Universitat Oberta de Catalunya

uoc.edu

Creación del videojuego Time Ride en Unity

Aida Capó Martí
Grado en Multimedia
Trabajo de Final de Grado - Videojuegos

Ester Arroyo Garriguez
Joan Arnedo Moreno

07/06/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Creación del videojuego Time Ride en Unity*
Nombre del autor:	*Aida Capó Martí*
Nombre del consultor/a:	*Ester Arroyo Garriguez*
Nombre del PRA:	*Joan Arnedo Moreno*
Fecha de entrega (mm/aaaa):	06/2020
Titulación:	*Grado en Multimedia*
Área del Trabajo Final:	*Videojuegos*
Idioma del trabajo:	*Castellano*
Palabras clave	*Videojuego, Unity, Roguelike*

Resumen del Trabajo (máximo 250 palabras):

La presente memoria describe el proceso de conceptualización, desarrollo y pruebas de usuario de *Time Ride,* un videojuego para la plataforma Windows creado mediante el entorno de desarrollo Unity y el lenguaje de programación C#.

Time Ride es un *roguelike* 2D con estética píxel art en el que el jugador debe conducir al personaje a través de las distintas épocas históricas mientras recoge el combustible que ha perdió durante su viaje por el espacio-tiempo y vence a los enemigos que le impiden avanzar.

Una de las características principales del género *roguelike* es la aleatoriedad, así que uno de los retos del proyecto es diseñar un mapa generado de forma procedimental mediante el uso de variables aleatorias.

Dadas las restricciones temporales propias de un TFG, el producto resultante solo implementa un primer nivel, la era de los dinosaurios, y unos pocos enemigos. Sin embargo, la inteligencia artificial de estos enemigos, que utiliza una máquina de estados finitos para hacer la transición entre sus diferentes comportamientos, es bastante compleja y cumple con creces con las expectativas generadas durante la fase de planificación.

La realización de este proyecto nos deja lecciones importantes de cara a futuros proyectos, entre ellas la importancia de planificar meticulosamente los objetivos durante las fases iniciales, ya que las modificaciones en fases avanzadas pueden ser costosas y perjudicar el resultado final.

Abstract (in English, 250 words or less):

This report describes the process of designing, developing and testing *Time Ride,* a video game for Windows platform made with Unity development platform and C# programming language.

Time Ride is a 2D roguelike with pixel art graphics in which the player must guide the character through the different historical epochs while collecting the fuel he has lost during his space-time travel and defeating enemies that don't let him advance.

One of the main characteristics of the roguelike genre is randomness. Thus, one of the challenges of the project is to design a procedurally generated map using random variables.

Taking into account the temporal restrictions of a dissertation, the resulting product only implements the first level, the age of dinosaurs, and a few enemies. However, the artificial intelligence of these enemies, which uses a finite state machine to make the transition between their different behaviors, is quite complex and more than meets the expectations generated during the planning stage.

Carrying out this project has taught us important lessons for future works, including the importance of planning goals in detail during the initial stages, since modifications in advanced phases of the project can be both time consuming and expensive.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

Dentro del sector de los videojuegos, los entornos de desarrollo Unity y Unreal Engine están cobrando cada vez más importancia, y son muchos los juegos aclamados por la crítica que han sido desarrollados con estas plataformas de desarrollo comerciales. Un ejemplo claro de ello es *Ori and the Blind Forest*, hecho en Unity.

Así pues, este proyecto nace de la necesidad de profundizar en un entorno de programación de videojuegos ampliamente utilizado, que nos permita dar nuestros primeros pasos hacia el sector.

Sobre la elección del tipo de videojuego, ha sido motivada, por una parte, por el deseo de tener un primer contacto con la inteligencia artificial, y por otra parte, por la necesidad de escoger un género en que la jugabilidad prevaleciera antes de la estética, de forma que se pudiese conseguir un resultado de calidad sin que fuesen necesarios unos gráficos muy elaborados. En ambos sentidos, el *roguelike* con estética píxel art encaja a la perfección.

1.2 Objetivos del Trabajo

- Crear un videojuego con un alto grado de aleatoriedad, que derive en una elevada rejugabilidad.
- Programar una inteligencia artificial basada en una FSM (*Finite State Machine*), con un elevado nivel de abstracción a fin de que pueda ser usada para la creación de los comportamientos de todos los enemigos del juego.
- Planificar el proceso de creación de un videojuego desde su conceptualización hasta su versión Gold Master.
- Aprender a manejar un repositorio Git Hub a través de la herramienta Source Tree.
- Profundizar en el entorno de desarrollo Unity.
- Adquirir las nociones básicas de píxel art.

1.3 Enfoque y método seguido

Teniendo en cuenta que uno de los objetivos del proyecto es llevar a cabo el proceso de creación de un videojuego desde cero, se ha optado por desarrollar un producto nuevo en vez de adaptar un producto existente. Esta decisión nos permite, por un lado, profundizar más en la fase de conceptualización de la idea, puesto que debemos tomar decisiones que ya vendrían dadas en el caso de haber partido de un juego existente; procesos como la creación enemigos con interacciones originales y divertidas, la elección de un personaje principal carismático y de una historia lógica pero atractiva, el diseño de mecánicas

útiles y adaptadas al contexto, etc., son imprescindibles en la conceptualización de un producto nuevo, y probablemente sean innecesarios, o menos profundos, si se parte de una idea ya trabajada.

Por otro lado, esta decisión posibilita que, al final del proceso, podamos aprender de los errores que son fruto de la conceptualización; es decir, podamos analizar qué elementos funcionan de nuestro videojuego porque parten de una buena idea, así como qué funcionalidades resultan agradables y divertidas para el usuario. En resumen, nos ayuda a conocer cuáles de nuestras ideas son reutilizables para proyectos futuros, y cuáles deben ser descartadas.

1.4 Planificación del Trabajo

La primera decisión que se ha tomado referente a la planificación de objetivos ha sido el alcance del proyecto: debido al tiempo disponible, no resultaba realista crear más de un nivel; preferimos crear un solo nivel con todas las funcionalidades básicas implementadas para que el juego sea divertido, que no varios niveles con pocos elementos.

Los objetivos principales establecidos al inicio del proyecto son los siguientes:

1) Definición del proyecto

2) Diseño gráfico

   2.1) Diseño de *sprites* del personaje principal (Rufo).

   2.2) Diseño de *sprites* de los enemigos del nivel 1 (dilophosaurus, meganeura, raptor y Olghoï Khorkhoï).

   2.3) Diseño de *sprites* del escenario del nivel 1.

   2.4) Diseño de *sprites* del combustible y los elementos rompibles del nivel 1 (huevos de tres tipos).

   2.5) Diseño de *sprites* de los ítems.

3) Programación

   3.1) Generación de terreno de forma aleatoria: salas con forma irregular, conectadas entre ellas mediante pasillos. En función del nivel en que se encuentre el jugador, deben usarse unos *sprites* u otros para el escenario.

   3.2) Generación aleatoria de enemigos: según el nivel, deben aparecer unos enemigos u otros. La posición de los enemigos en la sala debe ser aleatoria.

   3.3) Movimiento del personaje principal, controlado por las flechas del teclado.

   3.4) Disparos del personaje principal, controlados con el ratón. Si la bala colisiona con un enemigo o elemento rompible, este debe recibir daño y morir si se queda sin vida.

3.5) Interacción con los enemigos: si el personaje choca con un enemigo, debe recibir daño, y si se queda sin corazones, debe morir.

3.6) Mecánica especial de Rufo: parar el tiempo con la barra espaciadora. Durante unos segundos, puede moverse pero no disparar. Mientras tanto, los enemigos no pueden ni moverse ni atacar.

3.7) Movimiento, ataque y *stats* de cada tipo de enemigo.

3.8) Comportamiento del combustible y de los elementos rompibles del primer nivel (huevos que pueden explotar, crear ítems o crear enemigos).

3.9) Generación aleatoria de los elementos (huevos y combustible).

3.10) Implementar sonidos.

3.11) Capacidad de guardar la partida y retomarla en otro momento.

4) Diseño de interfaces

4.1) Creación del menú principal.

4.2) Creación de la pantalla de opciones, que ofrezca la posibilidad de cambiar el idioma (entre español e inglés) y el volumen de la música y los efectos de sonido.

4.3) Creación de la pantalla de pausa.

4.4) Creación de la pantalla Game Over.

4.5) Creación de la pantalla de estadísticas del jugador.

5) Sonidos: si no se encuentran sonidos gratuitos que se adapten al proyecto, deberán ser de creación propia.

6) Animación: crear una cinemática inicial que sirva de introducción del juego.

7) Pruebas con usuarios: antes de la entrega final, testar el juego con usuarios reales que cumplan las características de nuestro público objetivo, para descubrir mejoras y correcciones necesarias.

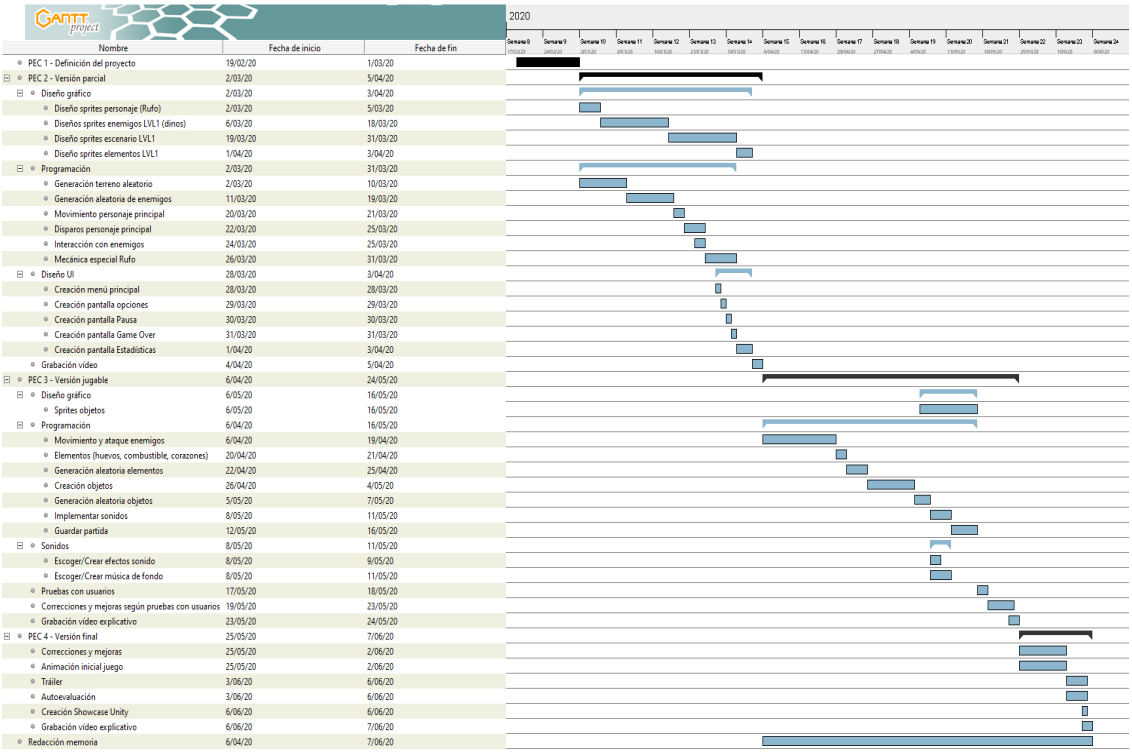
8) Creación de contenido para dar a conocer el juego:

8.1) Creación del tráiler.

8.2) Creación de un *showcase* en Unity.

9) Documentación del proceso en forma de memoria.

La planificación temporal de estos objetivos se muestra en el siguiente diagrama de Gantt:



Nombre	Fecha de inicio	Fecha de fin
PEC 1 - Definición del proyecto	19/02/20	1/03/20
PEC 2 - Versión parcial	2/03/20	5/04/20
Diseño gráfico	2/03/20	3/04/20
Diseño sprites personaje (Rufo)	2/03/20	5/03/20
Diseños sprites enemigos LVL1 (dinos)	6/03/20	18/03/20
Diseño sprites escenario LVL1	19/03/20	31/03/20
Diseño sprites elementos LVL1	1/04/20	3/04/20
Programación	2/03/20	31/03/20
Generación terreno aleatorio	2/03/20	10/03/20
Generación aleatoria de enemigos	11/03/20	19/03/20
Movimiento personaje principal	20/03/20	21/03/20
Disparos personaje principal	22/03/20	25/03/20
Interacción con enemigos	24/03/20	25/03/20
Mecánica especial Rufo	26/03/20	31/03/20
Diseño UI	28/03/20	3/04/20
Creación menú principal	28/03/20	28/03/20
Creación pantalla opciones	29/03/20	29/03/20
Creación pantalla Pausa	30/03/20	30/03/20
Creación pantalla Game Over	31/03/20	31/03/20
Creación pantalla Estadísticas	1/04/20	3/04/20
Grabación vídeo	4/04/20	5/04/20
PEC 3 - Versión jugable	6/04/20	24/05/20
Diseño gráfico	6/05/20	16/05/20
Sprites objetos	6/05/20	16/05/20
Programación	6/04/20	16/05/20
Movimiento y ataque enemigos	6/04/20	19/04/20
Elementos (huevos, combustible, corazones)	20/04/20	21/04/20
Generación aleatoria elementos	22/04/20	25/04/20
Creación objetos	26/04/20	4/05/20
Generación aleatoria objetos	5/05/20	7/05/20
Implementar sonidos	8/05/20	11/05/20
Guardar partida	12/05/20	16/05/20
Sonidos	8/05/20	11/05/20
Escoger/Crear efectos sonido	8/05/20	9/05/20
Escoger/Crear música de fondo	8/05/20	11/05/20
Pruebas con usuarios	17/05/20	18/05/20
Correcciones y mejoras según pruebas con usuarios	19/05/20	23/05/20
Grabación vídeo explicativo	23/05/20	24/05/20
PEC 4 - Versión final	25/05/20	7/06/20
Correcciones y mejoras	25/05/20	2/06/20
Animación inicial juego	25/05/20	2/06/20
Tráiler	3/06/20	6/06/20
Autoevaluación	3/06/20	6/06/20
Creación Showcase Unity	6/06/20	6/06/20
Grabación vídeo explicativo	6/06/20	7/06/20
Redacción memoria	6/04/20	7/06/20

Figura 1: Diagrama de Gantt con la planificación inicial

Esta planificación parte de la base de que gran parte del píxel art debía ser realizado por un compañero de trabajo. Sin embargo, debido a la situación del COVID-19 esto no ha sido posible, lo que ha obligado a modificar la planificación y a descartar objetivos para poder cumplir con los plazos de entrega. Estos objetivos han sido la implementación de guardado de partida y la creación de una cinemática inicial.

La planificación temporal de los objetivos finalmente ha sido la que sigue:

Nombre	Fecha de inicio	Fecha de fin
PEC 1 - Definición del proyecto	19/02/20	1/03/20
PEC 2 - Versión parcial	2/03/20	5/04/20
Diseño gráfico	2/03/20	4/04/20
Diseño sprites personaje (Rufo)	2/03/20	5/03/20
Diseño sprites escenario LVL1	19/03/20	31/03/20
Diseño sprites ratón	4/04/20	4/04/20
Programación	2/03/20	4/04/20
Generación terreno aleatorio I	2/03/20	6/03/20
Generación aleatoria de enemigos	6/03/20	14/03/20
Movimiento personaje principal	13/03/20	14/03/20
Disparos personaje principal	14/03/20	16/03/20
Interacción con enemigos	16/03/20	19/03/20
Mecánica especial Rufo	20/03/20	20/03/20
Generación terreno aleatorio II	1/04/20	4/04/20
Diseño UI	23/03/20	29/03/20
Creación menú principal	23/03/20	23/03/20
Creación pantalla opciones	23/03/20	25/03/20
Creación pantalla Pausa	25/03/20	25/03/20
Creación pantalla Game Over	26/03/20	26/03/20
Creación pantalla Estadísticas	27/03/20	29/03/20
Grabación vídeo	4/04/20	5/04/20
PEC 3 - Versión jugable	6/04/20	24/05/20
Diseño gráfico	6/04/20	19/05/20
Modificar sprites background	6/04/20	6/04/20
Sprites enemigos	7/04/20	20/04/20
Sprites elementos	18/04/20	22/04/20
Sprites personaje	23/04/20	7/05/20
Sprites objetos	8/05/20	19/05/20
Programación	6/04/20	16/05/20
Correcciones y mejoras	6/04/20	6/04/20
Movimiento y ataque enemigos	7/04/20	16/05/20
Elementos (huevos, combustible, corazones)	20/04/20	22/04/20
Generación aleatoria elementos	23/04/20	25/04/20
Creación objetos	26/04/20	1/05/20
Generación aleatoria objetos	2/05/20	4/05/20
Animator personaje	7/05/20	8/05/20
Implementar sonidos	8/05/20	11/05/20
Sonidos	8/05/20	11/05/20
Escoger/Crear efectos sonido	8/05/20	9/05/20
Escoger/Crear música de fondo	8/05/20	11/05/20
Pruebas con usuarios	17/05/20	18/05/20
Correcciones y mejoras según pruebas con usuarios	19/05/20	23/05/20
Grabación vídeo explicativo	23/05/20	24/05/20
PEC 4 - Versión final	25/05/20	7/06/20
Correcciones y mejoras	25/05/20	2/06/20
Diseño gráfico menú principal	25/05/20	31/05/20
Tráiler	1/06/20	6/06/20
Autoevaluación	3/06/20	6/06/20
Creación Showcase Unity	6/06/20	6/06/20
Grabación vídeo explicativo	6/06/20	7/06/20
Redacción memoria	6/04/20	7/06/20

Figura 2: Diagrama de Gantt con la planificación final.

1.5 Breve sumario de productos obtenidos

- Dos vídeos que resumen el estado del proyecto en el momento de las diferentes entregas (PEC2 y PEC3).
- Un vídeo de presentación del trabajo realizado hasta el momento de entrega del TFG.
- El tráiler del juego.
- Un informe de autoevaluación.
- La memoria final del proyecto.
- Un ejecutable con la versión Gold Master del videojuego *Time Ride* para la plataforma Windows.
- Un *showcase* en Unity Connect.

1.6 Breve descripción de los otros capítulos de la memoria

En los próximos capítulos nos proponemos a exponer los aspectos más relevantes relacionados con la creación y desarrollo del videojuego *Time Ride.*

En el capítulo 2 se explican las características comunes en los juegos definidos como *roguelikes* y se mencionan algunas de las plataformas de desarrollo más usadas en este tipo de juegos.

En el capítulo 3 se profundiza en la idea del juego y en su relación con los *roguelikes* existentes.

En los capítulos 4 y 5 se justifican las decisiones técnicas tomadas durante el proceso de creación (elección del entorno de desarrollo, diseño de la inteligencia artificial, etc.) y se indican los criterios de diseño seguidos para crear los niveles del juego de forma procedimental.

En el capítulo 6 se ofrece un manual de usuario con los requerimientos técnicos del *hardware* para jugar y las instrucciones de juego.

En el capítulo 7 se explica cómo se han llevado a cabo las pruebas de usuario y cuáles han sido las conclusiones extraídas.

En el capítulo 8 se exponen las conclusiones del proyecto. En concreto, se hace una reflexión sobre las lecciones aprendidas durante el proceso y la consecución de objetivos y su planificación, y se habla del futuro del juego.

Por último, el capítulo 9 contiene un glosario con la definición de los términos más importantes usados a lo largo de este documento.

2. Estado del arte

2.1. El género del juego

Time Ride pertenece a la familia de los *roguelikes* o videojuegos de mazmorras. Hasta hace poco, el término “roguelike” se utilizaba para designar videojuegos que eran muy similares a *Rogue*, un juego de mazmorras creado en 1980 que se convirtió en el padre del género que recibió su nombre. Así pues, un “roguelike” era un juego que incorporaba las principales características de *Rogue.* A continuación se describen algunas de las más importantes:

- Entornos generados de forma procedimental: esto significa que cada mapa es creado mediante un algoritmo aleatorio, lo que aumenta la rejugabilidad, ya que los escenarios son diferentes cada vez que se ejecuta el juego.
- Muerte permanente (conocida como *permadeath* en el sector): cuando muere, el jugador pierde todo el progreso (posición, nivel, ítems, etc.). Tal y como expone Alexander King, “la intención es que lo único que el jugador mantiene de un juego a otro es su propia habilidad y conocimiento del sistema” (King, 2015) [1]. Esto hace que, en general, la dificultad elevada se considere otra característica básica.
- Aparición de ítems de forma aleatoria: al igual que el mapa, generalmente los ítems se colocan en posiciones aleatorias, y sus efectos desaparecen al morir.
- La jugabilidad prevalece antes de la estética.

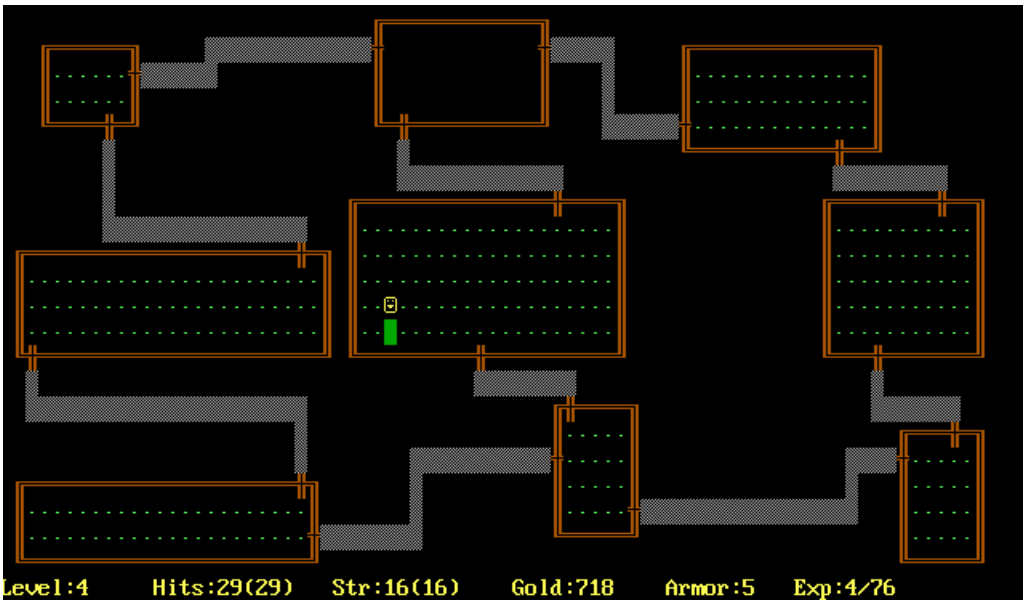


Figura 3: Videojuego *Rogue* (1980), padre de los *roguelikes*. Imagen extraída de Wikimedia Commons.

Asimismo, *Rogue* y sus descendientes directos formaban parte del género RPG (*Role Playing Game*), por lo que las particularidades del *roguelike* se encontraban combinadas con las características de los juegos de rol, motivo por el cual en muchas definiciones del concepto “roguelike” se añaden las siguientes propiedades a la lista anterior:

- Combate por turnos
- Entornos basados en una cuadrícula (*grid,* en inglés)
- Muerte de monstruos

Con el paso del tiempo, juegos que no pertenecen al género RPG han incorporado las tres características principales de *Rogue* expuestas al principio, con lo que en la actualidad un juego puede definirse como un “roguelike” a pesar de que no implementa las funcionalidades de *Rogue* estrechamente relacionadas con el género RPG. En definitiva, más que a un género en sí mismo, el concepto “roguelike” hace referencia a un conjunto de mecánicas y principios de diseño.

Como resultado, videojuegos muy diferentes como por ejemplo *Spelunky* (juego de plataformas de acción), *Dwarf Fortress* (exploración y construcción de bases) y *The Binding of Isaac* (videojuego de rol de acción o ARPG) forman parte de la familia de *roguelikes*.

Figura 4: Videojuego *The Binding of Isaac: Afterbirth+* (2015), de Edmund McMillen.

Figura 5: Videojuego Spelunky (2008), de Derek Yu.

2.2. La tecnología del género

Algunos de los videojuegos más populares de la familia de los *roguelikes* utilizan el entorno de desarrollo Game Maker Studio. Es el caso de *Spelunky* y *Nuclear Throne.* Otros, como por ejemplo *Enter the Gungeon* y *Moonlighter,* prefieren la plataforma Unity. Además, también existen juegos que han optado por un motor propio, como es el caso de *The Binding of Isaac*, que utiliza un motor escrito en C++ por su desarrollador.

Respecto a Game Maker, se trata de un entorno de desarrollo especializado en el desarrollo de videojuegos en dos dimensiones, y cuenta con una interfaz amigable e intuitiva, aunque poco adaptable. Su versión de pago más compleja permite exportar para Windows, Linux, HTML5, Android, iOS, macOS, Windows

UWP, PS4, Xbox One y Nintendo Switch. En cuanto a sus puntos débiles, destacan su licencia gratuita muy limitada y la falta de un editor interno de animaciones. Asimismo, a diferencia de Unity, Game Maker solo soporta el lenguaje GML (*Game Maker Language*), no permite alternan con otro lenguaje.

Por lo que se refiere a Unity 3D, si bien es un entorno inicialmente creado para juegos en tres dimensiones, también cuenta con muchas funcionalidades para las obras en 2D. Su editor es intuitivo y adaptable, y su tienda en línea de *assets* es la más completa que existe en la actualidad, lo que puede facilitar mucho la tarea de los desarrolladores. Además, permite exportar a una gran variedad de plataformas: WebGL, Windows, SteamOS, OS X, Linux, Windows Store Apps, iOS, Android, Windows Phone, Xbox 360, Xbox One, PlayStation Vita, Nintento Switch, etc.

3. Definición del juego

3.1. Subgénero y referencias a videojuegos existentes

El juego pertenece a la familia de los *roguelikes.* A continuación se muestra la relación entre algunas de las características más destacadas de esta familia y las características de *Time Ride*:

- Generalmente, en los *roguelikes* el objetivo principal es alcanzar el último nivel.
  - En *Time Ride,* el objetivo principal es sobrevivir hasta el último nivel: el futuro. De todos modos, el videojuego realizado para este trabajo solo cuenta con un primer nivel, la era de los dinosaurios, si bien se pretende seguir creando niveles más adelante.
- En los *roguelikes,* se pone especial énfasis en la generación de contenido de forma aleatoria.
  - En *Time Ride,* al inicio de cada nivel se genera un mapa en que las siguientes variables se calculan de forma aleatoria: número de salas y anchura y altura de estas, número de irregularidades de las paredes de cada sala, número de elementos rompibles y su tipo, número de enemigos de cada sala y su tipo, probabilidad de que un enemigo deje un ítem al morir, y tipo de ítem que deja.
- En los *roguelikes,* la muerte es permanente.
  - En *Time Ride,* también lo es; si el jugador muere, pierde todas las mejoras que le hayan podido aportar los ítems, y debe empezar de cero en la primera sala del primer nivel.

Otra característica que podemos encontrar en algunos de los *roguelikes* más conocidos (por ejemplo, en *The Binding of Isaac* o en *Enter the gungeon*) es la presencia de la tienda, una sala en la que poder comprar objetos con las monedas recogidas en las salas anteriores. En nuestro caso, sin embargo, no habrá tienda, puesto que no parece encajar en el contexto del juego.

Algunos de los roguelikes más conocidos son *Moonlighter*, *Enter the Gungeon, Nuclear Throne*, *The Binding of Isaac* y *Spelunky,* y si bien todos cumplen las características comentadas anteriormente, presentan diferencias destacables, como por ejemplo la posición de la cámara (en todos ellos vemos el personaje desde arriba, excepto en *Spelunky*, que la cámara se sitúa a la altura del personaje), la forma del mapa aleatorio (por ejemplo, en *The Binding of Isaac* se crean salas cuadradas o rectangulares, sin pasillos, comunicadas entre ellas a través de puertas, mientras que en *Nuclear Throne* apenas se pueden diferenciar salas, y no se utilizan puertas), etc. En nuestro juego, hemos decidido optar por una cámara similar a la de *Enter the gungeon*, y por una generación del mapa parecida a la de *Nuclear Throne* pero sin tantos pasillos.

Figura 6: Videojuego *Nuclear Throne* (2015), de la desarrolladora Vlambeer.

3.2. Historia, ambientación y trama

La historia empieza en el año 2019, en el estudio de Rufo, un historiador que desea desvelar todas las incógnitas de la era Mesozoica y para ello está creando la máquina del tiempo (una bicicleta que funciona gracias a un extraño líquido azul) que le permitirá viajar hasta esa época y estudiar el comportamiento de los dinosaurios.

El problema del protagonista empieza cuando llega a su destino y se da cuenta de que su máquina del tiempo ha ido perdiendo combustible mientras viajaba por las distintas eras históricas, por lo que deberá ir haciendo pequeños saltos en el tiempo en busca del combustible. Así, deberá volver a las siguientes eras:

1) Era de los dinosaurios
2) Prehistoria
3) Edad Antigua
    3.1) Antiguo Egipto
    3.2) Antigua Grecia
    3.3) Imperio Romano
4) Edad Media
    4.1) Castillos y cruzadas
    4.2) Vikingos
    4.3) El pueblo (las ratas de la Peste)
5) Edad Moderna

6) Edad Contemporánea

7) Futuro

Asimismo, navegar en el tiempo puede modificar la historia tal y como la conocemos, y el protagonista se dará cuenta de ello durante su aventura, durante la cual puede que se encuentre con cavernícolas armados con revólveres, ratas con rayos láser, entre muchas otras cosas extrañas fruto de haber jugado con el frágil espacio tiempo.

3.3. Definición de los personajes y elementos

El personaje protagonista del juego es **Rufo**, un historiador que pretende viajar en el tiempo para estudiar la fauna y flora de la era Mesozoica. Puesto que no sabe qué esperar de las criaturas de la época, va equipado con un traje protector capaz de disparar balas y de parar el tiempo a su alrededor durante unos segundos. Esto último le permitirá posicionarse mejor durante un ataque.

Respecto a los enemigos, varían en cada nivel o etapa histórica. En el primer nivel, la era de los dinosaurios, Rufo puede encontrarse con los siguientes enemigos:

1) **Velociraptor**: dinosaurio terrestre carnívoro que corre a gran velocidad para matar a sus presas.

2) **Dilophosaurus**: criatura terrestre que escupe veneno a cualquier elemento o ser que le resulte desconocido. Este veneno ralentiza el movimiento del afectado.

3) **Meganeura**: insecto volador carnívoro que convive en grupos de hasta 12 miembros de su misma especie. Si bien no suele ser agresivo, ataca si se siente amenazado o hambriento. Así pues, atacará al protagonista si este se acerca demasiado.

4) **Olghoï-Khorkhoï (jefe final)**: gusano gigante que habita en el desierto. Vive bajo tierra, y cuando detecta a un enemigo sale rápidamente a la superficie y lo ataca. La primera fase de su ataque consiste en escupir bolas venenosas de forma radial, y si esto no surge efecto genera hijos (gusanos pequeños) que persiguen al enemigo.

Por otra parte, Rufo también puede hallar huevos, que son elementos que, al ser golpeados, presentan diferentes comportamientos según su color:

1) **Huevos rojos**: producen una explosión al recibir un golpe, la cual inflige daño a cualquier criatura que se encuentre cerca.

2) **Huevos azules**: cuando se rompen, aparece un enemigo normal de los descritos anteriormente.

3) **Huevos dorados**: cuando se rompen, generan un ítem.

En cuanto a los ítems, que pueden aparecer al romper un huevo dorado o al matar a un enemigo, tienen un nivel de rareza entre 1 y 5, siendo los ítems de rareza 1 los más fáciles de encontrar, y los ítems de rareza 5 los que es menos

probable que aparezcan, ya que solo los jefes finales generan ítems de ese grado de rareza. En la era de los dinosaurios se pueden encontrar los siguientes:

- Ítems de rareza 1:
  - **Corazón**: el personaje recupera una vida.
  - **Ala de Meganeura**: aumenta la velocidad de movimiento del protagonista.
- Ítems de rareza 2:
  - **Seta de crecimiento rápido**: disminuye la cadencia del arma del personaje.
  - **Seta podrida**: disminuye la velocidad de movimiento de Rufo y cambia sus balas a balas venenosas, que ralentizan a los enemigos.
- Ítems de rareza 3:
  - **Corazón de Dilo**: aumenta ligeramente la cadencia de disparo de Rufo pero, a cambio, le da un corazón base vacío, es decir, un corazón más que poder rellenar.
- Ítems de rareza 4:
  - **Seta enorme**: aumenta el tamaño de la bala del personaje.
- Ítems de rareza 5:
  - **Diente de sable**: elimina un corazón base del personaje, y a cambio disminuye la cadencia de disparo y le da balas mucho más dañinas que las balas por defecto.
  - **Gota cristalina**: modifica el tipo de disparo del personaje, de forma que se disparan 2 balas, una en la dirección en que apunta Rufo y otra en la dirección contraria.

3.4. Interacción entre los actores del juego

A continuación se expone un ejemplo de cómo avanzaría el personaje por el primer nivel, si bien debe tenerse en cuenta que nunca habrá dos niveles iguales, ya que el tipo, la posición y el número de enemigos y de huevos es aleatorio:

1. El personaje aparece en el primer nivel, y se mueve a través de la primera sala, en la que se encuentra con dos dilophosaurus que están patrullando por la sala. Cuando divisan a Rufo (es decir, cuando este está a una distancia igual o menor al rango de detección de los dilophosaurus), empiezan a seguirlo, y cuando están a una distancia inferior o igual a su rango de ataque, escupen una bola venenosa en dirección al personaje. Si la bola alcanza al personaje, este pierde velocidad de movimiento durante un corto tiempo. El personaje dispara hacia los Dilophosaurus y los mata. Camina hacia la segunda sala.

2. En la segunda sala se encuentra con un huevo rojo y dispara para romperlo. Cuando la bala toca el huevo, este explota. El personaje no se ha apartado a tiempo, así que recibe daño (pierde un corazón), pero también una meganeura que se encontraba cerca del huevo ha recibido daño. Rufo dispara varias veces más a la meganeura herida y a las dos compañeras que vuelan a su lado, hasta matarlas. Al morir, una de ellas deja un corazón, que cura una vida del protagonista.

3. Al entrar en la tercera sala, el aventurero pasa demasiado cerca de un grupo de 10 meganeuras, que empiezan a volar hacia él rápidamente. Desde esa posición no puede matarlas a todas sin morir en el intento, así que usa su habilidad especial para parar el tiempo unos segundos y moverse unos pasos más atrás; durante este tiempo no puede atacar, solo moverse. El tiempo vuelve a la normalidad, y el personaje ahora se encuentra mejor posicionado y puede disparar a todas las meganeuras.

4. En la última sala se encuentra con el jefe final, el Olghoï-Khorkhoï, que primero dispara doce balas venenosas de forma radial y luego se esconde bajo tierra y aparece en una nueva posición. El enemigo continúa con este proceso hasta que el personaje consigue vaciarle la mitad de la barra de vida. Cuando esto sucede, el gusano empieza a generar 4 gusanos cada 8 segundos, los cuales siguen al personaje. Si consiguen tocarle, le restan una vida.

5. Finalmente Rufo consigue esquivar los gusanos y seguir disparando al Olghoï, que muere y deja una seta enorme, que aumenta el tamaño de la bala del protagonista, y un frasco de combustible, que podrá utilizar para viajar hasta la próxima época histórica.

3.5. Objetivos planteados al jugador

- Matar al jefe final de cada nivel a fin de obtener el combustible para viajar hasta la próxima época histórica. Se trata de alcanzar el nivel final: el futuro.

- Matar a todos los enemigos de las salas. Esto no es un requisito indispensable, pero resulta conveniente hacerlo, ya que cualquier enemigo tiene una pequeña posibilidad de generar un ítem. Asimismo, los enemigos pueden moverse entre salas, por lo que conviene matar los enemigos de una sala antes de pasar a la siguiente, puesto que de lo contrario el nivel de dificultad puede aumentar.

- Aprovechar las ventajas que ofrece la habilidad especial del personaje.

- Sobrevivir hasta alcanzar el nivel final, ya que si el personaje muere se pierden todas las mejoras obtenidas hasta el momento.

3.6. *Concept Art*

A continuación se presentan algunas de las referencias que sirvieron de ayuda para definir la idea y estilo del juego:

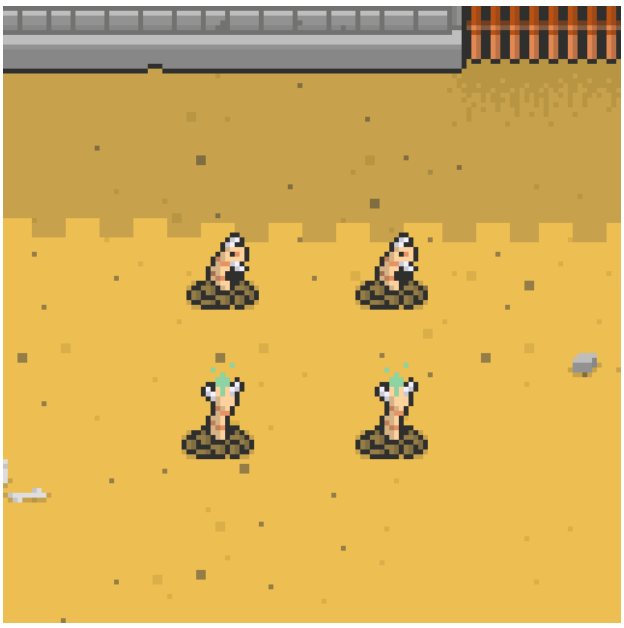


Figura 7: Imagen que inspiró la idea del Olghoï Khorkhoï. Tomada de OpenGameArt.org. CC-BY 4.0.

Figura 8: Imagen que sirvió de referencia para el diseño del menú inicial. Tomada de la web Tenor.

Figura 9: Videojuego *Moonlighter* (2018), de Digital Sun Games, que nos ayudó a escoger la perspectiva de la cámara.

Figura 10: Videojuego *Sparklite* (2019), de Red Blue Games, que inspiró la idea de escoger la estética píxel art.

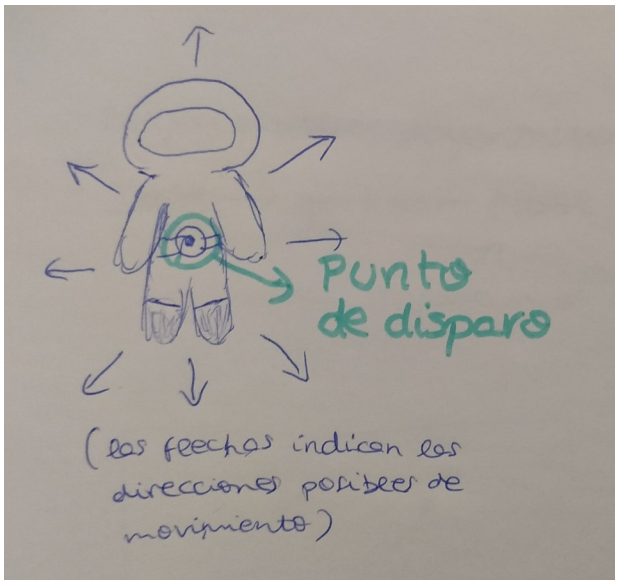


Figura 11: Primer boceto del protagonista.

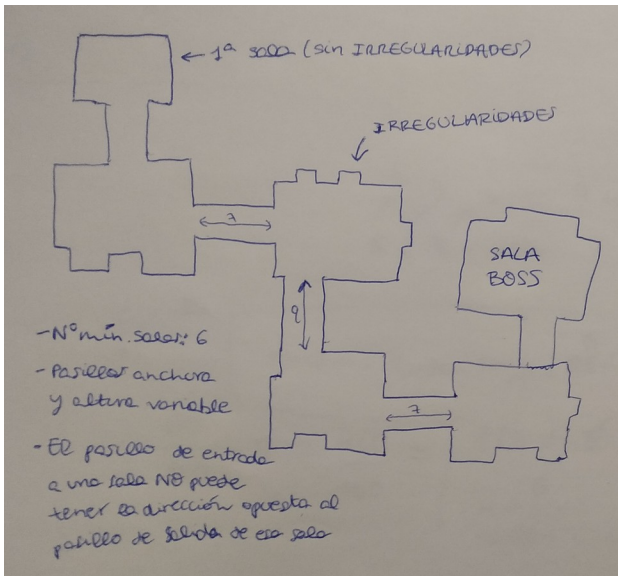


Figura 12: Boceto con las características básicas del mapa, realizado al inicio de la etapa de desarrollo.

4. Diseño técnico

4.1. Justificación del entorno escogido

Puesto que uno de los motivos principales de haber escogido el área de videojuegos para este trabajo final era poder experimentar a fondo con un motor ampliamente utilizado en el sector, desde el inicio tuvimos claro que la elección del entorno de desarrollo debía estar entre los dos motores más utilizados en la actualidad: Unity y Unreal Engine.

Tras analizar ambos entornos, se escogió Unity por las siguientes razones:

- Unreal Engine utiliza C++, un lenguaje más potente y, por consiguiente, más complejo que el lenguaje C# usado por Unity.
- Unity presenta una interfaz más amigable e intuitiva que Unreal Engine, por lo que parece más apropiada para usuarios sin mucha experiencia con entornos de desarrollo de videojuegos.
- Ya había realizado diversos proyectos en Unity, y preferí dedicar tiempo a pulir mis capacidades de programación con C# a tener que aprender a utilizar el motor Unreal Engine y su lenguaje C++ de cero.

Desde que iniciamos la fase de desarrollo del juego, comprobamos que efectivamente Unity era el entorno ideal para el proyecto, especialmente porque ofrece una interfaz fácil de entender, así como una gran cantidad de recursos de gran calidad disponibles desde su *Assets Store*. Además, cuenta con una comunidad enorme, por lo que resulta muy fácil encontrar información en la red.

4.2. Requerimientos técnicos del entorno de desarrollo

Según la documentación oficial de Unity [2], los requerimientos técnicos de la versión 2019.1.3f utilizada para el desarrollo del proyecto son los siguientes:

Tabla 1: Requerimientos técnicos de Unity 2019.1

Sistema	**Requerimientos mínimos**
Sistema operativo	**Windows**: 7 SP1+, 8, 10, solo versiones 64-bit **macOS**: 10.12+
CPU	Conjunto de instrucciones SSE2.
GPU	Tarjetas gráficas con capacidades DX10
Dispositivos	
iOS	Ordenador Mac con macOS 10.12.6 y Xcode 9.4 o superior
Android	Android SDK y Java (JDK).

	El backend de secuencias de comandos IL2CPP requiere Android NDK.
Plataforma Windows Universal	Windows 10 (64-bit), Visual Studio 2015 con componente *C++ Tools* o posterior y Windows 10 SDK

Cabe mencionar que se ha utilizado el entorno Unity en un sistema operativo Windows 10 x64.

4.3. Inventario de herramientas empleadas

A continuación se indican las herramientas empleadas a lo largo de todo el proceso de creación del videojuego:

- **GanttProject**: programa de código abierto con licencia GPL utilizado para la creación del diagrama de Gantt.
- **Aseprite**: se trata de un editor de gráficos de pago diseñado específicamente para la creación de *sprites* y animaciones de estilo píxel art. Se ha escogido Aseprite para la creación de los gráficos del juego por su interfaz intuitiva y por ser uno de los software más utilizados para píxel art.

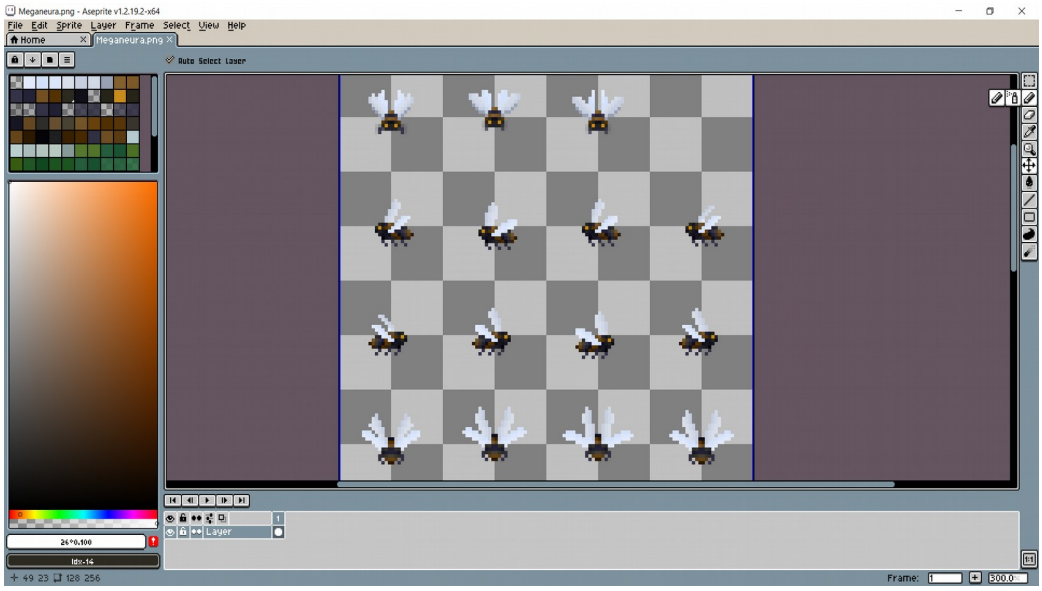


Figura 13: Aseprite, software utilizado para la creación de los gráficos.

- **Unity 2019.1.3f**: motor de videojuegos multiplataforma disponible para Microsoft Windows, Mac Os y Linux y que permite utilizar los lenguajes C# y JavaScript.
- **Microsoft Visual Studio**: se trata del entorno de desarrollo integrado utilizado para escribir el código de *Time Ride*. Es compatible con una

gran variedad de lenguajes de programación, entre ellos C#, C++, Java y PHP.

- **Wondershare Filmora**: se ha utilizado Filmora para la edición de los vídeos de presentación de la PEC2 y la PEC3. Su elección fue motivada por su fácil curva de aprendizaje y su interfaz intuitiva.
- **Adobe After Effects**: el software de edición de vídeos After Effects se ha empleado para la creación del tráiler y del vídeo de presentación final. Mientras que Filmora presenta solo las características básicas de un software de edición de vídeo y por ello resulta muy fácil aprender a usarlo, After Effects ofrece una amplia variedad de opciones para composición, realización de gráficos profesionales en movimiento y efectos especiales, lo que hace que su curva de aprendizaje sea más pesada, pero permite obtener resultados de alta calidad profesional. Por este motivo, Filmora se ha escogido para editar vídeos simples que requerían de poca edición, y Adobe After Effects se ha utilizado para los vídeos que requerían de un acabado más profesional.
- **Adobe Illustrator**: software comercial utilizado para crear el diagrama de flujos del juego.
- **Adobe Audition**: software comercial de edición de audio utilizado para editar algunos efectos de sonido del juego.
- **Source Tree**: aplicación de escritorio gratuita que nos ha permitido interactuar con el repositorio Git a través de una interfaz simple e intuitiva.

4.4. Inventario de recursos empleados

4.4.1. Fuentes de texto

- **PF Arma Five**: se trata de una fuente de texto gratuita de la web Dafont (https://www.dafont.com/es/pf-arma-five.font#null).

  Esta fuente se utiliza en todos los textos del juego, excepto en los números flotantes que indican el daño que recibe un enemigo al ser golpeado.

- **Teko**: fuente de código abierto de Google Fonts (https://fonts.google.com/specimen/Teko).

  La fuente Teko es la que usan los números flotantes que indican el daño.

4.4.2. Gráficos

En primer lugar, cabe destacar que todos los recursos gráficos tienen una estética píxel art, a fin de que interfaz y elementos del juego se perciban como una unidad. Asimismo, desde la fase inicial del proyecto nuestra intención ha sido conseguir una estética *top down*, por lo que se ha intentado que todas las ilustraciones de los elementos del juego (huevos, enemigos, explosiones, etc.)

parezcan captadas por una cámara que se encuentra un poco más arriba que el propio elemento.

A continuación se exponen los recursos gráficos de terceros que se encuentran implementados en el videojuego:

- **Explosión**:

Figura 14: Explosión animada, extraída del paquete *Insect Enemy Assets* de la web Itch.io.

Se trata de la explosión que se instancia cuando el jugador golpea un huevo rojo. Para que fuese más vistosa y más coherente con el color del huevo, en Unity se ha modificado su color a un tono más anaranjado.

- **Olghoï Khorkhoï e hijos**:

Figura 15: Gusanos pequeños (hijos del Olghoï Khorkhoï), extraídos del paquete *Insect Enemy Assets* de la web Itch.io.

Para crear las animaciones del hijo del Olghoï se han utilizado los *sprites* del paquete *Insect Enemy Assets.*

Respecto al Olghoï Khorkhoï, hemos usado el software Aseprite para modificar los *sprites* de sus hijos y conseguir así un gusano gigante con un aspecto similar a estos. El resultado es el siguiente:

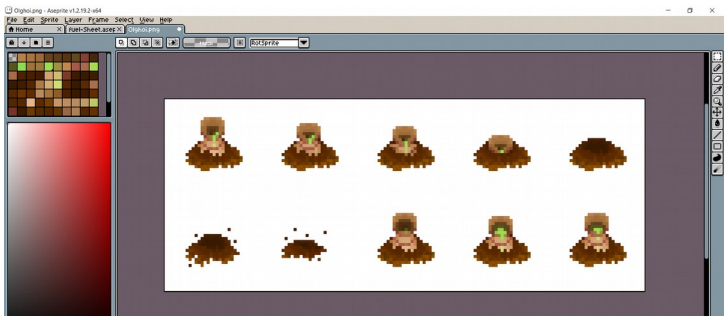


Figura 16: Sprites del Olghoï Khorkhoï creados con Aseprite.

- **Explosión de las balas**:

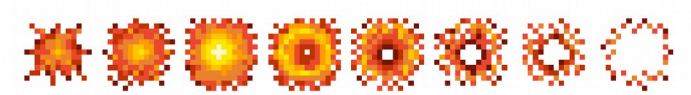


Figura 17: Animación de 8 *frames* de una explosión, recuperada de OpenGameArt.org.

Esta animación se instancia cuando una bala muere, ya sea porque se ha terminado su tiempo de vida o porque ha colisionado con otro elemento. Puesto que hay diferentes tipos de balas, en Unity cambiamos el color de la explosión según el tipo de bala que la haya originado.

En cuanto a los gráficos de creación propia, son los siguientes, todos ellos creados con Aseprite:

- **Huevos**:

Figura 18: Huevos (elementos rompibles del primer nivel)

El gris de las sombras no es opaco, para que el color de la sombra se adapte al color del suelo del mapa.

- **Cursores**:

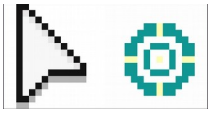


Figura 19: Cursor para interactuar con la interfaz y cursor de la pantalla de juego.

El cursor para interactuar con la interfaz tiene unas medidas de 11x16 píxeles, y el cursor en forma de mirilla, utilizado durante la partida, es de 11x11 píxeles.

- **Barra de vida del jefe final**:

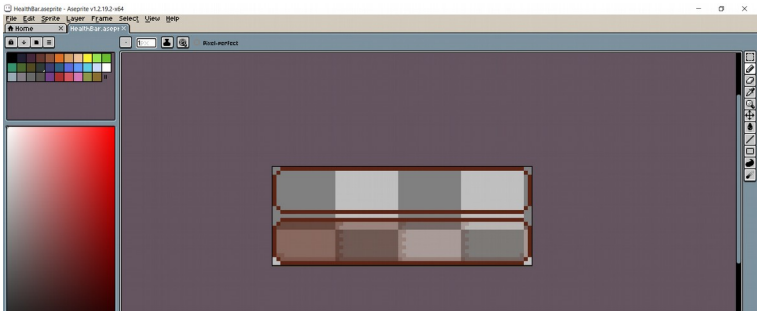


Figura 20: Captura de pantalla de la barra de vida en Aseprite.

La barra de consta de tres partes: la primera es el *sprite* superior que se muestra en la imagen, cuyo tamaño y opacidad no se modifica durante el juego. La segunda es el *sprite* inferior de la imagen, que pretende ser el sombreado de la barra, y la tercera es el relleno, creado desde el propio Unity. En estas dos últimas imágenes se sigue el mismo procedimiento: la cantidad que se muestra de la imagen depende de la cantidad de vida del jefe final. Es decir, si tiene la mitad de la vida, solo se verá la primera mitad del *sprite* de sombras y la primera mitad del cuadro de relleno.

- **Enemigos normales**:

Figura 21: *Sprites* de los enemigos normales (dilophosaurus, raptor y meganeura).

- **Imagen de fondo del menú inicial**:

Figura 22: Imagen de fondo del menú inicial.

La idea de este diseño es que el usuario pueda interactuar con el menú como si se tratara del ordenador de Rufo. De todos modos, en la versión del juego que se entrega para este TFG puede que el jugador no llegue a la conclusión de que se trata del ordenador del protagonista, porque no hemos tenido tiempo de crear la cinemática inicial en la que pretendíamos que se viera a Rufo en su estudio, con su escritorio y sus máquinas.

- **Elementos de la interfaz de usuario del menú**:

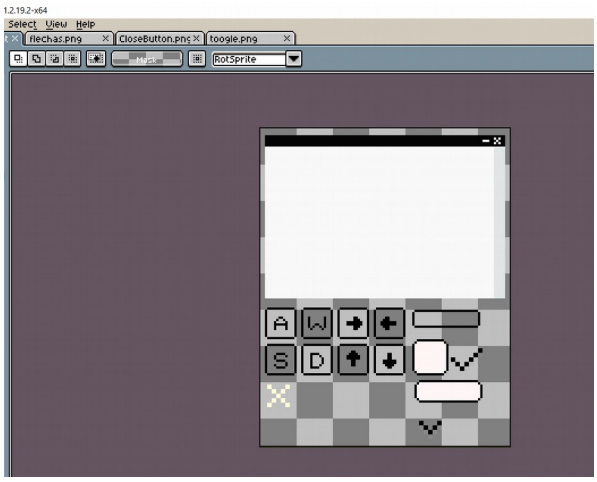


Figura 23: Captura de pantalla de los elementos del menú en Aseprite.

- **Logotipo Team Bee**:

Figura 24: Logotipo de nuestro equipo de desarrollo.

- **Ítems**:

Figura 25: Sprites de los ítems del primer nivel.

Se trata de diseños provisionales; no estamos convencidos con su resultado, y pretendemos mejorarlos en el futuro.

- **Combustible**:

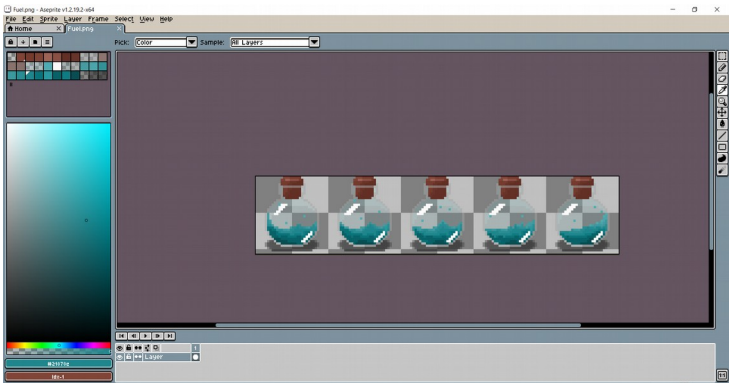


Figura 26: Animación del combustible en Aseprite.

- ***Sprites* del mapa**:

Figura 27: Sprites del mapa del primer nivel.

Se trata de *sprites* de 32 x 32 píxeles. Cabe destacar que algunas de las decoraciones no están implementadas en la versión final de este TFG, ya que creemos que su forma es demasiado similar a la de los charcos y puede resultar confuso.

- **Corazones de la interfaz de la pantalla de juego**:

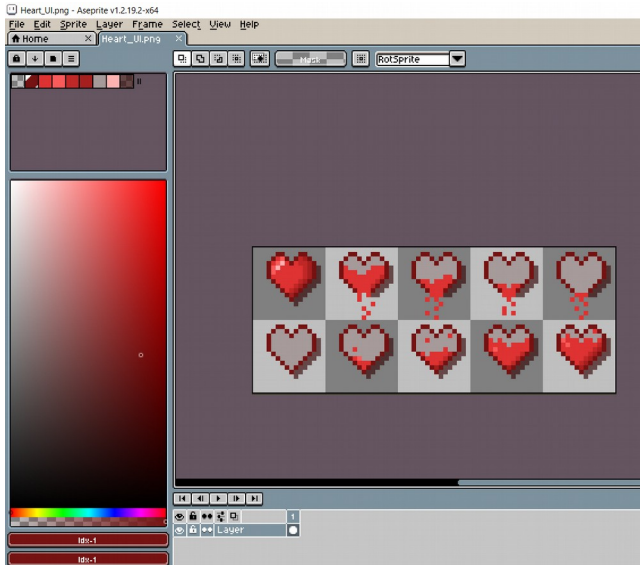


Figura 28: Animación de los corazones en Aseprite.

- **Balas**:

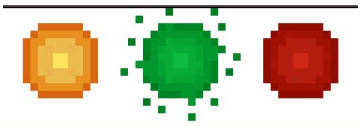


Figura 29: Sprites de los tres tipos de bala implementados.

- **Rufo**:

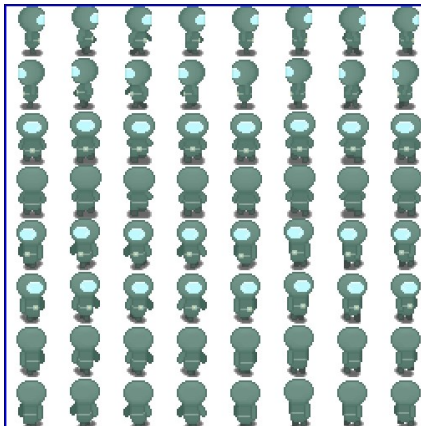


Figura 30: Sprites de Rufo en sus 8 direcciones posibles.

4.4.3. Sonidos

Ninguno de los sonidos utilizados es de creación propia, sino que se trata de sonidos de terceros con licencias no restrictivas. Listamos a continuación las fuentes de todos ellos:

- **Música de fondo, sonido de la pantalla 'Game Over' y sonido del botón 'Nueva partida'**: Asset gratuito de Unity *Action RPG Music Free V1.3.*
- **Música del menú**: *Owl Woods,* de la web PlayOnLoop (https://www.playonloop.com/2018-music-loops/owl-woods/)
- **Música del tráiler**: *Heroism,* de Edward J.Blakeley (https://opengameart.org/content/heroism).
- ***Hover* de los botones y sonido del botón 'Guardar'**: Paquete de sonidos *GUI Sound Effects* del usuario *Lokif* de Open Game Art (https://opengameart.org/content/gui-sound-effects).
- **Pulsar un botón**: Sonido *Click Tick* del usuario *malle99* de Free Sound (https://freesound.org/people/malle99/sounds/384187/).
- **Perder una vida**: Sonido *8-bit Hurt1.aif,* del usuario *timgormly* de Free Sound (https://freesound.org/people/timgormly/sounds/170148/).

- **Explosión del huevo rojo**: Sonido *Rumble/Explosion,* de Michel Baradari (https://opengameart.org/content/rumbleexplosion).
- **Disparo de Rufo**: Sonido *laser*, del usuario *Leszek_Szary* de Free Sound (https://freesound.org/people/Leszek_Szary/sounds/146725/).
- **Disparo, golpe y muerte del dilophosaurus**: Sonido *Slimy monster or murder sounds(9)*, del usuario *pauliuw* de Oper Game Art (https://opengameart.org/content/slimy-monster-or-murder-sounds9).
- **Golpe y muerte de la meganeura**: Sonido *hit3.wav*, del usuario *Tissman* de Free Sound (https://freesound.org/people/Tissman/sounds/456168/).

  Cabe comentar que el sonido de la muerte es una modificación del sonido del golpe hecha en Adobe Audition.
- **Muerte, golpe y aparición del Olghoï**: Sonido *Monster Sound Effects Pack*, del usuario *Ogrebane* de Open Game Art (https://opengameart.org/content/monster-sound-effects-pack).
- **Golpe del raptor**: Sonido *Attack miss or hit sounds (2),* del usuario *pauliuw* de Open Game Art (https://opengameart.org/content/attack-miss-or-hit-sounds2).
- **Muerte del raptor**: Sonido *Mutant Death,* del usuario *Gobusto* de Open Game Art (https://opengameart.org/content/mutant-death).

4.4.4. Assets para el inspector de Unity.

Para crear las salas sin tener que estar pendientes de si la posición actual es una esquina, o la parte central de la pared, o una pared inferior, etc. utilizamos los *Rule Tiles* del asset **2d-extras** de Unity Technologies. Esto reduce las comprobaciones a realizar mediante código, en el que solo deberemos indicar si el *tile* actual es una pared de sala, una pared exterior, el suelo de la sala, agua o un obstáculo, y se pinta un *tile* u otro mediante las reglas indicadas en los *Rule Tiles*. Asimismo, esta forma facilita la creación de los diferentes niveles: es tan simple como crear *Tile Palettes* para cada nivel, y usar un *Tile Palette* u otro según el nivel.

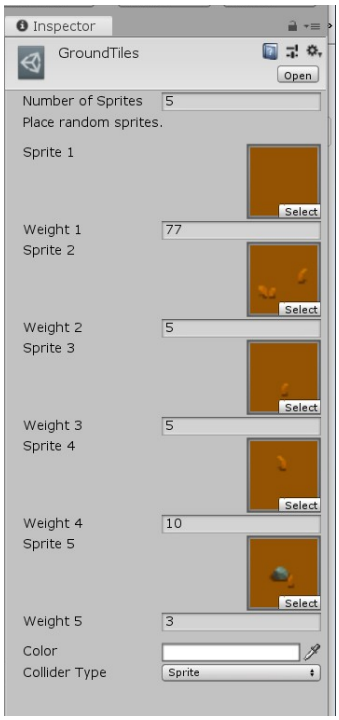


Figura 31: Asset 2d-extras en el inspector de Unity.

4.5. Diagrama de flujo del juego

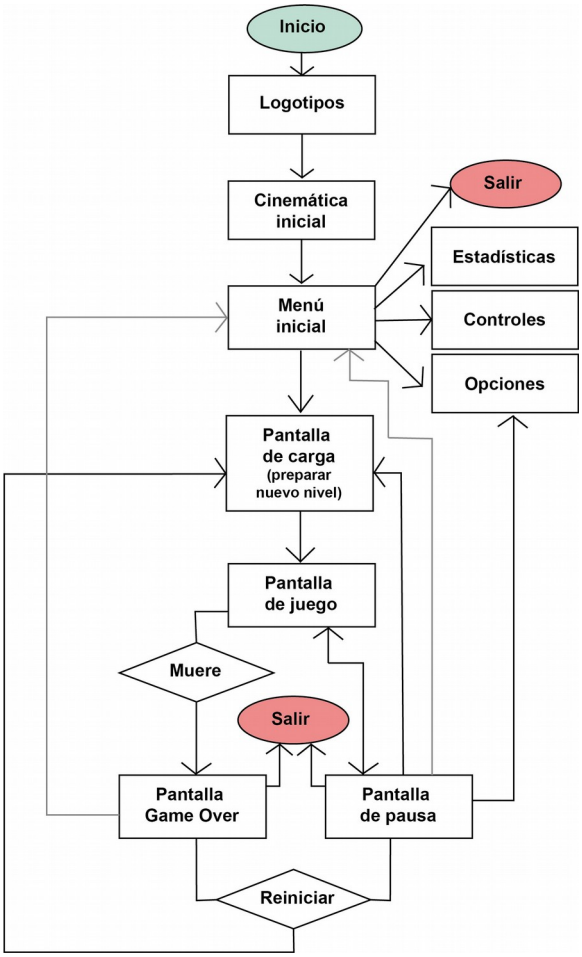


Figura 32: Diagrama de flujo de *Time Ride.*

Si bien en el diagrama se indica que después de los logotipos (logotipo de Unity y logotipo de Team Bee) se inicia la cinemática inicial, en la versión final de este proyecto no se ha podido crear dicha cinemática ya que, con el tiempo del que se disponía, hemos preferido dedicar recursos al desarrollo de las funcionalidades indispensables del juego. De todos modos, en un futuro se planea crear esta animación, que primero mostrará a Rufo en su estudio, preparando la máquina del tiempo, seguidamente se le verá viajando en su máquina del tiempo y, finalmente, al llegar a su destino, Rufo explicará que ha perdido el combustible y debe encontrarlo para volver a casa.

4.6. Funcionamiento de la IA de los enemigos

Time Ride consta de enemigos en cada sala del mapa, los cuales deben pasar por distintas fases o comportamientos desde su aparición hasta su muerte. Para ello, es necesario crear una inteligencia artificial o IA, o dicho de otra

forma, utilizar técnicas que hagan que el comportamiento de los enemigos parezca inteligente a ojos del jugador.

Al realizar la planificación inicial del proyecto, se planeó utilizar las características de NavMesh ofrecidas por Unity, que facilitan mucho la creación de inteligencia artificial, pero una vez en la etapa de desarrollo descubrimos que NavMesh solo es aplicable a proyectos 3D, y eso nos obligó a desarrollar dicha inteligencia desde cero.

Puesto que el objetivo era escribir un código con un elevado nivel de abstracción, que pudiera ser usado para la creación de los comportamientos de todos los enemigos del juego (tanto los implementados en el momento de entrega de esta memoria como los que se planean añadir en un futuro cercano), decidimos crear una máquina de estados finitos, en inglés *finite state machine* (FSM), encargada de almacenar el estado de un enemigo en un momento dado de la ejecución del juego. A continuación se listan las características más importantes de una FSM:

- Solo permite un único estado activo a la vez.
- Cuenta con un conjunto fijo de estados.
- Cuando recibe una entrada o *input*, cambia a un estado diferente. Cada estado especifica, para una entrada dada, el próximo estado que debe ser activado.

Así pues, en la FSM se indican los posibles estados de la inteligencia artificial, que en nuestro caso son los siguientes:

- ***Idle***: mientras la cámara no muestre al enemigo, este no realiza ninguna acción, está quieto. Cuando ya aparece en el plano de la cámara, se pasa al siguiente estado válido del enemigo.

- ***Patrol Walking*** y ***Patrol Flying***: en el *Patrol Walking*, el enemigo patrulla por tierra, lo que implica que al moverse debe esquivar las paredes, las piedras, el agua y al resto de enemigos. En cambio, en el *Patrol Flying* el enemigo patrulla por el aire, por lo que al moverse puede pasar sobre las piedras y el agua, si bien debe esquivar las paredes y al resto de enemigos.

  Para lograr estos comportamientos, se utilizan los *raycast* de Unity, rayos que dan información del elemento con el que colisionan. Una característica importante de estos es que solo colisionan con los objetos que se encuentran en las capas que indicamos, y esto es lo que nos permite indicar qué elementos deben impedir el movimiento de los enemigos.

  Respecto al proceso para realizar el movimiento, en ambos estados es el siguiente: primero se escoge una dirección al azar (izquierda, derecha, arriba o abajo), seguidamente se comprueba si en esa dirección hay elementos a esquivar cerca, y si los hay, se cambia la dirección. Este proceso sigue hasta que se encuentra una dirección válida o hasta que se llega a los 15 intentos, en cuyo caso la dirección debe ser cero. A continuación, se calcula la posición de destino a partir

de la dirección dada, y se mueve al enemigo en esa dirección hasta que llega al destino. Cuando esto sucede, el proceso empieza de nuevo.

Existen dos diferencias relevantes entre ambos estados: por un lado, las capas examinadas por los *raycast* no son las mismas, ya que los enemigos voladores no deben tener en cuenta las colisiones con las capas que contienen el agua y las piedras; por otro lado, los enemigos terrestres generan 3 rayos ligeramente separados entre ellos para comprobar si la dirección está disponible, mientras que los enemigos voladores solo generan uno, ya que consideramos que no supone un problema si el ala de un enemigo pasa por encima de otro enemigo.

En cuanto a la condición para pasar al próximo estado, en ambos estados existe en todo momento un *raycast* que solo tiene en cuenta la capa que contiene las paredes y la capa que contiene al protagonista, y cuya longitud es equivalente al rango de detección del enemigo. Así, cuando el rayo colisione con un objeto que no pertenezca a la capa de paredes significará que el enemigo ya ve a Rufo, por lo que se informará a la máquina de estados de que debe avanzar al próximo estado.

- ***Follow Walking*** y ***Follow Flying***: en este estado se genera un *raycast* que va desde el enemigo al personaje, y si no hay obstáculos en el camino (piedras, agua, paredes u otros enemigos), la dirección de este rayo será la dirección en que avanzará el enemigo. Por el contrario, si esta dirección no es válida el enemigo se moverá a la derecha, a la izquierda, arriba o abajo en función de la posición del protagonista. Así, por ejemplo si Rufo está más a la derecha que el enemigo, este se moverá hacia la derecha si ningún obstáculo se lo impide. Si esta dirección tampoco está disponible, se intentará ir hacia arriba o hacia abajo, según donde se encuentre Rufo en el eje Y. Si la dirección vertical que tendría que escogerse también está bloqueada, entonces el enemigo deberá intentar ir hacia la izquierda y, si tampoco puede, hacia abajo. Una vez la dirección ha sido determinada, se calcula el destino y el enemigo avanza en esa dirección hasta que llega a él y el proceso empieza de nuevo.

  Las diferencias entre ambos estados son las mismas que las expuestas en los estados *Patrol Walking* y *Patrol Flying*; esto es, el estado *Follow Flying* genera menos rayos de comprobación, y dichos rayos no tienen en cuenta las capas que contienen el agua y las piedras.

  Cabe comentar que, a diferencia de los estados de patrulla, que hacen que la máquina entre en el próximo estado válido, los estados de seguimiento pueden indicar a la máquina que avance al próximo estado de la lista (si el *raycast* generado constantemente por los estados *Patrol,* cuya longitud es igual al rango de ataque, colisiona con el enemigo) o bien que retroceda al estado anterior (si la distancia entre el jugador y el enemigo es mayor o igual al rango de abandono de ese enemigo).

- ***Range Attack***: en primer lugar se comprueba que la distancia entre enemigo y jugador sea inferior o igual al rango de ataque y que el jugador no tenga activada su habilidad especial. Si estas condiciones se

cumplen, se realiza un disparo. Por el contrario, si la condición de la distancia no se cumple, la máquina vuelve al estado anterior.

- ***Melee Walking Attack*** y ***Melee Flying Attack***: se comprueba si el personaje puede avanzar en dirección al jugador, y si no puede porque algún obstáculo se lo impide, se indica que la dirección deberá ser nula. A continuación, se calcula el destino en función de la dirección, y se mueve al enemigo en esa dirección. Al llegar al destino, el procedimiento se repite.

  En cuanto al cambio de estado, si la distancia entre el enemigo y el jugador es superior a la distancia de ataque, la máquina que debe volver al estado anterior.

- ***Boss Intro***: si el jugador está en el rango de detección del enemigo y no hay paredes que se interpongan entre ellos, se muestra la barra de vida de jefe final, se reproduce una animación introductoria y se pasa al próximo estado.

- ***Boss Shoot***: el enemigo dispara una vez y avisa a la máquina para que avance al próximo estado

- ***Underground Move***: el enemigo está un tiempo en la superficie, luego se esconde, calcula su próxima posición de forma aleatoria y, por último, aparece de nuevo. En ese momento, si tiene menos de la mitad de la vida, se pasa a la próxima fase. En caso contrario, se vuelve a la fase anterior.

- ***Generate Childs***: cada cierto tiempo se genera un número de hijos determinado en posiciones aleatorias disponibles de la sala. Justo en el momento en que aparecen los hijos, el enemigo padre desaparece, y vuelve a aparecer cuando falta la mitad del tiempo para que nazca la próxima ronda de hijos.

A fin de que la máquina sepa cuáles de estos estados son válidos para un enemigo concreto, añadimos el script *FiniteStateMachine* a cada tipo de enemigo (es decir, a cada *prefab* de enemigo), y en el inspector indicamos cuáles son los estados posibles y su orden lógico de aparición.

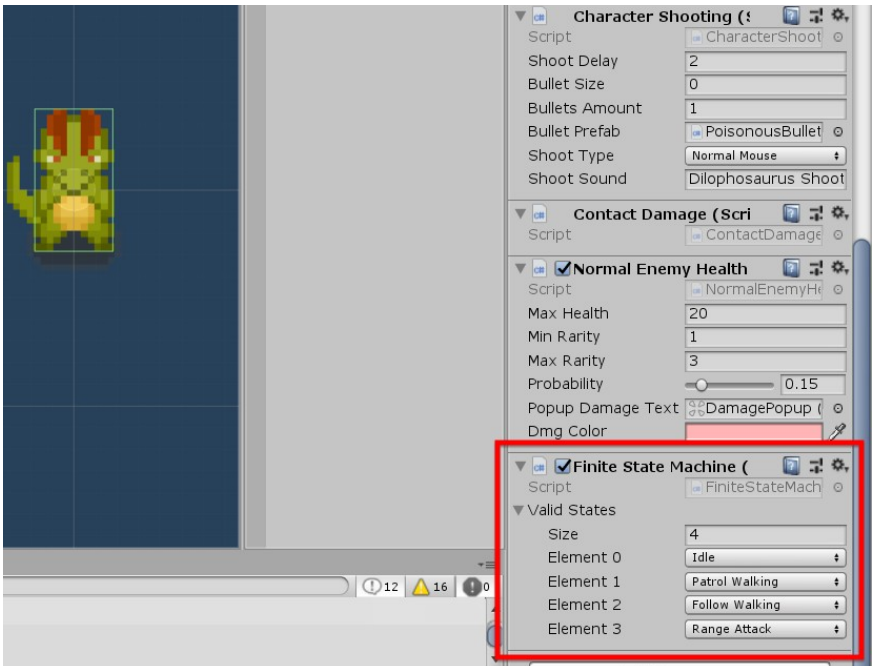


Figura 33: FSM del dilophosaurus en el inspector de Unity.

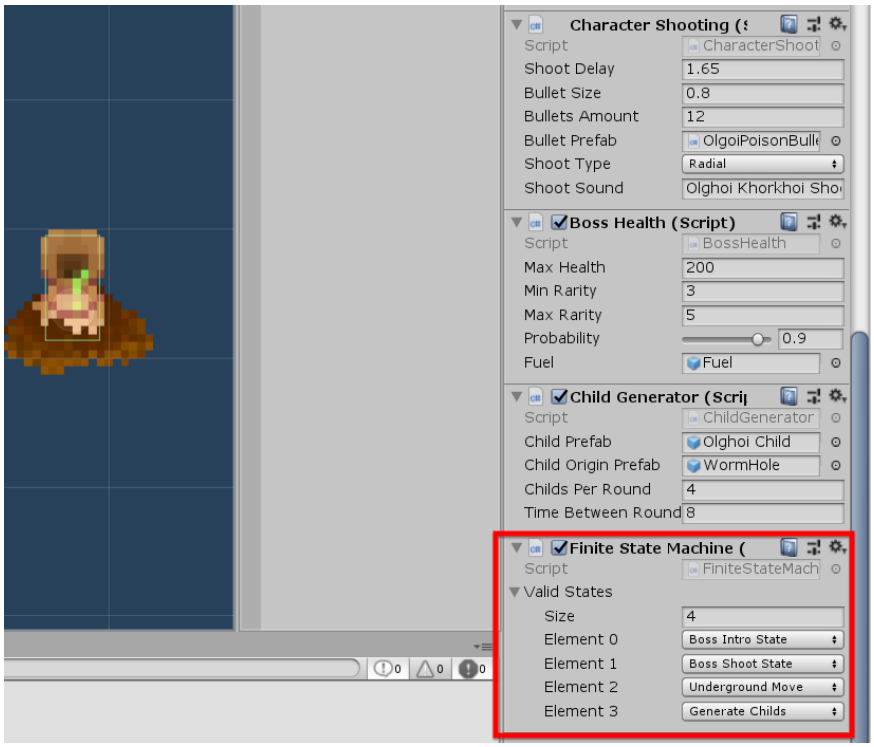


Figura 34: FSM del Olghoï en el inspector de Unity.

Cabe destacar que se ha escogido la creación de estados diferentes para personajes de tierra y de aire (por ejemplo, *Patrol Walking* y *Patrol Flying*) porque hemos considerado que había demasiadas diferencias entre ambos (no solo las ya mencionadas en este apartado, sino también diferencias respecto a la forma en que se animan, ya que los personajes voladores siguen “volando” aunque no cambie su posición, mientras que los terrestres deben quedarse

quietos si no se mueven), lo que terminaría derivando en un código con muchos booleanos y condicionales si se decidiese crear solo un estado para los dos.

5. Diseño de niveles

Salas

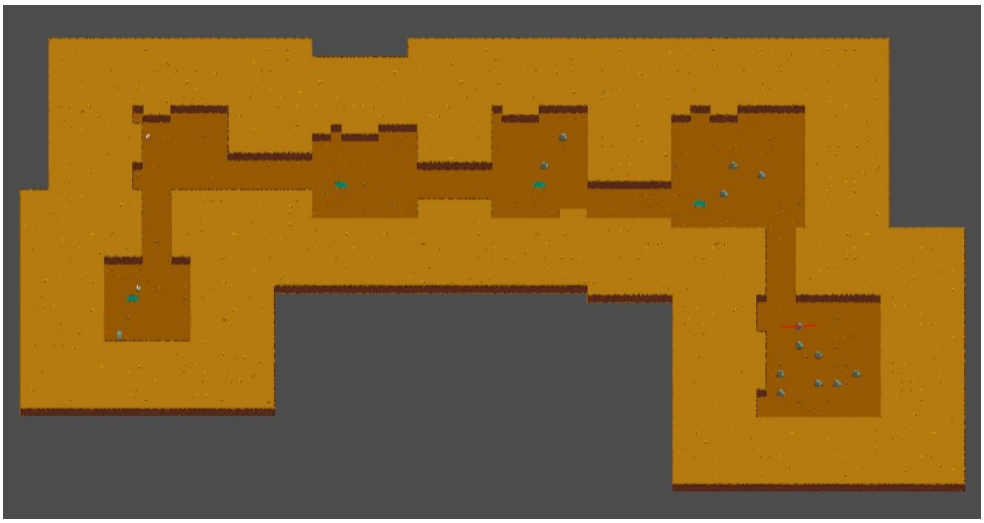


Figura 35: Mapa del primer nivel, generado de forma procedimental.

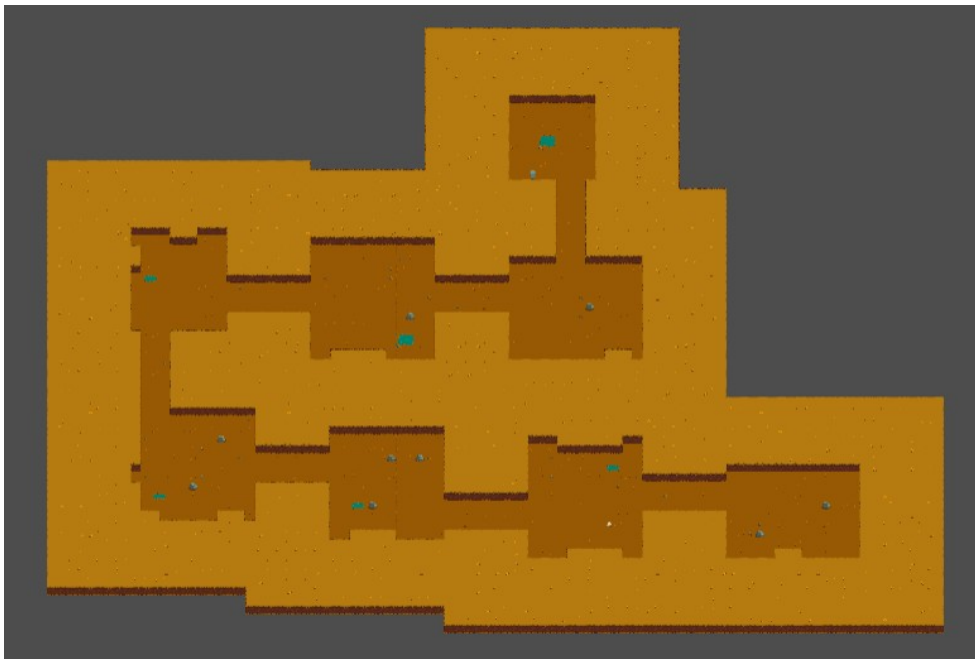


Figura 36: Otro posible mapa del primer nivel.

Una de las características principales de los videojuegos *roguelikes* es la generación del terreno de forma procedimental. En *Time Ride,* el mapa se crea al inicio de cada nivel, y las variables aleatorias que se utilizan para ello son las siguientes:

- Número de salas: un nivel puede contener entre 6 y 9 salas.

- Anchura de la sala: la anchura mínima de una sala es de 11 unidades, y la anchura máxima, de 19. La primera sala, sin embargo, no utiliza una anchura aleatoria, sino que siempre tiene la anchura mínima, puesto que se trata de una sala más bien introductoria.
- Altura de la sala: la altura puede estar entre 10 y 18 unidades. La primera sala siempre tendrá la altura mínima.
- Longitud de los pasillos: las salas se comunican entre ellas mediante pasillos rectangulares, cuya longitud puede ser 7, 8 o 9. Todas las salas intermedias tienen un pasillo de entrada a la sala y un pasillo de salida.
- Anchura de los pasillos: puede ser de 5 o 6 unidades.
- Dirección de los pasillos: la dirección (Este, Oeste, Norte o Sur) de los pasillos de entrada y salida de una sala también es aleatoria, si bien se tiene en cuenta que las direcciones de ambos pasillos no pueden ser opuestas; esto es, si el pasillo de entrada va hacia el Norte, el pasillo de salida no puede ir hacia el Sur y viceversa.
- Posición de los pasillos: los pasillos con dirección Norte o Sur deciden su posición vertical de forma aleatoria, mientras que en los de dirección Este u Oeste la posición horizontal es aleatoria.
- Irregularidades de las paredes de la sala: si una pared no tiene pasillo, puede presentar irregularidades. Para saber cuando crearlas, se calcula un número al azar del 0 al 1, y si el resultado es superior a 0.85, se genera la irregularidad. La primera sala no puede contener irregularidades.
- Cantidad de obstáculos pequeños (piedras): el número máximo de obstáculos pequeños de una sala se determina mediante el siguiente cálculo: *(Altura de sala * Anchura de sala) / (Altura de sala / 1.25 + Anchura de sala /1.25).* Para cada posición de la sala, primero se calcula un número del 0 al 1 al azar, y si el número es superior a 0.965 (o 0.925 si se trata de la última sala, lo que hace que su probabilidad de contener piedras sea mayor) y aún no se ha llegado al número máximo de piedras en la sala, puede crearse una piedra en esa posición, siempre y cuando no haya ya una piedra a su izquierda, a su diagonal izquierda superior e inferior, o debajo.

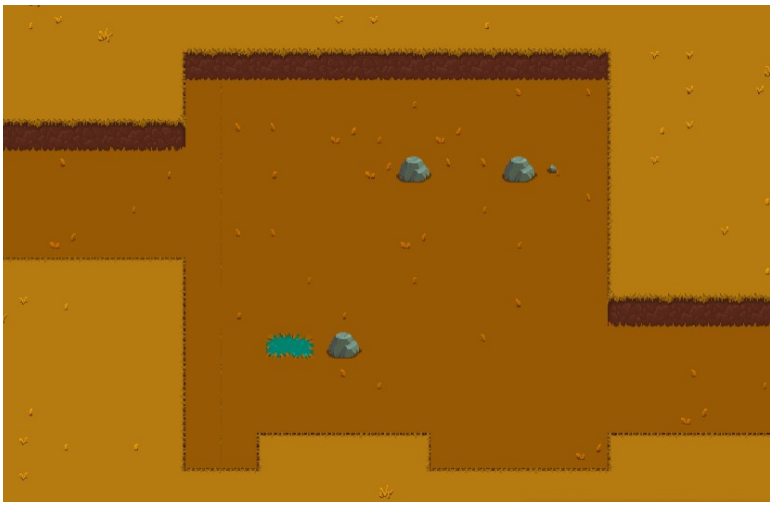


Figura 37: Sala con 3 piedras y un charco pequeño.

- Cantidad de charcos: una sala como máximo tendrá 2 charcos, excepto si se trata de la última sala, donde no habrá ninguno, puesto que el Olghoï Khorkhoï vive en zonas secas. Se pueden encontrar 2 tipos de charcos, uno de 2 *tiles* y otro de 4.

  Para que una posición sea válida para colocar un charco, no puede haber otro a su izquierda, a la izquierda de su izquierda, a sus diagonales izquierdas superior e inferior, o bien en las dos posiciones inferiores.

  Se decide si el charco a crear es grande o pequeño mediante un número al azar entre 0 y 1. Si es inferior a 0.97, no se genera charco, si es un número entre 0.97 y 0.99 el charco es pequeño, y si es superior a 0.99, el charco es grande.

  Cabe destacar que no puede haber charcos ni piedras en las posiciones que están en contacto con las paredes de la sala.

Enemigos

En el primer nivel pueden encontrarse los siguientes enemigos normales:

1) Meganeura: puede haber grupos de entre 3 a 12 meganeuras. Estos enemigos voladores pueden sobrevolar las piedras y los charcos, y su nivel de peligrosidad deriva del número de integrantes del grupo. Si el jugador se queda a solas con una meganeura y está cerca de ella, es muy probable que acelere para matarlo.

   Su salud es de 15 puntos (se requieren 3 golpes de la bala por defecto para matarla). La probabilidad de que suelte un ítem de rareza entre 1 y 3 al morir es de 0.08 entre 1.

Figura 38: Sala con tres meganeuras, dos de ellas en fase de ataque.

2) Raptor: se generan entre 1 y 3 raptors juntos. Se trata de enemigos que deben esquivar los obstáculos para poder llegar al jugador.

   Su salud es de 20 puntos (se necesitan 4 golpes de la bala por defecto para matarla). La probabilidad de que suelte un ítem de rareza entre 1 y 3 al morir es de 0.13 entre 1.

Figura 39: Sala con 2 raptors, uno de ellos en fase de seguimiento.

3) Dilophosaurus: pueden encontrarse de 1 a 3 dilophosaurus juntos. Estos enemigos terrestres deben esquivar los obstáculos para acercarse al protagonista, pero atacan a distancia.

   Su salud es de 20 puntos (se necesitan 4 golpes de la bala por defecto para matarla). La probabilidad de que suelte un ítem de rareza entre 1 y 3 al morir es de 0.18 entre 1.

Figura 40: Sala con un dilophosaurus en fase de ataque y un raptor en fase de patrulla.

En cuanto al jefe final, el Olghoï Khorkhoï, se genera en la última sala, y durante la fase de batalla se desplaza cada pocos segundos a una zona nueva de la sala en la que no haya obstáculos. Para matarle se requieren 200 puntos de daño, y la probabilidad de que suelte un ítem de rareza entre 3 y 5 al morir es de 0.9 entre 1. Asimismo, siempre deja una botella de combustible, cuya localización se calcula de forma aleatoria entre las posiciones disponibles de la última sala.

Cabe comentar que el número máximo de enemigos que puede haber en una sala se calcula a partir de la cantidad de posiciones vacías de la sala (es decir, posiciones sin piedras ni charcos), y el número mínimo es un poco más de la mitad del máximo. De todos modos, en la primera sala siempre habrá 2 enemigos, para evitar que el jugador muera justo al aparecer, y en la última sala solo se generará el jefe final, aunque este puede crear enemigos que le ayuden a vencer al jugador durante la batalla.

Elementos rompibles

Cuando se genera el mapa, también se calculan números aleatorios entre 0 y 1 que permiten determinar si una sala debe contener un huevo (resultado entre 0.89 y 0.97), dos (entre 0.97 y 1) o ninguno. La posición del huevo es un cálculo aleatorio entre todas las posiciones sin obstáculos de la sala.

También el tipo del huevo es aleatorio; existen 3 tipos de huevos (huevos con enemigos, huevos rompibles y huevos con ítems), y todos ellos presentan la misma probabilidad de aparecer.

Ítems

En un nivel pueden aparecer los ítems de ese nivel y de todos los niveles anteriores, siempre y cuando no se hayan agotada (cada ítem solo puede ser recogido un determinado número de veces).

Los huevos de ítems y las muertes enemigas pueden producir ítems. Solo los enemigos finales generarán ítems de mayor valor, es decir, de mayor rareza. Es importante tener en cuenta que, si un enemigo puede engendrar ítems de rareza entre 3 y 5, la probabilidad de que el ítem resultante sea de rareza 5 es mucho menor que de que sea de rareza 3, ya que cada nivel de rareza tiene menos probabilidades de aparecer que su anterior.

6. Manual de usuario

6.1. Requisitos técnicos del equipo

Los requerimientos técnicos mínimos del *hardware* para jugar a la versión de escritorio de *Time Ride* son los siguientes:

- **Sistema operativo**: Windows 7 SP1+
- **CPU**: Conjunto de instrucciones SSE2
- **GPU**: Tarjeta gráfica con capacidades DX10

6.2. Instrucciones del juego

Menú del juego

Figura 41: Menú de *Time Ride* sin ningún botón activado.

Al ejecutar el juego, se mostrarán los logos de Unity y del equipo de desarrollo y, a continuación aparecerá el menú del juego (imagen superior), que consta de los siguientes elementos:

1) **Botón 'Nuevo viaje'**: si se pulsa este botón, se abrirá la pantalla de carga de una nueva partida y, una vez se haya terminado de cargar, aparecerá la pantalla de juego.

2) **Botón 'Estadísticas'**: abre la ventana con las estadísticas del jugador. Para ver todas las estadísticas debe utilizarse la barra de desplazamiento vertical de la ventana.

   Para cerrar la ventana, se puede pulsar la cruz de la parte superior derecha.

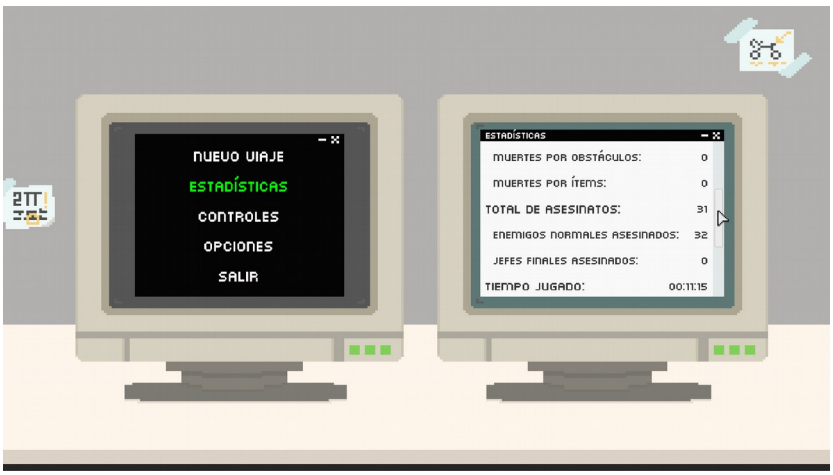


Figura 42: Menú de *Time Ride* con la ventana 'Estadísticas' abierta.

3) **Botón 'Controles'**: abre la ventana con información sobre los controles necesarios para jugar. De momento se trata de un panel únicamente informativo, no modificable.

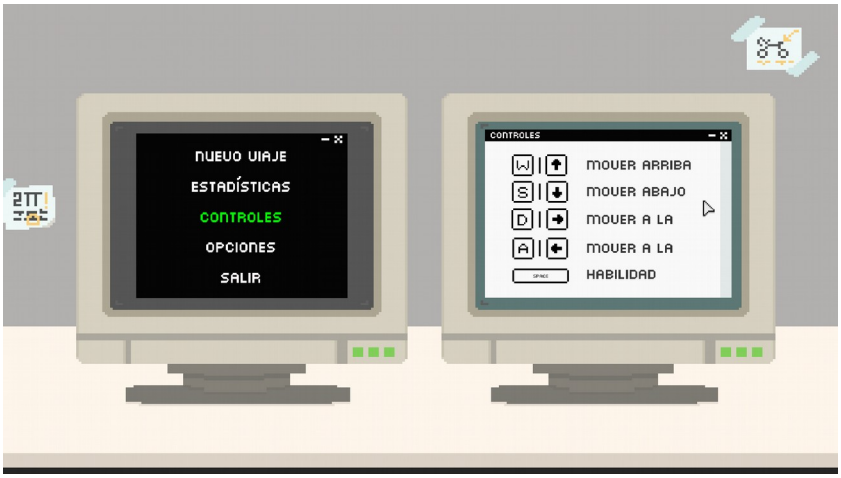


Figura 43: Menú de *Time Ride* con la ventana 'Controles' abierta.

4) **Botón 'Opciones'**: activa el panel de opciones, en el que se puede hacer lo siguiente:

- Activar o desactivar la visualización del cronómetro en la pantalla de juego.
- Activar o desactivar la visualización de las estadísticas del personaje en la pantalla de juego.
- Cambiar a pantalla completa o a modo ventana.
- Seleccionar la resolución más adecuada.
- Cambiar el idioma. De momento se encuentran disponibles las traducciones al inglés y al español.

- Cambiar el volumen de la música y de los efectos de sonido.

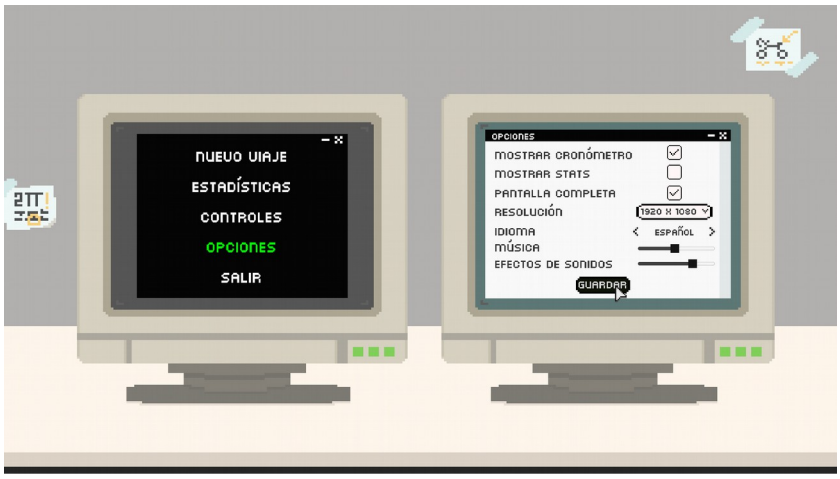


Figura 44: Menú de *Time Ride* con la ventana 'Opciones' activa.

Para aplicar los cambios, debe pulsarse el botón ‘Guardar’.

5) **Botón ‘Salir’**: permite salir del juego.

Pantalla de juego

Llegamos a esta pantalla tras pulsar el botón ‘Nuevo viaje’. Los corazones de la parte superior izquierda indican la cantidad de vidas que le quedan al personaje. Si el corazón es de color rojo, significa que está lleno, y si es gris está vacío.

Sobre los textos que se ven en pantalla, el de la parte superior derecha es el cronómetro, e indica cuánto tiempo lleva el jugador en la partida actual, y el de la parte inferior izquierda muestra las estadísticas del personaje en cada momento. Cabe destacar que para ver ambas informaciones deben haberse activado, desde el panel de ‘Opciones’, las casillas correspondientes.

Figura 45: Pantalla de juego de *Time Ride*, con cronómetro y estadísticas activados.

En cualquier momento de la partida se puede pulsar la la tecla escape para pausar el juego. Aparecerá, entonces, el panel de pausa, que puede volver a cerrarse pulsando la misma tecla.

Figura 46: Pantalla de juego con panel de pausa.

Las opciones de este panel son las siguientes:

1) **Reanudar**: desactivar el panel de pausa para seguir jugando.
2) **Reiniciar**: reiniciar la partida implica volver a la primera sala del primer nivel y perder todo el progreso de la partida actual.
3) **Opciones**: abre el panel de opciones explicado anteriormente.

Figura 47: Pantalla de juego pausada y con el panel de opciones activo.

4) **Menú**: la partida se termina, con lo que se pierde todo el progreso, y se dirige al usuario al menú inicial.
5) **Salir**: permite cerrar el juego.

En relación con los controles, el usuario puede mover al personaje con las flechas del teclado o bien las teclas A, W, S y D, y puede controlar hacia donde mira y apunta mediante la posición del ratón. Para disparar debe pulsar el botón izquierdo del ratón en la dirección deseada. Si el disparo colisiona con un huevo o enemigo, este perderá vida.

Figura 48: Pantalla de juego *Time Ride*. Rufo disparando a un enemigo.

Por otra parte, puede pulsar la barra espaciadora para activar la habilidad especial de Rufo, que consiste en parar el tiempo durante 2 segundos, lo que implica que los enemigos no pueden moverse ni disparar, mientras que el jugador puede moverse pero no disparar. Pasados esos segundos, ha habilidad se desactiva, y no estará disponible hasta el próximo nivel.

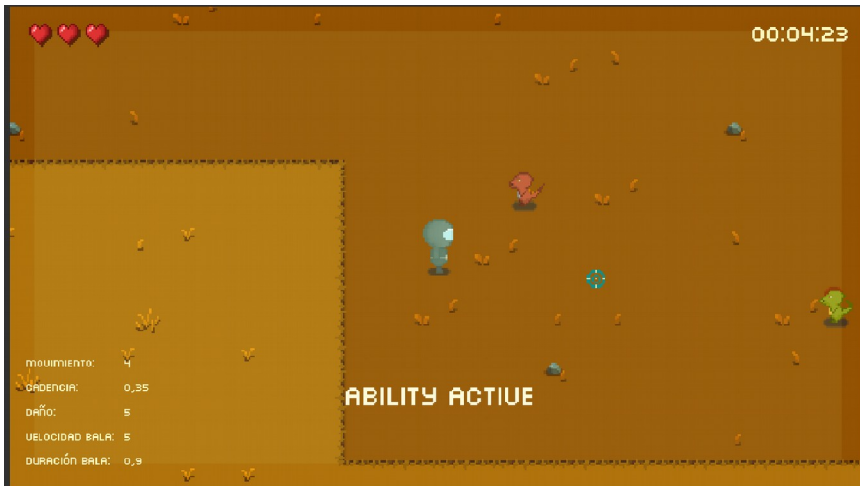


Figura 49: Pantalla de juego con la habilidad especial activada.

En cuanto al objetivo del juego, consiste en recorrer todas las salas del nivel hasta llegar a la última sala, la del jefe final, al que el jugador deberá asesinar para conseguir una botella de combustible que le permitirá subir de nivel. Es posible que, al morir, el jefe desprenda un ítem, además de la botella. En ese caso, primero debe cogerse el ítem y después la botella, puesto que de lo contrario el ítem desaparecerá.

Figura 50: Última sala del primer nivel de *Time Ride*, tras derrotar al jefe final.

Es conveniente que, antes de llegar a la última sala de un nivel, se intenten asesinar al máximo número de enemigos posible, ya que cada enemigo tiene una pequeña probabilidad de dejar un ítem al morir, lo que hará mejorar al personaje y dará facilidades de cara al enfrentamiento con el jefe final. También algunos elementos rompibles (en el caso del primer nivel, huevos) generan ítems al ser destruidos.

Figura 51: Sala con un elemento rompible: un huevo azul.

Por último, en caso de que el personaje pierda todas sus vidas, muere, y pierde así todo el progreso. Cuando esto sucede, aparece una pantalla que permite reiniciar, volver al menú inicial, o bien cerrar el juego.

Figura 52: Pantalla *Game Over* de *Time Ride*.

7. Pruebas con usuarios

Con la finalidad de comprobar la solidez del producto antes de la entrega final, durante la PEC3 se llevaron a cabo pruebas con usuarios reales.

Los objetivos principales de las pruebas fueron:

- Comprobar que el usuario pudiera jugar un gran número de partidas sin que el juego generase ningún error.
- Comprobar que la interfaz aportase información útil y clara.
- Conocer la opinión de los usuarios sobre la dificultad de los enemigos.

A continuación se expone cómo se realizaron las pruebas y cuáles fueron las conclusiones extraídas.

7.1. Perfil de los participantes

Cuando se realizan pruebas con usuarios reales, el perfil de los participantes debe ser coherente con el perfil del público objetivo final.

Sabemos que los videojuegos con elementos *roguelike*, por ser de dificultad elevada, suelen tener un público con cierta experiencia en videojuegos, ya que los usuarios sin experiencia pueden verse frustrados ante la imposibilidad de avanzar en los niveles.

Así pues, en las pruebas participaron dos usuarios, ambos aficionados a los videojuegos, y uno de ellos experimentado en *roguelikes* como *The Binding of Isaac* y *Nuclear Throne.* Cabe comentar que hubiésemos preferido realizar pruebas con un mayor número de participantes, pero no ha sido posible debido al plazo de tiempo del que se disponía para la entrega del proyecto.

7.2. Procedimiento

El primer usuario utilizó un ordenador de sobremesa con una tarjeta gráfica GTX 1070 y un procesador Intel Core i5-7600, mientras que el segundo empleó un ordenador portátil con una gráfica NVIDIA MX150 y un procesador Intel Core i5-8250U. En ambos casos se usó el sistema operativo Windows 10 64-bit.

Al inicio de las pruebas se dejó que los participantes tuvieran un primer contacto con el juego (menos de 1 minuto), y seguidamente expresaron en voz alta sus primeras impresiones.

A continuación, pedimos a los usuarios que jugaran libremente y que expresasen en voz alta cualquier idea o problema que les viniese a la cabeza. El jugador más experimentado en *roguelikes* no tuvo problema en hacerlo, y aportó una gran cantidad de *feedback* sin necesitar apenas de nuestra intervención. En el caso del segundo jugador, en cambio, formulamos preguntas para incentivar su participación (por ejemplo, “¿cuál crees que ha

sido el efecto del ítem que acabas de recoger?” o “¿sabes hacia donde dirigirte en cada momento, o te sientes perdido?”).

Al terminar, contestaron a las siguientes preguntas:

- ¿Te has sentido perdido?

- ¿Los gráficos te han confundido en alguna ocasión?

- ¿Crees que la interfaz es clara y aporta información útil?

- ¿Crees que el nivel de dificultad es adecuado?

7.3. Conclusiones

Las pruebas sirvieron para detectar los siguientes problemas:

- Las estadísticas de la pantalla de juego no se veían lo suficiente, y al cambiar alguno de sus valores, era necesaria una retroalimentación visual.
- El ratón era poco preciso en la pantalla de menú.
- La meganeura detectaba muy tarde al enemigo, y se movía muy lenta. Asimismo, el hecho de que se pudiera mover sobre las paredes molestaba a los usuarios, ya que en esa posición no podían matarla.
- El dilophosaurus resultaba muy frustrante. Las principales causas de esto eran que la cadencia entre sus disparos era demasiado corta, y que a veces disparaban antes de que el usuario lograra verlos. Además, un enemigo de tal nivel de dificultad debía dejar más ítems.
- Al ser golpeados, los enemigos no daban suficiente *feedback,* ni visual ni auditivo.
- La barra de vida del jefe final permanecía en pantalla al morir en vez de desaparecer.
- El efecto de los ítems no era evidente, se requería información al recogerlos.
- En ocasiones, los enemigos disparaban justo al iniciarse el juego, lo que hacía imposible esquivarlos.
- Los usuarios nunca llegaban a utilizar la habilidad especial, puesto que no veían ningún indicio de su existencia.
- El cambio de idioma no se aplicaba a todos los textos.
- El juego se rompía al pasar de nivel.
- La base del juego resultaba divertida, pero eran necesarios cambios, especialmente en la retroalimentación, para disminuir la frustración que provocaba.

Gracias a una realización a tiempo de las pruebas, contamos con el tiempo suficiente para solventar gran parte de estos problemas antes de la entrega final. Por lo tanto, concluimos que realizar pruebas de usuario a tiempo es imprescindible para conseguir un producto usable y agradable.

8. Conclusiones

8.1. Lecciones aprendidas

La lección aprendida más importante es la necesidad de planificar los objetivos con el mayor detalle posible desde las primeras fases. Este aprendizaje deriva de que, si bien en la PEC 1 se había preparado una planificación temporal con todos los objetivos, a medida que avanzó el proyecto nos encontramos problemas derivados de la falta de documentación de estos objetivos (en concreto estamos haciendo referencia a la creación de inteligencia artificial en Unity para proyectos 2D, que planificamos de forma optimista puesto que creíamos conocer un método sencillo que al final no se pudo implementar); habría sido necesario hacer, junto con la planificación temporal, una búsqueda de información profunda sobre los objetivos cuya consecución no conocíamos lo suficiente, lo que habría permitido una planificación más realista.

Otra lección a destacar es la necesidad de dejar más margen de tiempo en la planificación, a fin de que, si sucede un imprevisto, tengamos el tiempo suficiente para solventarlo.

También se ha aprendido sobre la importancia de realizar pruebas de usuario, las cuales permiten detectar las mejoras que es necesario implementar para conseguir un producto con un nivel de usabilidad adecuado para su público objetivo.

Por otra parte, este proyecto nos ha ayudado a profundizar en la plataforma Unity, en el lenguaje C# y en el uso de Source Tree como herramienta para manejar un repositorio Git Hub. Asimismo, nos ha permitido aprender las bases de la estética píxel art.

En cuanto a la metodología de programación, tener que escribir el código de un juego que planea seguir creciendo y que, por lo tanto, requiere de abstracción para evitar repetir el mismo código para funciones muy similares, nos ha obligado a documentarnos sobre los métodos más eficientes para programar en Unity. Así, hemos conocido los *Scriptable Objects* y el concepto de herencia, que creemos que serán muy útiles de ahora en adelante.

En definitiva, podemos decir que los objetivos generales del proyecto se han cumplido, si bien somos conscientes de que aún debemos seguir mejorando en aspectos como la eficiencia del código y el proceso de planificación inicial.

8.2. Análisis del seguimiento de la planificación y los hitos planteados.

En cuanto a los objetivos planteados en la fase de planificación inicial, se han cumplido todos excepto la implementación de guardado de partida y la creación de una cinemática inicial. La decisión de abandonar estos objetivos se tomó al revisar la planificación durante la PEC 2, y fue motivada por la incapacidad del diseñador gráfico de permanecer en el proyecto durante la situación del COVID-19. Esto me obligó a aprender los pilares básicos del píxel art para

poder llevar a cabo todas las tareas de diseño gráfico por mi propia cuenta, y para contar con el tiempo suficiente para hacerlo decidí prescindir de dos objetivos que no resultaban indispensables para la entrega.

Sobre la planificación temporal de los objetivos, han sido necesarias pequeñas modificaciones, pero se ha cumplido a grandes rasgos. Las modificaciones más importantes han afectado a los hitos de diseño gráfico y a los de programación de la inteligencia artificial. En cuanto a los cambios de temporalización de los objetivos de diseño gráfico, han sido provocados por la situación que se acaba de exponer. Y referente a los cambios en el desarrollo de la inteligencia artificial, su origen se encuentra, como ya se ha expuesto en el subapartado anterior, en el desconocimiento de las limitaciones de la funcionalidad NavMesh de Unity; al inicio del proyecto se planificó este objetivo sin realizar un proceso de documentación, puesto que creíamos conocer la solución al problema a resolver y por ello planeamos solo 13 días para el comportamiento de los enemigos. No fue hasta la PEC 3 que descubrimos que la funcionalidad que se planeaba usar solo servía para proyectos en tres dimensiones, y tuvimos que buscar una solución más compleja que hizo el objetivo se alargará toda la duración de la PEC 3. De todos modos, esta modificación no afectó a la temporalización del resto de objetivos, ya que el desarrollo de la inteligencia artificial se realizó paralelamente a estos.

8.3. Líneas de trabajo futuro

La versión actual de *Time Ride* es funcional, pero debido a las limitaciones propias de un TFG solo cuenta con un primer nivel, y esperamos poder seguir creando el resto de niveles y obtener un juego que permita al jugador viajar a través de diferentes eras históricas.

Además, es necesario realizar muchas más pruebas de usuario, ya que es muy probable que un juego con tantas variables aleatorias genere errores en contextos muy específicos y que hasta el momento no se han determinado.

Por otra parte, pretendemos seguir mejorando el arte del primer nivel; el diseño del protagonista resulta adecuado para el alcance de este proyecto inicial, pero creemos que en el futuro debemos darle más personalidad. Tampoco el diseño del primer mapa es el definitivo: es muy monótono, así que planeamos añadirle más elementos (árboles, plantas, más variedad de piedras, etc.).

Otras de las mejoras que queremos aplicar de ahora en adelante son:

- Añadir más personajes, con habilidades especiales diferentes. Cabe destacar que las características de Rufo se extraen de un *Scriptable Object,* por lo que los primeros pasos hacia este objetivo ya están hechos.
- Crear más ítems.
- Mejorar los *sprites* de los ítems actuales.
- En la versión actual el jugador puede coger un corazón aunque no esté herido, y no debería ser así.

- Crear las animaciones de los dilophosaurus y los raptors, que en la versión final de este proyecto solo se balancean de un lado a otro.
- Evitar que los enemigos puedan envenenarse entre ellos.
- Mejorar el código de la FSM y sus estados, para lograr una mayor abstracción.
- Crear la cinemática introductoria del juego.

9. Glosario

- Assets: del inglés *asset*, hace referencia a los recursos que utiliza un videojuego. Cualquier recurso que forma parte del juego (animaciones, modelos 3D, sonidos, etc.) es un asset.
- FSM: siglas del término inglés *finite state machine*, máquina de estados finitos en español. En los videojuegos, una FSM tiene la función de almacenar el estado de un elemento o personaje en un momento dado de la ejecución del juego. Cuenta con un conjunto fijo de estados, y solo permite un único estado activo a la vez. Cada estado especifica, para una entrada dada, el próximo estado que debe ser activado por la máquina.
- Píxel art: técnica de dibujo propia de los medios digitales que consiste en unir píxeles para crear una imagen. Es un arte similar al puntillismo, si bien difieren en su herramienta de creación: el puntillismo es un arte físico creado con pinturas y pinceles, mientras que el arte de píxel es creado en un ordenador.
- *Raycast*: en el motor de desarrollo Unity, la funcionalidad *Raycast* consiste en generar un rayo desde un origen especificado, con una longitud concreta hacia una dirección determinada. Este rayo permite detectar colisiones con otros elementos de la escena.
- *Sprite*: imagen asignada a un objeto del videojuego.

10. Bibliografía

Crios Devs. 2019. [en línea]. *¿Por qué elegimos Game Maker Studio?*. CriosDevs. [Fecha de consulta: 06/06/2020]. <http://criosdevs.com/es/nuestro-engine/>

Lancelot, Mátyás. 2018. [en línea]. *What is a Finite State Machine?.* Medium. [Fecha de consulta: 04/06/2020]. <https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c>

Shead, Mark. 2018. [en línea]. *Understanding State Machines.* Free Code Camp. [Fecha de consulta: 04/06/2020]. <https://www.freecodecamp.org/news/state-machines-basics-of-computer-science-d42855debc66/>

Referencias del texto

[1] **King, Alexander**. 2015. [en línea]. *The Key Design Elements of Roguelikes.* Envatotuts+. [Fecha de consulta: 06/06/2020]. <https://gamedevelopment.tutsplus.com/articles/the-key-design-elements-of-roguelikes--cms-23510>

[2] **Unity Technologies**. 2019. [en línea]. *System Requirements for Unity 2019.1.* [Fecha de consulta: 03/06/2020]. <https://docs.unity3d.com/2019.1/Documentation/Manual/system-requirements.html>

Figuras

[Fig.3] **Usuario de Wikimedia Commons *Artotransformation***. 2008. [imagen en línea]. *Rogue Screen Shot CAR.PNG.* Wikimedia Commons. [Fecha de consulta: 06/06/2020]. <https://commons.wikimedia.org/wiki/File:Rogue_Screen_Shot_CAR.PNG>

[Fig.4] *The Binding of Isaac: Afterbirth+* (2015). Edmund McMillen.

[Fig.5] *Spelunky* (2008). Derek Yu.

[Fig.6] *Nuclear Throne* (2015). Vlambeer.

[Fig.7] **Usuario de Open Game Art *PixElthen***. 2018. [imagen en línea]. *Pixel Art Monster Worm Sprites.* Open Game Art. [Fecha de consulta: 03/04/2020]. <https://opengameart.org/content/pixel-art-monster-worm-sprites>

[Fig.8] **Usuario de Tenor *Cybear***. 2020. [imagen en línea]. *Computer Pixel Art GIF.* Tenor. [Fecha de consulta: 20/05/2020].

<https://tenor.com/es/ver/computer-pixel-art-monitors-computer-set-up-blinking-cursor-gif-17153742>

[Fig.9] *Moonlighter* (2018). Digital Sun Games.

[Fig.10] *Sparklite* (2019). Red Blue Games.

[Fig.14] **Usuario de OpenGameArt *helpcomputer*.** 2014. [imagen en línea]. *Explosions.* OpenGameArt.org. [Fecha de consulta: 05/06/2020]. <https://opengameart.org/content/explosions-2>

[Fig.15] **Usuario de Itch.io *Chesire*.** [imagen en línea]. *Insect Enemy Assets.* Itch.io. [Fecha de consulta: 05/06/2020]. <https://jeevo.itch.io/insect-enemies>