Desarrollo de un videojuego y documentación de su diseño

Rubén González López

Director: Marc Alier Forment

Tutor GEP: Xavier Llinas Audet

Grado de ingeniería informática

Especialidad de Ingeniería del Software

Facultat d'informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - Barcelona Tech

16 de marzo de 2020

Índice

Índice de figuras

Abstract

Este trabajo de fin de grado consiste en el diseño y desarrollo de un videojuego 2D de plataformas, cuya historia narra las experiencias y emociones que experimenta una persona a lo largo de una transición de género. El objetivo es poder sensibilizar, informar y normalizar el concepto de la transexualidad para la sociedad, en un formato ameno y entretenido como es el de los videojuegos. Para ello, se utiliza el motor gratuito Unity.

Aquest treball de fi de grau consisteix en el disseny y desenvolupament d'un videojoc 2D de plataformes, el qual la seva història narra les experiències que experimenta una persona al llarg d'una transició de gènere. L'objectiu és poder sensibilitzar, informar i normalitzar el concepte de la transexualitat per a la societat, en un format amè i entretingut com es el dels videojocs. Per poder realitzar-lo, s'utilitza el motor gratuït Unity.

This end-of-degree project consists in the design and development of a 2D platformer video game, the story of which tells the experiences and feelings of a person throughout their gender transition. The objective of this work is to sensitize, inform and normalize the concept of transexuality for society, in a nice and enjoyable way such as a videogame. In order to make this, the Unity engine was used.

1. Contexto

1.1 Introducción

El trabajo de fin de grado “Desarrollo de un videojuego y documentación de su diseño” pertenece a los estudios del Grado de Ingeniería Informática de la Facultad de Informática de Barcelona [1] de la Universidad Politécnica de Cataluña, más concretamente de la especialidad de Ingeniería del Software.

El proyecto será realizado por Rubén González López (en proceso de transición para poder cambiarse el nombre a Rebeca) y está dirigido por Marc Alier Forment.

El proyecto consiste en la creación de software, específicamente la creación de un videojuego y la documentación de su diseño.

1.2 Términos y conceptos

A continuación se puede observar un listado ordenado alfabéticamente con diferentes términos y sus definiciones que son necesarios para la comprensión del documento.

Cisgénero: también conocido por su acrónimo cis, que hace referencia a las personas cuya identidad de género coincide con su fenotipo sexual, es decir, lo contrario a una persona trans.

FPS: Fotogramas Por Segundo, es la frecuencia a la que se muestran imágenes por pantalla. Usualmente en los videojuegos se suele poner a 30 o 60 FPS.

FSM: Finite State Machine, una máquina de estados finitos es un modelo de computación en el cual el autómata que lo implementa está en uno de esos estados y puede transicionar a otro dependiendo de una serie de variables. Permite facilitar la creación de sistemas inteligentes como la IA de los enemigos del juego.

GDD: Game Design Document, documento empleado en la industria de los videojuegos que contiene el diseño de un videojuego.

Público hardcore: jugador avezado que dedica gran parte de su tiempo a los videojuegos principalmente a aquellos que les suponen un desafío.

LGBT: siglas que representan Lesbianas Gays Bisexuales y Transexuales, se usa para referenciar a todo el colectivo que no es cis hetero.

RPG: Role Playing Game, se trata de un tipo de videojuegos en los que los jugadores adoptan un papel y se ven envueltos por la historia que les sucede al personaje en cuestión.

Unity: motor de desarrollo de videojuegos muy popular debido a su disponibilidad, precio y accesibilidad.

1.3 Identificación del problema

A día de hoy los videojuegos forman parte de la vida de millones de personas y cada vez este número va más en aumento. La gran mayoría de videojuegos tienen el único propósito de entretener a sus consumidores, pero mi TFG intenta ir más lejos y pretende concienciar a los jugadores de la existencia del colectivo trans aprovechando la popularidad del medio, ya que si bien es cierto que hoy en día cada vez hay más representación del colectivo LGBT en el mundo de los videojuegos, esta representación está principalmente formada por personas cisgénero homosexuales o bisexuales. Con este videojuego pretendo concienciar a las personas que lo jueguen, dando una perspectiva sobre cómo es hacer la transición basada en experiencias de primera mano, ayudando así al entendimiento popular respecto a la transición e intentando normalizarlo.

Dado que el juego pretende exponer al jugador a la transición, es importante el factor narrativo, del cual muchas veces se prescinde en otros videojuegos. Para poder reforzar la narrativa, entonces, es importante que el género del juego sea uno adecuado, pero también es importante que el juego no sea muy complicado mecánicamente, para que pueda llegar a tantas personas como sea posible en lugar de limitarse a un público más hardcore. Es por esto que el juego combinará dos géneros: los plataformas 2D, dada su simplicidad y fácil adaptación por parte del público casual, y los RPG, dada su inmersión y exposición de historia.

Además, una ventaja adicional de combinar estos dos géneros es el hecho de que no es una combinación muy usual, por lo que puede llamar la atención de posibles jugadores interesados y así atraerlos a jugarlo, que al fin y al cabo es el objetivo principal para que pueda extender su mensaje.

1.4 Actores implicados

Seguidamente se muestra un listado ordenado alfabéticamente de los actores implicados en el proyecto ya sean personas interesadas, afectadas o los participantes del proyecto.

1.4.1 Artista

Para poder llevar a cabo este proyecto y que el producto final tenga un acabado profesional, se ha considerado el hecho de contratar a un artista externo para plasmar las ideas de los personajes a la pantalla y que el videojuego resultante tenga personalidad propia y un estilo diferenciado de otros productos.

1.4.2 Equipo de desarrollo

El equipo de desarrollo estará formado en su totalidad por la autora del TFG, y tendrá como responsabilidad definir arquitectónicamente cómo será el código, diseñar los diferentes aspectos que compondrán el videojuego, implementar todo el código relacionado con el susodicho y hacer pruebas y tests para comprobar el correcto funcionamiento de todo el código.

1.4.3 Tutor

El tutor de este proyecto se trata de Marc Alier Forment y se encargará de revisar y el proceso de documentación y desarrollo dando retroalimentación cuando sea necesario con tal de realizar el proyecto de manera satisfactoria.

1.4.4 Usuarios finales

Los usuarios finales serán aquellas personas que adquieran el videojuego una vez lanzado al mercado. El mayor beneficio que obtendrán a parte de disfrutar de un tiempo de ocio, será el de poder empatizar y comprender mejor a las personas trans gracias a la trama y los personajes que componen el juego.

2. Justificación

Se ha de realizar un estudio de mercado para estudiar y analizar distintos juegos, tanto de plataformas como RPG, y ver qué funcionalidades ofrecen y qué hace que se diferencien de sus competidores.

En cuanto a reutilización de código de una solución previa no va a ser posible debido a que este proyecto pretende realizar un videojuego desde cero sin reutilizar nada excepto assets.

En el estudio de mercado nos centraremos en dos tipos de productos, videojuegos de plataformas y videojuegos RPG.

2.1 Videojuegos de plataformas

Debido a la cantidad tan elevada de videojuegos de plataformas en el mercado, se han seleccionado tres productos suficientemente diferentes entre ellos de los que inspirarse y tomar referencias.

2.1.1 Super Mario

Los videojuegos de Super Mario creados por Nintendo [2], son el referente en cuanto a videojuegos de plataformas se refiere. Desde sus orígenes, Mario Bros. no se ha

caracterizado por unas mecánicas complejas ni una historia intrigante y con gancho demostrando que a veces lo simple es mejor.

2.1.2 Castlevania

La saga de videojuegos de Castlevania [3] ha sufrido una clara evolución a lo largo de los años. Al principio constaba de una serie de escenarios, con un mapeado muy lineal y directo, que había que completar usualmente derrotando a un jefe y finalmente a Drácula y cambió a otro tipo de videojuego donde se recompensa la exploración y revisitar antiguos mapas con mejoras que se obtienen en puntos más avanzados en la historia. Este tipo de videojuego se conoce como “Metroidvania” en honor a Metroid y Castlevania.

2.1.3 Hollow Knight

Hollow Knight [4] es un buen referente en el que fijarse, debido a que es un juego indie realizado por Team Cherry un equipo de cinco personas en Unity y ha explotado en popularidad y ventas.

2.2 Videojuegos RPG

Igual que con el apartado anterior, existe una gran variedad de juegos RPG en el mercado pero solo estudiaremos aquellos que nos resulten más interesantes a la hora de extraer ideas.

2.2.1 Bravely Default

Bravely Default [5] es un RPG por turnos que sigue la clásica historia de los primeros Final Fantasy de salvar los cuatro cristales elementales pero con una vuelta de tuerca. Lo realmente interesante de su sistema de combate es el hecho de que puedes acumular turnos para realizar ataques más poderosos o varios ataques a la vez, dando así una nueva capa de profundidad que no se encuentra en la mayoría de RPGs.

2.2.2 Octopath Traveller

Similar a Bravely Default, Octopath Traveller [6] fue desarrollado por Square Enix y añade una mecánica similar al del susodicho haciendo que puedas acumular puntos de acción para realizar versiones más poderosas de ataques y hechizos. En cuanto a historia, narra las aventuras de ocho diferentes personajes y cómo al final todas las historias están relacionadas.

2.3 Conclusión

Como individuo no resulta una opción viable reaprovechar o adaptar un código ya existente porque primero, los juegos sobre los que se tomarán referencias no son open source y su

código es privado y segundo, la idea del proyecto es realizar de cero un videojuego y poder comercializarlo.

Se ha podido observar al hacer este pequeño estudio que se han realizado ya anteriormente juegos exitosos de ambos tipos de jugabilidad y con las funcionalidades deseadas. Sin embargo no hay ningún videojuego popular que junte ambos tipos de jugabilidades.

Además este proyecto cuenta con que la historia y los personajes tratan de transmitir un mensaje de progreso y tolerancia hacia un colectivo del cual la sociedad por lo general no tiene mucha idea por lo que tiene que pasar. Así que este motivo es uno de peso a la hora de decidir realizar el trabajo.

3. Alcance

En este apartado se concreta el alcance del proyecto especificando los distintos objetivos del proyecto, los requisitos no funcionales y qué posibles riesgos y obstáculos que pueden aparecer durante el desarrollo del trabajo.

3.1 Objetivos principales del proyecto

A continuación se muestran los objetivos principales que se tienen que haber cumplido al finalizar el proyecto.

3.1.1 Diseñar el videojuego

Antes de poder empezar a desarrollar el videojuego en sí, hace falta especificar clara y concisamente en qué consistirá el videojuego. Para poder realizar el diseño completo, hay que confeccionar los distintos elementos que formarán el producto final.

Subobjetivos

1. Diseñar las mecánicas que compondrán el juego.
2. Diseñar el mapeado que formará las distintas localidades del videojuego.
3. Diseñar una historia y personajes que transmitan las experiencias y sensaciones que experimenta una persona que está transicionando con el fin de concienciar a los jugadores sobre el tema.
4. Documentar todo el proceso de diseño.

3.1.2 Programar la sección de plataformas

Como anteriormente se ha mencionado, el juego contará con dos tipos de jugabilidad bien diferenciadas, en este apartado se hablará de los diferentes subobjetivos relacionados con esta fracción.

Subobjetivos

1. Un jugador ha de poder realizar saltos que resulten satisfactorios.
2. Un jugador ha de poder enfrentarse a enemigos y derrotarlos en tiempo real con las habilidades que le ofrece el juego.
3. El videojuego ha de contener recompensas que premien al jugador por explorar el mapeado y derrotar enemigos en forma de experiencia y objetos que le ayudarán a avanzar en la trama.

3.1.3 Programar la sección RPG

A continuación se listan los distintos subobjetivos que se han de cumplir para considerar realizado el objetivo global.

Subobjetivos

1. El jugador ha de poder enfrentarse a enemigos poderosos que supongan un desafío estratégico al jugador en un sistema de combate por turnos.
2. El juego ha de ofrecer un sistema de combate por turnos pero con alguna mecánica adicional que haga entretenido para los estándares de hoy un sistema así, tal y como hacen en “Bravely Default” u “Octopath traveller”.

3.2 Requisitos no funcionales

Seguidamente se listan los diferentes requisitos no funcionales que ha de cumplir el producto final.

3.2.1 Apariencia

El producto final ha de ser agradable a la vista y satisfactorio a nivel estético para así atraer y retener la atención de los posibles usuarios finales. Ya que es un hecho que hoy en día con la sobresaturación del mercado de los videojuegos, un producto ha de destacar e intentar diferenciarse de los demás [7].

3.2.2 Extensibilidad

El producto ha de poder permitir un incremento en el número de funcionalidades que ofrece de forma que sea fácilmente programable. Así en un posible futuro se pueden lanzar actualizaciones sin que sea un problema ampliar las funciones del juego.

3.2.3 Fiabilidad

Los datos de las partidas guardadas no se pueden perder bajo ningún concepto dado que es lo que marca el progreso del jugador una vez esté usando el producto final y una pérdida de estos datos frustraría en sobremanera al jugador y posiblemente abandone el videojuego sin llegarlo a completar.

3.2.4 Mantenibilidad

El tiempo medio a reparar un problema (MTTR) que surja tendrá que ser reducido de forma que los usuarios finales no se vean afectados durante mucho tiempo. Si pese a haber testeado y comprobado que todo funcione como es debido, un usuario final detecta un error crítico en el juego es importante que sea fácil de detectar y de arreglar porque si el tiempo de espera es demasiado largo, el jugador puede perder el interés y abandonar el producto.

3.2.5 Portabilidad

El programa final ha de poder ser fácilmente portable a distintos entornos y sistemas operativos. El hecho de que sea portable aumenta en gran medida el número de posibles usuarios ya que hay mucho mercado en el mundo de los videojuegos para móvil que no hay que pasar por alto.

3.2.6 Rendimiento

El videojuego ha de ser programado de tal forma que aproveche muy bien los recursos y pueda funcionar en máquinas poco potentes. Este hecho hace que cualquier persona pueda ser un posible consumidor.

3.2.7 Usabilidad

La interfaz de usuario ha de ser práctica y fácil de usar y de aprender. Debido a que es la forma que tiene el jugador de interactuar con el videojuego resulta crucial que sea satisfactorio.

3.3 Obstáculos y riesgos

Finalmente en este apartado se enumeran un un conjunto de posibles riesgos que pueden aparecer durante la realización del proyecto y que afectarían negativamente al desarrollo de este y por tanto también afectarían al valor y calidad del producto.

3.3.1 Limitación temporal en la entrega

El hecho de tener una fecha límite inamovible para la entrega final es un riesgo que puede hacer que un contratiempo externo que haga que el ritmo de trabajo se vea demorado afecte seriamente al producto final, ya que no se le dedicaría el tiempo necesario estimado al proyecto.

3.3.2 Falta de experiencia con la tecnología

Pese ya haber trabajado con Unity anteriormente, es un hecho que falta experiencia con el uso de este motor y por ello tiempo que se podría dedicar a la realización de avances en el proyecto se tendrán que dedicar a aprender.

3.3.3 Virus en el ordenador

Como cualquier sistema informático, puede suceder que en el ordenador personal donde se realizará el proyecto se corrompan los datos o un virus haga que malfuncione. En caso de que esto suceda, el proyecto no se verá gravemente afectado ya que todos los datos estarán en la nube y el progreso realizado no se perdería.

4. Metodología y rigor

La metodología que se seguirá durante el desarrollo del proyecto es Scrum adaptada para una persona. Se ha escogido esta metodología ya que fue la que se usó en la asignatura de Projecte d'Enginyeria del Software, y por lo tanto, gracias a esta experiencia previa dedicaré menos tiempo a aprender distintas metodologías y por lo tanto destinaré este tiempo a realizar trabajo útil.

Scrum [8] es una metodología de desarrollo del software ágil que se basa en aspectos como la flexibilidad en la adopción de cambios y nuevos requisitos durante el proyecto, el factor humano, y un desarrollo iterativo e incremental. Es iterativo porque se va trabajando de manera cíclica haciendo tareas de un backlog, e incremental porque en cada iteración se añade valor al producto final.

El backlog anteriormente mencionado es un conjunto de tareas basadas en los objetivos y requisitos del proyecto, y en cada iteración o sprint se cogen tareas del backlog a realizar.

A continuación se listarán un conjunto de herramientas que sirven para hacer un seguimiento del estado del proyecto y ayudan a obtener un desarrollo eficaz.

4.1 Herramientas de seguimiento

4.1.1 Git

Git [9] es un sistema de control de versiones open source. Es utilizado para desarrollar proyectos software de manera ordenada y para llevar a cabo un control de versiones y funcionalidades de manera eficiente y práctica.

4.1.2 Gitlab

Gitlab [10] es un gestor de repositorios de git de pago, sin embargo al ser estudiante de la FIB obtenemos una licencia gratuita que nos otorga acceso a estas funcionalidades, por lo que se puede usar en este proyecto sin aumentar los costes del proyecto.

4.1.3 GitFlow

GitFlow [11] es una metodología de trabajo con las ramas de git que dicta una cierta forma de organización a la hora de crear un proyecto.

A continuación se muestra una imagen representativa.

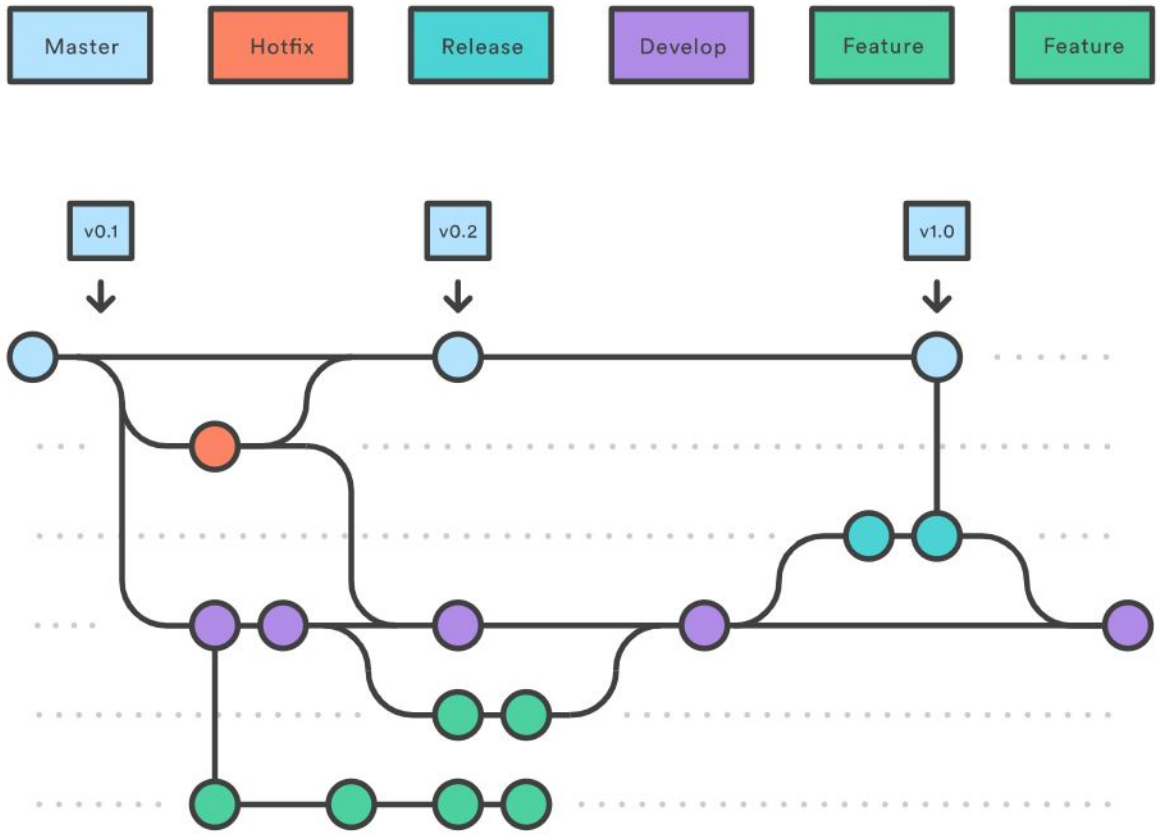

Figura 1. Estructura de ejemplo del modelo GitFlow

La rama Master es la rama donde se haya el código ya en entorno de producción, es decir el código ahí presente ha sido validado y testeado.

La rama develop es la que contiene las funcionalidades o features una vez han sido probadas y validadas.

La rama release tiene como origen develop y en ella se juntan varias funcionalidades ya validadas y se limpia el código antes de hacer el merge con master.

La rama feature contiene el desarrollo de una única funcionalidad y una vez esta esté realizada se junta con la rama develop.

La rama hotfix existe para solventar pequeños errores encontrados en el entorno de producción.

4.1.4 GitKraken

GitKraken [12] es una herramienta que permite controlar con una interfaz visual muy práctica un repositorio de git en vez de trabajar con la consola, es positivo ya que evita trabajar con comandos y facilita el trabajo de control de seguimiento.

4.1.5 Taiga

Taiga [13] es una herramienta de gestión de proyectos que te permite gestionar un backlog creando los sprints y asignándole tareas de este.

4.2 Validación de objetivos

Para validar objetivos, al final de cada sprint se valorará si se han completado las tareas correspondientes y en caso de no ser así se considerará incrementar la carga de trabajo para poder ajustarse a la planificación.

La validación se hará a mano, ya que la naturaleza de este proyecto permite comprobar si una funcionalidad está implementada correctamente o no, probando a jugar y observar que el comportamiento obtenido es el esperado.

5. Descripción de las tareas

5.1 Gestión del proyecto

Esta tarea hace referencia al curso de GEP el cual forma parte del trabajo de fin de grado de la FIB. En este curso se pretende aprender a cómo realizar una planificación del proyecto y documentarlo. Además también consiste en el conjunto de entregas siguiente:

1. **[T1] Contexto y alcance (25 horas).** Se realizará la redacción de la primera entrega del GEP la cual define el contexto, alcance y posibles riesgos del proyecto además de especificar la metodología que se seguirá. No tiene dependencias.
2. **[T2] Planificación temporal (15 horas).** Se realizará la redacción de la segunda entrega del GEP que tiene que ver con la planificación del proyecto y ofrecer posibles soluciones a diferentes problemas que puedan suceder. Esta tarea depende de **[T1]**.
3. **[T3] Presupuesto y sostenibilidad (15 horas).** Consiste en la redacción de la tercera entrega del GEP la cual es un documento con el presupuesto del proyecto y su sostenibilidad. Para realizar esta tarea, **[T2]** tiene que estar finalizada.

4. **[T4] Evaluación final (25 horas).** Esta es la última tarea relacionada con la gestión de proyectos y en ella se hará el documento final reuniendo las tareas **[T1, T2, T3]** y mejorando los escritos de estas gracias a la retroalimentación recibida a lo largo de estas semanas de cada entrega. Depende de haber acabado **[T3]**.

Para poder realizar estas tareas se requiere de un dispositivo con conexión a internet, preferiblemente un ordenador, además es necesario el uso de Google Drive para poder redactar los distintos entregables, TeamGantt para crear el Gantt necesario en **[T2]** y el Racó y Atenea para entregar los susodichos entregables.

5.2 Familiarización con el entorno

Aunque ya posea conocimientos de Unity, estos no son suficientes para este proyecto y por ello habré de dedicar horas a leer manuales y mirar tutoriales para ampliar mis conocimientos.

1. **[T5] Ampliar conocimientos de Unity (15 horas)**. No tiene dependencias.

5.3 Diseño del videojuego

Antes de empezar a programar, el primer paso es diseñar el videojuego a crear. Se han distinguido cinco diferentes tareas relacionadas con el diseño:

1. **[T6] Análisis de referencias (10 horas)**. Antes de empezar a diseñar mis propias mecánicas y escenarios, se tendrá que estudiar otros productos ya existentes y exitosos, tanto plataformas como RPG, que hay en el mercado para así tener referencias sobre las cuales basarme. Esta tarea no tiene dependencias.
2. **[T7] Diseño de las mecánicas (35 horas)**. Esta tarea consiste en especificar de qué mecánicas consistirá el videojuego, tanto para la parte de plataformas como la RPG. Para poder realizar esta tarea se tiene que haber finalizado **[T6]**.
3. **[T8] Diseño del mapeado (15 horas)**. En esta tarea se crearán los distintos mapas del videojuego en la parte de plataformas gracias al uso del programa Tiled. Tiene como dependencia haber acabado la tarea **[T6]**.
4. **[T9] Diseño de la historia (15 horas)**. Se creará la historia que compone la trama principal del juego. No tiene dependencias.
5. **[T10] Diseño de los personajes (10 horas)**. Se crearán los diferentes personajes que formarán parte del videojuego y se verán envueltos en la historia. Esta tarea no tiene dependencias.

5.4 Programación del videojuego

Una vez esté el diseño finalizado, se puede empezar a programar el videojuego. Se han distinguido estas diferentes tareas:

1. **[T11] Programar la fracción correspondiente al formato de plataformas (125 horas)**. Esta tarea hace referencia a la programación de una de las dos partes principales del videojuego, en este caso la de plataformas. Para poder realizar esta tarea, **[T7]** y **[T8]** tienen que estar finalizadas.
2. **[T12] Programar la fracción correspondiente al formato de RPG (75 horas)**. Similar a la tarea anterior, pero esta se refiere al componente RPG del juego, e igual que antes,**[T7]** y **[T8]** tienen que haber sido realizadas.
3. **[T13] Testear el programa (50 horas)**. En esta tarea se testeará que todo lo programado funcione correctamente. Para poder empezar esta tarea, **[T11]** o **[T12]** tienen que haber avanzado pero no necesariamente finalizado por eso no es dependiente completamente de estas dos tareas.

5.5 Hito final

Finalmente, el hito final consta de dos tareas, la documentación del proyecto en sí y la preparación de la defensa ante el tribunal.

1. **[T14] Redacción de la memoria (60 horas).** La tarea consiste en finalizar de escribir la memoria final. Esta tarea tiene como dependencias cada tarea del diseño y del desarrollo **[T5 - T13]**.
2. **[T15] Preparación de la defensa (20 horas).** Se preparará una presentación con tal de mostrar al tribunal mi proyecto. Debido a que es la tarea final tiene como dependencia **[T14]**.

6. Tabla de tareas

Id.	Descripción	Tiempo estimado	Dependencias
-	Gestión del proyecto	75 horas	-
T1	Contexto y alcance	20 horas	-
T2	Planificación temporal	15 horas	T1
T3	Presupuesto y sostenibilidad	15 horas	T2
T4	Evaluación final	25 horas	T3
-	Familiarización con el entorno	15 horas	-
T5	Ampliar conocimientos de Unity	15 horas	-
-	Diseño del videojuego	85 horas	-
T6	Análisis de referencias	10 horas	-
T7	Diseño de las mecánicas	35 horas	T6
T8	Diseño del mapeado	15 horas	T6

T9	Diseño de la historia	15 horas	-
T10	Diseño de los personajes	10 horas	-
-	Programación del videojuego	250 horas	-
T11	Programar la fracción correspondiente al formato de plataformas	125 horas	T7 - T8
T12	Programar la fracción correspondiente al formato RPG	75 horas	T7 - T8
T13	Testear el programa	50 horas	-
-	Hito final	80 horas	-
T14	Redacción de la memoria	60 horas	T5 - T12
T15	Preparación de la defensa	20 horas	T14

Figura 2. Tabla con la estimación temporal de las tareas a realizar

7. Concurrencia entre tareas

En esta sección se explican qué tareas se realizarán de forma concurrente durante el proyecto.

- La redacción del documento final del GEP **[T4]** y el análisis de referencias para la parte de diseño **[T6]** se pueden realizar de manera simultánea ya que el proyecto estará bien definido y se podrá analizar lo necesario para el juego.

- Ampliar conocimientos de Unity **[T5]** se puede realizar concurrentemente con cualquier tarea mientras esta sea anterior a la programación. Así que se ha decidido que se hará junto con las tareas de diseño **[T7 - T10]**.

- El diseño de las mecánicas **[T7]** y el diseño del mapeado **[T8]** se realizarán de forma concurrente ya que ambas tareas están muy relacionadas. Lo mismo sucede con tareas de diseño de historia y personajes **[T9]** y **[T10]** respectivamente.

- Las tareas **[T11]** y **[T12]** que hacen referencia a la programación del videojuego, tanto la parte de plataformas como la RPG se pueden realizar, y de hecho es lo aconsejable, concurrentemente con la de testeo **[T13]**.

8. Gantt

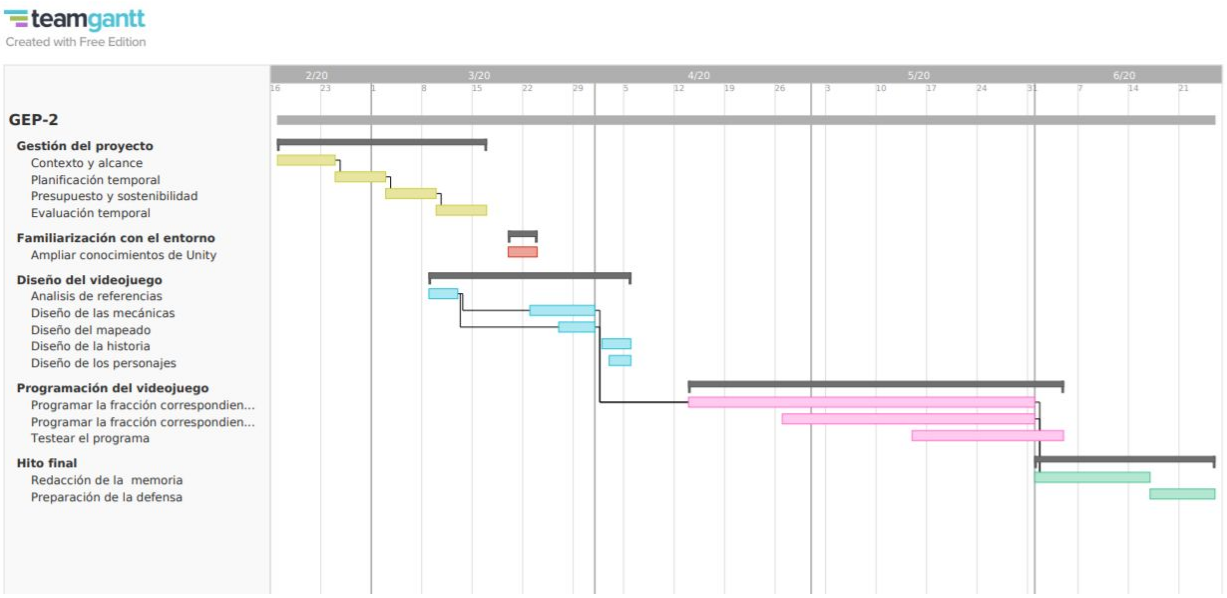

Figura 3. Gantt que contiene la planificación temporal del proyecto

9. Recursos

En este apartado se especificarán los diferentes recursos que se prevé que serán necesarios a lo largo del proyecto.

9.1 Recursos humanos

La programadora será la principal responsable del proyecto, pero también se verán involucrados el director del proyecto, los profesores responsables del GEP y el posible artista contratado.

9.2 Recursos físicos

El principal recurso indispensable para la posible realización de este proyecto es un ordenador en el cual poder trabajar tanto en la programación como la redacción del documento y además también es necesario un espacio de trabajo.

9.3 Recursos software

- Google Drive: se usará para la redacción de los diferentes entregables de GEP y para la memoria final.
- Unity: motor con el cual se realizará el videojuego.
- Atenea: página web relacionada con GEP y sus profesores.
- Racó: página web relacionada con GEP y sus profesores
- Windows 10: sistema operativo del ordenador donde se realizará el proyecto.
- TeamGantt: página web con la que se realiza el gantt.

- Taiga: página web para la organización de los sprints.
- Gitlab: gestor de repositorios de git.
- GitKraken: herramienta que permite controlar de forma visual un repositorio de git.

10. Gestión del riesgo

Durante el desarrollo del proyecto, pueden aparecer diferentes riesgos e imprevistos que pueden hacer que se demore el trabajo y la planificación temporal no se cumpla. Por ello se habla en este apartado sobre las posibles soluciones a los riesgos.

- **Mala planificación temporal**. Debido a que el proyecto aún está en sus primeras fases, resulta probable el hecho de haber estimado mal el tiempo necesario para la realización de una tarea. Es por eso que se han planeado tantas horas a la hora de programar, porque así si surge cualquier imprevisto se puede resolver dentro del tiempo estimado, y si aún así no es suficiente se pueden ampliar las horas dedicadas a la tarea en cuestión y en el peor de los casos se podría reducir el scope del proyecto.

- **Virus o error crítico en el ordenador**. Es un hecho que por muy improbable que sea, el ordenador donde se realizará el proyecto podría malfuncionar, pero con tal de no perder el progreso, todo el proyecto estará en la nube en gitlab, tal y como se especificó en el apartado de metodología y rigor de la entrega anterior.

- **Dificultades técnicas imprevistas**. Pese a haber dedicado toda una tarea para ampliar los conocimientos en Unity, siempre cabe la posibilidad de que aún así surjan diferentes dificultades respecto al motor. Para solventar este riesgo habría que mirar en diferentes foros sobre el problema en cuestión y dedicar más tiempo a aprender Unity.

11. Presupuesto

En esta sección se especificará la gestión económica del proyecto. En el anterior entregable se detallaron diferentes recursos necesarios para la realización del proyecto: humanos, hardware y software; y son sobre estos sobre los cuales se cataloguizarán los diferentes costes. También hay que tener en cuenta costes indirectos que podrían ser las diferentes prestaciones presentes en el entorno de trabajo.

En los diferentes apartados se podrán encontrar con detalle los costes, amortizaciones, contingencias e imprevistos. También se incluye más adelante un apartado con el control de gestión de las posibles diferentes desviaciones que afectarían al presupuesto, con indicadores numéricos de cálculo que facilitarán el control.

11.1 Costes humanos

El coste de las personas se puede estimar fácilmente sabiendo la cantidad de horas que cada rol desempeñará en el proyecto y cuánto cobra ese rol de media por hora. Por ello se ha de realizar una búsqueda del sueldo de estos roles.

En este proyecto se han detectado cuatro diferentes roles, el diseñador gráfico, el desarrollador de software, el gestor de proyectos y el tester. Con la ayuda de la página web https://www.payscale.com/research/ES/Country=Spain/Salary hemos podido establecer el sueldo medio por hora de estos diferentes trabajos.

Rol	Sueldo (€/h)
Diseñador gráfico	15,47
Desarrollador software	24,63
Gestor de proyectos	32,53
Tester	24,94

Figura 4. Roles del proyecto y sus sueldos

En la siguiente tabla se muestran cuántas horas dedica cada rol a las diferentes tareas del proyecto definidas en la planificación temporal y el Gantt.

Tarea	Horas (h)	Horas por rol (h)			
		Dis.gráfico	Des. software	Gestor	Tester
Gestión del proyecto	75	-	-	75	-
Familiarización con el entorno	15	-	15	-	-
Diseño del videojuego	85	25	60	-	-
Programación del videojuego	250	-	200	-	50
Hito final	80	-	60	20	-
Total	505	25	335	95	50

Figura 5. Tabla con la distribución de las horas por rol

Una vez tenemos estos datos podemos estimar el coste humano del proyecto.

Rol	Horas (h)	Sueldo (€/h)	Coste (€)
Diseñador gráfico	25	15,47	386.75
Desarrollador software	335	24,63	8251.05

Gestor de proyectos	95	32,53	3090.35
Tester	50	24,94	1247
Total	505	-	12975.15
Total + Seguridad Social (30%)	505	-	16867.70

Figura 6. Tabla con el coste de los recursos humanos

Debido a que los imprevistos y desviaciones del diagrama de gantt estimado afectan directamente a la cantidad de horas de una tarea, el coste real puede desviarse al estimado ahora.

11.2. Costes hardware

En la siguiente tabla aparecen los costes especificados en el anterior entregable en el apartado de recursos hardware.

Hardware	Coste (€)	Vida útil (años)	Amortización (€)
Ordenador de sobremesa	1000	5	200
Televisor Qilive	200	6	33.33
Teclado hp	15	4	3.75
Ratón hp	10	4	2.5
Total	1025	-	239.58

Figura 7. Tabla con el coste de los recursos hardware

Debido a la naturaleza de este tipo de recursos, la variabilidad es prácticamente nula, el único motivo por el que los costes aumentarían sería debido al malfuncionamiento de algún elemento.

11.3 Costes software

Todo software que se utilizará en el proyecto es libre y por lo tanto no conlleva ningún coste, a excepción de windows 10 y GitLab que serían de pago pero al ser estudiante de la UPC, se pueden adquirir de forma gratuita.

Software	Coste (€)	Vida útil (años)	Amortización (€)
Google Drive	0	-	0
Unity	0	-	0
Atenea	0	-	0
Racó	0	-	0
Windows 10 Home	145	-	0
TeamGantt	0	-	0
Taiga	0	-	0
GitLab	99	1	0
GitKraken	0	-	0
Total	145	-	0

Figura 8. Tabla con los costes software

Cabe destacar que como posible desviación en el coste sería utilizar software de pago que no se ha previsto utilizar.

11.4 Costes indirectos

Este apartado hace referencia a los gastos del entorno de trabajo, en este caso mi habitación, y las prestaciones de este.

Recurso	Coste (€)	Vida útil (meses)	Amortización (€)
Habitáculo	4000	180	88.88
Mobiliario	1000	120	33.33
Recursos	800	4	800
Transporte	80	4	80
Total	5880	-	1002.21

Figura 9. Tabla con los costes indirectos del proyecto

Hay que tener en cuenta que la duración del proyecto son cuatro meses.

12. Control de gestión

Pese a haber mencionado en otras entregas algunas técnicas para mitigar contratiempos y desviaciones, es muy probable que haya ciertos desvíos en el proyecto que afecten a la planificación y por ende al presupuesto.

El presupuesto destinado a los recursos humanos, es el que más probabilidades tiene de verse afectado por estas desviaciones. Las horas que se conjeturaron en el anterior entregable, son aproximadas y no se puede saber a ciencia cierta a priori cuáles serán sus valores reales.

En cambio el presupuesto para recursos hardware es muy estable ya que el coste dado es exacto y el único motivo por el que debería aumentar es por un fallo grave, cuya probabilidad es baja.

En cuanto al presupuesto que se ha destinado para los recursos software solo incrementaría si se usara un software de pago que no está previsto utilizar.

Los costes indirectos son muy aproximados y no se pueden especificar mecanismos para controlar desviaciones.

En la siguiente tabla se especifican los costes de contingencia e imprevistos.

Actividad	Coste (€)	Riesgo	Observaciones
Contingencia	3373.54	-	Se trata de un presupuesto que es un margen de seguridad calculado como el 20% del coste de recursos humanos.
Fallo crítico en el ordenador Coste ~= (1000€)	150	15%	Si fallara por completo el ordenador, el coste especificado es el de reemplazarlo por otro.
Total	2745.03	-	-

Figura 10. Tabla con los costes de contingencia

Para poder controlar la desviación se usará el siguiente indicador numérico a las diferentes partidas anteriormente calculadas.

$$d = \frac{100 * (Creal - Cest)}{Cest}$$

Siendo Creal el coste real calculado gracias al coste de cada recurso y el tiempo que realmente se ha empleado y donde Cest es el coste estimado para la misma partida. El

resultado de esta ecuación es el porcentaje de desviación. Si este supera el 20%, superará los fondos que se destinaron para la contingencia.

13. Presupuesto final

Ahora que ya se han calculado todos los costes, es sencillo calcular el coste total del proyecto.

Recurso	Coste (€)
Recursos humanos	16867.70
Recursos hardware	239.58
Recursos software	0
Costes indirectos	1002.21
Contingencia e imprevistos	3373.54
Total	21483.03
Total + IVA	25994.47

Figura 11. Tabla con el presupuesto final del proyecto

14. Sostenibilidad y compromiso social

14.1 Autoevaluación

Gracias a la tecnología disponemos de una serie de comodidades y avances que no somos capaces de vivir sin. Pero estos avances tienen un coste muy serio y grave para nuestro planeta ya que todo proyecto contamina y favorece el calentamiento global.

Además también hay consecuencias a un nivel más humano, por ejemplo en Ghana o en el pueblo de Guiyu, situado en China, van a parar una gran parte de los desechos de los aparatos electrónicos del primer mundo que se han quedado obsoletos. Este hecho hace que los ciudadanos de esos lugares tengan que vivir entre escombros e intentando encontrar piezas de valor de formas precarias y poco seguras.

Pero debido a que este proyecto es uno software, no se generarán residuos físicos que afecten a estas poblaciones del tercer mundo.

En cuanto a las dimensiones ambiental y social en un proyecto informático, soy parcamente conocedora de los impactos que pueden tener. Sin embargo, no soy muy consciente del impacto económico que puede llegar a tener, o al menos no lo era a priori, ya que gracias al

trabajo realizado en esta entrega me he dado cuenta de cuánto puede costar un proyecto por simple o corto que sea.

Hasta la fecha, cada vez que me embarcaba en un proyecto únicamente me fijaba en los objetivos y el beneficio de las partes implicadas, pero gracias a este trabajo también consideraré estos aspectos económicos ambientales y sociales.

14.2 Dimensión económica

-Reflexión sobre el coste que has estimado para la realización del proyecto

En el coste estimado del proyecto se han tenido en cuenta tanto recursos humanos hardware y software. Sin olvidarnos de los costes generales o indirectos y los riesgos y posibles desviaciones y cómo influirían en el presupuesto final.
Para los recursos humanos se ha buscado la información del salario medio en España de cada rol y se ha estimado el número de horas que desempeñará en el proyecto.

-**¿Cómo se resuelven actualmente los aspectos de costes del problema que quieres abordar (estado del arte)?**

Hoy en día siempre se intenta abaratar costes si se puede, esto se puede conseguir utilizando exclusivamente software libre, aumentando la carga de trabajo por hora de los trabajadores así pueden realizar el trabajo esperado en menos tiempo y por tanto disminuyendo el coste de recursos humanos.

- **¿En qué mejorará económicamente (costes...) tu solución respecto a las existentes?**

Económicamente este proyecto es realmente similar a cualquier otro proyecto parecido, la única diferencia sería que las horas de desarrollador software estarán hechas por mí, que como no poseo experiencia mi sueldo sería considerablemente más barato que el de la media.

14.3 Dimensión ambiental

-¿Has estimado el impacto ambiental que tendrá la realización del proyecto?

Debido a que es un proyecto software, ambientalmente no genera ningún residuo que pueda dañar el medioambiente. Lo único que consume este proyecto es electricidad y debido a que es simplemente un ordenador de sobremesa y no hace uso de máquinas externas el consumo es bajo.

-¿Te has planteado minimizar el impacto, por ejemplo, reutilizando recursos?

En este proyecto se reutilizarán recursos software en la forma de assets de la tienda de Unity. El trabajo del diseñador será el de hacer los personajes de una forma concreta para transmitir el mensaje deseado, pero hay sprites que se pueden reusar.

- ¿Cómo se resuelve actualmente el problema que quieres abordar (estado del arte)?, y. ¿En qué mejorará ambientalmente tu solución respecto a las existentes?

La vida útil de este proyecto es virtualmente infinita, ya que una vez realizado se publicará en Steam y ahí estará disponible en la tienda. Así que una vez finalizado ya no consumirá más recursos con la excepción de sacar algún parche.

14.4 Dimensión social

-¿Qué crees que te aportará a nivel personal la realización de este proyecto?

El hecho de crear un videojuego por mi cuenta me aportará mucho conocimiento para mi futuro además de ampliar mi currículum ya que me gustaría dedicarme profesionalmente al mundo de los videojuegos.

-¿Cómo se resuelve actualmente el problema que quieres abordar (estado del arte)?, y. ¿En qué mejorará socialmente (calidad de vida) tu solución respecto a las existentes?

Actualmente no muchos juegos o medios de entretenimiento tratan el tema trans como algo serio así que socialmente tendrá un impacto positivo para que la gente se mentalice sobre esta comunidad.

-¿Existe una necesidad real del proyecto?

Considero que hay una necesidad real para el proyecto ya que su objetivo es concienciar a la gente sobre las personas trans y que sepan qué es por lo que tienen que pasar en unas etapas tempranas del tratamiento hormonal.

15. Cambios respecto al GEP inicial

Debido a la pandemia y al impacto que ha tenido a nivel personal, el proyecto que estaba inicialmente planificado para exponer en julio, se ha retrasado hasta octubre. Es por ese motivo que la planificación real ha cambiado entre otros elementos que se explicarán a continuación.

15.1 Objetivos finales

Los objetivos finales del proyecto han cambiado debido a que al principio el proyecto fue planteado como un videojuego que mezclaba el género de plataformas sidescroller 2D con el de RPG al estilo Final Fantasy antiguo con un sistema de combate por turnos. Pero conforme avanzaba con el proyecto me di cuenta que considero más importante hacer un buen juego de plataformas con distintas mecánicas y que sea divertido y suponga un reto al jugador que no hacer una mezcla y que ambas partes sean de calidad regular u olvidables, así que decidí eliminar la parte RPG del proyecto. El hecho de incluir jugabilidad de estilo RPG en el proyecto nació de que es un género que realmente disfruto y porque para explicar una historia es el género por excelencia. Pero para solventar esto, se ha implementado un sistema de diálogos dentro del videojuego que permite mostrar los pensamientos de la protagonista, o una conversación entre dos personajes.

A continuación se muestran los objetivos finales del proyecto.

Diseñar el videojuego

Antes de poder empezar a desarrollar el videojuego en sí, hace falta especificar clara y concisamente en qué consistirá el videojuego. Para poder realizar el diseño completo, hay que confeccionar los distintos elementos que formarán el producto final.

Subobjetivos

1. Diseñar los distintos bloques que compondrán el mapeado.
2. Diseñar enemigos a los que el jugador se podrá enfrentar.
3. Diseñar el mapeado que formará las distintas localidades del videojuego y que incorporen las distintas mecánicas y enemigos.
4. Diseñar una historia y personajes que transmitan las experiencias y sensaciones que experimenta una persona que está transicionando con el fin de concienciar a los jugadores sobre el tema.
5. Documentar todo el proceso de diseño.

Programar el videojuego

En este apartado se hablará de los diferentes subobjetivos relacionados con la sección relacionada a la programación en sí del videojuego.

Subobjetivos

1. Un jugador ha de poder realizar saltos que resulten satisfactorios.
2. Un jugador ha de poder enfrentarse a enemigos y derrotarlos en tiempo real con las habilidades que le ofrece el juego.
3. Un jugador ha de poder ir encontrando los distintos bloques y enemigos de manera progresiva de forma que la experiencia sea entretenida.
4. Un jugador ha de poder guardar su progreso para así no perder el progreso al cerrar el programa.

15.2 Planificación final

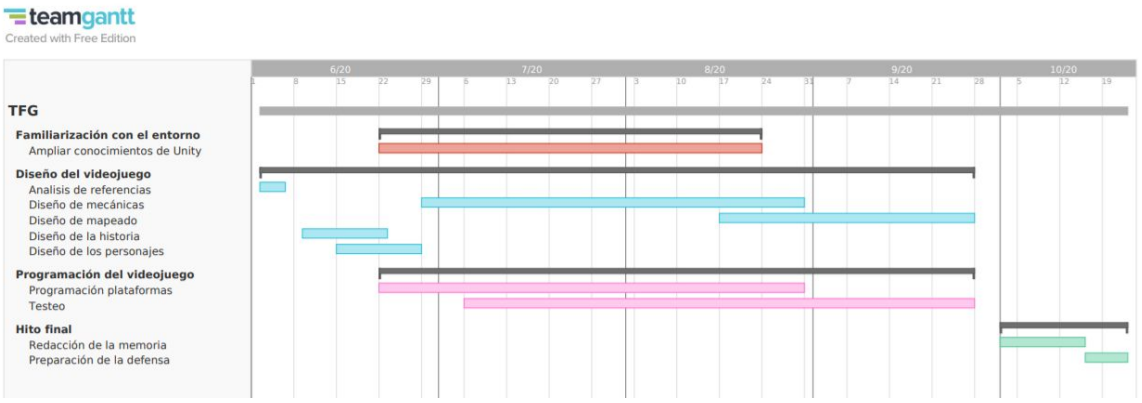

Figura 12. Gantt con la planificación final del proyecto.

La planificación final difiere respecto a la original en varios aspectos. El primero es que por motivos personales debidos a la pandemia actual los meses en los que se ha realizado el proyecto se han visto desplazados, el plan inicial era acabar en julio.

El otro aspecto en que difiere es la dedicación de tiempo a distintas tareas, al principio se estimó que con un par de días mirando tutoriales de Unity tendría suficiente para poder realizar el proyecto; pero la realidad ha sido completamente distinta. Durante este proyecto no he parado de buscar información de distintas fuentes para poder realizar las tareas propuestas. El diseño de mecánicas también fue otro punto que se calculó erróneamente ya que la realidad ha sido que mientras se programaba el videojuego, nuevas ideas iban surgiendo e implementando, y con estas nuevas ideas surgieron nuevos posibles niveles, y es por eso que esa tarea se ha visto alargada y solapada con el diseño de mecánicas.

En la primera versión de la planificación se puso la tarea de testeo muy avanzado ya el proceso de desarrollo pero a la que se tuvo una versión ya jugable se empezó a testear que todo funcionara.

16. GDD

Un GDD (game design document) es el documento que explica el diseño del videojuego en cuestión. Estos documentos se utilizan en la industria del videojuego para organizar al equipo ya que sirve de guía.

16.1 Personajes

16.1.1 Protagonista

En “Transcendental Journey”, tomamos el control de Aurora a lo largo de un viaje de introspección para descubrir quién realmente es y ganar la confianza y el apoyo suficientes para que el mundo la pueda aceptar.

Figura 13. Sprite de la protagonista

El diseño de la protagonista es bastante simple puesto que está realizado mediante la técnica de pixel art y debido a mi falta de experiencia artística el modelo se ha inspirado en uno ya existente, el de Curly Brace de Cave Story. Los colores que lleva en la ropa y el pelo; azul, rosa y blanco; hacen referencia a los colores de la bandera trans.

16.1.2 Coprotagonista

A lo largo del videojuego, nos encontraremos con Sam el cual aparece para ofrecernos su apoyo y ser la persona a la que recurrir cuando hayan problemas emocionales.

Figura 14. Sprite del amigo

Los colores del amigo están todos centrados alrededor del verde, que es el color de la tolerancia [14].

16.1.3 Antagonista

El antagonista del videojuego es una versión masculina de la protagonista. Ese aspecto es el que realmente tiene ella pero no es así cómo se siente en su interior: así pues, esta versión de ella representa los miedos e inseguridades de la protagonista. Los ojos amarillos son una referencia a la saga de videojuegos Persona, en específico a la entrega “Persona 4”, donde los protagonistas se enfrentan a su versión shadow, que se representa con los ojos amarillos y simbolizan aquello a lo que más temen.

Figura 15. Sprite del antagonista

Los colores de la chaqueta del antagonista están basados en la bandera heterosexual, que aunque no sea un concepto opuesto a la transexualidad, algunas personas malintencionadamente la usan como símbolo de odio hacia la comunidad LGBT.

16.2 Controles

Los controles de “Transcendental Journey” son simples, el jugador se mueve con la tecla ‘a’ para ir a la izquierda y ‘d’ para ir a la derecha, con la barra espaciadora es capaz de realizar saltos cuya altura depende del tiempo que esta se mantenga pulsada; además, estos saltos también se pueden realizar en la pared. Finalmente, con el click izquierdo se realiza un ataque cuerpo a cuerpo que sirve para derrotar a los distintos enemigos.

16.3 Enemigos y unidades

La única unidad jugable en “Transcendental Journey” es la protagonista, dado que la narrativa del juego va sobre su viaje y experiencias. Cuenta con un total de vida de 50 puntos y un daño de ataque de 10. Si este total de puntos de vida se reduce a 0, la protagonista morirá y hará respawn en el último checkpoint alcanzado con la vida al máximo. A parte de perder progreso volviendo hacia atrás en el nivel, no hay ninguna penalización extra ni una posibilidad de game over.

“Transcendental Journey” cuenta con dos tipos diferentes de enemigos: uno cuerpo a cuerpo y otro a distancia. Cada enemigo tiene una conducta distinta, siguiendo una inteligencia artificial basada en un sistema de Finite State Machine. Podemos ver un análisis de estos comportamientos en el apartado 17.3.

Figura 16. Sprite enemigo cuerpo a cuerpo

Figura 17. Sprite enemigo a distancia

16.4 Bloques

En “Transcendental Journey” hay diferentes bloques que componen el mundo. Aunque cada uno tiene unas propiedades distintas, se pueden dividir en dos grandes grupos: los que no son colisionables, cómo los checkpoints o la llave, y que en su mayoría son objetos para recoger o triggers para lanzar otros eventos; y los que sí son colisionables, que componen la estructura general de un nivel permitiendo el plataformeo. Los bloques colisionables lo son en todas las direcciones, es decir que no se puede saltar a través de ellos.

16.4.1 Tiles normales

El tipo de bloque más básico es el denominado tile normal. Estos son los tiles del tilemap que tienen el comportamiento simple esperado de un suelo en un videojuego: aguantan al personaje y enemigos sobre él y también sirve como pared que bloquea un camino o sobre la cual el jugador puede saltar. Aunque funcionalmente sean iguales, existen varias versiones del tile para añadir variedad gráfica.

Figura 18. Ejemplo de algunos tiles de suelo que componen el mapeado

16.4.2 Pinchos

Los tiles de pinchos componen el bloque más básico considerado un obstáculo para el jugador: si este entra en contacto con ellos, se le reduce la vida a 0 instantáneamente y muere, regresando al último checkpoint.

Figura 19. Tile de pinchos

16.4.3 Checkpoints

Los checkpoints son el bloque que permite a un jugador no perder su progreso en un nivel cada vez que muere. No son colisionables, y en su lugar cuando el jugador atraviesa uno, este guarda la posición del jugador para hacer respawn en él de vuelta cuando muera.

Figura 20. Tile de checkpoint

16.4.4 Bloque destruible

Los bloques destruibles son un tipo de bloque colisionable especial, dado que es un bloque que, cuando un jugador se posa encima de él, se comienza a destruir. Una vez destruído, pasados unos segundos el bloque vuelve a aparecer.

Figura 21. Animación del bloque destruible

16.4.5 Bloques azules y rojos

Los bloques azules y rojos son un tipo de bloques colisionables que se van activando y desactivando periódicamente, siguiendo el mismo ritmo constante. Además, están sincronizados de tal manera que cuando un tipo está activo el otro permanece inactivo y viceversa.

Figura 22. Animación bloque azul activo

Figura 23. Bloque azul inactivo

Figura 24. Animación bloque rojo activo

Figura 25. Bloque rojo inactivo

16.4.6 Llave y puerta

La llave es un bloque no colisionable la cual, cuando el jugador entra en contacto con ella, desaparece y se guarda en el jugador. Las llaves pueden estar colocadas visiblemente en un nivel o pueden ser contenidas por diferentes enemigos de manera que, al ser derrotados, las sueltan en el lugar en el que han muerto.

Las puertas son bloques colisionables que no se pueden atravesar a no ser que tengas una llave. Si el jugador posee una llave, al estar cerca de la puerta esta hará una animación y desaparecerá despejando el camino para el jugador.

Figura 26. Sprite llave

Figura 27. Sprite puerta

16.4.7 Bloques dispensadores de flechas

Estos bloques son colisionables y se mantienen fijos en un lugar disparando flechas constantemente en una dirección específica, que puede variar de un bloque a otro. Los parámetros de los distintos bloques pueden cambiar de uno a otro: además de la ya mencionada dirección de las flechas, otro parámetro variable es la distancia máxima que pueden recorrer las flechas generadas por ese bloque antes de ser afectadas por la gravedad.

Figura 28. Sprite dispensador de flechas, procedente del videojuego “Spelunky HD”.

16.4.8 Dialog Trigger

Estos son un tipo de bloques invisibles cuya finalidad es avanzar la trama llegados a ciertos puntos en un mapa. Dado que son invisibles, a ojos del jugador estos bloques no son perceptibles: la primera vez que atraviese uno, el juego se detiene para mostrar el diálogo asociado y, tras finalizarlo, el bloque desaparece permanentemente. Estos bloques pueden ocupar más de un tile, para garantizar que el jugador los atraviese cuando cruce por la zona correspondiente.

16.4.9 Estalactitas

Las estalactitas son un tipo de bloque que está flotando en el aire y a la que detectan al jugador pasar por debajo, se desprenden y caen, haciendo que si esta colisiona con el jugador, lo mate instantáneamente.

Figura 29. Sprite estalactita

16.4.10 Out Of Bounds

Los bloques de Out Of Bounds son un tipo de bloques invisibles que rodean un nivel y sirven para que, cuando el personaje caiga al vacío y entre en contacto con uno, este sea teletransportado al último checkpoint alcanzado.

16.4.11 Tiles fantasma

Estos bloques fantasma son un tipo de bloques que se ven exactamente igual que los tiles normales pero no se puede colisionar con ellos. Así, estos bloques cogen por sorpresa al jugador haciendo que caiga cuando esperaba poder andar sobre unos bloques. Se utilizan únicamente en un punto de todo el juego, y sirven un propósito narrativo más que jugabilistico dado que permiten expresar sorpresa o simbolizar situaciones inesperadas.

16.4.12 Rampas

A lo largo del desarrollo de este proyecto se invirtieron muchas horas en la incorporación de rampas en los niveles. Sin embargo, esos esfuerzos al final fueron descartados ya que pese a estar implementadas, aparecían muchos bugs referentes a ellas y, finalmente, tras analizar su aportación en comparación con otros bloques, no generaban ninguna situación interesante ni se podían combinar con otros bloques para favorecer un gameplay emocionante y variado, como es el caso de los otros bloques. Es por estos motivos por los que se decidió descartarlas, pese al tiempo invertido en su implementación.

16.5 Diseño de niveles

En el videojuego hay 12 niveles distintos, y estos han sido diseñados para mostrar de manera progresiva los diferentes bloques de forma que el juego se sienta fresco y presente nuevos retos. En algunos casos, el nivel también está diseñado para representar una emoción mediante una metáfora con las mecánicas del propio videojuego, complementando así su narrativa.

16.5.1 Nivel 1

La figura 29 corresponde al primer nivel al que se enfrentará el jugador. Puesto que es el primer nivel, es bastante sencillo y, tomando como inspiración el nivel 1-1 del Super Mario Bros de la NES, sirve para que el jugador aprenda por sí mismo las distintas mecánicas: en lugar de presentarle un tutorial informativo, el jugador pone en práctica lo que debe aprender [15].

Al principio del nivel se encontrará con un pequeño escalón que no podrá sortear simplemente andando así que tendrá que pulsar la barra espaciadora, descubriendo el salto. Acto seguido habrá un muro más elevado con el que simplemente pulsar la barra no será suficiente y tendrá que mantenerla para tener suficiente altura y poder saltar por encima, de esta forma el jugador comprenderá de forma interactiva que dependiendo de cuánto tiempo deje pulsada la barra espaciadora el personaje saltará más o menos alto.

A continuación el jugador se encontrará una señal con una calavera indicando peligro, y justo después un enemigo patrullando entre dos bloques: aquí el jugador podrá tanto enfrentarse con el enemigo para intentar derrotarlo o huir, presentándole ambas opciones.

Una vez continúe más a la derecha, se encontrará con una pared más alta de lo que su salto puede llegar a saltar incluso en su máximo potencial, pero tras intentarlo verá que el personaje al descender, se desliza por la pared. Así, si intenta saltar de nuevo verá que se impulsa hacia la otra pared y, en el proceso, gana altura. De esta forma descubre el salto entre paredes y finalmente puede obtener la altura necesaria para llegar a la plataforma donde le esperará un checkpoint que guardará su progreso en caso de morir.

En esta plataforma, a parte del checkpoint, se encuentra una señal de peligro para advertir al jugador de los pinchos que se encuentran debajo, fuera de cámara. Si el jugador no hace caso de la señal y baja directamente, acabará cayendo en los pinchos y morirá, así que se dará cuenta de que la forma correcta de descender es bajando poco a poco por la pared hasta visualizar el obstáculo y saltar. Una vez hecho eso llegará por fin al trofeo que simboliza el final del nivel y, tras recogerlo, podrá avanzar al siguiente nivel.

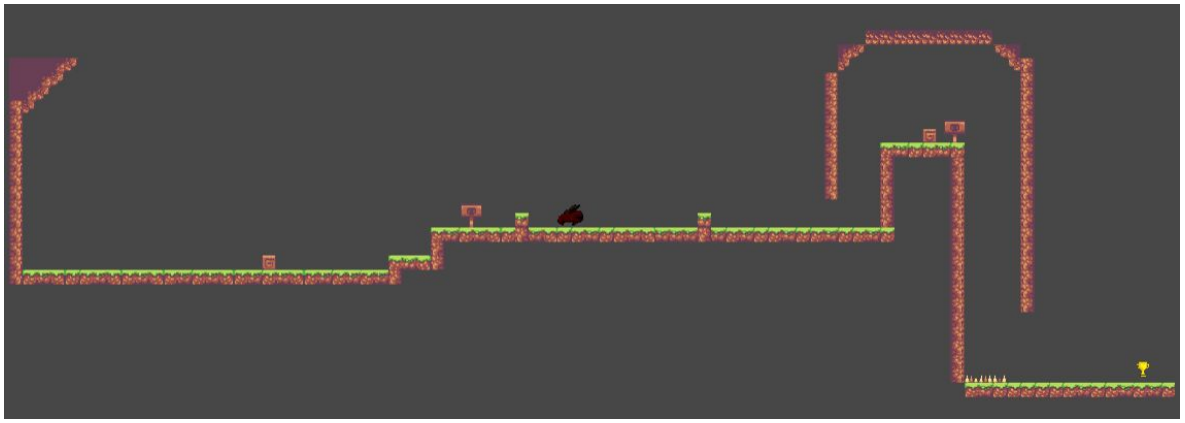

Figura 30. Diseño nivel 1

16.5.2 Nivel 2

Este es el segundo nivel del videojuego y está diseñado para que el jugador descubra los límites del salto y se acostumbre a las distancias que este es capaz de recorrer.

Al principio hay un único tile de pinchos y es fácilmente esquivable con un pequeño salto, después hay dos y finalmente tres antes de llegar al checkpoint. Una vez ahí hay un pequeño montículo que permite al jugador saltar sobre un cuarto tile de pinchos.

A continuación el jugador tendrá que saltar en la pared y desde la altura evitar caer en los pinchos, sortear unos obstáculos más un poco más desafiantes y finalmente llegar a la meta.

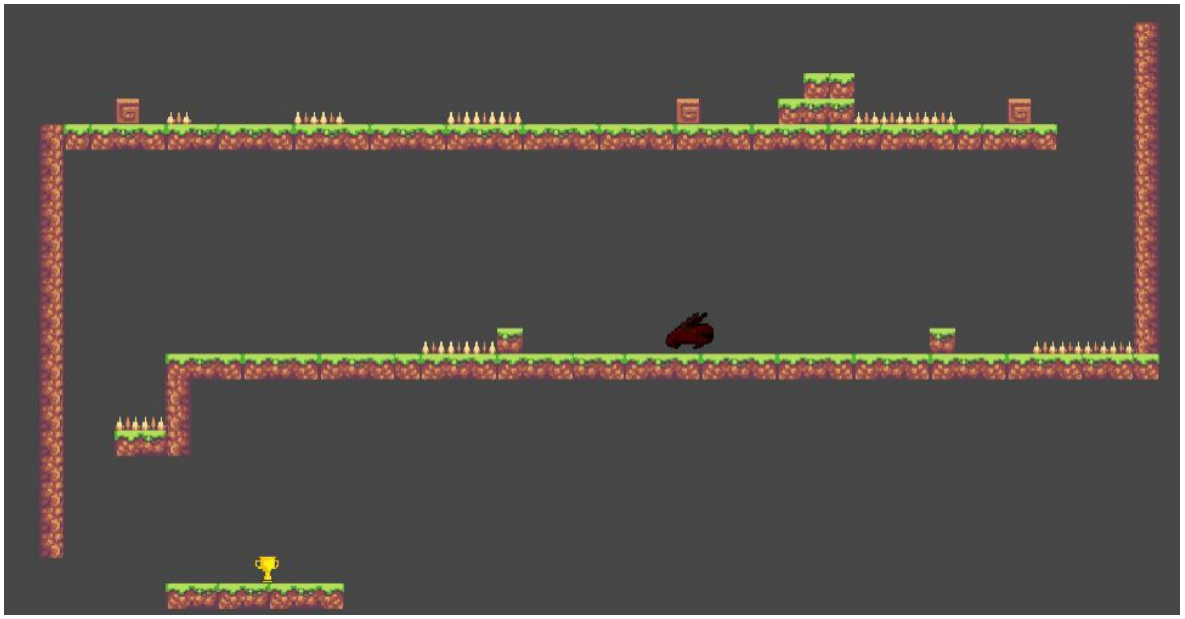

Figura 31. Diseño nivel 2

16.5.3 Nivel 3

Aquí se introduce por primera vez al amigo de la protagonista y podremos ver su personalidad y nos animará a seguir con esta lucha interna que tiene la protagonista.

En este nivel se ha querido otorgar protagonismo a los bloques alternantes azules y rojos, haciendo que para lograr pasarse el nivel, el jugador tenga que ir contando los ticks en su cabeza para saltar de un bloque a otro sin caerse, acostumbrándose así a saltar de unos a otros sin caer.

La sección del medio consiste en ir esquivando las flechas mientras que esperas que la plataforma desaparezca y puedas bajar a la siguiente.

Al final del nivel se han puesto dos bloques del mismo color para despistar a los jugadores y hacer que caigan, ya que rompe con el confort del compás aprendido y habrán de cambiar su tiempo en los saltos.

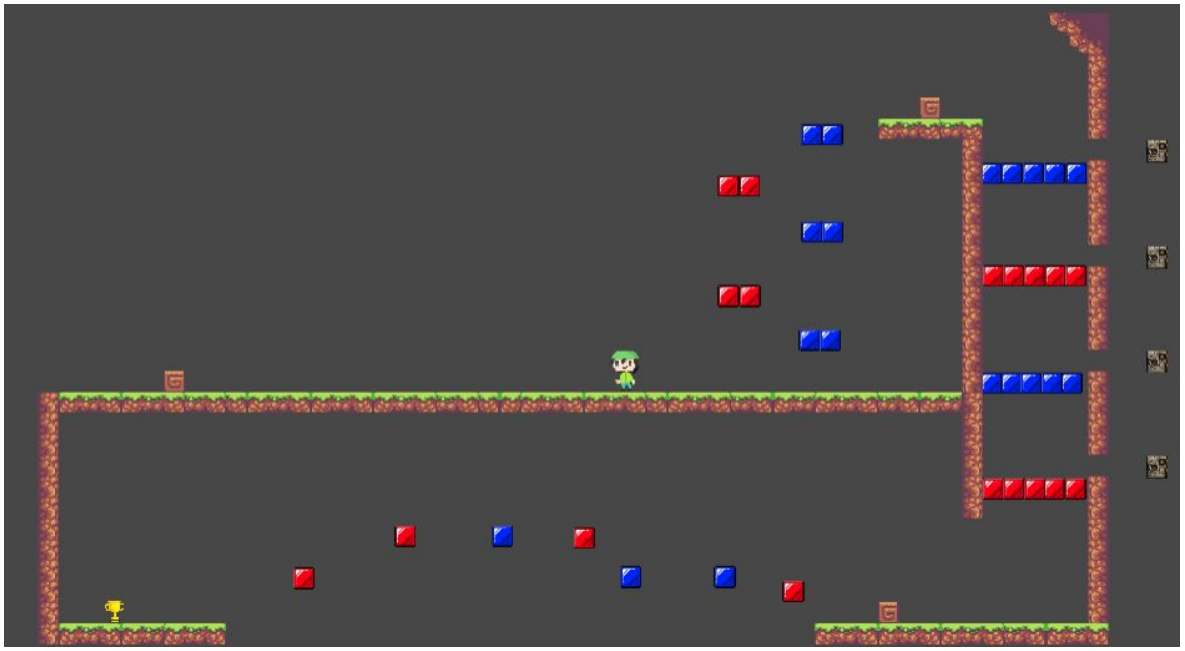

Figura 32. Diseño nivel 3

16.5.4 Nivel 4

Este sencillo nivel juega con el hecho de haber una puerta que bloquea el camino a la victoria y para alcanzarla hay que hacer el mismo recorrido de ida y de vuelta una vez conseguida la llave. Así, sirve para enseñar al jugador el funcionamiento de las llaves.

Figura 33. Diseño nivel 4

16.5.5 Nivel 5

En este nivel se introduce el hecho de que algunos enemigos pueden contener llaves, pero lo primero que tendrá que hacer el jugador es alcanzar la llave custodiada por el enemigo cuerpo a cuerpo para poder abrir la primera puerta.

Una vez abierta se liberará un enemigo y el camino estará bloqueado por otra puerta, por lo que lo único que puede hacer el jugador es enfrentarse a este enemigo y al derrotarlo verá que éste ha soltado una llave que le permite continuar.

Finalmente hay dos enemigos más y solo uno de ellos contiene la llave necesaria para alcanzar la meta.

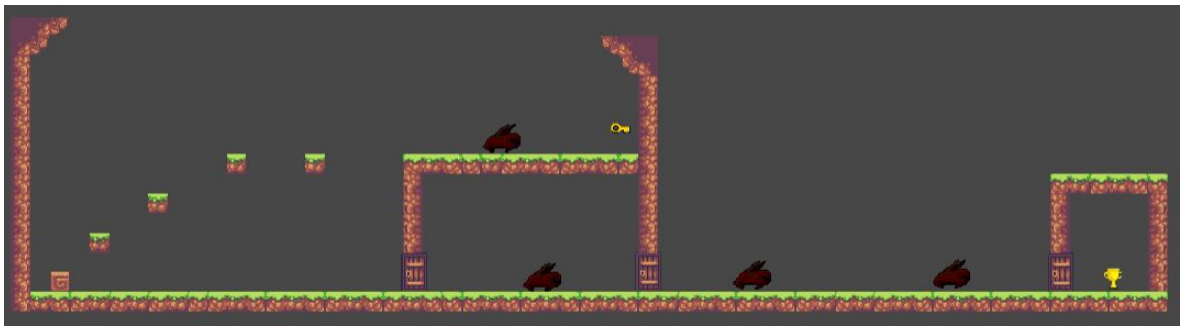

Figura 34. Diseño nivel 5

16.5.6 Nivel 6

Este nivel introduce los proyectiles, tanto por los dispensadores de flechas como por los arqueros.

Además para ver cómo funcionan los arqueros como enemigos, el juego te fuerza a matarlos para así obtener la llave que abre la puerta.

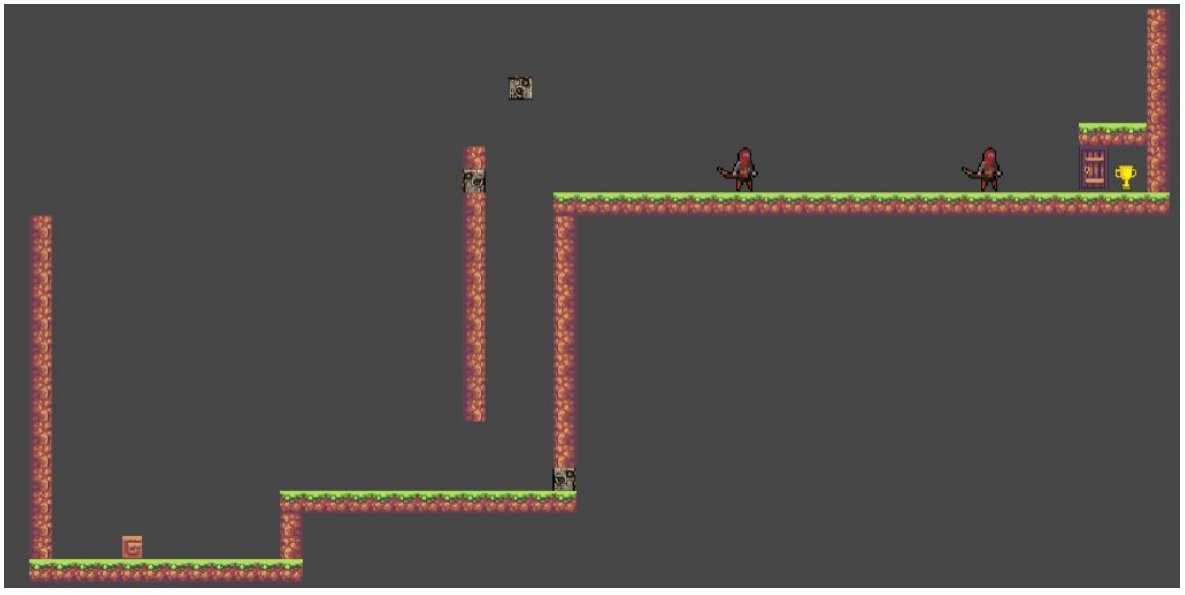

Figura 35. Diseño nivel 6

16.5.7 Nivel 7

En este nivel hay que saltar rápidamente entre los bloques que se destruyen para no caer al vacío intentando evitar las estalactitas. En el segundo tramo, además, un dispensador nos va tirando flechas constantemente.

Además, una vez llegados a la plataforma central nos encontramos junto a un checkpoint un enemigo que nos puede empujar al vacío, dificultando así la tarea de obtener el checkpoint antes de una posible muerte.

Debido a la estalactita que hay en la parte final, el jugador no puede simplemente saltar rápidamente hacia la meta, ha de ponerse primero debajo de la estalactita en la plataforma inferior y una vez caiga saltar por encima, todo esto mientras los bloques pisados se van destruyendo. Así pues, es una parte delicada que pone a prueba la habilidad del jugador.

Figura 36. Diseño nivel 7

16.5.8 Nivel 8

Este nivel empieza mostrando la meta al jugador en un punto aparentemente inalcanzable, y lo único que puede hacer es avanzar por el camino evitando pinchos y escalando la pared.

Una vez alcance el checkpoint seguirá bajando por las paredes sorteando los pinchos, una vez abajo el único camino posible parece que sea seguir bajando, pero eso lleva a la muerte del personaje.

El amigo está aquí para hacerle ver a la protagonista que hay que ver las cosas de otro modo, una pista que hace que el jugador llegue a la conclusión de que tiene que escalar la pared y recorrer el nivel por encima hasta llegar a la meta.

Este nivel es una metáfora que pretende ilustrar que aún cuando parece que no hay salida, siempre se puede pensar de forma diferente y poder ver una solución.

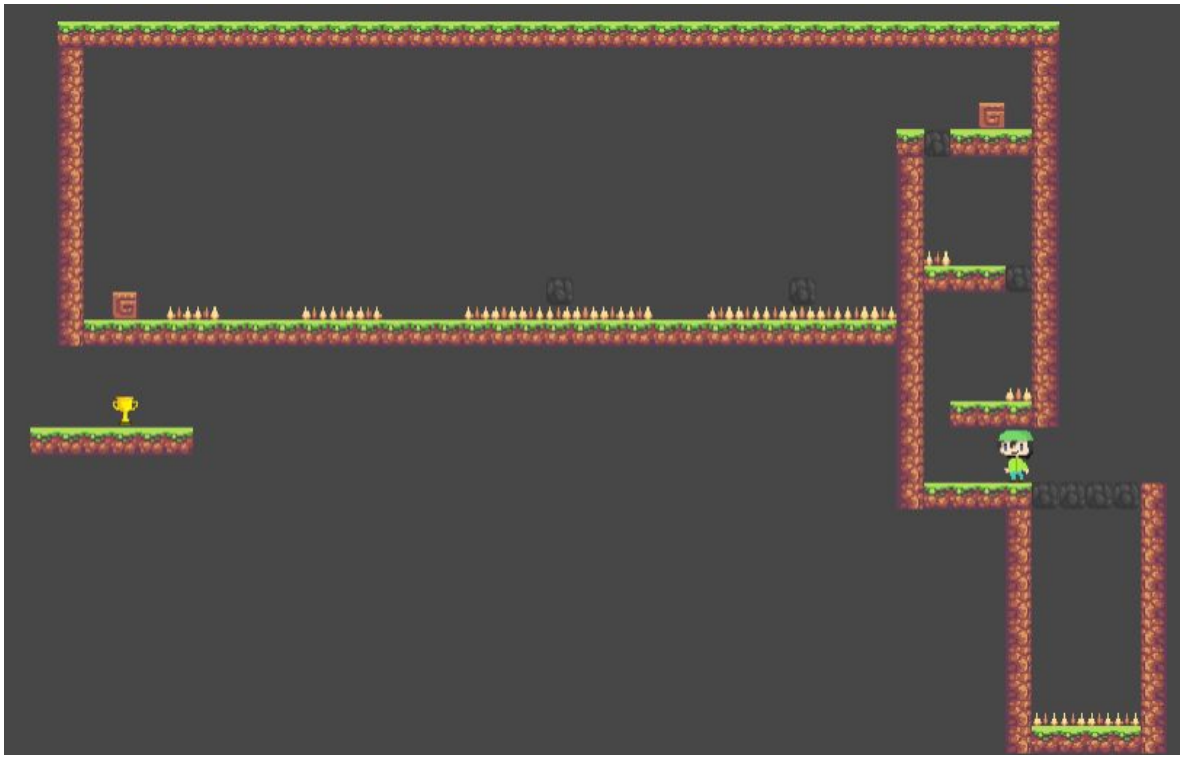

Figura 37. Diseño nivel 8

16.5.9 Nivel 9

Este nivel está más centrado en el combate, y además introduce un uso alternativo para los bloques azules y rojos. La llave que permite abrir la puerta se obtiene cuando se derrota al enemigo que está más elevado.

Al principio el jugador solo se enfrentará a dos enemigos cuerpo a cuerpo un arquero mientras un dispensador de flechas va disparando. A medida que pase el tiempo, los bloques azules y rojos irán alternando y de esa forma actúan como temporizador para que caigan dos enemigos más eventualmente.

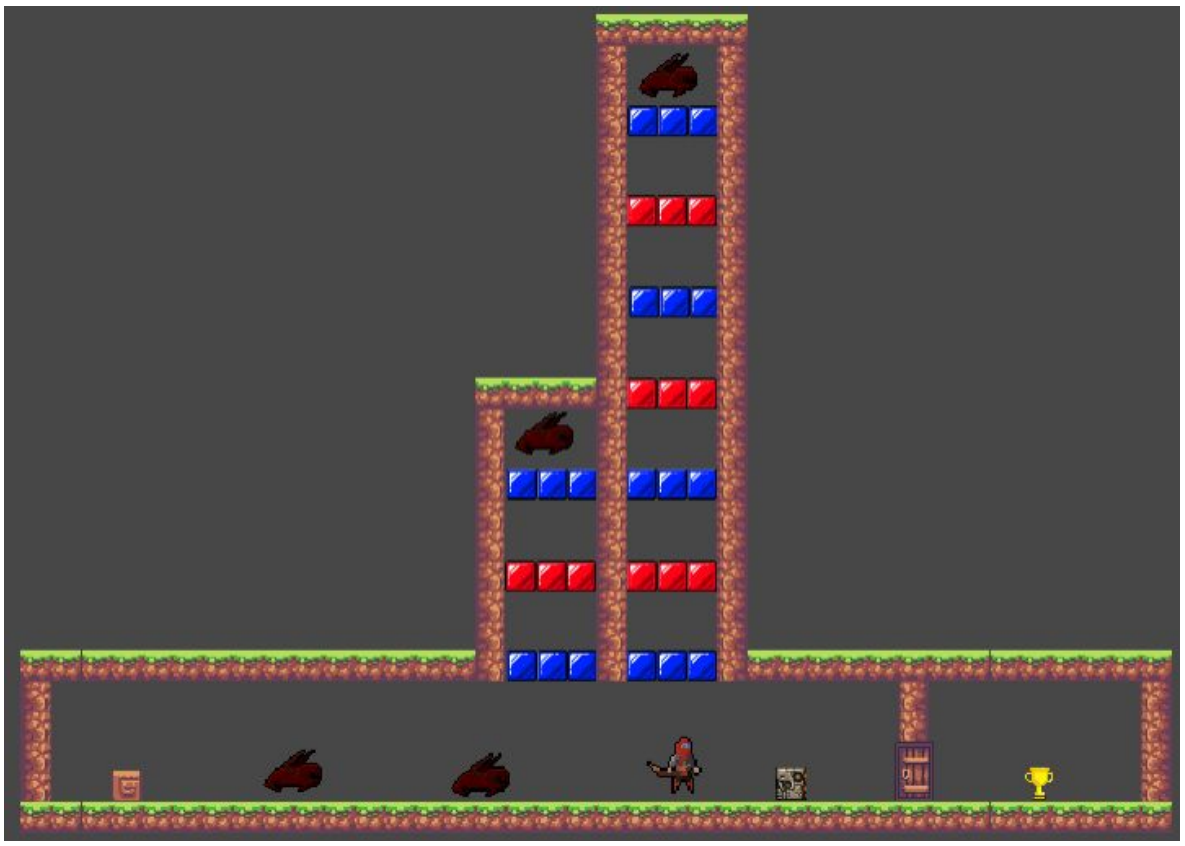

Figura 38. Diseño nivel 9

16.5.10 Nivel 10

En este nivel se ha querido explorar más la verticalidad que en anteriores niveles.

En la primera parte los 3 dispensadores de flechas disparan a tiempos distintos, haciendo que sea posible para el jugador ascender por las paredes intentando evitar flechas recibiendo el menor daño posible.

Una vez en la parte central hay un checkpoint que nos guarda el progreso, ahí los dispensadores disparan de dos en dos, de tal forma que los dos de la izquierda disparan a la vez y pasados cierto tiempo lo hacen los de la derecha. Ambos pares de dispensadores tienen la misma frecuencia de disparo. Aquí el jugador tendrá que calcular bien cuando saltar y hacer todos los saltos necesarios de forma casi ininterrumpida debido a que los bloques centrales son del tipo de los que se deshacen cuando se pisan, dando poco espacio de maniobra al jugador.

Figura 39. Diseño nivel 10

16.5.11 Nivel 11

Este nivel es uno que se centra en la narrativa más que en la jugabilidad, al principio, el nivel parece realmente simple, únicamente hay que saltar un par de pinchos y esquivar flechas para llegar a la meta, pero la verdad es que el suelo que hay frente a la meta está compuesto por bloques fantasma que el jugador atraviesa.

Después de esto el jugador únicamente puede seguir descendiendo a plataformas que no alcanza a ver con la cámara hasta finalmente llegar a la meta.

Este nivel representa cómo a veces las cosas parecen fáciles y sencillas pero tu vida o estado de ánimo puede dar un vuelco y de repente sentirte mal, viendo que lo único que puedes hacer es bajar más y más dando saltos de fe esperando que las cosas vayan mejor, algo que sucede a menudo durante la transición.

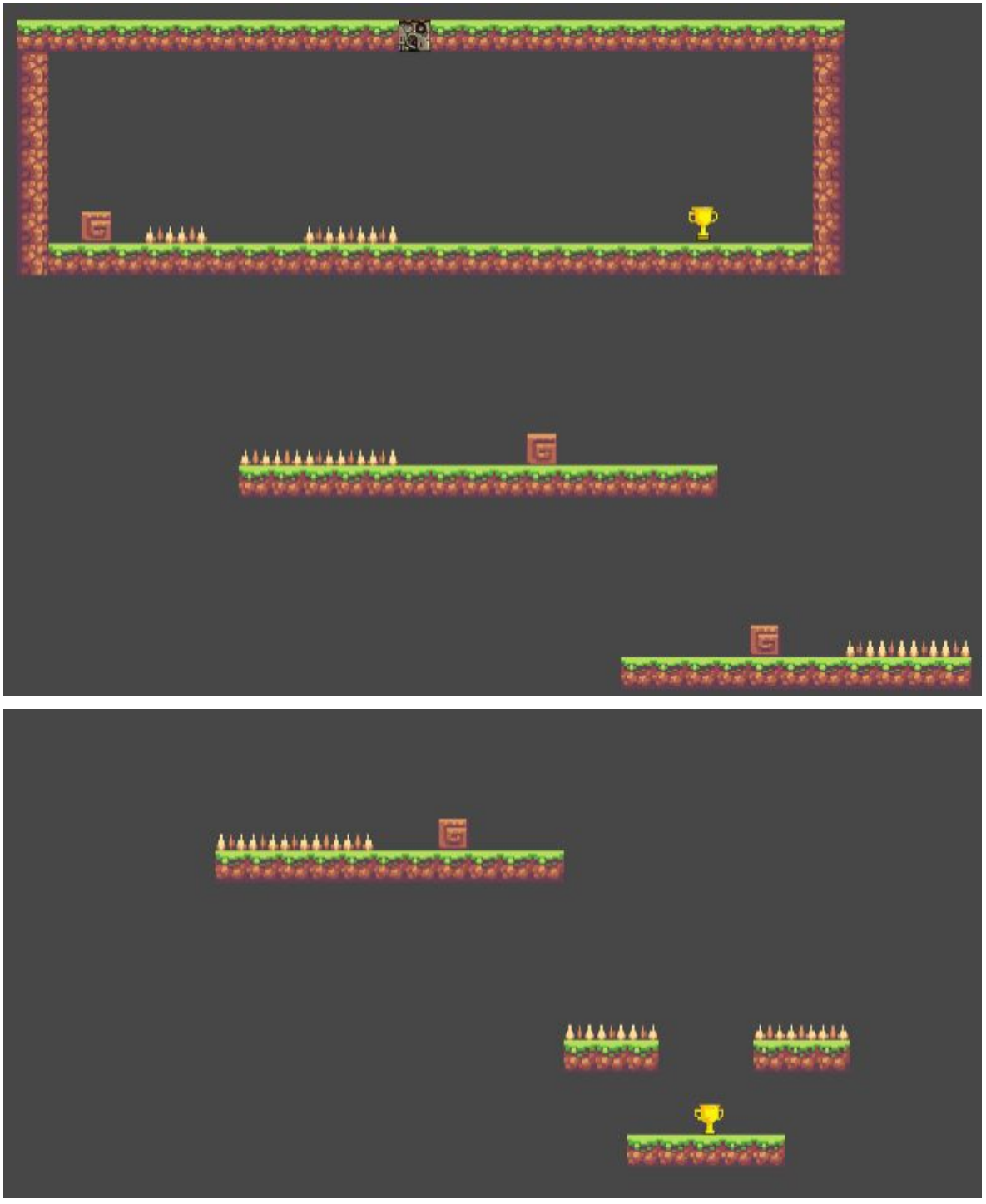

Figura 40. Diseño nivel 11

16.5.12 Nivel 12

Este es el nivel final del juego, en él la protagonista primero tendrá que saltar unas piedras hacia arriba hasta llegar a una plataforma de tiles normales donde le espera un checkpoint y su amigo, que está ahí para darle ánimos y motivarle para que no se rinda.

Una vez sorteados los últimos bloques esquivando las flechas, llegamos ante el jefe final, dónde la protagonista se enfrentará a él de manera verbal.

Esto simboliza lo difícil que es a veces continuar hacia delante, ya que no puedes tomar descansos, esto se representa con el hecho de que los bloques son de los que se deshacen y no puedes pararte. La excepción es cuando alguien cercano está ahí para ayudarte que en ese punto puedes tomar un respiro y continuar hasta afrontar tus miedos e inseguridades.

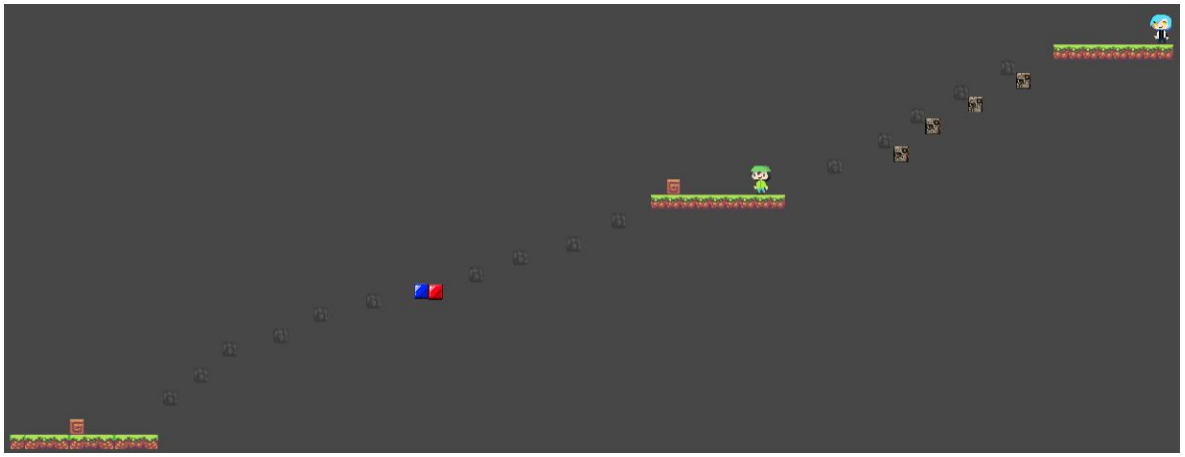

Figura 41. Diseño nivel 12

16.6 Cámara

En este proyecto, se usa una cámara ortográfica que sigue constantemente al jugador. De forma adjunta a esta cámara tenemos una serie de imágenes que forman el background de los niveles. Este background usa la técnica de parallax o paralaje para recrear profundidad simplemente haciendo que las diferentes imágenes que componen el fondo se muevan a diferentes velocidades con respecto al jugador.

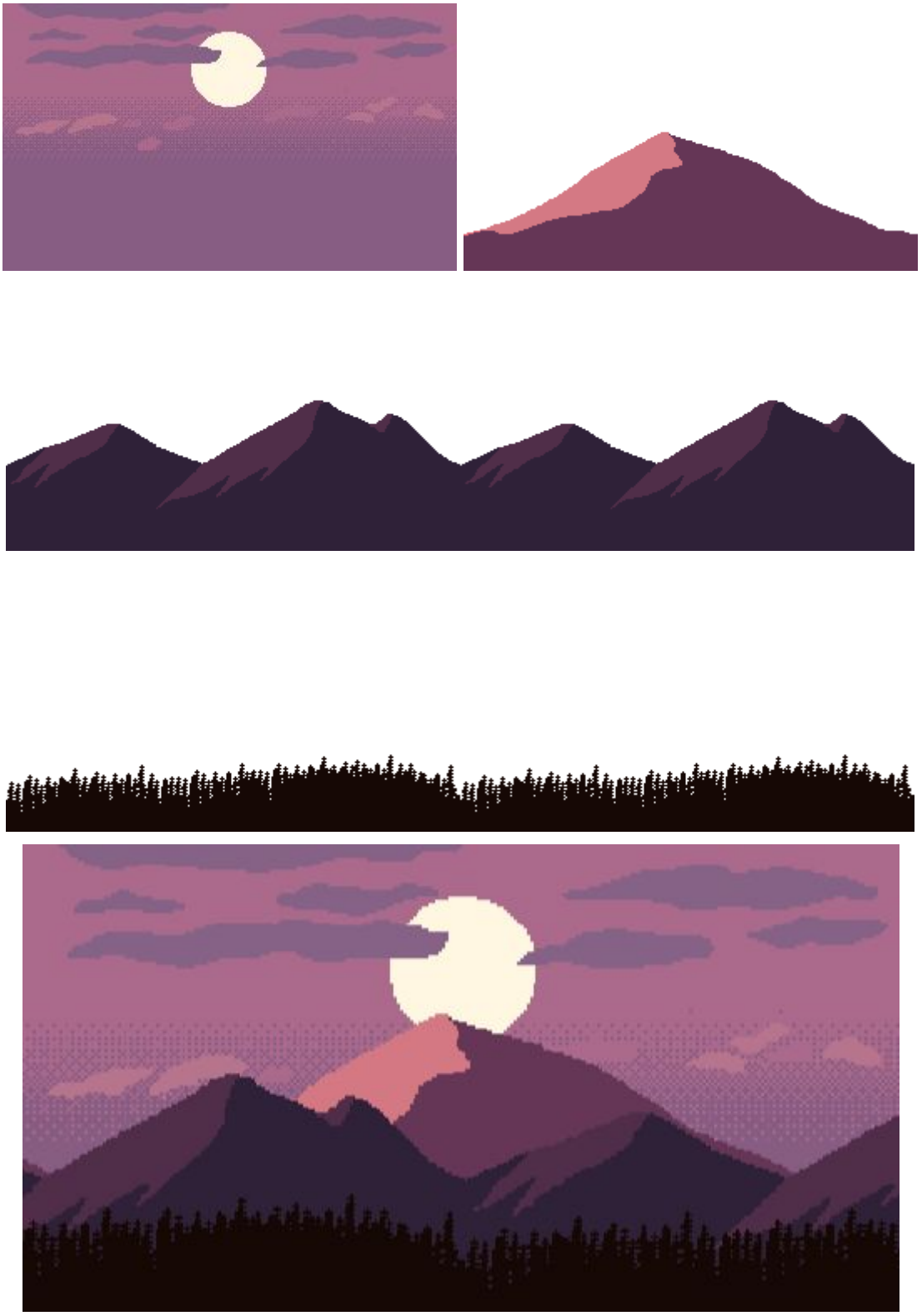

Figura 42. Diferentes elementos que componen el parallax y resultado de todos en conjunto

17. Implementación

17.1 Unity

Antes de empezar a hablar sobre la implementación específica del juego, hay que hablar del motor donde está implementado, en este caso Unity.

Unity está compuesto por diferentes escenas, en cada una de las cuales se encuentran diferentes gameobjects que la componen. Un gameobject es un elemento al que se le acoplan distintos componentes; por defecto todos tienen un componente obligatorio que recibe el nombre de Transform. La Transform contiene la posición, rotación y escala dentro de la escena, aunque también pueden ser valores relativos al padre dependiendo de la jerarquía como se verá a continuación.

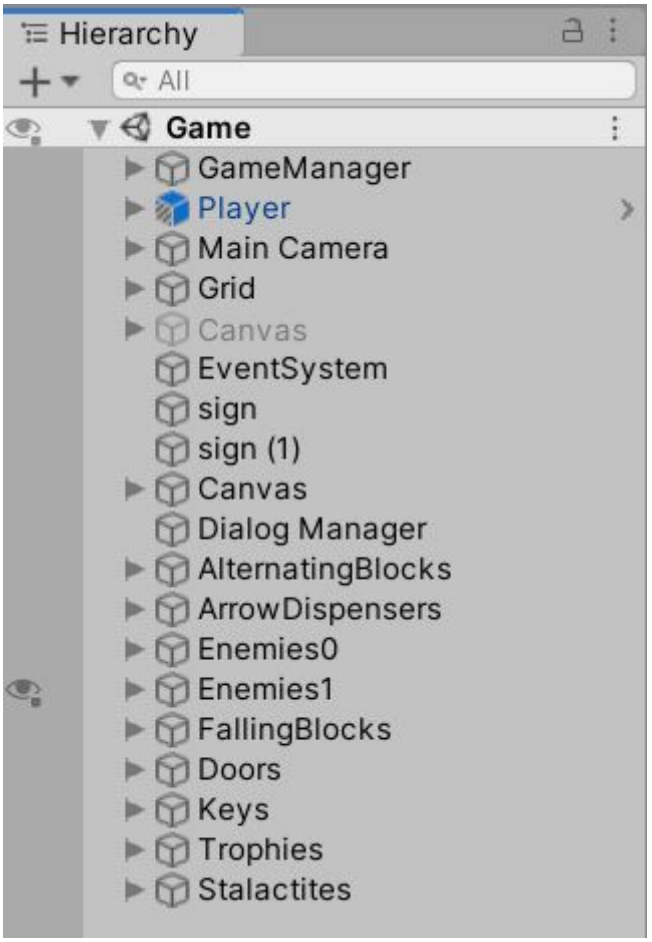

Figura 43. Jerarquía de Unity en la escena el juego.

En las siguientes imágenes podemos ver las Transforms de Player en cierto punto y de GroundCheck. El jugador se encuentra en la posición de la escena indicada en Position y como el GroundCheck es hijo de Player, los valores mostrados indican cuánto difieren del padre.

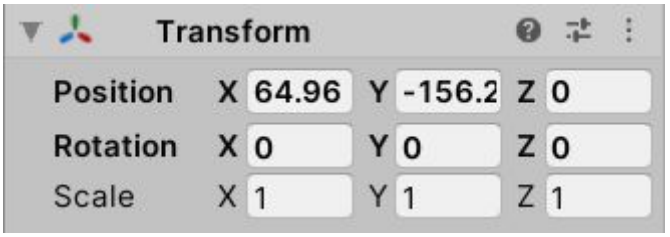

Figura 44. Ejemplo Transform Player.

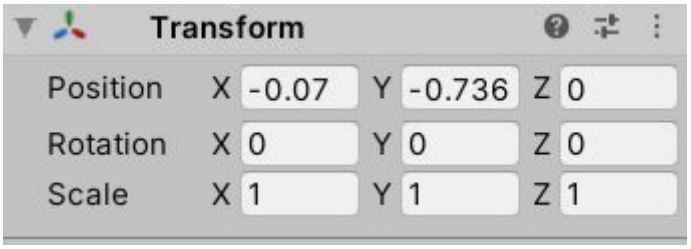

Figura 45. Ejemplo Transform GroundCheck.

A parte de la Transform, a un gameobject se le pueden añadir muchos otros componentes y cada uno sirve un propósito distinto, por ejemplo se puede añadir un rigidBody y un collider para que este gameobject pueda tener físicas y colisionar e interactuar con su entorno. Otros componentes comunes son el sprite renderer, para que el objeto tenga un sprite que lo represente y no solo sea un punto en el espacio, y el animator, que dota de animación y vida al objeto gracias al uso de diferentes sprites.

Pero sin duda el componente más importante que se le puede añadir a un gameobject es un script. Los scripts permiten programar el comportamiento deseado de los diferentes componentes que componen el gameobject.

Para poder trabajar con estos elementos, Unity nos ofrece el inspector que nos muestra los distintos componentes del gameobject seleccionado: en la figura 45 se muestra lo que nos enseña el inspector al seleccionar Player.

Figura 46. Inspector de Unity al seleccionar Player.

Los diferentes componentes en este caso están minimizados, pero pulsando en la flecha a la izquierda del nombre se despliegan los distintos valores y variables que los componen y los podemos modificar a voluntad para obtener el resultado deseado.

Unity también nos ofrece una forma muy conveniente de organizar todos los assets que componen el proyecto en una pestaña llamada project. Aquí podemos crear carpetas para tener todo bien estructurado y separar los assets en diferentes directorios, así cuando necesites buscar sprites por ejemplo simplemente has de ir a la carpeta sprites.

Unity además también permite un tipo de objetos llamados prefabs, estos se crean arrastrando un gameobject de la escena hacia un directorio. Esto crea una copia del gameobject en ese punto y lo almacena, pudiendo ahora instanciar estos prefabs. Además cada instancia de prefab se puede modificar por separado para que difiera del resto o se puede modificar el prefab principal y de esta forma cambiar el elemento modificado en todas las copias.

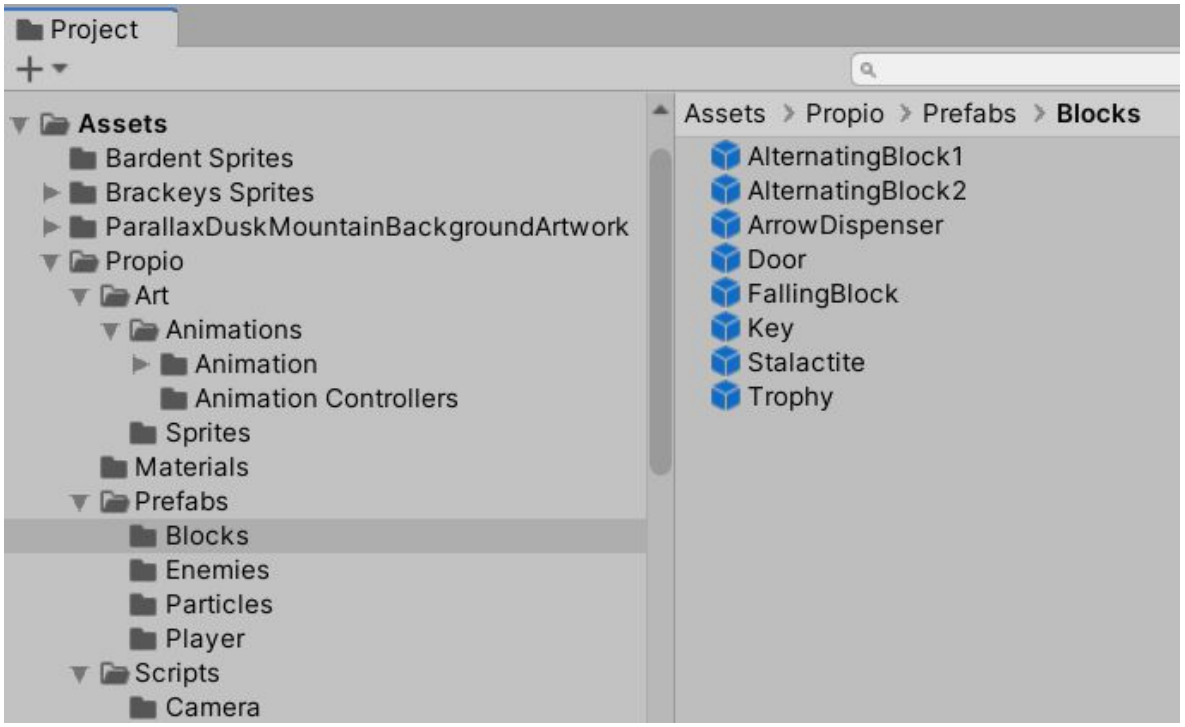

Figura 47. Diferentes directorios y assets que componen el proyecto de Unity.

Si entramos a modificar el prefab general, en la jerarquía de escena nos saldrá únicamente este prefab y sus hijos en caso de que tuviera.

Figura 48. Jerarquía de un prefab

En Unity también disponemos de una consola donde nos salen los warnings y errores y en la mayoría de casos también indica qué script y en qué línea nos salta el problema. También sirve como cualquier consola para debugar gracias a la función Debug.Log que nos permite escribir por consola.

Finalmente contamos con la pantalla principal donde podemos ver la escena o el juego en ejecución entre otras funcionalidades.

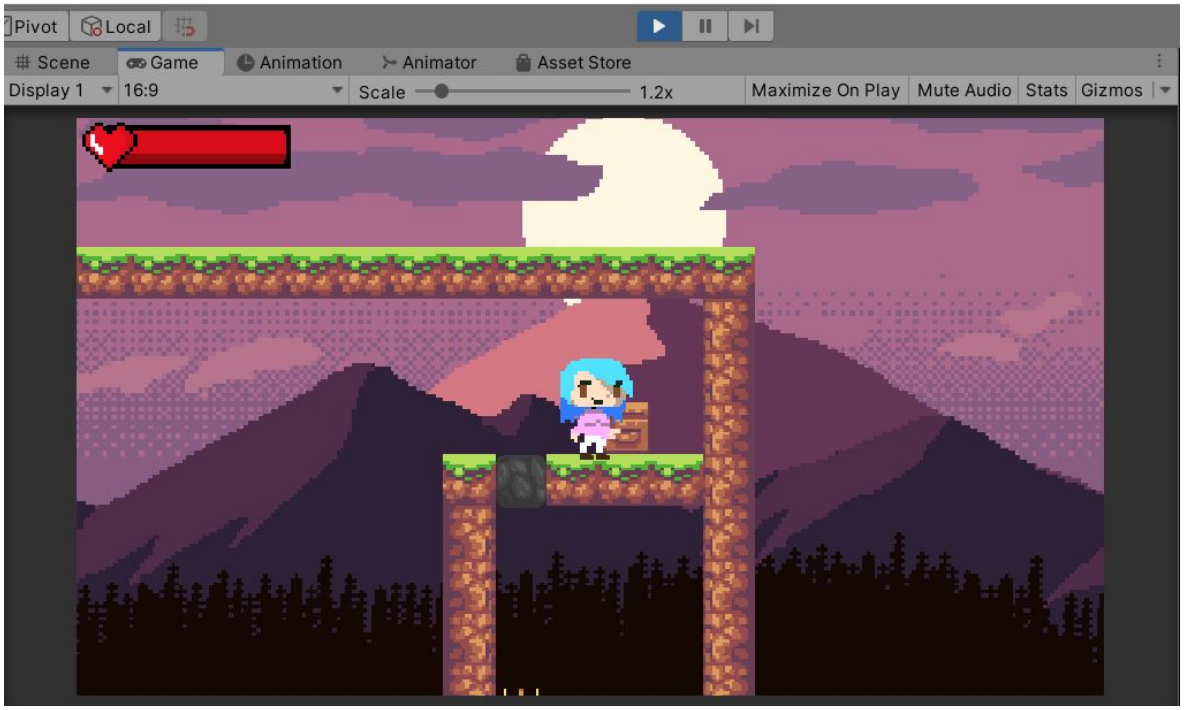

Figura 49. Pantalla principal de Unity.

17.2 Jugador

Para lograr el desarrollo de un player que se sienta bien jugar con él y que tenga una calidad más profesional se han seguido distintas guías de Youtube [16] [17] además de diversos foros donde se solventan dudas [18].

El jugador está representado por un prefab ‘player’ que contiene tres distintos game objects, ‘CharacterAnimation’, ‘Attack1Position’, ‘GroundCheck’. El primero de ellos contiene el sprite renderer y el animator del personaje que se encarga de realizar las distintas transiciones entre animaciones. Attack1Position es un game object que simplemente tiene una transform que indica dónde está el centro del radio de colisión de ataque del jugador, y finalmente ground check es otra transform situada a los pies del personaje que indica el centro del radio que detecta el contacto con el suelo.

El game object Player contiene un rigidbody 2D que se encarga de las físicas del personaje, un capsule colider 2D que se ajusta al sprite del personaje y detecta las colisiones de este. Finalmente hay 3 distintos scripts que son los responsables de toda la parte jugable del jugador, estos son Player, Player Stats y Player Combat Controller.

Player Stats contiene la vida máxima y la actual del jugador, además de las partículas que spawnea al morir. Es también el encargado de actualizar la barra de vida de la UI cuando el jugador recibe daño y de hacer que el jugador muera si su vida actual es inferior o igual a 0.

```
private void Die()
{
    Instantiate(chunkParticles, transform.position, Quaternion.Euler(0.0f, 0.0f, Random.Range(0.0f, 360.0f)));
    GM.Respawn();
    Destroy(gameObject);
}
```

Figura 50. Función Die del jugador.

En la función Die podemos ver como antes de destruir el gameObject, se instancian las particulas que suelta el personaje al morir y se llama a la función Respawn del Game Manager que veremos en un apartado posterior.

Player Combat Controller es el encargado del combate y de sus distintos parámetros, una vez hemos detectado el input necesario para atacar, el personaje hará la animación de ataque si es posible.

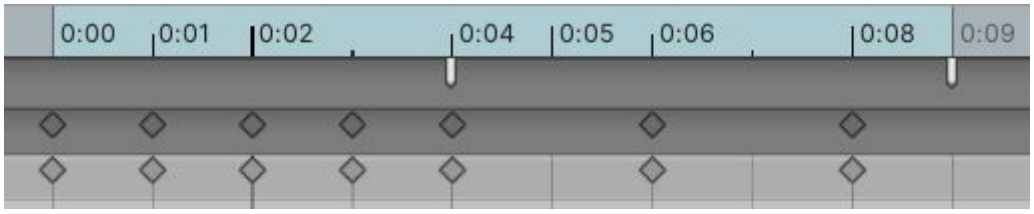

Figura 51. Animación de ataque del jugador en la ventana animation de Unity.

Durante esta animación se llaman a dos eventos, el primero invoca CheckAttackHitBox que lo que hace es detectar los objetos que hay en un círculo de radio definido por attack1Radius que tiene como centro attack1HitBoxPos.position. Entonces para cada objeto encontrado, se invoca a la función Damage (en caso de que tengan) gracias a la función SendMessage.

SendMessage es un método de GameObject que invoca la función que se le pasa como primer parámetro a todos sus scripts que hereden de MonoBehaviour, gracias a esto no hace falta tener acceso al script específico que contenga esta función, solo se necesita el gameObject que contenga el script.

El problema que tiene usar SendMessage es que únicamente se puede pasar un parámetro a la función que se llame, así que como en nuestro caso queremos pasar tres, se creó un struct AttackDetails que contiene toda la información necesaria para llamar a damage, y este struct es lo que se le pasa como parámetro.

```
public void CheckAttackHitBox()
{
    Collider2D[] detectedObjects =
        Physics2D.OverlapCircleAll(attack1HitBoxPos.position, attack1Radius, whatIsDamageable);

    attackDetails.damageAmount = attack1Damage;
    attackDetails.position = transform.position;
    attackDetails.stunDamageAmount = stunDamageAmount;

    foreach (Collider2D col in detectedObjects)
    {
        col.transform.parent.SendMessage("Damage",attackDetails);
    }
}
```

Figura 52. Función CheckAttackHitBox del jugador.

```
public struct AttackDetails
{
    public Vector2 position;
    public float damageAmount;
    public float stunDamageAmount;
}
```

Figura 53. Struct utilizado para la función Damage.

Finalmente, el segundo evento que se llama en la animación es una llamada a FinishAttack, que como su nombre indica finaliza el ataque.

```
public void FinishAttack()
{
    isAttacking = false;
    gotInput = false;
    anim.SetBool("isAttacking", isAttacking);
}
```

Figura 54. Función FinishAttack.

Otra funcionalidad que se encuentra en el script de Player Combat Controller es que al recibir daño el personaje cambie de color a un tono rojizo y se ralentice el tiempo por unos instantes para así dar más impacto al hecho de ser golpeado.

```
private void FlashInRed()
{
    Color c = new Color(1f, 0.5f, 0.5f, 1f);
    spriteRenderer.color = c;
    Time.timeScale = 0.75f;
    Invoke("ResetColor", flashTime);

}
```

Figura 55. Función FlashInRed.

En la anterior función podemos ver que se utiliza Invoke para resetear el color y hacer que el personaje vuelva a la normalidad, se ha hecho de esta manera porque Invoke permite un segundo parámetro, en este caso flashTime, y este parámetro es un float que determina en cuantos segundos se llamará a la función especificada por el primer parámetro. De esta forma pasados flashTime segundos se llama a la función ResetColor.

```
void ResetColor()
{
    spriteRenderer.color = originalColor;
    Time.timeScale = 1f;
}
```

Figura 56. Función ResetColor.

El script más importante en Player es uno con el que comparte nombre. En el script Player se ejecuta todo lo relacionado con el movimiento y el salto del jugador además de las físicas.

El salto del jugador no es perfectamente parabólico, es decir, hay más frames de subida que de bajada. Esto se ha hecho así para emular el comportamiento de juegos de plataformas populares como Super Mario World, donde el salto sigue este comportamiento.

Figura 57. Imagen de Super Mario World donde se muestran los frames en un salto.

El salto también es dependiente de cuanto tiempo mantengas pulsado la barra espaciadora, a más tiempo presionado mayor será el salto. Para poder realizar estos comportamientos, en el fixed update se llama a la función ModifyPhysics la cual contiene, entre otras cosas, el siguiente fragmento de código.

```
rigidBody.gravityScale = gravity;
rigidBody.drag = linearDrag * 0.15f;
if (rigidBody.velocity.y < 0)
{
    rigidBody.gravityScale = gravity * fallMultiplier;
}
else if (rigidBody.velocity.y > 0 && !Input.GetButton("Jump"))
{
    rigidBody.gravityScale = gravity * (fallMultiplier / 1.5f);
}
```

Figura 58. Fragmento de la función ModifyPhysics.

Este snippet nos muestra cómo se modifican las físicas cuando el jugador se encuentra en el aire, de base la gravedad del jugador posee el valor 0 igual que pasa con el linear drag, para que el movimiento en tierra se sienta fluido. Pero a la que el jugador salta se le aplica

la gravedad y el drag. Una vez la velocidad en el eje Y es menor que cero, la gravedad que se le aplica al jugador se incrementa, y si se deja de pulsar la barra espaciadora mientras aún el personaje está subiendo, la gravedad aumenta pero no tanto. El resultado de esto es un salto que resulta satisfactorio y reminiscente de otros videojuegos.

A parte de las diferentes animaciones de las que dispone el personaje principal, en el código se ha programado una corrutina que alarga al personaje un poco durante unas décimas de segundo al saltar y que lo chafa cuando aterriza, haciendo que el acabado del videojuego sea más profesional de manera simple.

```
IEnumerator JumpSqueeze(float xSqueeze, float ySqueeze, float seconds)
{
    Vector3 originalSize = Vector3.one;
    Vector3 newSize = new Vector3(xSqueeze, ySqueeze, originalSize.z);
    float t = 0f;
    while (t <= 1.0)
    {
        t += Time.deltaTime / seconds;
        transform.localScale = Vector3.Lerp(originalSize, newSize, t);
        yield return null;
    }
    t = 0f;
    while (t <= 1.0)
    {
        t += Time.deltaTime / seconds;
        transform.localScale = Vector3.Lerp(newSize, originalSize, t);
        yield return null;
    }

}
```

Figura 59. Corrutina JumpSqueeze.

El movimiento del personaje se controla en la función MoveCharacter. Primero comprobamos que el personaje no se esté deslizando por una pared, ya que en esos momentos no se puede mover, únicamente puede saltar.

Si el personaje se encuentra tocando el suelo, al rigidbody del personaje se le aplica esta velocidad simplemente en la dirección que toque, esto viene marcado por la variable que se le pasa llamada horizontal.

Si el personaje no está en el suelo, es decir está en el aire, en vez de cambiar directamente la velocidad que tiene el rigidbody, a este se le añade una fuerza que hace que se desplace. El resultado de esto es que en el aire el jugador se mueve menos que en tierra como ocurre en Super Mario World.

```
private void MoveCharacter(float horizontal)
{
    if (!isWallSliding)
    {
        if ((onGround) && !knockback)
        {
            newVelocity.Set(speed * horizontal, 0.0f);
            rigidBody.velocity = newVelocity;
        }

        else if (!onGround && !knockback)
        {
            Vector2 forceToAdd = new Vector2(movementForceInAir * horizontal, 0);
            rigidBody.AddForce(forceToAdd);
        }

        if ((horizontal > 0 && facingLeft) || (horizontal < 0 && !facingLeft))
        {
            Flip();
        }
    }
    animator.SetFloat("Horizontal", Mathf.Abs(rigidBody.velocity.x));
    animator.SetFloat("Vertical", rigidBody.velocity.y);



    if (Mathf.Abs(rigidBody.velocity.x) > maxSpeed)
    {
        rigidBody.velocity = new Vector2 (Mathf.Sign(rigidBody.velocity.x) * maxSpeed, rigidBody.velocity.y);
    }
}
```

Figura 60. Función MoveCharacter.

En este script también se definen distintos comportamientos que afectan al personaje al entrar en contacto con distintos elementos de los diferentes escenarios.

- Si el jugador entra en contacto con pinchos, se llama a la función Damage, pasándole una cantidad muy elevada de daño que asegura que el jugador muera.
- Si entra en contacto con un checkpoint, le envía la posición actual al Game Manager para que la guarde como punto de respawn en caso de morir, y el propio Player también la guarda por si sale Out Of Bounds.
- Si colisiona con un DialogTrigger, activa el diálogo pertinente y elimina a estos bloques que desencadenan este diálogo para que al acabar de leerlo, en caso de volver a tocarlos no salga un diálogo de nuevo.
- Si el jugador toca una llave, el booleano que lo indica se pone a cierto y el del Game Manager también para que en caso de morir, el jugador cuando haga respawn siga teniendo la llave y además destruye la llave.
- Si el jugador se sale de los límites del mapa, este se verá transportado al último checkpoint alcanzado.
- Y finalmente si toca un trofeo significa que ha completado el nivel actual y el jugador se verá transportado al siguiente al cabo de un segundo.

```
public void OnTriggerEnter2D(Collider2D collision)
{
    AttackDetails a = new AttackDetails();
    a.damageAmount = 999f;

    if(collision.tag == "Spikes")
    {
        pcc.SendMessage("Damage", a);
    }
    else if (collision.tag == "Checkpoint")
    {
        GM.respawnPoint = transform.position;
        respawnPoint = transform.position;
    }
    else if (collision.tag == "DialogTrigger")
    {
        dialog.NextSentence();
        Destroy(collision.gameObject);
    }
    else if (collision.tag == "Key")
    {
        hasKey = true;
        GM.hasKey = true;
        Destroy(collision.gameObject);
    }
    else if (collision.tag == "OOB")
    {
        GoToCheckpoint();

    }
    else if (collision.tag == "Trophy")
    {
        Invoke("ChangeLevel", 1f);
    }
}
```

Figura 61. Función OnTriggerEnter2D.

17.3 Enemigos

El código de los enemigos se ha desarrollado gracias a diferentes tutoriales de youtube [19] y está distribuido en diferentes scripts que forman parte de una finite-state machine o FSM para abreviar. Esta FSM está compuesta por un script FiniteStateMachine que cuenta únicamente con dos funciones Initialize y ChangeState, la función Initialize llama a la función Enter del estado en el que se inicializa y la función ChangeState llama a Exit del estado actual y Enter del nuevo estado.

```
public void Initialize(State startingState)
{
    currentState = startingState;
    currentState.Enter();
}
```

Figura 62. Función Initialize del script FiniteStateMachine.

```
public void ChangeState(State newState)
{
    currentState.Exit();
    currentState = newState;
    currentState.Enter();
}
```

Figura 63. Función ChangeState del script FiniteStateMachine.

Estos estados mencionados son instancias de un hijo de State, el script State contiene el comportamiento que todos sus hijos comparten y eso es que al entrar en un estado se guarde el instante en el que ha sucedido, se ponga a true el booleano del animador correspondiente al estado y al salir este booleano se ponga en falso; además de hacer los checks correspondientes en el FixedUpdate.

```
public virtual void Enter()
{
    startTime = Time.time;
    entity.anim.SetBool(animBoolName, true);
    Checks();
}
```

Figura 64. Función Enter de State.

```
public virtual void Exit()
{
    entity.anim.SetBool(animBoolName, false);
}
```

Figura 65. Función Exit de State.

```
public virtual void PhysicsUpdate()
{
    Checks();
}
```

Figura 66. Función PhysicsUpdate de State.

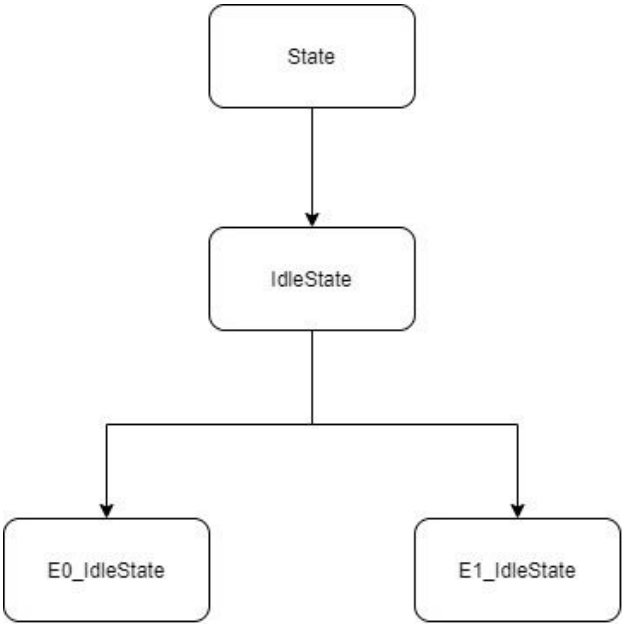

Figura 67. Jerarquía del estado IdleState.

Más abajo en la jerarquía nos encontramos diferentes estados genéricos que heredan de State, como por ejemplo IdleState entre otros. En cada estado específico se gestiona todo aquello relacionado con ese estado, como por ejemplo en el estado idle, se fija la velocidad de la entidad a 0, se aleatoriza dentro de unos valores el tiempo que va a estar en idle y cuando este tiempo ha transcurrido se indica en un booleano que ya se ha acabado.

Finalmente, en lo más bajo de la jerarquía están los estados específicos a cada enemigo, estos estados indican las transiciones entre los diferentes estados. Siguiendo el ejemplo anterior, E0_IdleState es el estado Idle del enemigo 0, el que es cuerpo a cuerpo, y en este script se indica que si el jugador está a la distancia suficiente, el enemigo ha de cambiar al estado playerDetected y si se ha acabado el tiempo de idle simplemente cambia a seguir moviéndose.

```
public override void LogicUpdate()
{
    base.LogicUpdate();
    if (isPlayerInMinAggroRange)
    {
        stateMachine.ChangeState(enemy.playerDetectedState);
    }

    else if (isIdleTimeOver)
    {
        stateMachine.ChangeState(enemy.moveState);
    }
}
```

Figura 68. Función LogicUpdate de E0_IdleState.

Cada uno de los estados específicos tiene asociado una serie de parámetros correspondientes a ese estado, estos parámetros están en unos scripts que heredan de la clase ScriptableObject en vez de MonoBehaviour. De esta forma es muy fácil ampliar el videojuego añadiendo nuevos enemigos, ya que simplemente hay que adjuntarle un scriptable object para cada estado y cambiar los valores de estos para que tenga un comportamiento distinto. Otra ventaja de usar scriptable objects es que los cambios que se realizan sobre estos en tiempo de ejecución se guardan.

Gracias a la FSM, las transiciones entre animaciones quedan muy ordenadas, ya que al entrar a cada diferente estado se establece un bool a true y otro a false, no hay que preocuparse de hacer que cada transición vaya a la que corresponda; únicamente hay que hacer una animación vacía a la que toda animación va cuando su bool es falso y de esta animación vacía hay transiciones a cada otra animación.

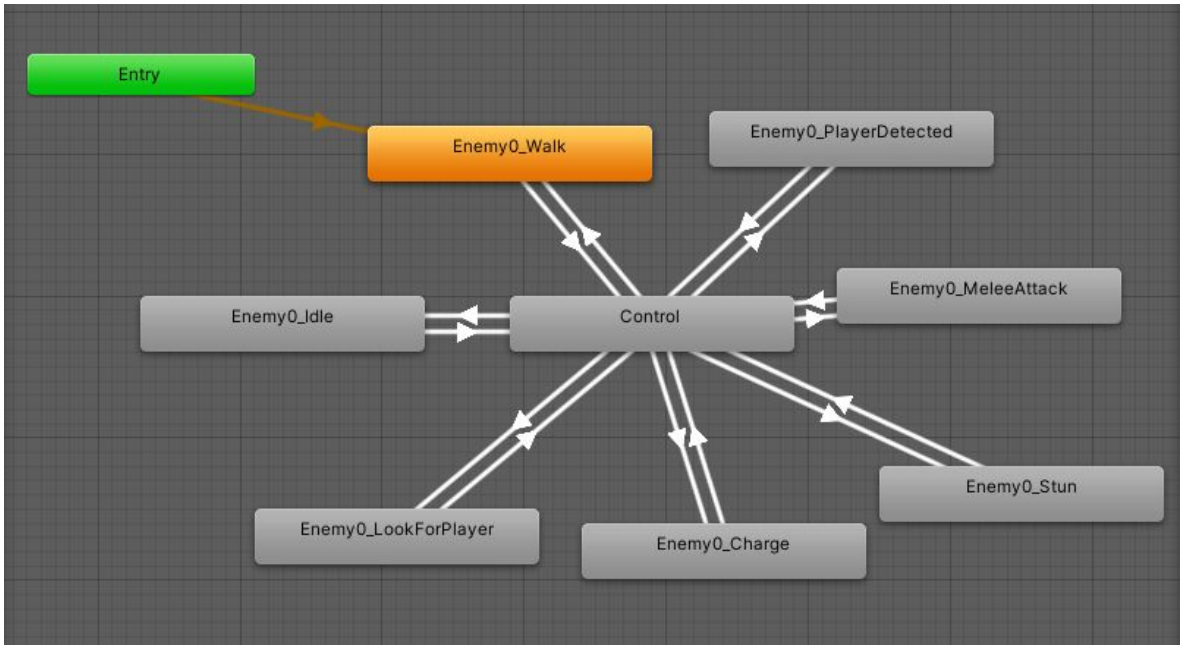

Figura 69. Árbol de transiciones entre animaciones del enemigo cuerpo a cuerpo

Los scripts de los enemigos específicos heredan de Entity. En este script, que sí hereda de MonoBehaviour, se encuentran todas las funciones básicas que todo enemigo ha de tener, detectar su entorno (suelo, borde de la plataforma, pared, jugador), cambiar la velocidad de movimiento y recibir daño entre otras.

Después, en cada enemigo específico se encuentran las llamadas a los constructores de todos los estados que corresponden a ese enemigo además de hacer override a la función Damage para saber a qué estado transicionar, ya que en entity no se tiene la visibilidad necesaria para ello.

Las FSM específicas de los diferentes enemigos son las siguientes:

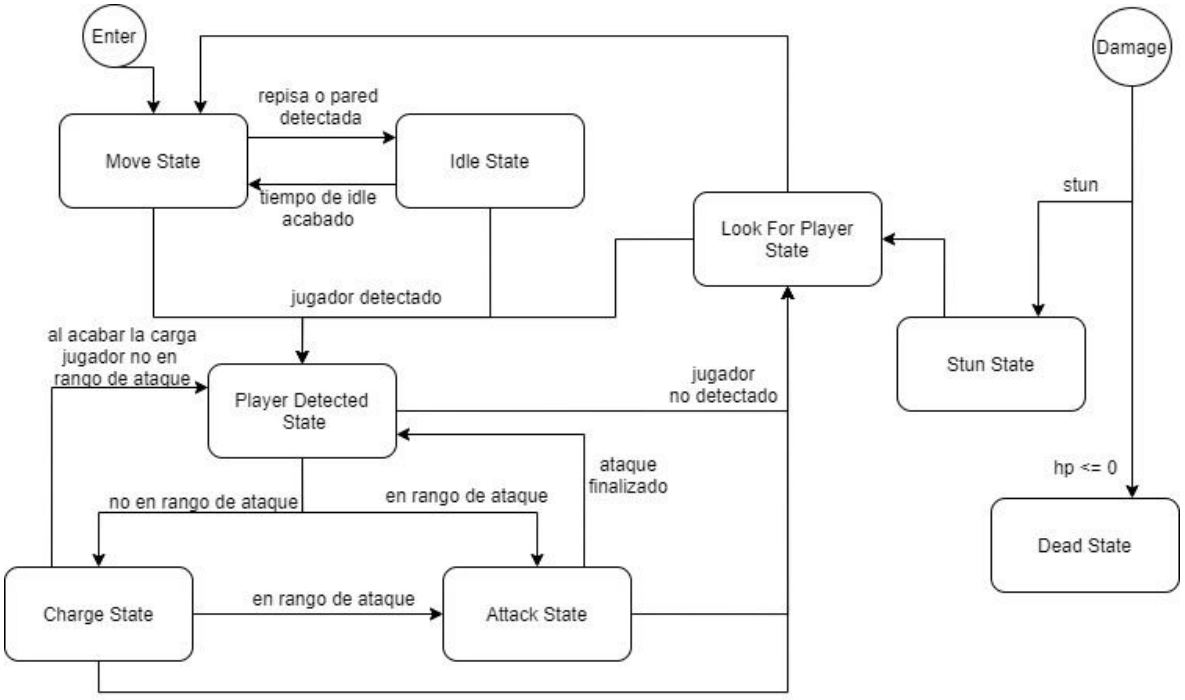

Figura 70. Diagrama de la máquina de estados del enemigo cuerpo a cuerpo.

El primer estado al que un enemigo entra es el de moverse, un enemigo se mueve hasta que detecte una pared o el final de una plataforma, que entonces cambiará al estado de idle, en el que simplemente está vigilando y a la que acabe el tiempo volverá a patrullar. Si en cualquier momento detecta al jugador cambiará al estado de jugador detectado en el que cambia la animación y se queda quieto un momento y decide qué hacer, aquí es donde difiere el comportamiento de los dos diferentes enemigos.

El enemigo cuerpo a cuerpo cargará hacia el jugador en caso de no estar suficientemente cerca como para atacar y si entonces está cerca, se dispondrá a atacar; de la misma forma si ya directamente le detecta cerca atacará sin necesidad de cargar primero. Al acabar el ataque volverá al estado de jugador detectado en caso de que lo detecte.

Si en algún punto el jugador se sale del rango de visión del enemigo una vez ha sido detectado, el jugador irá al estado de buscar jugador en el que girará varias veces con tal de encontrar al jugador, si lo detecta pasará al estado de jugador detectado y si no, simplemente se moverá.

En cualquier punto durante cualquier estado, el enemigo puede ser dañado por el jugador y en ese caso si es golpeado repetidamente en poco tiempo quedará aturdido y si su vida baja por debajo o llega a 0, morirá. Una vez salga del aturdimiento cambiará al estado de buscar jugador.

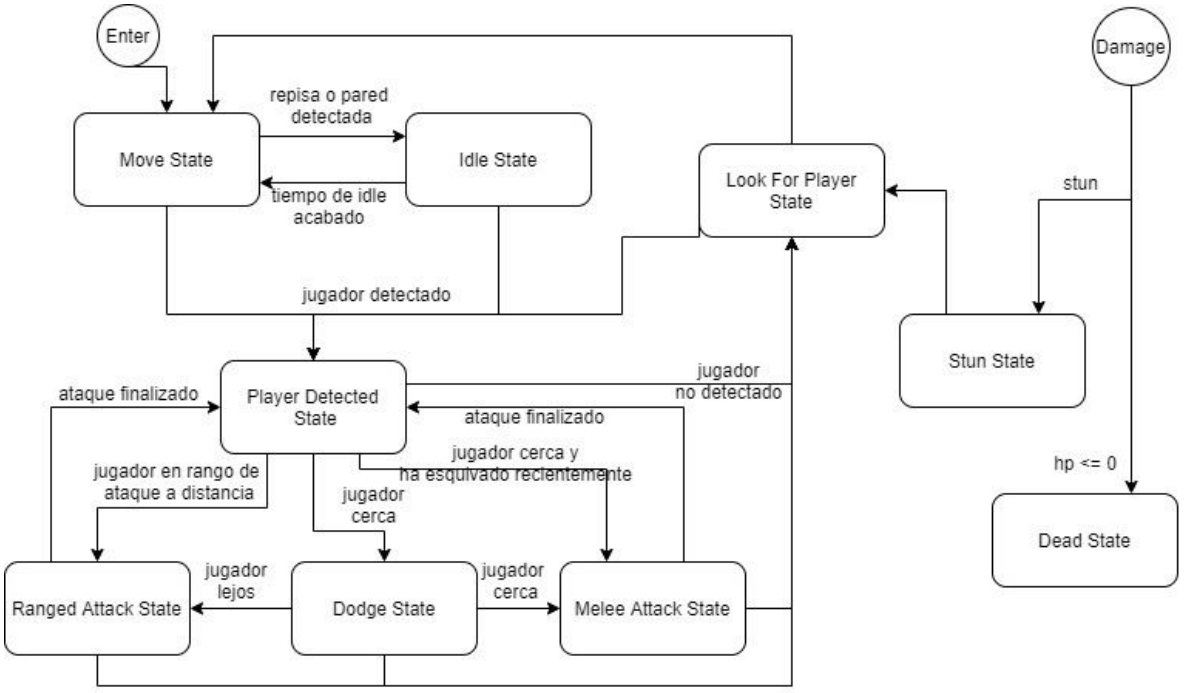

Figura 71. Diagrama de la máquina de estados del arquero.

El arquero comparte gran parte de los estados y transiciones del enemigo a cuerpo a cuerpo, únicamente difiere en qué hace al detectar al jugador.

Cuando el arquero detecta al jugador, si este está lejos procederá a disparar un flecha; si se encuentra cerca hará un salto hacia atrás para así alejarse del jugador y entonces si el jugador está lejos disparará y si se encuentra cerca pegará un puñetazo. Este salto del arquero tiene un tiempo de enfriamiento y mientras no se haya recargado, siempre que un jugador esté cerca pegará un puñetazo.

17.4 Sistema de guardado

El videojuego cuenta con un sistema de guardado que permite al jugador salvar su progreso en cualquier momento y cargarlo cuando quiera. Para ello se usa un formatter que hace que el archivo de guardado no sea fácilmente legible y por ende no sea fácilmente modificable. Para el path donde se guarda el documento se utiliza Application.PersistentDataPath que devuelve un path válido dependiendo del sistema operativo donde se utiliza, por ejemplo en Windows se guarda en C:/Users/Username/AppData/LocalLow/CompanyName/GameName en Android apunta a /storage/emulated/0/Android/data/<packagename>/files, en IOS /var/mobile/Containers/Data/Application/<guid>/Documents.

Para guardar hay que usar un fichero serializable que no herede de MonoBehaviour y con tipos no específicos de Unity, por ejemplo un Vector3 que represente la posición del jugador se tiene que guardar como un array de 3 floats.

```
public PlayerData (Player p)
{
    health = p.ps.currentHealth;
    level = p.level;
    hasKey = p.hasKey;
    pos = new float[3];
    pos[0] = p.transform.position.x;
    pos[1] = p.transform.position.y;
    pos[2] = p.transform.position.z;
}
```

Figura 72. Constructor de PlayerData.

```
public static void SavePlayer (Player p)
{
    BinaryFormatter formatter = new BinaryFormatter();

    string path = Application.persistentDataPath + "/player.txt";
    FileStream stream = new FileStream(path, FileMode.Create);

    PlayerData data = new PlayerData(p);

    formatter.Serialize(stream, data);
    stream.Close();
}
```

Figura 73. Función SavePlayer.

Cargar los datos es muy parecido a guardarlos, se mira que exista el archivo en el path indicado y en caso de que así sea se abre un stream para leer los datos y se deserializan y se le hace un cast a PlayerData.

```
public static PlayerData LoadPlayer()
{
    string path = Application.persistentDataPath + "/player.txt";
    if (File.Exists(path))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream = new FileStream(path, FileMode.Open);

        PlayerData data = formatter.Deserialize(stream) as PlayerData;
        stream.Close();
        return (data);
    } else
    {
        Debug.LogError("save file not found in " + path);
        return null;
    }
}
```

Figura 74. Función LoadPlayer.

17.5 UI

La UI del videojuego consiste en el menú principal, el menú de pausa, los diálogos y la barra de vida del jugador situada en la esquina superior izquierda.

Al iniciar el juego, lo primero que se muestra es una escena que contiene el menú principal del juego.

Figura 75. Menú principal del videojuego.

El menú principal está compuesto por 3 botones: el botón de new game carga la escena que contiene el juego con el personaje en el primer nivel; el botón load game carga la escena que contiene el juego pero pone un bool a true y es el script Player en su función Start que llama a LoadPlayer si este bool es cierto. Finalmente el botón Quit cierra el juego.

El menú de pausa es un elemento de la UI que está formado por una pantalla translúcida negra, que nos indica visualmente que el juego está pausado, y diferentes botones. Estos botones cumplen diferentes funciones cada uno:

- El botón de “Resume” reanuda la ejecución del juego.
- El botón de “Save” sirve para guardar el progreso del jugador y que la próxima vez que abra el juego pueda reanudarlo donde lo dejó.
- El slide “Volume” sirve para ajustar el volumen de todos los elementos de sonido del juego.
- El botón “Menu” nos lleva de vuelta al menú principal.
- El botón “Quit” detiene la ejecución del juego.

Figura 76. Menú de pausa del videojuego.

El código del menú de pausa es relativamente simple. En el update miramos si se ha pulsado escape y en caso de que así sea se pausa o reanuda el juego según la situación.

Para pausar el juego, se fija Time.timeScale a 0 y se activa la UI del menú de pausa. Poner Time.timescale a 0 hace que cualquier operación que dependa del tiempo, como una animación o el movimiento por ejemplo, se detiene.

```
void Pause()
{
    pauseMenuUI.SetActive(true);
    Time.timeScale = 0f;
    isGamePaused = true;
}
```

Figura 77. Función Pause.

Reanudar el juego es hacer lo contrario que pausarlo, poner el timescale a 1 y desactivar la UI del menú de pausa.

```
public void Resume()
{
    pauseMenuUI.SetActive(false);
    Time.timeScale = 1f;
    isGamePaused = false;
}
```

Figura 78. Función Resume.

La función de Save únicamente obtiene la instancia de Player gracias a que este es un singleton y llama a la función SavePlayer.

```
public void Save()
{
    Player.getInstance().SavePlayer();
}
```

Figura 79. Función Save del menú de pausa.

Para controlar el audio el juego, se ha creado un Audio Mixer que se encarga de cambiar el volumen de todos los elementos que generen audio.

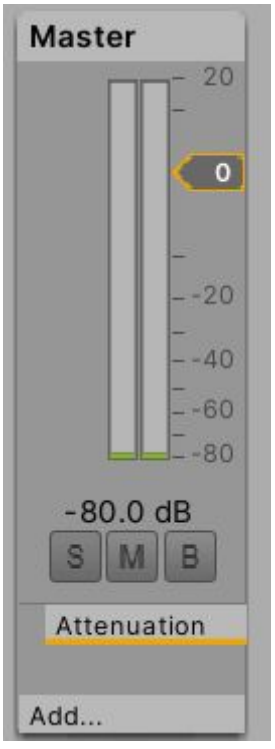

Figura 80. Audio Mixer del juego.

El volumen se mide en decibelios y suele ir de 0 a -80, pero si hacemos un slider que se relacione directamente con estos valores, el slider no será representativo ya los decibelios son una escala logarítmica y para bajar a la mitad el volumen hay que bajar solo 6 unidades y no cuarenta.

Es por eso que el slider va de 0.0001 a 1 y se hace el logaritmo en base 10 de ese valor y se multiplica por 20 para obtener el resultado deseado.

```
public void SetVolume(float volume)
{
    audioMixer.SetFloat("volume", Mathf.Log10(volume) * 20);
}
```

Figura 81. Función SetVolume del menú de pausa.

El botón de Menu simplemente carga la escena 0 que es la correspondiente al menú principal y vuelve a poner la escala de tiempo a 1.

```
public void Menu()
{
    Time.timeScale = 1f;
    SceneManager.LoadScene(0);
}
```

Figura 82. Función Menu del menú de pausa.

Finalmente la función Quit comprueba si está en el editor de Unity o si está en una aplicación y ejecuta la función necesaria para terminar la ejecución.

```
public void Quit()
{
    #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
    #else
        Application.Quit();
    #endif
}
```

Figura 83. Función Quit.

Los diálogos están hechos de tal manera de que el jugador al entrar en contacto con un bloque invisible llamado DialogTrigger se llame a la función NextSentence y el juego se pause y salga el texto correspondiente en pantalla.

Figura 84: Ejemplo de un diálogo, aparece en el nivel 3.

La forma en la que están almacenados los diálogos son dos arrays de enteros que representan los diferentes diálogos y las frases que los componen. Mientras el índice de frase sea menor que la cantidad de frases que hay en ese diálogo, se vaciará lo que hubiera en pantalla y se llama a la corrutina Type.

```
public void NextSentence()
{
    textDisplay.enabled = true;
    continueButton.SetActive(false);
    if (indexSentence < dialogues[indexDialogue] - 1)
    {
        indexSentence++;
        textDisplay.text = "";
        StartCoroutine(Type());
        Time.timeScale = 0f;
    }
    else
    {
        textDisplay.text = "";
        indexDialogue++;
        indexSentence = -1;
        continueButton.SetActive(false);
        textDisplay.enabled = false;
        Time.timeScale = 1f;
    }
}
```

Figura 85. Función NextSentence.

La corrutina Type escribe en pantalla la frase que corresponde del diálogo y lo hace letra a letra, de esta forma el texto aparece progresivamente en pantalla y no de golpe. Para poder lograr eso, se utiliza la función WaitForSecondRealtime que como su nombre indica espera el tiempo indicado para poder seguir ejecutando el código. Se utiliza el tiempo real porque si se usara el tiempo del juego nunca se acabaría de escribir ya que el juego está pausado durante los diálogos.

```
IEnumerator Type()
{
    int count = 0;
    for (int i=0; i < indexDialogue; i++)
    {
        count += dialogues[i];
    }
    foreach (char letter in sentences[count + indexSentence].ToCharArray())
    {
        textDisplay.text += letter;
        yield return new WaitForSecondsRealtime(typingSpeed);
    }
}
```

Figura 86. Corrutina Type.

En el update se mira si se ha acabado de escribir la frase que toca, y en caso de ser así, se activa el botón de continuar. Este elemento se encuentra en la esquina inferior derecha y al hacer hover sobre él cambia de color y si se clica llama a la función NextSentence continuando así con el diálogo.

```
if (textDisplay.text == sentences[count+indexSentence])
{
    continueButton.SetActive(true);
}
```

Figura 87. Condicional que mira si hay que activar el botón de continuar.

Una vez se han acabado todas las frases de un diálogo, el juego se reanuda hasta tocar otro bloque de DialogTrigger.

La barra de vida es un elemento de la interfaz de usuario que siempre se encuentra visible a diferencia de los dos anteriores, esta barra se actualiza cuando el jugador recibe daño y lo refleja dependiendo de cuán llena se encuentre.

Figura 88. Diferentes estados de la barra de vida.

17.6 Game manager

El Game manager es el encargado de hacer respawn tanto de los enemigos como del jugador cuando muere. Esta figura del Game manager es necesaria porque al morir, el personaje y los enemigos o bien son destruidos o se quedan en estado inactivo, y así los componentes que contienen ya no pueden hacer nada, por eso el Game manager guarda referencias de todos ellos y los activa o instancia cuando es necesario.

Anteriormente vimos que el jugador al morir, antes de ser destruído llama a Respawn en el Game manager, lo único que hace esta función es guardar el momento en el que el jugador la ha llamado y poner un booleano a verdadero.

```
public void Respawn()
{
    respawnTimeStart = Time.time;
    respawn = true;
}
```

Figura 89. Función Respawn del Game manager.

Como se puede observar en el siguiente código, una vez haya pasado el tiempo suficiente, se hará respawn de los enemigos que hubieran muerto, se setea el booleano a falso para no entrar en cada update aquí y después se instancia al jugador. Como el jugador fue destruído, la cámara ya no sabría a qué seguir a lo largo de los niveles y es aquí donde se indica al script de la cámara (cf) que ha de seguir al nuevo jugador instanciado.

```
private void CheckRespawn()
{
    if (Time.time >= respawnTimeStart + respawnTime && respawn)
    {
        RespawnEnemies();
        respawn = false;
        var temp = Instantiate(player, respawnPoint, Quaternion.identity);
        cf.followObject = temp;
        cf.rb = cf.followObject.GetComponent<Rigidbody2D>();

    }
}
```

Figura 90. Función CheckRespawn.

El hecho de instanciar un jugador se puede hacer fácilmente ya que el Game manager tiene un prefab de Player y puede acceder a él, pero para respawnear a los enemigos es más

dificil, ya que no es viable que el Game manager contenga a todos los diferentes enemigos de todos los niveles con su punto de respawn específico, y es por eso que para poder lograrlo se usan los eventos de Unity.

En el script del Game manager se crea el evento respawnEnemies y la función que llama a respawnEnemies en caso de que este no esté vacío.

```
public event Action respawnEnemies;

  public void RespawnEnemies()
  {
      if(respawnEnemies != null)
      {
          respawnEnemies();
      }
  }
```

Figura 91. Declaración del evento respawnEnemies y función RespawnEnemies

Una vez esas dos cosas están creadas, en el script Entity del que todos los enemigos heredan, en su Start se suscribe a este evento con su propia función RespawnEnemy que en caso de Entity está vacía pero existe para que sus hijos hagan override.

```
GameManager.getInstance().respawnEnemies += RespawnEnemy;
```

Figura 92. Suscripción al evento respawnEnemies del Game manager.

Ya en los enemigos específicos, se hace override y en este caso se comprueba si el gameObject está activo, en caso de no estarlo significa que el enemigo está en el dead state y entonces se le cambia al estado idle y se le lleva de vuelta al punto inicial que le toca.

```
public override void RespawnEnemy()
{
    if (!gameObject.activeSelf)
    {
        stateMachine.ChangeState(idleState);
        aliveGO.transform.SetPositionAndRotation(initialPosition, Quaternion.identity);
    }
}
```

Figura 93. Implementación de RespawnEnemy de un enemigo concreto.

18. Competencias técnicas

A continuación se explica cómo se han tenido en cuenta las competencias técnicas del proyecto.

CES1.1: Desenvolupar, mantenir i avaluar sistemes i serveis software complexos i/o crítics. [Bastant]

Este videojuego, como todo sistema software ha seguido este ciclo de vida. Primero empezó siendo diseñado y desarrollado y a lo largo que el proyecto iba creciendo, se fueron revisitando scripts anteriores para comprobar que no hubiera surgido ningún bug.

CES1.3: Identificar, avaluar i gestionar els riscos potencials associats a la construcció de software que es poguessin presentar. [Una mica]

En el apartado 3.3 que forma parte del GEP, se hace un análisis de los diferentes riesgos y obstáculos que se podrían haber presentado durante el desarrollo.

CES2.1: Definir i gestionar els requisits d'un sistema software. [Bastant]

Los requisitos de los stakeholders, que en este caso serían los jugadores, han sido implementados puesto que el videojuego funciona, tiene una gran diversidad de niveles y cuenta una historia interesante que mantiene al jugador interesado y por lo tanto lo invita a seguir jugando para descubrir cómo progresa.

CES2.2: Dissenyar solucions apropiades en un o més dominis d'aplicació, utilitzant mètodes d'enginyeria del software que integrin aspectes ètics, socials, legals i econòmics. [Una mica]

El proyecto hace hincapié en el aspecto ético y social debido al carácter de la historia de este.

CES3.1: Desenvolupar serveis i aplicacions multimèdia. [En profunditat]

Esta competencia técnica define básicamente este proyecto, ya que consiste en el desarrollo de una aplicación multimedia, en este caso un videojuego.

19. Conclusiones y trabajo futuro

Finalmente podemos decir que “Transcendental Journey” es un hecho y se ha logrado realizar correctamente. La elección de este trabajo fue por motivos personales y vocacionales, y gracias a él he adquirido conocimientos realmente valiosos sobre la creación de videojuegos que confío me sirvan en un futuro.

Para la realización de este proyecto se han empleado conocimientos adquiridos a lo largo de la carrera de diferentes asignaturas. Como por ejemplo IDI que ha servido para crear interfaces de usuario estéticas y funcionales o VJ por el hecho de introducirme a Unity y al diseño de videojuegos.

Considero que el producto final podría lanzarse al mercado en su estado actual, si sustituyera un sprite que pertenece a Spelunky por uno propio el resto de assets son open source, pero he decidido no hacerlo por un motivo personal: este juego narra mis experiencias y sentimientos pero aún me queda mucho por vivir respecto a mi trayectoria en la transición, y es por ello que como trabajo futuro quiero ampliar el juego con más mecánicas, más enemigos y sobre todo más historia y personajes que expliquen aquello que me ocurra.

20. Referencias

[1] «Facultat d'Informàtica de Barcelona |». [Online]. Disponible en: https://www.fib.upc.edu/. [Accedido: 16-mar-2020]

[2] «Página web oficial de Nintendo Ibérica», *Nintendo of Europe GmbH*. [Online]. Disponible en: https://www.nintendo.es/. [Accedido: 16-mar-2020]

[3] «Castlevania Wiki | Fandom». [Online]. Disponible en: https://castlevania.fandom.com/es/wiki/Castlevania_Wiki. [Accedido: 16-mar-2020]

[4] «Hollow Knight – An atmospheric adventure through a surreal, bug-infested world». [Online]. Disponible en: https://hollowknight.com/. [Accedido: 16-mar-2020]

[5] «Bravely Default | Final Fantasy Wiki | Fandom», *Final Fantasy Wiki*. [Online]. Disponible en: https://finalfantasy.fandom.com/wiki/Bravely_Default. [Accedido: 16-mar-2020]

[6] «Octopath Traveler Wiki | Fandom». [Online]. Disponible en: https://octopathtraveler.fandom.com/wiki/Octopath_Traveler_Wiki. [Accedido: 16-mar-2020]

[7] K. M. Farrar, M. Krcmar, y R. P. McGloin, «The Perception of Human Appearance in Video Games: Toward an Understanding of the Effects of Player Perceptions of Game Features», *Mass Communication & Society*, vol. 16, n.º 3, p. 299, may 2013.

[8] «¿Qué es Scrum?», *Scrum.org*. [Online]. Disponible en: https://www.scrum.org/resources/blog/que-es-scrum. [Accedido: 16-mar-2020]

[9] «Git». [Online]. Disponible en: https://git-scm.com/. [Accedido: 16-mar-2020]

[10] «The first single application for the entire DevOps lifecycle - GitLab», *GitLab*. [Online]. Disponible en: https://about.gitlab.com/. [Accedido: 16-mar-2020]

[11] Atlassian, «Gitflow Workflow | Atlassian Git Tutorial», *Atlassian*. [Online]. Disponible en: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Accedido: 16-mar-2020]

[12] «Free Git GUI Client - Windows, Mac & Linux | GitKraken», *GitKraken.com*. [Online]. Disponible en: https://www.gitkraken.com/. [Accedido: 16-mar-2020]

[13] W. C. Oechel y W. T. Lawrence, «Taiga», en *Physiological Ecology of North American Plant Communities*, B. F. Chabot y H. A. Mooney, Eds. Dordrecht: Springer Netherlands, 1985, pp. 66-94.

[14] «Color de la Tolerancia». [Online]. Disponible en: http://www.colores.org.es/color-de-la-tolerancia.php. [Accedido: 20-oct-2020]

[15] Eurogamer, «Miyamoto on World 1-1: How Nintendo made Mario's most iconic level», 07-sep-2015. [Online]. Disponible en: https://www.youtube.com/watch?v=zRGRJRUWafY. [Accedido: 20-oct-2020]

[16] «Brackeys». [Online]. Disponible en: https://www.youtube.com/channel/UCYbK_tjZ2OrIZFBvU6CCMiA. [Accedido: 20-oct-2020]

[17] «Press Start». [Online]. Disponible en: https://www.youtube.com/channel/UCe45-2uomTfrnGZwJcATeUA. [Accedido: 20-oct-2020]

[18] «Unity Forum». [Online]. Disponible en: https://forum.unity.com/. [Accedido: 20-oct-2020]

[19] «Bardent». [Online]. Disponible en: https://www.youtube.com/channel/UCKrEpRpu7isPB3p_nRv9Jwg. [Accedido: 20-oct-2020]