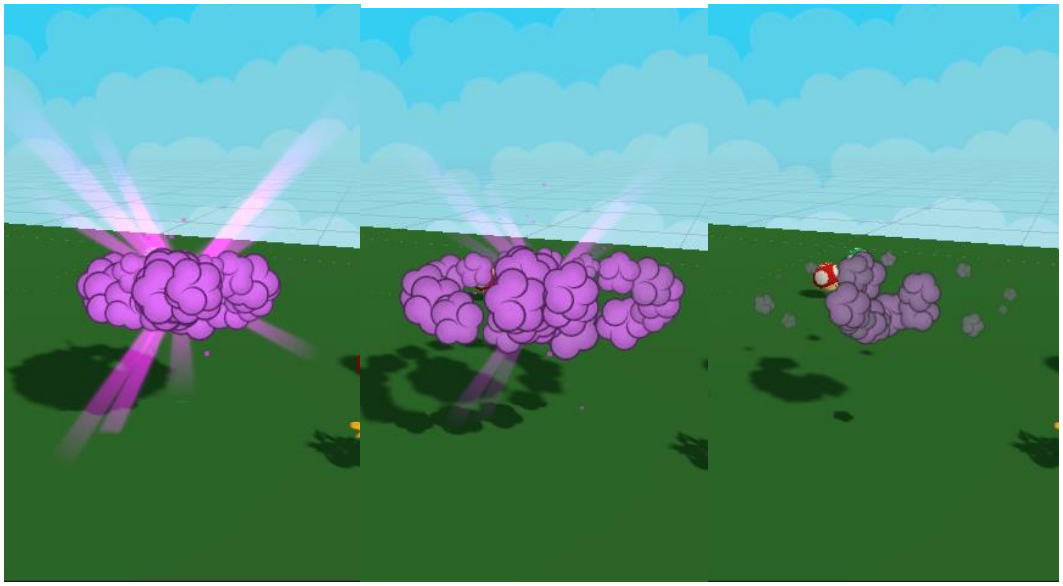


Ilustración 115 Sistema de partículas para el jefe final

Por último, se desaparecerá el modelo de Bowsy y la pared del final del nivel, desbloqueando la zona final del nivel.

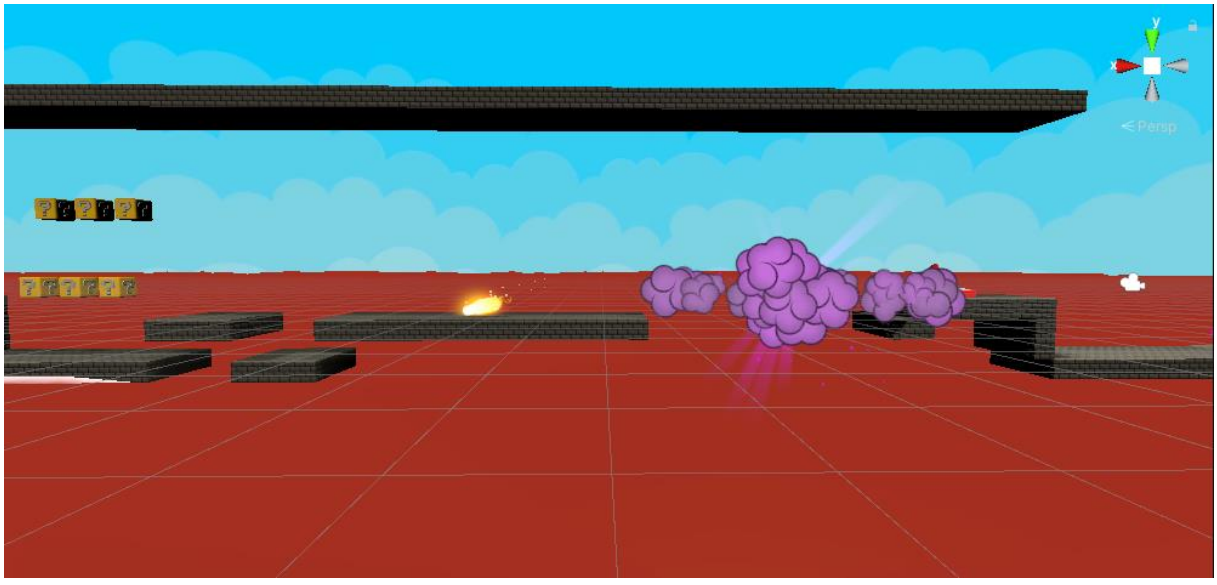


Ilustración 116 Estado final del nivel 4

Para la zona final del nivel se ha creado un objeto vacío que mediante un *Box Collider* con la opción *isTrigger* activada, desencadenará el final del nivel. Este estará ubicado justo detrás de la pared final del nivel y será completamente invisible.

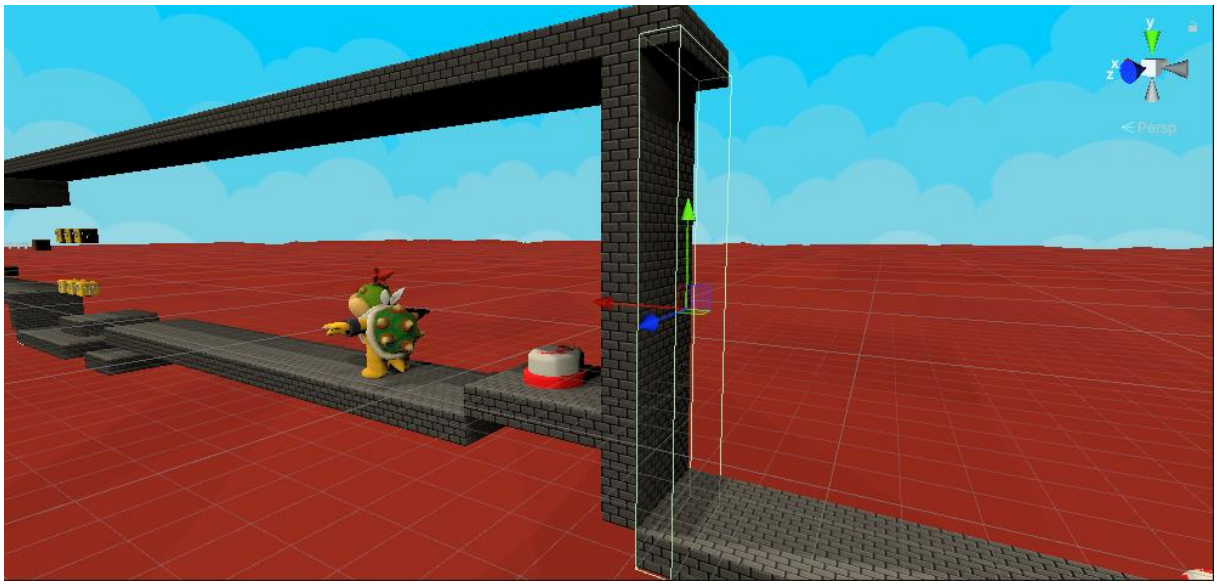


Ilustración 117 Trigger para el final del nivel 4

Contendrá el *script CastleLevelEnd*, en el cual en el método *OnTriggerEnter* esperará a un objeto con el *tag "Player".* Una vez el jugador active el *trigger*, se detendrá el contador de tiempo, y una variable *booleana end* cambiará su valor a true. En el método *Update* se iniciará una corutina para cambiar de nivel, una vez que la variable *end* tenga el valor *true*. Esta corutina añadirá la puntuación correspondiente al *score* y, tras seis segundos, se cargará la siguiente escena.

La zona final contendrá un Toad que mediante el *script ActivateEndDialog* activará un bocadillo de texto. Esto será realizado en la función *OnTriggerEnter*, activando el objeto que contiene el bocadillo de texto. Este script será añadido al *trigger* del final.

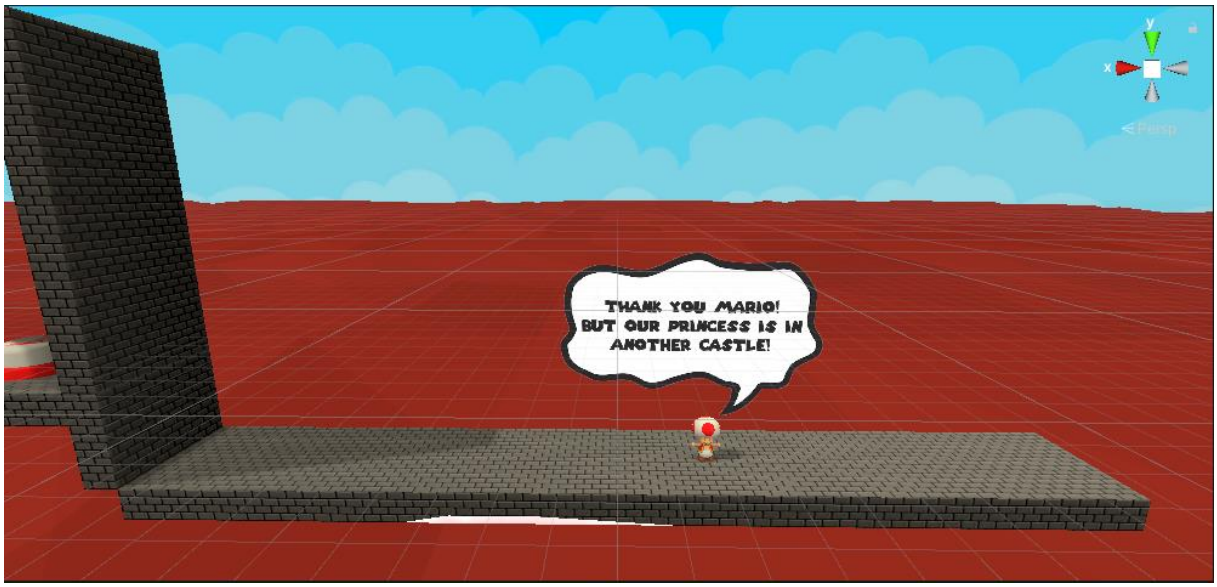


Ilustración 118 Habitación final, Toad y bocadillo de texto

7.6 Sexta iteración

En esta iteración se desarrollarán los niveles 5 y 6, así como todos los elementos necesarios para completar los niveles. También se desarrollará el sistema de sonido, añadiendo la música y efectos de sonido a todo el juego.

7.6.1 Nivel 5 (2-1)

Para el desarrollo de este nivel se ha creado una nueva plataforma que estará presente en la segunda mitad del nivel, en su parte superior. Para ello se ha usado un modelo del paquete Hello Mario Assets.

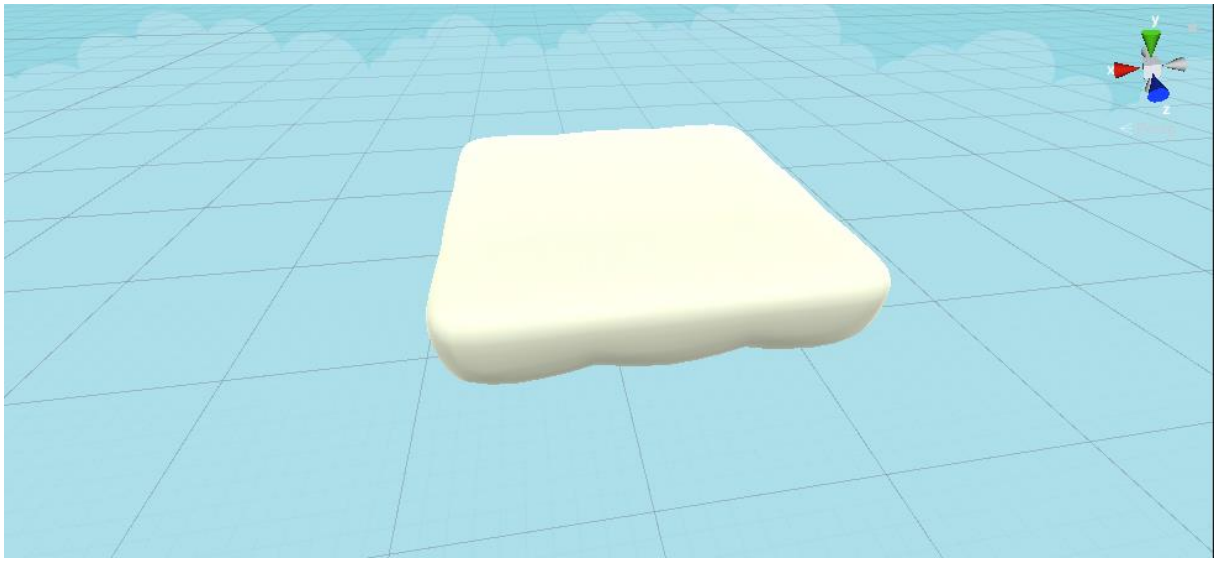


Ilustración 119 Modelo de la plataforma de nube

A esta se le ha añadido un componente *Box Collider*. A continuación, se ha creado el nivel en base al diseño. Se han reutilizado los bloques, enemigos y objetos creados en las iteraciones pasadas.

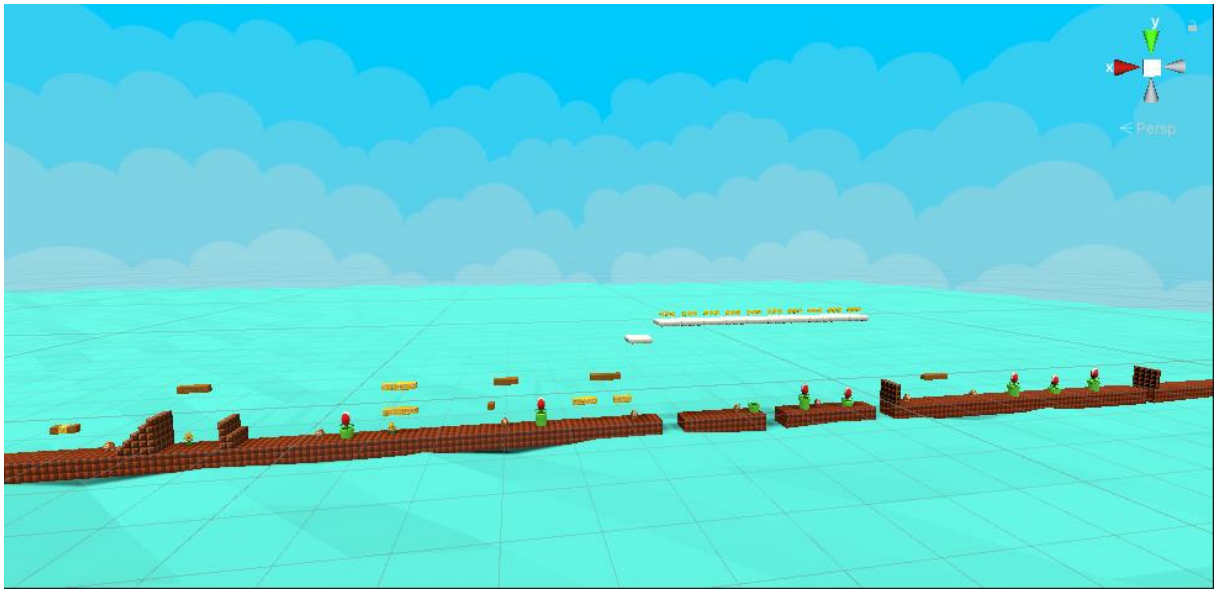


Ilustración 120 Estructura del nivel 5

Por último, se ha creado una zona de bonus a la que se accederá por la tubería ubicada en la mitad del nivel. Esta, al igual que la del nivel 1, contendrá monedas que podrán ser recogidas por el jugador. También se han añadido los objetos *LevelEndTrigger* y *LevelDeathTrigger*.

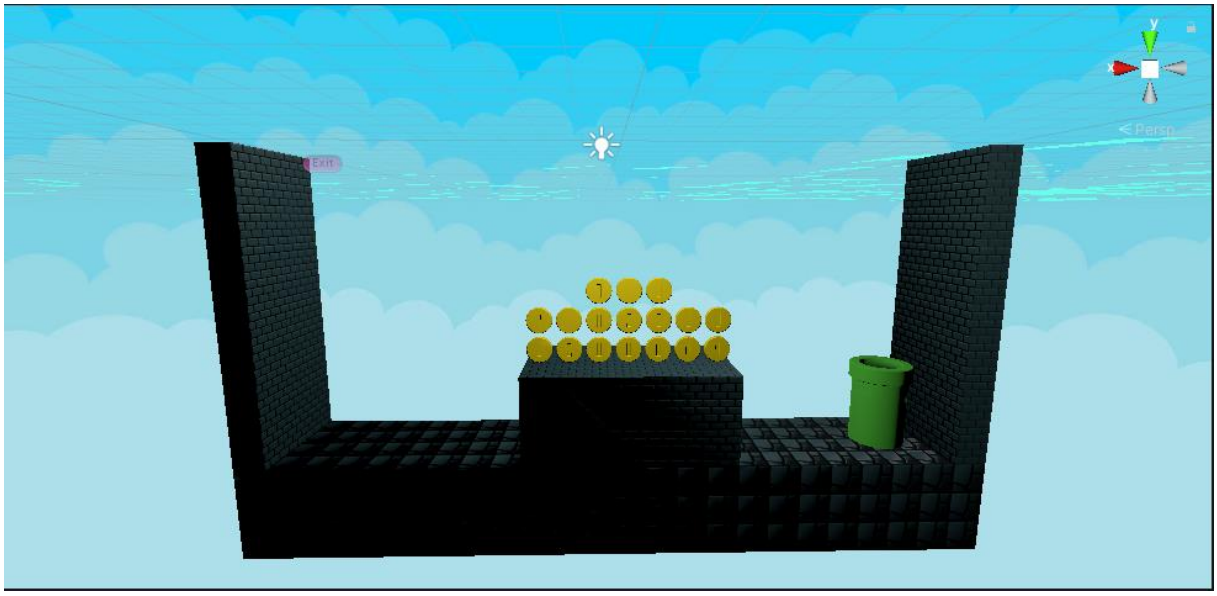


Ilustración 121 Zona de bonus del nivel 5

7.6.2 Nivel 6 (2-2)

Para el desarrollo de las estructuras del nivel 6 se han reutilizado los bloques, objetos y enemigos creados en las iteraciones anteriores.

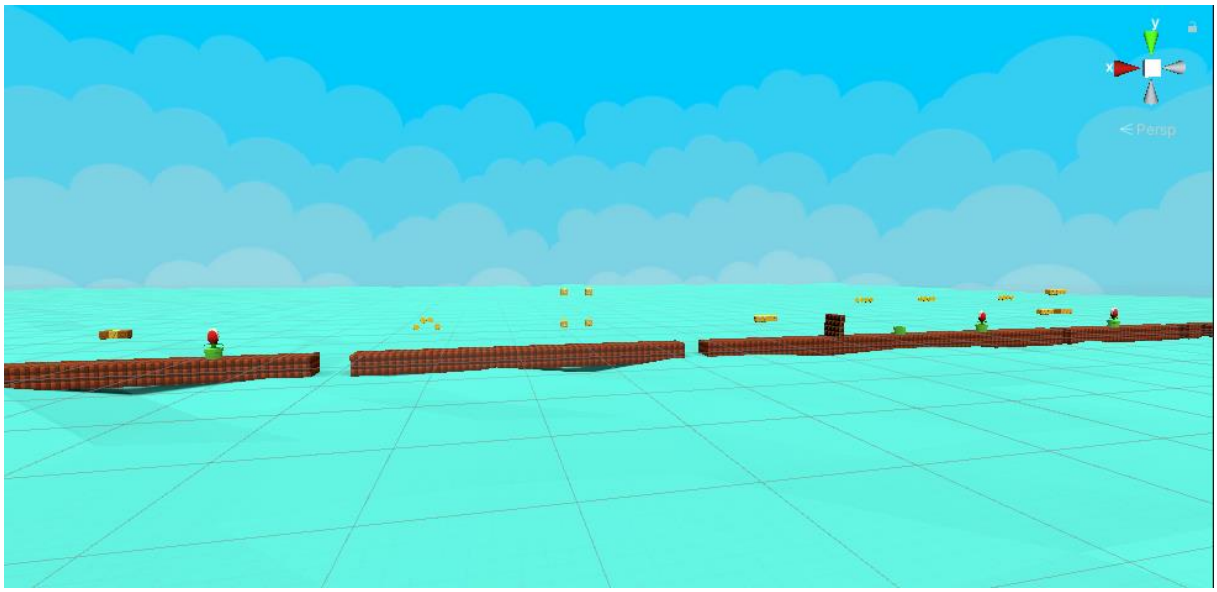


Ilustración 122 Estructura del nivel 6

Además, este incluirá un nuevo tipo de enemigo que será desarrollado a continuación.

7.6.2.1 Spiny

Para el inicio del desarrollo de este enemigo se ha obtenido un modelo del paquete Hello Mario Assets.

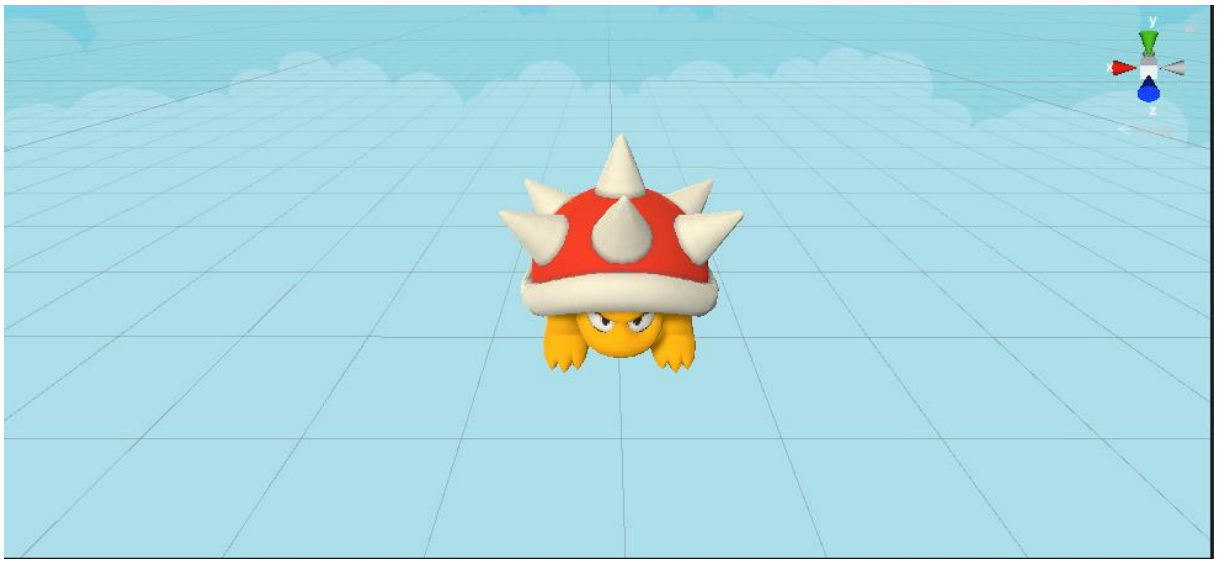


Ilustración 123 Modelo de Spiny

Este modelo se ha añadido a un objeto al cual se le ha añadido un componente *Box Collider* con la opción *isTrigger* activada. A continuación, se han añadido los *scripts EnemyMakeDamage* y *EnemyMove2D*, y se han creado los puntos izquierdo y derecho entre los que el enemigo se moverá.

Ilustración 124 Objeto final del enemigo Spiny

Por último, se le ha añadido un *Animator Controller* junto con una animación de caminar que será reproducida en *loop*.

7.6.3 Sistema de sonido

Para el sistema de sonido se ha creado un objeto llamado *Audio* el cual contendrá cada uno de los efectos de sonidos del juego como objetos hijo.

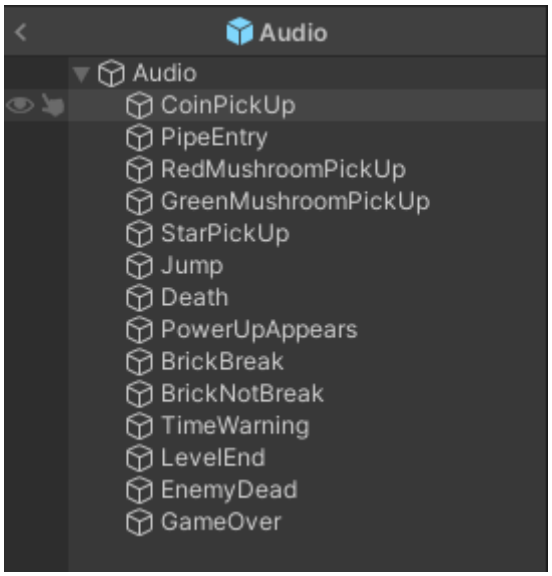


Ilustración 125 Objeto Audio

Cada uno de ellos contendrá un objeto *Audio Source* con la opción *Play On Awake* desactivada. Los efectos de sonido han sido descargados desde la wiki Video Game Music Preservation Foundation [11]. Los efectos de sonido que se han añadido son:

- Recoger una moneda
- Recoger una seta roja
- Recoger una seta verde
- Recoger una estrella
- Entrar en una tubería
- Saltar
- Morir
- *Powerup* aparece
- Bloque de ladrillo se rompe
- Bloque de ladrillo rebota
- Poco tiempo restante en el nivel
- Final de nivel
- Enemigo derrotado

- *Game over*

El objeto principal *Audio* contendrá el *script DontDestroy* para que este no sea destruido entre escenas, facilitando el uso de ellos por el *script GameControl* y el resto de *los scripts,* desde los que se usarán los efectos de sonido mediante el método *Play* del componente *AudioSource*. Con esto, quedaría terminado también el script GameControl.

```
public class GameControl : MonoBehaviour
{
    private static GameObject player;

    //Vidas
    public static int startingLives = 3;
    private static int internalLives;
    private GameObject livesDisplay;

    //Monedas
    private static int internalCoins;
    private GameObject coinsDisplay;

    //Estrellas
    private static int internalStars;
    private GameObject starsDisplay;

    //Timer
    public int startingTime = 300;
    public static float internalTime;
    private GameObject timeDisplay;
    private bool warning = false;
    public static bool stopTimer = false;

    //Powerup
    public bool bigMario = false;
    private static bool internalBigMario;
    private static bool makeMarioBigger = false;
    private static bool makeMarioSmall = false;

    //Pipe
    private static bool usingPipe = false;

    //Score
    private static int score = 0;
    private int displayScore = 0;
    private GameObject scoreUI;

    //Control
    private static bool restart = false;
    private static bool endGame = false;

    //DontDestroy
    private static GameControl gameControl = null;

    private void Awake()
    {
        if (gameControl == null)
        {
            gameControl = this;
            DontDestroyOnLoad(gameObject);
        }
        else if (gameControl != this)
        {
            Destroy(this);
        }
    }

    // Start is called before the first frame update

    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player").gameObject;

        //Vidas
        livesDisplay = GameObject.Find("LivesDisplay").gameObject;
        internalLives = startingLives;
```

```
    //Monedas
    coinsDisplay = GameObject.Find("CoinsDisplay").gameObject;
    internalCoins = 0;

    //Estrellas
    starsDisplay = GameObject.Find("StarsDisplay").gameObject;
    internalStars = 0;

    //Timer
    internalTime = startingTime;
    timeDisplay = GameObject.Find("TimeDisplay").gameObject;

    //Score
    scoreUI = GameObject.Find("ScoreDisplay").gameObject;
    scoreUI.GetComponent<TextMeshProUGUI>().text = displayScore.ToString();
    scoreUI.GetComponent<TextMeshProUGUI>().text = displayScore.ToString();
    StartCoroutine(updateScore());

    //Powerup
    internalBigMario = bigMario;
    makeMarioBigger = bigMario;
}

// Update is called once per frame

void Update()
{
    if (!stopTimer)
    {
        updateTimer();
    }

    updateDisplays();
    updateMarioScale();

    if (endGame)
    {
        endGame = false;
        StartCoroutine(restartGame());
    }

    if (restart)
    {
        restart = false;
        StartCoroutine(restartLevel());
    }
}


private void updateDisplays()
{
    livesDisplay.GetComponent <TextMeshProUGUI>().text = "" + internalLives;
    coinsDisplay.GetComponent<TextMeshProUGUI>().text = "" + internalCoins;
    starsDisplay.GetComponent<TextMeshProUGUI>().text = "" + internalStars;
    timeDisplay.GetComponent<TextMeshProUGUI>().text = "" + (int)internalTime;
}


private IEnumerator restartGame()
{
    player.GetComponent<PlayerControllerCharacterController>().enabled = false;
    player.GetComponent<CharacterController>().enabled = false;

    GameControl.score = 0;

    GameObject.Find("LevelMusic").GetComponent<AudioSource>().Stop();
    player.GetComponentInChildren<Animator>().SetBool("die", true);
    GameObject.Find("GameOver").GetComponent<AudioSource>().Play();

    yield return new WaitForSeconds(3.7f);

    SceneManager.LoadScene(1);
}
```

```
private IEnumerator restartLevel()
{
    stopTimer = true;

    player.GetComponent<PlayerControllerCharacterController>().enabled = false;
    player.GetComponent<CharacterController>().enabled = false;

    player.GetComponentInChildren<Animator>().SetBool("die", true);
    GameObject.Find("LevelMusic").GetComponent<AudioSource>().Stop();
    GameObject.Find("Death").GetComponent<AudioSource>().Play();

    yield return new WaitForSeconds(2.7f);

    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex - 1);

    player.GetComponentInChildren<Animator>().SetBool("die", false);

    player.GetComponent<PlayerControllerCharacterController>().enabled = true;
    player.GetComponent<CharacterController>().enabled = true;
}

//Vidas

public static void decreaseLives()
{
    internalLives--;

    if (internalLives <= 0)
    {
        endGame = true;
    }
    else
    {
        restart = true;
    }
}
public static void increaseLives()
{
    internalLives++;
    GameControl.addScore(150);
    GameObject.Find("GreenMushroomPickUp").GetComponent<AudioSource>().Play();
}

//Monedas

public static void decreaseCoins()
{
    internalCoins--;
}


public static void increaseCoins()
{
    internalCoins++;
    GameControl.addScore(100);
    if (internalCoins == 100)
    {
        internalCoins = 0;
        increaseLives();
    }
    GameObject.Find("CoinPickUp").GetComponent<AudioSource>().Play();
}

//Estrellas

public static void decreaseStars()
{
    internalStars--;
}

public static void increaseStars()
{
    internalStars++;
    GameControl.addScore(200);
    GameObject.Find("StarPickUp").GetComponent<AudioSource>().Play();
}

//Timer

private void updateTimer()
{
    internalTime -= Time.deltaTime;
    if (internalTime < 0)
    {
        stopTimer = true;
        decreaseLives();
        GameObject.Find("Death").GetComponent<AudioSource>().Play();
    }

    if(internalTime < startingTime / 2 && !warning)
    {
        warning = true;
        GameObject.Find("TimeWarning").GetComponent<AudioSource>().Play();
    }
}
```

```
//Powerup

public static void powerUpCollect()
{
    if (internalBigMario)
    {
        increaseLives();
    }
    else
    {
        GameControl.addScore(50);
        internalBigMario = true;
        makeMarioBigger = true;
        GameObject.Find("RedMushroomPickUp").GetComponent<AudioSource>().Play();
    }
}


private IEnumerator powerUpCollectAnimation()
{
    player.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.7f, 0.7f, 0.7f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.7f, 0.7f, 0.7f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f);
}

private void updateMarioScale()
{
    if (makeMarioBigger)
    {
        StartCoroutine(powerUpCollectAnimation());
        makeMarioBigger = false;
    }

    if (makeMarioSmall)
    {
        StartCoroutine(damageReceivedAnimation());
        makeMarioSmall = false;
    }
}

//Pipe

public static void setUsingPipe(bool isUsing)
{
    usingPipe = isUsing;
    if (usingPipe)
    {
        GameObject.Find("PipeEntry").GetComponent<AudioSource>().Play();
    }
}


public static void damageReceived()
{
    if (internalBigMario)
    {
        internalBigMario = false;
        makeMarioSmall = true;
    }
    else
    {
        decreaseLives();
    }
}


private IEnumerator damageReceivedAnimation()
{
    GameObject.Find("PipeEntry").GetComponent<AudioSource>().Play();
    player.transform.localScale = new Vector3(0.7f, 0.7f, 0.7f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.7f, 0.7f, 0.7f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f);
    yield return new WaitForSeconds(0.05f);
    player.transform.localScale = new Vector3(0.7f, 0.7f, 0.7f);
}


public static bool isBigMario()
{
    return internalBigMario;
}
```

```
public static bool isUsingPipe()
{
    return usingPipe;
}

//Score

private IEnumerator updateScore()
{
    while (true)
    {
        if (displayScore < score)
        {
            displayScore++;
            scoreUI.GetComponent<TextMeshProUGUI>().text = displayScore.ToString();
        }
        yield return new WaitForSeconds(0.01f);
    }
}


public static void addScore(int points)
{
    score += points;
}

//Reset

public static void resetGame()
{
    Debug.Log("Started GameControl");
    player = GameObject.FindGameObjectWithTag("Player").gameObject;

    //Vidas
    internalLives = startingLives;

    //Monedas
    internalCoins = 0;

    //Estrellas
    internalStars = 0;

    //Timer
    internalTime = 300;

    //Powerup
    internalBigMario = false;
}
```

Ilustración 126 Script GameControl completo

Para la música de cada uno de los niveles se ha creado un objeto y *prefab* *LevelMusic* que contiene un *AudioSource* con las opciones *Play On Awake* y *Loop* activadas. Este objeto será añadido a cada uno de los niveles desarrollados, y contendrá la música que se escuchará a lo largo de cada nivel. Este *AudioSource* podrá ser controlado desde los *scripts*, reiniciando la música cuando el jugador muera, o pausando la música cuando se acceda al menú de pausa, que será desarrollado en las siguientes iteraciones. La música ha sido obtenida de la wiki Video Game Music Preservation Foundation.

7.7 Séptima iteración

En esta iteración se desarrollarán los niveles 7 y 8, así como todos los elementos necesarios para completar los niveles.

7.7.1 Nivel 7 (2-3)

Para el desarrollo del presente nivel se ha obtenido del paquete Hello Mario Assets un nuevo modelo para la creación de una nueva plataforma, la cual será un puente.

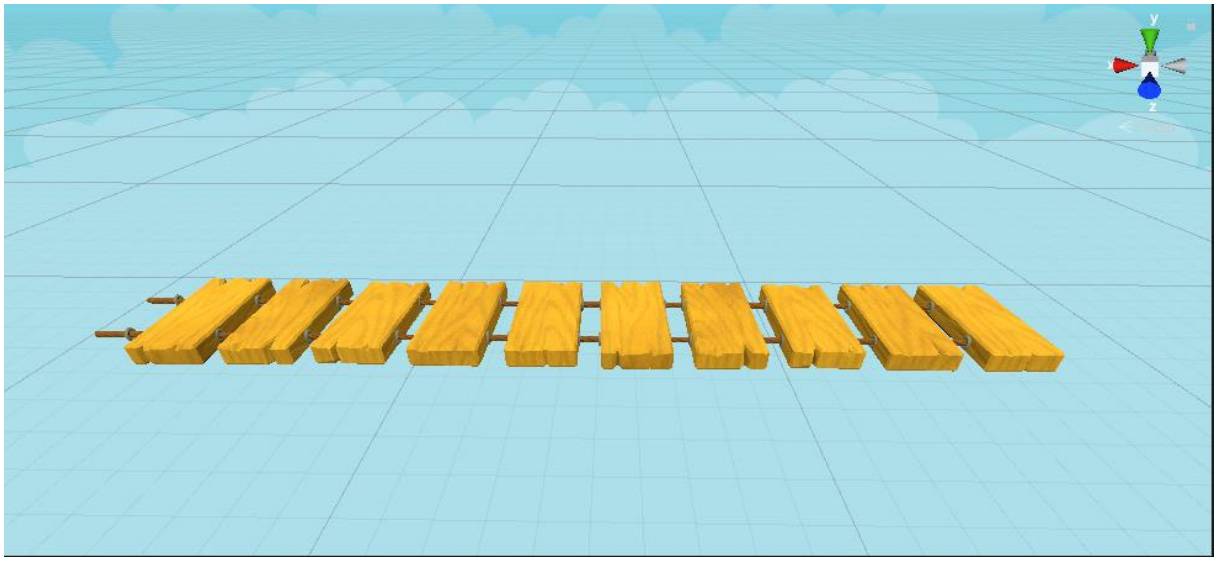


Ilustración 127 Modelo del puente

Esta nueva plataforma contendrá un componente *Box Collider* y será usada a lo largo de todo el nivel, siendo la estructura básica que usaremos a lo largo del mismo.

También, se ha desarrollado una segunda plataforma que estará presente en la segunda mitad del nivel. Su modelo ha sido obtenido del paquete Hello Mario Assets.

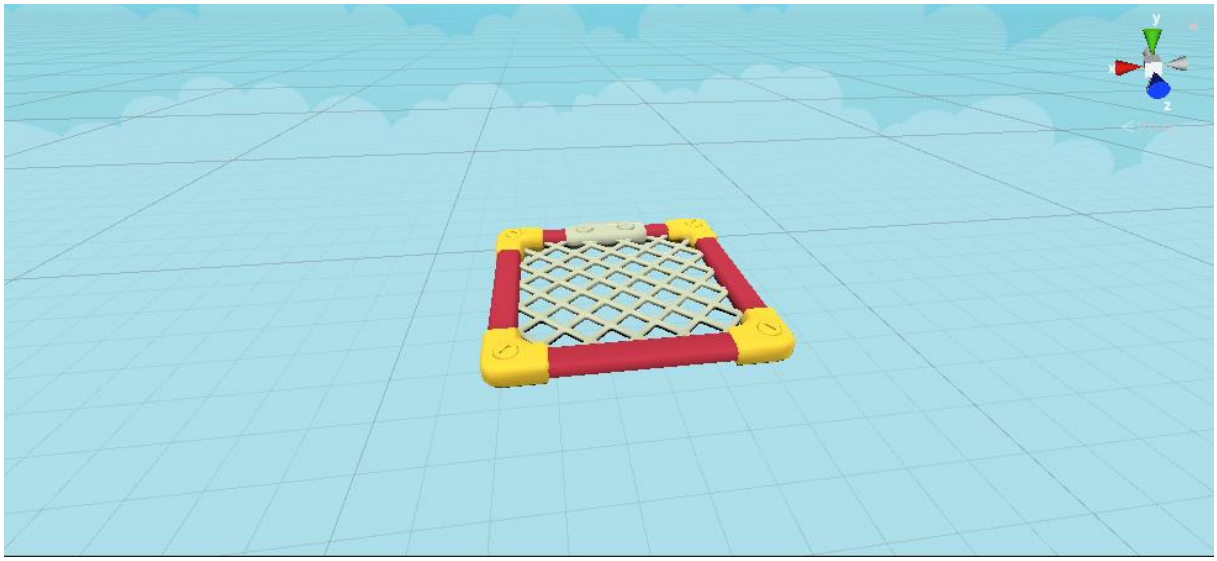


Ilustración 128 Modelo de la plataforma nivel 7

Junto con los bloques desarrollados en iteraciones pasadas se ha creado la estructura del nivel en base al diseño.

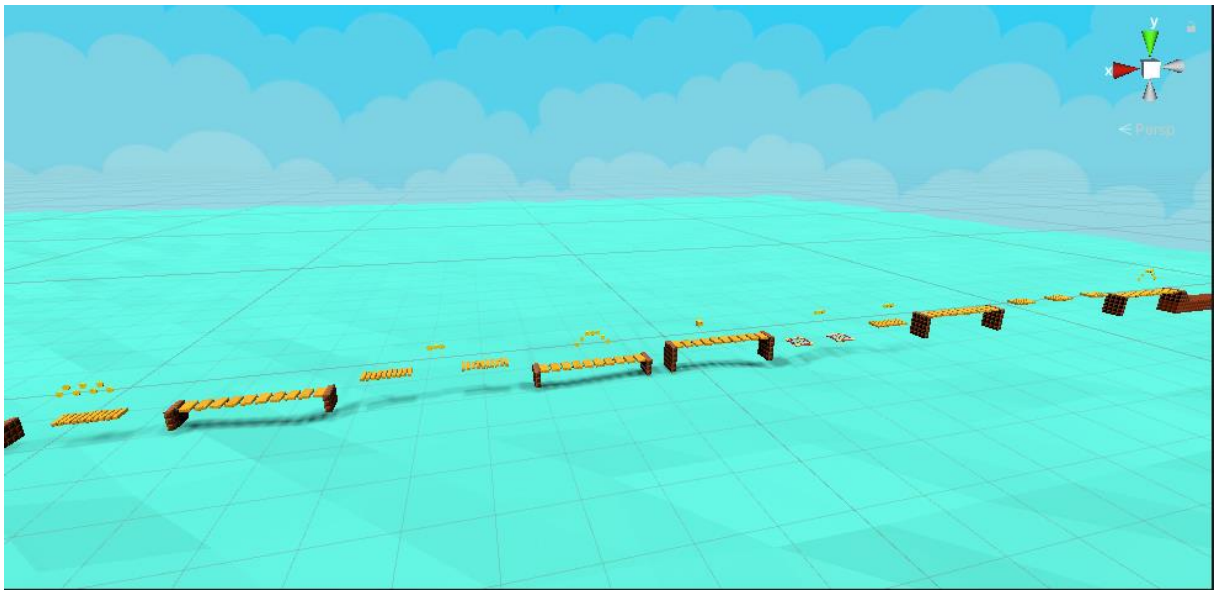


Ilustración 129 Estructura del nivel 7

7.7.1.1 Cheep Cheep

En este nivel se ha creado un nuevo enemigo el cual estará oculto bajo el nivel. Este realizará un movimiento de salto hacia arriba, realizando una parábola y volverá a ocultarse.

Para iniciar el desarrollo de este enemigo se ha obtenido el modelo del paquete Hello Mario Assets.

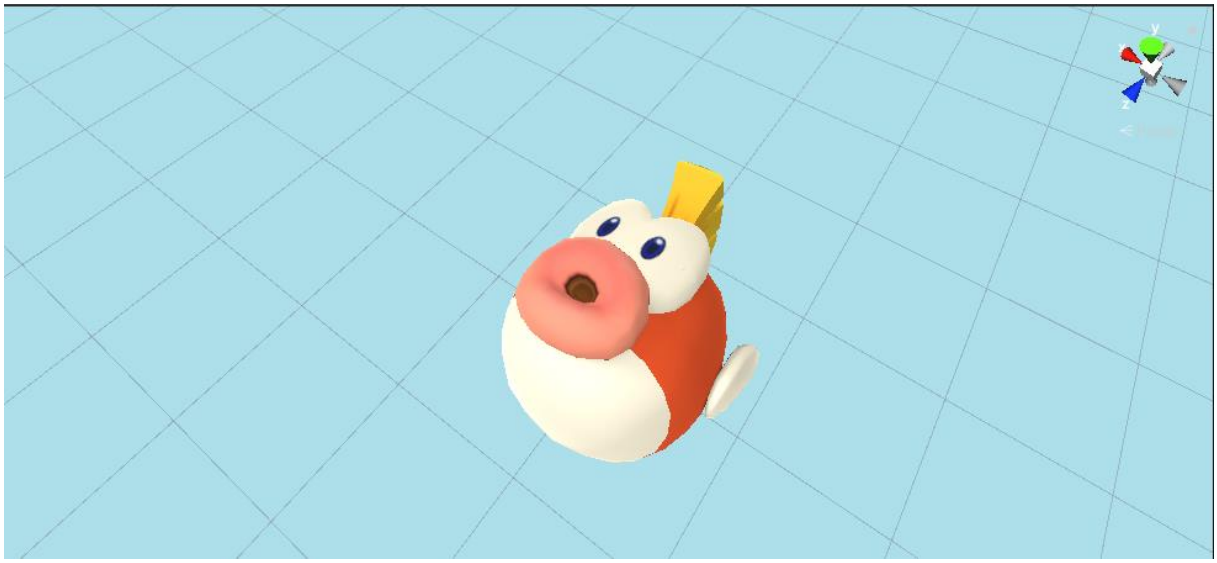


Ilustración 130 Modelo del Cheep Cheep

Se ha añadido un componente *Box Collider* con la opción *isTrigger* activada junto con el *script EnemyMakeDamage*. A continuación, se ha creado y añadido una animación para el salto.

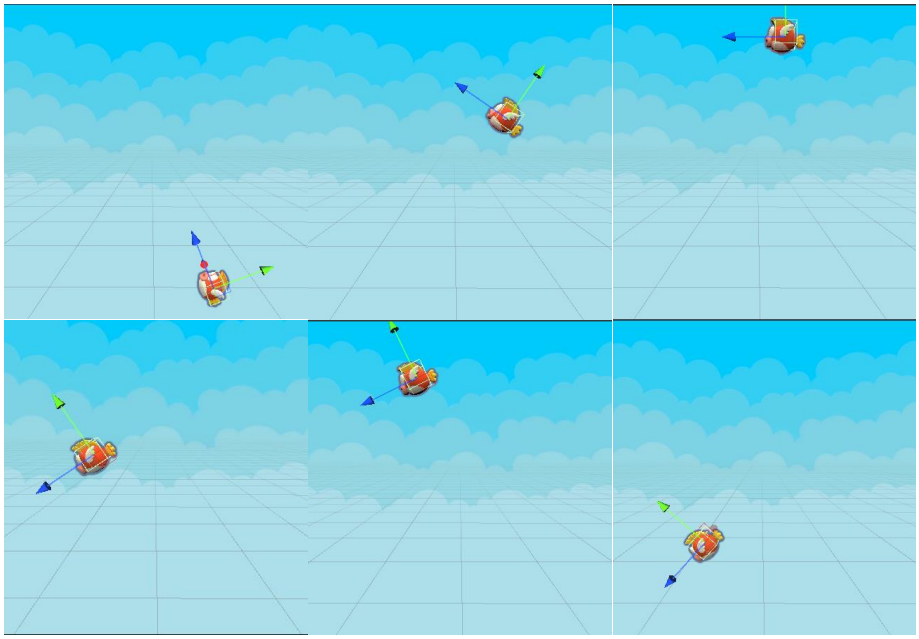


Ilustración 131 Animación de salto Cheep Cheep

Por ultimo, se han añadido al nivel y ajustado para que dificulten el progreso del jugador.

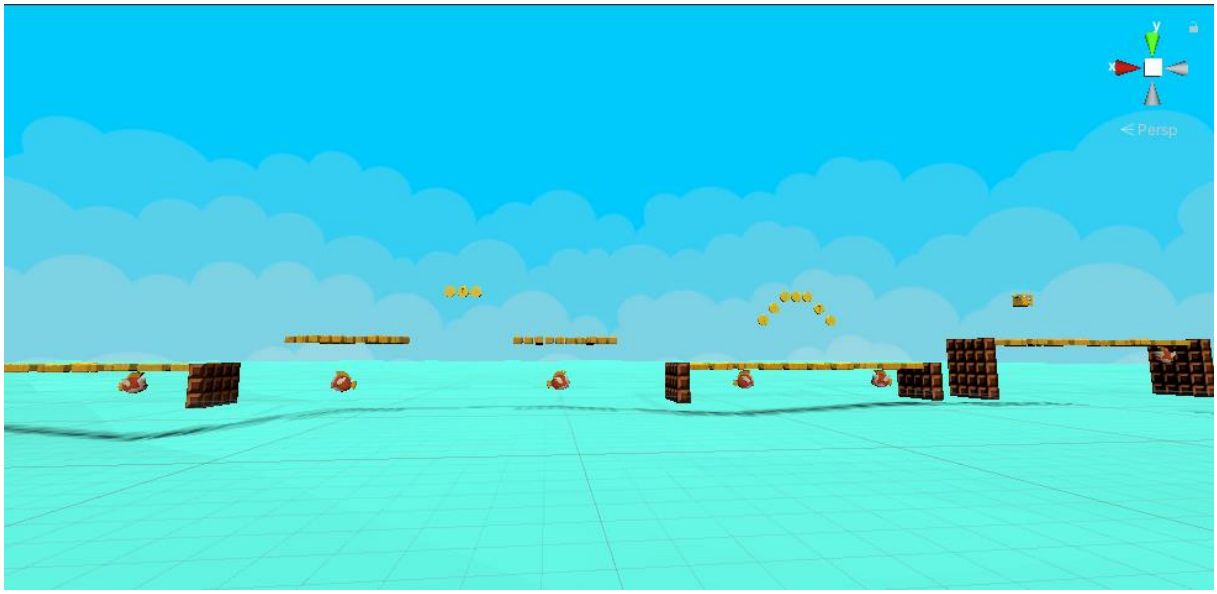


Ilustración 132 Cheep Cheep en el nivel 7

7.7.2 Nivel 8 (2-4)

Para el desarrollo del último nivel de este proyecto se han reutilizado bloques y objetos creados en las iteraciones anteriores, así como el jefe final para completar su estructura básica.

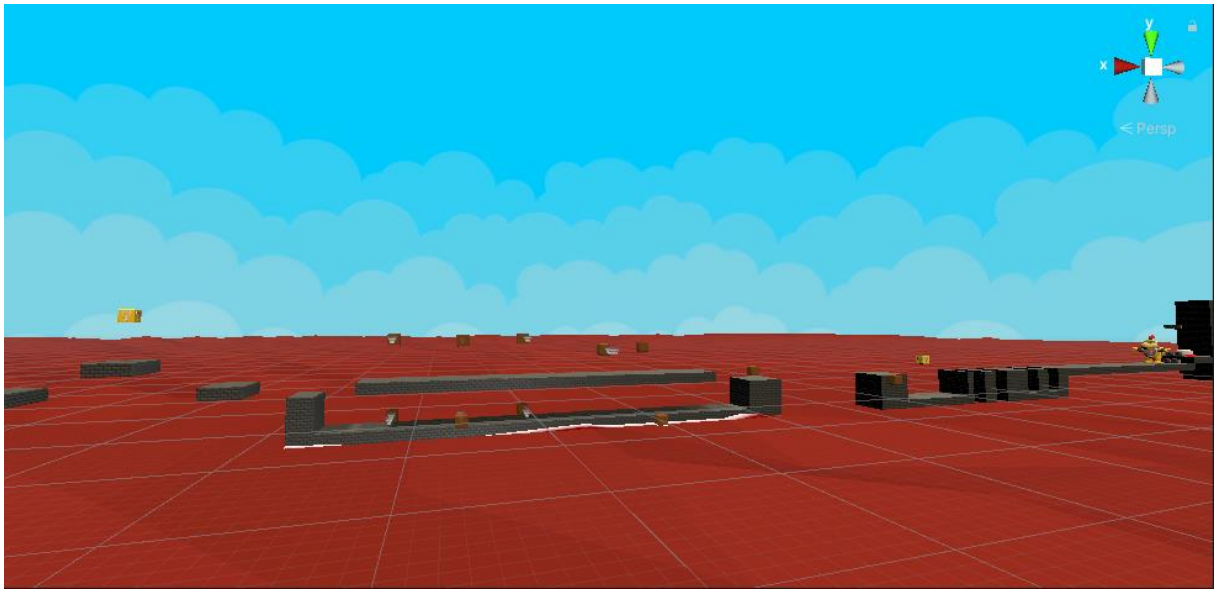


Ilustración 133 Estructura del nivel 7

7.7.2.1 Plataforma con desplazamiento vertical y reinicio

Esta plataforma estará ubicada al final del nivel y se moverá de arriba hacia abajo hasta llegar a un punto predeterminado. En ese momento, esta se trasladará al punto superior desde el que inicio el movimiento. Para el desarrollo de esta plataforma se ha utilizado el modelo de la desarrollada para el nivel 7.

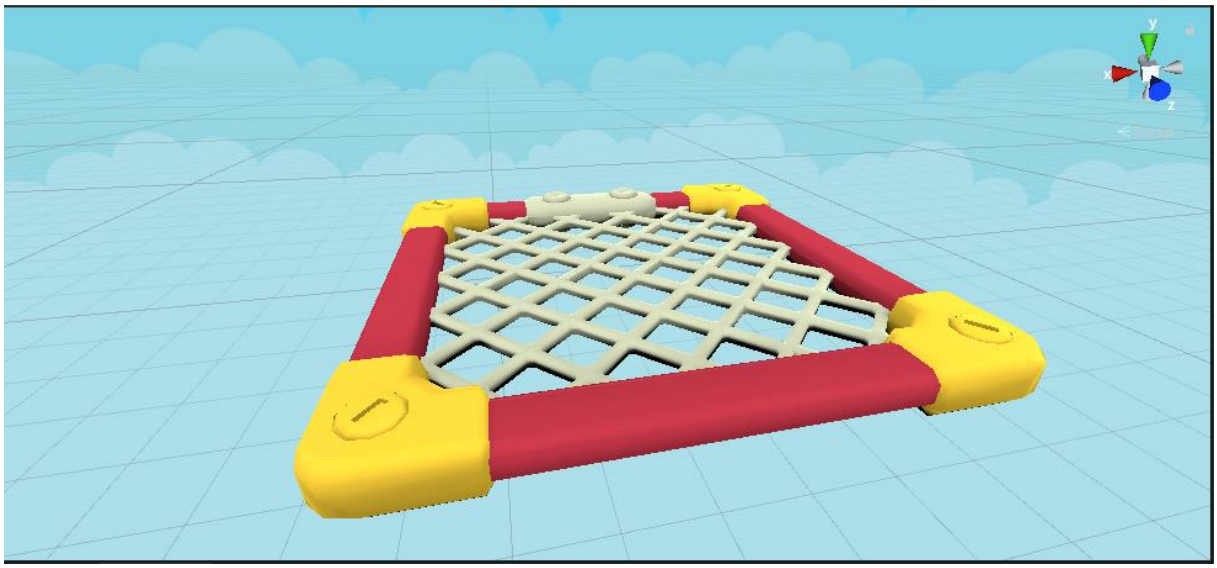


Ilustración 134 Modelo de la plataforma

Se le ha añadido un componente *BoxCollider*, y se ha creado y añadido un *script LiftMoveVerticalCastle*. A continuación, se han creado dos objetos vacíos que estarán ubicados por encima y por debajo de la plataforma. Estos serán los puntos donde la plataforma desaparecerá y reaparecerá de nuevo, y podrán ser ubicados manualmente desde el editor.

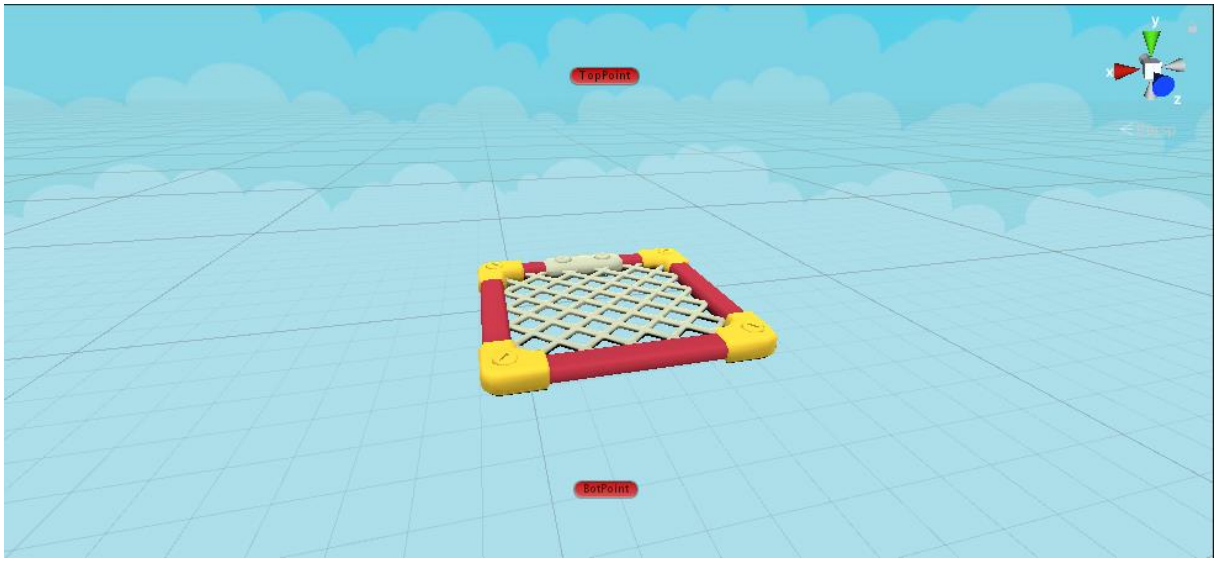


Ilustración 135 Objeto final de la plataforma

En el *script LiftMoveVerticalCastle*, en el método *Start* se han obtenido la coordenada Y de los puntos superior e inferior; en el método *Update* se comprobará si la plataforma está por debajo de la posición del punto inferior y, de ser así, transportará la plataforma al punto superior. Si esta está aún por encima del punto inferior, se trasladará hasta llegar hasta él.

Estas se han añadido al final del nivel, y se han colocado dos plataformas en cada una de las ubicaciones.

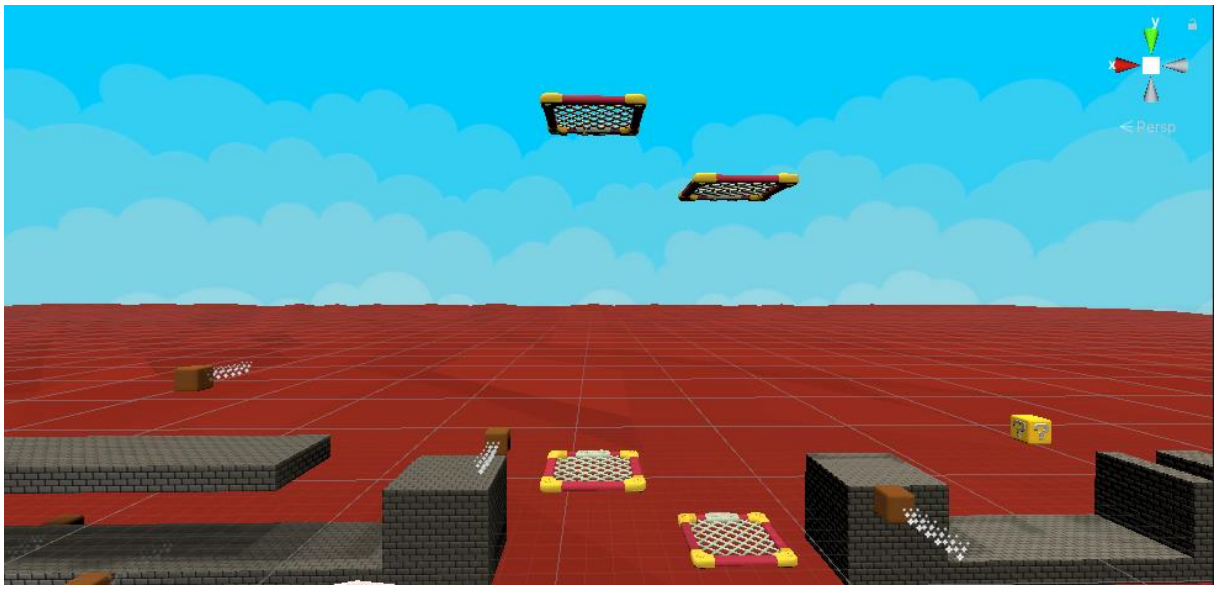


Ilustración 136 Plataformas en nivel 7

7.7.2.2 Bolas de fuego

Las bolas de fuego estarán ubicadas al inicio del nivel. Este enemigo subirá desde su posición inicial hasta un punto predeterminado y, después, caerán hasta llegar de nuevo a la posición inicial. Para el desarrollo de estas se ha usado el mismo sistema de partículas usado en la quinta iteración, desarrollado para las barras de fuego giratorias y el cañón de bolas de fuego.

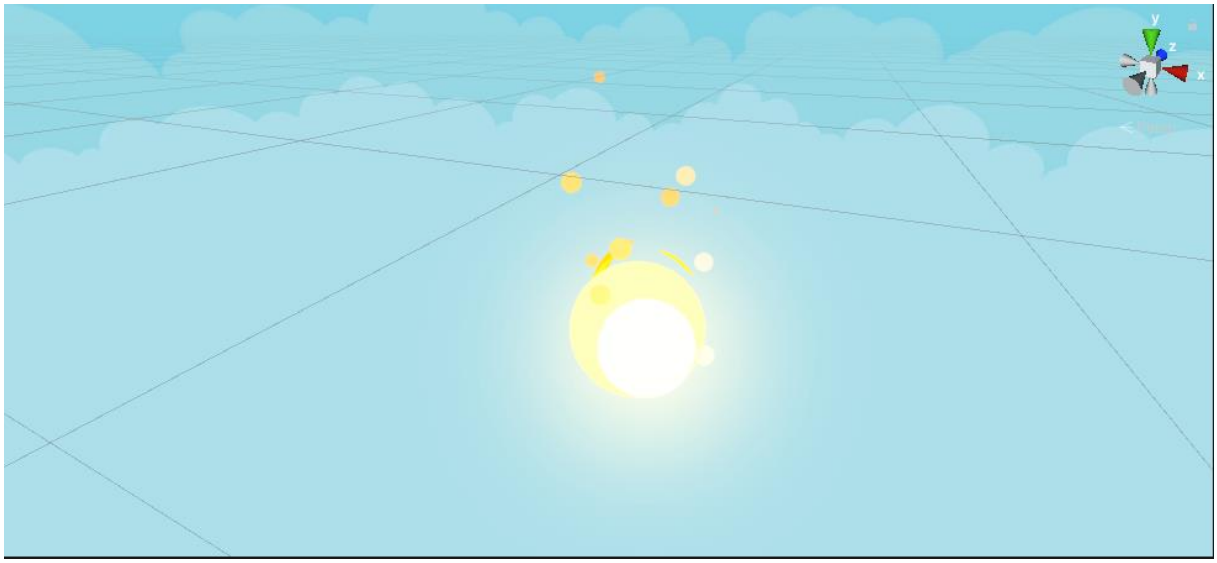


Ilustración 137 Sistema de partículas de la bola de fuego

A este sistema de partículas se le ha añadido un componente *Rigidbody* con los parámetros por defecto, y el *script* *EnemyMakeDamage*, desarrollado

anteriormente. Además, se ha desarrollado un nuevo *script* llamado *JumpingFireball* en el cual, en el método *Start* se obtendrá el componente *Rigidbody* para posteriormente, en el método *Update* actualizar el parámetro *velocity* del *Rigidbody*, en el eje Y. Se le aplicara una velocidad de quince, que hará que la bola se eleve. Para que no se aplique esta velocidad en cada *frame*, se ha añadido una variable de control *jump* y una corutina, para que cada vez que se aplique una velocidad al objeto se espere cuatro segundos antes de volver a aplicarse. A continuación, se ha añadido un objeto cubo que será la base desde la que partirá el objeto y al que caerá. Este tendrá el componente *Mesh Renderer* desactivado.

Por último, se ha creado un *prefab* de este objeto, y se ha añadido al nivel en la zona inicial.

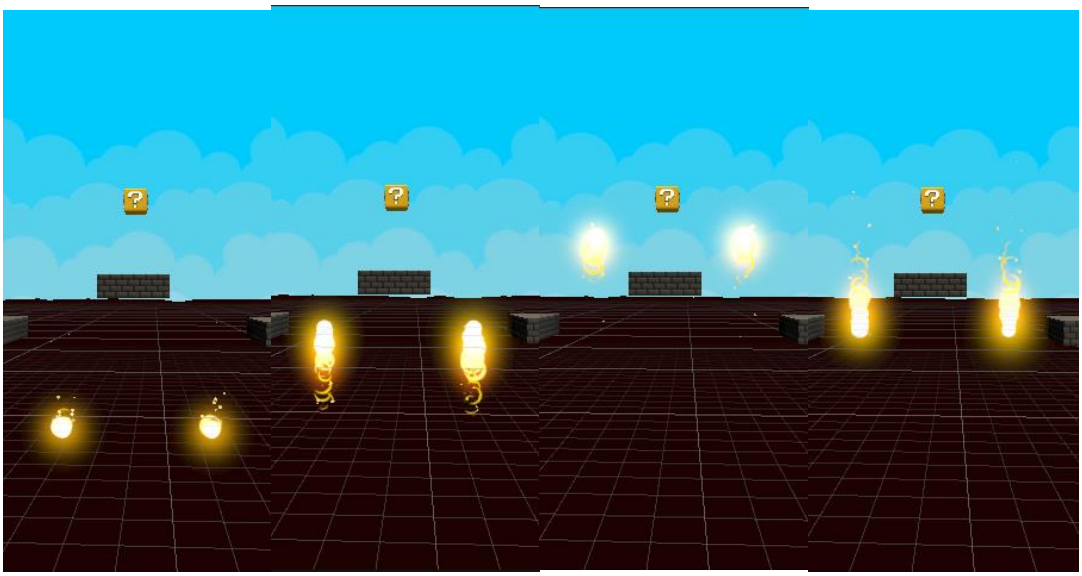


Ilustración 138 Animación de la bola de fuego

7.8 Octava iteración

En esta iteración se desarrollarán la pantalla de inicio, el menú principal y el menú de pausa.

7.8.1 Menú principal

Para el menú principal se ha creado una nueva escena llamada *MainMenu* en la que se ha añadido un objeto canvas. En este se ha creado un nuevo objeto *Image*, que será el fondo del menú, y al que se le ha aplicado un color rojo. A continuación, se ha añadido un nuevo objeto *MainMenu* que contendrá tres objetos de tipo *Button*, que serán los botones *PLAY, OPTIONS* y *QUIT*.

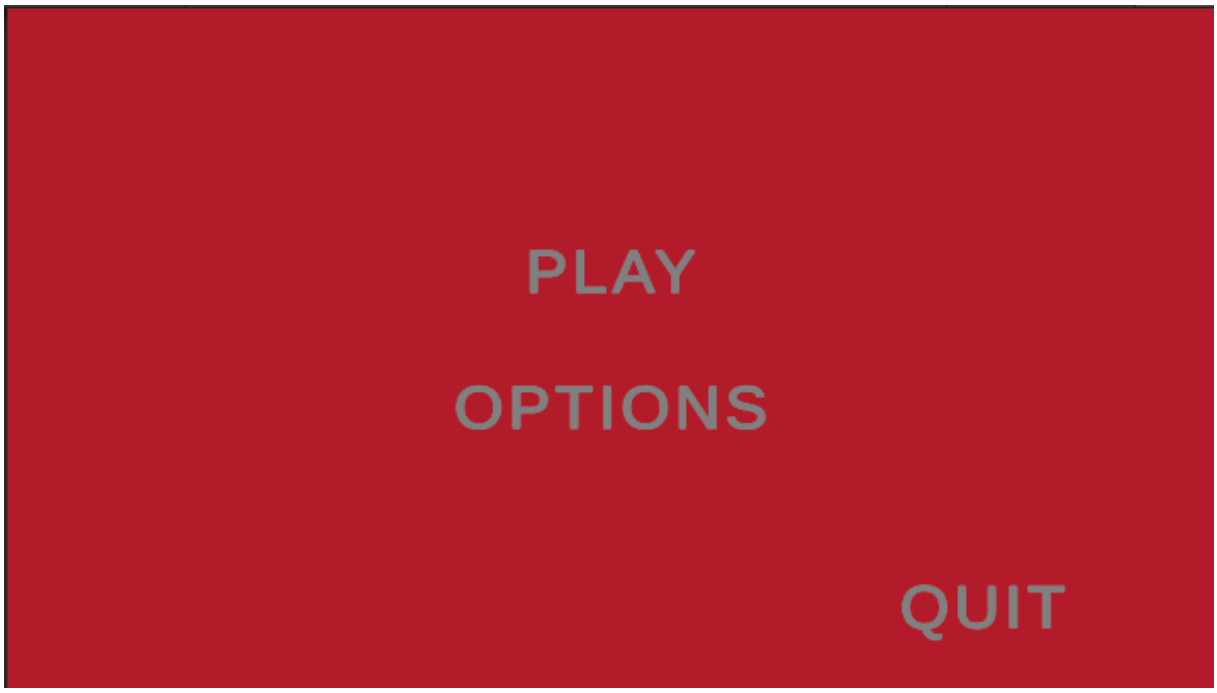


Ilustración 139 Menú principal

Se ha creado un objeto vacío llamado *OptionsMenu*, el cual estará desactivado por defecto. Dentro de este se ha añadido un objeto *Dropdown* el cual mostrará las resoluciones disponibles para cambiar entre ellas; también se ha creado un objeto de tipo *Toggle*, el cual permitirá que el juego se muestre en pantalla completa o no; por último, se ha creado un objeto *Slider* para controlar el volumen del juego.

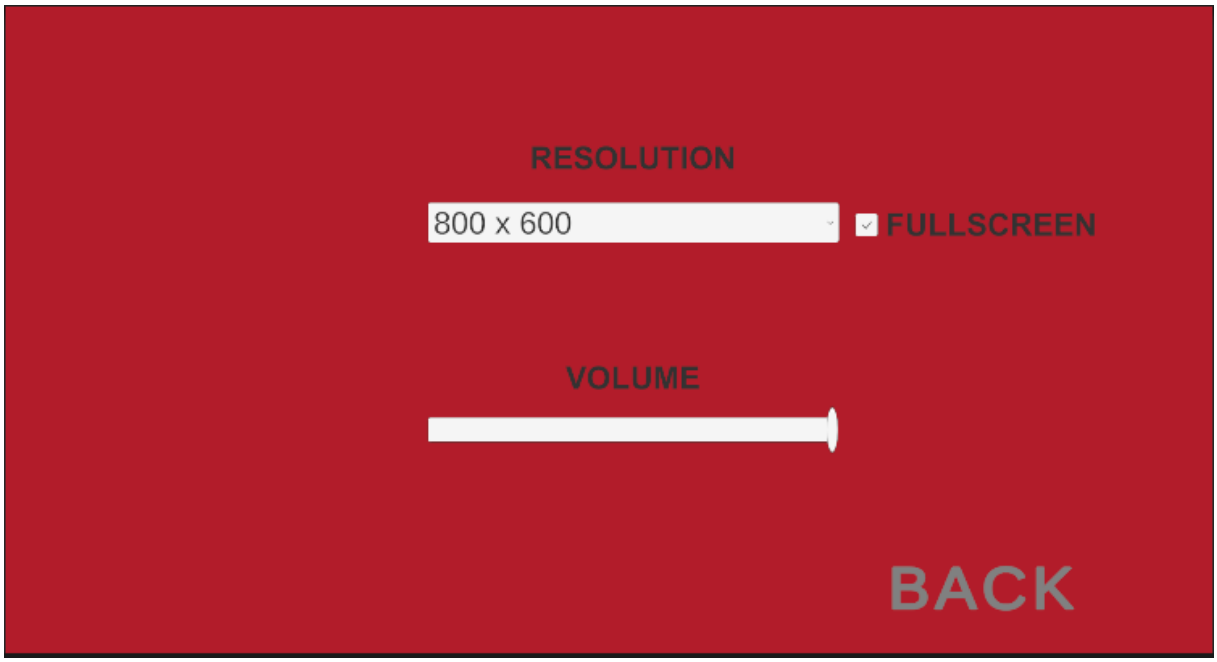


Ilustración 140 Menú de opciones

Se ha creado un *script* llamado *MainMenu,* y se ha añadido al objeto del mismo nombre. En este se ha creado un método para cada uno de los botones: para el botón *PLAY*, se ha creado el método *PlayGame*, el cual cargará la escena que contiene la pantalla de carga del primer nivel; para el botón *OPTIONS* se ha creado el método *OptionsMenu*, el cual desactivará el objeto llamado *MainMenu*, y activará el objeto *OptionsMenu*; por último, para el botón *QUIT* se ha creado el método *QuitGame* el cual cerrará el juego. Estos métodos han sido asignados a los botones correspondientes mediante el inspector, con el método *OnClick*.

Para el menú de opciones se ha creado el *script OptionsMenu* y, además, un *AudioMixer* junto con una variable que estará expuesta llamada *"volume",* para controlar el volumen mediante el slider.

En el método *Start* del *script* se ha creado una lista de resoluciones para el desplegable, que será obtenida mediante el metodo *Screen.resolutions* de Unity. La resolución por defecto será la resolución de la pantalla donde estemos jugando el juego. A continuación, se ha creado el método *setResolution* que, mediante el índice de la lista desplegable y la lista que se había creado en el *script*, modificará la resolución del juego. También, se ha creado el método *setFullscreen*, que recibirá un *booleano* indicando si debe cambiarse a pantalla completa o modo ventana. Se ha creado el método *setVolume* que, haciendo uso del *mixer*, modificará el volumen del juego. Por último, se ha creado el método *backToMenu* el cual desactivará el menú de opciones y volverá a activar el menú principal. Estos métodos han sido asignados a cada uno de los objetos del menú de opciones en el método *OnValueChanged.*

Para terminar, se ha creado un objeto *MainMenuMusic* al que se le ha añadido un componente *AudioSource* que contendrá la música del menú principal, con las opciones *Play On Awake* y *Loop* activadas. La música del menú principal ha sido descargada desde la wiki Video Game Music Preservation Foundation, del videojuego Super Mario 64.

7.8.2 Pantalla inicio

Para la pantalla de inicio se ha creado un nuevo objeto en el canvas de la escena *MainMenu* llamado *StartScreen*. A este se le ha añadido un modelo 3D de Mario, el mismo usado para el personaje principal. A continuación, se ha modificado

el *Render Mode* del *canvas* a *Screen Space – Camera*, para que el modelo en 3D pueda ser visualizado en el *canvas*.

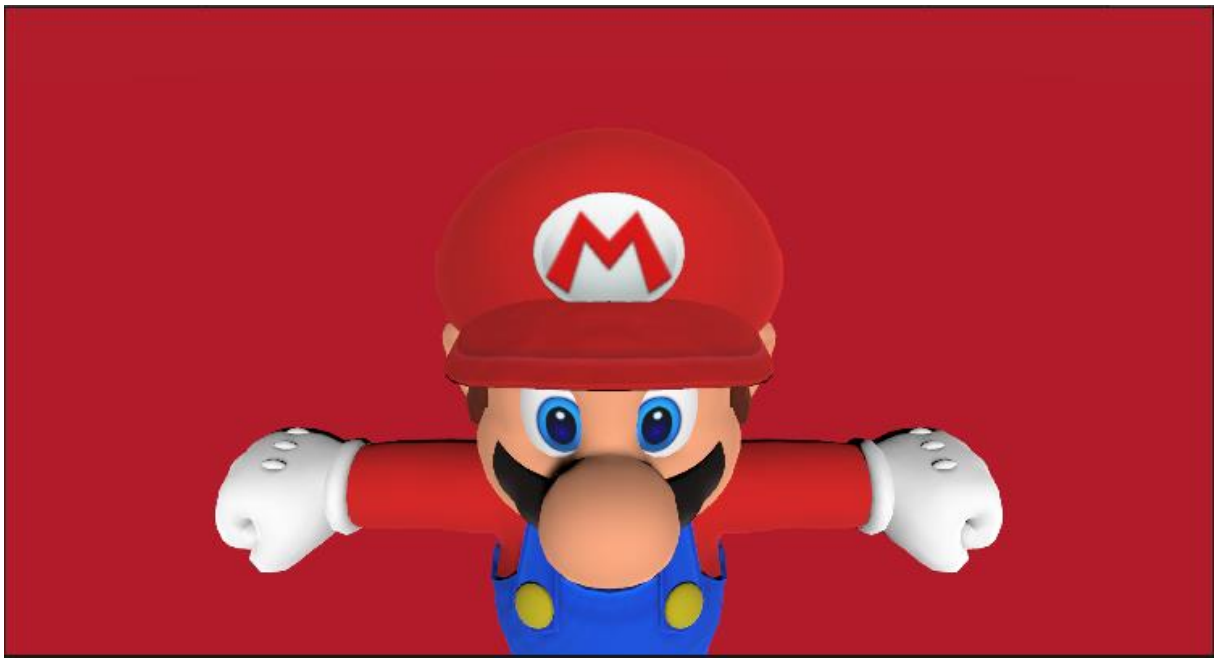


Ilustración 141 Primera versión de la pantalla de inicio

A este se le ha añadido una animación.

Ilustración 142 Animación de la pantalla de inicio

Se ha usado el logo del videojuego New Super Mario Bros, adaptándolo a nuestro proyecto. Se ha cambiado la palabra *New* por TFG.

Ilustración 143 Logo New Super Mario Bros. modificado

Se ha añadido el logo a la pantalla de inicio, creando un objeto *Raw Image* en el *canvas*.

Ilustración 144 Segunda versión de la pantalla de inicio

También, se ha añadido un objeto *TextMeshPro*, en la esquina inferior derecha, con el texto *Press any key*. A este, se le ha añadido una animación la cual hará que el texto pase a ser invisible gradualmente y viceversa.

Ilustración 145 Ultima versión de la pantalla de inicio

A continuación, se ha creado un objeto llamado *StartMusic*, dentro del objeto *StartScreen*, que contendrá un componente *AudioSource* con las opciones *Play On*

Awake y *Loop* activadas. Este será el encargado de reproducir la música de la pantalla de inicio. La música de la pantalla de inicio ha sido descargada desde la wiki Video Game Music Preservation Foundation, del videojuego Super Mario 64.

Por último, se han desactivado los objetos *MainMenu* y *MainMenuMusic*, creados anteriormente, y se ha desarrollado el *script GoToMainMenu*, en el cual en el método *Update* esperará a que sea pulsada cualquier tecla o botón. Una vez que esta sea pulsada se desactivará el objeto *StartScreen* y se activarán los que desactivamos al inicio, mostrándose el menú principal del juego.

7.8.3 Menú de pausa

Para el menú de pausa se ha añadido un objeto al *canvas* de la interfaz llamado *PauseMenu*. Este contendrá los botones *RESUME, MAIN MENU y QUIT*.

A continuación, se ha desarrollado un *script* llamado *PauseMenu*, que estará contenido en el objeto *canvas* de la interfaz, y que en el método *Update* esperará a que el jugador pulse la tecla *Escape*. En ese caso, se llamará al método *pauseGame* el cual detendrá la música y mediante el método *Time.timeScale* detendrá el juego. Si el juego ya estaba pausado, y el botón escape es pulsado de nuevo, se reanudará la música y el tiempo, tal y como estaba antes de acceder al menú, mediante el método *resumeGame*.

También se han creado los métodos *resume*, *mainMenu* y *quit* para los botones anteriores. El primero realizará una llamada al método *resumeGame* creado anteriormente; el segundo cargará de nuevo la escena del menú principal; el ultimo cerrará el juego.

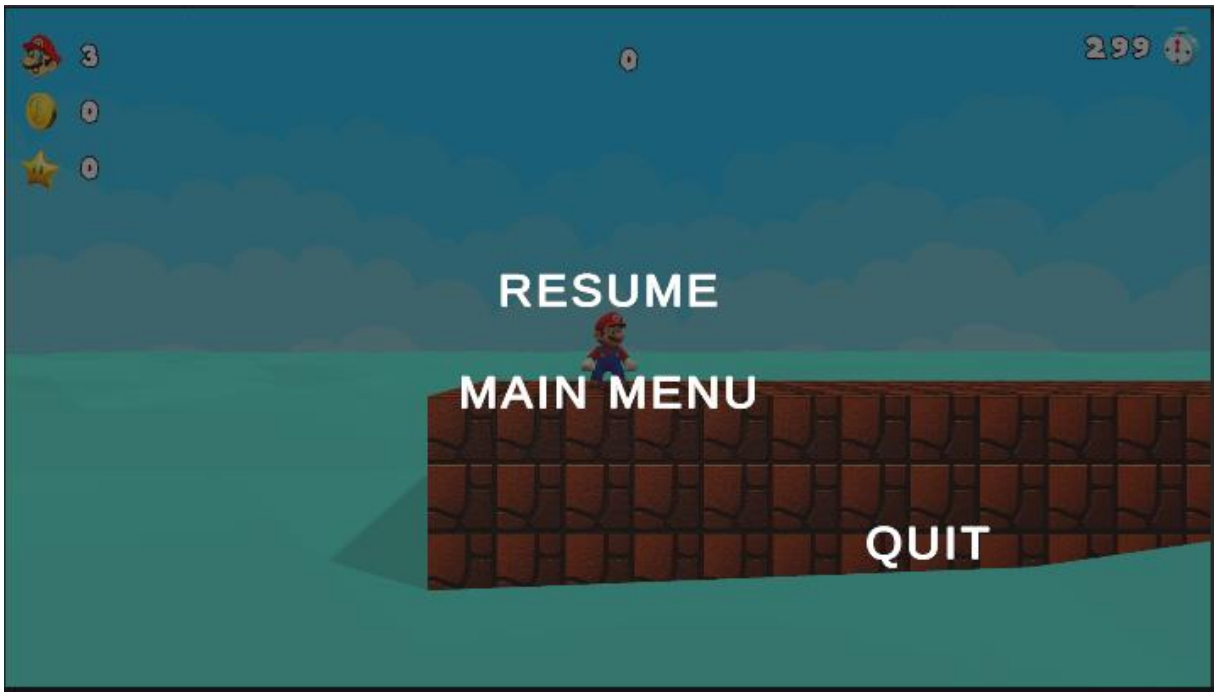


Ilustración 146 Menú de pausa

7.9 Novena iteración

En esta iteración se ha desarrollado el generador aleatorio de niveles, incluyendo la escena de carga y scripts necesarios para su correcto funcionamiento.

7.9.1 Generador aleatorio de niveles

El generador aleatorio de niveles generará un nivel completamente diferente cada vez que accedamos al nivel. Este será un modo de juego diferente al juego principal, desarrollado en las iteraciones anteriores, y se jugará de forma infinita hasta que el jugador pierda todas sus vidas. Cada vez que el jugador llegue al final del nivel se generará otro nivel nuevo completamente aleatorio. Estos usarán todos los objetos creados en iteraciones pasadas para los niveles del juego principal. A partir de ellos se crearán *prefabs* con fragmentos de niveles, que serán usados para producir un nivel completamente nuevo.

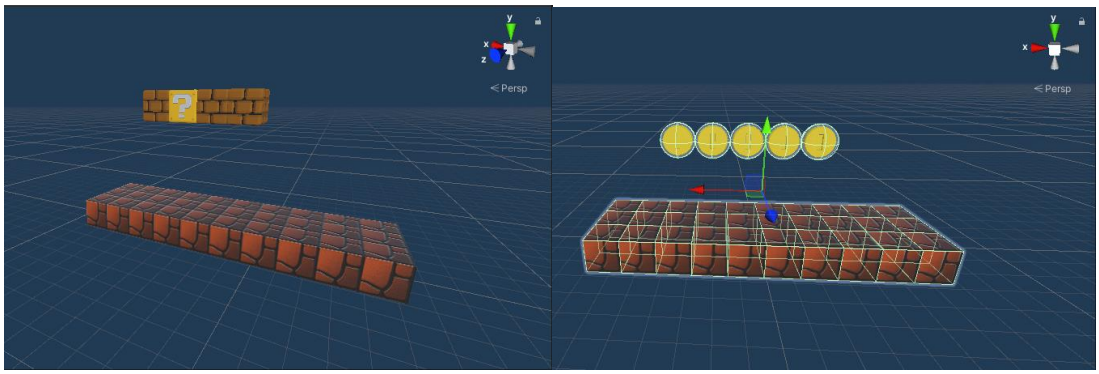


Ilustración 147 Fragmentos de niveles para el generador

Para comenzar con el inicio del generador se ha creado una nueva escena llamada *RandomLevel* a la que se han añadido los objetos *LevelMusic*, *LevelDeathTrigger y StartPosition* desarrollados anteriormente. Se ha modificado el *prefab* del objeto *StartPosition* y se ha eliminado el *script* que contenía. A continuación, se ha creado una copia de este *script*. Este se ha renombrado a *CreateStartLevel* y dentro del mismo se ha creado el método *createRandomLevel*. En el método *Update* se hará una llamada a este antes de realizar las acciones que ya teníamos en él. Dentro del método *createRandomLevel* se ha inicializado el objeto *Random* para su uso posterior, así como un objeto *Vector3,* que será la posición inicial desde la que se empezará a generar el nivel. Se ha creado una variable entera, que almacenará el número del fragmento aleatorio, y una cola, que almacenará los últimos números de los fragmentos usados. A continuación, se generará el nivel mediante un bucle en el que se obtendrá un número aleatorio, que será el nombre del fragmento y este se cargará mediante el método *Resources.Load* y se instanciará en la posición inicial que creamos anteriormente. A esta se le sumarán diez unidades en el eje X en cada iteración del bucle. El número obtenido se añadirá a la cola, a la vez que se elimina el elemento más antiguo. Cuando el bucle acabe, se instanciará el *prefab LevelEnd*, que contendrá el objeto *LevelEndTrigger*. Para terminar, se colocará al jugador en la posición inicial del nivel, que será la posición del objeto *StartPosition*.

```
21      private void createRandomLevel()
22      {
23          Random.InitState(System.DateTime.Now.Millisecond + System.DateTime.Now.Minute);
24
25          Vector3 pos = new Vector3(0, 0, 0);
26          int randomNumberPrefab;
27          int notRepeat = numPrefabs/2 + 1;
28          Queue<int> lastRandomNumbers = new Queue<int>(notRepeat);
29          for (int i = 0; i < notRepeat; i++)
30          {
31              lastRandomNumbers.Enqueue(0);
32          }
33
34          for (int i = 0; i < levelLenght; i++)
35          {
36              do
37              {
38                  randomNumberPrefab = Random.Range(1, numPrefabs + 1);
39              } while (lastRandomNumbers.Contains(randomNumberPrefab));
40
41              lastRandomNumbers.Dequeue();
42              lastRandomNumbers.Enqueue(randomNumberPrefab);
43
44              Instantiate(Resources.Load(randomNumberPrefab.ToString()), pos, Quaternion.identity);
45              pos.x += -10;
46          }
47
48          Instantiate(Resources.Load("LevelEnd"), pos, Quaternion.identity);
49      }
50  }
```

Ilustración 148 Código del generador de niveles aleatorio

Se ha creado una carpeta llamada *Resources*, dentro de la jerarquía del proyecto, para que el método *Resources.Load* del *script* anterior pueda encontrar los *prefabs* y cargarlos correctamente. Estos estarán nombrados con números para que, mediante el numero aleatorio generado, puedan ser seleccionados. También tendremos el *prefab LevelEnd*, nombrado anteriormente.

Ilustración 149 Carpeta Resources con los fragmentos de nivel

También, se ha creado una pantalla de carga personalizada para los niveles generadores aleatoriamente.

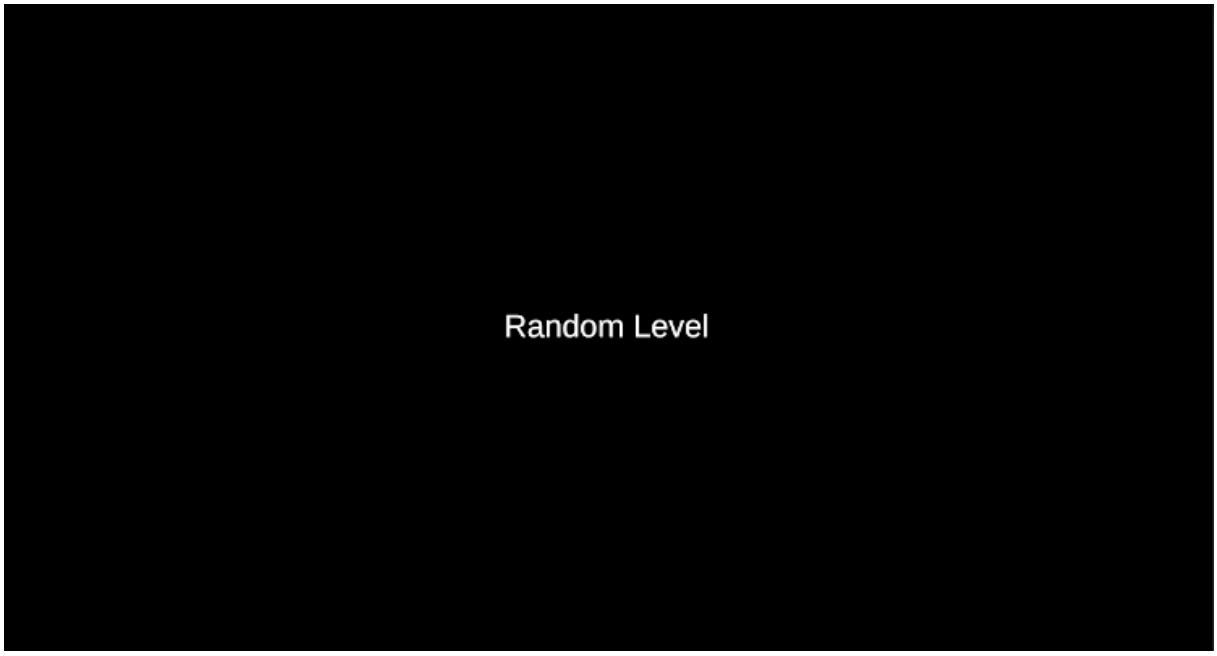


Ilustración 150 Pantalla de carga del nivel aleatorio

Por último, se ha añadido un nuevo botón al menú principal mediante el cual accederemos a estos niveles aleatorios, junto con un nuevo método en el *script* *MainMenu* llamado *PlayRandomGame*, que cargará la escena correspondiente.

7.10 Decima iteración

En esta iteración se introducirán mejoras al menú principal y se realizarán pruebas a todo el proyecto para comprobar su correcto funcionamiento. A continuación, se resolverán los posibles problemas que se puedan hallar en las pruebas y se introducirán mejoras y ajustes, en caso de ser necesarios. Por último, se realizarán pruebas para comprobar que se han resuelto los errores encontrados.

7.10.1 Mejora del menú principal

Para la mejora del menú principal se ha modificado su diseño y disposición respecto del diseño original y se han introducido algunos elementos nuevos.

Se ha modificado la posición de los botones del menú principal, ubicándolos en la parte inferior izquierda. Además, los botones *OPTIONS y QUIT* tienen un tamaño menor que los botones de *PLAY*. También, se ha añadido el logo en la parte superior izquierda.

Ilustración 151 Nuevo diseño del menú principal

Al igual que para la pantalla de inicio, se ha añadido el modelo de Mario en la parte derecha de la escena junto con una animación.

Ilustración 152 Modelo de Mario en el menú principal

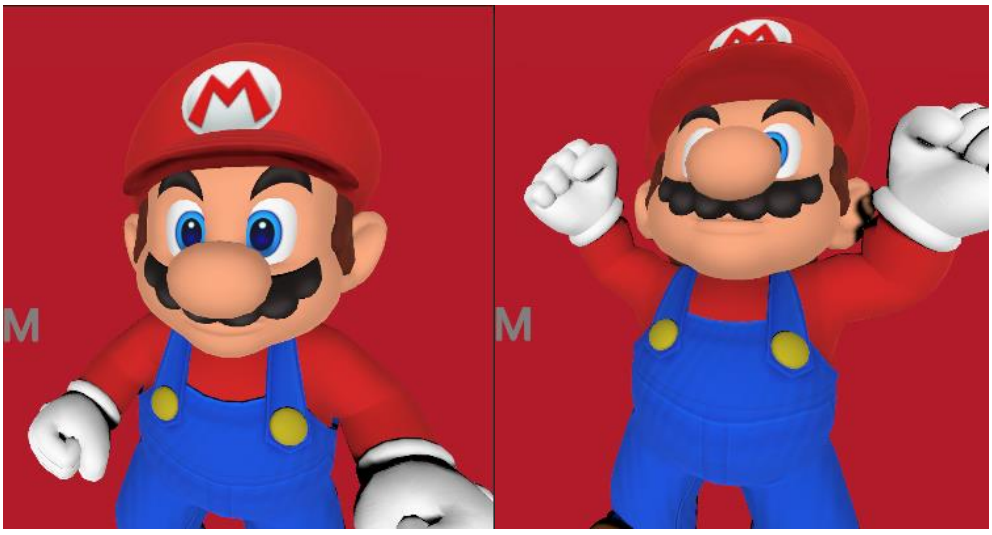


Ilustración 153 Animación del modelo de Mario en el menú principal

Por último, se han añadido los controles en la pantalla de opciones, para que el jugador pueda consultarlos antes de empezar a jugar.

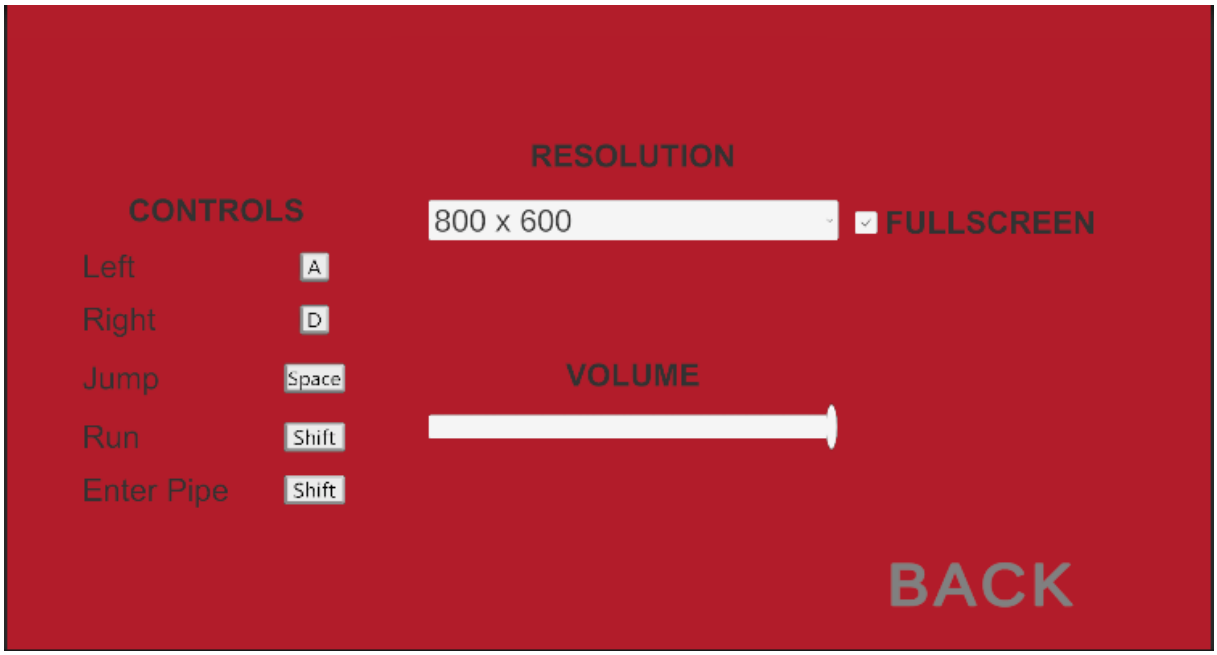


Ilustración 154 Controles en el menú de opciones

7.10.2 Pruebas y resolución de errores

Durante las pruebas se han realizado varias sesiones de juego en el modo de juego principal, superando todos los niveles hasta llegar de nuevo al menú principal. En estas sesiones se han encontrado varios errores a corregir:

1. Si el jugador recoge una estrella en un nivel y muere, esta no se resta de su contador de estrellas.
2. Si el jugador inicia una nueva partida después de terminar una, el score no se reinicia.

Los errores descritos se han corregido y probado correctamente las soluciones realizadas.

8 CONCLUSIONES

Al término de este proyecto se han obtenido gran cantidad de conocimientos en muchas áreas tales como programación, diseño de interfaces, diseño de niveles, algoritmos, planificación y desarrollo de documentación, entre otros. Esto ha contribuido en la adquisición de experiencia de cara al desarrollo de videojuegos, al ser un proyecto multidisciplinar realizado individualmente y no por un equipo de

desarrollo, en el cual cada integrante desarrollaría una función concreta. Se han afrontado gran cantidad de desafíos a lo largo del desarrollo de este y se han resuelto de forma satisfactoria.

Respecto al proyecto, consiste en la realización de un remake del videojuego clásico Super Mario Bros, para la consola NES. Se han reconstruido un total de ocho niveles, repartidos en dos mundos en los que se han desarrollado una gran cantidad de objetos y enemigos, presentes en el juego original. También, se ha incluido un generador de niveles aleatorios, el cual ha supuesto un gran reto en el proyecto y mediante el cual el jugador podrá jugar de forma indefinida, teniendo cada vez un nivel diferente. Esto permite que el jugador pueda volver a jugar el juego todas las veces que quiera, sin tener que repetir siempre el juego principal y los mismos niveles.

A partir de este punto, podría trabajarse en la mejora del generador aleatorio, incluyendo más variedad a la hora de la creación automática del nivel. Esto podría conseguirse incluyendo una mayor cantidad de *prefabs* para el mismo, así como incluyendo mejoras en el código de creación del nivel. También, podrían implementarse el resto de los mundos, desarrollando los objetos y enemigos que formaran los seis restantes del juego original. Podría crearse un sistema de logros y recompensas en base a las estrellas recogidas a lo largo de los niveles, los enemigos eliminados, el tiempo, etc., para que el jugador tuviera una motivación más para volver a jugar al juego, una vez terminado este. Por último, también podría incluirse un modo multijugador, local o en línea, donde poder jugar con hasta 4 personas, como en los juegos más actuales de la saga.

De cara al futuro, este proyecto podría convertirse también en un juego del estilo Super Mario Maker, para la consola Nintendo Wii U, o Super Mario Maker 2, para la consola Nintendo Switch, en el cual es el mismo jugador el que podrá crear y compartir niveles. Esto podría ser posible debido a la gran cantidad de *prefabs* que se tienen de todos los objetos y enemigos desarrollados a lo largo de todas las iteraciones, aprovechando que estos funcionan independientemente de lo que haya en el nivel o donde se coloquen. Esto podría ser posible por medio de una interfaz que visualizara todos estos *prefabs* y que permitiera combinarlos, arrastrando o con algún otro sistema, para dar lugar a un nivel nuevo.

Ilustración 155 Interfaz de creación de niveles Super Mario Maker 2

9 APÉNDICES

9.1 Guía original del Trabajo Fin de Título

(Cód.: 19/20-2990) Remake de un videojuego clásico

Tutor del TFG: **CARLOS JAVIER OGAYAR ANGUITA**
Modalidad: Proyecto de Ingeniería | Tipo: TFG General
Número máximo de estudiantes: 5 (5 asignados)
Idioma: Castellano

Este trabajo consiste en la realización de un remake de algún videojuego clásico utilizando un motor de videojuegos actual, como Unreal Engine, CryEngine o Unity. El videojuego original, así como el motor a utilizar, quedan a elección del alumnado.

Hay que describir las mecánicas de juego y la experiencia de usuario perseguida, incluyendo el sistema de recompensas, los mecanismos de progreso del usuario, los recursos gráficos y sonoros empleados, etc. Para la elaboración de escenas, personajes, NPCs, objetos, animaciones y todo tipo de material gráfico y sonoro, se podrán utilizar elementos descargables de la red, incluyendo sprites originales o cualquier otro recurso. Se recomienda elegir un videojuego de la década de los 80s y 90s, buscando la sencillez.

Conocimientos Previos

Es necesario tener experiencia con el uso de motores de videojuegos.

Objetivos del TFG

- Diseño del remake del videojuego clásico elegido, especificando las características y el sistema/entorno objetivo.
- Descripción de las dinámicas, mecánicas y componentes de juego, que deberían ser las mismas del juego original. Se permite añadir contenido adicional o variaciones.
- Diseño de la interfaz de usuario y los mecanismos de interacción.
- Diseño del entorno virtual 2D/3D, incluyendo todo el material gráfico y sonoro.
- Desarrollo del juego propuesto, incluyendo la programación de los componentes necesarios.
- Realización de pruebas de evaluación del sistema y análisis de resultados.

Metodología a Desarrollar

- Describir el videojuego clásico elegido para realizar el remake. Describir las dinámicas, mecánicas y componentes de juego, lo que incluye narrativa, objetivos, recompensas, etc.
- Realizar un breve estado del arte sobre motores de videojuegos y seleccionar uno de ellos. Justificar las decisiones tomadas.
- Seleccionar y utilizar una metodología basada en Ingeniería del Software para el análisis de requisitos, diseño, implementación, prueba y documentación. Se recomienda utilizar una metodología incremental.
- Detallar la estimación temporal y costes de desarrollo.
- Realizar el desarrollo del remake, incluyendo el modelado del entorno virtual 2D/3D.
- Realizar pruebas para el prototipo y documentar los resultados.
- Redacción de la documentación final requerida.

Documentos y Formatos de Entrega

- Documentación generada durante el proceso, de acuerdo a las buenas prácticas de la ingeniería del software. Esto se aplicará tanto en los proyectos de Ingeniería como en los trabajos teóricos o experimentales, es decir, siempre que se genere algún tipo de desarrollo, tanto software como webs, scripts, etc.
- Para el formato de documentación de los **Proyectos de Ingeniería** se utilizará la norma **UNE 157801**. Tal y como especifican las normas de estilo de la EPSJ: '*Los Proyectos de Ingeniería deberán presentar una estructura y contenido adaptados a la norma **UNE 157001** - Criterios generales para la elaboración de proyectos - u otra derivada de ésta, en función de la naturaleza del proyecto*'. '*…para el caso de los sistemas de información informatizados (sistemas informáticos, sistemas de información geográfica, etc.) está la norma derivada de la* anterior **UNE 157801** - *Criterios generales para la elaboración de proyectos de sistemas de información*'. Para ello, se puede utilizar también la Norma CCII-N2016-02 (**Norma Técnica para la realización de la Documentación de Proyectos en Ingeniería Informática**) del Consejo General de Colegios Profesionales de Ingeniería en Informática, que incluye e implementa la norma **UNE 157801**.
- Memoria del trabajo, incluyendo manuales y anexos, en formato PDF.
- Código fuente completo y ejecutables del prototipo o software desarrollado.
- Documentos y archivos adicionales a los que se haga mención expresa en la memoria.
- Vídeo de demostración de la aplicación o de la experimentación realizada.
- Anexos para la presentación del trabajo:
    - Informe favorable del tutor.
    - Autorización de publicación, si procede.

9.2 Manuales de usuario

9.2.1 Controles

Movimiento del jugador, izquierda o derecha respectivamente.

Correr.

Saltar.

Usar tuberías.

10 BIBLIOGRAFÍA

[1]Unity Technologies, 2021. *Unity.* [online]. Disponible en: <https://unity.com/es> [Último acceso 19 agosto 2021].

[2]Epic Games, 2021. *Unreal Engine.* [online]. Disponible en: <https://www.unrealengine.com/en-US> [Último acceso 19 agosto 2021].

[3]Crytek, 2021. *CryEngine.* [online]. Disponible en: <https://www.cryengine.com> [Último acceso 19 agosto 2021].

[4]Unity Technologies, 2021. *Unity Assets Store.* [online]. Disponible en: <https://assetstore.unity.com/> [Último acceso 19 agosto 2021].

[5]Microsoft, 2021. *Visual Studio.* [online]. Disponible en: <https://visualstudio.microsoft.com/es/> [Último acceso 19 agosto 2021].

[6]Microsoft, 2021. *Microsoft 365.* [online]. Disponible en: <https://www.microsoft.com/es-es/microsoft-365> [Último acceso 19 agosto 2021].

[7]Adobe, 2021. *Adobe Photoshop.* [online]. Disponible en: <https://www.adobe.com/es/products/photoshop.html> [Último acceso 19 agosto 2021].

[8]NESMaps, 2021. *Nintendo Game Maps.* [online]. Disponible en: <https://nesmaps.com/> [Último acceso 19 agosto 2021].

[9]Unity Technologies, 2021. *Farland Skies – Cloudy Crown.* [online]. Disponible en: <https://assetstore.unity.com/packages/2d/textures-materials/sky/farland-skies-cloudy-crown-60004> [Último acceso 19 agosto 2021].

[10]Hello Fangaming, 2021. *Hello Mario Assets.* [online]. Disponible en: <https://hellofangaming.github.io/HelloMarioAssets/> [Último acceso 19 agosto 2021].

[11]Video Game Music Preservation Foundation, 2021. *Video Game Music Preservation Foundation.* [online]. Disponible en: <http://www.vgmpf.com/> [Último acceso 19 agosto 2021].

[12]Ministerio de Empleo y Seguridad Social, 2018. *BOE XVII Convenio colectivo estatal de empresas de consultoría, y estudios de mercados y de la opinión pública.* [online]. Disponible en: <https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf> [Último acceso 21 agosto 2021].

[13]DaFont, 2021. *Super Mario 256.* [online]. Disponible en: <https://www.dafont.com/es/super-mario-256.font> [Último acceso 21 agosto 2021].

[14]Unity Technologies, 2021. *LowPoly Water.* [online]. Disponible en: <https://assetstore.unity.com/packages/tools/particles-effects/lowpoly-water-107563> [Último acceso 19 agosto 2021].

[15]Unity Technologies, 2021. *Epic Toon FX | VFX Particles.* [online]. Disponible en: <https://assetstore.unity.com/packages/vfx/particles/epic-toon-fx-57772> [Último acceso 19 agosto 2021].

[16]Ferrone, H. 2020. *Learning C# by Developing Games with Unity 2020: An enjoyable and intuitive approach to getting started with C# programming and Unity, 5th Edition*. Packt Publishing. Birmingham.

[17]Borromeo, N.A. 2020. *Hands-On Unity 2020 Game Development: Build, customize, and optimize professional games using Unity 2020 and C#.* Packt Publishing. Birmingham.

[18]Pressman, R. 2010. *Ingeniería del software*. McGraw-Hill. Nueva York.