Katergáris' Labyrinth

Carlos Martínez Gómez
Máster universitario de Diseño y Programación de Videojuegos
Diseño de Experiencias de Juego

Helio Tejedor Navarro
Joan Arnedo Moreno

02/01/2022

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Katergáris' Labyrinth*
Nombre del autor:	*Carlos Martínez Gómez*
Nombre del consultor/a:	*Helio Tejedor Navarro*
Nombre del PRA:	*Joan Arnedo Moreno*
Fecha de entrega (mm/aaaa):	01/2022
Titulación:	*Máster universitario de Diseño y Programación de Videojuegos*
Área del Trabajo Final:	*Diseño de Experiencias de Juego*
Idioma del trabajo:	*Español*
Palabras clave	*Videojuego, roguelite, mazmorras*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo.*

Katergáris' Labyrinth es un juego de mazmorras 2D con ambientación en la Grecia clásica mitológica, donde el pequeño Katergáris deberá adentrarse en las entrañas del archiconocido Laberinto del Minotauro para sobrevivir a sus cámaras generadas aleatoriamente y salir con todo el botín posible sin poder defenderse.

Este juego constará principalmente de mecánicas del género *roguelite* como *The Binding of Isaac* (Nicalis, 2011), *Moonlighter* (Digital Sun, 2018) o *Pokémon Mundo Misterioso: Equipo de Rescate DX* (Spike Chunsoft, 2020), con elementos de juegos de sigilo como *Metal Gear Solid V: The Phantom Pain* (Kojima Productions, 2015) o *Sly 3: Honor Entre Ladrones* (Sucker Punch Productions, 2005), sumando a todo ello la inhabilidad del jugador para atacar a los enemigos, tomando inspiración de, por ejemplo, *Amnesia: The Dark Descent* (Frictional Games, 2010).

Todas las mecánicas y posibles estrategias que se brindan al jugador giran en torno a la no agresión y el sigilo: esconderse dentro de cofres, posibilidad de moverse a hurtadillas para hacer menos ruido o las habilidades que irá desbloqueando (congelación, parálisis, invisibilidad, generar ruido, etc). Todo ello anima al jugador a tomar un acercamiento sigiloso y premeditado a cada piso de la mazmorra, incentivando a reconocer primero el terreno para memorizar posibles escondites, rutas de escape, comportamiento de los enemigos, ubicación del botín o las trampas… Dentro del abanico de dinámicas de la misma índole, el jugador podrá elegir y ejecutar las que mejor se adapten a cada situación, más le diviertan o hagan más eficiente su aventura por el laberinto.

Abstract (in English, 250 words or less):

Katergáris' Labyrinth is a 2D dungeon crawler videogame set in mithologycal Greece, where little Katergáris will have to go deep into the bowels of the well-known minotaur's labyrinth to survive his randomly generated chambers and flee with as much loot as posible without being able to defend himself.

This game mainly consists of mechanics of the roguelite genre seen in The Binding of Isaac (Nicalis, 2011), Moonlighter (Digital Sun, 2018) or Pokémon Mystery Dungeon: Rescue Team DX (Spike Chunsoft, 2020), with elements taken from stealth games like Metal Gear Solid V: The Phanton Pain (Kojima Productions, 2015) or Sly 3: Honor Among Thieves (Sucker Punch Productions, 2005), adding up player's inability to attack enemies, taking inspiration from, for example, Amnesia: The Dark Descent (Frictional Games, 2010).

All the mechanics and possible strategies offered to the player revolve around non-agression and stealth: hiding inside chests, sneaking around to make less noise or skills to unlock (like freezing, paralysis, invisibility, generating noise, etc). All of this encourages the player to take an stealthy and premeditated approach to each floor of the dungeon, encouraging first to recognise the terrain to memorize possible hiding places, escape routes, enemy behaviour, the location of loot or traps… Within the range of dynamics of the same nature, the player will be able to choose and execute the ones that best suit each situation, the ones that they find more fun or that make their adventure through the maze more efficient.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

Actualmente se pueden encontrar en el mercado muchos videojuegos del género *roguelite* con un gran éxito entre su público objetivo como *The Binding of Isaac* (Nicalis, 2011), *Moonlighter* (Digital Sun, 2018) o *Pokémon Mundo Misterioso: Equipo de Rescate DX* (Spike Chunsoft, 2020). También podemos encontrar un gran elenco de juegos dedicados al sigilo, como *Metal Gear Solid V: The Phantom Pain* (Kojima Productions, 2015) o *Sly 3: Honor Entre Ladrones* (Sucker Punch Productions, 2005). A su vez, podemos encontrar algunos videojuegos basados en la mitología griega como *Immortals Fenyx Rising* (Ubisoft, 2020) o *Hades* (Supergiant Games, 2018). E incluso podemos encontrar algún ejemplo de videojuego que no permite al jugador atacar a enemigos, como *Amnesia: The Dark Descent* (Frictional Games, 2010). Para este proyecto se decidió implementar todos estos elementos simultáneamente de manera que se interconectasen coherentemente, siendo esto algo bastante original, dado que no hay ningún *roguelite* que mezcle elementos de sigilo y la no agresión.

Figura 1.1 Pokémon Mundo Misterioso: Equipo de Rescate DX

1.2 Objetivos del Trabajo

El objetivo principal del trabajo era conseguir desarrollar lo que se llama “Producto Mínimo Viable” (Minimum Viable Product) con los apartados mínimos

para tener una imagen relativamente fiel de la idea tras el videojuego completo y determinar así si realmente se podría considerar un videojuego apto para la comercialización o no. Por ello, se decidió que los puntos mínimos a desarrollar fuesen:

- Generación de niveles de manera procedural y aleatoria.
- Sistema de vida del jugador.
- Sistema de sigilo.
- IA de cuatro tipos de enemigos (el minotauro, un enemigo grande, un enemigo pequeño y uno con ataques a distancia)
- Jefe final de la mazmorra.
- *Hub* central básico.
- Sistema de menús simplificado.
- Los *sprites* mínimos para el diseño del primer “bioma” de niveles.
- Animaciones y sprites sencillos para objetos, enemigos y jugador.
- Mínimo indispensable de efectos de sonido y música.
- El inicio de la historia del juego.

Con estos elementos, y todo enfocado solamente a la versión de PC, se consideró que había carga de trabajo suficiente para terminar con las bases del juego implementadas, aunque sin mucho contenido, mostrando claramente si merecería la pena continuar con este proyecto para comercializarlo.

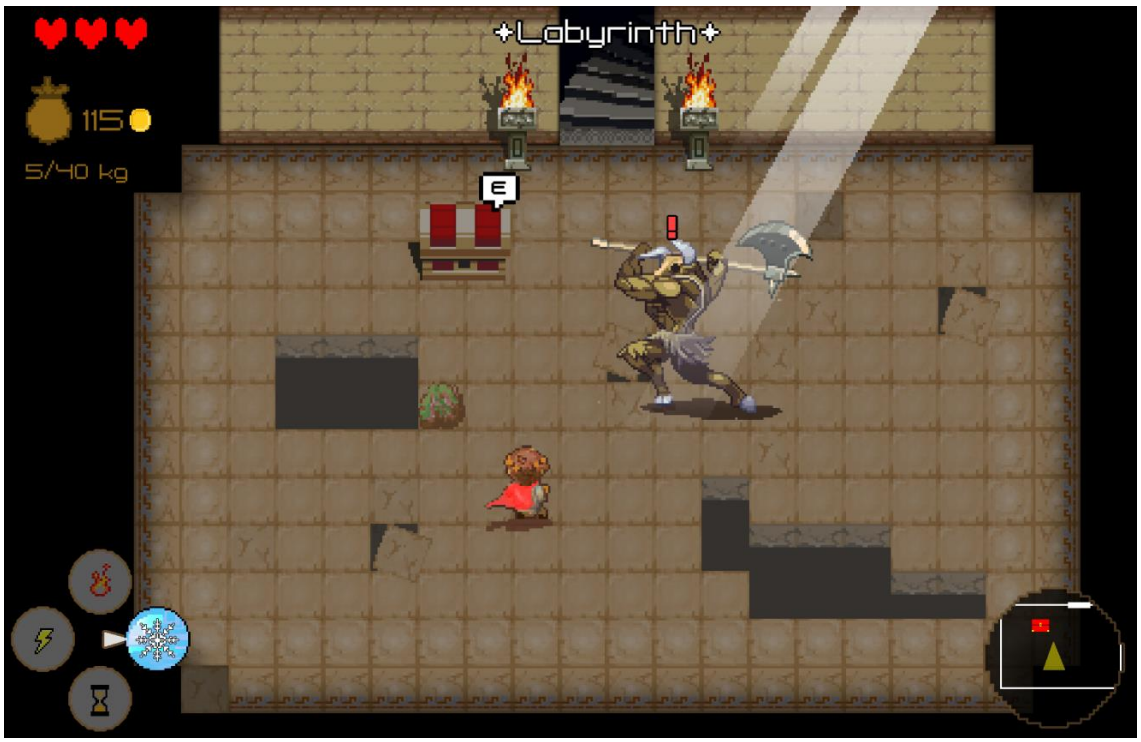

Figura 1.2 Concept Art de Katergáris’ Labyrinth

1.3 Enfoque y método seguido

Para el desarrollo del videojuego, se decidió usar la metodología de trabajo “agile”, utilizada previamente en anteriores trabajos. Esta metodología de trabajo consiste en dividir el desarrollo en distintos periodos más o menos de la misma duración denominados “*sprints*”. En cada uno de los *sprints* se pasa por todas las fases típicas del desarrollo software (diseño, desarrollo, implementación, testeo, revisión, etc.) y se termina con un prototipo funcional para entregar al cliente. Esta forma de trabajo se implementa a la perfección con la evaluación mediante PECs que se ha llevado en esta asignatura, en la que en cada entrega hay que reiterar sobre el proyecto entregado en la anterior. Aunque teóricamente fuese una duración algo larga para un sprint, cada uno de ellos se correspondería al mes que hay entre cada entrega. De esta manera, desde el principio se trabajó con la idea en mente de tener un prototipo funcional al mes, enfocando de esta manera el proyecto de una forma más directa y siendo más sencillo ver los resultados y diferencias entre los prototipos de cada uno de los *sprints*.

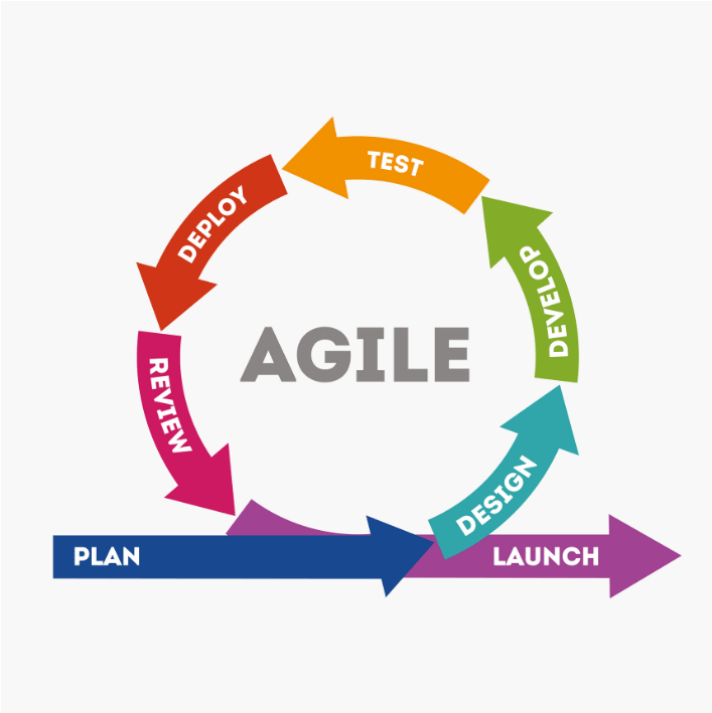

Figura 1.3 Metodología Agile. Imagen de cognodata.com

1.4 Planificación del Trabajo

Como se ha mencionado en el apartado anterior, los puntos clave del desarrollo de este videojuego dependían de las entregas y *sprints*, por lo que se dividieron los objetivos en función de dichas entregas y sus plazos, siempre con la vista puesta en tener un prototipo funcional al final de cada una de ellos.

<u>Entrega de la versión parcial, 31 de octubre de 2021 (4 semanas):</u>

- Planificación del proyecto de Unity.

- Generación procedural aleatoria de los primeros pisos del laberinto.
- *Placeholder* del personaje.
- Movimiento del personaje.
- Sistema de vida del personaje.
- *Placeholder* de los enemigos.
- IA básica de los enemigos.
- "*Blockout*" del hub de la isla.
- Sistema de menús básico a modo de *placeholder*.
- Primeras ideas para objetos, armaduras y habilidades.
- Interfaz básica.

Entrega de la versión jugable, 5 de diciembre de 2021 (5 semanas):

- Generación de niveles más compleja (objetos, cofres, enemigos…)
- Sistema de sigilo.
- Refinamiento de la IA de los enemigos.
- Refinamiento del movimiento del personaje.
- Implementación de primeros objetos, armaduras y habilidades.
- Refinamiento de los *sprites* y diseños del jugador, de los enemigos, la mazmorra y el *hub*.
- Refinamiento y funcionalidad de los menús.
- Implementación de primeros NPCs y sistema de diálogos.
- Implementación de las primeras escenas de introducción al juego y la historia.
- Diseño para el *boss*.

Entrega de la versión final, 2 de enero de 2022 (4 semanas):

- Implementación de efectos de sonido y música.
- Implementación de diálogos.
- Refinamiento de *sprites* y animaciones.
- Implementación del *boss*.
- Terminar de implementar objetos, armaduras y habilidades.
- Terminar de implementar lo que falte.

Esta fue la cuantificación de tiempo para cada apartado que se estimó al principio del desarrollo:

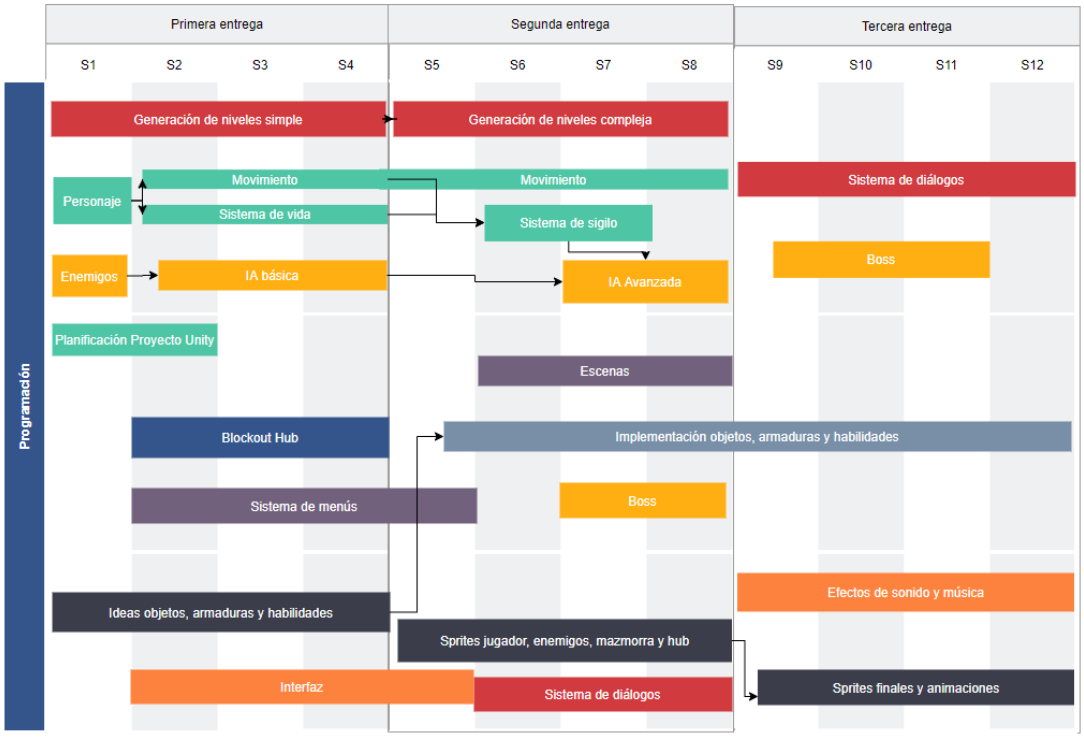

Figura 1.4 Cuantificación de tiempo para objetivos parciales.

1.5 Breve sumario de productos obtenidos

Tomando como referencia el producto mínimo viable que se expuso con anterioridad, se puede afirmar que, a pesar de no haber implementado absolutamente todos los puntos en toda su extensión, el videojuego obtenido sí es un producto que refleja con claridad la idea original sobre el proyecto. Es cierto que a medida que pasaban las entregas se iba reflejando que el proyecto abarcaba demasiado para tres meses, pero igualmente se ha ahondado en las mecánicas principales que se habían considerado como parte esencial del proyecto, como la inteligencia artificial de los enemigos, el sistema de sigilo, el sistema de tiendas, el abanico de movimientos disponibles para el jugador o el algoritmo de generación de niveles de forma aleatoria.

1.6 Breve descripción de los otros capítulos de la memoria

Durante los siguientes capítulos del documento se hará énfasis en los pilares básicos que realmente dan forma a la idea que es *Katergáris' Labyrinth*.

En primer lugar se explicará el contexto narrativo, base fundamental sobre la que se cimentan las mecánicas y la localización del juego. Una vez se tiene clara la contextualización, se puede ahondar en las bases teóricas del gameplay. ¿Qué géneros definen a *Katergáris' Labyrinth*? ¿Cómo interactúan y se complementan entre ellos? ¿Qué mecánicas se obtienen y qué aportan al

videojuego? ¿Cómo afecta esto al comportamiento del jugador y qué percibe del mismo? Partiendo de ello, se explicará cómo se han ido implementando las diferentes mecánicas del juego sin adentrarse excesivamente en el código. La generación de los niveles, el jugador, los enemigos, el jefe final, el sistema de sigilo, las tiendas, las diferentes escenas…

Finalmente, se hará un repaso de todo el proceso de desarrollo junto a un análisis crítico del mismo, dando una respuesta a los errores que se hayan producido durante estos meses y seguido de una explicación de todas las mejoras y profundizaciones que se le podrían aplicar al videojuego en caso de haber tenido más tiempo o a futuro.

2. Contexto narrativo

Katergáris' Labyrinth está ambientado en la Grecia clásica mitológica. Concretamente, en el mítico Laberinto del Minotauro de la isla de Creta, donde el poder de la bestia se dice que produce cambios extraños en la estructura del lugar y nunca dos exploradores han relatado las mismas salas y habitaciones. Tras unos cuantos años de convivir con tal edificación, los habitantes de Creta descubrieron que el laberinto atraía a muchos curiosos, héroes y turistas, por lo que decidieron convertir la isla en un gran resort turístico utilizando el Laberinto del Minotauro como gran reclamo. Así es como la isla pasó a ser un peculiar negocio muy exitoso a base de vender souvenirs, togas, camisetas, peluches, atracciones… El laberinto no solamente atrae a turistas ansiosos por llevarse un pictograma de recuerdo, sino que también vienen héroes de toda la región por los tesoros que se dice que esconde el laberinto… y por la gloria que conllevaría derrotar al Minotauro, aunque, a decir verdad, de momento ninguno ha salido con vida de las entrañas de la mazmorra.

En este contexto tan peculiar, encontramos a un niño huérfano, Katergáris, que, por recomendación de un misterioso anciano, comenzará a intentar sobrevivir por su cuenta entrando en el laberinto. Allí descubre que, a pesar de su pequeño tamaño y no poder empuñar las armas o portar las armaduras de los mayores, tiene mucha más facilidad que cualquier otro para salir con vida de él. Con su astucia y sigilo, Katergáris podrá explorar el laberinto y conseguir el botín necesario para sobrevivir día a día en Creta. Aunque su habilidad para desenvolverse dentro de la guarida de la bestia despertará más de un interesado, y puede que hasta sirva para salvaguardar el futuro de la isla cuando, al final de la historia, Teseo decida ir a Creta para poner fin a la vida del minotauro, lo que acabaría con la principal vía de ingresos del pueblo.

Figura 2.1 Teseo y el Minotauro. Obtenida de Wikipedia.org

3. Gameplay

En este capítulo se abarcarán los puntos generales que definen el *gameplay* de Katergáris' Labyrinth. De qué género es, de qué otros géneros toma mecánicas, las dinámicas que se busca generar en el jugador, cómo funcionaría a grandes rasgos su *gameplay*… Todo lo necesario para poner el juego en contexto y, en los siguientes capítulos, ahondar en los diferentes elementos que se han implementado en el proyecto.

Figura 3.1 Gameplay de Katergáris' Labyrinth.

3.1 Género

Como ya se menciona anteriormente en este documento, *Katergáris' Labyrinth* es un juego de mazmorras 2D (lo que se conoce habitualmente como *dungeon crawler*) de mazmorras generadas de manera aleatoria. Esto podría colocarlo como un juego del género *roguelike*, basados en el videojuego *Rogue* (A.I. Design, 1980). Este género de videojuegos se caracteriza por algunos elementos clave, como la generación aleatoria, el estar distribuido en forma de cuadrícula, la exploración o el desbloqueo de nuevos elementos y habilidades.

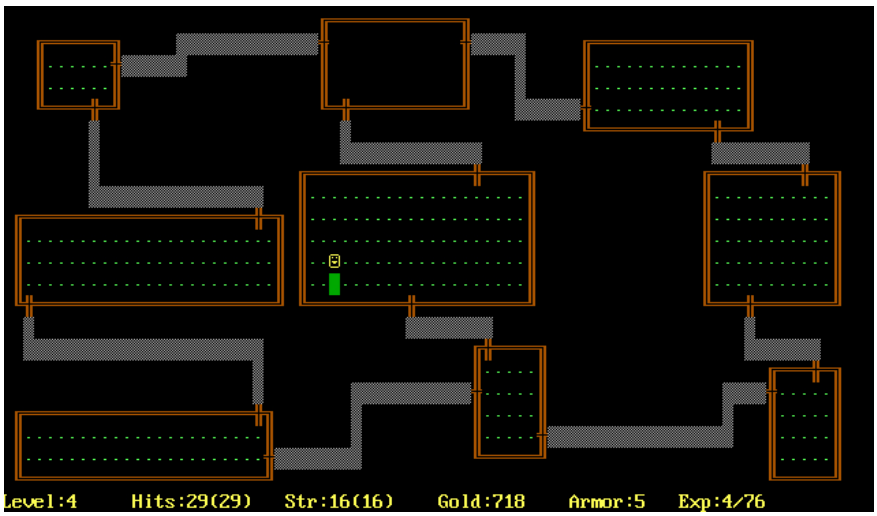

Figura 3.2 Gameplay de Rogue. Obtenida de Wikipedia.org

Sin embargo, Katergáris' Labyrinth encaja mejor en una evolución de este género: los *roguelite*, que se caracterizan por ser iguales a los *roquelike*, pero sin muerte permanente, o lo que es lo mismo, con un progreso que se mantiene persistente a pesar de completar la mazmorra o morir en ella. En este caso, Katergáris mantendrá objetos, dinero, armaduras, habilidades y progresión en la historia aunque muera en el laberinto o lo complete, por lo que encaja a la perfección en este género.

Figura 3.3 Ejemplo de roguelite: The Binding of Isaac.

Además, como se comentará con más detalle en siguientes apartados, *Katergáris' Labyrinth* también consta de mecánicas de otros géneros como el sigilo, la gestión de recursos o la supervivencia.

3.2 Mecánicas

En *Katergáris' Labyrinth* controlamos a un niño, que será nuestro *focus loci* desde una perspectiva de tercera persona isométrica. A través de él, el jugador puede interactuar con distintos elementos del juego de diferentes maneras: podemos hablar, comprar y vender en las tiendas, escondernos, huir, frenar momentáneamente a los enemigos, recolectar objetos… Todo enfocado en el *goal point* del videojuego, que sería la propia supervivencia durante las aventuras dentro del laberinto y el completar la historia mientras se exploran las diferentes capas de la mazmorra. En la interacción entre las mecánicas de ambas zonas del juego (el *hub* central y la mazmorra) encontramos el *loop* jugable del videojuego: el jugador explorará el laberinto para conseguir dinero y objetos, los cuales podrá usar en el *overworld* para comprar otros objetos y habilidades y, de esta manera, poder sobrevivir más tiempo en el laberinto para encontrar más objetos y dinero.

Aunque trataremos en detalle su implementación en los siguientes capítulos, *Katergáris' Labyrinth* se cimenta sobre tres tipos de mecánicas principales:

- Las que se engloban dentro de las reglas de funcionamiento del *hub* del juego: En esta zona se emplazaría todo el *gameplay* correspondiente a la preparación del personaje y el inventario de cara a la exploración del laberinto, como zona de recreo y exploración relajada o como lugar donde dialogar con los habitantes y avanzar con la historia.

Figura 3.4 Hub del videojuego: Cnosos.

- Las que se relacionan con el laberinto: cómo se ha generado, cuántos niveles puede explorar, cuántos tesoros o lugares para esconderse hay, cuántos enemigos y cuánto de peligrosos…

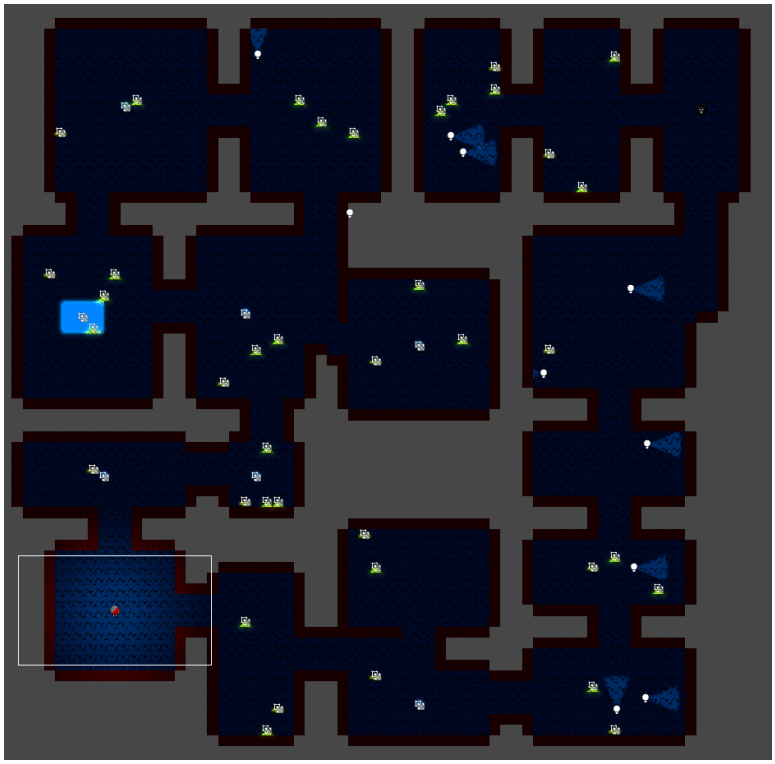

Figura 3.5 Laberinto del Minotauro.

- Y por último, las mecánicas que engloban las posibilidades del jugador, que sobre todo dependerán del equipo que lleve a la mazmorra y de la propia habilidad del jugador a la hora de leer su entorno: en qué lugares se puede esconder, qué trampas puede utilizar y en qué momentos, cuánto ruido hace al caminar según la carga que lleve, cuánto puede alcanzar su carga máxima, su velocidad de movimiento corriendo o en sigilo, qué habilidades tiene disponibles y de qué tipos…

Figura 3.6 El jugador huyendo de enemigos.

3.2 Dinámicas

La idea de la combinación de géneros tan particular comentada en los apartados anteriores, es que el jugador deba tomarse un momento antes de afrontar cada piso. Puede hacer un reconocimiento, tomar nota mentalmente de rutas de escape o lugares donde esconderse, la localización y patrones de los enemigos, caminos sin salida que pueda haber…

Al no poder atacar, todo el peso del juego recae sobre el sigilo y la capacidad del jugador para manejar o escapar de las situaciones que se encuentre. Esto, sobre todo, tendrá mucho impacto al principio del juego, cuando el jugador todavía no se ha acostumbrado a las diferentes mecánicas del mismo, desconozca los movimientos de los enemigos o no tenga suficientes habilidades, armaduras u objetos en su poder. Una vez la aventura vaya avanzando, el jugador obtendrá muchas más formas de afrontar cada una de las situaciones que se le presenten.

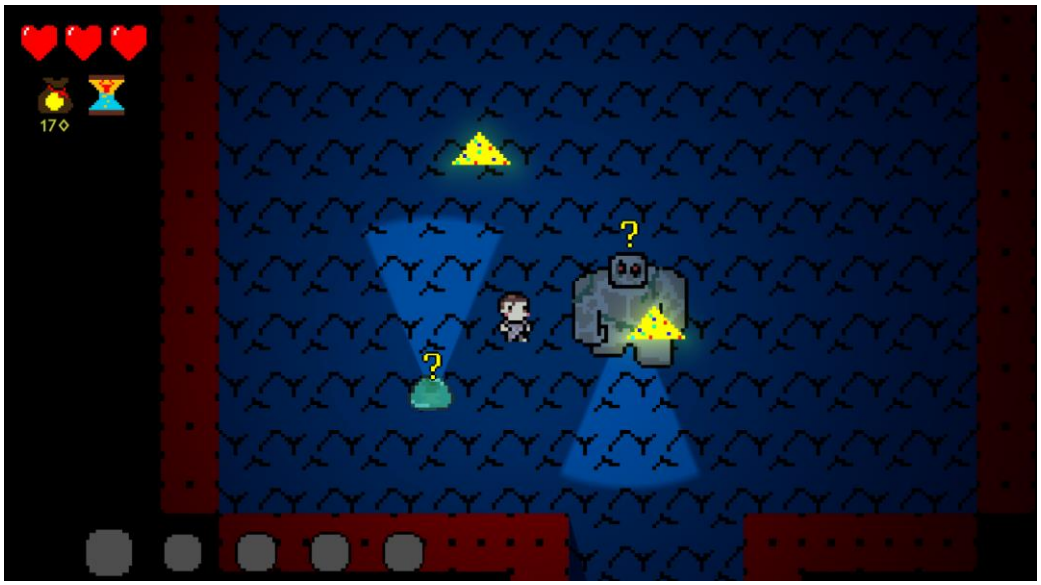

Figura 3.7 El jugador pasando a hurtadillas entre enemigos.

La idea es transmitir una sensación de indefensión al principio, pero una gran sensación de satisfacción cuando se termina haciendo un uso eficiente de los medios a su disposición, pero siempre con la sensación de alerta causada por el no poder atacar.

3.3 Estética

Aunque la estética del juego sea lo que menos se ha trabajado debido a la falta de tiempo y a la prioridad de otras áreas más definitorias del *gameplay* final, ya se puede ver en el estado actual del juego que se ha buscado un estilo gráfico de *pixelart* de 16x16 píxeles por cuadrícula muy típico de aventuras clásicas de sagas como *Pokémon* o *The Legend of Zelda*. Esto se ha elegido no solamente por ser más sencillos de modificar y crear para alguien inexperto en arte 2D y prácticamente nada de experiencia en modelado 3D y *rigging*, sino, precisamente, por apelar a la nostalgia y el sentimiento de aventura de dichas sagas y otras mencionadas anteriormente típicas de los años 1990 y 2000. Esta simplicidad brinda al jugador un marco mucho más sencillo en el que proyectarse y sumergirse en el videojuego. Evidentemente, con algo más de tiempo se podría haber hecho un trabajo mucho más profundo en la ambientación y localización de los diferentes *sprites* que conforman actualmente el juego.

Figura 3.8 Vista aérea del pueblo.

3.4 Descripción del gameplay

La sección principal de *gameplay* se desarrolla en el Laberinto del Minotauro. El objetivo del jugador será aventurarse lo más profundo que pueda, conseguir todo el botín que considere oportuno o pueda cargar, y regresar a la entrada del nivel para regresar a la ciudad de Cnosos.

Figura 3.9 El jugador recogiendo un objeto.

En el laberinto, el jugador se encontrará diferentes salas y pasillos con botín, pero que estarán custodiadas por enemigos errantes a los que no podrá hacer frente o, al menos, no podrá atacar directamente. El jugador podrá aprovechar el estar controlando a un niño para esconderse, tomar atajos por pasadizos estrechos, activar trampas o esconderse en cofres, vasijas… La

acción girará en torno al sigilo, no alrededor del combate, como en otros *roguelite* de aspecto similar. Más adelante en el juego, el jugador podrá ir desbloqueando diferentes habilidades que le permitirán avanzar más cómodamente por el laberinto, como aturdir a enemigos, ralentizarlos, paralizarlos, dejarlos desarmados, cegarlos, desplazarse haciendo menos ruido, volverse invisible, llamar la atención de los enemigos a un lugar concreto, portar más botín o artefactos… El jugador deberá estudiar su entorno y aprovecharlo de la mejor manera posible para salir con vida y con el mayor número de tesoros posible.

Figura 3.10 El jugador encontrando la sala de curación mientras huye de un enemigo.

Una vez el jugador salga airoso del laberinto, se encontrará con la bulliciosa ciudad de Cnosos. En ella, además de descansar o dialogar con los habitantes, podrá invertir lo conseguido en el laberinto para adquirir mejor equipamiento, personalizar su apariencia o pasar un buen rato en los minijuegos de las atracciones.

Figura 3.11 El jugador vendiendo los objetos recolectados en el laberinto.

4. Generación de niveles

En este capítulo se tratará uno de los pilares fundamentales del videojuego: la generación de cada piso de la mazmorra de manera aleatoria, pasando por los parámetros modificables de cada plantilla de pisos y los algoritmos empleados para la generación de las habitaciones o los pasillos.

4.1 Parámetros de generación

El primer paso de la generación de los niveles fue definir en un *Scriptable Object* los parámetros que se podrían modificar de cada piso, pudiendo de esta manera guardar plantillas ya prefijadas para cada tipo de generación y siendo muy sencillo probarlas y modificarlas en caso de que fuese necesario. Cada plantilla de parámetros tiene un total de 20 parámetros divididos en tres categorías: piso, habitaciones y pasillos. No todas tienen una implementación en el estado actual del proyecto, pero sirven de *placeholders* para más adelante.

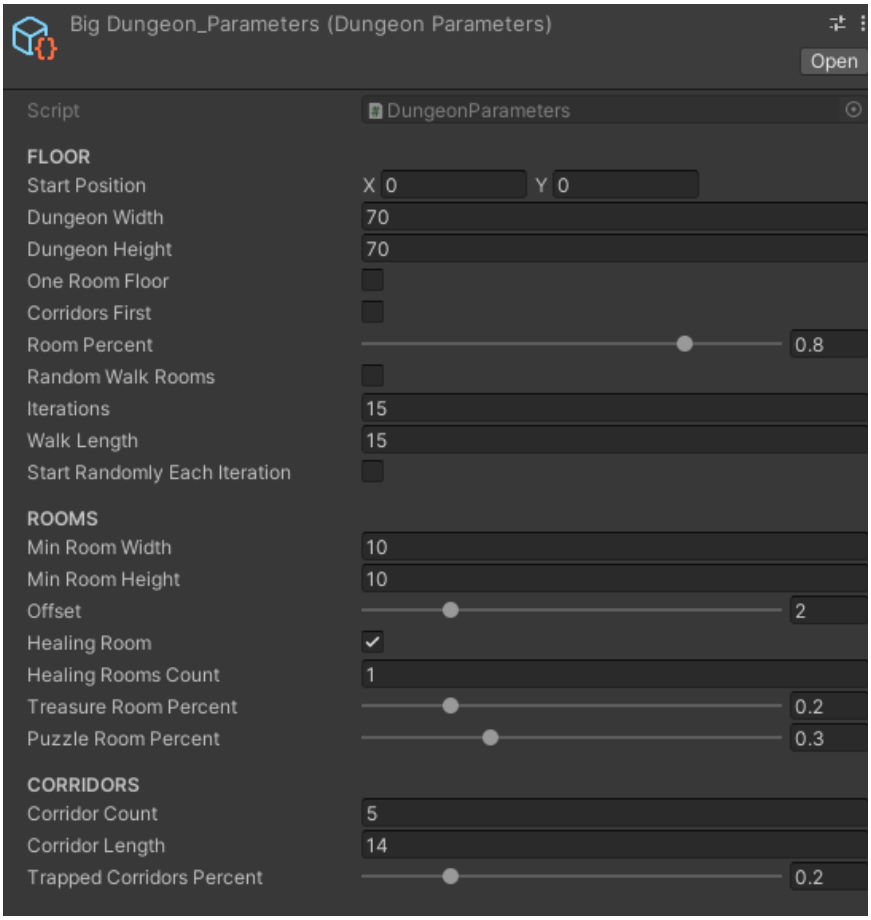

Figura 4.1 Parámetros de generación del laberinto.

La parte de definición del piso, la más importante de las tres, consta de los siguientes parámetros:

- Posición inicial: Punto desde el que se empieza a generar la mazmorra.

- Ancho y largo de la mazmorra: Definen el tamaño del piso.

- Piso de una habitación: En caso de marcarse, toda la extensión de la mazmorra será ocupada por una sola habitación, mientras que desmarcada se generarán las habitaciones unidas por pasillos que entren en las dimensiones preestablecidas.

- Pasillos primero: Este booleano afecta al algoritmo utilizado para la generación del piso. En caso de marcarse, se elegirá el algoritmo que genera en primer lugar los pasillos, y posteriormente las habitaciones. En caso contrario, primero se generan las habitaciones y posteriormente se unen entre ellas con pasillos. Esto se explicará con más detalle en la sección de los algoritmos.

- Porcentaje de habitaciones: En caso de haber marcado “Pasillos primero”, esta variable controla el porcentaje de habitaciones que se generarán al final de un pasillo.

- Habitaciones de camino aleatorio: Este booleano determina si las habitaciones se generarán usando un algoritmo de camino aleatorio o si se mantendrán como cuadrados perfectos.

- Iteraciones: En caso de haber marcado la variable anterior, este entero define el número de *walkers* que se utilizarán a la hora de crear las habitaciones.

- Longitud del camino: Este entero limita la cantidad de pasos que dará cada iteración de los *walkers* de la variable anterior.

- Empezar aleatoriamente en cada iteración: Este booleano permite que los *walkers* comiencen en posiciones aleatorias de las casillas de la habitación que se hayan creado hasta el momento. En caso de no marcarlo, los *walkers* siempre empezarán su recorrido desde el punto de origen.

Figura 4.2 Parámetros de generación del piso.

La sección de las habitaciones consta de los parámetros que controlan la generación de las mismas en caso de haber o no haber marcado la variable “Pasillos primero”:

- Anchura mínima: Como su propio nombre indica, fija la anchura mínima que debe tener una habitación. No se generarán habitaciones más pequeñas.

- Altura mínima: Igual que la variable anterior.

- *Offset*: Las casillas vacías que habrá de distancia entre las paredes de las habitaciones.

- Habitación de curación: Si el piso consta de habitaciones de curación. De momento, sin implementación.

- Cantidad de habitaciones de curación: La cantidad de habitaciones de curación que se generarán en ese piso. De momento, sin implementación.

- Porcentaje de habitaciones del tesoro: Porcentaje de habitaciones que contendrán un tesoro en ellas. De momento, sin implementación.

- Porcentaje de habitaciones de puzle: Porcentaje de habitaciones que contendrán un puzle. De momento, sin implementación.

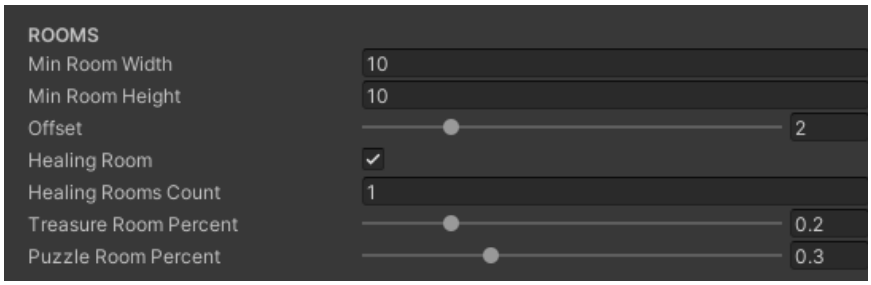

Figura 4.3 Parámetros de generación de las habitaciones.

Y por último, la sección de los pasillos utiliza sus variables en caso de haber marcado “Pasillos primero”:

- Cantidad de pasillos: El número de pasillos a generar.

- Longitud de los pasillos: La longitud que tendrán los pasillos generados.

- Porcentaje de pasillos con trampas: Como su propio nombre indica, el porcentaje de los pasillos generados que tendrán trampas. De momento, sin implementación.

Figura 4.4 Parámetros de generación de los pasillos.

De esta manera tan sencilla, una vez definidos todos estos parámetros, se pueden guardar como plantillas de tipos de pisos para ser generadas en el juego. Actualmente el juego tiene guardadas cuatro plantillas: mazmorra grande, “isla”, mazmorra pequeña y habitación pequeña.

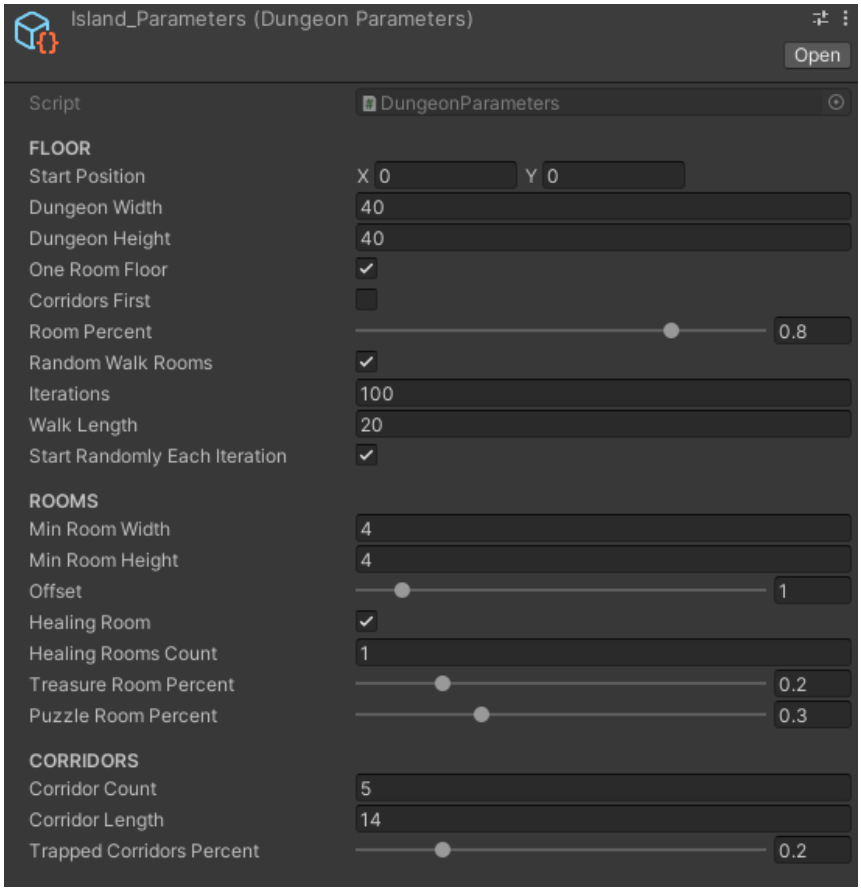

Figura 4.5 Ejemplo de parámetros de piso “isla”.

4.2 Algoritmos empleados

Una vez se han predefinido las plantillas a utilizar para los posibles pisos generados durante la estancia en la mazmorra, estos son procesados hasta por 7 *scripts* diferentes, pero en esta sección se tratarán principalmente los distintos algoritmos utilizados en ellos. Importante destacar que en todos ellos se utiliza el tipo de variable *HashSet*, dado que este conjunto descarta automáticamente las posiciones duplicadas, a diferencia de una lista.

- *Simple Random Walk*: Este algoritmo se utiliza para la creación de las habitaciones en caso de haber marcado la opción de generarlas por caminos aleatorios tanto si se han generado primero las habitaciones como si se ha decantado por un piso de una sola habitación. Este algoritmo recibe los parámetros de la plantilla y una posición inicial, desde la que los *walkers* comenzarán a "caminar" en direcciones aleatorias, añadiendo las casillas que vayan "pisando" a las casillas que conforman dicha habitación. En caso de no haber marcado la opción de que empiecen en posiciones aleatorias, se generarán habitaciones más cohesionadas y pequeñas. Si no, se generan habitaciones más grandes pero más dispersas.

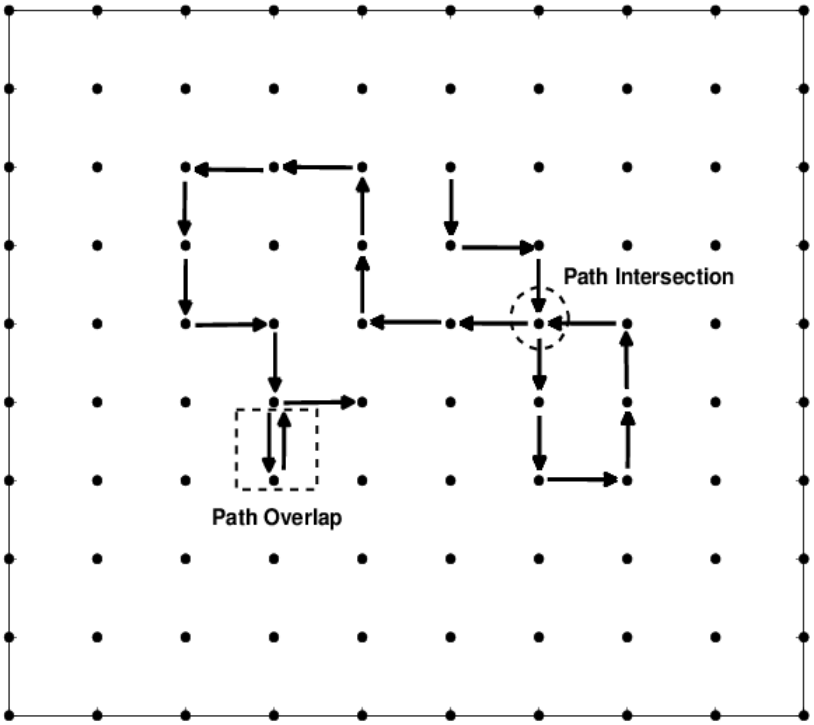

Figura 4.6 Algoritmo Simple Random Walk. Obtenida de reserchgate.net.

- Pasillos primero: En caso de haber elegido que los pasillos se generen primero, estos se crean mediante una variante del algoritmo anterior en el que se elige aleatoriamente la dirección en la que se va a crear cada pasillo, enlazando desde el punto de origen. A su vez, con cada pasillo creado guardamos los potenciales puntos de generación de habitación (los finales de cada pasillo, básicamente). Posteriormente, usando el porcentaje de habitaciones al final de un pasillo de los parámetros, se generan las habitaciones que toquen. Sin embargo, volvemos a hacer una pasada por dichos finales de pasillo para comprobar que ninguno de ellos se ha quedado como un callejón sin salida. En caso de que lo sean, generamos una habitación en dicho final. Las habitaciones se crean utilizando el algoritmo del punto anterior.

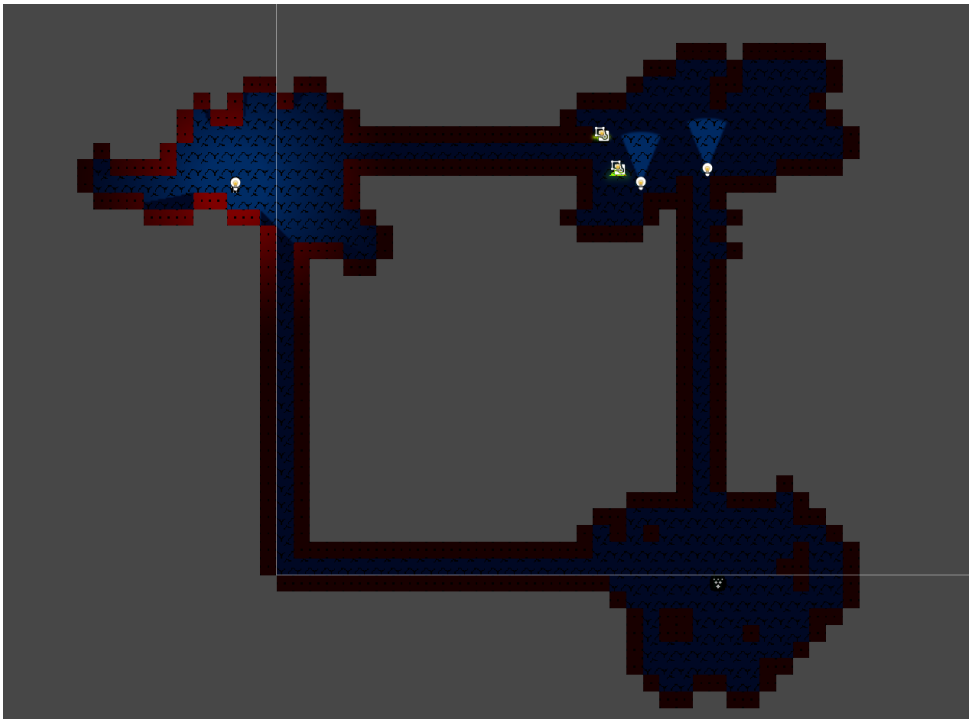

Figura 4.7 Ejemplo de generación con "Pasillos primero".

- Habitaciones primero (partición binaria): Este algoritmo se emplea en la creación de las habitaciones si no se ha marcado "Pasillos primero". En primer lugar, se delimita el área máxima de la mazmorra, marcada por el punto de origen y el tamaño de la misma. Esta superficie se guarda en una variable de tipo *BoundsInt,* que podemos imaginar como una caja. El algoritmo, en base a los parámetros que hemos fijado, irá dividiendo esta caja en cajas más pequeñas y, cuando no pueda dividirlas más por el tamaño mínimo de las habitaciones, creará una habitación para dicha caja. La partición de cada área se hace de manera aleatoria entre horizontal y vertical. Una vez marcadas las habitaciones que vamos a tener en el piso, en caso de haber marcado que se generen por camino aleatorio, se utilizaría el algoritmo explicado en el punto anterior. En caso contrario, toda la superficie rectangular pasará a ser parte de la habitación. Finalmente, se hace que todas las habitaciones estén conectadas con, como mínimo, otra habitación para que todas sean alcanzables por el jugador. Generalmente, se unen mediante un pasillo a la habitación más cercana.

Figura 4.8 Ejemplo de generación con partición binaria.

4.3 Generación de habitaciones

Una vez se han creado las habitaciones, éstas se guardan como una variable de tipo *Room*, en la que se guardan sus posiciones, el tipo de habitación, el número de enemigos que contiene, si necesita una llave (de momento sin uso), si tiene trampas (de momento sin uso), el punto central de la habitación y el *GameObject* que la contiene (se creó para aumentar la optimización al poder deshabilitar habitaciones lejanas al jugador). Ahora que todas las habitaciones se han guardado, se puede ir una por una definiéndolas con otro script.

```
public class Room
{
    private HashSet<Vector2Int> roomPositions = null;

    private int roomType = -1;

    private int numEnemies = 0;

    private bool needsKey = false;

    private bool hasTraps = false;

    private GameObject parentObject = null;
```

Figura 4.9 Variables de la classe Room.

El script llamado *RoomGenerator* se encarga de esta tarea. En primer lugar, comprueba si la lista de habitaciones contiene una sola habitación, en cuyo caso es un piso de una sola habitación y en ella se deben generar todos los elementos que toquen: el jugador, los enemigos, la salida, las termas y los tesoros. Se elige una posición aleatoria para el jugador, se coloca la salida en la posición más alejada, y el resto de elementos se generan, como mínimo, ligeramente alejados del jugador. En caso de haber más habitaciones, se elige una aleatoria como la habitación de aparición del jugador y se deja vacía. Posteriormente se elige la habitación más alejada y en su centro se genera la salida del piso. El resto de habitaciones se eligen aleatoriamente como habitaciones de enemigos, habitaciones con tesoro o una habitación con termas, que curan al jugador. Los elementos de las habitaciones se generan gracias al código de *PrefabGenerator*, que guarda una referencia a todos los posibles *prefabs* que se pueden generar mediante un índice.

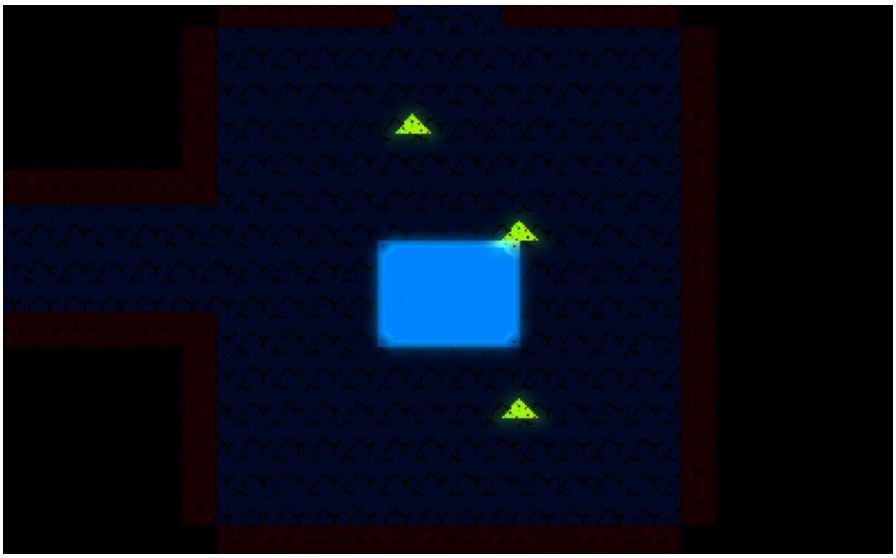

Figura 4.10 Ejemplo de habitación de termas.

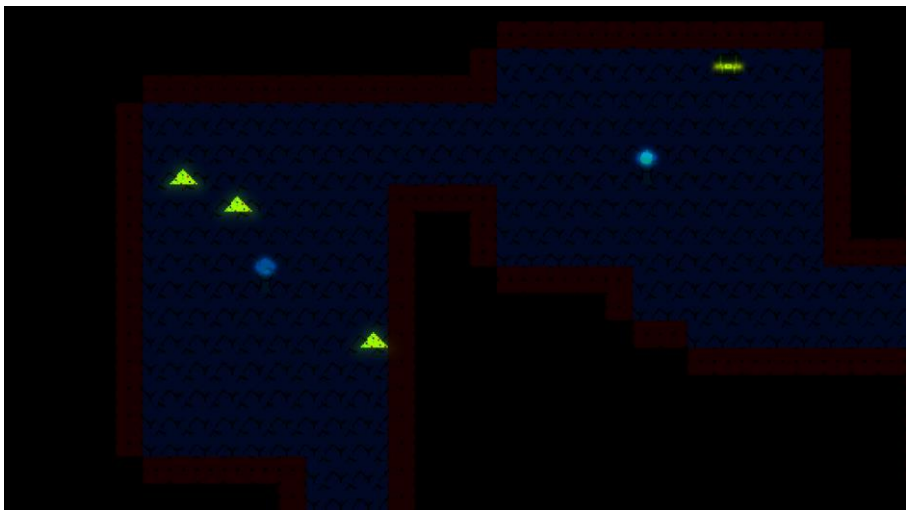

Figura 4.11 Ejemplo de habitaciones de objetos.

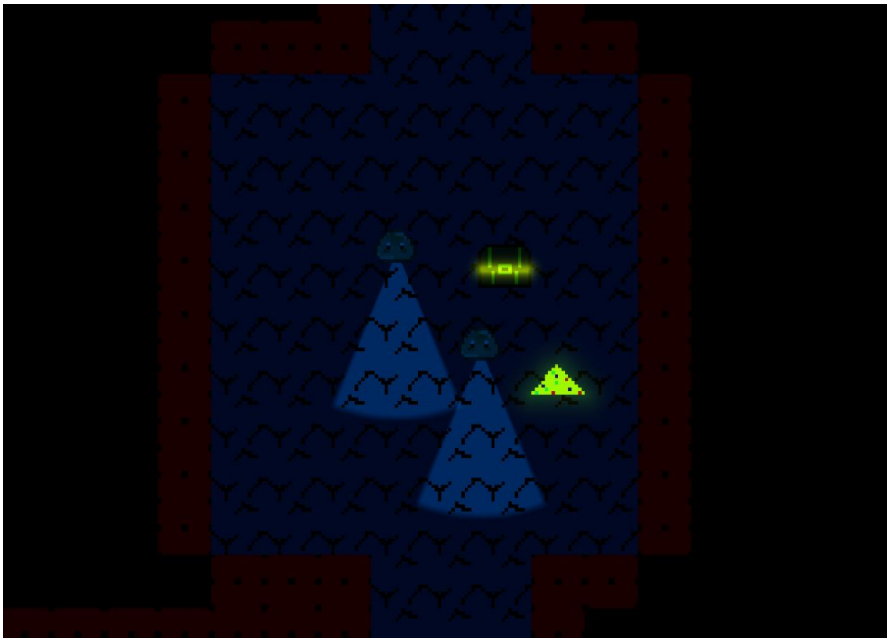

Figura 4.12 Ejemplo de habitación de enemigos.

5. Jugador

Siendo el jugador el pilar fundamental del *gameplay* del juego, había que cuidar mucho lo que era y no era capaz de hacer. Pulir el movimiento del protagonista era una prioridad al no poder atacar, así que se dedicó una considerable parte del desarrollo a pulir las variables que lo controlaban. Eso sí, hubiese faltado añadir un medidor de estamina para limitar el tiempo en el que el jugador puede correr.

Figura 5.1 Sprite del jugador.

5.1 Estadísticas

El código que controla las estadísticas del jugador puede que sea uno de los más importantes de este aspecto del juego. Desde él se definen:

- Velocidad al caminar.
- Velocidad al correr.
- Velocidad al ir agachado.
- Velocidad al rodar.
- La salud actual.
- La salud máxima.
- El tiempo de invencibilidad tras sufrir daño.
- Variables relacionadas al sigilo.

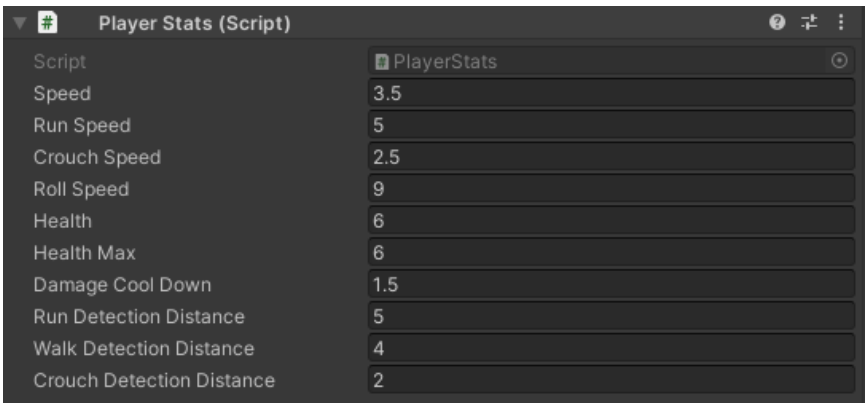

Figura 5.2 Estadísticas del jugador.

De las variables que se usan en el sistema de sigilo se hablará con detalle más adelante. Otras funciones importantes de las estadísticas son: curar al jugador, dañar al jugador y matarlo en caso de que su vida llegue a 0.

5.2 Movimiento y controles

Como se ha mencionado en la introducción de este capítulo, el movimiento era una parte esencial para conseguir el *gamefeel* de controlar a un niño viviendo una aventura. Se debía encontrar el equilibrio perfecto entre un movimiento que no se sintiese demasiado lento y pesado con un movimiento acorde a la sensación de pequeñez y vulnerabilidad que se quería transmitir. A su vez, un movimiento demasiado rápido mandaría al traste todo el sistema de sigilo, dado que el jugador podría pasarse la mazmorra corriendo sin ninguna clase de peligro. Aunque, como ya se ha mencionado en la introducción, faltaría añadir un contador para el tiempo que puede estar el jugador corriendo sin cansarse, se ha llegado a unas variables bastante acordes a lo que se quería conseguir. Decir también que en cuanto al movimiento, el protagonista se encuentra animado prácticamente al completo.

Figura 5.3 Jugador rodando para huir de unos enemigos.

En cuanto a los controles, se eligió un mapeado similar a los estándares de hoy en día en el medio para que sean sencillos de memorizar y se ajusten a los atajos que el jugador ya tuviese interiorizados. Se muestran a continuación los controles planteados tanto para PC como para consola, aunque este último sea simplemente de ejemplo. Cabe destacar que en el prototipo actual, todo el control de interfaces y menús se hace con el ratón, pero la idea sería cambiarlo para que el jugador no tenga que mover las manos de su teclado.

	PC	**Consola (PS5 como ejemplo)**
Movimiento	WASD	*Joystick* izquierdo
Rodar	Espacio	X o *Joystick* derecho
Interactuar	E	□
Habilidades	Seleccionar: 1, 2, 3, 4, etc Usar: Q	L1/R1 para cambiar + R2 Selección rápida: *pad*
Agacharse	Control	O
Sprint	Shift	R3 / Presionar *stick* derecho
Inventario	Escape	Δ

Figura 5.4 Tabla con los controles del jugador.

5.3 Objetos y armaduras

Se crearon dos *ScriptableObjects* para los objetos y armaduras, de manera que fuese muy sencillo crear uno nuevo, añadirlo al juego y editarlo. En el caso de los objetos, constan de:

- Nombre.
- Descripción.
- Efecto.
- Sprite.
- Precio.
- Peso.

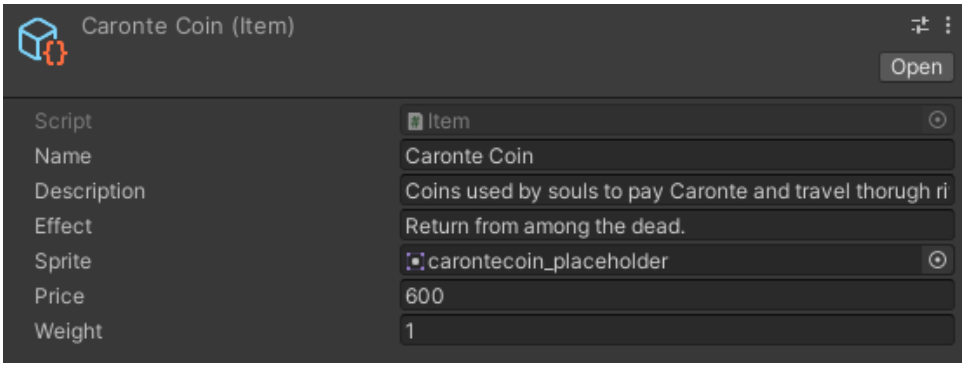

Figura 5.5 Ejemplo de objeto.

Las armaduras se definen con:

- Nombre.
- Parte (Cabeza, cuerpo, pies, brazos, etc.)
- Descripción.

- Efecto.
- Sprite.
- Precio.
- Peso.

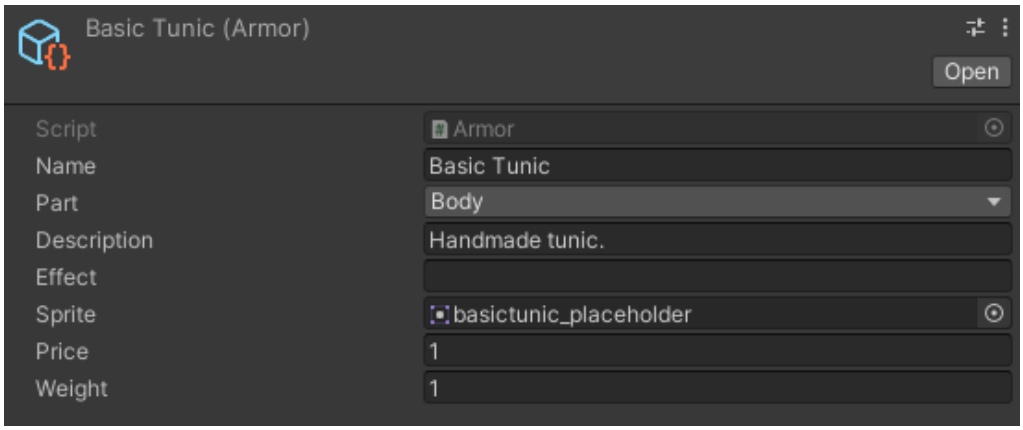

Figura 5.6 Ejemplo de armadura.

Una vez creados, es muy sencillo añadir los nuevos objetos a los pedestales que los generan durante la aventura en el laberinto o, tanto los objetos como las armaduras, en sus respectivas tiendas, de las que se hablará con detalle más adelante en el documento.

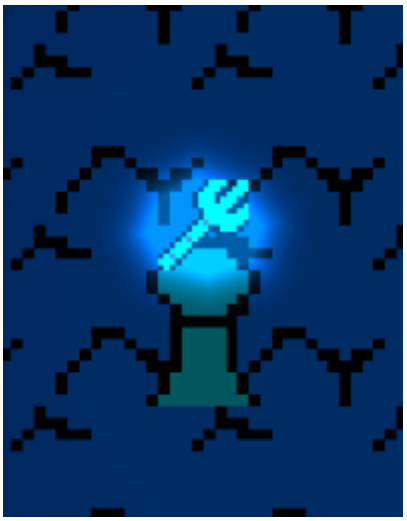

Figura 5.7 Objeto generado sobre pedestal.

Las armaduras se han basado en lo típico de cualquier RPG, pero los objetos se han basado en mitos y leyendas de la mitología griega, como por ejemplo:

- Las monedas de Caronte.
- Las manzanas doradas de las Hespérides.
- El tridente de Poseidón.
- Una almohada de una de las camas de Procusto.
- Madre de un pino de Sinis.

- La caja de Pandora.
- El fuego de Prometeo.
- La piel del león de Nemea.
- El caballo de Troya.
- El barro de la creación de los humanos.

Figura 5.8 Sprite de la caja de Pandora.

La idea sería ir ampliando esta lista para poder tener una gran cantidad de objetos que vender, comprar, usar o con los que crear otros objetos. Los efectos de cada uno de ellos todavía no se han implementado. Por ejemplo, tener en tu inventario un par de monedas de Caronte te permitiría seguir jugando en caso de morir.

Figura 5.9 Sprite de la moneda de Caronte.

Tanto los objetos como las armaduras se muestran en *slots* propios en el inventario si el jugador pausa el juego. De momento no se pueden usar los objetos ni ver los detalles de las armaduras, pero sí se pueden tirar en caso de que el jugador no tenga espacio para otros mejores. En el caso de los objetos, también se muestra su valor y su peso, y en la bolsa se puede ver el peso total que lleva el jugador, aunque de momento no se ha implementado la función de ralentizarle una vez sobrepase un límite.

Figura 5.10 Inventario con objetos.

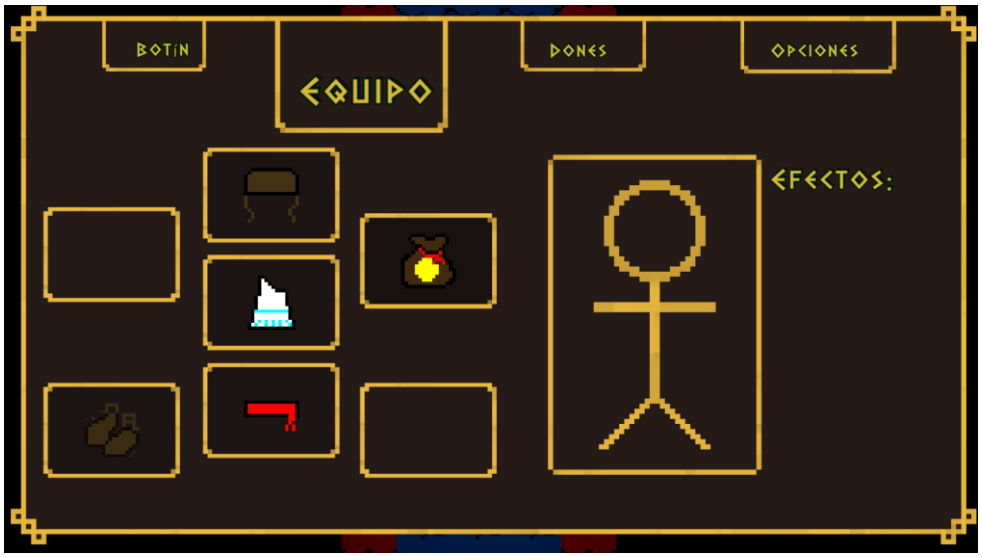

Figura 5.11 Inventario con armaduras.

5.4 Dones

Los dones han sido creados de una manera idéntica a los objetos y armaduras, pero su funcionamiento es totalmente diferente. Su *ScriptableObject* consta de las variables que guardan:

- Nombre.
- Efecto.
- Sprite.
- Tipo (Curación, estadísticas, área u objetivo).
- Distancia del efecto.
- Duración del efecto.
- Tiempo de recarga.

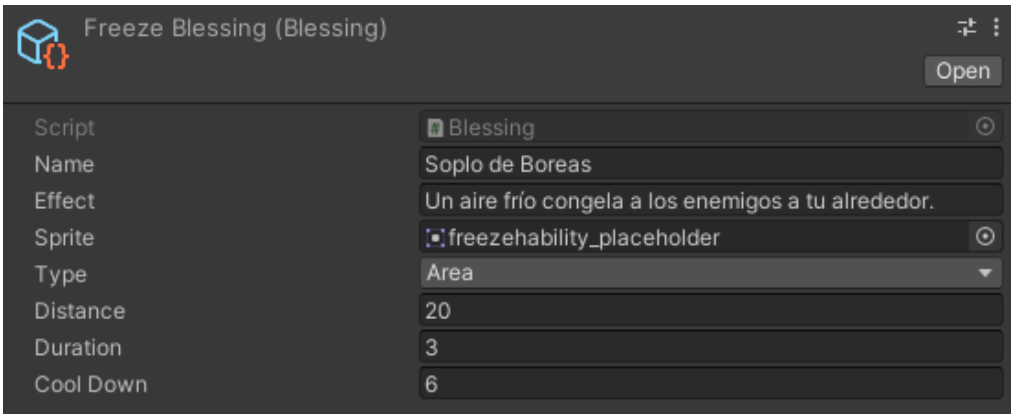

Figura 5.12 Ejemplo de don.

Los dones serían un poco el equivalente a las típicas habilidades de cualquier videojuego, simplemente adaptadas al contexto narrativo de la mitología griega. De momento los que están implementados son:

- Soplo de Bóreas (congelación).
- Maldición Titán (encoger).
- Reloj de Cronos (para el tiempo).
- Don de Zeus (parálisis).

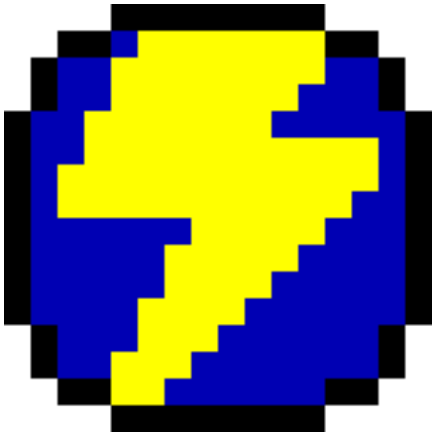

Figura 5.13 Sprite del Don de Zeus.

No ha habido tiempo para implementar todos correctamente tanto en efecto como en gráficos, por lo que actualmente simplemente generan un área en la que los enemigos quedan paralizados durante el tiempo de efecto de cada uno de los dones. Además, como no ha habido tiempo de implementar los avances en la historia para obtener estos dones, se pueden conseguir de manera provisional al pulsar el espacio. La parte de la selección de cada uno de ellos funciona correctamente.

Figura 5.14 Jugador paralizando a un enemigo.

Los dones también se muestran correctamente en el inventario del menú de pausa, y, en caso de pulsar sobre ellos, se puede leer su nombre y su descripción en el recuadro de la izquierda.

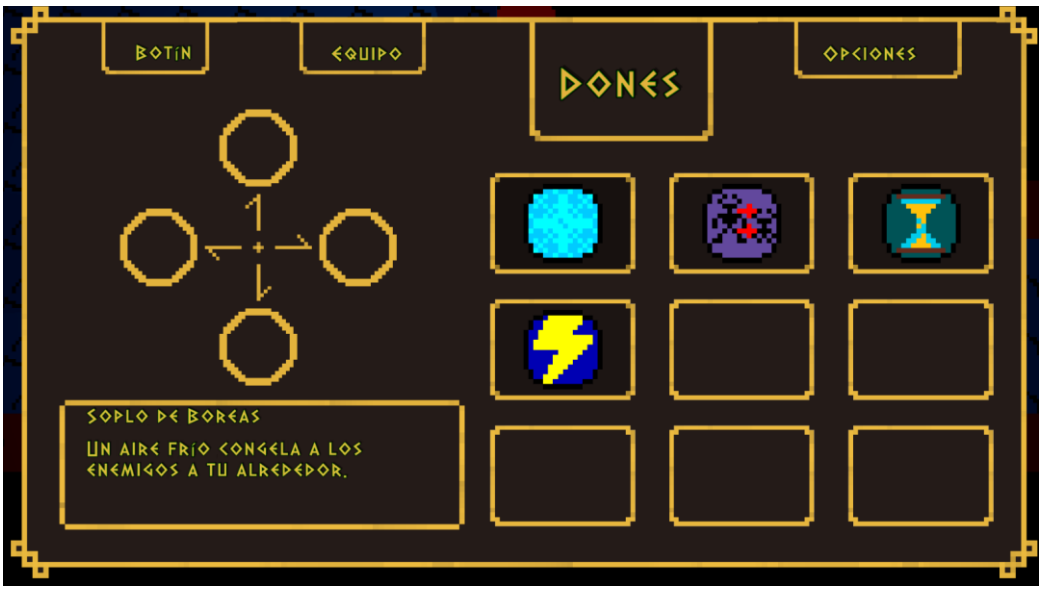

Figura 5.15 Inventario de los dones.

6. Enemigos

No ha dado tiempo a implementar más que tres enemigos, pero su sistema de funcionamiento e inteligencia artificial sí está implementado a la perfección para el futuro. También se ha implementado un *script* que contenga las estadísticas del enemigo, pero no está completamente implementado en la lógica del juego y se ha dejado a modo de *placeholder* de las variables que podrían ser útiles o usarse en el futuro, que son las siguientes:

- Tipo de enemigo (perseguir, melé, huye, estático o a distancia).
- Rango de detección.
- Rango de alerta.
- Velocidad.
- Velocidad cuando ha detectado al jugador.
- Tiempo de recarga de ataque.
- Velocidad de proyectiles.
- *Prefab* de los proyectiles.
- Rango de ataque.
- Patrulla una habitación.

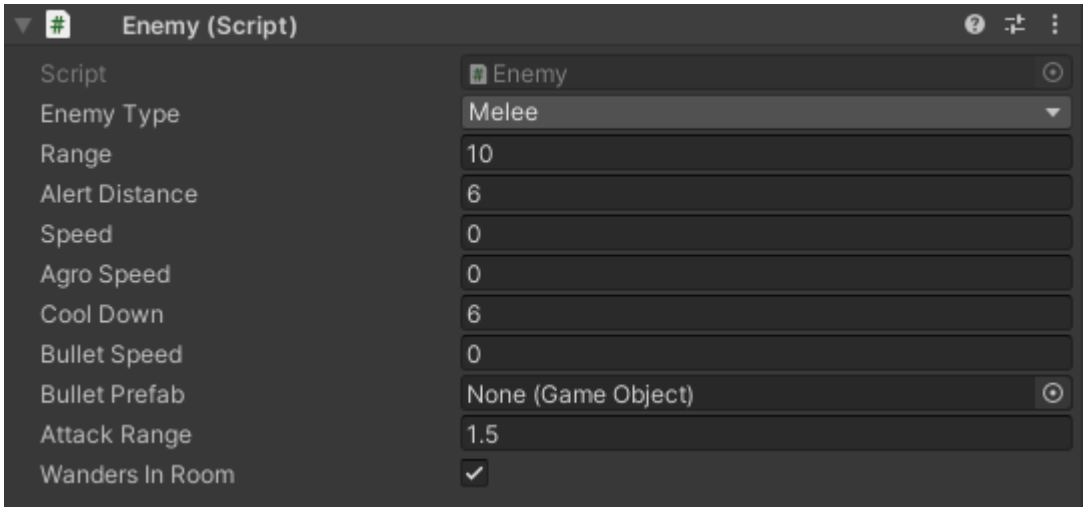

Figura 6.1 Estadísticas de un enemigo.

6.1 Inteligencia Artificial

El desarrollo de la inteligencia artificial (a partir de ahora IA) de los enemigos se ha hecho dos veces. En primer lugar, se hizo una IA basada en un único *script* con lo que se denomina una máquina de estados finitos. Sin embargo, tras algunos errores sin explicación aparente y el tamaño que empezaba a tomar dicho script, se decidió sustituir todo el sistema de la IA por otro en el que hay un *script* central que inicializa todo y se basa en el resto de *scripts* que definen cada estado que puede tomar el enemigo mediante una interfaz común. Los distintos estados y su funcionamiento son:

- *EnemyIA*: El *script* central que se encarga de la inicialización y de declarar los estados. Contiene todas las variables que se necesitan compartir entre los ellos (las estadísticas del enemigo, el estado actual,

el agente del *NavMesh*, el jugador, las estadísticas del jugador, etc.). También es el punto desde el que el enemigo es alertado de la presencia o cercanía del jugador, teniendo así un punto de entrada único en el código, muy útil de cara a modificaciones o corregir errores. Desde este *script* se van actualizando los distintos estados en cada fotograma, dado que ellos no heredan de *MonoBehaviour*.

- *WanderState*: Este estado se encarga del movimiento del enemigo mientras patrulla. Simplemente busca una casilla dentro de la habitación, la marca como destino y se dirige a ella.

- *IdleState*: Este estado se encarga de que el enemigo pase un tiempo aleatorio (dentro de un rango) estático en la misma posición tras haber llegado a su destino patrullando.

- *AlertState*: Estado en el que se ha detectado ruido por parte del jugador. Tras un tiempo de reacción, el enemigo se mueve hasta el punto donde se originó el ruido a investigar su origen. En caso de no ver al jugador, vuelve a patrullar.

- *AttackState*: Si se ha detectado al jugador, el enemigo se mueve hacia él, persiguiéndolo, hasta que esté se encuentre rango de ataque, donde inflige daño a la salud del jugador e inicia una cuenta atrás hasta el siguiente golpe. En caso de que el jugador consiga huir y salir del rango de detección del enemigo, este pasa a estado de alerta yendo al último lugar donde lo detectó.

Figura 6.2 Sprite de un golem.

Hay un tercer enemigo que no se genera al entrar en el laberinto: el Minotauro. Este ser aparece pasados 5 minutos en el mismo piso, tras agotarse el reloj de arena de la interfaz, y su único estado es el de perseguir al jugador. En caso de tocarle, éste morirá inmediatamente. En principio era una mecánica de presión basada en *Pyramid Head* de la saga *Silent Hill* (Konami, 2001) que añadía un pequeño toque de terror, pero se asemeja mucho a una mecánica

que ya existe dentro de los *roguelites* a modo de *time limit*, dado que la mayoría constan de un elemento que impide que el jugador sobreexplote los recursos del mismo piso durante demasiado tiempo. Algunos ejemplos son: una fuerza desconocida en Pokémon Mundo Misterioso (Spike Chunsoft, 2005), un monstruo de limo verde en *Moonlighter* (Digital Sun, 2018) o un fantasma en *Spelunky* (Derek Yu, 2008).

Figura 6.3 El monstruo verde persiguiendo al jugador en el videojuego Moonlighter.

6.2 Sistema de sigilo

Como pilar fundamental del *gameplay*, se ha implementado un sistema de sigilo para dar más profundidad al movimiento del jugador y al comportamiento de los enemigos. El jugador tiene un área invisible a su alrededor con una lista a la que se van añadiendo y sacando enemigos según estén dentro de ella o no. En caso de que el jugador se mueva, se alertará a los enemigos que estén a diferentes distancias según las estadísticas del jugador para la detección al realizar dichos movimientos. Si, por ejemplo, el jugador pasa corriendo a la distancia de ser detectado por un enemigo, este recibirá la posición en la que el jugador ha provocado ese ruido e irá a investigar según los patrones vistos en el punto anterior. Sin embargo, ese mismo enemigo no le hubiese detectado si el jugador hubiese pasado por ese mismo punto pero andando. O incluso podría acercarse mucho más al enemigo si se moviese agachado.

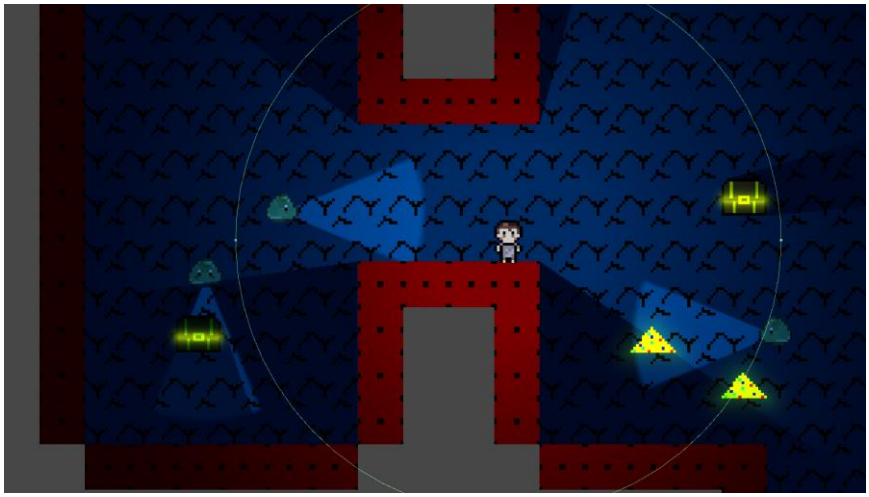

Figura 6.4 Área del sistema de sonido del jugador.

Además, todos los enemigos tienen una luz que representa un campo de visión por donde el jugador, en caso de cruzar, será detectado inmediatamente, salvo que esté escondido tras una cobertura como una pared o un objeto. Al añadir luz dinámica al proyecto, esta luz también choca contra dichos objetos y se ve de manera mucho más clara y directa si el enemigo puede detectar al jugador o no. Este campo de visión de los enemigos es totalmente editable en cuanto a distancia y ángulo de amplitud.

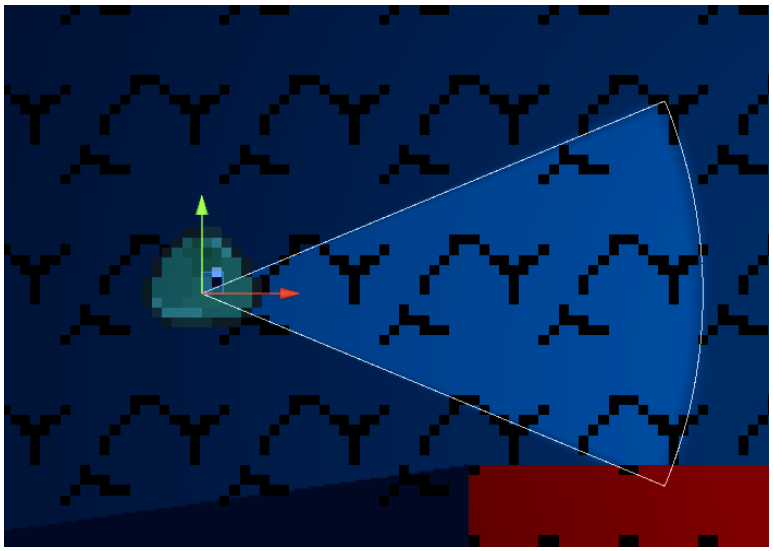

Figura 6.5 Campo de visión de un enemigo.

Para complementar estos sistemas, todos los enemigos tienen a su vez un área en la que avisan al resto de enemigos cercanos de la presencia del jugador en caso de haberle detectado, avisando así de manera natural al resto de enemigos de la zona de la presencia del jugador.

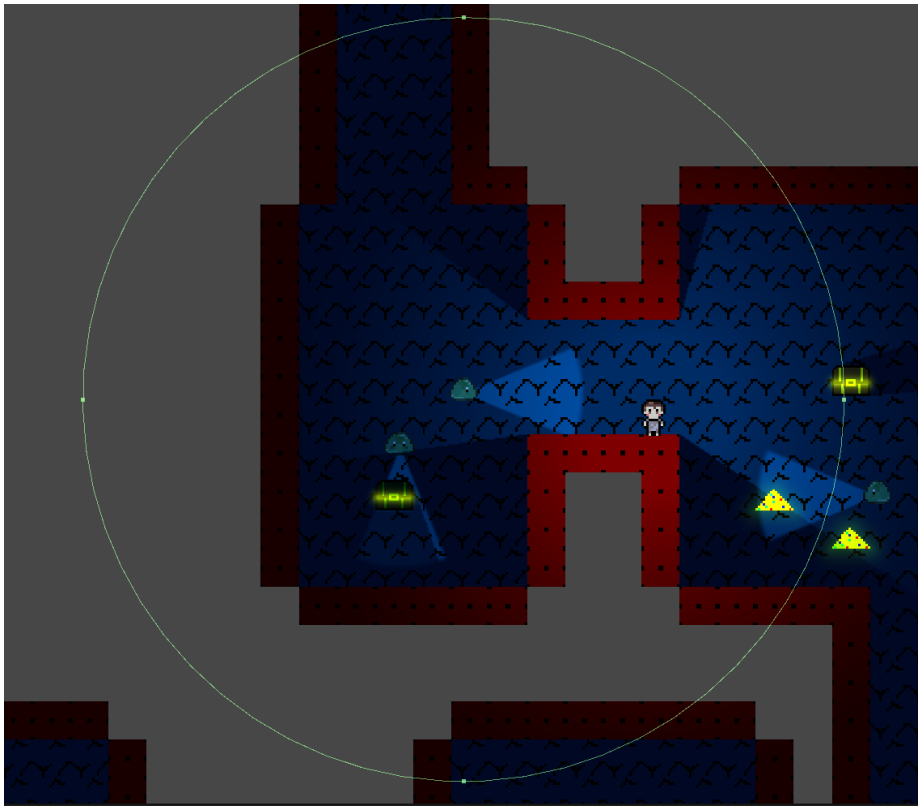

Figura 6.6 Área de alarma de un enemigo.

También se añadieron cofres a la generación de niveles, que si típicamente en un juego sirven para encontrar tesoros, en *Katergáris' Labyrinth* su principal utilidad sería la de poder esconderse dentro de ellos una vez se hubiese obtenido el botín de su interior. La inteligencia artificial de los enemigos está diseñada para volver al estado de alerta en caso de que el jugador se esconda dentro de uno de ellos.

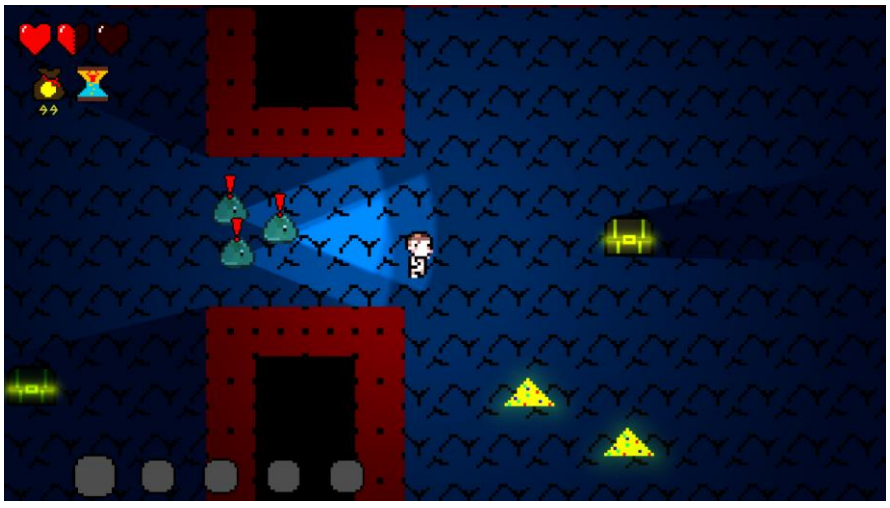

Figura 6.7 Jugador huyendo de enemigos.

Figura 6.8 Enemigos perdiendo de vista al jugador, oculto en un cofre.

6.3 Jefe final

Para el último nivel de la mazmorra y como parte final del prototipo, se ha implementado un jefe final para dejar constancia del teórico funcionamiento de posteriores jefes que se pudiesen añadir más adelante. Crear un jefe final para este tipo de juego en el que no se puede atacar no era tarea sencilla, dado que se salía de los estándares tipos de un enemigo más fuerte que el resto al que hay que derrotar en una sala cerrada y que sirve como examen último de lo aprendido durante el juego. En este caso, como por la mecánica de no agresión no se podía hacer una pelea final, se decidió que los jefes finales fuesen batallas-puzle a resolver por el jugador, y teniendo la temática mitológica griega siempre presente, se decidió ambientarlos en los 12 trabajos de Heracles, dado que muchos de ellos no consistían en una pelea y encajaban a la perfección con el juego y la mentalidad de puzle. Para este en concreto, se eligió el trabajo del robo de las manzanas de oro de las Hespérides, donde en una de las versiones los árboles se encuentran custodiados por un dragón. Al tratar sobre un robo, también parecía cumplirse el ser un examen final de la mecánica principal usada durante el laberinto: el sigilo.

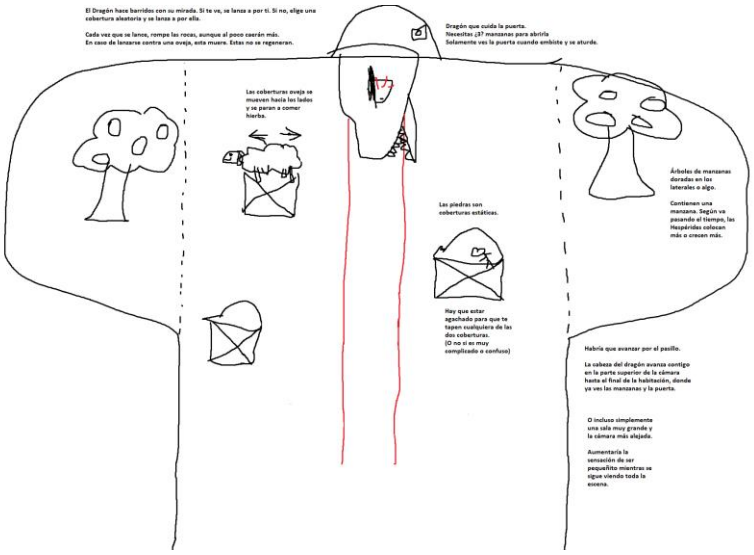

Figura 6.9 Boceto del nivel del jefe final.

Por lo tanto, se diseñó un jefe final que realiza barridos con la mirada (marcada por dos luces) por la sala y, en caso de ver al jugador, atacaba en su dirección. En caso de no vislumbrar al jugador, que se podría ocultar detrás de coberturas como rocas y ovejas, atacaría a un punto aleatorio tras cierto tiempo. Si alguna de las coberturas se cruzaba en su ataque, esta sería destruida y así, con el paso del tiempo sería más difícil para el jugador el pasar desapercibido. Eso sí, siempre que el jugador se acercase a los árboles, el dragón detectaría al jugador y le atacaría salvo que ya estuviese atacando o recuperándose de un ataque.

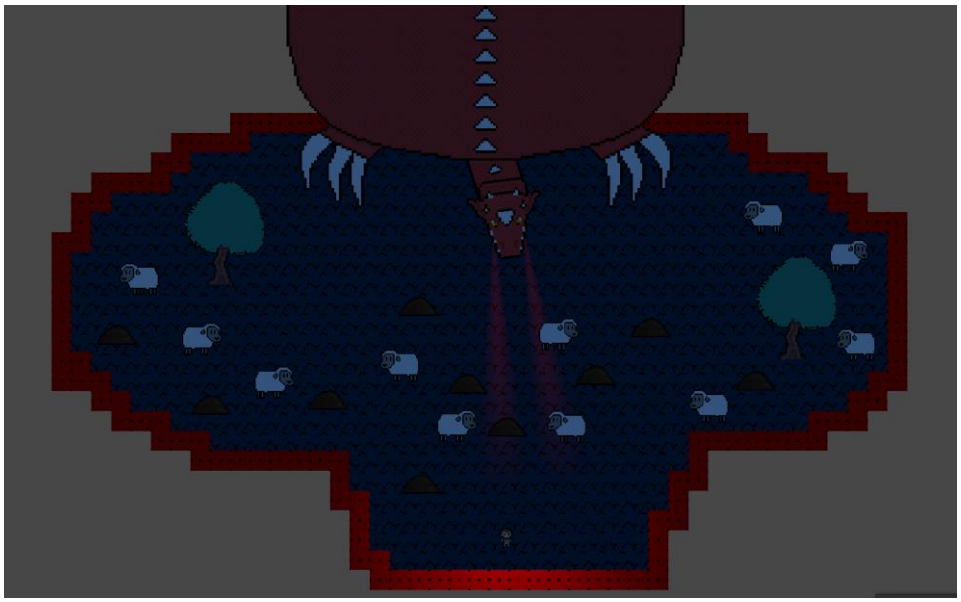

Figura 6.10 Nivel del jefe final.

La inteligencia artificial del dragón fue diseñada de una manera muy similar a la de los enemigos, con varios estados por los que pasa a lo largo del tiempo. La única diferencia es que tiene un estado dedicado a recuperarse de un ataque, donde simplemente recoge su cuello lentamente antes de volver al estado de búsqueda del jugador.

Figura 6.11 Jefe final detectando al jugador.

Al principio se había pensado en una batalla en dos fases: una primera en la que se tuviese que avanzar por un pasillo y la segunda en una sala ya con los árboles de las manzanas doradas, pero por falta de tiempo se recortó la primera fase. También se había pensado en mezclar dos tipos de coberturas: estáticas (piedras) y móviles (ovejas), pero finalmente no dio tiempo a implementar el movimiento de la inteligencia artificial de las ovejas. Tampoco dio tiempo a implementar el sistema de sigilo al completo, por lo que el ruido no tiene ningún efecto en esta batalla.

Figura 6.12 Jefe final atacando al no detectar al jugador.

7. Pueblo

El pueblo o *hub* principal es la zona de descanso entre diferentes incursiones en el laberinto. En él, el jugador puede charlar con sus habitantes o con los héroes que visitan el laberinto, entrar en casas, visitar las tiendas, la consigna, el bosque del oráculo, etc. Esta zona, aunque no tan importante para las bases del *gameplay* principal del laberinto, en principio iba a constar de muchas mecánicas que no ha dado tiempo a implementar como el avance de la historia, una tienda donde customizar tu casa, obtener dones en el oráculo, distintos eventos en la plaza según el día, minijuegos en la feria de la entrada del laberinto… Evidentemente todas estas funciones se salían por mucho del tiempo establecido para el trabajo, por lo que no se han añadido.

Figura 7.1 Vista aérea de la escena del pueblo.

7.1 Diálogos

En primer lugar, se implementó un sistema de diálogos sencillo, pero fácilmente modificable para ser reutilizado sin problema en cualquier NPC o incluso en cinemáticas si hubiese dado tiempo a hacerlas. El sistema de diálogos se basa en tres scripts:

- Clase *Dialogue*: Contiene el nombre del NPC que está hablando, la velocidad con la que se escriben las letras en pantalla y un *array* con las distintas oraciones que vaya a decir. Este *script* se coloca en el *GameObject* del NPC.
- *DialogueTrigger*: Este *script* se encarga de comprobar si el jugador se encuentra lo suficientemente cerca e interacciona con el NPC para empezar con el diálogo e ir avanzando por las distintas oraciones que tenga el emisor. También se encarga de cerrar la ventana de diálogo una vez haya terminado la conversación.
- *DialogueManager*: Este *script* recibe los diálogos del NPC a través del *Trigger* y los muestra en pantalla según se vaya interaccionando.

Guarda las oraciones en una cola y las va sacando letra a letra por orden con la velocidad de habla del NPC. Todo ello, el nombre del emisor y su oración, se muestran en la ventana de diálogo general de la interfaz.

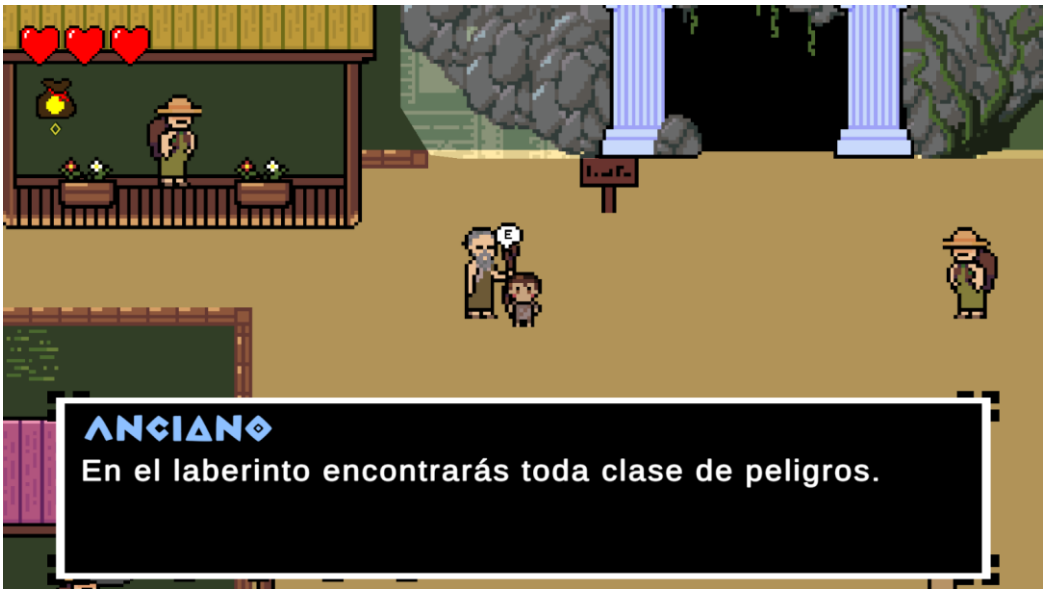

Figura 7.2 Diálogo con NPC 1.

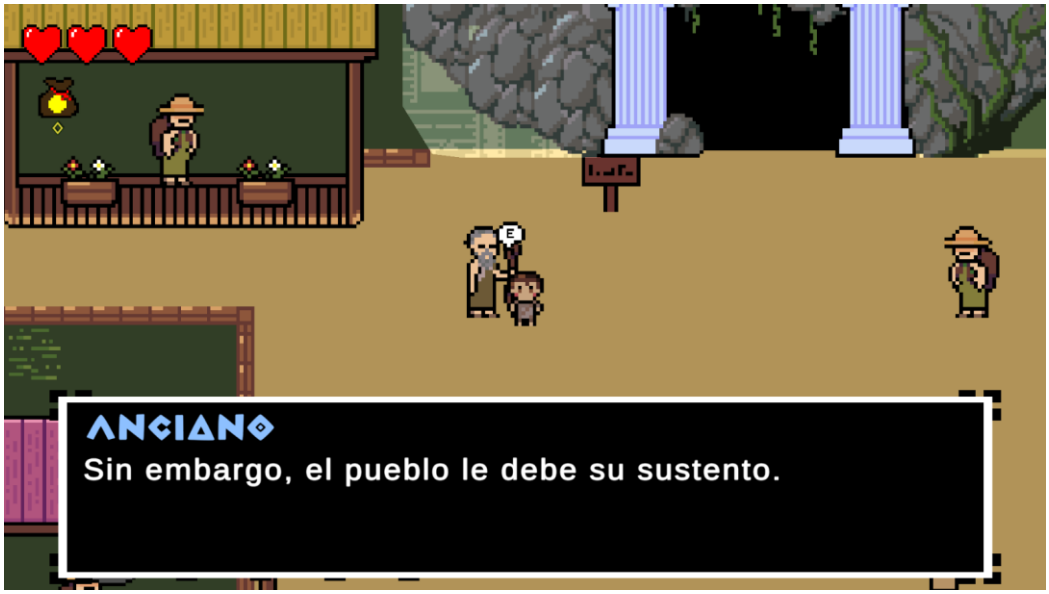

Figura 7.3 Diálogo con NPC 2.

7.2 Casas

Normalmente las casas en muchos videojuegos son escenas independientes en las que entra el jugador al cruzar la puerta. Sin embargo, esto presenta el problema de descargar la escena del pueblo, perdiendo así la posición del jugador al entrar en la casa o teniendo que duplicar muchos

elementos del juego que ya se encontraban en la escena del pueblo para que todo funcionase correctamente, como por ejemplo el sistema de diálogos.

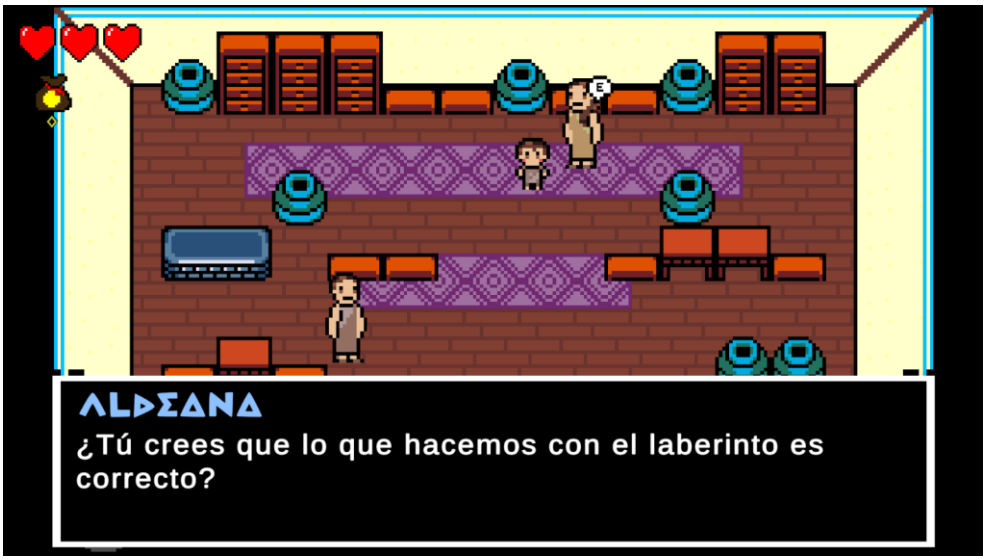

Figura 7.4 Escena de una casa.

Se podría hacer una variable global que guardase la posición del jugador antes de entrar en la casa, pero para probar algo más eficiente, se decidió cargar las escenas de las casas de manera aditiva, encima de la escena del pueblo pero muy alejadas. De esta manera, la escena del pueblo puede seguir cargada y así no se pierde ni la posición del jugador ni hace falta duplicar otros elementos innecesariamente para cada una de las casas. Simplemente hace falta guardar una referencia al jugador del pueblo para bloquear su movimiento y solamente controlar al que se encuentra en la casa, y todo funciona perfectamente. Una vez se sale de la casa, dicha escena se descarga y se continúa en el pueblo como si nada hubiese pasado.

Figura 7.5 Vista aérea del pueblo con la escena de una casa cargada aditivamente.

7.3 Tiendas

Como se mencionó en la introducción del capítulo, algunas de las casas tienen usos especiales. Más concretamente, las dos tiendas y la consigna. Ambas tiendas tienen un funcionamiento similar cambiando lo que se puede vender y comprar (una solamente trabaja con objetos, la otra con armaduras), mientras que la consigna se basa en ellas, pero en esencia funciona de manera diferente.

Figura 7.6 Tienda de objetos.

Las tiendas activan su interfaz tras hablar con el vendedor. Ésta se divide en dos partes: a la izquierda, los objetos o armaduras que el jugador tiene en su inventario y puede vender, y a la derecha los objetos o armaduras que se pueden comprar. Cada *slot* a ambos lados se genera en tiempo de ejecución en base a los objetos/armaduras que el jugador tenga en su inventario, y la lista de objetos/armaduras que el vendedor tenga en su tienda. En caso de vender algo, se obtendrá el dinero que cueste, ese *slot* desaparecerá y el objeto pasará a estar a la venta en la tienda. En caso de comprar algo, se perderá el dinero que cueste, el objeto/armadura irá al inventario y el slot no desaparecerá, pero dejará de estar disponible para vender.

Figura 7.7 Tienda de armadures.

En el caso de la consigna, la interfaz es similar, pero permitiendo que el jugador guarde los objetos/armaduras que tenga en su poder, o permitiéndole recoger los objetos/armaduras que haya guardado previamente en la consigna. No hubo tiempo suficiente para implementar el guardar y sacar dinero.

Figura 7.8 Consigna.

8. Persistencia

Finalmente no hubo tiempo para implementar un sistema de guardado y cargado de partidas, pero realmente las bases para hacerlo ya están en el juego. Como es evidente, todo el funcionamiento del inventario o las tiendas no es persistente por defecto en Unity. Como se expuso con la posición del jugador al entrar en una casa, una vez cargas una escena, todos los datos de la escena anterior se pierden aunque la vuelvas a cargar posteriormente. Por ello, hay un *script* que se encarga de guardar lo que en otros lenguajes se denominarían como variables globales. Estas variables se actualizan en puntos concretos del juego y sirven para mantener persistentes los avances del jugador durante su sesión de juego. En este *script* se guardan:

- Los objetos del jugador. Esta lista se actualiza cuando el jugador entra en el laberinto o consigue salir de él con vida.
- Las armaduras del jugador. Se actualiza en los mismos casos que los objetos.
- El dinero del jugador. Se actualiza en los mismos casos que las variables anteriores.
- La lista de dones del jugador. Igual que las dos anteriores.
- Los objetos guardados en la consigna. Se actualiza cada vez que se guarda o saca un objeto de ella.
- Las armaduras guardadas en la consigna. Se actualiza igual que los objetos.
- El dinero guardado en la consigna. De momento no tiene funcionalidad.
- Los pisos que ha superado el jugador. Esta variable se utiliza para que el jugador pueda avanzar a la escena del jefe final cuando ha superado 4 pisos normales. Se vuelve a poner a 0 al entrar en el laberinto.
- La salud del jugador. Se utiliza para mantener la salud persistente entre pisos, volviendo a recuperarse al máximo una vez se sale del laberinto.
- Un *struct* que contiene los datos de la tienda de armaduras. Los objetos de la tienda se cargan desde la propia tienda y se actualizan aquí. Sobre todo se usa para mantener la persistencia de los objetos que están disponibles para comprar y los que no.
- Un *struct* que contiene los datos de la tienda de objetos. Mismo funcionamiento que la variable anterior.

Como se puede ver, actualmente con estas variables se mantiene una imagen exacta del estado de la partida del jugador, por lo que para añadir

un sistema de guardado y cargado de partidas simplemente haría falta pasar este *script* a un fichero e implementar la capacidad de cargar dicho fichero para replicar el estado guardado. Como mucho podría hacer falta guardar también la posición del jugador en el pueblo.

```
public class GlobalVariables : MonoBehaviour
{
    public static int money = 0;
    public static List<Item> items = null;
    public static List<Armor> armor = null;
    public static List<Item> sloganItems = new List<Item>();
    public static List<Armor> sloganArmor = new List<Armor>();
    public static int sloganMoney = 0;
    public static List<Blessing> blessings = null;

    public static int dungeonFloor = 0;

    public static int health = 6;

    public static Shop armorShop = new Shop();

    public static Shop itemsShop = new Shop();
}
```

Figura 8.1 Variables persistentes.

Esta versión del sistema de guardado no permitiría guardar en la mazmorra, dado que tal y como se genera actualmente no habría manera de replicarla. Para poder guardar en la mazmorra (aunque es una función que omiten muchos *roguelites* para evitar su abuso), haría falta implementar un sistema de semillas para utilizarlas a la hora de generar la mazmorra. Guardando dicha semilla, se podrían reconstruir los pisos en los que guardase el jugador.

9. Conclusiones

Como capítulo final de la memoria, se expone un análisis de los objetivos iniciales del trabajo, así como las conclusiones del mismo a modo de revisión del éxito o fracaso de su implementación y la metodología de trabajo elegida para su desarrollo. Finalmente, se enumerarán las mejoras a futuro que se podrían aplicar a *Katergáris' Labyrinth*.

9.1 Conclusiones generales del trabajo

En líneas generales, los objetivos se cumplieron satisfactoriamente. Se puede ver que los objetivos parciales de cada entrega, que aparecieron en este documento en la sección de la planificación del capítulo 1, se completaron prácticamente en su totalidad. Hubo algunas carencias menores como los efectos de sonido o la implementación completa de objetos y armaduras, pero dado el volumen de trabajo y al haber realizado esta planificación de manera posterior a la inicial, se hizo una buena estimación del tiempo empleado en implementar cada uno de los objetivos en el videojuego. Sin embargo, los objetivos generales del trabajo que se habían fijado con anterioridad, los que conformarían el Producto Mínimo Viable, sí presentan alguna carencia a mayores (un cuarto enemigo a distancia o un inicio de la historia si no contamos la cinemática inicial). Aquí se ve más claramente que desde un principio el videojuego era demasiado ambicioso para el tiempo disponible y, a pesar de haber dedicado mucho tiempo y esfuerzo en él, hubo que dejar fuera algunos elementos no esenciales.

Esta falta de tiempo seguramente también se deba a algo positivo, y es que se ha alcanzado un producto muy completo que implementa algunas mecánicas esenciales de prácticamente cualquier videojuego. Nunca había hecho un algoritmo de generación de niveles tan complejo y variado, o un sistema de tiendas, o un sistema de inventario, o un sistema de diálogos, o un sistema de persistencia… Pero creo que todos ellos se han implementado de manera más que satisfactoria a pesar de ser la primera vez que realizaba algo de estas dimensiones por iniciativa propia y sin guía.

Sobre la planificación y la metodología de trabajo elegida, decir que ha sido un éxito. El ir entregando prototipos funcionales en todas y cada una de las entregas ayuda mucho a ver el avance real del proyecto y a ser consciente del tiempo que va a requerir la siguiente. Ha sido el enfoque adecuado, y creo que se ha demostrado completando prácticamente todo el trabajo a tiempo para cada una de las PECs. El único fallo podría ser el no haber considerado el tiempo para pruebas y testeos en cada uno de los *sprints*.

Viendo lo expuesto en esta sección, diría que el único fallo real ha sido la envergadura del videojuego. Si en vez de haberme decantado por un *roguelite* con unas bases muy originales y con muchos elementos típicos de un RPG, hubiese pensado en hacer algo más sencillo, seguramente tendría un videojuego con todo lo planteado inicialmente y con algo más de pulido. De

todas formas, considero que esto también es algo normal y parte de este trabajo final de máster, dado que se podría decir que es el primer proyecto que dirijo por completo de inicio a fin y, sin experiencia en estos temas, era normal no estimar de manera del todo correcta los tiempos y la carga de trabajo. Pero como ya mencioné en este apartado, también he visto una mejora en mis estimaciones a lo largo de los meses y eso se ha reflejado en las entregas parciales. También creo que se refleja que la idea original era buena y consistente cuando, a la hora de la verdad, no ha habido prácticamente cambios en los objetivos durante el desarrollo. Todos los elementos que se han quedado fuera o que se podrían mejorar, han quedado así por falta de tiempo, no porque hubiese un mal planteamiento inicial.

9.2 Mejoras a futuro

Desde el primer momento se planteó este videojuego como algo más que un trabajo final de máster. Siempre se consideró la opción de continuar su desarrollo más allá del periodo educativo, por lo que se pueden enumerar una lista de elementos que podrían implementarse o mejorarse con más tiempo de desarrollo. Algunas de estas mejoras podrían ser:

- Medidor de resistencia. Serviría para evitar que el jugador pudiese correr por toda la mazmorra sin cansarse y así evitar a todos los enemigos sin esfuerzo.

- Medidor de oxígeno. Para evitar que el jugador pueda esconderse indefinidamente dentro de un cofre. Añadiría una mayor planificación y gestión de tiempos a esta mecánica.

- Enemigos que detecten el haberse escondido en un cofre. Un poco en la línea del punto anterior para que los cofres no desbalanceen tanto el sistema de sigilo, se podría hacer algo típico de juegos de estilo *Survival Horror*, en los que si un enemigo ve al jugador escondiéndose en algún elemento, como por ejemplo un armario, este fuera capaz de detectarlo y sacarlo de ahí.

- Reajustar las capas de dibujado de los *sprites*. Ahora mismo hay muchos *sprites* que se superponen a otros cuando no deben. Se programó un *script* que modificaba la profundidad de dibujado en tiempo de ejecución, pero el resultado todavía no es perfecto.

- Mejoras generales de la interfaz y gráficos. Todos los elementos gráficos que se presentan actualmente en el prototipo son provisionales. En caso de querer comercializar el juego, habría que darle un buen lavado de cara en todos los aspectos (*sprites*, animaciones, diseño visual…) e incluso adaptar algunos elementos de algunas partes de la interfaz para que se adaptasen a un *scroll* lateral en caso de constar de demasiados elementos, como pasa actualmente en las tiendas.

- Mayor variedad de enemigos y biomas. A mayores del enemigo a distancia que se dejó fuera durante el desarrollo, faltaría añadir mucha variedad de enemigos basados en la mitología griega. El laberinto también contará con diferentes niveles de distinta temática, enemigos, tesoros y dificultad (arquitectura griega, ruinas, el río Estigia, el inframundo, el Tártaro…). En ellos también podrá encontrarse con *bosses*, que al no poder derrotar, funcionarán como una especie de sala-puzle gigante que el jugador deberá resolver antes de debilitarse.

- Mayor profundidad en la toma de decisiones del jugador. Actualmente el jugador no tiene un gran abanico de posibilidades para la toma de decisiones, pero se podrían implementar también momentos algo más arriesgados, como el encontrarte un héroe errante intentando completar el laberinto. El jugador puede decidir ignorarlo, beneficiarse de su lucha contra las criaturas del lugar… o utilizar sus habilidades para dificultarle el avance, dado que en caso de ser derrotado por el laberinto, el jugador podrá *lootear* su cadáver y obtener valiosos premios.

- Mayor variedad de mecánicas en el pueblo. A mayores de unas tiendas y consigna más funcionales, se añadiría el funcionamiento del oráculo para la obtención de habilidades, el avance en la historia principal, alguna *sidequest* de recolección de objetos en el laberinto, la funcionalidad de la tienda del alquimista y el laboratorio de monstruos mediante un sistema de *crafteo*, la funcionalidad de la tienda de ropa y de decoración de la casa del jugador, los minijuegos de la feria, zonas para explorar… Este es el aspecto del juego en el que menos se ha podido profundizar durante el desarrollo y el que más juego podría dar en el futuro.

- Enfoque didáctico. La idea de la biblioteca era que el jugador pudiese investigar y leer sobre los mitos de los objetos o enemigos que fuese encontrándose por el laberinto. Puede que le llamase la atención una manzana dorada y descubriese el mito del trabajo de Heracles en el que las roba del jardín de las Hespérides, o que descubra una leyenda que le parezca interesante tras huir de las aves de estínfalo. Esto daría no solamente una serie de coleccionables a mayores que alentarían al jugador a explorar el laberinto en busca de diferentes enemigos y objetos, sino que le daría además un toque didáctico sobre la mitología griega al videojuego.

10. Glosario

- *Roguelike*: Género de videojuegos que comparten aspectos comunes con el videojuego *Rogue* (A.I. Design, 1980). Se traduciría, literalmente, como “similar a *Rogue*”.

- *Roguelite*: Variación del género *roguelike* en el que no todos los avances conseguidos por el jugador se pierden al morir en el videojuego.

- *Minimum Viable Product* (MVP): Producto mínimo viable.

- *Placeholder*: Se llama así al archivo provisional cuyo objetivo es ocupar el espacio que, más adelante, será ocupado por el archivo definitivo.

- Inteligencia Artificial (IA): Conjunto de algoritmos y código que definen el comportamiento de un personaje del videojuego que no es controlado por el jugador.

- *Blockout*: Se denomina con este nombre al conjunto de formas geométricas primitivas que conforman el boceto en tres dimensiones de un nivel al que más adelante se le añadirá más detalle.

- *Non Playable Characters* (NPCs): Se denomina así a cualquier personaje de un videojuego que no pueda ser controlado por el jugador. Normalmente se usa para referirse a los personajes pasivos con los que el jugador puede interaccionar.

- *Hub*: Se llama así a la zona central de un videojuego desde la que se accede al resto de zonas del mismo. Suele ser una zona de descanso o preparación a la que el jugador regresa regularmente.

- *Dungeon crawler*: Juego de mazmorras. Género de videojuegos basado en el juego de rol “Dragones y Mazmorras” que consiste en la exploración de una estructura laberíntica con diferentes enemigos y tesoros a conseguir por uno o más jugadores.

- *Loop* jugable: En diseño de videojuegos, se denomina “*loop* jugable” al bucle de retroalimentación de la experiencia que el jugador repetirá durante sus sesiones para seguir avanzando. Por ejemplo: matar monstruos para hacerse más fuerte y matar monstruos más fuertes.

- *Rigging*: Estructura de huesos ficticios que se utiliza para crear animaciones de modelados 3D.

- *Scriptable Object*: En Unity, se denomina como *ScriptableObject* a un contenedor de datos con una estructura concreta predefinida

fácilmente replicable y modificable. Su principal utilidad es el ahorro de memoria.

- *GameObject*: En Unity, se denomina como *GameObject* al contenedor de las características y componentes de cualquier elemento en una escena.

- *Prefab*: En Unity, un *prefab* es una plantilla de un *GameObject* guardada en memoria para replicarse numerosas veces en uno o varios niveles de manera sencilla. Por ejemplo, un mismo tipo de enemigo podría tratarse como un *prefab*.

- *Script*: Archivo que contiene un fragmento de código de programación.

- *Struct*: Estructura utilizada en programación para definir un conjunto de variables agrupadas bajo un mismo nombre. Por ejemplo, se podría utilizar un *struct* llamado “coche” desde el que acceder a sus variables velocidad, marca y número de ruedas.

- *Sidequest*: Se denomina así a las misiones secundarias de un videojuego que no forman parte de la historia principal del mismo.

- *Crafteo*: Sistema típico de videojuegos en el que se utilizan una serie de objetos para formar otro diferente. Ejemplo: en *Minecraft* (Mojang Studios, 2011) se pueden utilizar cuatro lingotes de hierro para formar un solo bloque de hierro.

11. Bibliografía

[1] **Varios autores** (2015). *Game & Play. Diseño y análisis del juego, el jugador y el sistema lúdico*. Editorial Advisory Board.

[2] **Philip Wilkinson** (2009). *Mitos y Leyendas. Una guía ilustrada de su origen y significado*. Editorial Dorling Kindersley.

[3] **Arthur Cotterell** (2008). *Enciclopedia de mitología universal*. Editorial Parragon Books.

[4] **Moonlighter Gamepedia** (2020) [en línea] *Controls*. Gamepedia. Disponible en: https://moonlighter.gamepedia.com/Controls

[5] **Josh Bycer** (2019) [en línea] The Roguelike Debate – Roguelikes vs Roguelites. Gamasutra. Disponible en: https://www.gamasutra.com/blogs/JoshBycer/20191125/354673/The_Roguelike_Debate__Roguelikes_vs_Roguelites.php#:~:text=Rogue%2Dlite%20are%20typically%20designed,persistence%20and%20carryover%20between%20runs.

[6] **Brackeys** (2018) [en línea] *How to make a 2D Game*. YouTube. Disponible en: https://youtu.be/on9nwbZngyw?list=PLPV2KyIb3jR6TFcFuzI2bB7TMNIIBpKMQ

[7] **Sunny Valley Studio** (2020) [en línea] *Procedural 2d Dungeon Tutorial*. YouTube. Disponible en: https://youtu.be/-QOCX6SVFsk?list=PLcRSafycjWFenI87z7uZHFv6cUG2Tzu9v

[8] **h8man** (2021) [en línea] *NavMeshPlus*. GitHub. Disponible en: https://github.com/h8man/NavMeshPlus

[9] **Unity** (2019) [en línea] *2D Lights and Shadows in Unity 2019! (Tutorial)*. YouTube. Disponible en: https://youtu.be/F5l8vP90EvU

[10] **Sebastian League** (2015) [en línea] *Field of view visualisation (Unity 5)*. YouTube. Disponible en: https://youtu.be/rQG9aUWarwE?list=PLFt_AvWsXl0dohbtVgHDNmgZV_UY7xZv7

[11] **Code Monkey** (2019) [en línea] *Simple Shop in Unity (Buy Items, Weapons, Armor)*. YouTube. Disponible en: https://youtu.be/HuXy4XX0hzg

Ludografía:

Nicalis (2011). The Binding of Isaac. Headup Games.

Spike Chunsoft (2020). Pokémon Mundo Misterioso: Equipo de Rescate DX. The Pokémon Company.

Digital Sun (2018). Moonlighter. 11 bit studios.

Frictional Games (2011). Amnesia: The Dark Descent. THQ.

Ubisoft Quebec (2018). Assassin's Creed Odyssey. Ubisoft.

Ubisoft Quebec (2020). Immortals Fenyx Rising. Ubisoft.

Supergiant Games (2018). Hades. Supergiant Games.

Kojima Productions (2015). Metal Gear Solid V: The Phantom Pain. Konami Digital Entertainment.

Sucker Punch Produtions (2005). Sly 3: Honor Entre Ladrones. Sony Interactive Entertainment.

A.I. Design (1980). Rogue. Epyx.

Konami (2001). Silent Hill. Konami.

Digital Sun (2018). Moonlighter. 11 bit studios.

Derek Yu (2008). Spelunky. Mossmouth.

Mojang Studios (2011). Minecraft. Mojang.