Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN

Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

DESARROLLO DE UN RPG DE ACCIÓN USANDO TÉCNICAS DE GENERACIÓN PROCEDURAL DE MAZMORRAS

Alumno: Ramón Cruz Blanco

Tutor: Carlos Javier Ogayar Anguita

Dpto: Departamento de Informática

Junio, 2023

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Carlos Javier Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: '**Desarrollo de un RPG de acción usando técnicas de generación pocedural de mazmorras**', que presenta Ramón Cruz Blanco, da el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, julio de 2023

El alumno: El tutor:

Ramón Cruz Blanco Carlos Javier Ogayar Anguita

(Página intencionalmente en blanco)

Agradecimientos

A mis padres, que sé que me quieren con locura pese a los debates políticos que montamos que ni en la tele, por hacer todo lo posible y más para que pueda llegar hasta aquí. No creo que jamás pudiera pagar la deuda de todo lo que habéis hecho por mí si me la pidiérais.

A Bogu, mi compañera de vida y mi amor, la persona más bonita y más linda del mundo, en la que más confío y a la que más quiero, por aguantar mis infodumps autistas sobre arqueología o lo que sea que sea mi hiperfoco del momento, por dormir conmigo la siesta abrazaditos con Moris en medio espachurrao. Por haceme ver, en definitiva, que soy, después de todo, una persona digna de amor. Te quiero infinito omega bb <3.

A Vero, por haber sido durante años un pilar fundamental en mi vida y haberme acompañado durante tanto tiempo, en los buenos y en los malos momentos, cuando damos la putivuelta en el botellón para ver si hay alguien y cuando estoy de bajón y te mando mensajes de quinientos millones de palabras y tu me mandas audios de trescientas mil horas. No sé qué haría sin ti.

A mis amigos, Fernando, Manuel, Enrique, José, Antonio, David, Tomás, Sergiu y todos los demás, por haberme acompañado algunos desde la infancia, otros desde la universidad, por meternos un bocadillo de salchichón o lo que tocase mientras estábamos medio dormidos en la última fila, por jugar conmigo al pokemon cuando éramos pequeños, por hacer el tonto cada vez que salimos con nuestro humor rotísimo.

Solo me gustaría deciros que os quiero muchísimo a todos y que solo quiero seguir recorriendo mi vida a vuestro lado durante todo lo que pueda, que me habéis cambiado a mejor y que no tengo palabras para agradecéroslo. Ahora sé que el hogar no es la casa donde vives sino aquellos que te recuerdan cuando ya no estés. Algún día, todos nos iremos y yo sé que todas estas personas, si me voy, me recordarán, y, si se van, yo sé que no habrá un día en que no las recuerde. Y eso es todo lo que importa al final.

Gracias, por todo.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Ramón Cruz Blanco
Fecha de asignación	10/11/2021
Descripción corta	Desarrollo de un videojuego RPG de acción en 3D en el cual se utilizarán una serie de técnicas de generación procedural con el objetivo de explorarlas y discutir su relevancia y posibles usos en el contexto de la industria del videojuego.

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

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

Contenido

Índice de ilustraciones

Índice de tablas

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparecen en el apartado 6 (Definiciones y abreviaturas).

1.1 Introducción

El objetivo principal de este proyecto es el desarrollo de un videojuego *Role-Playing Game* (RPG) de acción al estilo de los de la saga *The Legend of Zelda* (Nintendo, 1986) o los de la saga *Dark Souls* (FromSoftware, 2011), donde un personaje recorre un mundo de fantasía en el cual se deberá de enfrentar a diversos enemigos usando armas tales como la espada o el arco. Este tipo de videojuegos tienen una gran importancia histórica en la industria y han adquirido una popularidad extraordinaria en época reciente, influyendo en gran medida a otros géneros e incluso otros medios, como el cine o la literatura.

No obstante, estos videojuegos tradicionalmente han seguido un esquema de diseño donde los niveles son **monolíticos**, es decir, se crean previamente y su disposición y contenido no cambia a lo largo del tiempo o entre partidas. Recientemente, han venido surgiendo esquemas de **generación procedural**, esto es, niveles que se generan de manera aleatoria siguiendo una serie de reglas (I. Fernández, 2016). La principal motivación de este proyecto es, pues, la exploración de estas técnicas de generación procedural y su aplicación al desarrollo de un videojuego RPG de acción.

1.2 Objetivos del trabajo

A continuación, se realizará una descripción general de los objetivos de este proyecto:

- Definir las mecánicas de juego más típicas del género RPG e implementar algunas de ellas en nuestro proyecto; fundamentalmente, las propias de un sistema de combate.
- Realizar una **exploración** de las técnicas de generación procedural más populares a día de hoy y seleccionar algunas de ellas para implementarlas.
- Diseñar los diferentes **desafíos** a los que el jugador se deberá de enfrentar progresivamente, y cuáles serán sus condiciones de **victoria** y **derrota**.
- Llevar a cabo la selección y/o creación de *assets* para usarlos en el videojuego, tales como escenarios, modelos 3D, efectos de sonido y música…
- Ejecutar el diseño de una **interfaz** de usuario que indique estado actual del jugador y el de una serie de **menús** a través de los cuales podrá navegar para cambiar parámetros (volumen, sensibilidad…) y pausar el juego, entre otras cosas.

1.3 Antecedentes y estado del arte

1.3.1 Historia del RPG

La **historia de los RPG** se remonta a los años 80. Inspirados por los juegos de rol de mesa como *Dungeons & Dragons* (Tactical Studies Rules, 1974) o novelas de fantasía como *El Señor de los Anillos* (J. R. R. Tolkien, 1954), estos primeros títulos, lanzados para microordenadores como el *TRS-80*, ya tenían algunos de los elementos fundamentales y que hoy en día identificamos con ellos: ambientaciones de fantasía y/o ciencia ficción, exploración de mazmorras, sistemas de combates basados en las batallas medievales de espada, lanza y escudo (ya sea en tiempo real o por turnos), gestión de inventario o creación de personajes.

Ilustración 1. Ultima I: The First Age of Darkness (Origin Systems, 1981)

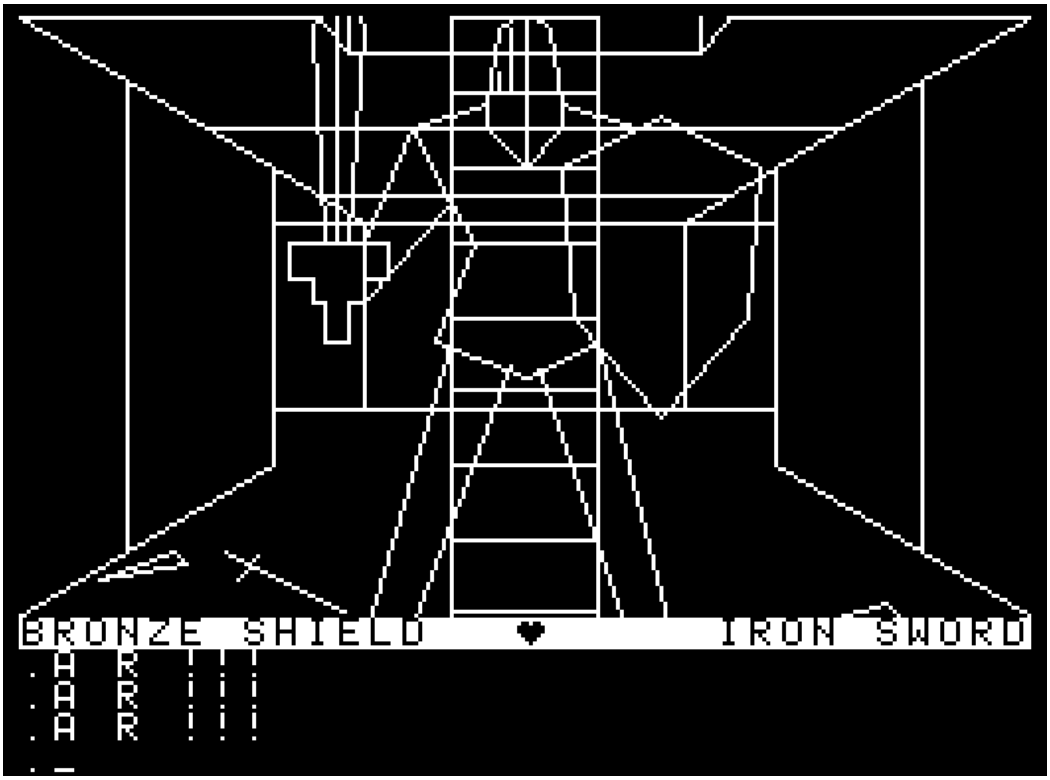


Ilustración 2. Dungeons of Daggorath (DynaMicro, 1983)

Pasando a las consolas, un hito fundamental fue el lanzamiento de *The Legend of Zelda* (Nintendo, 1986) para la *Nintendo Entertainment System* (NES), un videojuego que se alejaba de las convenciones del RPG clásico de ordenador (el *Computer Rol-*

Playing Game (CRPG), como se le suele denominar) y eliminaba algunos de los elementos característicos de estos (creación de personajes, estadísticas, inventario complejo) para ofrecernos un videojuego mucho más centrado en la aventura, la exploración y el combate a tiempo real.

Ilustración 3. The Legend of Zelda (Nintendo, 1986)

Durante esta época también se desarrolló el otro gran exponente del RPG de consolas, *Final Fantasy* (Square, 1987), también originalmente lanzado para la NES. Este videojuego, pese a presentarnos también un mundo de fantasía, se aleja de la fórmula de *The Legend of Zelda* y nos ofrece combates por turnos en lugar de combate a tiempo real, manejo de inventarios complejo y progresión del personaje, pero con una sensibilidad distinta al CRPG clásico, ofreciéndonos una aventura con gran énfasis en el apartado narrativo.

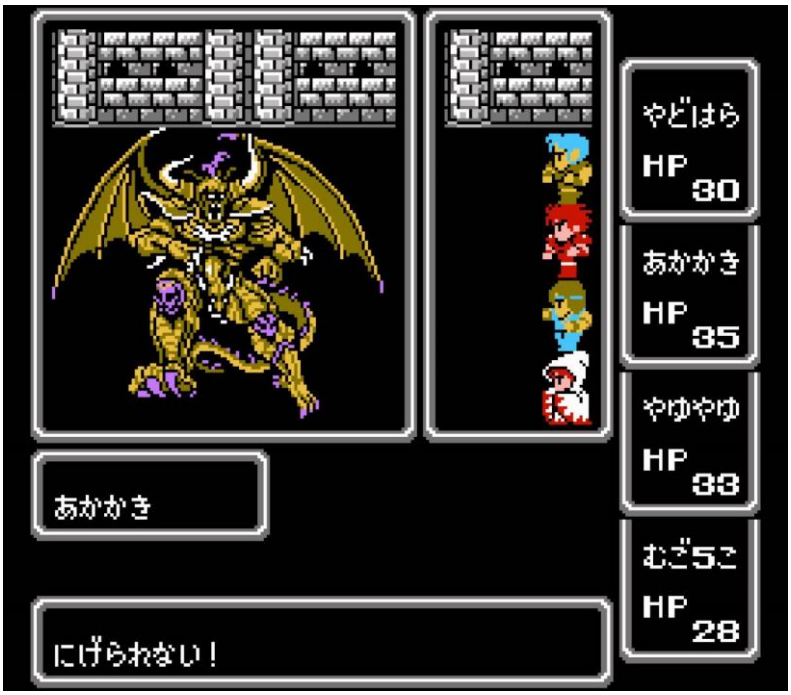


Ilustración 4. Final Fantasy (Square, 1987)

Así pues, en los 90 se crea una **diferenciación** importante entre los juegos RPG de consola y de ordenador, los cuales seguirán desarrollos diferentes; más aún, los de consola serán los que predominen en esta época, juegos que seguirán la estela de *The Legend of Zelda* y *Final Fantasy*. Hasta que, a finales de los 90, nos encontramos con un resurgimiento de los juegos de rol en ordenadores con títulos como *Fallout* (Black Isle Studios, 1997), que se caracterizaban por la perspectiva isométrica y la toma de decisiones en tramas ramificadas.

Ilustración 5. Fallout (Black Isle Studios, 1997)

Ya entrando en el siglo XXI, las diferencias entre consolas y computadores se suavizaron; ahora, los títulos solían salir tanto para consolas como para ordenador y los videojuegos no se quedaban en una única plataforma. De esta forma, las diferencias entre RPG de consola y de ordenador pasaron a ser diferencias culturales, entre títulos RPG japoneses (JRPG) y títulos RPG occidentales (WRPG). Los primeros se caracterizaban por gráficos más brillantes y caricaturescos, por tener tramas lineales y cinematográficas protagonizadas por personajes predefinidos y por un sistema de combate por turnos. En contraste, los occidentales presentaban personajes más viejos, tramas y gráficos más oscuros y desarrollos narrativos ramificados, así como combate en tiempo real.

Ilustración 6. Final Fantasy X (Squaresoft, 2002)

Ilustración 7. Mass Effect (BioWare, 2007)

Hasta la fecha, esta dicotomía entre JRPG y WRPG perdura, pero ahora los RPG se han diversificado mucho más y nos encontramos con juegos que es difícil categorizar estrictamente en una etiqueta u otra. Este es el caso de *Dark Souls* (FromSoftware, 2011), un juego que recupera algunos elementos clásicos (creación del personaje, estadísticas) pero que aporta una nueva sensibilidad en muchos aspectos, como su combate actualizado y una narrativa difusa y críptica.

Ilustración 8. Dark Souls (FromSoftware, 2011)

Este último ejemplo será vital para nuestro proyecto, así como los *The Legend of Zelda* en 3D, juegos que abandonan algunas de las características definitorias del RPG y se acercan más a la aventura y la exploración: los llamados **RPG de acción**.

1.3.2 Historia de las técnicas de generación procedural de contenidos en videojuegos

La programación procedural para la generación de contenidos lleva siendo usada en videojuegos desde los años 80, con títulos como *Rogue* (Epyx, 1980) y *Elite* (D. Braven e I. Bell, 1984). *Rogue* es el precursor del género que lleva su nombre, el *rogue-like*, y se carateriza por el mazmorreo, la interfaz visual en ASCII, los niveles aleatorios y la muerte permanente del protagonista, enviándole directamente a la pantalla de título al morir.

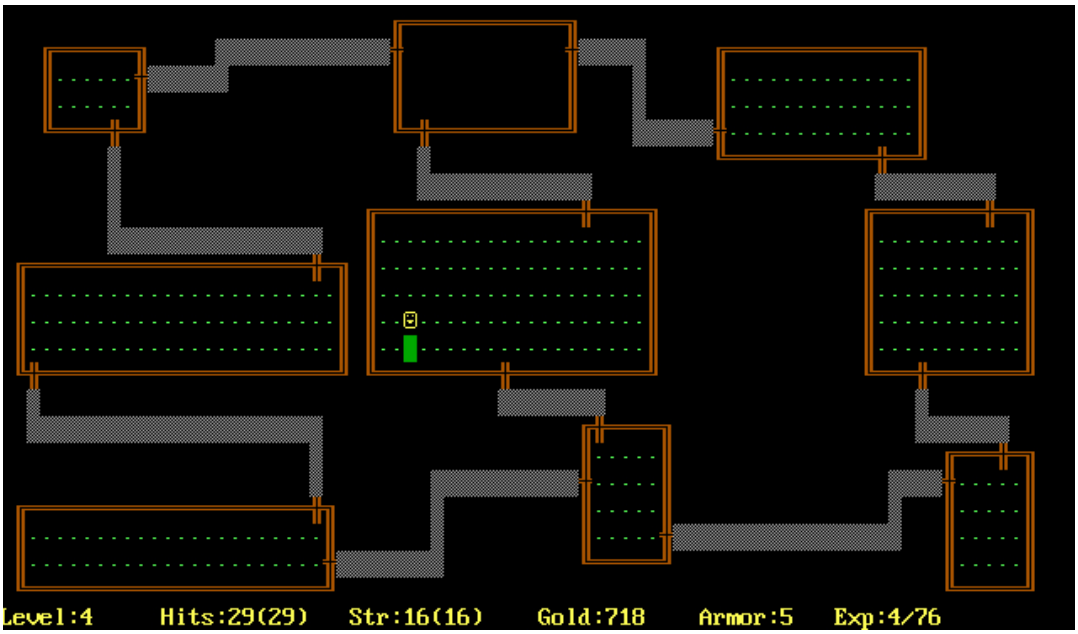


Ilustración 9. Rogue (Epyx, 1980)

Por su parte, *Elite* fue todo un hito: en un ordenador obsoleto como el *Acorn Electron*, era capaz de generar 8 galaxias con 256 planetas cada una, y cada planeta con información como su descripción, posición en cada galaxia, tipos de minerales o precios de las mercancías generada de forma pseudo-aleatoria por medio de semillas, sin necesidad alguna de guardar esta información en disco.

Ilustración 10. Elite (David Braven e Ian Bell, 1984)

El siguiente salto importante ocurrió con *Diablo* (Blizzard North, 1996) y *The Elder Scrolls II: Daggerfall* (Bethesda, 1996). El primero usó la generación procedural para elementos pequeños, como determinados *items*, mientras que el segundo fue un logro colosal: un mapa de 229,848 km² lleno de mazmorras, *Non Playable Characters* (NPCs), misiones y objetos generados de forma procedural, en tiempo real y en 3D.

Ilustración 11. The Elder Scrolls II: Daggerfall (Bethesda, 1996)

Una década después llega *Dwarf Fortress* (Bay 12 Games, 2006), una obra colosal que genera mundos absolutamente únicos e increíblemente complejos: altura, condiciones climáticas, temperatura ambiental, pueblos, NPCs, conflictos bélicos, flora, fauna y hasta la misma historia y mitología del mismo es generado proceduralmente.

Ilustración 12. Dwarf Fortress (Bay 12 Games, 2006)

En esta primera década del siglo XXI tenemos además a *Left 4 Dead* (Valve, 2008), que usará técnicas procedurales para hacer que, si se nos está dando demasiado bien el videojuego, aparezca menos munición o que haya menos salidas, o *Spore* (W. Wright, 2008), que creaba organismos y galaxias enteras donde crecer y evolucionar junto al ecosistema.

Ilustración 13. Spore (Will Wright, 2008)

Por supuesto, no se puede hablar de generación procedural sin mencionar a *Minecraft* (Mojang, 2009), juego que es capaz de generar mundos cúbicos de miles

de millones de km^2 con sus respectivos biomas, recursos, fauna, flora y formaciones geográficas como cuevas, playas, montañas o islas.

Ilustración 14. Minecraft (Mojang, 2009)

Como hemos visto, hay muchas propuestas y las técnicas procedurales están muy extendidas en la industria y son usadas en aspectos muy diferentes de un videojuego (en la música, por poner un ejemplo poco conocido). Para nuestro proyecto, las ideas propuestas por *Rogue* son fundamentales; tanto que posiblemente el videojuego que hemos desarrollado se deba de catalogar como un ***rogue-like***.

1.4 Descripción de la situación de partida

La motivación de este proyecto consiste, fundamentalmente, en explorar algunas de las **técnicas de generación procedural** que se han venido usando actualmente, ya que la industria del videojuego ha atravesado toda una revolución a este respecto en los últimos años. Por tanto, nuestro trabajo apunta en esa dirección e intenta proponer algunas ideas y discutir su utilidad.

Como veremos más adelante, los recursos software de los que hemos partido son ***Unity*** como motor gráfico y **C#** como lenguaje de programación, entre otras tecnologías. Además, se ha dispuesto de un **PC de gama media** para el desarrollo del videojuego y, en cuanto a **recursos humanos**, quien ha llevado el grueso del proyecto ha sido un servidor, alumno de Ingeniería Informática de la UJA cuya experiencia está limitada a cuatro años de carrera.

1.5 Requisitos iniciales

A continuación, se realizará un resumen de los requisitos a muy alto nivel del proyecto, pues la descripción de requisitos funcionales y no funcionales se realizará más adelante:

- El jugador **debe** tener el control de un personaje a través del cual poder explorar el mundo del videojuego.
- El personaje del jugador **debe** ser capaz de desplazarse a través del escenario.
- El mundo de juego **debe** estar habitado por diversas criaturas hostiles que atacarán al jugador.
- El jugador **debe** poder defenderse de estas criaturas de alguna manera.
- El jugador **debe** poder atacar a los enemigos de alguna forma.
- Los escenarios del juego y su contenido **deben** generarse de forma procedural.
- El jugador **debe** tener acceso a su estado actual por medio de una interfaz de usuario.
- El jugador **debe** poder ganar la partida de alguna forma.
- El jugador **debe** poder ser derrotado por los monstruos.
- Los enemigos **deberían** ser variados, algunos más poderosos que otros y con ataques diferentes.
- Los escenarios **deberían** ser variados también y estar decorados de alguna forma.
- El jugador **debería** tener acceso a varias formas de atacar y defenderse, no solo una.
- El jugador **debería** poder recoger recursos a lo largo del escenario.
- El juego **debería** hacer uso de diversos efectos audiovisuales para explosiones, música…
- El juego **debería** de contar con un menú principal al comienzo de la partida.

- El juego **debería** tener un menú de pausa con el que pausar la partida.
- El juego **debería** permitir al jugador modificar determinados parámetros (volumen, sensibilidad…) por medio de menús.
- El juego **debería** permitir al usuario usar tanto teclado/ratón como mando de consola.

1.6 Alcance

Al finalizar el proyecto se entregarán los siguientes elementos:

- **Ejecutable del videojuego.** Archivo con el que se podrá ejecutar el videojuego, en formato *.exe*.
- **Proyecto en *Unity*.** Proyecto a través del cual se ha desarrollado el videojuego, con todo el código fuente, los *assets*, etc… el cual se encuentra alojado en un repositorio de *GitLab*.
- **Documentación.** Se entregará, además, este documento, en el que se detalla todo el proceso de desarrollo, en formato *pdf*.
- **Video demostrativo.** Siendo como es nuestro proyecto un videojuego, es importante mostrar su funcionamiento en vivo a través de material audiovisual. Se entregará, pues, un vídeo en formato *mkv*.

1.7 Hipótesis y restricciones

El TFT se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de 300 horas, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por consiguiente, la principal restricción aplicable es la limitación de la duración del trabajo.

1.8 Estudio de alternativas y viabilidad

1.8.1 Elección del motor gráfico

La primera elección de software a la que nos enfrentamos es la del **motor gráfico** que vamos a utilizar. Un motor gráfico es un software complejo que requiere

proporcionar un conjunto variado de operaciones en tiempo real y usando diferentes dispositivos con el objetivo de proporcionar al desarrollador un entorno de trabajo para el proceso de creación de un videojuego u otro software gráfico. Es por ello que generalmente se encuentran estructurados en **subsistemas** comunicados entre sí, los cuales pueden ser **capas** (que representan niveles de abstracción y donde las capas inferiores más cercanas al hardware son transparentes para el usuario y cuyos recursos son usados por las capas superiores) y en **particiones** (donde cada partición está enfocada a una determinada funcionalidad) (J.M. Gregory, 2009).

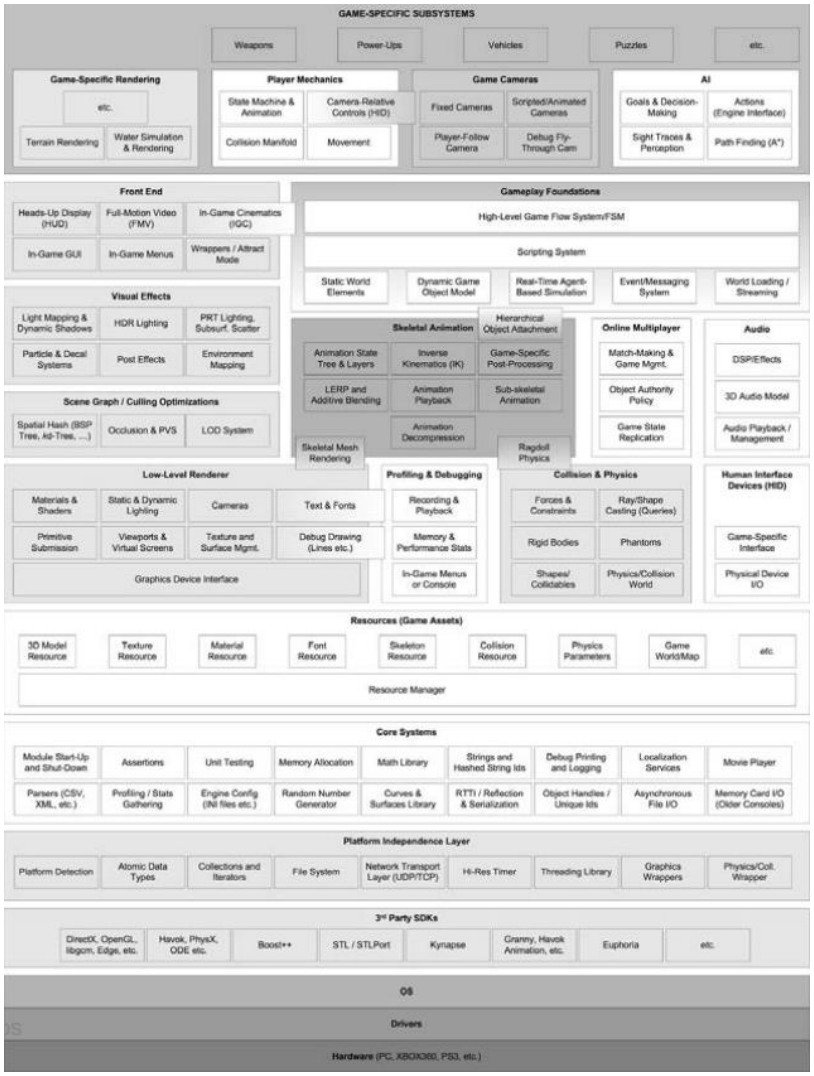


Ilustración 15. Ejemplo de la arquitectura de un motor gráfico (J. Gregory, 2009)

Por lo general, durante el desarrollo de un videojuego se suele optar entre dos alternativas: bien construir tu propio motor gráfico, bien usar uno comercial de terceros. La primera está totalmente fuera del alcance de nuestro proyecto ya que

requiere una inmensa cantidad de recursos y está casi únicamente destinada a grandes proyectos de las empresas más punteras. Por tanto, nos queda la segunda, que es la más común para proyectos pequeños y medianos, como el nuestro.

A día de hoy, existen muchas opciones entre las que elegir, pero, en general, el mercado está dominado por dos: ***Unity*** y ***Unreal Engine***. Ambos motores tienen ventajas y desventajas y apartados en los que uno es superior al otro. Aquí una comparación resumida de las características de cada uno:

	Unreal Engine	**Unity**
Engine Type	Cross-platform	Cross-platform
Developed by	Epic Games	Unity Technologies
Programming Languages	C++ for development	C# for development
Usage	Develop games for PCs, mobiles, consoles, and more	Develop games for PCs, mobiles, consoles, and more
Features	A robust multiplayer framework, VFX, and particle simulation	2D improvements, animation, creating snapshots
Source Code	Open-source	Not open-source.
Pricing	Free	Basic version is free
Learning Curve	Difficult to learn	Easy to learn with an intuitive interface
Graphics	Photorealistic graphics used in AAA games	Good overall graphics, but less refined than Unreal

Ilustración 16. Comparativa de características entre Unity y Unreal Engine (S. K. Arora, 2023)

Entrando en más detalle, *Unity* presenta las siguientes características clave (S. K. Arora, 2023):

1. *Workflow* fácil de entender y una arquitectura preparada para desarrollo rápido de videojuegos.
2. Creación de videojuegos de alta calidad vía gráficos AAA, tasas de frames suaves y audio HD.
3. API de *scripting* que da un control preciso sobre el funcionamiento del videojuego usando C# como lenguaje.
4. Herramientas dedicadas tanto para desarrollo 2D como 3D.
5. Despliegue rápido de videojuegos en cualquier plataforma, incluyendo PC, consolas y móviles.
6. Tienda de *assets* enorme que permite reducir considerablemente el tiempo de desarrollo.
7. Gran cantidad de opciones para personalizar la *rendering pipeline*.
8. La posibilidad de crear y destruir *game objects* personalizados.

En retrospectiva, nos encontramos con que este motor gráfico presenta las siguientes ventajas y desventajas:

Pros	Cons
Free to use	Lower graphic quality than Unreal Engine
C# code is fast for development	Asset store quality is variable
Easy to learn	Less popular with AAA gaming studios
Huge user base	Lacks real-time multiplayer abilities
Extensive 2D game support	Lack of open-source code base
Vast asset store	

Ilustración 17. Ventajas y desventajas de Unity (S. K. Arora, 2023)

Pasando ya a *Unreal Engine*, este presenta también sus propias características clave (S. K. Arora, 2023):

1. Sistema de *scripting* visual para no programadores.

2. *Unreal Audio Engine* y *MetaSounds* para audio de alta calidad.
3. Apoyo para Realidad Aumentada (AR) y Realidad Virtual (VR).
4. Posibilidad de *scripting* con *Python*.
5. Creación de escenarios y mundos con el *Unreal Editor*.
6. Posibilidad de animar personajes con *machine learning*.
7. Motor Nanite para rendering, lightning y materiales.
8. Simulación de partículas y efectos.
9. Unreal Motion Graphics (UMG).
10. Uso de C++ como principal lenguaje de *scripting*.

Asimismo, vamos a ver las ventajas y desventajas que nos ofrece *Unreal Engine*:

Pros	Cons
Free to use and open-source	Harder to learn due to C++
Excellent 3D-realistic graphics quality	Lack of 2D abilities vs Unity
Popular with AAA studios	Smaller asset store compared with Unity
Real-time multiplayer capabilities	Smaller user base
Faster rendering than Unity	Difficult to run on older hardware

Ilustración 18. Ventajas y desventajas de Unreal Engine (S. K. Arora, 2023)

1.8.2 Elección de las técnicas de generación procedural

Existen una gran diversidad de algoritmos que podemos usar para la generación procedural de mazmorras; por tanto, vamos a realizar una breve inspección de las diferentes alternativas que tenemos disponibles para elegir la más adecuada a nuestro caso concreto (J. R. Mendicute Arminio, 2018):

- **Sistemas basados en restricciones.** Digamos que este es el enfoque *naive* donde se codifican cada una de las reglas y condiciones asociadas a cada objeto prediseñado para colocarlo dentro de la escena atendiendo a una serie de parámetros, como el tamaño de la escena, el punto de inicio…

De esta forma, tiene como puntos a favor la “facilidad” de implementación, pero como puntos negativos la baja variabilidad del entorno resultante, ya que estas condiciones atan el resultado final.

- **Autómatas celulares.** Consisten en una malla de células (generalmente una matriz bidimensional o tridimensional) en la que cada una de estas células contiene cierta información (el estado de la célula, que para una mazmorra puede ser simplemente “caminable” y “no caminable”). Cada cécula conoce además el estado de sus vecinas, de forma que este va variando según determinado algoritmo; en concreto, este algoritmo debería asegurarse que las células caminables estén, en su mayoría, conectadas, para formar habitaciones. Este tipo de algoritmos es muy útil para, por ejemplo, la generación de cuevas, pues proporciona un aspecto muy irregular, pero no tanto para el tipo de mazmorras clásicas divididas en habitaciones que queremos.

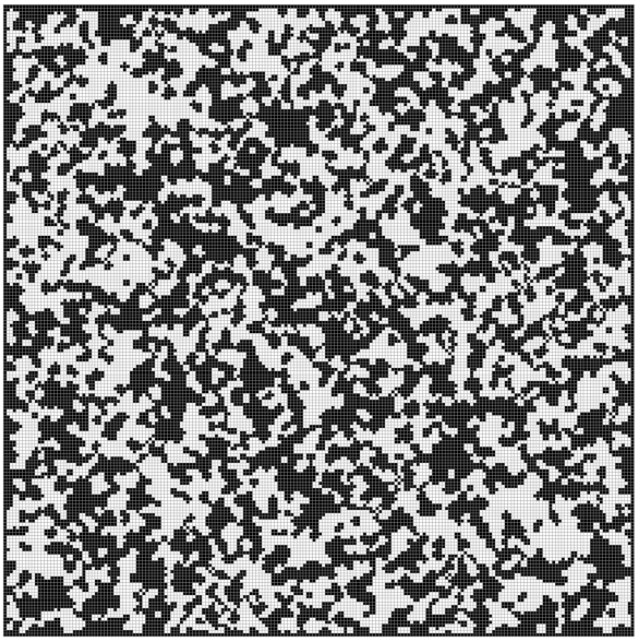


Ilustración 19. Mapa generado por medio de autómatas celulares (J. Kun, 2012)

- **Algoritmos genéticos.** Los algoritmos genéticos son algoritmos de optimización, búsqueda y aprendizaje inspirados en los procesos de **evolución natural y genética**. Para nuestro problema, se podría representar cada posible mapa de una mazmorra como un individuo de una población sujeta a las leyes de la selección natural, de forma que cada

individuo a su vez tenga su codificación como un **genotipo**, el más habitual siendo la representación con **cromosomas**; por lo general, vectores de parámetros de algún tipo (números reales, binarios, etc…) que representaran a su vez parámetros de ese determinado mapa. Por medio de una **función de *fitness*** (básicamente, una función que nos indique la calidad de una solución determinada y, por tanto, su probabilidad de sobrevivir a la selección) el algoritmo realiza un proceso de optimización donde las posibles soluciones (cromosomas) se combinan entre sí (se **reproducen**) y **mutan** hasta alcanzar una solución óptima a través de sucesivas selecciones donde se descartan las soluciones malas y sobreviven las más adecuadas (A. E. Eiben & J. G. Smith, 2003). Esta técnica tiene la ventaja de proporcionar una enorme variedad de posibles soluciones al problema, pero tiene el incoveniente de la difcultad de establecer una correcta representación de los cromosomas y de elegir una buena función de *fitness*, lo cual no es trivial.

- **Gramáticas generativas.** Las gramáticas generativas parten de una gramática dada (esto es, un **alfabeto** de símbolos y unas **reglas** que establecen las formas en que estos símbolos se pueden combinar) para generar **palabras**, es decir, conjunto de símbolos. En nuestro caso, estos símbolos del alfabeto serían escenarios y misiones, los cuales serían generados a partir de una serie de símbolos iniciales. Este método es particularmente bueno para generar **grafos de misiones**, de los cuales hablaremos más adelante.

- **Aproximaciones geométricas.** No se trata de un algoritmo en concreto sino de un conjunto muy variado de algoritmos en los cuales se aplican una serie de operaciones geométricas para determinar la disposición de las habitaciones. Algunas de estas operaciones pueden ser cálculo de triangulaciones, algoritmos de cálculo de camino óptimo para grafos, cálculo de intersecciones, etc…

1.9 Descripción de la solución propuesta

Por una parte, en cuanto a la elección de motor gráfico, como hemos visto, *Unity* y *Unreal Engine* en realidad ofrecen características muy similares: ambos proveen de herramientas para el desarrollo de juegos multiplataforma, ya sea de *rendering*, de físicas, de iluminación, para la animación de los personajes, para editar terrenos o soporte para VR y AR. No obstante, existen también una serie de diferencias: *Unreal* no cuenta con soporte específico para videojuegos en 2D mientras que *Unity* sí, de la misma forma que *Unreal* cuenta con herramientas para el desarrollo de juegos multijugador de las que *Unity* carece, por poner algunos ejemplos.

Teniendo en cuenta la naturaleza de nuestro proyecto, nos hemos decidido decantar por ***Unity*** por diversos motivos, entre los cuales está su curva de dificultad de aprendizaje más suave, la mayor y mejor documentación que está disponible, su amplísima *asset store* (algo sumamente importante, pues no contamos con artistas u otros especialistas que se encarguen de generar *assets* y crearlos supondría un gasto enorme de tiempo) y, sobre todo, la experiencia previa que personalmente tengo con *Unity* de la cual carezco para *Unreal Engine*. Además, en general, diría que *Unreal Engine* tiene ventajas fundamentalmente en el apartado gráfico, el cual no será tan relevante para nuestro proyecto, que contará con uno relativamente sencillo.

Por otra parte, en cuanto a la elección de las técnicas de generación procedural de contenidos, vistas las distintas alternativas, para la generación de nuestras mazmorras vamos a seguir varios **enfoques geométricos** aplicados a la generación de escenarios, y, además, vamos a intentar implementar una **gramática generativa** para la generación de misiones.

1.10 Tecnologías utilizadas

A continuación, haremos un repaso de las tecnologías que se han venido usando a lo largo del proyecto:

- **Unity.** Como ya hemos comentado extensivamente en el anterior apartado, *Unity* es el motor gráfico seleccionado para llevar a cabo este proyecto, debido fundamentalmente a que es muy adecuado para proyectos pequeños y con fines educativos como este. En concreto, *Unity* como tal está dividido en dos softwares: el primero es *Unity Hub*, que es sencillamente el gestor de proyectos de *Unity* donde podemos seleccionar

el proyecto que queremos editar, así como la versión del *Unity Editor*, el otro software, que es el propio *framework* de *Unity*. En particular, vamos a estar usando la versión 3.4.2 del *Unity Hub* y la 2020.3.26f1 del *Unity Editor*, fundamentalmente porque son las versiones que he venido usando hasta la fecha.

- **C#.** Como también hemos comentado ya, este el lenguaje que usa *Unity* para su API de *scripting*, por lo que será el que usaremos en su mayor parte. Se trata de un lenguaje orientado a objetos de alto nivel con gestión autónoma de la memoria a diferencia de C++, por lo que se acercaría más a *Java*, que cuenta además con multitud de librerías para estructuras de datos muy diversas y otras tantas funcionalidades.
- **Visual Studio 2019.** Se trata del IDE de programación que vamos a utilizar para escribir todo el código de nuestro proyecto. *Unity* integra automáticamente este IDE, de forma que todas sus librerías son accesibles fácilmente y no hay necesidad de configurar dependencias ni nada por el estilo. Por lo demás, se trata de un IDE increíblemente popular y bien asentado que cuenta con muchas opciones de depuración de código, entre otras funcionalidades.
- **Json.** Es un formato de texto sencillo para el intercambio de datos, de forma que podemos codificar en ficheros objetos de una determinada clase que luego queremos cargar en nuestro proyecto. Puesto que, como veremos más adelante, hemos necesitado guardar en disco algunas **reglas de la gramática** y *Unity* nos proporciona buenas herramientas para codificar y decodificar en *Json*, esta es una gran opción.
- **Librerías.** A lo largo del proyecto se han tenido que usar muchas librerías de *Unity* y C#, tales como ***System*** y ***UnityEngine***, por citar algunas, las cuales contienen algunas funcionalidades muy importantes como estructuras de datos u operaciones matemáticas. Importante es citar el uso de ***Delaunator***, una librería que nos facilita el cálculo de la triangulación de Delaunay, un algoritmo que usaremos para la generación procedural de mazmorras.

- **Unity Assets.** Para la elaboración del videojuego se han necesitado usar diferentes *asset packages* para, por ejemplo, el modelo y las animaciones del personaje principal y los enemigos, así como de diferentes entornos, *props* y efectos audiovisuales, entre otras cosas. Más adelante se irán viendo cada uno de los que se ha usado conforme se expliquen los incrementos del proyecto.
- **Draw.io.** Se trata de una aplicación de diagramación propiedad de *Google* que es la que hemos usado en la elaboración de esta memoria para diagramas y otros esquemas, como los diagramas de actividad y los esquemas de la planta de las mazmorras.
- **GitLab.** Es la herramienta de **control de versiones** que vamos a utilizar en nuestro proyecto. El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que se puedan recuperar versiones específicas más adelante. Permite además revertir cambios en el proyecto o ver quién modificó por última vez algo. Prácticamente todos los sistemas "reales" utilizan algún tipo de control de versiones y es esencial para proyectos en equipo y muy útil en proyectos individuales. En concreto, *Git* es una herramienta de control de versiones distribuidos, en el que los clientes no solo descargan la última instantánea de los archivos, sino que se replica completamente el repositorio, de manera que, si un servidor falla, se puede recuperar el repositorio, y, además, cada vez que se descarga una instantánea se hace una copia de seguridad completa de todos los datos.
- **Microsoft Word.** Es la herramienta de procesamiento de texto que se ha usado para la elaboración de la documentación.

1.11 Metodología de desarrollo de software

Un paso crucial en el desarrollo de cualquier software es elegir la **metodología** que se va a seguir a lo largo de todo el proyecto. La ingeniería del software nos proporciona varios **modelos del proceso**, esto es, conjuntos de actividades que conducen a la creación de un producto software (R. S. Pressman, 2010). Entre los cuales, vamos a seleccionar para nuestro proyecto una metodología **incremental**. A

diferencia del modelo clásico en cascada, en lugar de entregar el producto en su totalidad este se divide en incrementos de tal modo que cada entrega es un producto operativo pero incompleto, los incrementos se estructuran de modo que los primeros incluyan los requisitos más importantes y cada incremento es desarrollado en un periodo de tiempo determinado y breve.

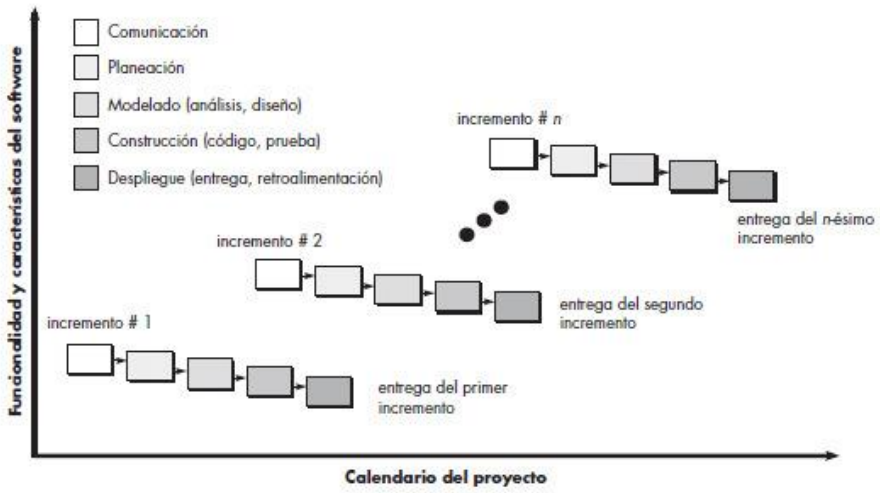


Ilustración 20. Esquema del modelo incremental (R. Pressman, 2010)

Este modelo presenta una serie de ventajas:

1. Los clientes pueden hacer uso de un software con las características primordiales en etapas tempranas.
2. Se pueden utilizar los primeros incrementos como prototipos para obtener información de los mismos.
3. El riesgo de que el proyecto falle se disminuye con respecto al modelo clásico.
4. Los incrementos prioritarios del sistema son los más probados.

Para nuestro proyecto, la metodología incremental es muy adecuada, ya que un videojuego es fácilmente compartimentalizable en diversos paquetes, de manera que se van añadiendo funcionalidades en cada uno de ellos.

1.12 Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFT, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados el tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (la persona autora del trabajo).

Para la estimación del tamaño y el esfuerzo vamos a hacer uso de **modelos de estimación**, que se tratan de modelos matemáticos que relacionan determinados parámetros del proyecto con su coste y/o esfuerzo requerido. En concreto, vamos a usar el modelo **COCOMO**, que se basa en una estimación previa del tamaño del software en líneas de código (LDC) o en miles de líneas de código ($KLDC$), y que nos da el esfuerzo en personas por mes (p/mes) y el tiempo de desarrollo en meses.

En COCOMO nos encontramos con diferentes tipos de desarrollo:

- **Orgánico.** Se trata de un desarrollo realizado en entornos estables, con poca innovación técnica, de un tamaño pequeño ($< 50.000\ LDC$) y de un tiempo relativamente corto.
- **Empotrado.** Software con requerimientos muy restrictivos, gran volatilidad (inestable), complejo y desarrollado en un entorno de gran innovación técnica.
- **Semi-libre.** Entre los dos anteriores.

Puesto que calculamos que nuestro proyecto tendrá un tamaño de entorno a las $7\ KLDC$ y, por lo general, no tendrá un desarrollo demasiado largo y el entorno es relativamente estable, se tratará de un desarrollo **orgánico**.

COCOMO hace uso de una serie de ecuaciones para realizar las estimaciones de esfuerzo y tiempo de desarrollo:

$$E_D = a \cdot (KLDC)^b$$

$$T_D = c \cdot (E_D)^d$$

Siendo E_D el esfuerzo de desarrollo, T_D el tiempo de desarrollo, y a, b, c, d constantes que nos vienen dadas por el tipo de desarrollo.

Existen además diferentes versiones de COCOMO:

- **Básico.** Se usan fundamentalmente para estimaciones iniciales del proyecto, las cuales no son muy precisas. No hay detalles sobre el proyecto y se usa la ecuación básica.
- **Intermedio.** Usa la ecuación nominal, aunque el esfuerzo nominal no está adaptado al entorno de desarrollo. Aquí ya sí nos encontramos con el uso de otros parámetros y ofrece mayor precisión.
- **Detallado.** Existe detalle de cada componente independiente del sistema. Permite refinar y ajustar factores adaptándose a cada etapa del proyecto.

En concreto, las constantes antes mencionadas son las siguientes dependiendo del tipo de desarrollo:

	a	b	c	d
Orgánico	3,20	1,05	2,50	0,38
Semi-libre	3,00	1,12	2,50	0,35
Empotrado	2,80	1,20	2,50	0,32

Tabla 1. Parámetros del COCOMO según el tipo de proyecto

Como ya hemos mencionado, existe la ecuación básica, que es la detallada más arriba, y la ecuación nominal para COCOMO intermedio y detallado, que es la siguiente:

$$E_D = M(x) \cdot a \cdot (KLDC)^b$$

Siendo $M(x)$ el factor de ajuste dado a partir de diferentes parámetros que se especifican en esta tabla:

Conductores de coste	Muy Bajo	Bajo	Nominal	Alto	Muy Alto	Extr. Alto
Fiabilidad requerida del software	0,75	0,88	1,00	1,15	1,40	-
Tamaño de la base de datos	-	0,94	1,00	1,08	1,16	-
Complejidad del producto	0,70	0,85	1,00	1,15	1,30	1,65
Restricciones del tiempo de ejecución	-	-	1,00	1,11	1,30	1,66
Restricciones del almacenamiento principal	-	-	1,00	1,06	1,21	1,56

Volatilidad de la máquina virtual	-	0,87	1,00	1,15	1,30	-
Tiempo de respuesta del ordenador	-	0,87	1,00	1,07	1,15	-
Capacidad del analista	1,46	1,19	1,00	0,86	0,71	-
Experiencia en la aplicación	1,29	1,13	1,00	0,91	0,82	-
Capacidad de los programadores	1,42	1,17	1,00	0,86	0,70	-
Experiencia con el S.O.	1,21	1,10	1,00	0,90	-	-
Experiencia con el lenguaje de programación	1,14	1,07	1,00	0,95	-	-
Prácticas de programación modernas	1,24	1,10	1,00	0,91	0,82	-
Utilización de herramientas hardware	1,24	1,10	1,00	0,91	0,83	-
Limitaciones de planificación del proyecto	1,23	1,08	1,00	1,04	1,10	-

Tabla 2. Conductores del coste del COCOMO

Para el cálculo del COCOMO intermedio se deben elegir entre **seis niveles** de valoración de dichos parámetros. Para este proyecto hemos escogido los niveles que aparecen en amarillo en la tabla, de manera que aplicando las fórmulas nos quedaría el siguiente **esfuerzo**, es decir, el número de personas que necesitaríamos para acabar el proyecto en un mes:

$$E_D = (1{,}15 \cdot 0{,}94 \cdot 1{,}15 \cdot 1 \cdot 1 \cdot 0{,}87 \cdot 0{,}87 \cdot 1{,}19 \cdot 1 \cdot 0{,}86 \cdot 0{,}90 \cdot 1 \cdot 1 \cdot 0{,}91 \cdot 1) \cdot 3{,}20 \cdot (7\ KLDC)^{1{,}05} = 19{,}471\ p/mes$$

Además, vamos a calcular ahora el **tiempo de desarrollo** promedio que nos indica el COCOMO:

$$T_D = 2{,}5 \cdot (19{,}471\ p/mes)^{0{,}38} = 7{,}72\ meses$$

Y el **número de personas** que necesitaríamos para completar el proyecto en el tiempo de desarrollo que nos indican:

$$Nº\ Personas = \frac{19{,}471\ p/mes}{7{,}72\ meses} = 2{,}52 \simeq 3\ personas$$

Por último, vamos a calcular la **productividad** medida en las líneas de código al mes que tendría que escribir cada persona para completar el proyecto en un mes:

$$PR = \frac{7.000\ LDC}{19{,}471\ p/mes} = 359{,}5\ LDC$$

Puesto que el proyecto únicamente lo va a llevar a cabo una persona tenemos que el tiempo de desarrollo en realidad sería el esfuerzo:

$$T_{Real} = \frac{19{,}471\ p/mes}{1\ persona} = 19{,}471\ meses$$

No obstante, no disponemos de tanto tiempo, y queremos realizar el desarrollo con una duración de en torno a 10 meses, con lo cual la productividad debería de ser de:

$$PR_{Real} = \frac{7.000\ LDC}{10\ meses} = 700\ LDC/mes$$

1.13 Planificación temporal

Para la planificación temporal del proyecto vamos a utilizar un **diagrama de Gantt**. Se trata de un gráfico de barras horizontales que se usa para ilustrar el cronograma de un proyecto, programa o trabajo. Es una forma de visualizar la programación de un proyecto, de dar seguimiento a los logros y de estar siempre familiarizado con el cronograma del trabajo. Cada barra de un diagrama de Gantt representa una etapa del proceso (o una tarea del proyecto) y su longitud, la duración de la tarea (J. Martins, 2022).

En concreto, nuestro proyecto va a estar dividido en **análisis**, **diseño**, **desarrollo**, **pruebas** y **documentación**. El desarrollo a su vez estará dividido en las iteraciones previstas.

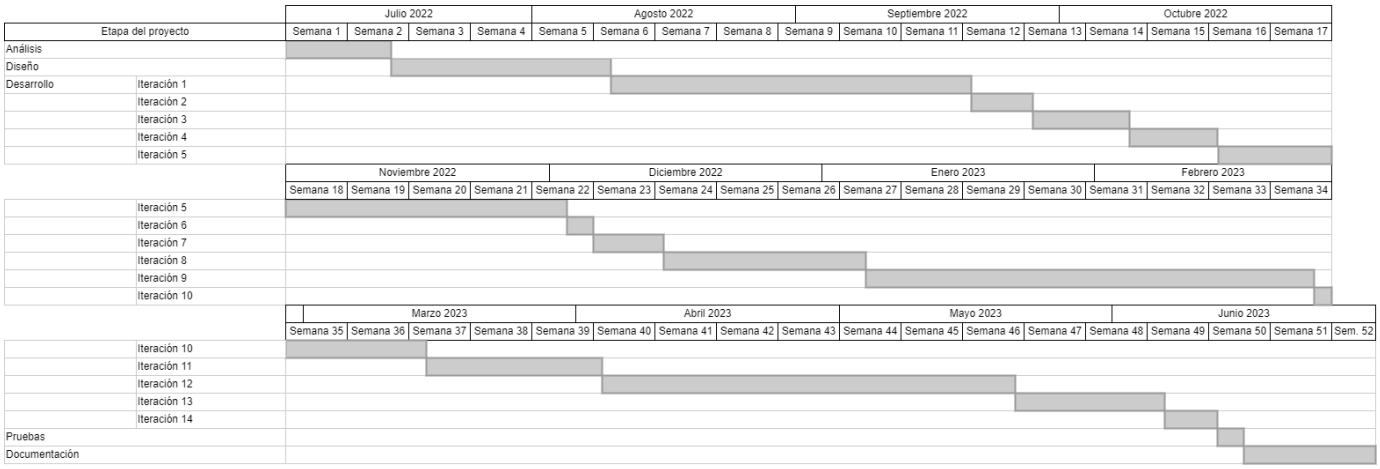


Ilustración 21. Diagrama de Gantt

1.14 Presupuesto

Para realizar una estimación adecuada del coste del proyecto hemos dividido los gastos en tres categorías: **recursos humanos**, **recursos software** y **recursos hardware**.

El proyecto va a ser íntegramente realizado por una única persona. Para conocer cuál podría ser su salario se ha consultado el XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública. Puesto que soy, a efectos prácticos, programador júnior, correspondería un salario anual de 14.800,66 €.

Recurso humanos	Salario anual (€)	Salario por horas (€ / hora)	Tiempo de proyecto (horas)	Coste Total (€)
Programador Júnior	14.800,66	7	300	1.800,00

Tabla 3. Desglose del coste de los recursos humanos

Para los recursos hardware y software se va a considerar una amortización en 5 años. En cuanto a los segundos, en realidad el único software no gratuito que se ha usado es *Word* para la elaboración de la documentación. No obstante, es posible que se necesitase invertir cierto dinero en paquetes de *assets* de *Unity*. Por tanto, vamos a hacer una estimación aproximada de unos 150 € para ello.

Recurso software	Precio (€)	Amortización por hora (€ / hora)	Tiempo de proyecto (horas)	Amortización Total (€)
Microsoft Word	149,00	0,01	300	3,10
Unity	0,00	0,00	300	0,00
Visual Studio 2019	0,00	0,00	300	0,00
Draw.io	0,00	0,00	300	0,00
Unity Assets	150,00	0,01	300	3,12
Coste Total: 299,00 €				
Total Amortizado: 6,22 €				

Tabla 4. Desglose del coste y amortización de los recursos software

En cuanto los recursos hardware, fundamentalmente hemos dispuesto para el proyecto de un ordenador personal para realizar el grosso del proyecto y un *ipad* con lápiz digital para llevar a cabo ciertos dibujos.

Recurso hardware	Precio (€)	Amortización por hora (€ / hora)	Tiempo de proyecto (horas)	Amortización Total (€)
Ordenador	700,00	0,05	300	14,59
Ipad	400,00	0,03	300	8,33
Coste Total: 1100,00 €				
Total Amortizado: 22,92 €				

Tabla 5. Desglose del coste y amortización de los recursos hardware

Para los sobrecostes indirectos (luz, agua, gas…) vamos a utilizar un 10% del total del coste.

Coste recursos humanos (€)	Coste recursos software (€)	Coste recursos hardware (€)	Costes indirectos (€)
1800,00	299,00	1100,00	319,90
Coste Total: 3518,90 €			

Tabla 6. Desglose del coste total del proyecto

2 DISEÑO INICIAL

2.1 Especificaciones del sistema

El desarrollo de un videojuego es, en ciertos aspectos, distinto del de otros productos software, sobre todo en la etapa del desarrollo correspondiente a la ingeniería de requisitos, específicamente en la etapa de **obtención de requisitos**, ya que no hay normalmente ningún cliente al que debamos de preguntar las funcionalidades deseadas de nuestro producto. En su lugar, es el equipo creativo el que debe encontrar ideas para especificar cómo debe de ser el videojuego. Además, los requisitos en el caso de un videojuego se corresponderían, más o menos, con las **mecánicas** que debe implementar. Por todo ello, la mejor estrategia de obtención de requisitos que se podría utilizar es un *brainstorming*.

A continuación, vamos a empezar proponiendo una serie de mecánicas de juego comunes en los RPG de acción que más adelante nos ayudarán para extraer los requisitos funcionales y no funcionales.

2.1.1 Mecánicas de juego

Para empezar el diseño de nuestro videojuego, lo primero será pensar en las **mecánicas** que queremos incluir en él, provenientes de otros RPGs de acción. Las mecánicas, como ya hemos visto aquí, son fundamentalmente las acciones que el jugador puede realizar y la lógica que altera el estado del videojuego de alguna forma de acuerdo a estas acciones. De esta forma, atendiendo a esa definición, nos quedan las siguientes mecánicas, catalogadas además siguiendo una determinada clasificación (E. Adams, 2013):

- **Mecánicas físicas.** Se trata de aquellas que involucran el comportamiento físico de los objetos, su movimiento, posición, etc… Dentro de estas nos encontramos las siguientes mecánicas:
  - Que el jugador sea capaz de **mover** al personaje en las diferentes direcciones del plano, pues debe estar sujeto al suelo por la gravedad.
  - Que haya una **cámara** que siga en todo momento al personaje y que el jugador pueda mover a su gusto.

- Que el jugador sea capaz de atacar con una **espada** al enemigo y hacerle daño.
- Que el jugador sea capaz de defenderse con un **escudo** de los ataques del enemigo.
- Que el jugador sea capaz de **esquivar** los ataques del enemigo en cualquier dirección del plano.
- Que el jugador pueda **correr**, es decir, desplazarse más deprisa.
- Que el jugador sea capaz de usar un **arco** para atacar a los enemigos.
- Que el jugador sea capaz de usar **bombas** para atacar a los enemigos.
- Que el jugador sea capaz de **fijar** a un enemigo, de forma que la cámara siempre esté apuntando hacia él, y que pueda desfijar cuando quiera.

- **Mecánicas de economía interna.** Mecánicas relacionadas con la creación, consumo e intercambio de recursos cuantitativos como el dinero, la vida, etc… En nuestro proyecto nos encontramos con los siguientes:
  - Que, al **sufrir daño**, la vida del jugador baje.
  - Que, si la vida del jugador llegue a 0, **muera**.
  - Que pueda **recuperar vida** por medio de determinados objetos.
  - Que haya una serie de **recursos** que pueda conseguir a lo largo de la partida.
  - Que tenga un número limitado de **flechas** y que gaste una cada vez que usa el arco.
  - Que tenga un número limitado de **bombas** y que gaste una cada vez que use una bomba.
  - Que tenga un número determinado de **llaves** y que gaste un número determinado de ellas al abrir una puerta (para las islas).
  - Que pueda **conseguir flechas** que se vaya encontrando.

  - Que pueda **conseguir bombas** que se vaya encontrando.
  - Que pueda **conseguir llaves** que se vaya encontrando (para las islas).
  - Que, al **sufrir daño**, la vida de los enemigos baje.
  - Que, si la vida del enemigo llega a 0, **muera**.
  - Que, al morir un enemigo, pueda **soltar** algún tipo de recurso.

- **Mecánicas de progresión.** Aquellas relacionadas con el progreso del jugador a través de una serie de desafíos. Podemos enumerar las siguientes:
  - Que pueda abrir **puertas** con un cierto número de llaves (para las islas).
  - Que pueda **ganar** el juego consiguiendo una corona.
  - Que pueda **teletransportarse** en determinados puntos (para las islas).
  - Que haya un **boss** al que tenga que derrotar al final de la partida para poder ganarla.
  - Que el jugador pueda **pausar** el juego cuando quiera.
- **Mecánicas de táctica.** Son aquellas que determinan la táctica del jugador, esto es, la manera a la que se enfrenta a los desafíos. Un ejemplo sería hacer que, en un juego de estrategia, el jugador tuviera menos probabilidades de acertar a un enemigo cubierto. Vamos a ver cuáles tendríamos en nuestro caso:
  - Que el jugador sea **invencible** mientras está esquivando, esto es, que no pueda recibir daño.
  - Que tampoco pueda recibir daño cuando está con el **escudo** en alto.
  - Que los recursos que dejan los enemigos al morir lo hagan con una determinada **probabilidad**.
  - Que haya unos **niveles** de la mazmorra con enemigos y otras con objetos atendiendo a algún criterio.

- Que el personaje pueda **cambiar de arma** cuando el jugador desee.

El otro tipo de mecánicas son las de **interacción social**, pero no aplican en nuestro caso.

2.1.2 Requisitos funcionales

Los **requisitos funcionales** son declaraciones de los servicios que prestará el sistema, esto es, la forma en que reaccionará a determinadas entradas; dicho de otra forma, el **qué hace** la aplicación. En este sentido, son más o menos (pero con varias excepciones) análogos a las mecánicas de juego en el caso particular del desarrollo de un videojuego. Vamos a enumerarlos:

1. **Empezar juego.** El sistema debe permitir al jugador empezar una partida, seleccionando previamente el mapa en el que quiere jugar.
2. **Ajustar volumen.** El sistema debería permitir al jugador ajustar el volumen del audio del juego.
3. **Seleccionar resolución.** El sistema debería permitir al jugador seleccionar la resolución de pantalla.
4. **Seleccionar nivel de gráficos.** El sistema debería permitir al jugador ajustar las opciones gráficas de alguna forma.
5. **Seleccionar pantalla completa.** El jugador debería ser capaz de elegir entre pantalla completa o ventana.
6. **Salir del videojuego.** El jugador debería ser capaz de salir del juego en cualquier momento.
7. **Mover cámara.** El jugador debe ser capaz de mover la cámara de la forma que desee, de manera que siempre esté orbitando alrededor del personaje principal.
8. **Mover personaje.** El jugador debe ser capaz de mover al personaje en la dirección del plano que desee.
9. **Hacer que el personaje ruede.** El sistema debe permitir al jugador hacer que su personaje ruede cuando desee y en la dirección del plano que más crea conveniente.

10. **Hacer que el personaje corra.** El jugador debe ser capaz de hacer que su personaje corra cuando quiera, esto es, que se desplace más rápidamente.
11. **Atacar con la espada.** El personaje debe ser capaz de atacar con la espada cuando el jugador desee.
12. **Defenderse con escudo.** El jugador debe tener la posibilidad de defenderse de ataques enemigos con el escudo.
13. **Atacar con el arco.** El jugador debe poder atacar con el arco usando flechas a los enemigos.
14. **Atacar con bombas.** El personaje del jugador debe ser capaz de lanzar bombas a los enemigos.
15. **Fijar enemigos.** El sistema debería permitir al jugador hacer que la cámara apunte al enemigo en lugar de al jugador.
16. **Cambiar arma.** El sistema debe permitir al jugador cambiar entre espada/escudo y arco.
17. **Conseguir recursos.** El jugador debe ser capaz de recoger recursos del escenario.
18. **Pausar el juego.** El sistema debe permitir al jugador pausar el juego en cualquier momento.
19. **Seguir jugador.** Los enemigos deben ser capaces de seguir al jugador.
20. **Ataque enemigo.** Los enemigos deben ser capaces de atacar al jugador.
21. **Morir.** El jugador debe morir al bajar su vida a 0, lo cual produce la salida del videojuego y la pérdida del progreso.
22. **Recuperar vida.** El jugador debe ser capaz de recuperar vida al recoger un cierto recurso del escenario.
23. **Matar enemigos.** El sistema debe hacer que los enemigos mueran una vez su vida llega a 0, dejando caer un recurso con una cierta probabilidad.
24. **Abrir puertas.** El sistema debe hacer que una puerta se abra si el jugador tiene las llaves necesarias.

25. **Ganar la partida.** El sistema debe hacer que el jugador gane la partida si consigue una corona.
26. **Teletransportarse.** El sistema debe teletransportar al personaje del jugador en determinadas casuísticas.
27. **Emplazamiento de los enemigos.** El sistema debería emplazar a los enemigos atendiendo a algún criterio.
28. **Emplazamiento de los recursos.** Los recursos deberían estar situados a lo largo de la mazmorra atendiendo a algún criterio.
29. **Emplazamiento del escenario.** Los diferentes niveles deberían estar situados y conectados siguiendo algún criterio.

2.1.3 Requisitos no funcionales

Si los requisitos funcionales son el qué, los no funcionales son el **cómo**; requisitos que no se refieren a las funciones del sistema sino a sus propiedades: rendimiento, seguridad, disponibilidad… Vamos a ver cuáles serían en nuestro caso:

1. **Buen rendimiento.** El juego debería tener un consumo de recursos relativamente bajo para que la experiencia de juego sea lo mejor posible.
2. **Buena sensación de juego.** Las acciones del personaje deberían ser lo más fluidas posibles para no frustrar al jugador.
3. **Interfaz simple e intuitiva.** El juego debería tener una interfaz fácil de interpretar para no causarle problemas al jugador.
4. **Uso de menús.** Para tareas como iniciar el juego, pausarlo o cambiar la configuración del mismo se deberán usar menús.
5. **Tiempos de carga relativamente cortos.** Los procesos de cálculo deberían estar escondidos detrás de pantallas de carga y estas no ser excesivamente largas.
6. **Diseño de niveles coherente.** Fundamentalmente, que la mazmorra se pueda recorrer de principio a fin de manera que el jugador no se quede bloqueado sin poder terminar la partida.
7. **Variedad de enemigos.** Que haya diversos enemigos con ataques diferentes, unos más díficiles que otros.

8. **Variedad de escenarios.** Que los escenarios en los que se desarrolla el juego sean lo más diversos posibles.

2.2 Análisis y diseño del sistema

Una vez pasada la etapa de obtención de requisitos nos adentramos en la etapa de análisis y diseño de nuestro software. En este apartado ahondaremos en el diseño y el análisis de nuestro proyecto por medio de herramientas como los casos de uso o los diagramas de actividad, así como explicaciones detalladas del funcionamiento de ciertos algoritmos.

Vamos a dividir el análisis y diseño en tres partes: análisis y diseño de la **jugabilidad**, análisis y diseño de la **generación procedural**, y análisis y diseño de la **interfaz**.

2.2.1 Análisis y diseño de la jugabilidad

2.2.1.1 Casos de uso

Los casos de uso describen bajo la forma de **acciones** y **reacciones** el comportamiento de un sistema desde el **punto de vista del usuario**. Al igual que los requisitos, describen las funcionalidades del sistema independientemente de la implementación, pero a un mayor nivel de detalle; a un nivel más bajo de abstracción, si se quiere.

A continuación, se presenta los casos de uso que se han identificado en este proyecto, de forma que se van a usar las tablas clásicas de casos de uso:

Caso de uso	**Mover cámara**
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere mover la cámara de una determina forma. 2) El sistema comprueba si la cámara no está fijada en un enemigo. 2) El sistema hace que la cámara se mueva de acuerdo al *input*.

Alternativas	2ª) ¿La cámara estaba fijada? 2ª.1) Si no lo estaba, se sigue. 2ª.2) Si lo estaba, se sale.

Tabla 7. Caso de uso (Mover cámara)

Caso de uso	Mover personaje
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere mover al personaje de una determinada forma. 2) El sistema comprueba si el jugador está atacando. 3) El sistema comprueba si el jugador está rodando. 4) El sistema comprueba si el jugador está bloqueando. 5) El sistema comprueba si el personaje está siendo golpeado. 6) El sistema comprueba si el jugador está fijando. 7) El jugador se mueve de acuerdo al *input*.
Alternativas	2ª) ¿El jugador está atacando? 2ª.1) Si sí, se sale. 2ª.2) Si no, se sigue. 3ª) ¿El jugador está rodando? 3ª.1) Si sí, se sale. 3ª.2) Si no, se sigue. 4ª) ¿El jugador está bloqueando? 4ª.1) Si sí, se sale.

	4ª.2) Si no, se sigue. 5ª) ¿El personaje está siendo golpeado? 5ª.1) Si sí, se sale. 5ª.2) Si no, se sigue. 6ª) ¿El jugador está fijando? 6ª.1) Si sí, el movimiento se modificará de forma que el jugador se mueva en un círculo alrededor del enemigo y se sigue. 6ª.2) Si no, se sigue.

Tabla 8. Caso de uso (Mover personaje)

Caso de uso	Rodar
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere hacer que el personaje ruede. 2) El sistema comprueba si el jugador ya está rodando. 3) El sistema comprueba si el jugador está siendo golpeado. 4) El sistema comprueba si el jugador está fijando. 5) El jugador rueda de acuerdo al *input*.
Alternativas	2ª) ¿El jugador está rodando? 2ª.1) Si sí, se sale. 2ª.2) Si no, se sigue. 3ª) ¿El jugador está siendo golpeado? 3ª.1) Si sí, se sale. 3ª.2) Si no, se sigue.

	4ª) ¿El jugador está fijando? 4ª.1) Si sí, el movimiento se modificará de forma que el jugador se mueva en un círculo alrededor del enemigo y se sigue. 4ª.2) Si no, se sigue.

Tabla 9. Caso de uso (Rodar)

Caso de uso	Correr
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú y que tenga seleccionada como arma la espada
Operaciones básicas	1) El jugador le comunica al sistema que quiere hacer que el personaje corra. 2) El sistema comprueba si el jugador se está moviendo. 4) El jugador corre de acuerdo a cómo se está moviendo.
Alternativas	2ª) ¿El jugador está moviéndose? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale.

Tabla 10. Caso de uso (Correr)

Caso de uso	Atacar con la espada
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere atacar con la espada. 2) El sistema comprueba que tenga seleccionada la espada como arma.

	3) El sistema comprueba si el jugador ya está atacando. 4) El sistema comprueba si el jugador está rodando. 5) El sistema comprueba si el personaje está siendo golpeado. 6) El jugador ataca con la espada.
Alternativas	2ª) ¿El jugador tiene la espada seleccionada como arma? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale. 3ª) ¿El jugador ya está atacando? 3ª.1) Si sí, se sale. 3ª.2) Si no, se sigue. 4ª) ¿El jugador está rodando? 4ª.1) Si sí, se sale. 4ª.2) Si no, se sigue. 5ª) ¿El personaje está siendo golpeado? 5ª.1) Si sí, se sale. 5ª.2) Si no, se sigue.

Tabla 11. Caso de uso (Atacar con la espada)

Caso de uso	Alzar el escudo
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere alzar el escudo. 2) El sistema comprueba que tenga espada y escudo seleccionadas como armas.

	3) El sistema comprueba si el jugador está rodando. 4) El sistema comprueba si el personaje está siendo golpeado. 5) El sistema comprueba si está atacando con la espada. 6) El jugador alza el escudo.
Alternativas	2ª) ¿El jugador tiene seleccionadas como arma la espada y el escudo? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale. 3ª) ¿El jugador está rodando? 3ª.1) Si sí, se sale. 3ª.2) Si no, se sigue. 4ª) ¿El personaje está siendo golpeado? 4ª.1) Si sí, se sale. 4ª.2) Si no, se sigue. 5ª) ¿El personaje está atacando con la espada? 5ª.1) Si sí, se sale. 5ª.2) Si no, se sigue.

Tabla 12. Caso de uso (Alzar el escudo)

Caso de uso	Atacar con el arco
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú y que tenga el arco seleccionado como arma
Operaciones básicas	1) El jugador le comunica al sistema que quiere atacar con el arco.

	2) El sistema comprueba si el jugador ya está atacando. 3) El sistema comprueba si el jugador está rodando. 4) El sistema comprueba si el personaje está siendo golpeado. 5) El sistema comprueba si el jugador posee flechas. 6) El jugador ataca con el arco.
Alternativas	2ª) ¿El jugador ya está atacando? 2ª.1) Si sí, se sale. 2ª.2) Si no, se sigue. 3ª) ¿El jugador está rodando? 3ª.1) Si sí, se sale. 3ª.2) Si no, se sigue. 4ª) ¿El personaje está siendo golpeado? 4ª.1) Si sí, se sale. 4ª.2) Si no, se sigue. 5ª) ¿El personaje tiene flechas? 5ª.1) Si sí, se sigue y gasta una. 5ª.2) Si no, se sale.

Tabla 13. Caso de uso (Atacar con el arco)

Caso de uso	Lanzar bomba
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere lanzar una bomba.

	2) El sistema comprueba si el jugador tiene bombas disponibles. 3) El sistema hace que el personaje del jugador lance una bomba.
Alternativas	2ª) ¿El jugador tiene bombas? 2ª.1) Si sí, se sigue y descuenta una. 2ª.2) Si no, se sale.

Tabla 14. Caso de uso (Lanzar bomba)

Caso de uso	**Fijar enemigo**
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere fijar a un enemigo. 2) El sistema comprueba si hay algún enemigo visible y dentro del rango. 3) El sistema comprueba si el jugador ya tenía fijado algún enemigo. 4) El sistema hace que la cámara fije a un enemigo
Alternativas	2ª) ¿Hay un enemigo visible por el jugador y dentro del rango? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale. 3ª) ¿El jugador tenía ya fijado un enemigo? 3ª.1) Si sí, se desfija de forma que la cámara vuelve a funcionar con normalidad y se sale. 3ª.2) Si no, se sigue.

Tabla 15. Caso de uso (Fijar enemigo)

Caso de uso	**Cambiar arma**

Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El jugador le comunica al sistema que quiere cambiar de arma. 2) El sistema comprueba si posee tal arma. 3) El sistema hace que cambie de arma.
Alternativas	2ª) ¿El jugador posee esa arma? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale.

Tabla 16. Caso de uso (Cambiar arma)

Caso de uso	**Conseguir recursos**
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en un menú
Operaciones básicas	1) El sistema comprueba si el jugador está dentro del rango para conseguir un recurso. 2) El jugador consigue ese recurso.
Alternativas	1ª) ¿El jugador está lo suficientemente cerca del recurso? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale.

Tabla 17. Caso de uso (Conseguir recursos)

Caso de uso	**Pausar el juego**
Actor	Jugador
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida
Operaciones básicas	1) El jugador comunica al sistema que desea pausar el juego.

	2) El sistema comprueba si está en el menú de pausa ya o no. 3) El sistema pausa el juego.
Alternativas	1ª) ¿El jugador se encuentra en el menú de pausa? 1ª.1) Si sí, se reanuda el juego y se sale. 1ª.2) Si no, se sigue.

Tabla 18. Caso de uso (Pausar el juego)

Caso de uso	Seguir jugador
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú
Operaciones básicas	1) El sistema comprueba si el jugador está dentro del radio de detección del enemigo. 2) El sistema comprueba si el enemigo tiene contacto visual directo con el jugador. 3) El enemigo empieza a perseguir al jugador.
Alternativas	1ª) ¿El jugador se encuentra en el radio de detección? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale. 2ª) ¿El enemigo tiene contacto visual directo? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale.

Tabla 19. Caso de uso (Seguir jugador)

Caso de uso	Ataque enemigo
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú

Operaciones básicas	1) El sistema comprueba si el jugador está dentro del radio de ataque del enemigo. 2) El enemigo ataca. 3) El sistema comprueba si el ataque enemigo ha colisionado contra el jugador. 4) El sistema comprueba si el ataque ha sido bloqueado con el escudo. 5) El sistema comprueba si el ataque había colisionado mientras el jugador rodaba. 6) El jugador es golpeado y recibe daño.
Alternativas	1ª) ¿El jugador está dentro del radio de ataque? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale. 3ª) ¿El ataque ha impactado? 3ª.1) Si sí, se sigue. 3ª.2) Si no, se sale. 4ª) ¿El ataque ha sido bloqueado? 4ª.1) Si sí, se sale. 4ª.2) Si no, se sigue. 5ª) ¿El ataque ha impactado durante la esquiva del jugador? 5ª.1) Si sí, se sale. 5ª.2) Si no, se sigue.

Tabla 20. Caso de uso (Sufrir daño)

Caso de uso	Morir
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú

Operaciones básicas	1) El sistema comprueba si la vida del jugador ha llegado a 0. 2) El sistema mata al personaje y lo señala de la forma adecuada.
Alternativas	1ª) ¿El jugador tiene a 0 la vida? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale.

Tabla 21. Caso de uso (Morir)

Caso de uso	Recuperar vida
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú
Operaciones básicas	1) El sistema comprueba si el jugador está lo bastante cerca del recurso adecuado. 2) El sistema comprueba si la vida del jugador no está al máximo. 3) El sistema hace que el jugador recupere la vida correspondiente.
Alternativas	1ª) ¿El jugador está lo suficientemente cerca del recurso? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale. 1ª) ¿El jugador tiene al máximo la vida? 1ª.1) Si sí, se sale. 1ª.2) Si no, se sigue.

Tabla 22. Caso de uso (Recuperar vida)

Caso de uso	Matar enemigo
Actor	Sistema
Contexto	Durante la partida

Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú
Operaciones básicas	1) El sistema comprueba si la vida del enemigo ha llegado a 0. 2) El sistema mata al enemigo y lo señala de la forma adecuada. 3) El sistema comprueba la probabilidad de hacer aparecer un recurso. 4) El sistema hace aparecer un recurso.
Alternativas	1ª) ¿El enemigo tiene a 0 la vida? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale. 2ª) ¿La probabilidad de hacer aparecer un recurso se cumple? 2ª.1) Si sí, se sigue. 2ª.2) Si no, se sale.

Tabla 23. Caso de uso (Matar enemigo)

Caso de uso	**Abrir puerta**
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú
Operaciones básicas	1) El sistema comprueba si el jugador está lo suficientemente cerca de la puerta. 2) El sistema comprueba si el jugador tiene las llaves necesarias. 3) El sistema abre la puerta y quita el número correspondiente de llaves al jugador.
Alternativas	1ª) ¿El jugador está lo suficientemente cerca? 1ª.1) Si sí, se sigue.

	1ª.2) Si no, se sale. 1ª) ¿El jugador tiene las llaves necesarias? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale.

Tabla 24. Caso de uso (Abrir puerta)

Caso de uso	Ganar la partida
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú y que haya vencido al *final boss*
Operaciones básicas	1) El sistema comprueba si el jugador está lo suficientemente cerca del recurso necesario. 2) El sistema hace que el jugador gane mostrándolo adecuadamente.
Alternativas	1ª) ¿El jugador está lo suficientemente cerca del recurso? 1ª.1) Si sí, se sigue. 1ª.2) Si no, se sale.

Tabla 25. Caso de uso (Ganar la partida)

Caso de uso	Teletransportarse
Actor	Sistema
Contexto	Durante la partida
Condición previa	Que el jugador se encuentre jugando en la partida y no en ningún menú
Operaciones básicas	1) El sistema comprueba si el jugador está lo suficientemente cerca del teletransportador. 2) El sistema hace que el jugador se teletransporte a otro lado.
Alternativas	1ª) ¿El jugador está lo suficientemente cerca del teletransportador? 1ª.1) Si sí, se sigue.

	1ª.2) Si no, se sale.

Tabla 26. Caso de uso (Teletransportarse)

2.2.1.2 Diagramas de actividad

Los **diagramas de actividades** son una forma particular de **diagramas de máquinas de estados**, pero en lugar de modelar el ciclo de vida de los objetos modelan el ciclo de vida de una **tarea** o **función** de un sistema. En nuestro caso, resultan extremadamente útiles (a diferencia de otros diagramas propios del proceso unificado) porque contamos con muchas mecánicas con dependencias complejas que se visualizan fácilmente con estos diagramas.

A continuación, vamos a mostrar los diagramas de actividades obtenidos a partir de los casos de uso:

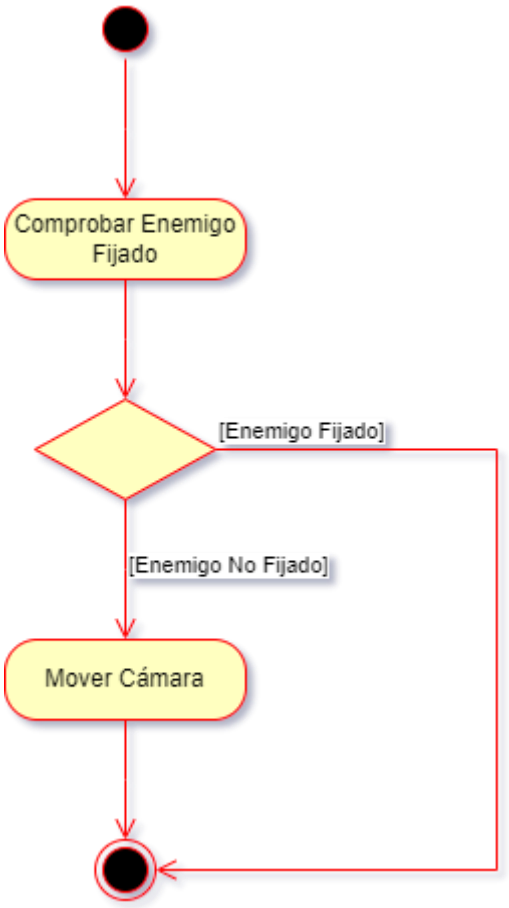


Ilustración 22. Diagrama de actividad (Mover Cámara)

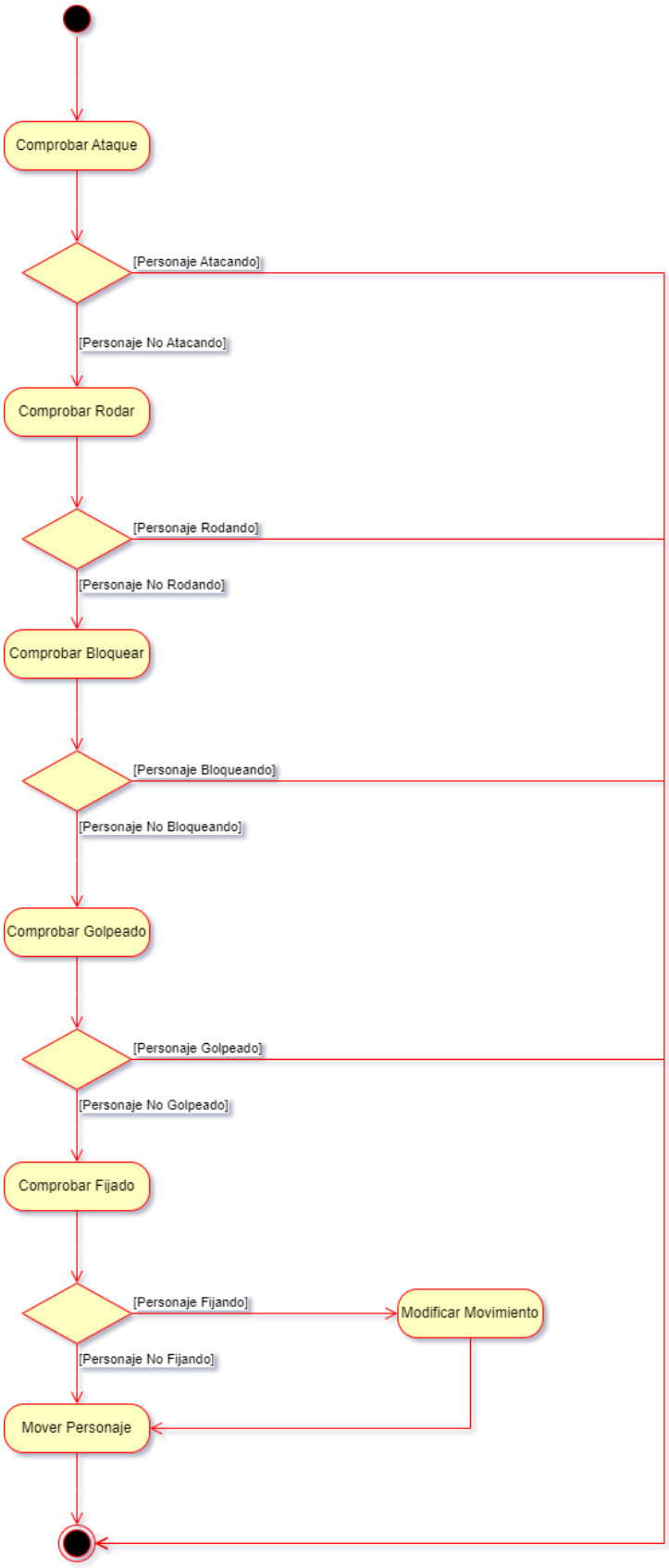


Ilustración 23. Diagrama de actividad (Mover personaje)

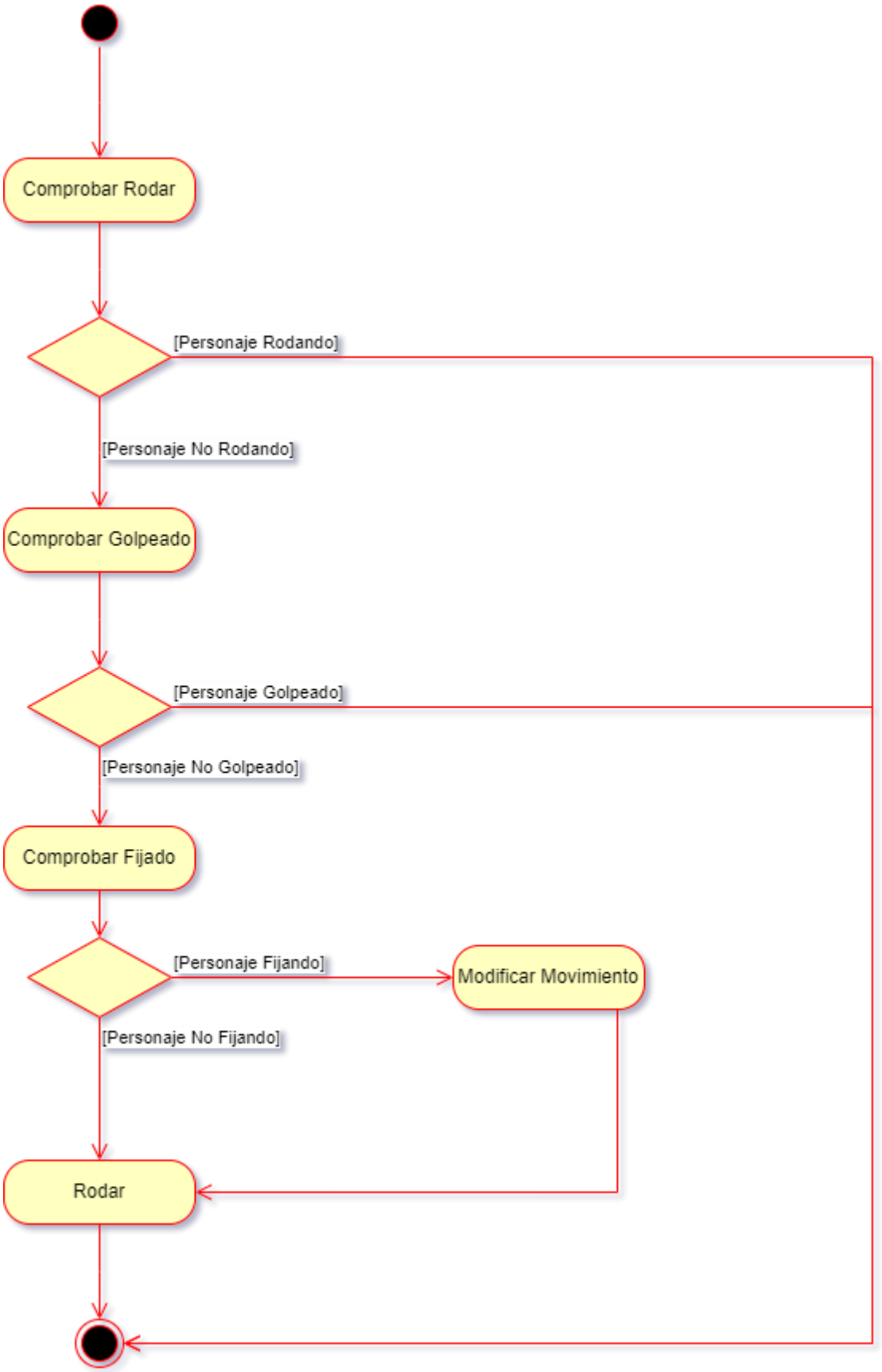


Ilustración 24. Diagrama de actividad (Rodar)

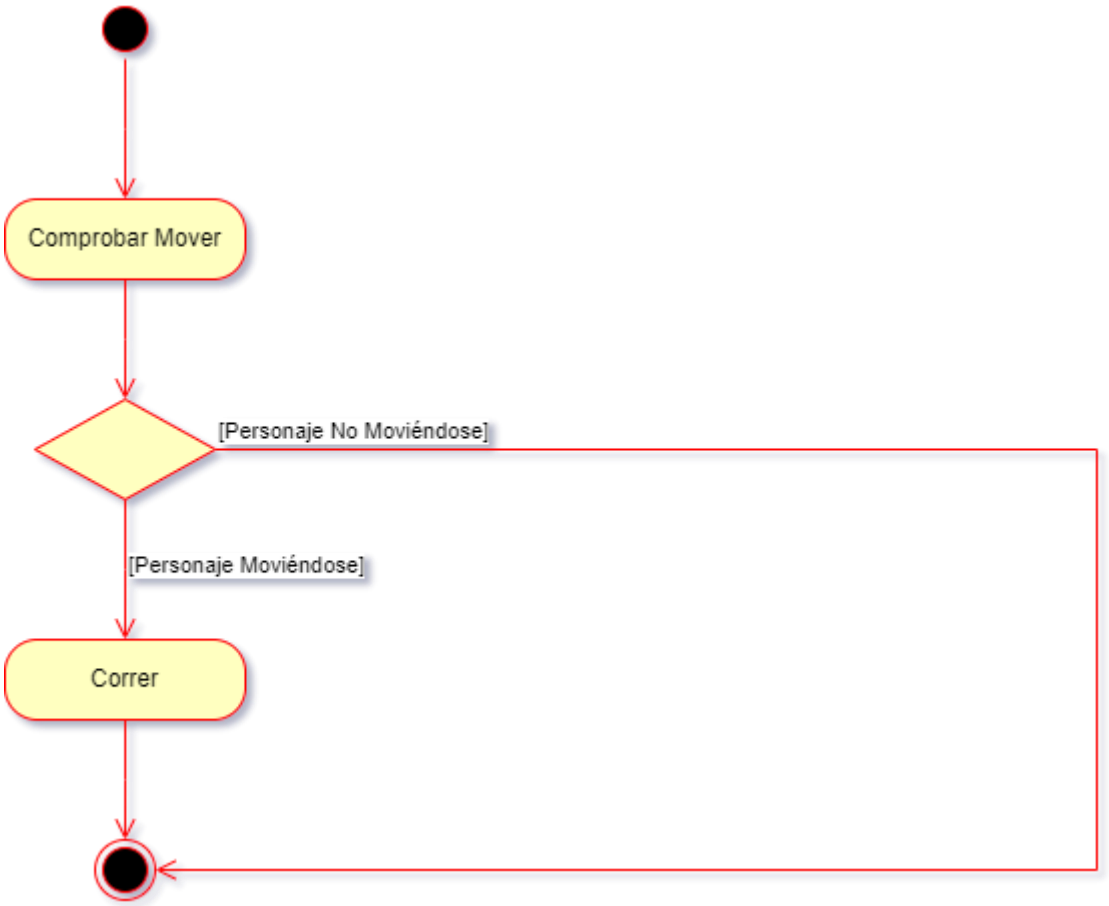


Ilustración 25. Diagrama de actividad (Correr)

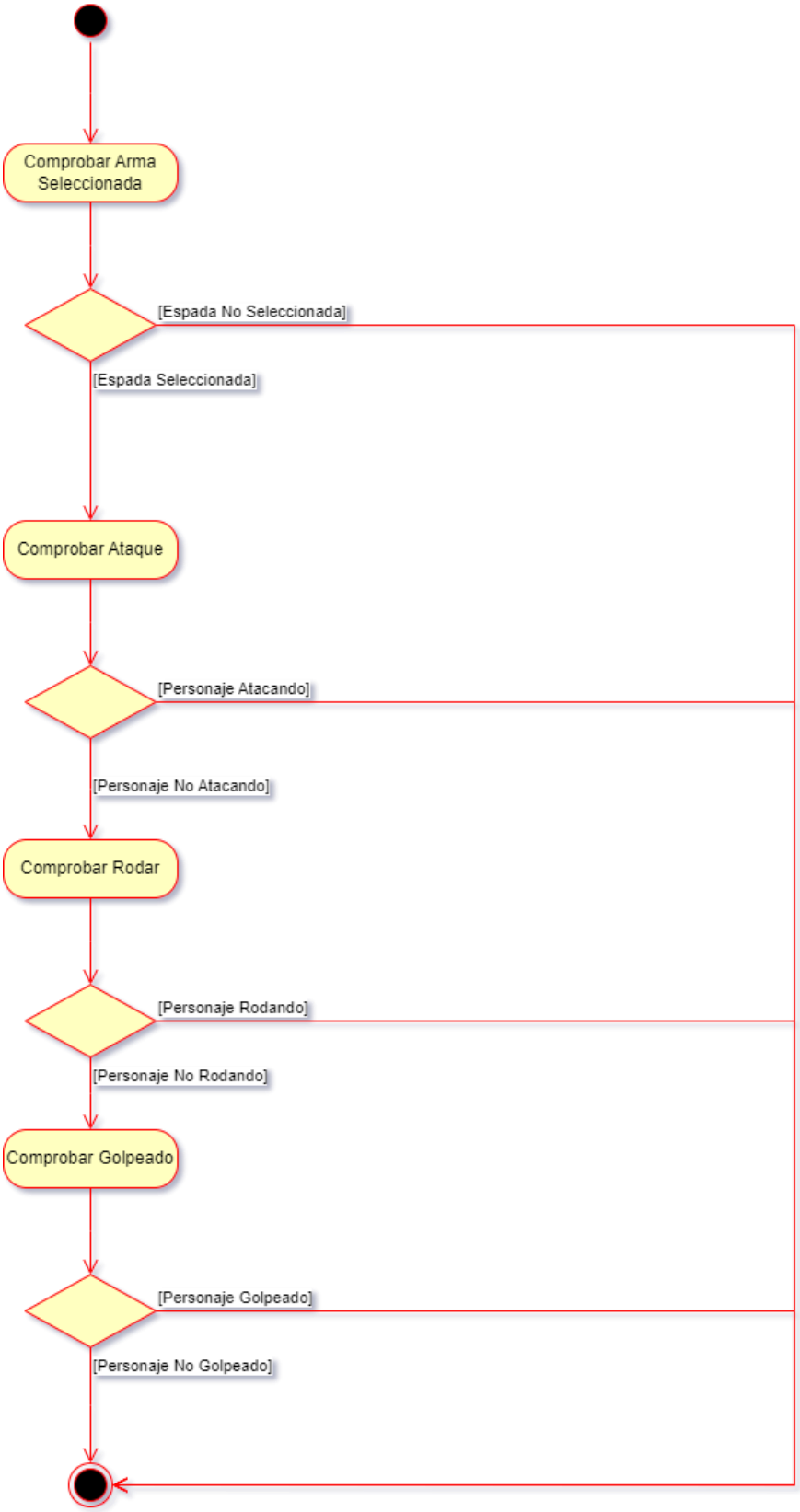


Ilustración 26. Diagrama de actividad (Ataque con la espada)

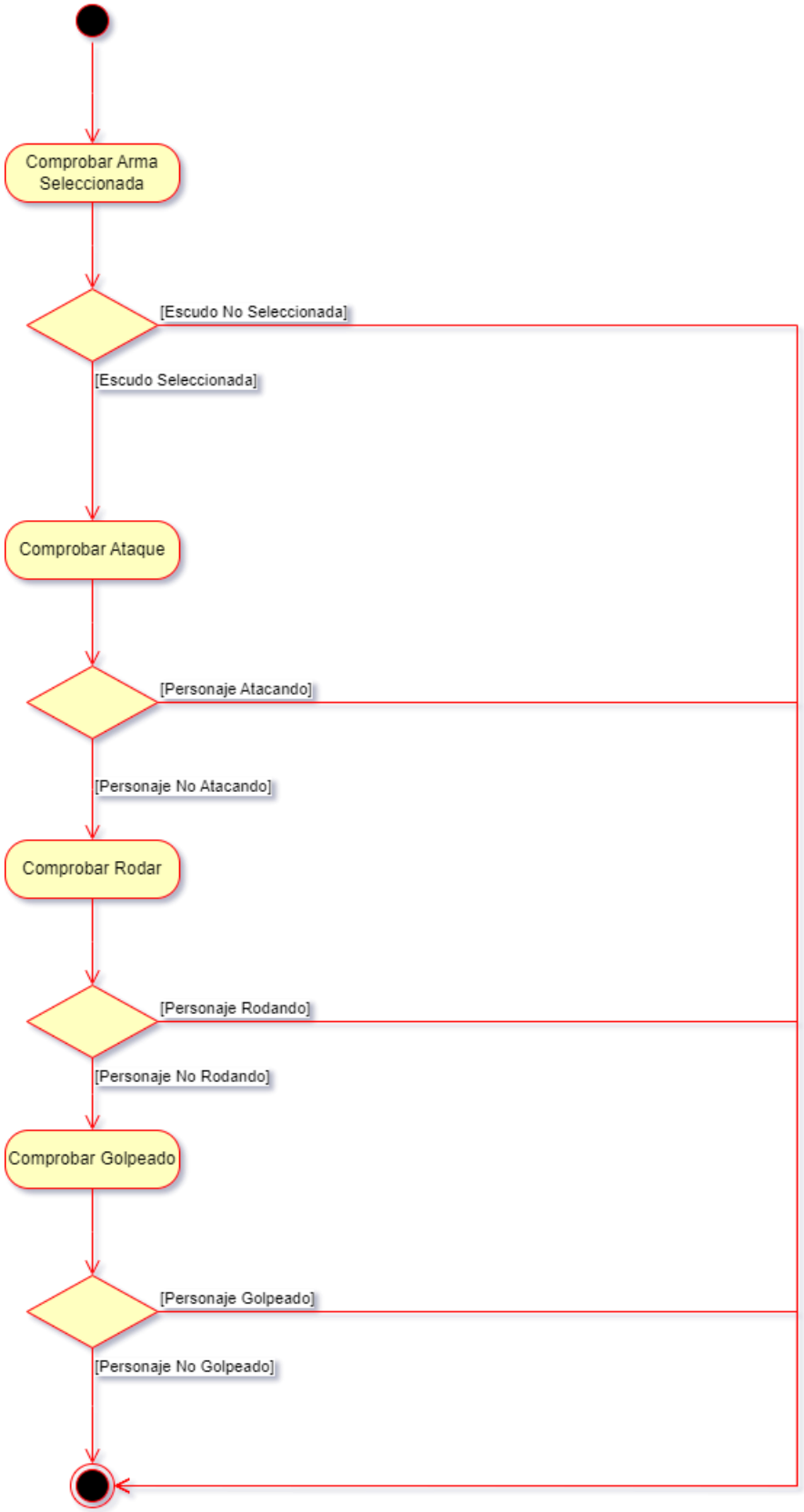


Ilustración 27. Diagrama de actividad (Alzar escudo)

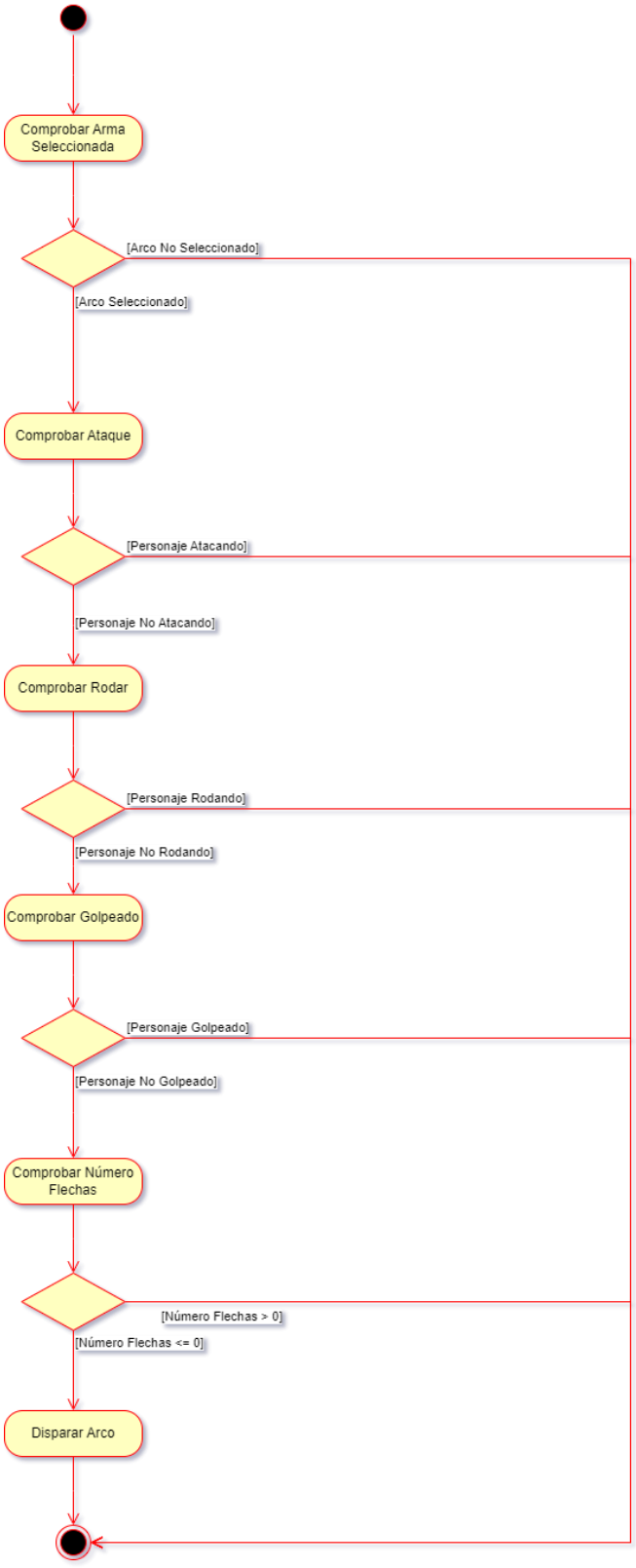


Ilustración 28. Diagrama de actividad (Atacar con arco)

Nota: La condición del número de flechas tiene una errata, evidentemente es al revés.

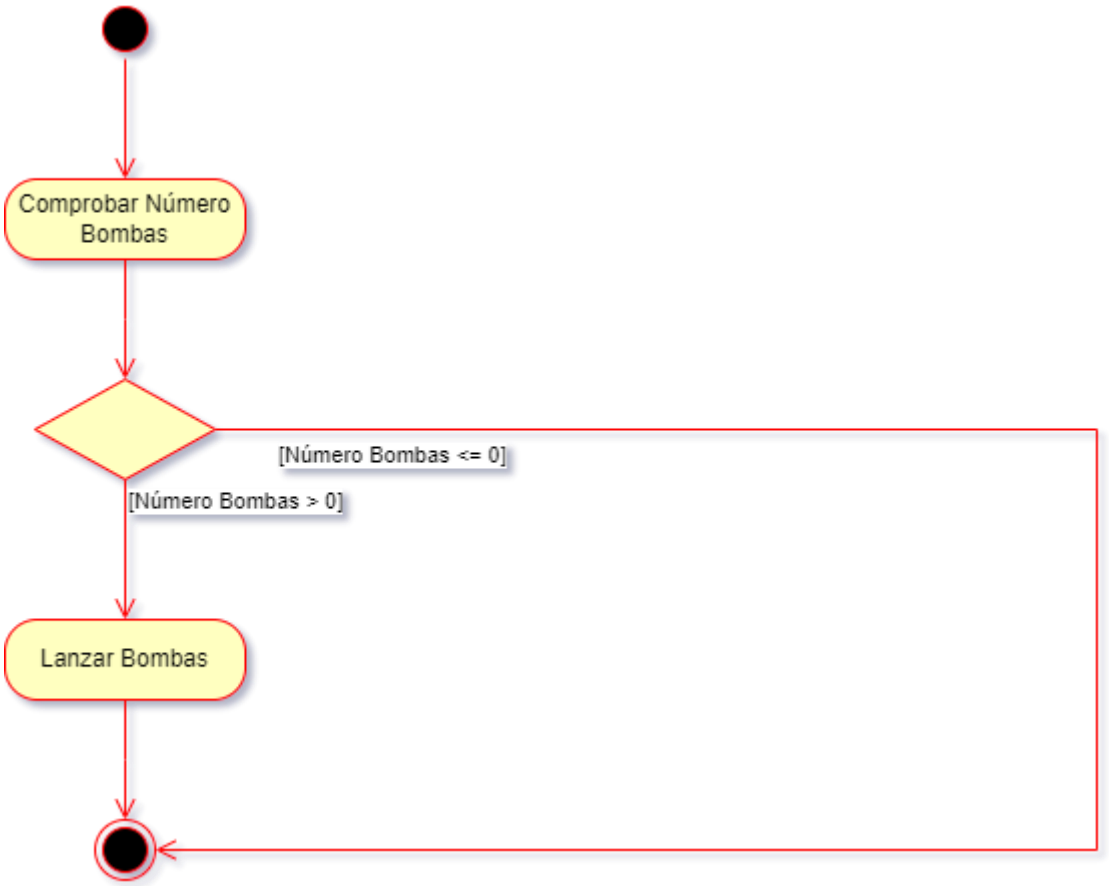


Ilustración 29. Diagrama de actividad (Lanzar bomba)

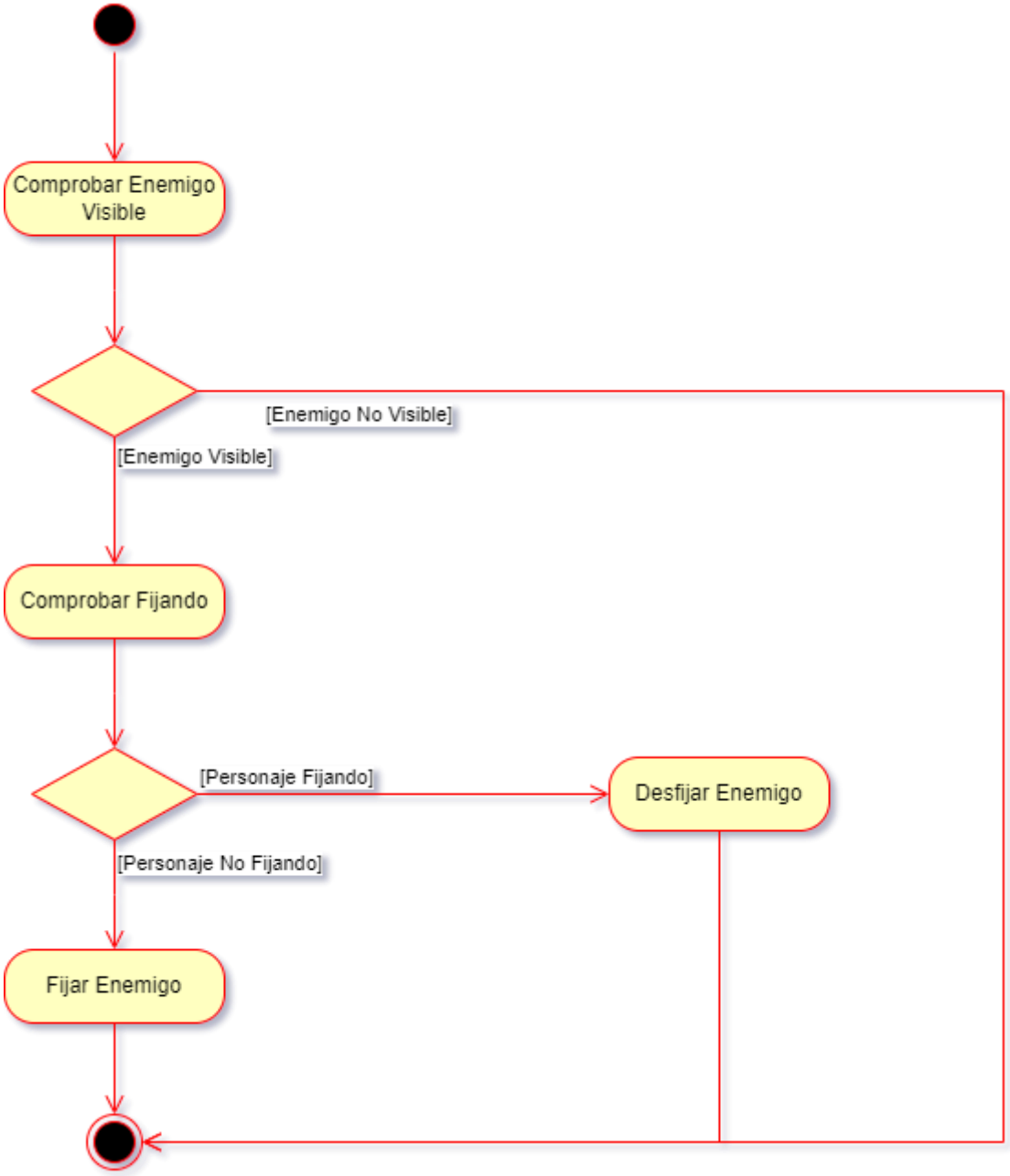


Ilustración 30. Diagrama de actividad (Fijar enemigo)

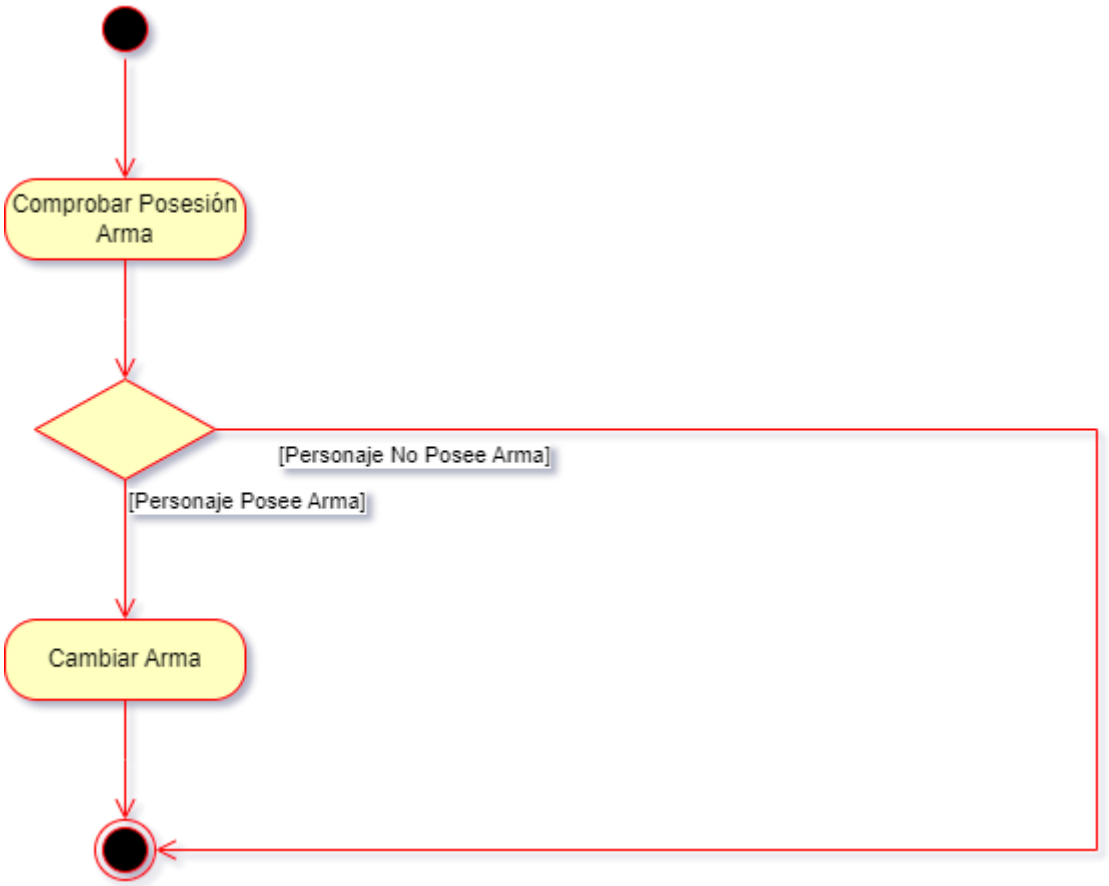


Ilustración 31. Diagrama de actividad (Cambiar arma)

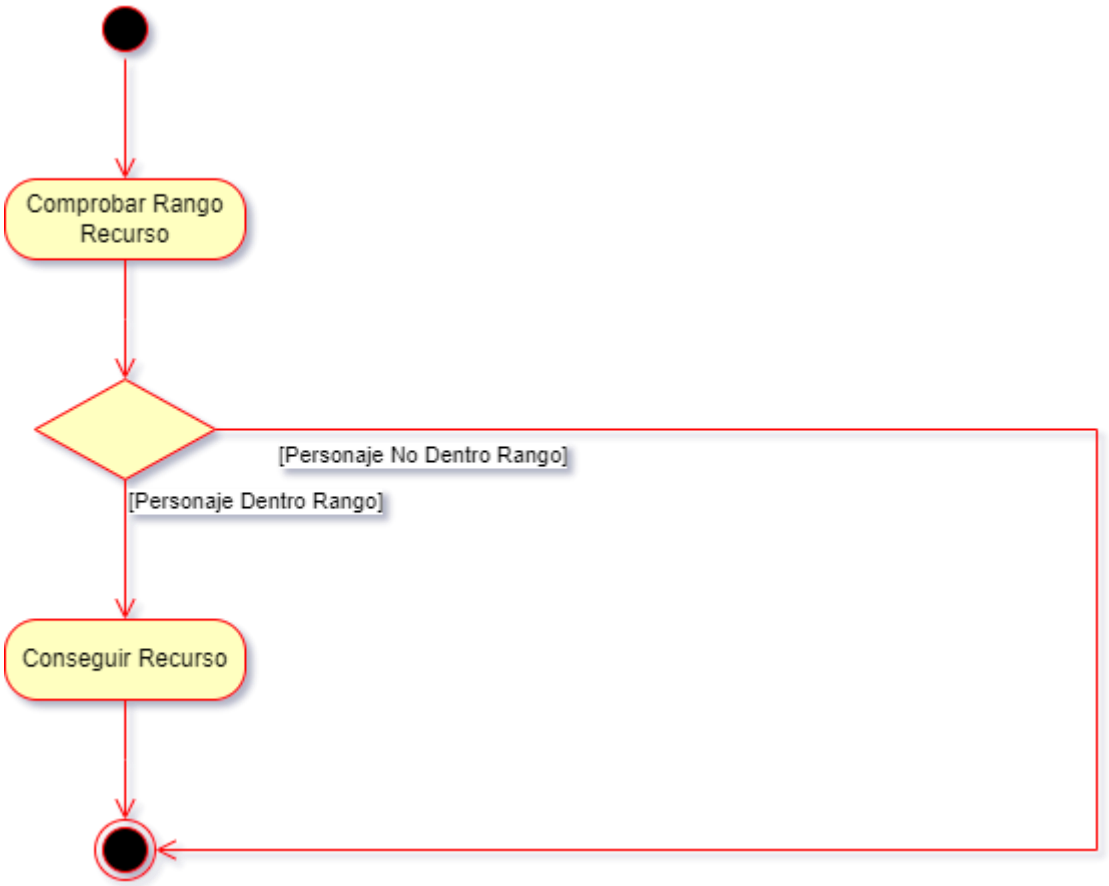


Ilustración 32. Diagrama de actividad (Conseguir recurso)

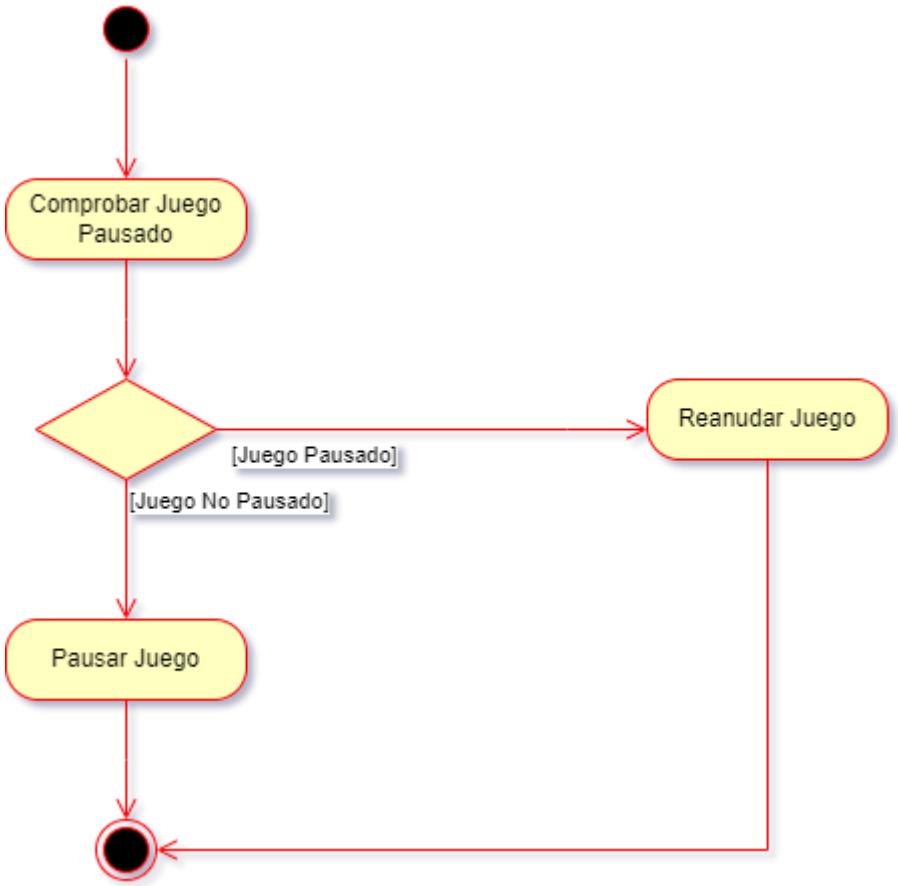


Ilustración 33. Diagrama de actividad (Pausar juego)

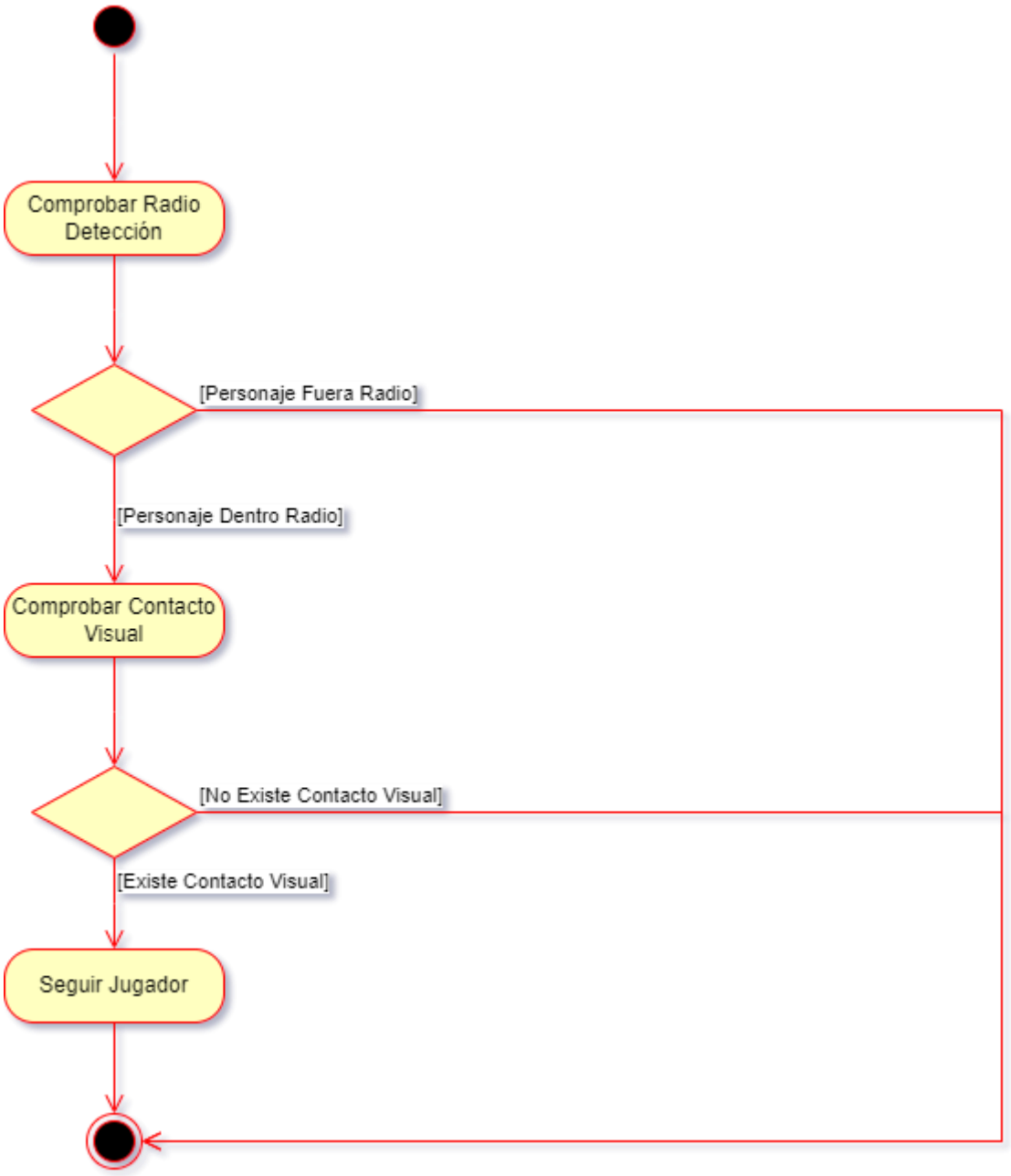


Ilustración 34. Diagrama de actividad (Seguir jugador)

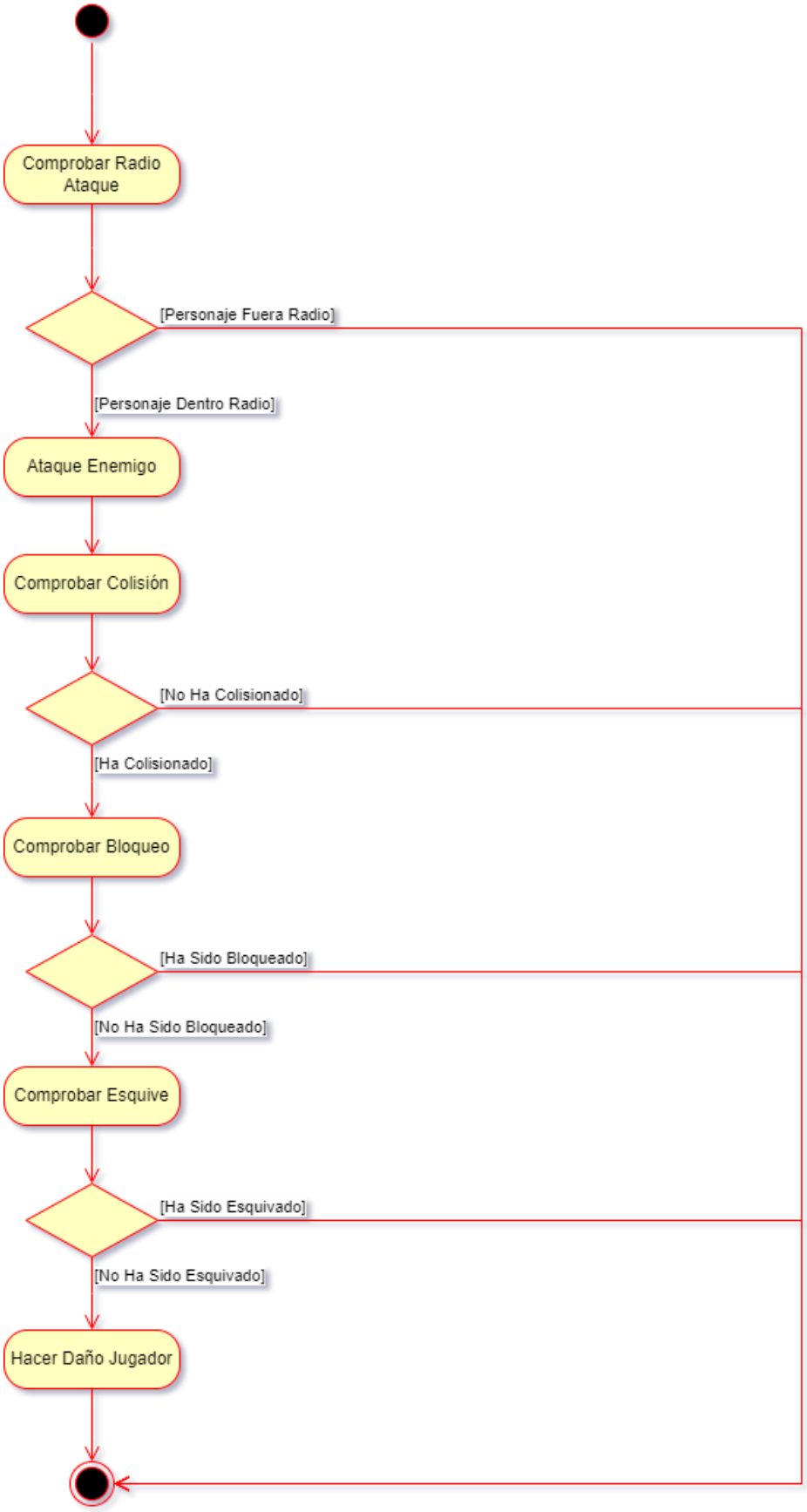


Ilustración 35. Diagrama de actividad (Ataque enemigo)

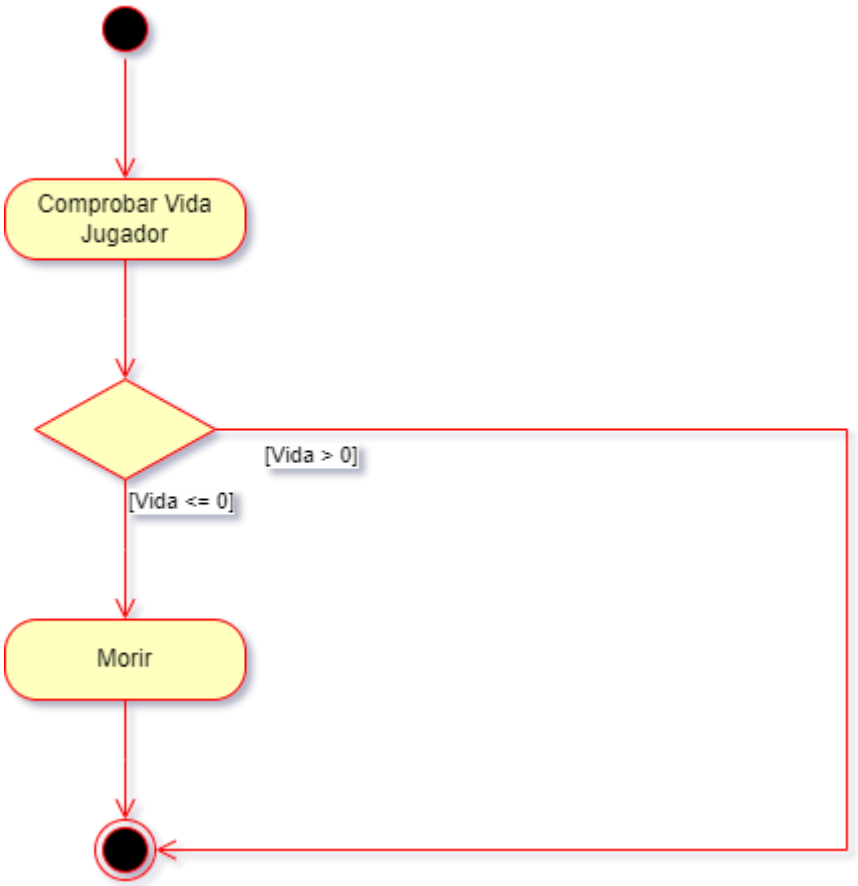


Ilustración 36. Diagrama de Actividad (Morir)

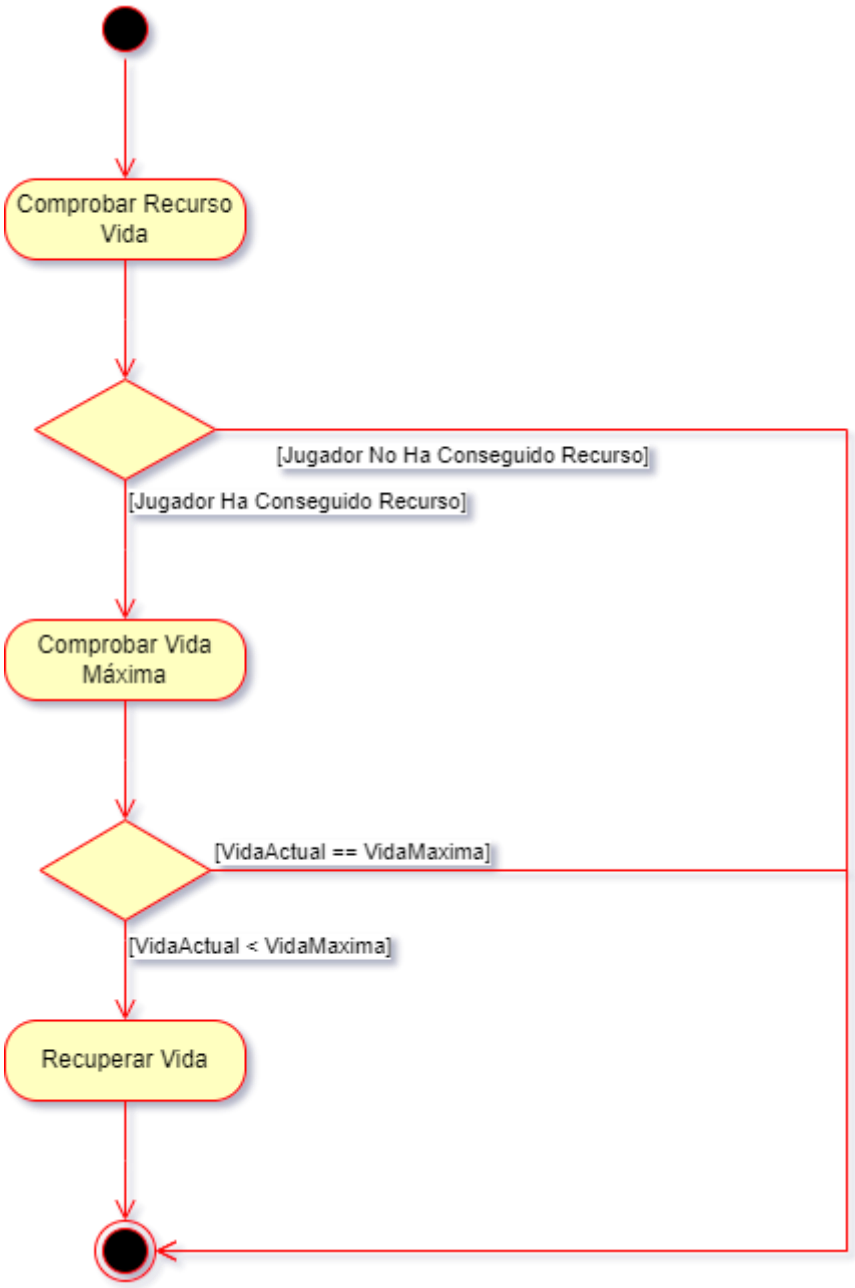


Ilustración 37. Diagrama de actividad (Recuperar Vida)

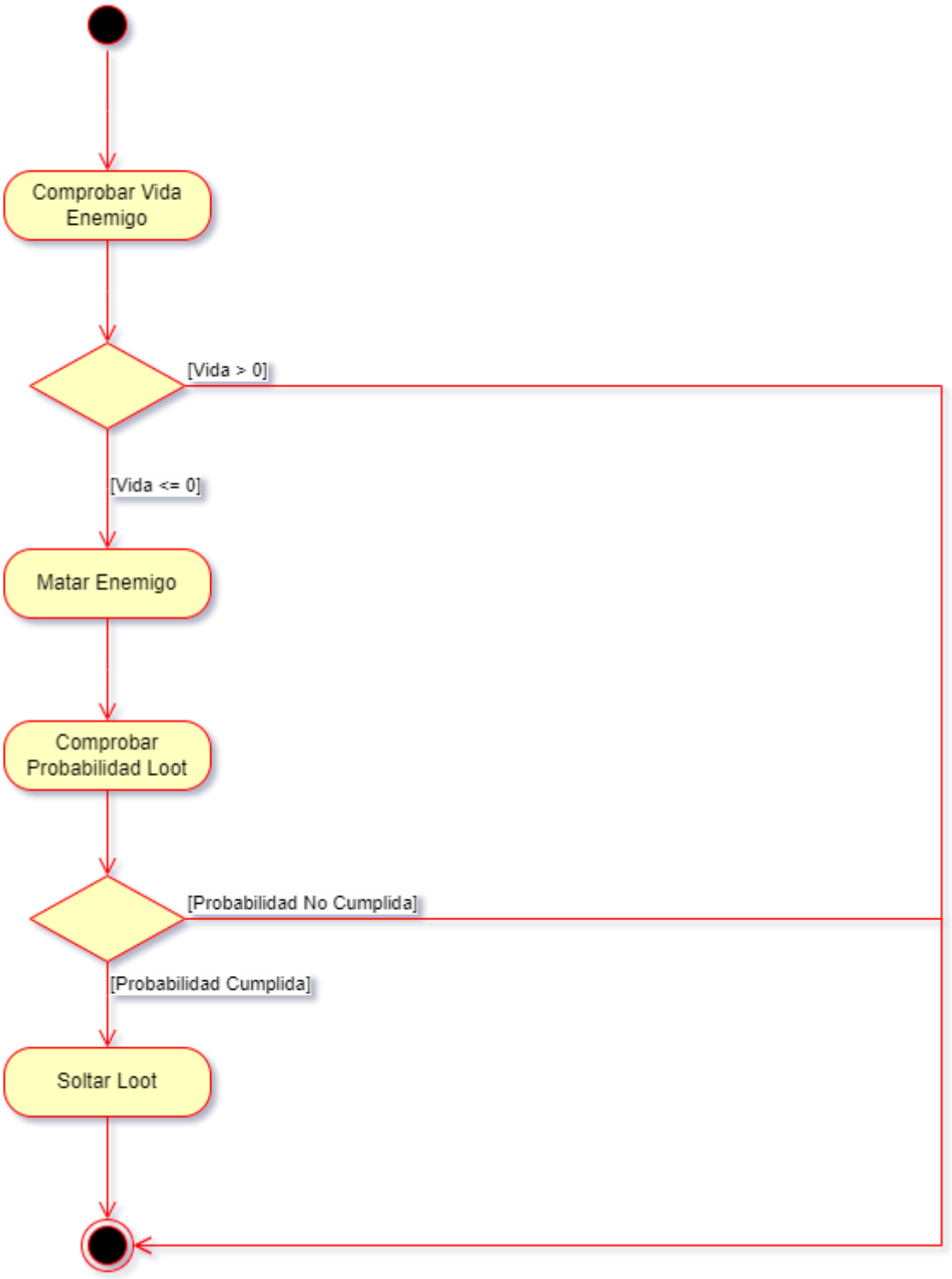


Ilustración 38. Diagrama de actividad (Mater enemigo)

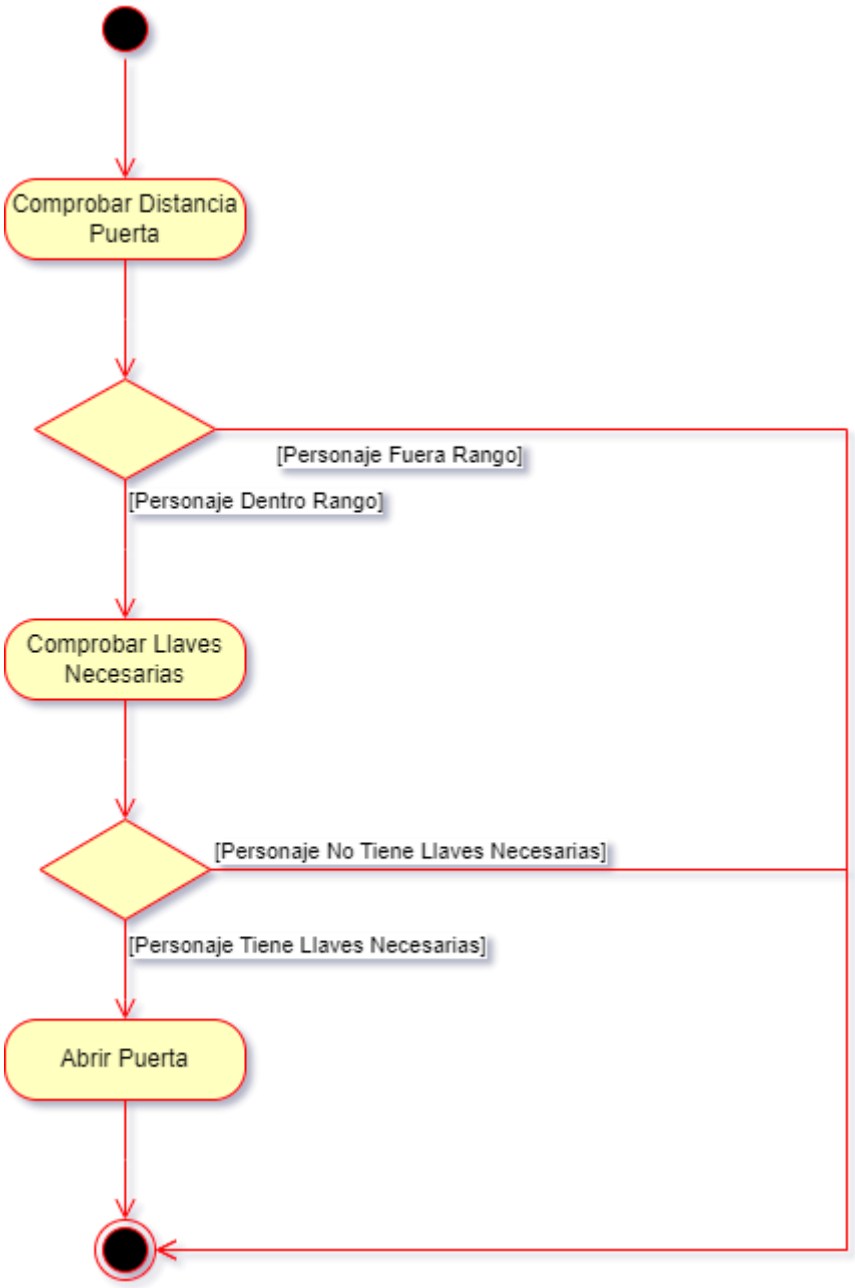


Ilustración 39. Diagrama de actividad (Abrir puerta)

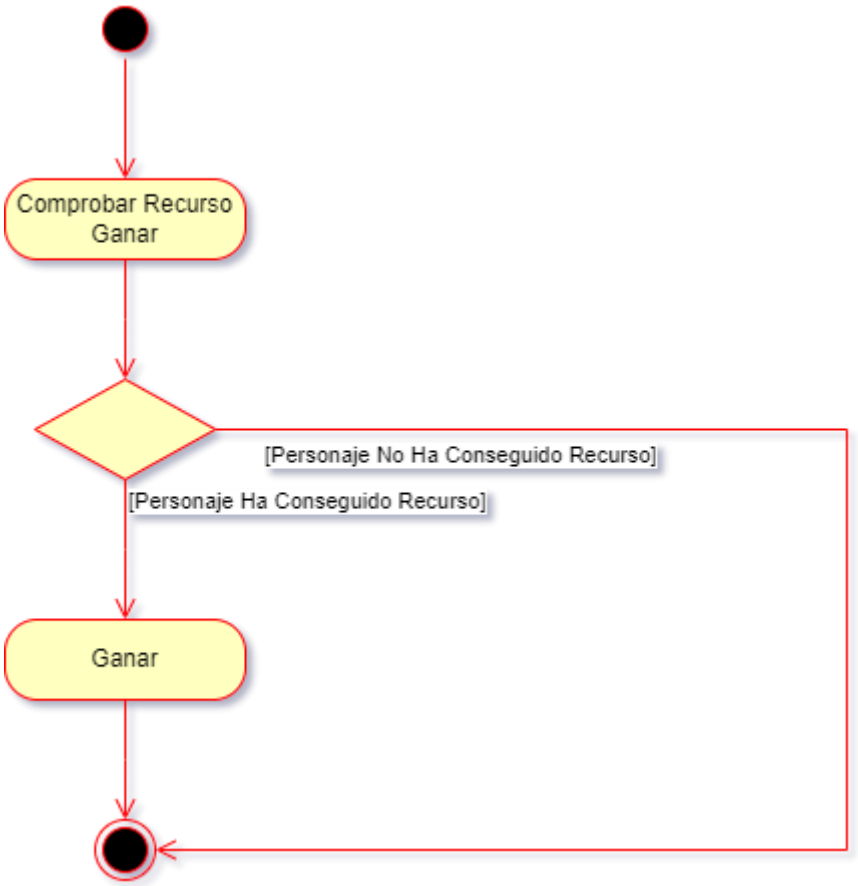


Ilustración 40. Diagrama de actividad (Ganar partida)

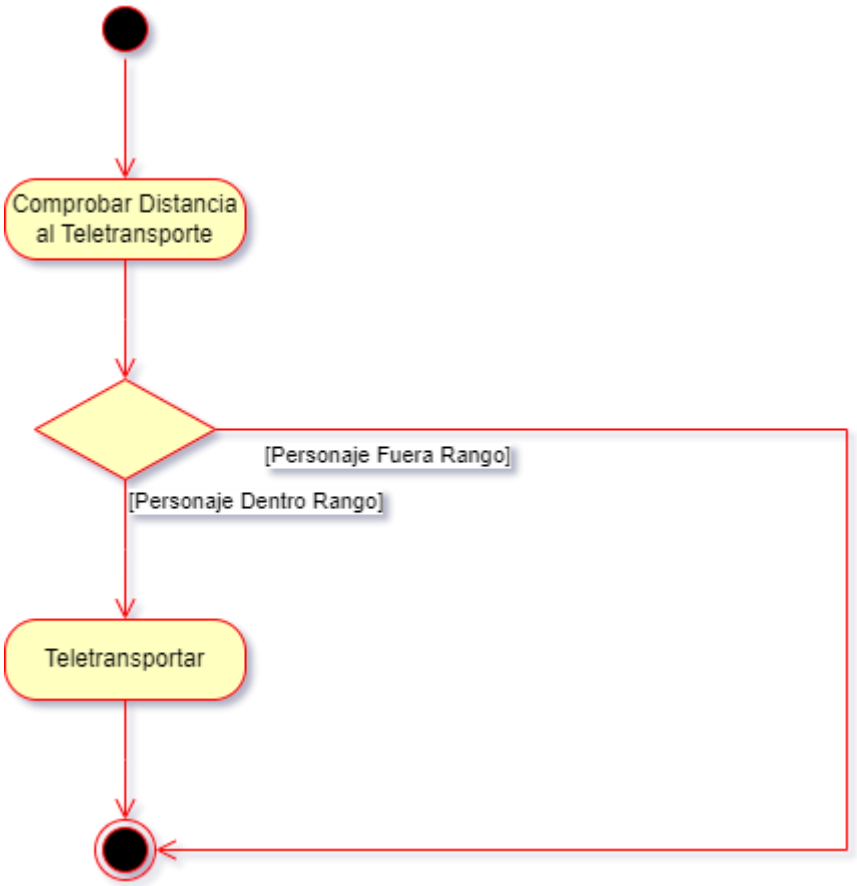


Ilustración 41. Diagrama de actividad (Teletransportarse)

En vista de cuáles son nuestras necesidades, vamos a plantear una serie de **diagramas de clase** divididos en tres categorías: aquellos que están relacionados con el **jugador** y aquellos que lo están con los **enemigos**. Notar que los diagramas de clases se encuentran simplificados y solo se han dejado los atributos/métodos/clases más importantes con el fin de que no resulten demasiado abrumadores.

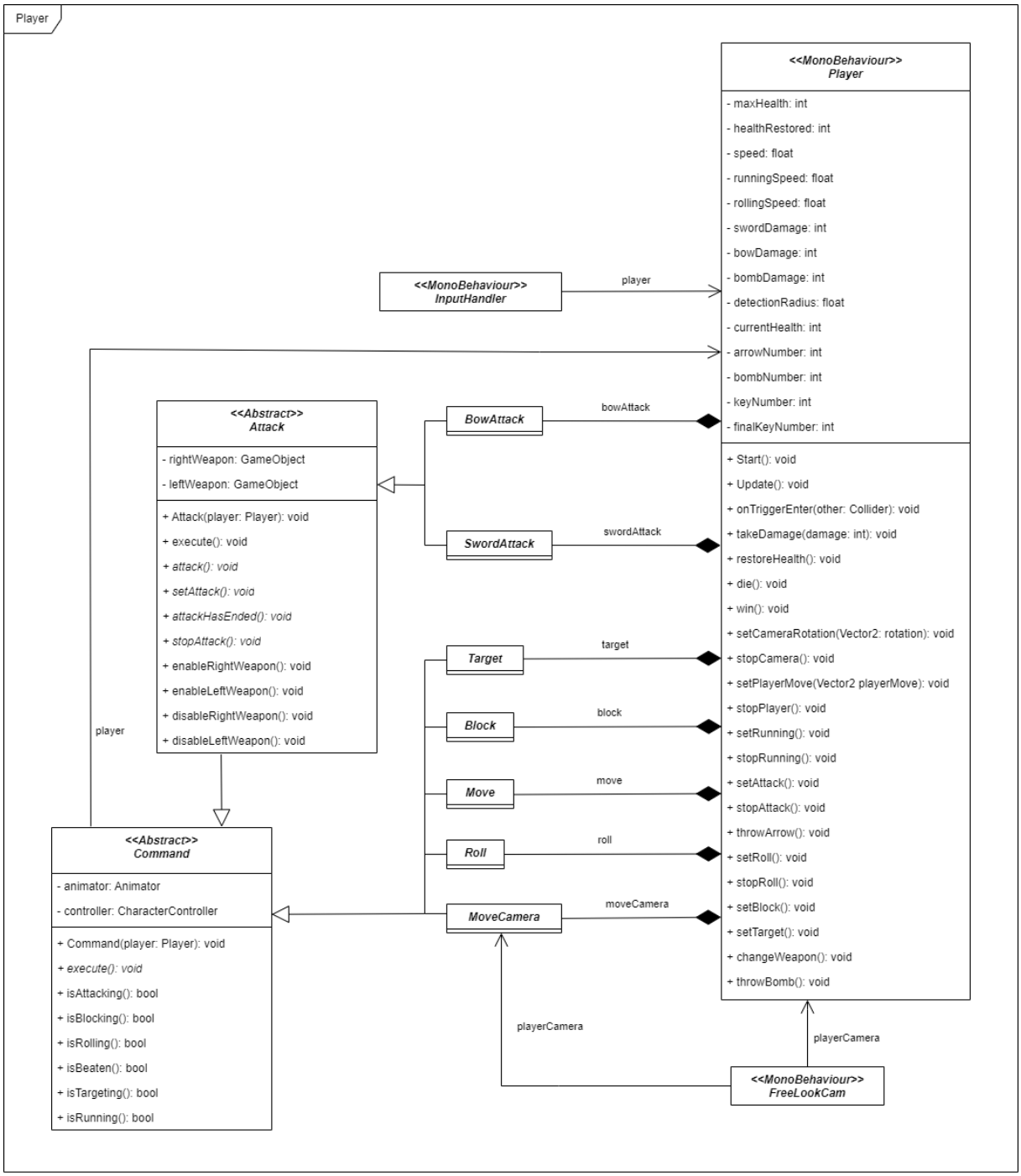


Ilustración 42. Diagrama de clases del jugador

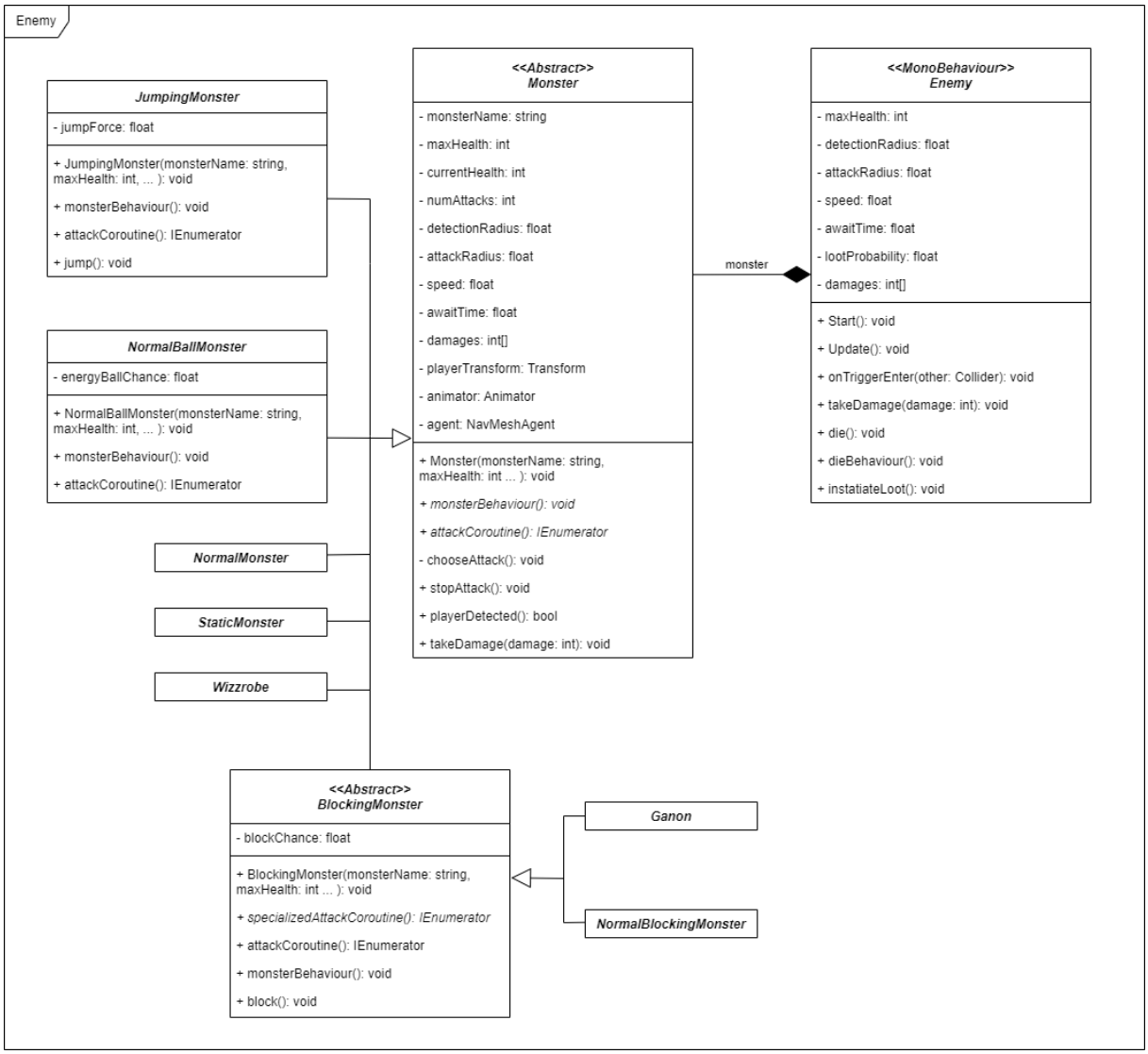


Ilustración 43. Diagrama de clases de los enemigos

Hay que entender que el funcionamiento del sistema de *scripting* de *Unity* consiste en asociar *scripts* a *GameObjects*; por tanto, tenemos una cantidad importante de clases (*scripts*) que no comparten relaciones con ninguna otra y resulta un tanto absurdo representarlas en estos diagramas de clases. Su funcionalidad y contenido se discutirá más adelante.

2.2.2 Análisis y diseño de la generación procedural

Para el diseño de esta parte de nuestro proyecto vamos a prescindir de la elaboración de casos de uso y demás etapas del proceso unificado, ya que, a nuestro modo de ver, resulta mucho más útil observar las etapas de estos algoritmos una por

una para entender su funcionamiento. Por ello, primeramente, vamos a comentar un poco cuáles son las ideas generales.

Para empezar, comentar que el videojuego va a poseer dos **tipos de mazmorra** bien identificadas: por una parte, un **castillo** cerrado, dividido en diferentes habitaciones; por la otra, **islas flotantes en el cielo** y conectadas de alguna forma. Mientras que el núcleo del juego sigue siendo el mismo, estos dos tipos de escenarios funcionarán de formas diferentes; por una parte, las habitaciones del castillo (generadas por medio de un **algoritmo geométrico** que se discutirá más adelante) seleccionan aleatoriamente qué habitaciones contienen enemigos y cuáles objetos y no están dispuestas de ninguna forma en particular, mientras que las islas en el cielo seguirán un determinado orden establecido por una **gramática generativa** y habrá puertas que se deban desbloquear con llaves y las islas con enemigos estarán determinadas con dicha gramática.

Como ya hemos comentado anteriormente, en este proyecto vamos a emplear dos algoritmos bien diferenciados de generación procedural: un **algoritmo geométrico** y un **algoritmo basado en gramáticas generativas**. No obstante, ambos son algoritmos complejos con varias etapas y ni el primero es totalmente geométrico ni el segundo consiste únicamente en gramáticas generativas.

A continuación, vamos a realizar una exploración de ambos:

2.2.2.1 Generación procedural geométrica

Para empezar, aclarar que nos hemos inspirado en buena medida en el algoritmo propuesto en el artículo de A. Adonaac (2015); no obstante, hemos realizado algunas modificaciones claves.

Vamos a presentar a continuación un esquema general a muy alto nivel de en qué va a consistir este algoritmo, de forma que iremos discutiendo cada una de sus etapas más adelante:

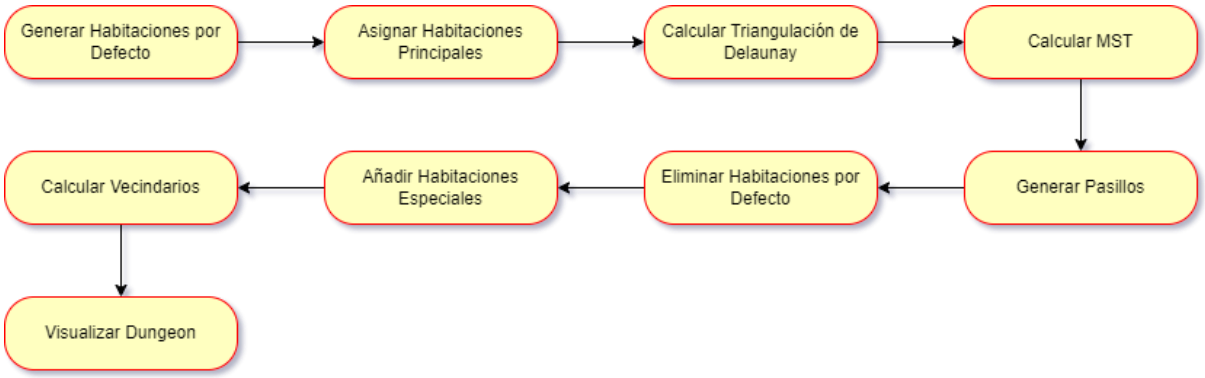


Ilustración 44. Esquema general del algoritmo

Vamos a ir explicando el algoritmo etapa por etapa (Nota: los ejemplos que se usan para ilustrar los pasos no son generaciones reales o siquiera correctas, únicamente sirven para ilustrar):

- **Generar habitaciones por defecto.** Esta etapa del algoritmo consiste fundamentalmente en una **subpartición binaria recursiva**. Partimos de un rectángulo determinado definido por una serie de parámetros (fundamentalmente su centro, longitud y anchura) que será el espacio total de la mazmorra, el cual está **discretizado**, ya que los suelos de una mazmorra poseen un tamaño determinado; por tanto, este espacio estará dividido en "suelos", que serán su unidad mínima, y cada habitación tendrá un tamaño de NxM suelos.

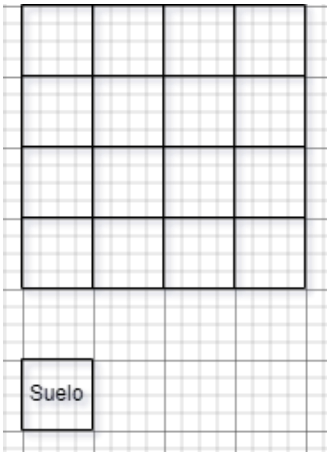


Ilustración 45. Representación de una habitación 4x4.

Recursivamente, seleccionamos un eje de partición (vamos alternando entre horizontal y vertical para dar más variedad a las habitaciones) aleatoriamente de forma que el eje produce dos habitaciones las cuales una

de ellas puede tener como mínimo una longitud/anchura de 1 suelo y como máximo la otra puede tener una longitud/anchura del número de suelos de la habitación madre menos uno. Usaremos además **números aleatorios normalizados** para que el eje suela partir más o menos por la mitad y no tener demasiadas habitaciones extremas, de manera que el área bajo la función de densidad, esto es, la probabilidad de que la variable adquiera un valor entre 1 y el máximo menos 1 sea la siguiente:

$$P(a < X < b) = F(b) - F(a) \simeq 1$$

Siendo $a = 1$ y $b = Max - 1$. De esta forma, nos debería quedar un rectángulo dividido que sea algo así (aunque con muchas más habitaciones):

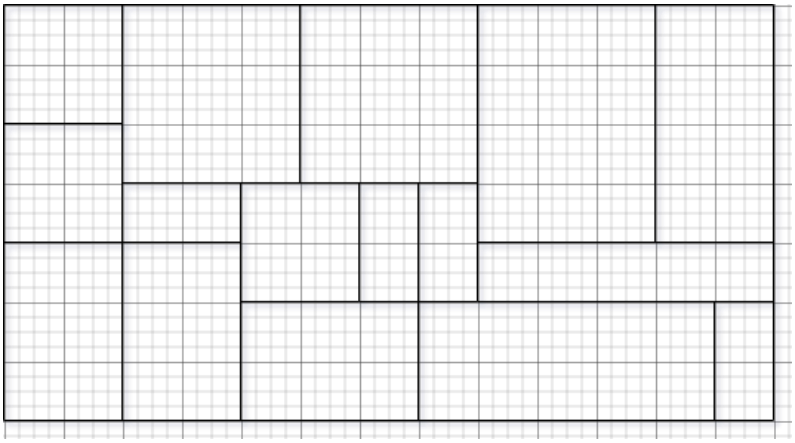


Ilustración 46. Ejemplo de mazmorra subdividida en habitaciones rectangulares

Como es lógico, puesto que es una subdivisión binaria, al final de la misma tendremos una potencia de 2 como número de habitaciones, dependiendo de la cantidad de subdivisiones que hagamos por habitación, esto es, de la **profundidad del árbol** virtual que se está formando. Podríamos pensar esta profundidad como el número de divisiones que ha sufrido el rectángulo original hasta llegar al rectángulo actual que está procesando el algoritmo recursivo.

$$N = 2^l$$

Siendo N el número de habitaciones y l la profundidad del árbol. Para la **condición de parada** del proceso recursivo podríamos elegir que pare cuando el número de habitaciones alcance el que se haya especificado

como parámetro. Esto tendría más sentido y nos permitiría tener más control sobre la generación procedural, pero hay un problema: al hacer esto, la mazmorra nos podría posiblemente quedar tal que así:

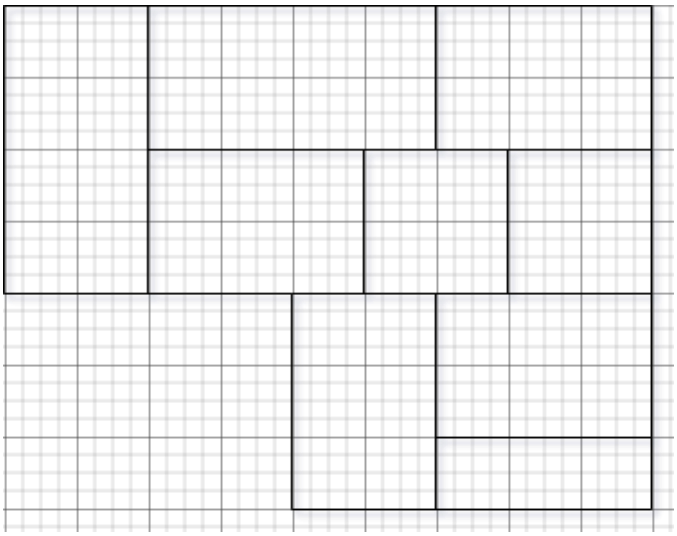


Ilustración 47. Ejemplo de mazmorra usando un número de habitaciones determinado como condición de parada

El principal problema que presenta esta topología es que nos puede causar muchos problemas en siguientes etapas a la hora de realizar la conectividad de las habitaciones. Por tanto, hemos optado por una condición de parada menos flexible pero que no los provoque, que es que el proceso recursivo pare **en la profundidad en la que el número de habitaciones resultante es inmediatamente mayor que el número especificado como parámetro**, esto es, que $n \leq 2^l$, siendo n el número de habitaciones especificado y l la profundidad del árbol; así, el número de habitaciones siempre será potencia de 2 y el rectángulo estará perfectamente conexo.

- **Asignar habitaciones principales.** Las habitaciones principales no son más que aquellas que sirven como **nodos** para el *Delaunay* y el *Minimum Spanning Tree* (MST); no cumplen ninguna función especial más allá de eso. No obstante, conviene (aunque no es obligatorio) que cumplan unos requisitos: para empezar, que sean de un tamaño relativamente **mediano**, y luego que estén relativamente **distribuidas** a lo largo de toda la mazmorra y lo suficientemente **separadas**. Tal vez se pudiera haber aplicado algún tipo de proceso de optimización, pero una solución empírica simple que da buenos resultados es **ordenar** todas las habitaciones por **tamaño** y elegir

aquellas que se encuentran más o menos en medio con **números aleatorios normalizados** (en este caso, $P(a < X < b) \simeq 1$ para $a = 0$ y $b = n$ siendo n el número de habitaciones total). De esta forma, le indicamos al algoritmo como parámetro el número de habitaciones principales y nos quedaría algo así:

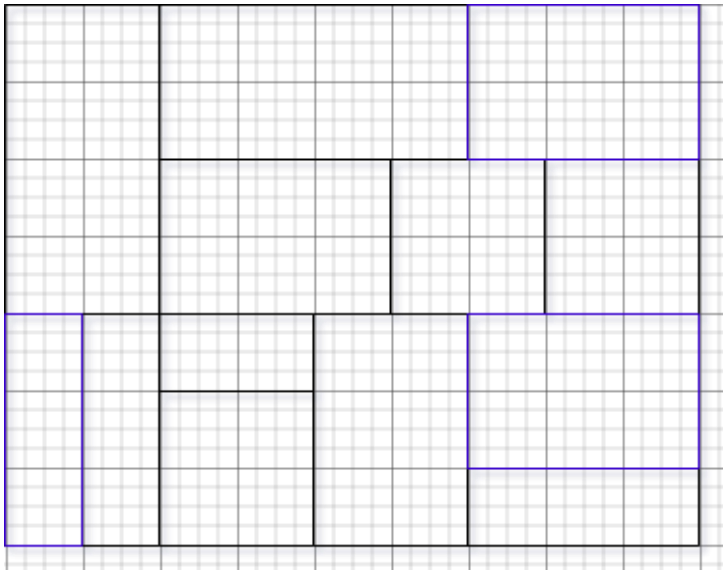


Ilustración 48. Asignación de habitaciones principales, las cuales aparecen en azul

- **Calcular triangulación de Delaunay.** Una triangulación es un caso particular de **teselación**, esto es, una partición del plano en distintas regiones o teselas, donde las regiones son triángulos. Las triangulaciones tienen múltiples usos, como por ejemplo cartográficas, donde se usan para medir distancias o determinar posiciones de puntos, o para informática gráfica, donde se usan como representación de objetos 3D. La triangulación de Delaunay en concreto es la **más equilátera posible** dada una determinada nube de puntos, lo cual quiere decir la triangulación de todas las posibles para esa nube de puntos cuyos triángulos tienden a estar más cerca de ser equiláteros.

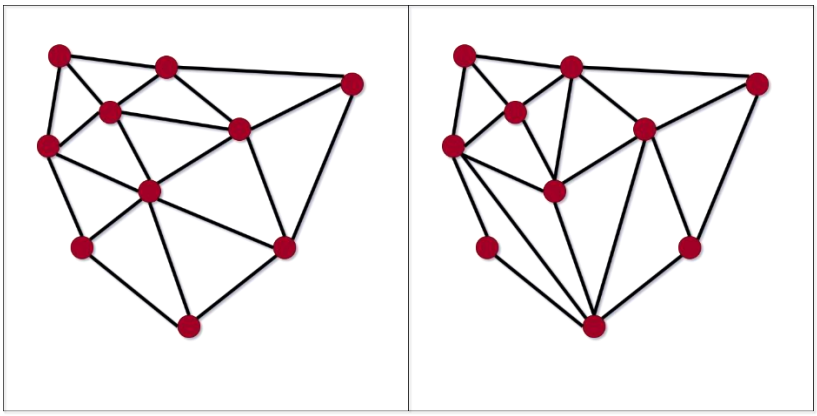


Ilustración 49. La triangulación de la izquierda es más equilátera que la de la derecha

Más rigurosamente, sea T una triangulación para la nube de puntos P de tamaño n, y sea $A(T)$ de tamaño $3n$ el vector de ángulos de T tal que $A(T) = \{\alpha_1, \alpha_2, \dots, \alpha_{3n}\}$, siendo $\alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_{3n}$, decimos que una triangulación T es más equilátera que una triangulación T' si ocurre que $\exists i \in [1, 3n]$ tal que $\alpha_j = \alpha'_j \; \forall j > i$ y además $\alpha_i > \alpha'_i$. Por tanto, la triangulación de Delaunay es aquella triangulación T para la que se cumple que $A(T) \geq A(T')$ (entendiéndose esto como que T es más equilátera que T') para **cualquier posible triangulación** T' de una determinada nube de puntos.

Ahora que ya sabemos lo que es la triangulación de Delaunay, ¿para qué la necesitamos? Pues porque queremos crear un **grafo** entre los centros de las habitaciones principales de forma que todos los centros sean accesibles entre ellos, esto es, que sea **conexo** (imprescincible) y en el que no haya **intersecciones** entre ejes (deseable).

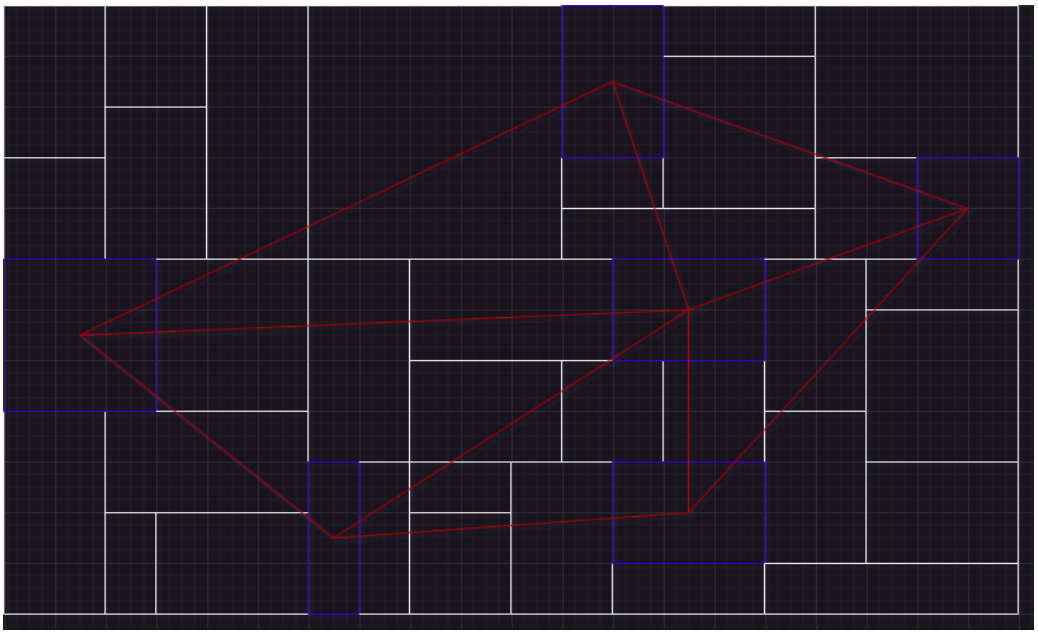


Ilustración 50. Ejemplo de triangulación para una mazmorra con 6 habitaciones principales

- **Calcular MST.** Un MST es un **subgrafo** de un **grafo conexo no dirigido ponderado** $G = (V, E, w)$, donde V es el conjunto de vértices, E el de aristas tal que $V = \{v_1, v_2, \dots, v_n\}$ y $E \subseteq \{(v_i, v_j) \in V \times V\}$ donde $(v_i, v_j) = (v_j, v_i) \forall i, j \in [1, n]$, $V \neq \emptyset$ y $\exists C \subseteq E$ tal que C es un subconjunto de las aristas, esto es, un camino, y se cumple que existe siempre un camino que une dos vértices cualesquiera de V, y w es el conjunto de pesos asociados a cada arista, representado como una función de asignación $w: E \to \mathbb{R}$ tal que el peso de una arista es $w(e) \in \mathbb{R}$. En concreto, el MST es un **árbol**, es decir, un grafo **acíclico conexo ponderado** $A = (V_A, E_A, w_A)$ donde no existe ni un solo camino $C \subseteq E$ tal que los dos vértices cualesquiera que une son el mismo, es decir, que no tiene ciclos, que cumple además que sus vértices son los mismos que los del grafo original ($V_G = V_A$) y que es el árbol de todos los posibles que se pueden obtener a partir de G donde $\sum_{i=1}^{n} w_A(i) \leq \sum_{i=1}^{n} w'_A(i)$ para cualquiera de los posibles A' que se pueden obtener de G; básicamente, que tenga **los pesos mínimos** posibles de entre todos los posibles árboles.

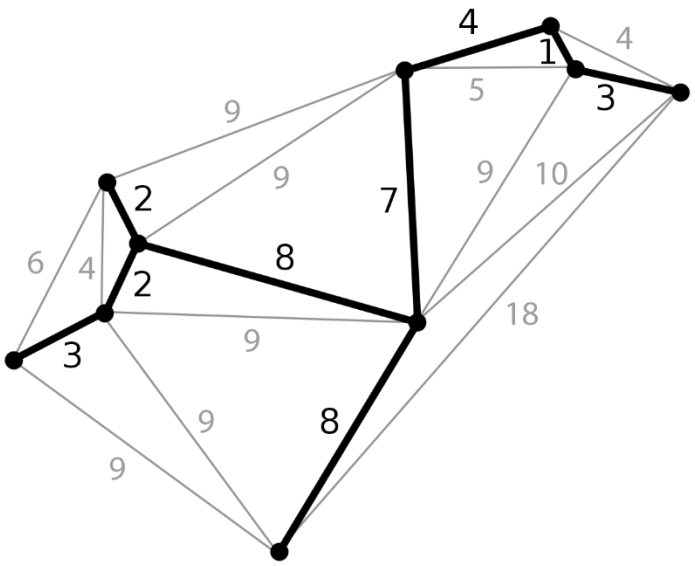


Ilustración 51. Ejemplo de MST (Wikipedia, 2023d)

Algorítmicamente, para calcular el MST vamos a usar el **algoritmo de Kruskal**, el cual tiene los siguientes pasos:

1. **Ordenar** todas las aristas del grafo original en orden ascendente según su peso (en nuestro caso, la longitud de la arista medida como la distancia entre los centros de las habitaciones).
2. Recorrer el vector ordenado de aristas y coger la menor arista cada vez, de forma que, **si no forma un ciclo** con el grafo solución (el MST) construido hasta el momento, se añade a la solución.
3. Seguir hasta que no queden aristas.

Como podemos ver, se escoge una **solución óptima local** en cada paso para acabar con una solución óptima global al final.

¿Por qué queremos calcular el MST? Fundamentalmente, porque la triangulación de Delaunay por sí misma nos puede devolver algún camino especialmente corto entre la entrada y el boss final, y queremos que el jugador tenga que recorrer una cierta cantidad mínima de habitaciones. El MST nos permite hacer esto sin hacer que el grafo sea inconexo (lo cual sería fatal para el diseño de la mazmorra) y haciendo además que los caminos no sean especialmente largos. Por otra parte, es cierto que dejar el grafo tal cual sin ningún ciclo empobrece la experiencia, así que

añadiremos una cierta cantidad de aristas para formar algunos caminos cíclicos (pero no demasiados).

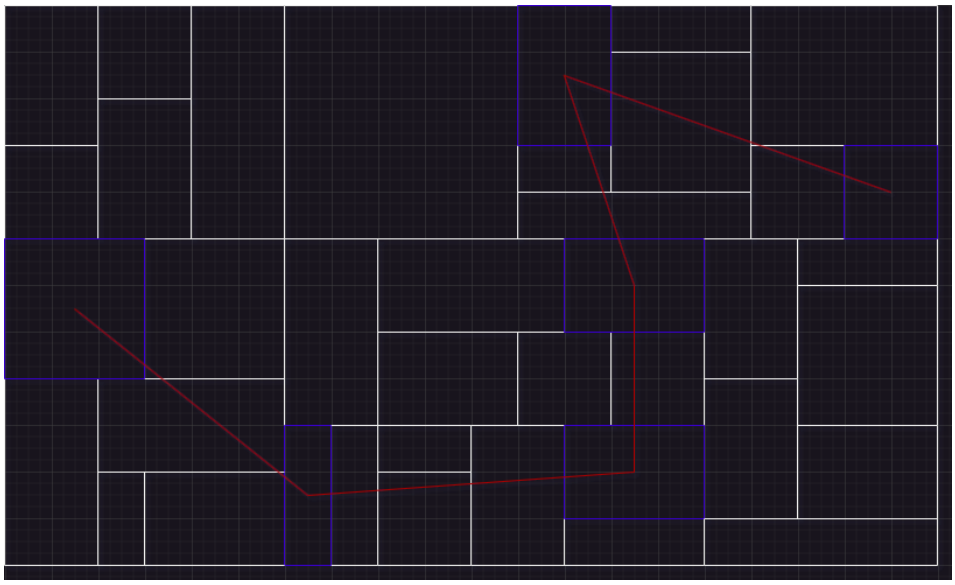


Ilustración 52. Grafo de la marmorra tras aplicarle el Kruskal

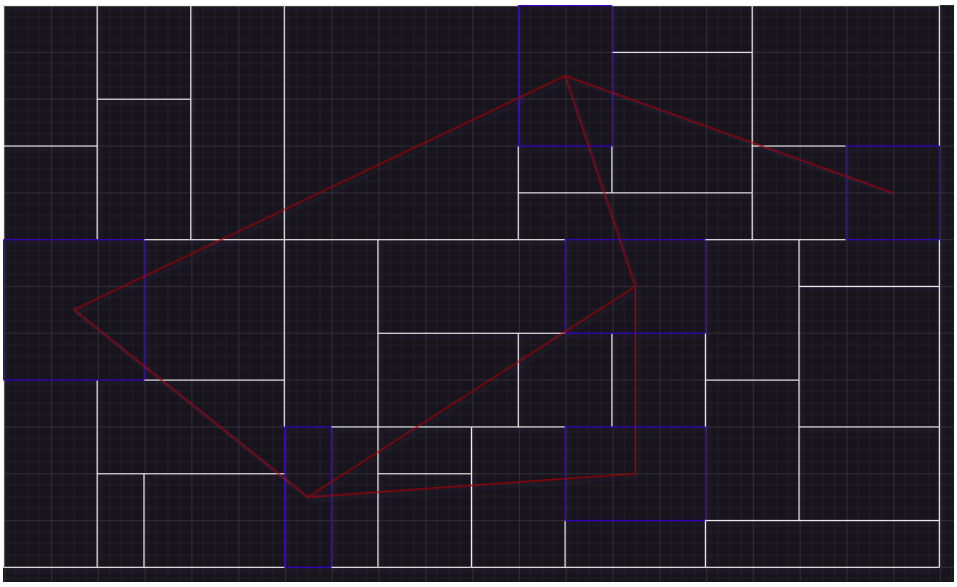


Ilustración 53. Grafo de la mazmorra tras aplicarle el Kruskal y añadirle 2 aristas aleatorias

- **Generar pasillos.** Este paso consiste en **conectar las habitaciones principales con otras por defecto** ateniéndonos al grafo que hemos generado en los pasos anteriores. Para ello, vamos a construir una serie de aristas tipo L entre las habitaciones conectadas según el grafo de la *dungeon* calculado anteriormente.

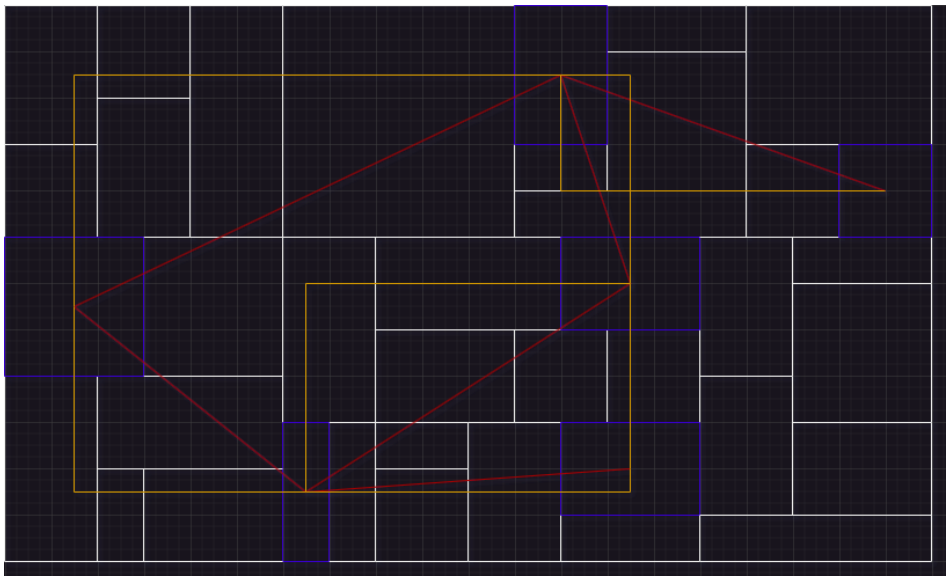


Ilustración 26. La mazmorra anterior con los pasillos, en verde

Posteriormente, comprobamos qué habitaciones pasan a través de estos pasillos y los añadiremos a la *dungeon* final junto con las habitaciones principales. Como podemos ver, existe la posibilidad de que un pasillo pase exactamente entre dos habitaciones, pero gracias a la implementación que comentaremos en otros apartados, esos casos se solucionan cogiendo las habitaciones a **ambos lados**.

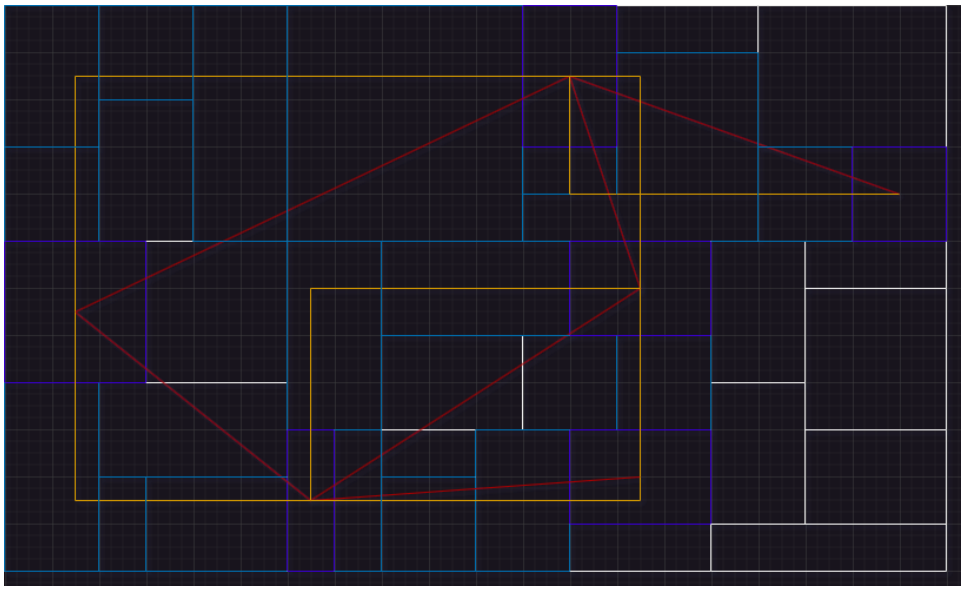


Ilustración 54. Mazmorra tras designar las habitaciones que formarán los pasillos

- **Eliminar habitaciones por defecto.** En esta etapa, sencillamente eliminamos todas aquellas habitaciones que no sean ni principales ni pasillos.

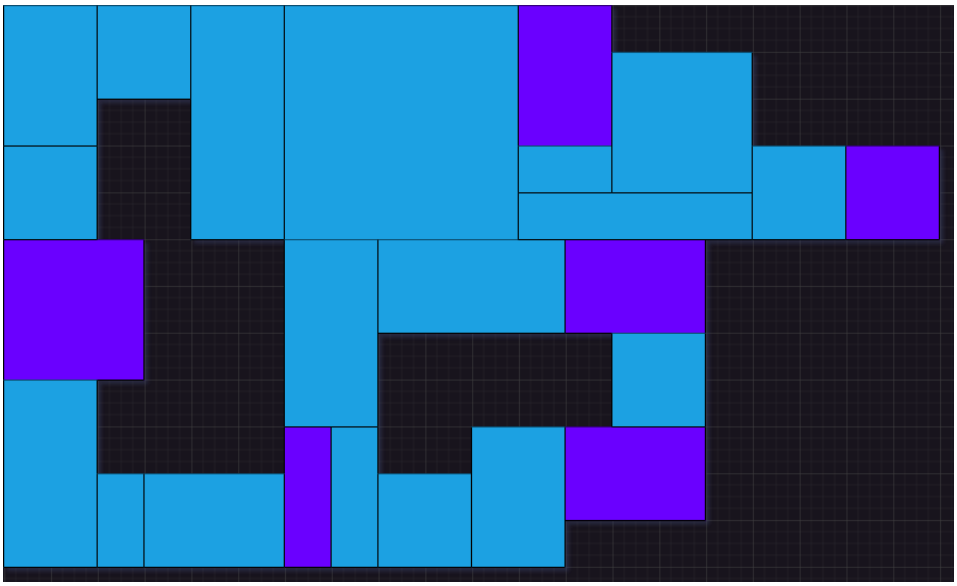


Ilustración 55. Mazmorra tras eliminar las habitaciones por defecto

- **Añadir habitaciones especiales.** Básicamente, tenemos dos habitaciones especiales que tenemos que añadir a posteriori ya que poseen unas dimensiones específicas y podríamos buscar dos habitaciones que coincidan con esas dimensiones, pero podrían o no podrían estar. Estas habitaciones son la **habitación inicial** y la **final**. En la primera aparecerá el jugador y en la segunda se encontrará el *boss final*. La estrategia que vamos a seguir para colocarlas es simple: buscamos la habitación de la **esquina inferior izquierda** y colocamos la habitación inicial de forma que esté alineada con el centro de dicha habitación en el eje x y que esté lo suficientemente abajo en el eje y como para que ambas habitaciones sean adyacentes pero no se solapen; la idea es la misma para la habitación final, buscamos la habitación de la **esquina superior derecha**, alineamos con el eje x y subimos lo suficientemente arriba en el eje y. En realidad no son exactamente la esquina superior derecha e inferior izquierda, porque lo que hacemos es buscar las habitaciones con la coordenada y máxima e y mínima y para resolver ambigüedades nos quedamos con aquellas habitaciones que además tengan x máxima y x mínima, respectivamente.

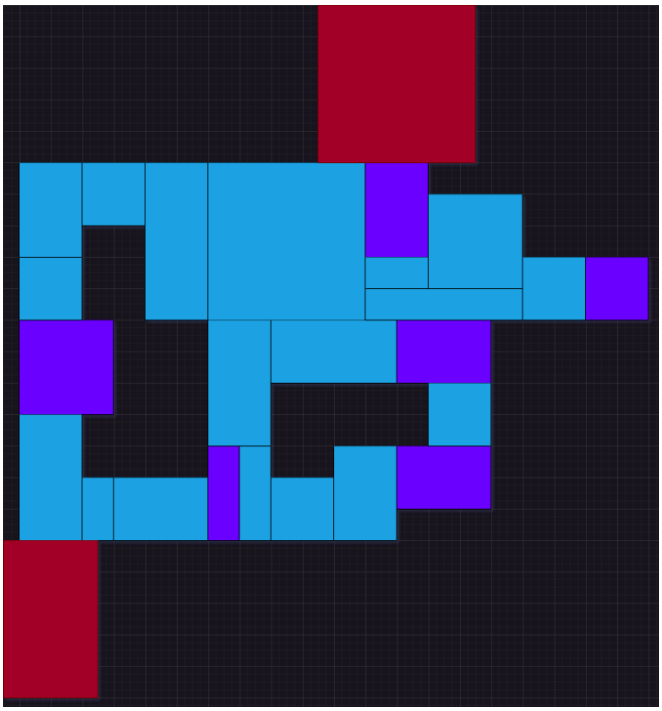


Ilustración 56. Mazmorra final con las habitaciones especiales colocadas en rojo

- **Calcular vecindarios.** En esta etapa, el algoritmo calcula cuáles habitaciones están **adyacentes** por cada una de la mazmorra. Esto nos servirá fundamentalmente para las **puertas**. Sin entrar en detalles de la implementación, la idea es crear un rectángulo **ligeramente mayor** que el de la habitación original y comprobar qué habitaciones se solapan con él.

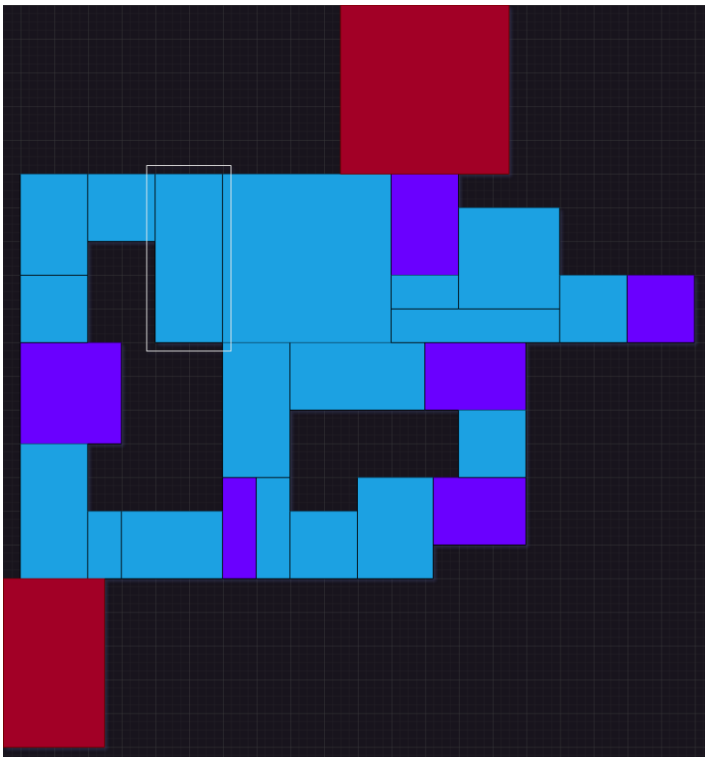


Ilustración 57. Cálculo de vecindario para la habitación rodeada por el rectángulo blanco que representa el que vamos a usar para calcular dicho vecindario.

Como podemos comprobar, los **falsos positivos** son comunes por medio de este método, concretamente aquellos que se encuentren justamente en la esquina, con los cuales no puede compartir puerta. Esto lo solucionaremos en el **cálculo de las puertas**, de manera que debemos comprobar los diferentes casos para las posiciones de las habitaciones:

1. En el primer caso, una habitación está por encima de la otra.

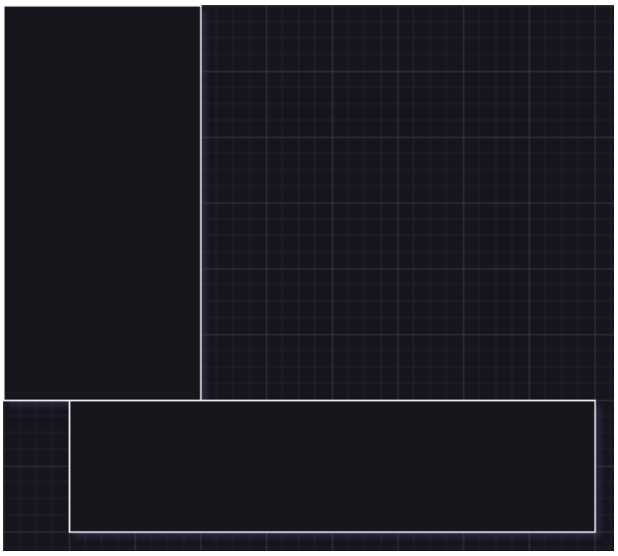


Ilustración 58. Ejemplo de caso 1

2. En el segundo caso, una habitación está al lado de la otra.

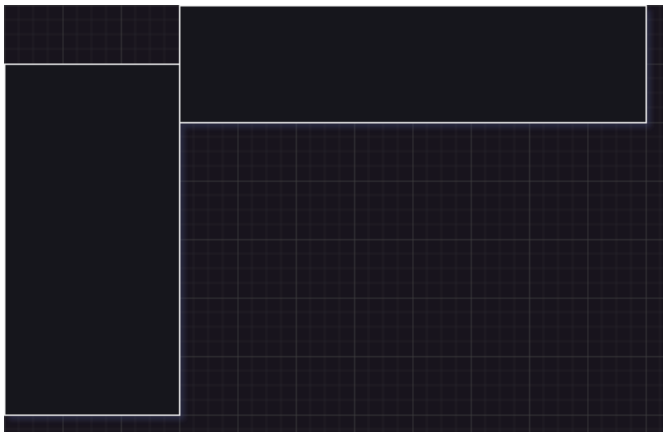


Ilustración 59. Ejemplo de caso 2

Lo primero que tenemos que hacer es situarnos en qué caso nos encontramos. Seguidamente, comprobamos si **comparten** una longitud de sus perímetros lo **suficientemente grande** como para albergar una puerta. Se podría pensar que pueda haber casos donde haya habitaciones que compartan una cierta longitud de sus perímetros pero no lo suficientemente grande para que haya puertas, por lo cual habría habitaciones inconexas, pero esa casuística la resuelve implícitamente la subdivisión binaria de la primera etapa, que únicamente es capaz de dividir habitaciones en “suelos”. Una vez hemos comprobado todo esto, solo nos falta crear las puertas, lo cual se hace sencillamente situándolas **en el medio** de la longitud de los perímetros que comparten las habitaciones.

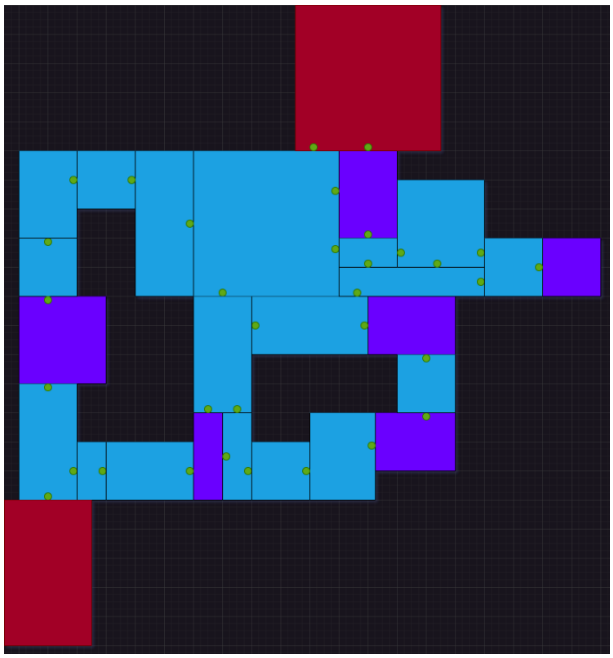


Ilustración 60. Localización aproximada de las puertas para esta mazmorra

- **Visualización de la dungeon.** Una vez que tenemos la planta de la mazmorra totalmente generada, solo resta crear la mazmorra propiamente dicha. Lo primero que hacemos es **rellenar los suelos** de cada habitación de acuerdo a su posición y tamaño. Posteriormente, **levantamos los muros** de manera que tenemos cuidado de **colocar las puertas** en sus posiciones correspondientes y no taparlas con alguna pared. Por último, **instanciamos la utillería** de acuerdo a las dimensiones de la habitación (podemos crear todos los *assets* que queramos con este propósito) y hacemos aleatoriamente (de acuerdo a los parámetros con los que contemos) que esa habitación tenga enemigos o *loot*, y también elejimos aleatoriamente el *final boss* al que nos enfrentaremos.

2.2.2.2 Generación procedural usando gramáticas

El otro algoritmo que vamos a implementar será uno basado en **gramáticas formales**. Antes de pasar a la explicación detallada del algoritmo, vamos a crear un poco de contexto.

En el diseño de videojuegos, los desarrolladores conciben el progreso de una determinada **misión** que el jugador deberá realizar para poder avanzar en el videojuego, usualmente empleando **grafos de misión** para indicar el esquema lógico de qué tareas van antes de cuáles, y cuáles se pueden realizar simultáneamente,

etc… Estos grafos de misión, pese a ser en principio independientes de la geometría de los escenarios y de su topología, en realidad en muchas ocasiones son isomorfos a los propios grafos que representan las conexiones entre niveles (J. Dormans, 2010).

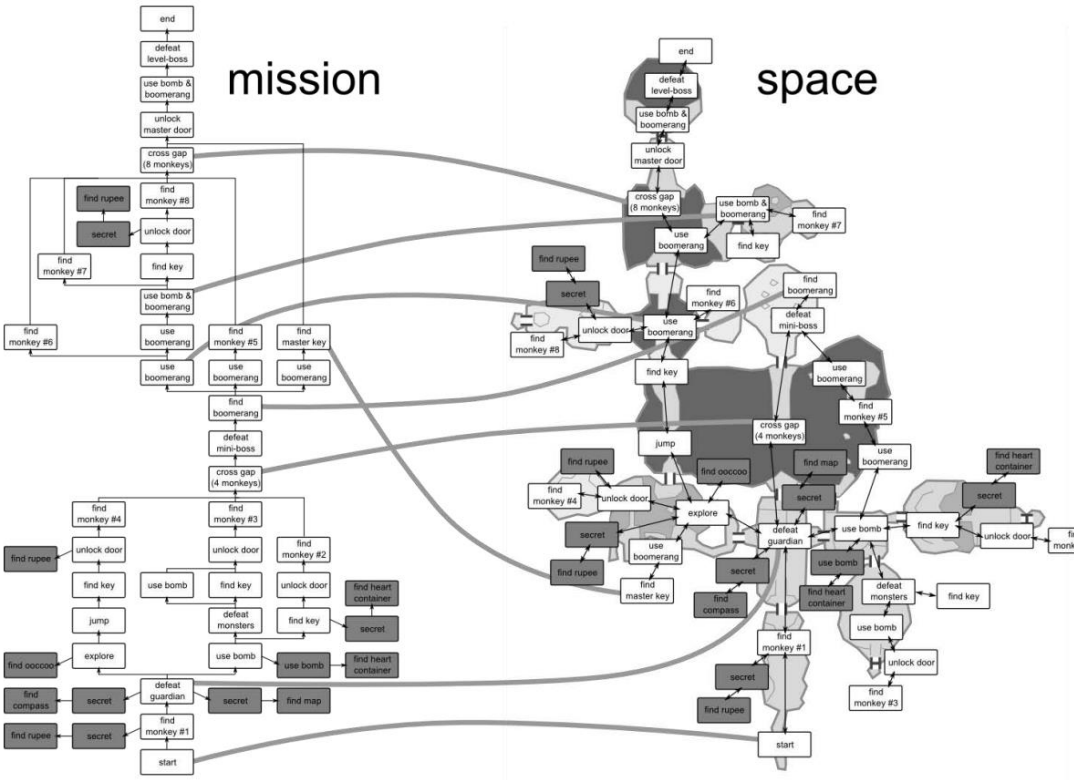


Ilustración 61. Misión y espacio en el nivel del Templo del Bosque de The Legend of Zelda: The Twilight Princess (J. Dormans, 2010)

El objetivo de nuestro algoritmo será entonces generar un grafo de misión de acuerdo a una determinada gramática formal, pero antes vamos a explicar qué es exactamente esto.

Una **gramática generativa** se define como una cuaterna $G = (V, \mathrm{T}, S, \mathrm{P})$ donde V es un conjunto finito llamado **alfabeto de símbolos no terminales**, o simplemente, alfabeto de variables, T es otro conjunto finito que verifica $\mathrm{T} \cap V = \emptyset$ y se suele denominar **alfabeto de símbolos terminales**, $S \in V$ es una variable distinguida que se denomina **axioma** y $P \subseteq (V \cup \mathrm{T})^* \times (V \cup \mathrm{T})^*$ es un conjunto finito llamado **conjunto de producciones**, en definitiva, las reglas que la gramática sigue para construir palabras. Aquí un ejemplo de gramática:

$$V = \{S, A, B\}; T = \{a, b\}; P = \{S \to aB|bA, A \to a|aS|bAA, B \to b|bS|aBB\}$$

Sea $G = (V, T, S, P)$ una gramática y $\alpha, \beta \in \{V \cup T\}^*$, decimos que β se deriva de α en un paso y se nota como $\alpha \Rightarrow^+ \beta$ si y solo si $\exists \gamma_1, \gamma_2 \in \{V \cup T\}^*$ y $\exists (\delta, \phi) \in P \,/\, \alpha = \gamma_1 \delta \gamma_2$ y $\beta = \gamma_1 \phi \gamma_2$. Decimos además que β se deriva de α y se nota como $\beta \Rightarrow^* \alpha$ si y solo si $\exists \gamma_1, \ldots, \gamma_n \in \{V \cup T\}^* / \alpha = \gamma_1 \Rightarrow^+ \ldots \Rightarrow^+ \gamma_n = \beta$ y este número de pasos es finito.

Las gramáticas generan **lenguajes**, que son el conjunto de todas las posibles cadenas compuestas por símbolos terminales que se derivan del axioma de la gramática dada, esto es, el conjunto de **palabras**:

$$L(G) = \{u \in T^* / S \Rightarrow^* u\}$$

Por tanto, dada la gramática que se puso antes de ejemplo, podemos tener la siguiente secuencia de generación de la palabra $abbaaabb$:

$$S \rightarrow aB \rightarrow abS \rightarrow abbA \rightarrow abbaS \rightarrow abbaaB \rightarrow abbaaaBB \rightarrow abbaaabb$$

Cabe destacar que los símbolos de los alfabetos no tienen por qué ser únicamente letras, pueden ser cualquier cosa: números, formas geométricas… y, para el algoritmo que nos ocupa, serán **nodos** y **grafos**. Nuestro algoritmo seguirá las ideas propuestas en Dormans (2010), donde el autor detalla un algoritmo para realizar los pasos de derivación y así generar palabras (grafos de misión) a partir de una gramática dada:

1. Después de que un grupo de nodos hayan sido seleccionados para reemplazo por una regla de producción determinada, los nodos seleccionados son numerados de acuerdo a la parte izquierda de la regla (Paso 2 de la Ilustración 63).
2. Después, todas las aristas entre estos nodos son eliminadas (Paso 3 de la Ilustración 63)**.**
3. Los nodos numerados son entonces reemplazados por sus equivalentes, esto es, los nodos que tienen los mismos números de la parte derecha (Paso 4 de la Ilustración 63).
4. El resto de nodos de la parte derecha son añadidos (Paso 5 de la Ilustración 63).
5. Las aristas entonces se añaden de acuerdo a la parte derecha de la regla (Paso 6 de la Ilustración 63).
6. Finalmente, se quita la numeración de los nodos (Paso 7 de la Ilustración 63).

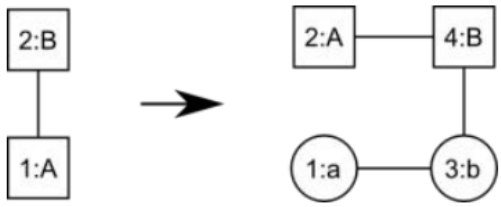


Ilustración 62. Ejemplo de regla de producción (J. Dormans, 2010)

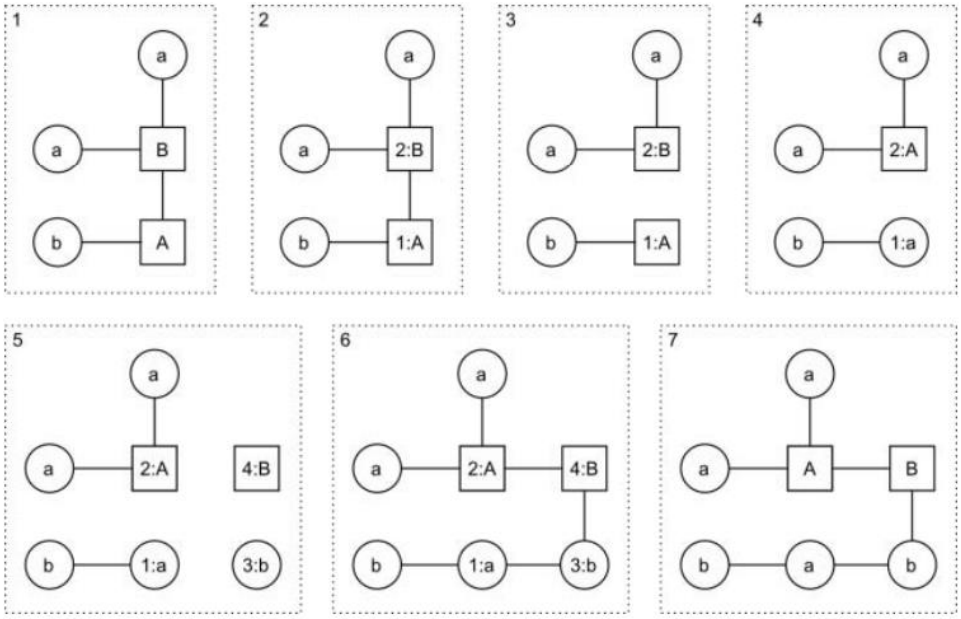


Ilustración 63. Ejemplo de secuencia de sustitución siguiendo la regla de producción de la ilustración 62 (J. Dormans, 2010)

Para nuestro proyecto además usaremos (aproximadamente) la misma **gramática** que el autor propone, que es la siguiente:

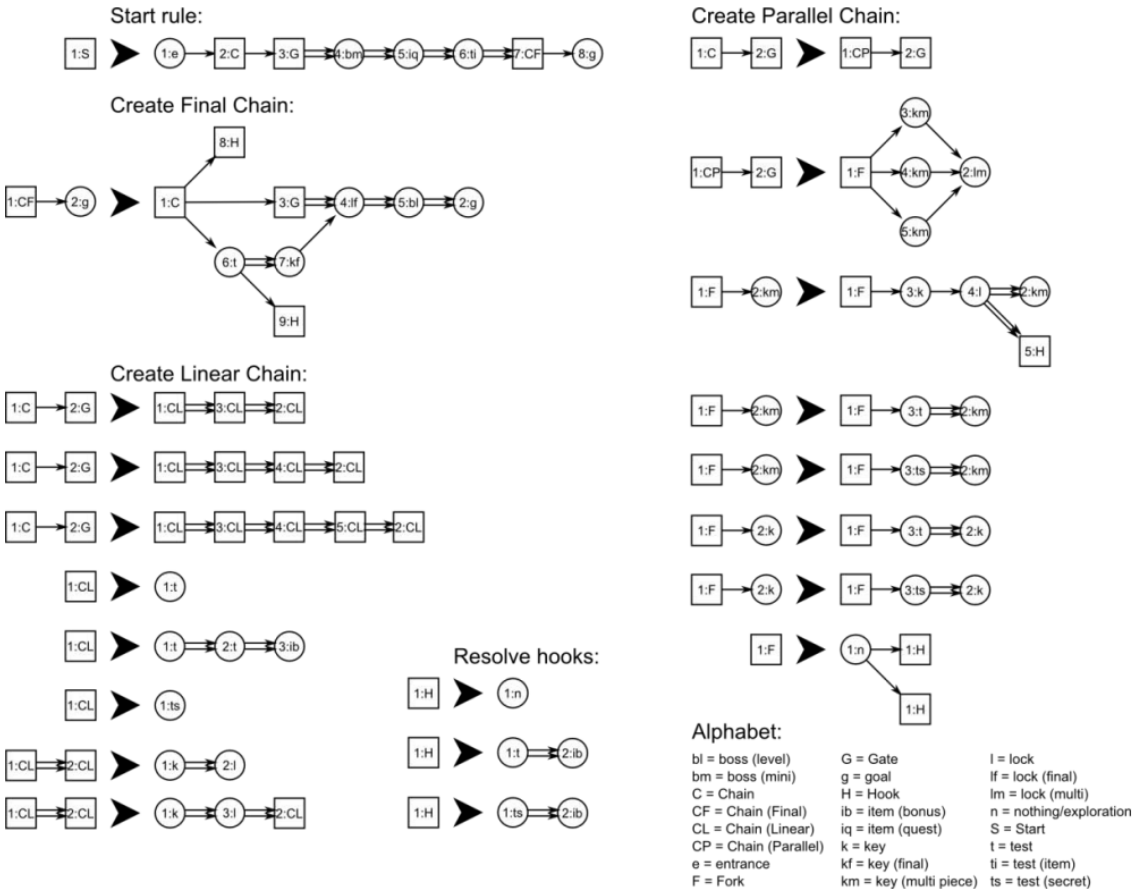


Ilustración 64. Gramática formal propuesta para la generación de grafos de misiones (J. Dormans, 2010)

Como podemos ver, en la imagen aparecen las reglas de producción, el alfabeto con los símbolos no terminales (en mayúsculas) y terminales (en minúsculas), y el axioma, que en este caso sería el nodo *Start*.

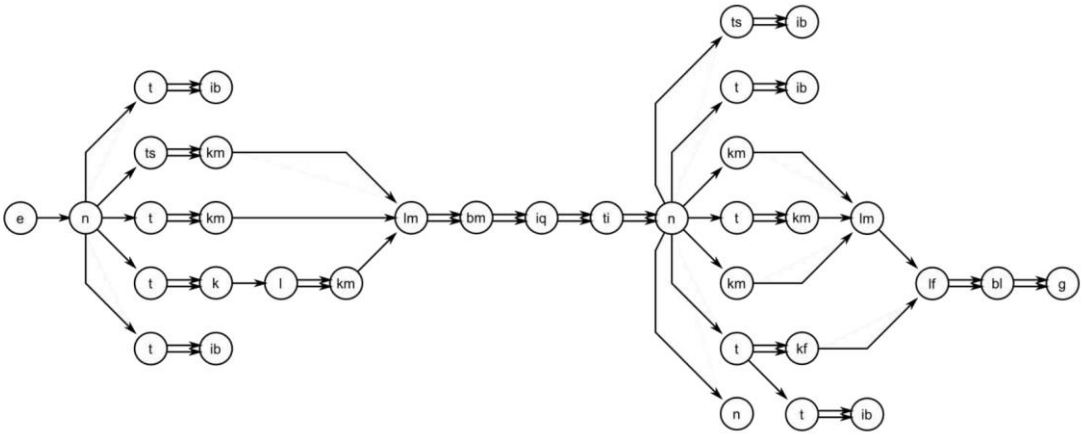


Ilustración 65. Ejemplo de grafo de misión generado con dicha gramática (J. Dormans, 2010)

Una vez hemos visto el algoritmo para generar el grafo de misión, vamos a pasar a ver en detalle cada una de las etapas de nuestro algoritmo, pues la generación del grafo no es la única parte ya que queremos generar todo un escenario de **islas flotantes**:

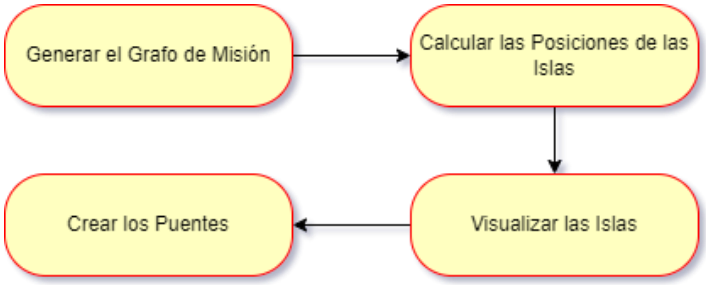


Ilustración 66. Esquema general del algoritmo

Vamos a comentar en detalle cada una de las etapas:

- **Generar el grafo de misión.** Pese a que ya hemos comentado en esencia cuál será el algoritmo que seguiremos, sí que es cierto que convendría ver en más detalle algunos aspectos. Para ello, vamos a hacer uso de un diagrama para ilustrar una secuencia de generación de grafo como ejemplo ilustrativo:

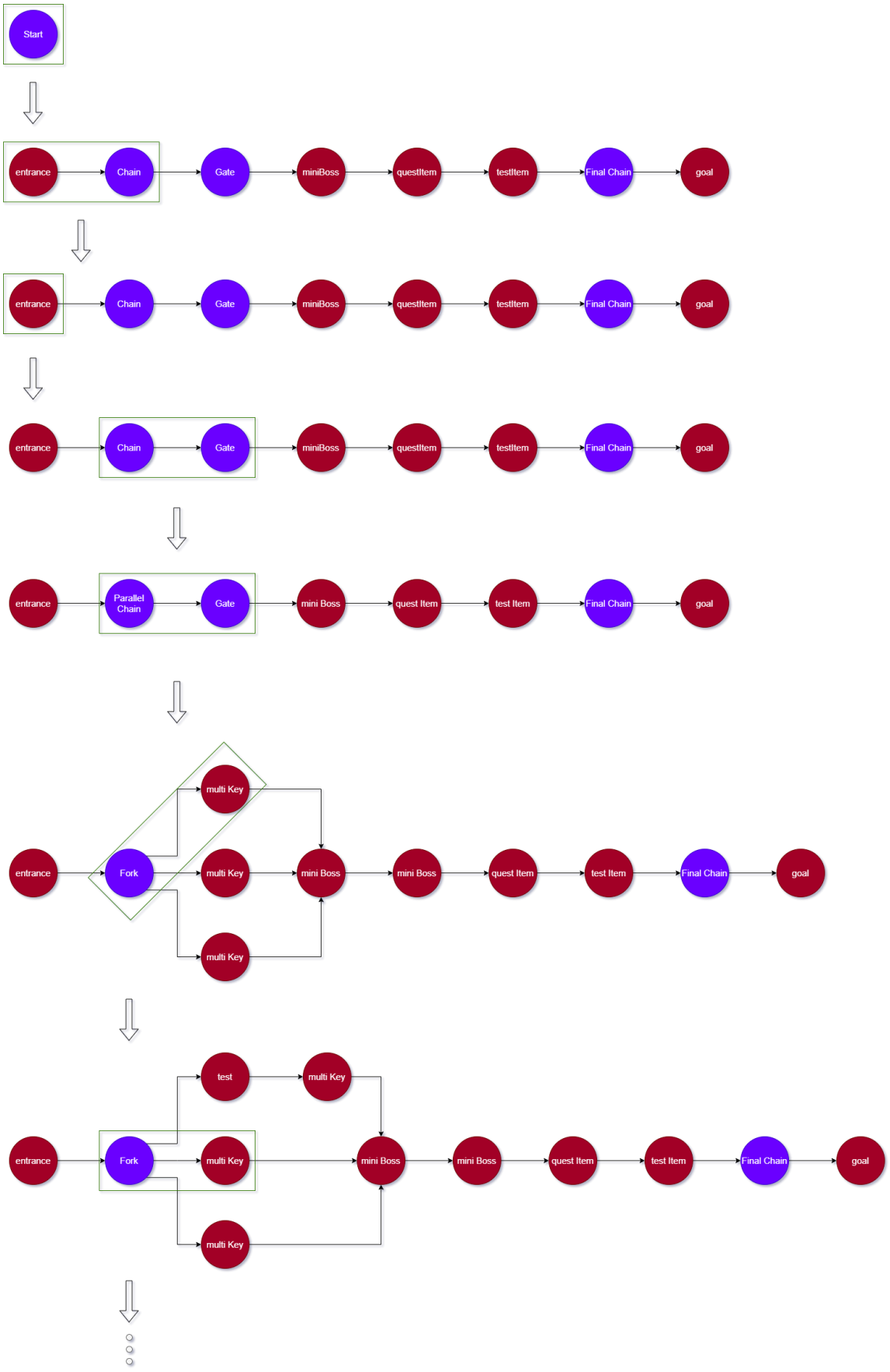


Ilustración 67. Ejemplo de los primeros pasos de la generación de un grafo, con los nodos terminales en rojo y los no terminales en azul.

Como se puede ver en este ejemplo, el algoritmo parte del **axioma**, y, en un proceso recursivo, realiza los siguientes pasos:

1. Comprueba si existe alguna regla aplicable al nodo actual y al primero de sus hijos.
    a. De ser así, elige una aleatoria entre las posibles que haya y la aplica, sustituyendo a ambos nodos.
    b. De no ser así, pasa al siguiente hijo y hace la misma comprobación.
2. Una vez se han aplicado todas las reglas posibles entre nodo e hijos, se comprueba si existe alguna regla aplicable al nodo por sí solo.
    a. De ser así, la aplica.
    b. De no ser así, el proceso recursivo pasa a centrarse ahora en el primero de sus hijos y se repite el proceso.

En la imagen, el rectángulo simboliza el/los nodo(s) que se están comprobando en ese momento: se comprueba el nodo *Start*, que no tiene hijos, por lo que se reemplaza únicamente a sí mismo; se comprueba el nodo *entrance* y su primer (y único) hijo, el nodo *Chain*, para los cuales no existe ninguna regla; se comprueba el nodo *entrance* por sí mismo, para el cual tampoco existe regla; se pasa a comprobar el nodo *Chain* con su hijo el nodo *Gate*, para los que sí hay reglas, y se elige una aleatoriamente, etc…

- **Calcular las posiciones de las islas.** En esta etapa, el algoritmo calculará los centros en los que se dispondrán las diferentes islas en el cielo. Para ello, generamos tantos **puntos aleatorios** dentro de una **circunferencia** dada a partir de su centro y radio como nodos tenga el grafo de misión, de manera que, una vez colocado el punto, se comprueba que esté a una **distancia mínima** del resto de puntos; de no ser así, se calculan los vectores que alejan a este punto de cada uno de los otros y se suman, de manera que el punto se aleja, en principio, de todos los demás, hasta que se cumpla que esté a una distancia mínima de todos los demás puntos.

$$v = \lambda \sum_{i=1}^{n} \frac{1}{|p - p_i|} (p - p_i)$$

Siendo v el vector de desplazamiento del nuevo punto, λ el factor por el que se multiplica el vector, n el número de puntos actualmente colocados, p la posición actual del nuevo punto y p_i las posiciones de cada uno de los demás puntos. Lo que nos quiere decir esta fórmula es que, mientras más lejos esté el punto p_i, menos contará para el cómputo del vector de desplazamiento final.

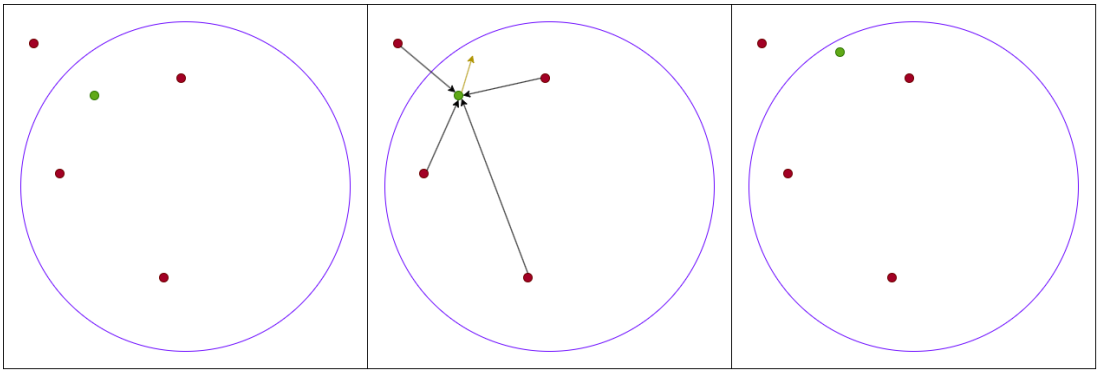


Ilustración 68. En la primera imagen, se puede ver la colocación de un nuevo punto en el interior de la circunferencia; en la segunda, cálculo del vector de desplazamiento; en la tercera, desplazamiento del punto

Como podemos ver en la imagen el punto se aleja de los demás, aunque tras desplazarlo sigue estando demasiado cerca de los demás, por lo que en siguientes iteraciones se repetirá el proceso hasta que esté lo suficientemente lejos.

Una vez tenemos la cantidad adecuada de centros, toca **asignarles un nodo del grafo de misión** a cada uno. La estrategia será muy simple: comenzamos asignándole al punto más inferior en el eje y el nodo inicial, de manera que a su primer hijo le asignamos el punto más cercano, a su segundo hijo el segundo más cercano, etc… y así con todos los nodos. Asimismo, la altura a la que se sitúe la isla vendrá determinada por el **nivel** en el que se halle el nodo dentro del grafo. Para ilustrar esto, nótese que los hijos de un determinado nodo estarán a la misma altura.

- **Visualizar las islas.** En esta etapa, el algoritmo creará dentro del mundo del juego las islas propiamente dichas a partir de las estructuras de datos que tengamos. Fundamentalmente, instanciará un *asset* u otro de isla dependiendo del nodo de grafo de misión que tenga asignado. Hablaremos sobre las diferentes posibilidades más en detalle en otros apartados.

- **Crear los puentes.** Por último, necesitamos conectar las islas de alguna forma. El problema de usar puentes sin más es que existe cierta posibilidad de que haya colisiones entre un puente y otro o entre un puente y una isla, y esto no es para nada trivial de solucionar. La solución que encontramos fue usar **teletransportadores**: básicamente, hacemos que, para ir de una isla a otra, el personaje deba teletransportarse, de forma que habrá dos de estos teletransportadores, uno por isla, colocados en el extremo de un puente que sale de cada isla apuntando aproximadamente a la otra.

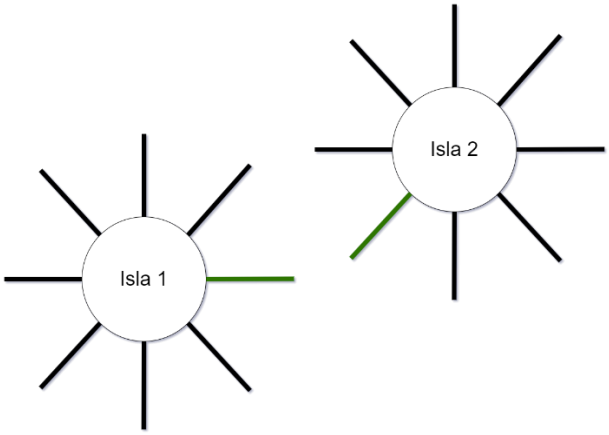


Ilustración 39. Dos islas junto con sus puentes que la conectan, en verde, y las posibles direcciones que un puente puede tomar, en negro. Nótese que el número de posibles direcciones en la imagen no tiene por qué coincidir con el usado en el proyecto

Lo que hacemos es que dichos puentes solo puedan tomar **ciertas direcciones** para que no se solapen demasiado y hacemos que se coja la dirección que apunte con mayor precisión a la otra isla, de forma que colocamos los teletransportadores en el extremos de los puentes.

2.2.2.3 Diagramas de clases

A continuación, vamos a presentar los diagramas de clases relacionados con la generación procedural:

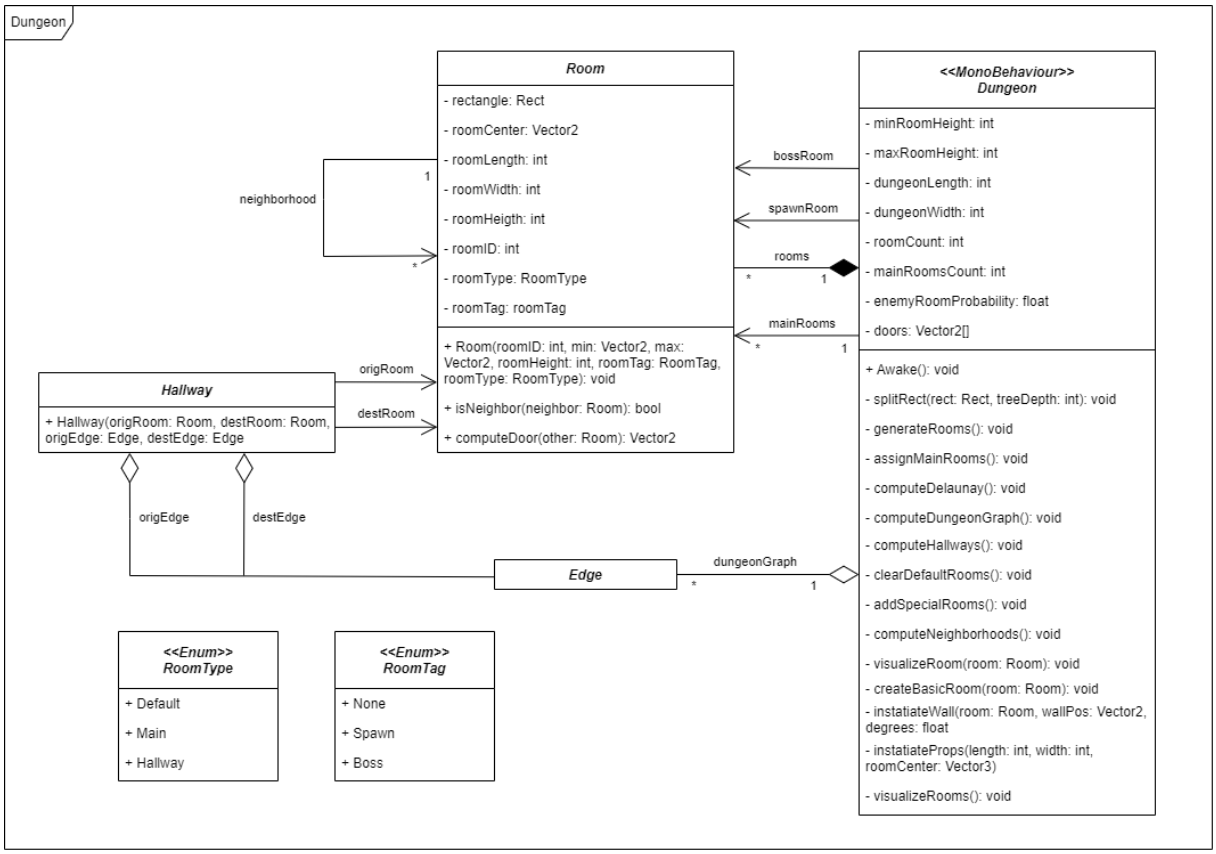


Ilustración 69. Diagrama de clases asociado a la generación procedural del castillo

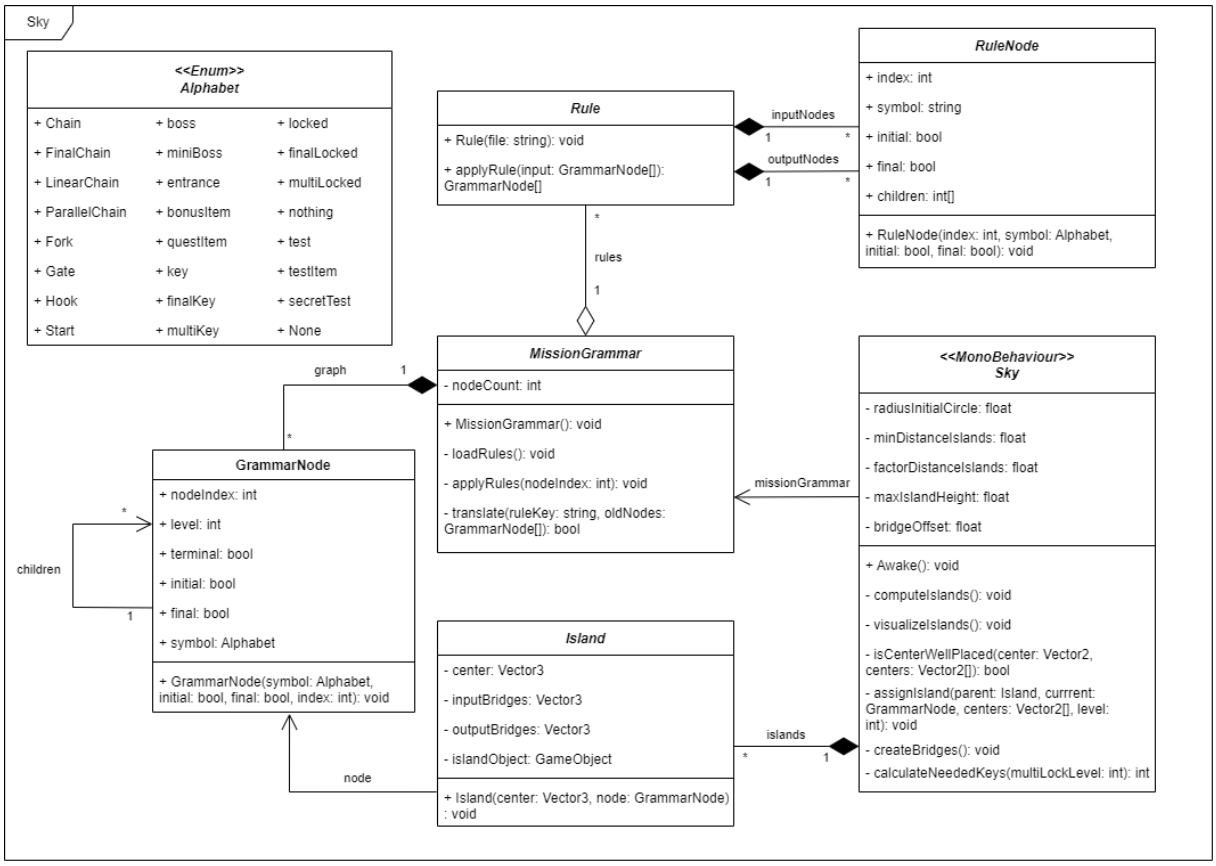


Ilustración 70. Diagrama de clases asociado a la generación de las islas flotantes

Nótese que, como en diagramas de clases anteriores, ha desaparecido cierto nivel de detalle que se ha creído conveniente no incluir.

2.2.3 Análisis y diseño de la interfaz

2.2.3.1 Casos de uso

Caso de uso	Empezar juego
Actor	Jugador
Contexto	Menú principal
Condición previa	Que el jugador se encuentre en el menú principal
Operaciones básicas	1) El jugador pulsa en el elemento correspondiente de la interfaz para iniciar una partida. 2) El sistema le lleva a una nueva interfaz donde se le presentan las diferentes opciones de modo de juego.

	3) El jugador pulsa en el elemento de la interfaz correspondiente a un determinado modo de juego. 4) El sistema le muestra una pantalla de carga y carga ese modo de juego, generando un escenario.
Alternativas	

Tabla 27. Caso de uso (Empezar juego)

Caso de uso	Ajustar volumen
Actor	Jugador
Contexto	Menú principal o menú de pausa
Condición previa	Que el jugador se encuentre en el menú principal o en el menú de pausa
Operaciones básicas	1) El jugador pulsa en el elemento correspondiente de la interfaz para opciones. 2) El sistema le lleva a una nueva interfaz donde se le presentan los diferentes parámetros que puede modificar. 3) El jugador ajusta el volumen a través de su respectivo elemento de la interfaz según su gusto. 4) El sistema cambia el volumen acorde a lo que el jugador ha dictaminado.
Alternativas	

Tabla 28. Caso de uso (Ajustar Volumen)

Caso de uso	Seleccionar resolución
Actor	Jugador
Contexto	Menú principal
Condición previa	Que el jugador se encuentre en el menú principal
Operaciones básicas	1) El jugador pulsa en el elemento correspondiente de la interfaz para opciones. 2) El sistema le lleva a una nueva interfaz donde se le presentan los diferentes parámetros que puede modificar. 4) El jugador pulsa en el elemento de la interfaz que muestra las resoluciones disponibles.

	5) El sistema le muestra las resoluciones disponibles. 6) El jugador selecciona una resolución de pantalla entre las que se le ofrece. 7) El sistema cambia la resolución de pantalla.
Alternativas	

Tabla 29. Caso de uso (Seleccionar Resolución)

Caso de uso	Seleccionar calidad de los gráficos
Actor	Jugador
Contexto	Menú principal
Condición previa	Que el jugador se encuentre en el menú principal
Operaciones básicas	1) El jugador pulsa en el elemento correspondiente de la interfaz para opciones. 2) El sistema le lleva a una nueva interfaz donde se le presentan los diferentes parámetros que puede modificar. 4) El jugador pulsa en el elemento de la interfaz que muestra las calidades de gráficos disponibles. 5) El sistema le muestra las calidades disponibles. 6) El jugador selecciona una calidad de gráficos entre las que se le ofrece. 7) El sistema cambia la calidad de los gráficos.
Alternativas	

Tabla 30. Caso de uso (Seleccionar calidad de los gráficos)

Caso de uso	Seleccionar pantalla completa
Actor	Jugador
Contexto	Menú principal
Condición previa	Que el jugador se encuentre en el menú principal
Operaciones básicas	1) El jugador pulsa en el botón correspondiente de la interfaz para opciones.

	2) El sistema le lleva a una nueva interfaz donde se le presentan los diferentes parámetros que puede modificar. 3) El jugador pulsa en el elemento de la interfaz correspondiente. 4) El sistema comprueba si el elemento de la interfaz estaba marcado o desmarcado. 5) El sistema cambia el modo de pantalla en concordancia.
Alternativas	4ª) ¿Estaba antes marcada la opción de pantalla completa? 4ª.1) Si lo estaba, entonces se desmarca y se sigue. 4ª.2) Si no, entonces se marca y se sigue.

Tabla 31. Caso de uso (Seleccionar pantalla completa)

Caso de uso	Salir del videojuego
Actor	Jugador
Contexto	Menú principal
Condición previa	Que el jugador se encuentre en el menú principal
Operaciones básicas	1) El jugador pulsa en el elemento correspondiente a salir del videojuego de la interfaz. 2) El sistema detiene el funcionamiento del videojuego.

Tabla 32. Caso de uso (Salir del videojuego)

2.2.3.2 Diagramas de actividad

Nótese que solo se ha incluido un diagrama de actividad ya que el resto de casos de uso son triviales, en el sentido de que no presentan ramificaciones.

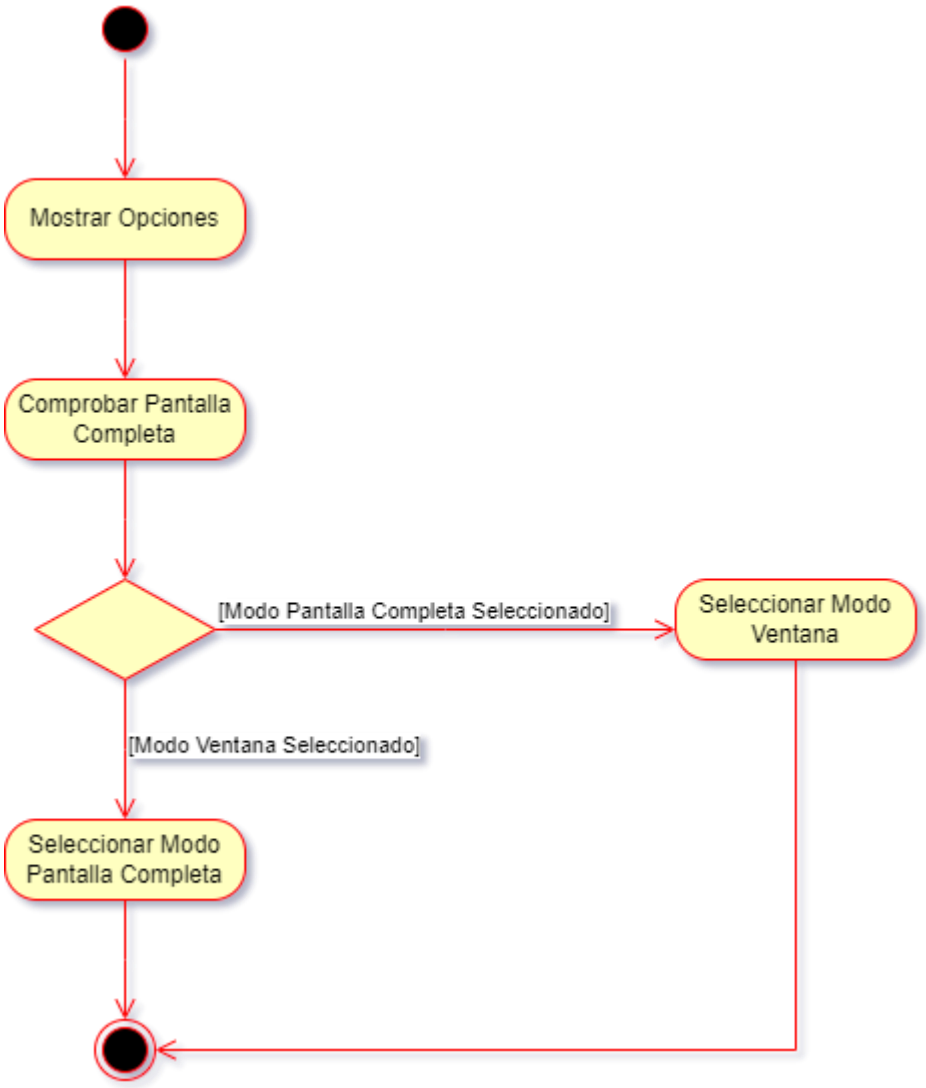


Ilustración 71. Diagrama de actividad (Seleccionar pantalla completa)

2.2.3.3 Sketchs y storyboard

A partir de los casos de uso y diagramas de actividad vamos a realizar una serie de **sketchs** para visualizar cómo será la interfaz de usuario:

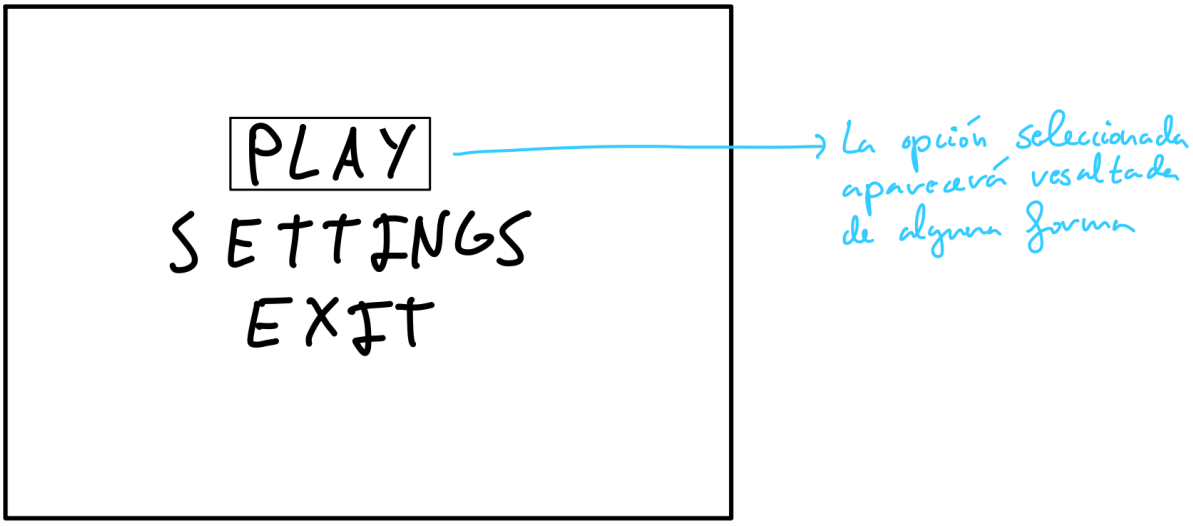


Ilustración 72. Sketch del menú principal

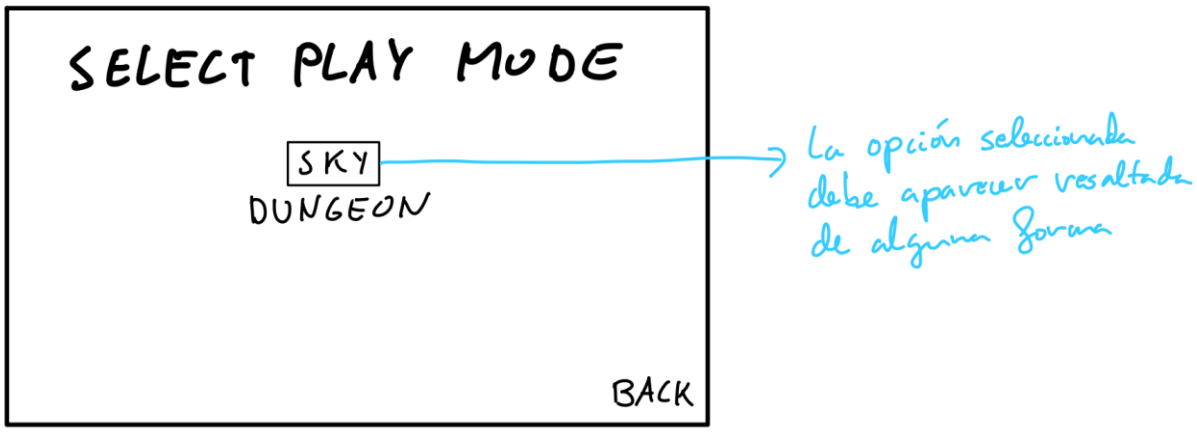


Ilustración 73. Sketch del menú de selección de modo de juego

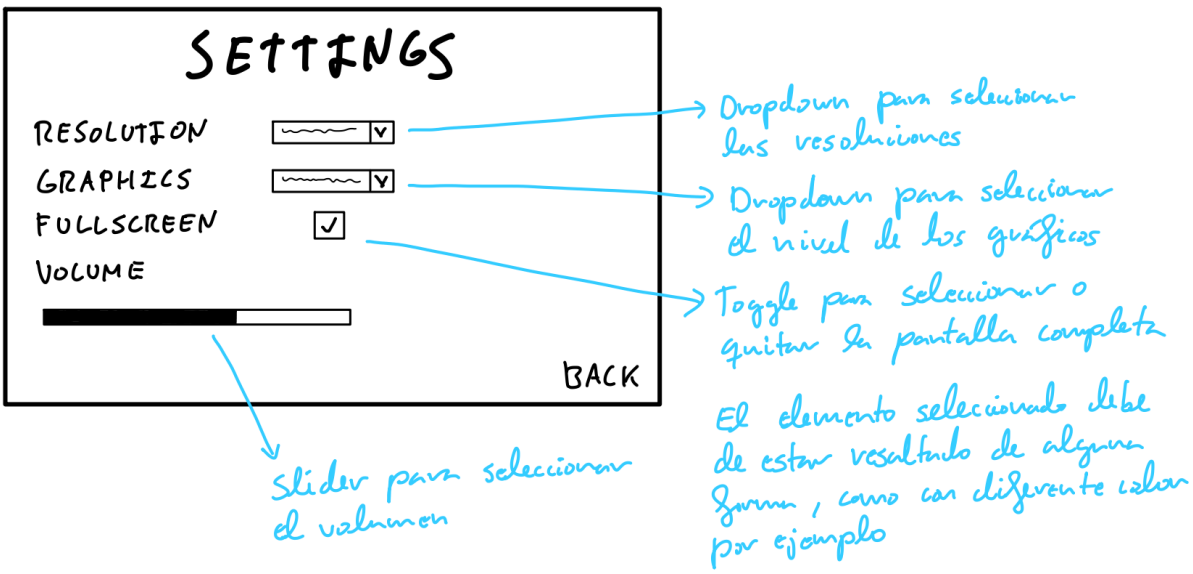


Ilustración 74. Sketch del menú principal de opciones

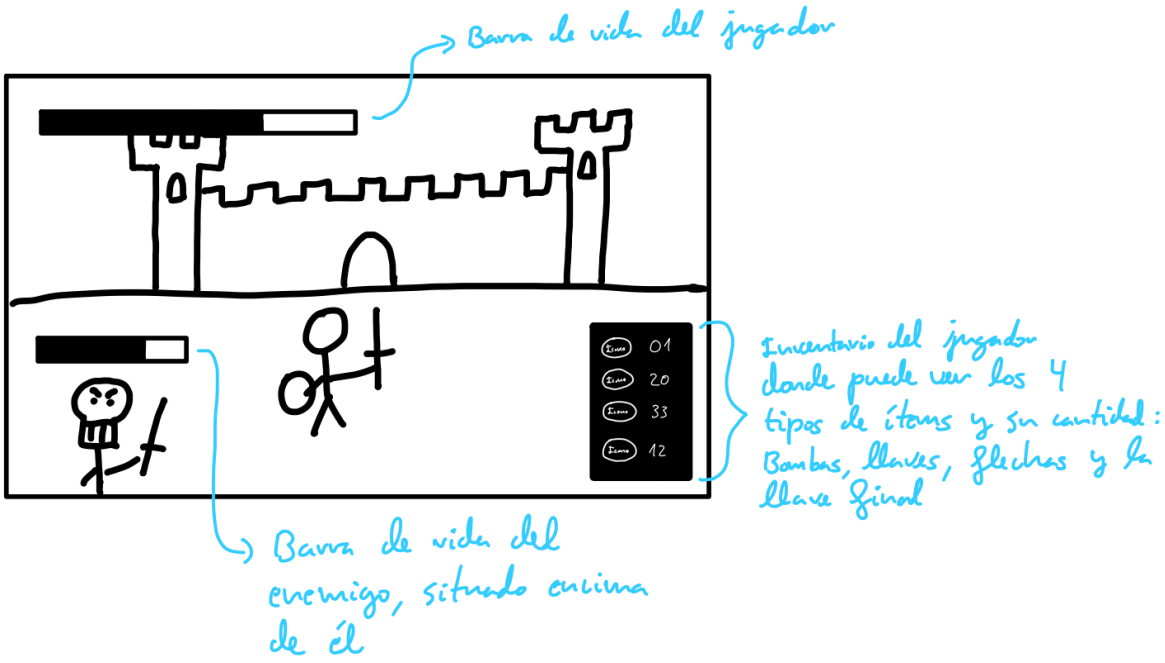


Ilustración 75. Sketch de la interfaz principal del juego

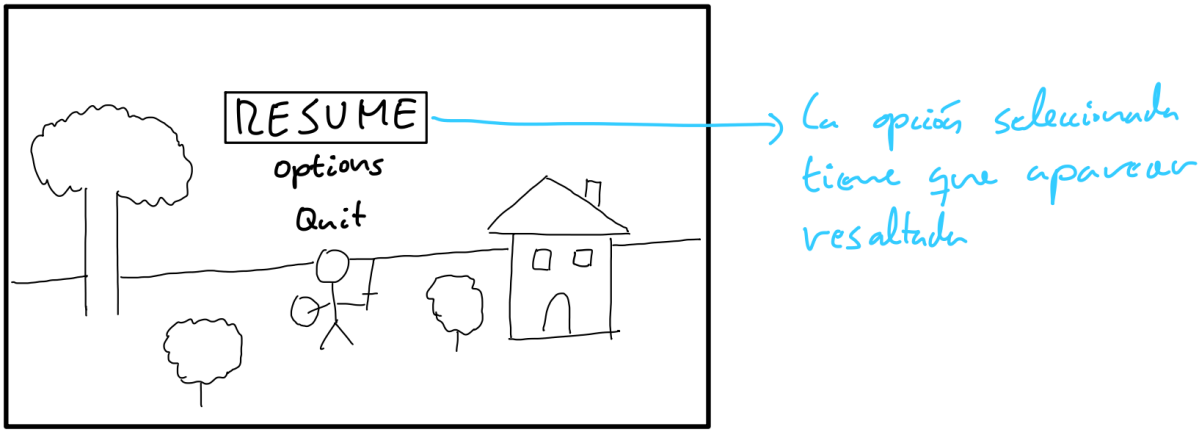


Ilustración 76. Sketch del menú de pausa

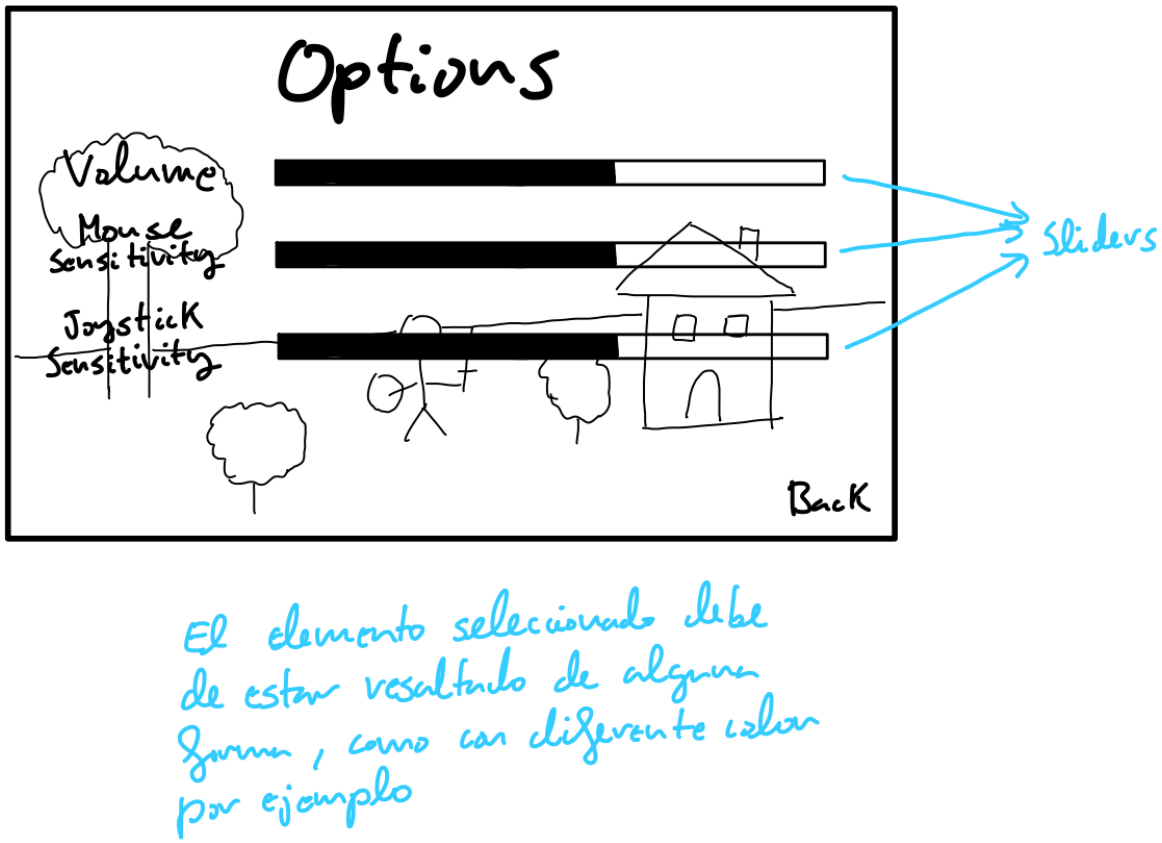


Ilustración 77. Menú de opciones durante el juego

Notar que el menú de opciones durante el juego tal vez no se vea todo lo bien que debería en este boceto debido a que el juego se encuentra de fondo pausado, pero, como veremos, oscureceremos el fondo lo suficiente para que no haya problemas en este sentido, lo cual se deja intuir en el dibujo ya que está más fino en el menú de pausa.

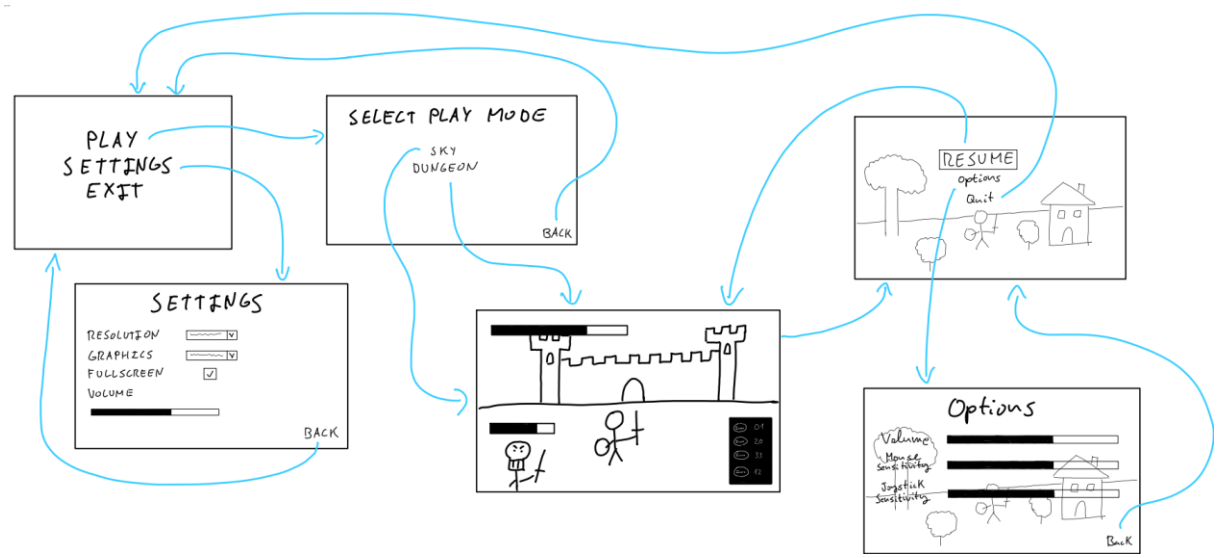


Ilustración 78. Storyboard general

2.2.4 Diagrama de clases global

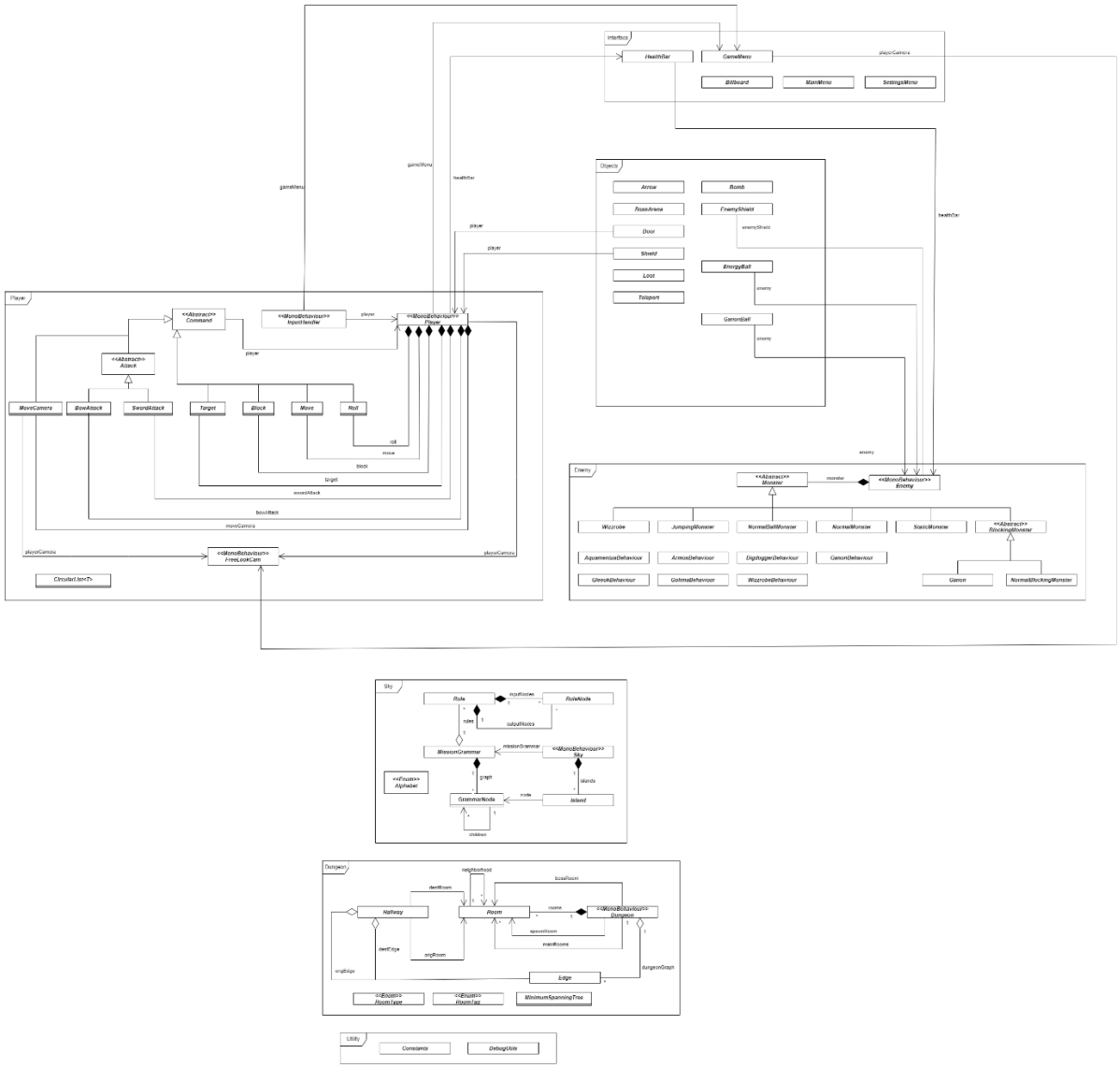


Ilustración 79. Diagrama de clases global

3 DESARROLLO

En este apartado vamos a realizar una exploración de las diferentes iteraciones en las cuales se ha dividido el desarrollo del proyecto y de las funcionalidades que implementa. En concreto, vamos a realizar primero un resumen de las iteraciones y después vamos a pasar a detallar las funcionalidades.

3.1 Iteraciones

1. **Primera iteración.** Comienza con la inicialización del repositorio, de manera que, a lo largo de esta iteración, se implementó el **sistema de *input*** y el **movimiento básico** de la cámara y el personaje, con las animaciones de andar y demás, de manera que se creó el andamiaje básico para el resto de acciones del jugador que se implementaron más adelante.
2. **Segunda iteración.** En esta iteración se implementó el **ataque básico** con la espada que el personaje puede realizar.
3. **Tercera iteración.** La mecánica de **rodar** fue lo que se llevó a cabo a lo largo de este período.
4. **Cuarta iteración.** Aquí se implementó la mecánica de **bloqueo** que el personaje puede realizar con el escudo.
5. **Quinta iteración.** Fundamentalmente, aquí implementamos todo lo de importancia para el **sistema de combate**, a saber: colisiones, la mecánica de fijar enemigos, de bloquear, recibir y propinar golpes, y se empezó con la creación de enemigos.
6. **Sexta iteración.** Aquí sencillamente se completó la mecánica de **correr**.
7. **Séptima iteración.** Básicamente fue todo lo concerniente al **arco** lo que se realizó en esta etapa.
8. **Octava iteración.** En este incremento se procedió a crear todos los **enemigos** que pueblan el mundo del videojuego.
9. **Novena iteración.** Durante esta iteración se creó la **generación procedural del castillo**.
10. **Décima iteración.** A lo largo del tiempo que duró se llevó a cabo la mecánica de la **bomba**.
11. **Undécima iteración.** Aquí lo que se hizo fue realizar la implementación de la **gramática** y de determinados **objetos interactuables**.
12. **Duodécima iteración.** Fundamentalmente, se llevó a cabo la implementación de la **generación procedural de las islas** y se

solucionaron ciertos **problemas** con la generación procedural de la mazmorra y la cámara.

13. **Decimotercera iteración.** Se completó todo lo concerniente a la **interfaz de usuario**.
14. **Decimocuarta iteración.** Durante este período se realizaron varias cosas: se añadieron **efectos audiovisuales**, se implementaron los estados de **victoria** y **derrota**, y se solucionaron ciertos ***bugs*** concernientes a la mecánica de fijar y a la generación procedural del castillo.

3.2 Funcionalidades

3.2.1 Sistema de input

El proyecto se empezó creando todo lo concerniente al ***input*** del usuario. Para ello, se hizo uso del nuevo *Input System* de *Unity*. Se trata de un sistema mucho más versátil y flexible que el anterior *Input Manager*, aunque algo más complejo, el cual se basa en *callbacks*. Para nuestro proyecto, empezamos creando un ***Input Action Asset***, el cual es un *asset* que nos permite establecer y modificar nuestro sistema de *input* de una forma centralizada a través de una interfaz.

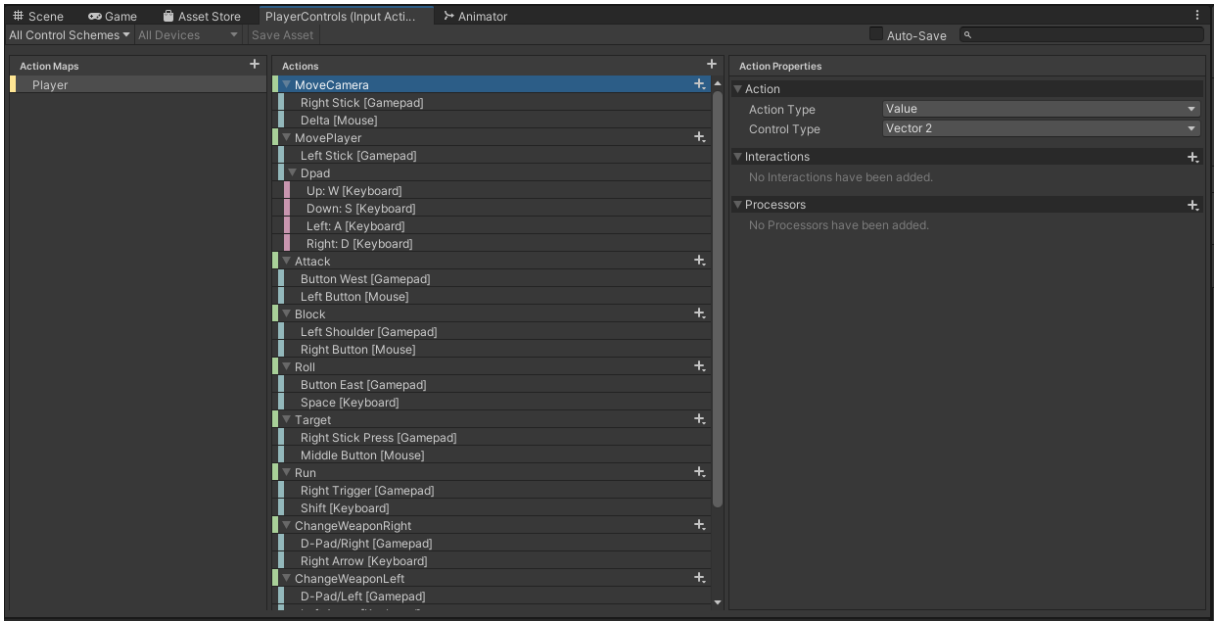


Ilustración 80. Input Action Asset de nuestro proyecto

Esta interfaz se divide en diferentes partes: por un lado, tenemos los ***Action Maps***, que separan diferentes tipos de actividades, de manera que podemos tener diferentes grupos de acciones según el contexto como controles de menú, jugabilidad, conducción, etc… Para nuestro proyecto solo vamos a necesitar uno. Por otro lado, tenemos las ***Actions***, que son la forma que tiene el sistema de abstraer las acciones del *input* concreto que se esté usando, de manera que son eventos que activan funciones en la escena. Cada *Action* tiene una serie de propiedades que indican su tipo y el valor que devuelven; por ejemplo, la acción *MoveCamera* es de tipo *Value* y devuelve un *Vector2* mientras que hay otras como *Attack* que son de tipo *Button*. Como podemos ver, cada acción puede tener diferentes ***Bindings***, esto es, los controles físicos propios de cada forma de *input* (teclas del teclado, botones de un mando, etc…). Cada *Binding* además está asociado a un ***Control Scheme***, que es la forma que tiene este sistema de agrupar los *Bindings* según el tipo de interfaz física que estemos usando (teclado y ratón, mando, etc…) (J. French, 2022).

Una vez tenemos establecido esto, necesitamos alguna forma de que se nos informe de los **eventos** a través de código. Para ello, vamos a hacer uso de la clase ***InputHandler***. Se trata de una clase bastante sencilla que únicamente se encarga de estar a la escucha de estos eventos por medio de *callbacks*.

```
controls.Player.MoveCamera.performed += context => performMoveCameraCallback(context);
controls.Player.MoveCamera.canceled += context => cancelMoveCameraCallback();

controls.Player.MovePlayer.performed += context => performMovePlayerCallback(context);
controls.Player.MovePlayer.canceled += context => cancelMovePlayerCallback();
controls.Player.Run.performed += context => performRunningCallback();
controls.Player.Run.canceled += context => cancelRunningCallback();

controls.Player.Attack.started += context => startAttackPlayerCallback();

controls.Player.Block.performed += context => performBlockPlayerCallback();
controls.Player.Block.canceled += context => cancelBlockPlayerCallback();

controls.Player.Roll.started += context => startRollPlayerCallback();

controls.Player.Target.started += context => startTargetPlayerCallback();

controls.Player.ChangeWeaponRight.started += context => startChangeWeaponRightCallback();
controls.Player.ChangeWeaponLeft.started += context => startChangeWeaponLeftCallback();

controls.Player.ThrowBomb.started += context => startThrowBombCallback();

controls.Player.Pause.started += context => startPauseGameCallback();
```

Ilustración 81. Inicialización de callbacks en la clase InputHandler

Como se puede ver en la ilustración, las *Input Actions* se asocian a *callbacks*, de forma que podemos ver que existen tres opciones: ***started***, ***performed*** y ***canceled***. La primera se asocia a la primera señal del *input*; esto es, si el *input* es un botón, entonces el *callback* únicamente se llamará una vez, cuando se empieza a pulsar el botón, pero si se deja pulsado no pasará nada. La segunda tiene que ver con el resto de señales; es decir, si de nuevo tenemos un botón, el *callback* se llamará una vez por *frame* durante el tiempo que lo mantengamos pulsado. La última está relacionada con dejar de recibir señales, de manera que, si el *input* se trata de un botón, este *callback* se llamará cuando se deje de pulsarlo.

3.2.2 Movimiento de la cámara y el personaje

Lo primero de todo que se realizó para esto fue establecer qué **modelo** de personaje sería usado por el jugador como avatar. Al final, nos decantamos por este proporcionado por el paquete *Modular RPG Hero PBR* de la tienda de *assets* de *Unity*, que cuenta además con todas las animaciones necesarias para nuestro proyecto, así como del modelo de la espada, el arco y el escudo. Notar que para todo lo concerniente al movimiento de la cámara y del personaje se consultó Brackeys (2020a).

Ilustración 82. Modelo usado como avatar del jugador

Se procedió a continuación a la creación de una clase ***Player*** que encapsulase toda la funcionalidad asociada al jugador, de forma que sería la clase central que se encargue de llevar a cabo los diferentes eventos de acuerdo a lo que establezca el *input* por medio de la clase *InputHandler*. No obstante, se encapsularon las propias acciones del personaje en otras clases, de forma que creamos además la clase abstracta ***Command*** para que sirviera de base para estas otras clases. Fundamentalmente, la idea era que *InputHandler* le comunicase a *Player* estos eventos, y esta a su vez lo hiciese a los diferentes hijos de *Command* para que ejecutasen estas acciones (por lo general, animaciones, activación/desactivación de *colliders* y demás lógica).

```
/// <summary>
/// This method is called in the Player update and
/// performs each frame all the code needed by the player for
/// doing a given action, like rolling or attacking.
/// </summary>
13 referencias
public abstract void execute();

1 referencia
public bool isAttacking()
{
    return this.animator.GetBool(Constants.IS_ATTACKING_CONDITION_BOOL);
}

2 referencias
public bool isBlocking()
{
    if (player.getSelectedWeapon() == Constants.WeaponSelector.Bow)
        return false;

    return this.animator.GetBool(Constants.IS_BLOCKING_CONDITION_BOOL);
}

10 referencias
public bool isRolling()
{
    return this.animator.GetBool(Constants.IS_ROLLING_CONDITION_BOOL);
}

6 referencias
public bool isBeaten()
{
    return this.animator.GetBool(Constants.IS_BEATEN_CONDITION_BOOL);
}

7 referencias
public bool isTargeting()
{
    return this.animator.GetBool(Constants.IS_TARGETING_CONDITION_BOOL);
}

5 referencias
public bool isRunning()
{
    return this.animator.GetBool(Constants.IS_RUNNING_CONDITION_BOOL);
}
```

Ilustración 83. Métodos de la clase Command

Como podemos ver en el código, la clase *Command* proporciona el método abstracto ***execute***, el cual se llama cada *frame* debido a que prácticamente todas las acciones deben estar sostenidas en el tiempo al requerir de cosas como por ejemplo actualizar constantemente la posición de la cámara en el caso de la acción de mover la cámara. Además, en otras tantas acciones se deben comprobar constantemente ciertas dependencias, las cuales se pueden ver fácilmente en los diagramas de actividad presentados con anterioridad. Es por ello que la clase *Command* ofrece también métodos para saber si se está ejecutando una cierta acción, lo cual se hace por medio del ***Animator*** del personaje, un tipo de *asset* que nos permite transicionar

de una animación a otra cuando se cumplen determinadas condiciones, que en nuestro caso nos dicta además si una determinada acción se está llevando a cabo.

Así pues, se procedió a la creación de la **cámara**. Para ello, hemos hecho uso del paquete *Standard Assets* que nos implementa ya de por sí una cámara en tercera persona perfectamente funcional. El único incoveniente es que el *script* a través del cual funciona, ***FreeLookCam***, funciona a través del antiguo sistema de *input* de *Unity*, por lo que se realizaron una serie de modificaciones al código bastante simples, que consistían en eliminar la parte en que la Δx y la Δy de la cámara se establecían a través de este sistema de *input* y hacer que se pudieran modificar desde otros *scripts*.

```
1 referencia
public void setX(float x)
{
    x_camera = x;
}

1 referencia
public void setY(float y)
{
    y_camera = y;
}
```

Ilustración 84. Setters de los deltas para que se puedan modificar desde otros scripts

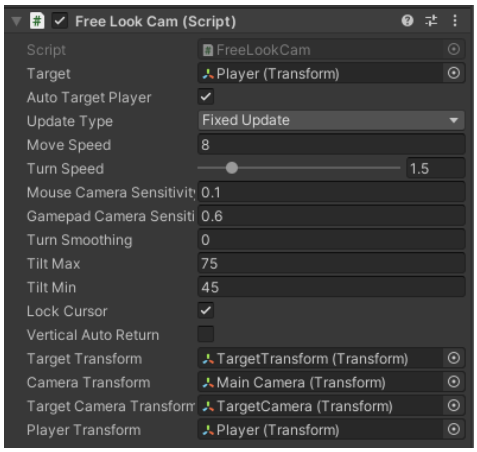


Ilustración 85. Parámetros usados para la cámara principal

Como podemos ver en la imagen, se ha asignado al jugador como el *Transform* que la cámara seguirá. Determinados parámetros han sido añadidos además, los cuales veremos en siguientes apartados. La cámara cuenta además con otro script, ***ProtectCameraFromWallClip***, que hace que la cámara se choque con los objetos y no los atraviese. Este *script* no se ha modificado.

Para realizar la acción de mover la cámara se creó un *script*, ***MoveCamera***, que hereda de *Command* y que cuenta con el siguiente contenido:

```
/// <summary>
/// This method basically sets the X and Y coordinates of the camera.
/// </summary>
13 referencias
public override void execute()
{
    if (!isTargeting())
    {
        playerCamera.setX(cameraRotation.x);
        playerCamera.setY(cameraRotation.y);
    }
}
```

Ilustración 86. Método execute de MoveCamera

```
/// <summary>
/// Callback for stopping the camera movement, called whenever we don´t move the camera.
/// </summary>
1 referencia
public void stopCamera()
{
    cameraRotation = Vector2.zero;
}

/// <summary>
/// Callback for moving the camera, called whenever we are moving the camera with the mouse.
/// </summary>
/// <param name="cameraRotation">Rotation of the camera.</param>
1 referencia
public void setCameraMouseRotation(Vector2 cameraRotation)
{
    this.cameraRotation = cameraRotation * player.getPlayerCamera().getMouseCameraSensitivity();
}

/// <summary>
/// Callback for moving the camera, called whenever we are moving the camera with the gamepad.
/// </summary>
/// <param name="cameraRotation">Rotation of the camera.</param>
1 referencia
public void setCameraGamepadRotation(Vector2 cameraRotation)
{
    this.cameraRotation = cameraRotation * player.getPlayerCamera().getGamepadCameraSensitivity();
}
```

Ilustración 87. Callbacks de la clase MoveCamera

Como se puede observar en el código, *MoveCamera::execute* no hace más que actualizar los valores de los deltas de la cámara para moverla, atendiendo a si el jugador está fijando a un enemigo, caso en el que la cámara no se debe de mover. En los *callbacks* nos encontramos uno para parar la cámara que será llamado al dejar de tocar el *joystick* o el *ratón*, y dos para establecer la rotación, esto es, los deltas de la

cámara, uno por dispositivo de entrada ya que ratón y mando tendrán diferentes sensibilidades.

En cuanto al movimiento del personaje, se empezó por añadirle al personaje un ***Character Controller***. Se trata de un componente de *Unity* que podría definirse como el "motor del personaje", el cual, entre otras cosas, es muy útil para moverlo en la dirección en la que queramos con la velocidad que deseemos, de manera que *Unity* se encarga internamente de proporcionarle un movimiento fluido y realista sin tener que preocuparnos nosotros de la implementación.

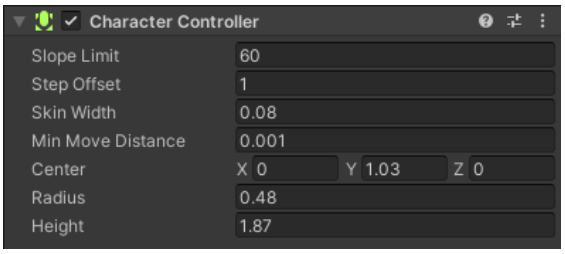


Ilustración 88. Parámetros del Character Controller del personaje del jugador

Antes de nada, hay que notar que, mientras que el *input* del *joystick* de un mando nos proporciona directamente el valor de un vector ya que es una entrada analógica, el esquema WASD del teclado es un poco distinto, ya que *Unity* tiene que componer un eje bidimensional a partir de entradas que no son analógicas, y la forma de componer dicho eje varía según lo establezcamos en el *Input Actions Asset* (J. French, 2022).

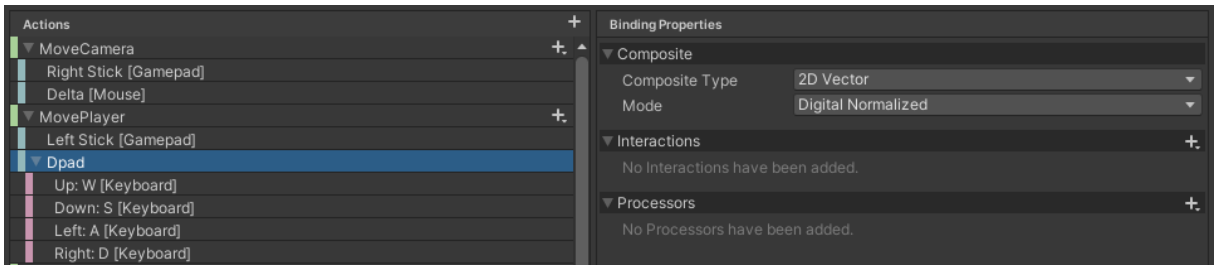


Ilustración 89. Dpad es un eje compuesto con el modo Digital Normalized

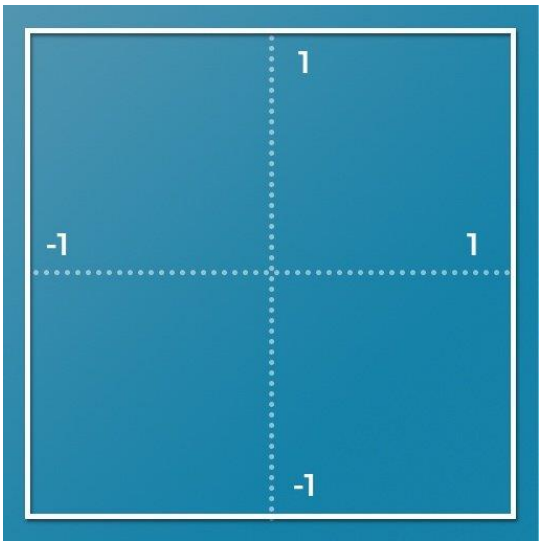


Ilustración 90. Eje compuesto con el modo Digital (J. French, 2022)

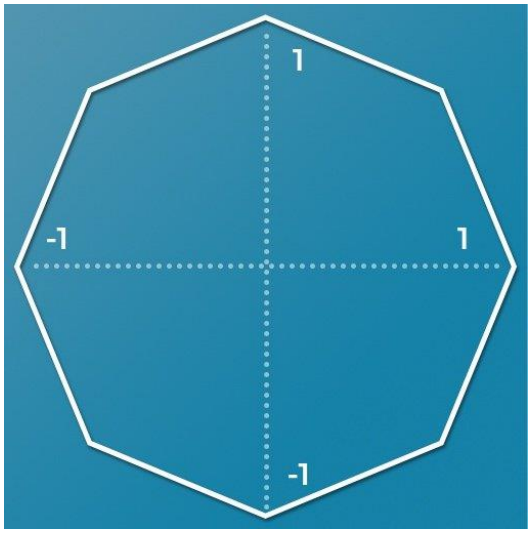


Ilustración 91. Eje compuesto con el modo Digital Normalized (J. French, 2022)

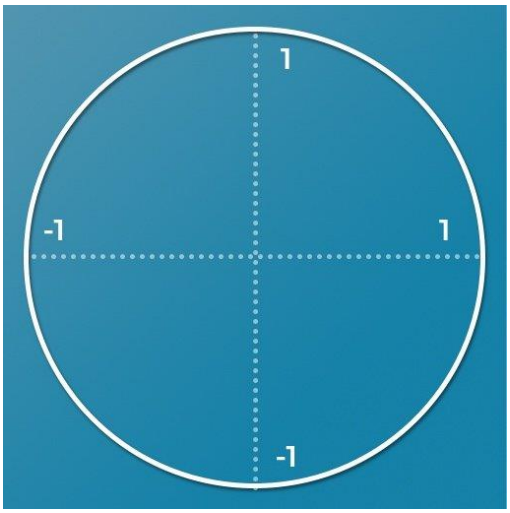


Ilustración 92. Eje compuesto con el modo Analogue (J. French, 2022)

El modo ***Digital*** funciona bien para controles de tanque donde el personaje solo tenga que ir hacia delante, hacia atrás y a los lados, pero para un movimiento en las 8 direcciones como el que queremos tiene un problema: el personaje se mueve más rápido en las diagonales. El modo ***Digital Normalized*** soluciona esto haciendo que los vectores de dirección tengan magnitud 1 en todas las 8 direcciones. El modo ***Analogue*** produce direcciones normalizadas en todos los sentidos, como lo haría un *joystick*, pero solo funciona bien para entradas analógicas.

Pasando ya a nuestro código, en principio mover al personaje sería tan fácil como hacer una llamada al *Character Controller* para moverlo en la dirección deseada con la velocidad correspondiente. No obstante, hay un par de inconvenientes. Primero, el movimiento del personaje sería independiente del de la cámara, lo cual quiere decir, por así decirlo, que su *forward* será siempre el eje x positivo. No obstante, no queremos eso; queremos que su *forward* sea el *forward* de la cámara, que cuando pulsemos para que el personaje avance lo haga en la dirección de la cámara y que él mismo rote hacia esa dirección. Para lo segundo, tenemos que **rotar la dirección de movimiento** de nuestro personaje desde la dirección *forward* (el eje x positivo) hasta la que nos proporciona el *input*. Ese ángulo será el siguiente:

$$\theta = tan^{-1}\frac{x}{y}$$

Siendo el vector $v = (x, y)$ la dirección que indica el *input* en los ejes x y z (puesto que movemos a nuestro personaje por el suelo, el eje y se queda a 0). Ahora bien,

esta fórmula puede no resultar inmediatamente lógica, así que repasemos el razonamiento.

La tangente de un ángulo θ se define como la razón entre cateto opuesto y cateto contiguo del triángulo rectángulo al que pertenece. Por trigonometría sabemos además que se cumple:

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

Por tanto, podemos decir que:

$$\theta = tan^{-1}\frac{sin\,\theta}{cos\,\theta}$$

La trigonometría nos dice además que, sea $v \in \mathbb{R}^2$ tal que $|v| = 1$, ese vector vendrá representado como un punto en el espacio perteneciente a la **circunferencia unitaria**, es decir, de radio 1 y centrada en el origen de coordenadas, y, por tanto, puesto que la hipotenusa del triángulo resultante será uno, nos queda que:

$$\sin\theta = \frac{a}{h} = \frac{a}{1} = a$$

$$\cos\theta = \frac{b}{h} = \frac{b}{1} = b$$

Siendo a el cateto opuesto, b el contiguo y h la hipotenusa. Sabemos, además, que, en este caso, a coincide con y y b con x.

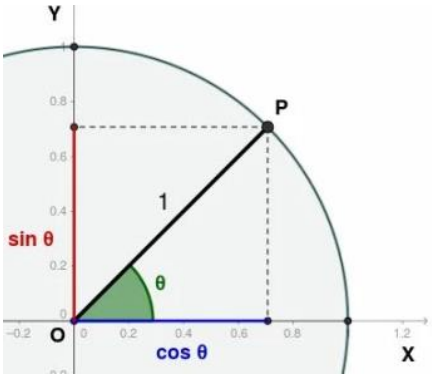


Ilustración 93. Visualización de las funciones trigonométricas para la circunferencia unitaria (Rodney, 2021)

Por todo esto, nos queda lo siguiente:

$$\theta = tan^{-1}\frac{y}{x}$$

No obstante, falta un último detalle. Esta última fórmula nos proporciona este ángulo:

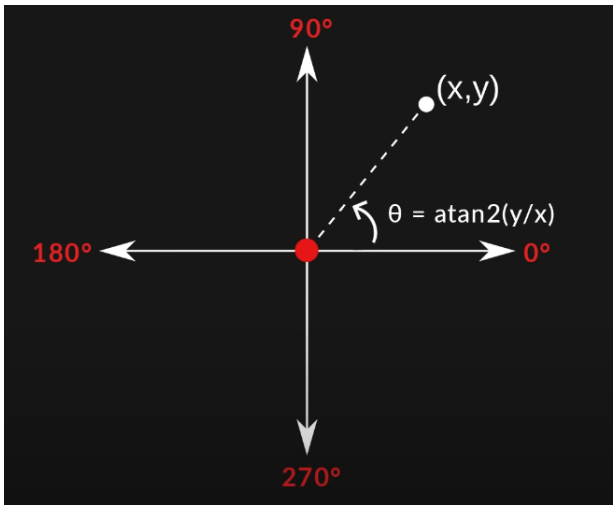


Ilustración 94. Ángulo proporcionado por la fórmula inicial (Brackeys, 2020)

Pero en realidad en *Unity* esto no es así; los 0º estarían situados en el eje x positivo, ya que esa es la dirección *forward* original de nuestro *input*. Por tanto, lo que queremos en realidad es este otro ángulo:

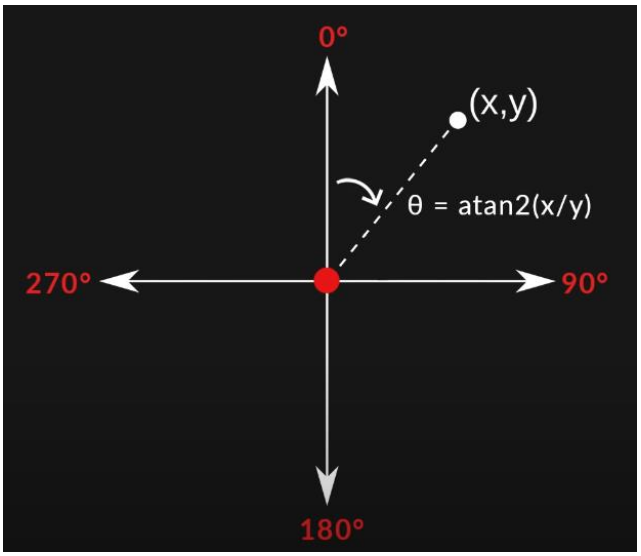


Ilustración 95. Ángulo que queremos calcular para que nuestro personaje rote de acuerdo a la dirección deseada (Brackeys, 2020)

Y ese ángulo se consigue precisamente invirtiendo la fracción, tal y como indicamos en la primera fórmula. Lo único que tendríamos que hacer es rotar al personaje con ese ángulo (siempre y cuando la dirección esté **normalizada**, esto es, que el vector tenga magnitud 1).

Ahora que hemos conseguido que nuestro personaje rote de acuerdo al *input*, nos quedaría hacer que se mueva de acuerdo a la dirección de la cámara. En realidad,

esto es muy sencillo: sencillamente al ángulo que hemos calculado antes le sumamos el **ángulo de Euler** correspondiente al eje y de la cámara, y así siempre que se mueva lo hará tomando como referencia el vector *forward* de la cámara.

Queda otro problema: el personaje se gira instantáneamente, cuando en realidad queremos que lo haga suavemente; es decir, que cuando cambie de dirección, empiece en la dirección inicial y poco a poco vaya adaptándose a la nueva dirección. Para ello, vamos a usar una función que nos proporciona *Unity* llamada *Mathf.SmoothDampAngle*, que gradualmente cambia un ángulo hacia otro ángulo objetivo durante un cierto tiempo que nosotros le especificamos, a una velocidad también determinada.

Por tanto, el código quedaría algo tal que así:

```
float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + Camera.main.transform.eulerAngles.y;
float angle = Mathf.SmoothDampAngle(player.transform.eulerAngles.y, targetAngle,
    ref turnSmoothVelocity, player.getTurnSmoothTime());
player.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

moveDirection = Quaternion.Euler(0.0f, targetAngle, 0.0f) * Vector3.forward;
```

Ilustración 96. Código que calcula la dirección de movimiento según la cámara y de manera suave

Como vemos, al ángulo objetivo (*targetAngle*) se le suma el ángulo de Euler correspondiente de la cámara (para ponerlo en el sistema de referencia de la cámara, de forma que el *forward* se el de la cámara, por así decirlo) y a la dirección de movimiento se la rota según el *targetAngle* multiplicando el *Quaternion* correspondiente con el vector $(0, 0, 1)$.

Una vez tenemos a nuestro personaje funcional, toca añadirle la animación de andar. Para ello, hicimos uso del *Animator* y de su grafo de transiciones, el cual tocaremos en otro apartado. Baste decir que la animación funciona siempre y cuando el personaje se esté moviendo, como es lógico, y que se hace si se cumplen ciertas dependencias que se pueden observar en los diagramas de actividad de apartados anteriores.

3.2.3 Mecánica de ataque de la espada

Para esta mecánica, lo primero que se hizo fue crear la clase ***Attack***, una clase abstracta que hereda a su vez de *Command* y que nos servirá no solo para este tipo de ataque sino para el del arco también.

```
13 referencias
public override void execute()
{
    attack();
}

/// <summary>
/// Function that is called each frame. It varies between attack types.
/// </summary>
3 referencias
public abstract void attack();
/// <summary>
/// Callback called by the Input Handler to start the attack.
/// </summary>
4 referencias
public abstract void setAttack();
/// <summary>
/// Event that is called by the attack animations to stop the attack.
/// </summary>
4 referencias
public abstract void attackHasEnded();
/// <summary>
/// Normally, this is the function that warns the animator controller to set isAttacking to false.
/// </summary>
6 referencias
public abstract void stopAttack();
```

Ilustración 97. Métodos de Attack

Los métodos que se detallan aquí arriba nos servirán para las clases concretas ***SwordAttack*** y ***BowAttack***. Nos vamos a centrar primero en esta.

Echándole un vistazo a la clase, puede resultar confusa ya que posee varios *flags*, así que vamos a explicar aquí el concepto. Lo que queremos hacer es que el personaje sea capaz de realizar **combos**, es decir, que si el jugador indica que quiere atacar mientras el personaje aún está en la animación de ataque, este ataque se guarda en memoria hasta que el primer ataque acabe y la siguiente animación sea diferente.

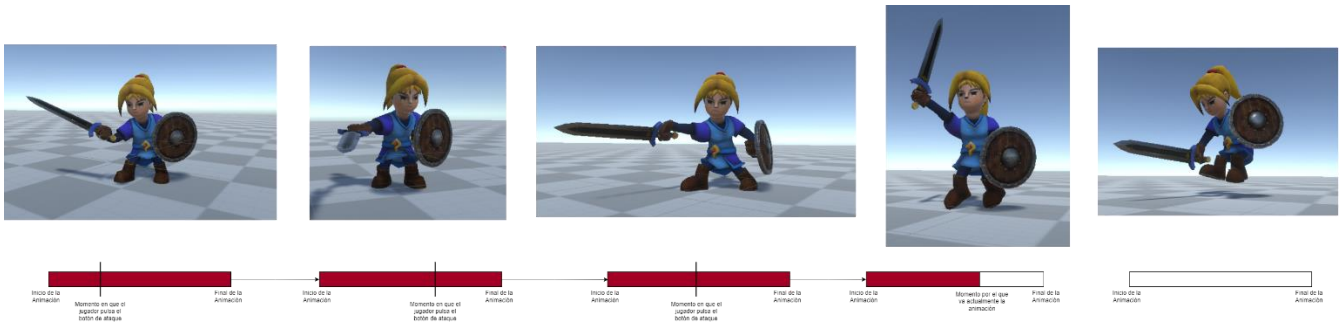


Ilustración 98. Imagen ilustrativa del sistema de combos

La idea es sencilla pues, pero tenemos un problema: ¿cómo hacemos para que se nos informe de que una animación ha acabado? *Unity* nos proporciona un sistema

para ello que consiste en establecer **eventos** en las propias animaciones tal y como se muestra en la imagen:

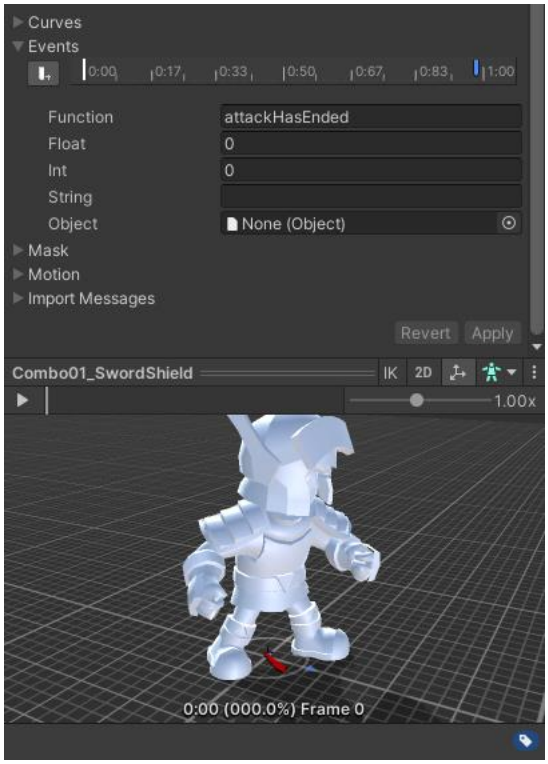


Ilustración 99. Imagen del evento attackHasEnded de la animación del primer ataque del combo

Como podemos ver, el propio inspector de *Unity* al pulsar sobre una animación nos muestra una serie de opciones, dentro de las cuales tenemos los eventos. El caso concreto de la imagen es la animación del primer ataque del combo, el cual tiene asociado un evento llamado *attackHasEnded*. Este el nombre de una función que debe de llamar un *script* que esté en el mismo *GameObject* que el *Animator* que invoca este evento. Por tanto, lo que hacemos es crear una función que llame a su vez al *SwordAttack::attackHasEnded* con su mismo nombre en *Player*.

Una vez tenemos esto claro y hemos creado los eventos (únicamente el evento *attackHasEnded*, en realidad) en las animaciones correspondientes, toca pensar en una forma para mantener la memoria del siguiente ataque porque, como hemos visto, queremos que el personaje ataque inmediatamente después con el siguiente ataque del combo en el caso de que haya pulsado el botón de ataque durante el ataque actual. Para ello, vamos a usar una serie de *flags*:

```
/// <summary>
/// This flag is for knowing if there is an attack executing right now. We want it because we can
/// have the nextAttack flag activated, which means that the player has pressed the attack button
/// while in the attack animation, and that means that another attack is in memory and will execute
/// right after the actual one.
/// </summary>
private bool actualAttack = false;
/// <summary>
/// Indicates if there is another attack in memory that will execute right after the actual one.
/// </summary>
private bool nextAttack = false;
```

Ilustración 100. Flags que se usan en SwordAttack para crear la mecánica del combo

Como se indican en los comentarios, *actualAttack* estará activo siempre que se esté ejecutando un ataque, mientras que *nextAttack* se activará únicamente cuando el jugador pulse el botón de ataque durante otro ataque. Vamos a ilustrarlo mejor gráficamente:

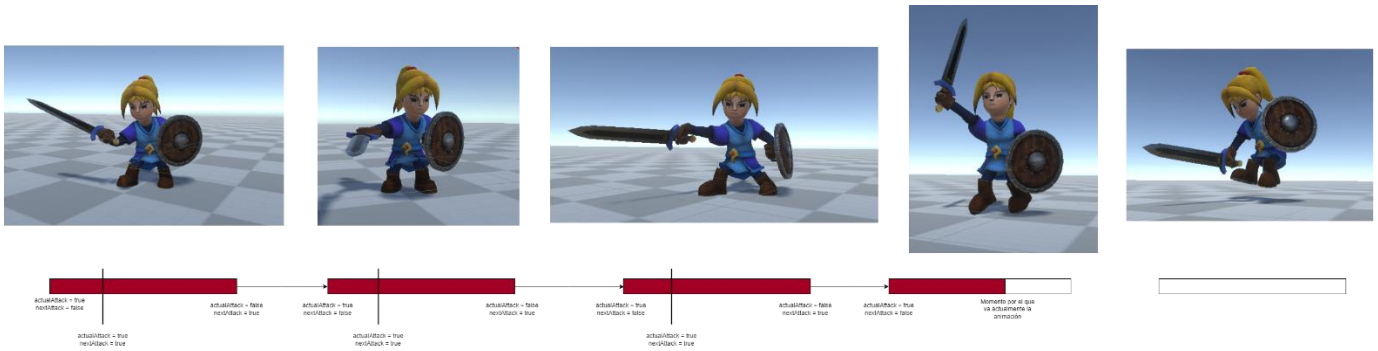


Ilustración 101. Mismo ejemplo de antes, pero ahora se indican los cambios de flags

En código, tenemos lo siguiente:

```
/// <summary>
/// Function that is called by the Input Handler. It initiates the attack activating the flag that enables
/// the attack function to execute and if the player is already executing an attack, it will put an attack in
/// the memory, that way the player will start an attack right after another, with no interruptions.
/// </summary>
4 referencias
public override void setAttack()
{
    if (!actualAttack)
    {
        actualAttack = true;
    }
    else if (!nextAttack)
    {
        nextAttack = true;
    }
}

/// <summary>
/// It is a callback function that is called as an event by the attack animations, and basically
/// checks if there is an attack in the memory, and if there isn´t then it calls stopAttack.
/// </summary>
4 referencias
public override void attackHasEnded()
{
    firstActualAttack = false;

    if (nextAttack)
    {
        actualAttack = true;
        nextAttack = false;
    }
    else
    {
        stopAttack();

        actualAttack = false;
        nextAttack = false;
    }
}
```

Ilustración 102. Algunos métodos de la clase SwordAttack

Como mencionan los comentarios, *setAttack* es llamado como *callback* por el *InputHandler* y pone *actualAttack* y *nextAttack* a *true* dependiendo de si el personaje estaba o no estaba atacando previamente. *attackHasEnded* se encarga de cambiar los *flags* acorde a lo que hemos visto al final de una animación de ataque.

Nos queda una cuestión, y es que no solo queremos que el personaje ejecute una animación, sino que queremos que haga daño a los enemigos. Para eso, vamos a hacer uso de ***Colliders***. Lo que haremos será añadirle un *Collider* a la espada del personaje, de forma que la idea es que, si entra en contacto con el *Collider* de un enemigo, este se haga daño.

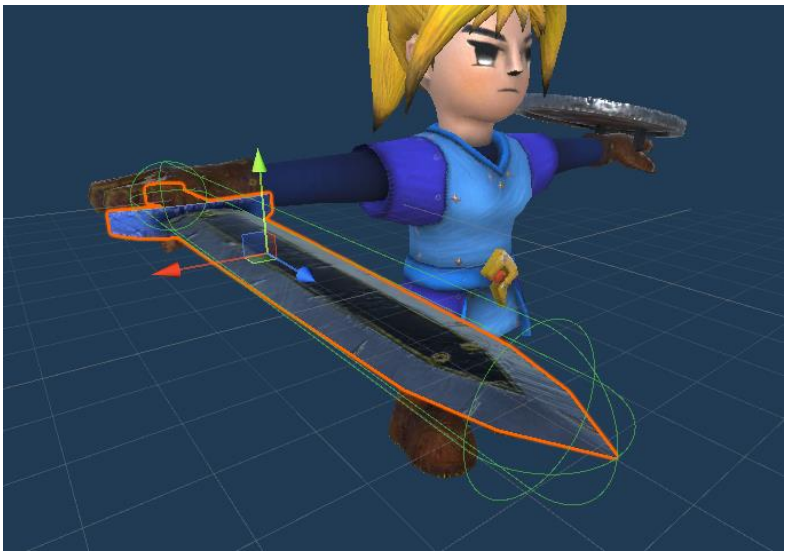


Ilustración 103. Collider de la espada del personaje

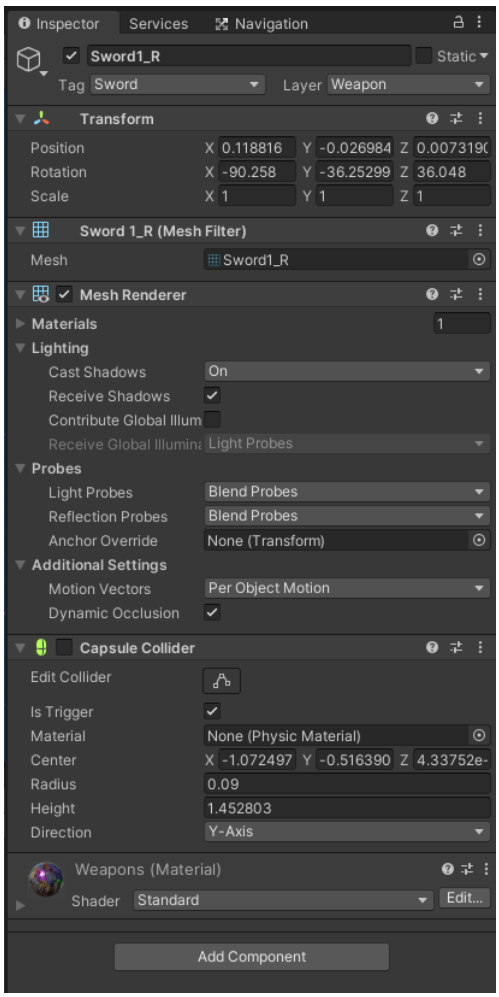


Ilustración 104. Inspector de la espada del personaje

Como podemos ver en el inspector, este *Collider* permanece desactivado hasta que el personaje ataca, momento en el que se activa. Además, tiene asignada la *tag Sword* y la *layer Weapon* (las *tags* son etiquetas que se les puede añadir a los *GameObjects* para especificar que son de un cierto tipo y se pueden aprovechar de muchas maneras, mientras que las *layers* sirven para separar objetos y hacer que, por ejemplo, un *raycast* solo afecte a los objetos de determinada *layer*). La segunda es usada precisamente para comprobar la colisión con el enemigo (el enemigo únicamente recibirá daño cuando colisione con un objeto perteneciente a esta *layer*) mientras que la primera es fundamentalmente para el cálculo del daño, ya que espada, arco y bomba tendrán daños diferentes.

Ahora bien, hay un incoveniente: si dejamos el *Collider* activado durante todo el ataque el enemigo recibirá daño todo el rato. Para solucionar esto vamos a usar otra *flag*: *firstActualAttack*. Esta *flag* nos indica si es el primer *frame* en el que el personaje está atacando, de forma que solo se active el *Collider* una única vez durante el ataque. En cuanto el ataque colisione con un enemigo, el *Collider* se desactivará a través del *onTriggerEnter* (una función que nos permite detectar colisiones entre el *collider* del objeto del script y otro cualquiera) del *script Enemy* del propio enemigo.

Por último, cabría echar un vistazo al *Animator* para ver cuál es la secuencia de transiciones entre ataques. El *Animator* es un *asset* de *Unity* que nos proporciona medios para establecer la lógica que determina las transiciones entre animaciones por medio de un grafo dirigido cuyos nodos son las animaciones y los ejes indican transiciones entre estados, por lo que, en general, estamos hablando de una **máquina de estados**. En el caso concreto del ataque con espada tenemos este subgrafo:

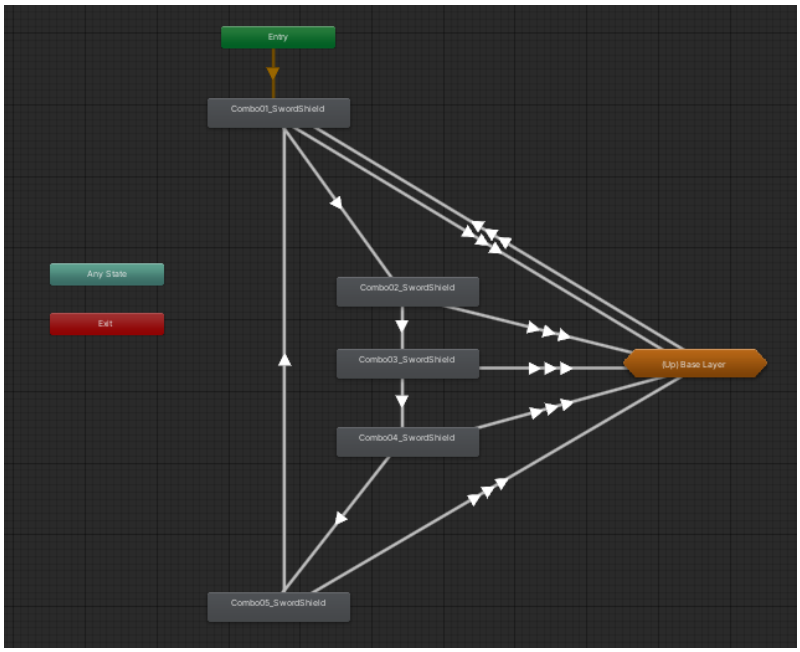


Ilustración 105. Grafo de transiciones asociado a los ataques con espada

Sin entrar en detalle de cuáles exactamente son las condiciones que producen las transiciones entre estados, la idea es que un ataque pase al siguiente del combo si el personaje está atacando, esto es, si *isAttacking* está a *true*, y transicione a otro estado si se cumple que la animación ha terminado por completo. Esto último quiere decir que el ataque no puede ser interrumpido por ninguna otra acción hasta que acabe (salvo rodar, que es la acción que más prioridad tiene y puede interrumpir el ataque). Esto se logra con un *Exit Time*, una propiedad de las transiciones que permite que se lance dicha transición si se cumple que ha pasado un determinado tiempo.

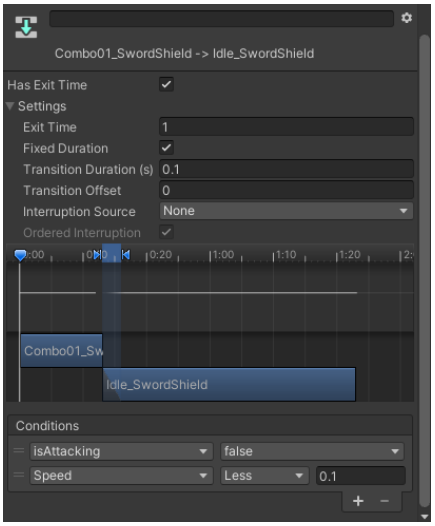


Ilustración 106. Inspector de la transición del Ataque 1 a la animación de Idle

Como se puede ver en la imagen, *Exit Time* está a 1, lo cual quiere decir que se producirá si se ha reproducido toda la animación y si se cumplen las condiciones de abajo.

3.2.4 Mecánica de bloqueo

Para la mecánica de bloqueo creamos, como con el resto de acciones, una clase ***Block*** heredera de *Command*. Este caso es más simple que el del ataque con la espada: sencillamente, si el jugador desea bloquear, el personaje levanta el escudo por medio de una animación, el *Collider* del escudo se activa y si un arma enemiga impacta contra él, el *Collider* de dicha arma se desactiva a su vez.

```
/// <summary>
/// This method simply activates the collider of the shield whenever is needed.
/// </summary>
13 referencias
public override void execute()
{
    if (shield.tag == "Shield")
    {
        if (isBlocking() && !isBeaten())
        {
            shield.GetComponent<Collider>().enabled = true;
        }
        else
        {
            shield.GetComponent<Collider>().enabled = false;
        }
    }
}

/// <summary>
/// Callback called by the Input Handler whenever the player is blocking.
/// </summary>
1 referencia
public void setBlock(bool block)
{
    this.animator.SetBool(Constants.IS_BLOCKING_CONDITION_BOOL, block);
}
```

Ilustración 107. Algunos métodos de Block

Como podemos ver, el *callback* pone la condición de *isBlocking* a *true*, de manera que el método *execute* sencillamente activa/desactiva el *Collider*. No obstante, queda otra cuestión: ¿dónde detectamos la colisión entre el *Collider* del escudo y el del arma enemiga? Para ello, creamos un *script* llamado ***Shield***. Se trata de un *script* muy sencillo que únicamente contiene un método *onTriggerEnter* para comprobar colisiones. La idea es como con la espada del personaje: si colisiona, entonces se desactiva el arma enemiga para que no colisione con el *Collider* del personaje.

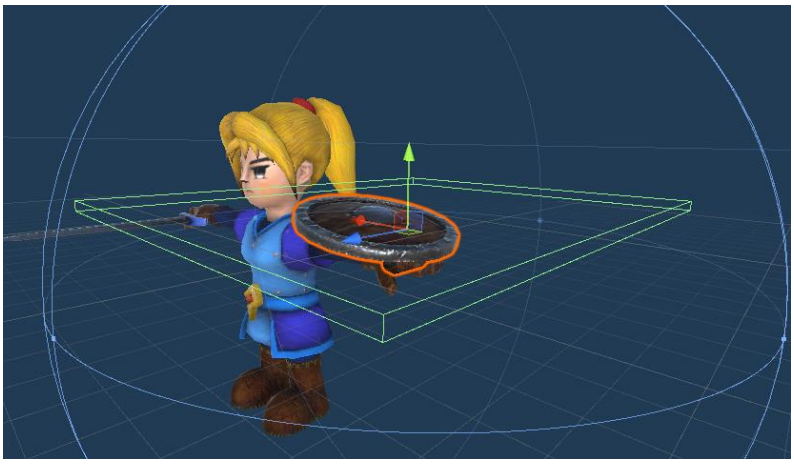


Ilustración 108. Collider del escudo

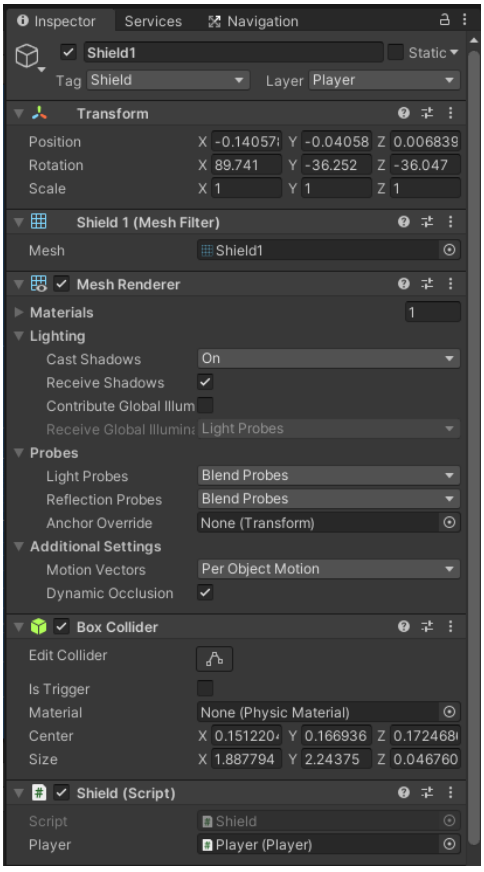


Ilustración 109. Inspector del escudo

Como podemos ver, se usa un mecanismo por medio de *tags* similar al de las armas. Cabe destacar también que el jugador dispone de una animación para cuando bloquea el ataque de un enemigo y que no se podrá mover mientras defiende, pero sí podrá apuntar el personaje hacia donde quiera (lo cual se hace estableciendo la velocidad del personaje a 0, de manera que puede girarse pero no avanzar) y,

además, el bloqueo puede ser interrumpido tanto por un ataque de la espada como por un esquive, tal y como se detalla en los diagramas de actividad.

3.2.5 Mecánica de rodar

Para empezar, como de costumbre, se creó la clase ***Roll*** heredera de *Command*. La idea y el código son muy similares a los de la mecánica de andar del personaje: se calcula el ángulo objetivo por medio de la tangente y se suaviza la rotación del personaje por medio de *smoothDampAngle*. La diferencia radica en que el personaje rueda más rápido que anda y en que aquí no tenemos un *input* que se mantiene presionado en el tiempo como un *joystick*, sino un botón, de forma que la animación de andar termina cuando el jugador así lo desea mientras que el personaje se mantiene moviendo hasta que termina de rodar. El código actual de *Roll* es más complejo pero esas añadiduras se discutirán más tarde pues tienen que ver con la mecánica de fijar y la de correr.

```
float targetAngle = Mathf.Atan2(rollingDirection.x, rollingDirection.z) * Mathf.Rad2Deg +
    Camera.main.transform.eulerAngles.y;
float angle = Mathf.SmoothDampAngle(player.transform.eulerAngles.y, targetAngle,
    ref turnSmoothVelocity, player.getTurnSmoothTime());
player.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

moveDirection = Quaternion.Euler(0.0f, targetAngle, 0.0f) * Vector3.forward;
```

Ilustración 110. Fragmento de código del execute de Roll

Caben destacar dos cosas: la primera, que mientras que el personaje esté rodando su *Collider* se mantendrá desactivado, es decir, es **invulnerable** durante el tiempo que pase rodando; la segunda, que rodar es la acción del personaje con más prioridad, es decir, ninguna acción puede interrumpir al personaje rodando, ni siquiera ella misma, por lo que el jugador permanece rodando incluso cuando quiere rodar otra vez.

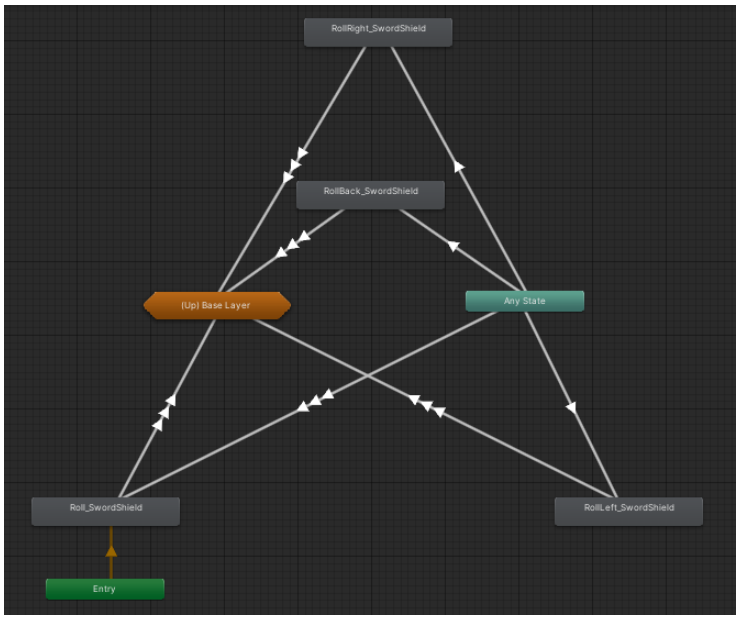


Ilustración 111. Subgrafo de rodar

Como se puede apreciar, existen varios tipos de esquive, pero de ellos hablaremos más adelante. Por ahora, nos centramos en el estado *Roll_SwordShield* que es el rodar hacia delante. Fundamentalmente, a lo que queremos llamar la atención aquí es al ***Any State***. Ese es un estado que representa cualquier estado; es decir, si tenemos una transición entre él y *Roll_SwordShield* significa que cualquier estado puede transicionar a *Roll_SwordShield* si se cumplen las condiciones necesarias. Con esto conseguimos que cualquier acción del personaje pueda ser interrumpida por el rodar.

Tal cual aparece en el grafo se podría pensar que otro rodar puede interrumpirlo, pero no es así debido a que en el *callback* de rodar se comprueba antes si ya se está rodando; si es así, no se hace nada.

Cabe destacar también que volvemos a usar los eventos de las animaciones en este caso para indicar que la animación de rodar ya ha acabado con el evento *Player::stopRoll* que sencillamente llama a *Roll::stopRoll*, lo cual es importante para actualizar las *flags* y volver a activar el *Collider* del jugador.

3.2.6 Enemigos

Lo primero de todo fue crear el *script* ***Enemy***. Es similar a *Player* en el sentido de que ambos almacenan y gestionan datos tales como la vida, el daño, etc… además, se encarga de procesar las colisiones de ataques que reciben los enemigos y de ejecutar determinados comportamientos del enemigo, como que al morir deje *loot* o

aparezcan ciertos enemigos a modo de hijos, la propia muerte del enemigo, computar el daño que sufre tras un golpe del jugador, etc… Es, en definitiva, la abstracción del propio enemigo.

Para el comportamiento de cada monstruo tenemos la clase ***Monster*** y sus subclases. Puesto que tenemos diversos enemigos que se comportarán a su vez de formas diversas, la clase *Monster* es abstracta y fundamentalmente provee métodos abstractos para implementar el comportamiento de cada tipo de enemigo en particular; principalmente, esto se hace por medio de *monsterBehaviour* y *attackCoroutine*.

```
/// <summary>
/// Function that is called every frame and determines some actions
/// of the monster, depending on the concrete type.
/// </summary>
7 referencias
public abstract void monsterBehaviour();
/// <summary>
/// Attack coroutine that is called every awaitTime seconds and determines
/// how the monster attacks.
/// </summary>
/// <returns></returns>
7 referencias
public abstract IEnumerator attackCoroutine();
```

Ilustración 112. Métodos abstractos de Monster

La corrutina usada se justifica por el hecho de que el enemigo atacará cada cierto tiempo y no continuamente, mientras que el otro método dependerá por entero del tipo de monstruo que sea y es llamado por *Update* de *Enemy*. Aparte, esta clase también proporciona métodos que permiten **elegir el ataque** cada vez que el monstruo ataca, que paran el ataque y que permiten detectar al jugador. Este último merece la pena explicarlo.

```
/// <summary>
/// This method tells us if the enemy has detected the player, via a raycast. In the
/// shake of performance, the raycast is only invoked whenever the player is within
/// the detection radius. The raycast avoids the enemy and weapon enemy layers with
/// a layer mask.
/// </summary>
/// <returns></returns>
13 referencias
public bool playerDetected()
{
    if (Vector3.Distance(monsterPos.position, playerPos.position) <= detectionRadius)
    {
        string[] layers = { Constants.ENEMY_LAYER, Constants.ENEMY_WEAPON_LAYER };
        int layerMask = LayerMask.GetMask(layers);
        layerMask = ~layerMask;

        RaycastHit hit;
        if (Physics.Raycast(monsterPos.position, ((playerPos.position - monsterPos.position) + Vector3.up).normalized, out hit,
            detectionRadius, layerMask))
        {
            if (hit.collider.CompareTag(Constants.PLAYER_NAME))
            {
                return true;
            }
        }
    }

    return false;
}
```

Ilustración 113. Método playerDetected de Monster

Detectar al jugador es esencial para todos los enemigos por razones obvias, así que este método es común a todos los tipos de monstruo. Se basa en realizar un *raycast* que, por medio de una máscara, evita a la capa de enemigo y de arma enemiga. Esto se hace porque es posible que el rayo colisionara con el propio *Collider* del enemigo o con su arma o algo por el estilo, con lo cual no se realizaría correctamente. Por el contrario, no se evitan el resto de capas porque queremos que los enemigos no sean capaces de detectar al jugador si hay una pared en medio u otro objeto. Además, no hacemos el *raycast* siempre porque es una tarea algo costosa, sino que comprobamos primero si el jugador se encuentra en el radio de detección del enemigo.

Por otra parte, (casi) todos los enemigos funcionan con el siguiente algoritmo básico de **inteligencia artificial**:

1. El jugador entra en el **radio de detección** del enemigo.
2. El enemigo comprueba por medio de la función antes descrita si **ha detectado** al jugador.
3. Si es así, comienza a seguirlo por medio de las herramientas de *pathfinding* que nos proporciona *Unity*, a saber, el ***NavMeshAgent*** asociado a cada enemigo que se encuentra colocado sobre una ***NavMesh*** calculada previamente.

4. El enemigo sigue al jugador hasta que este está dentro del **radio de ataque** del enemigo, momento en que la corrutina de ataque es lanzada y el enemigo comienza a atacar.
5. Si el jugador sale del radio de ataque, el enemigo vuelve a seguirlo hasta que esté dentro del radio de ataque, y así.

Cada enemigo tiene a su vez un *Animator* asociado el cual suele tener una estructura casi idéntica entre enemigos:

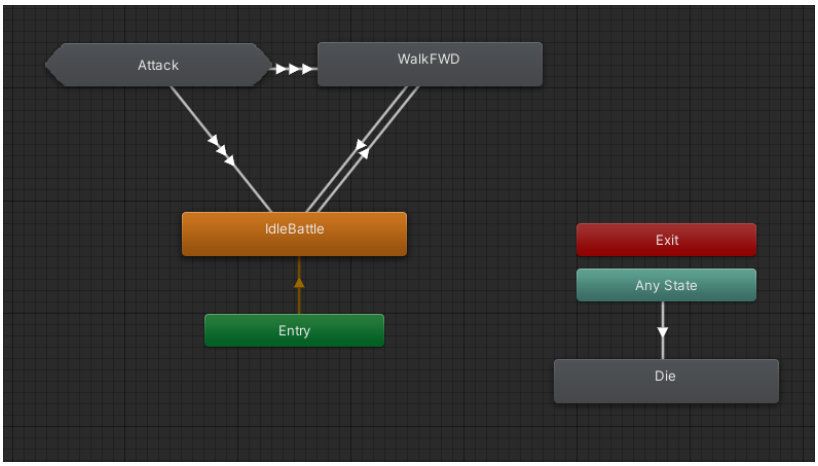


Ilustración 114. Animator de un enemigo cualquiera

Como se puede ver, simplemente va transicionando entre los estados de *Idle*, *Attack*, *Walk* y, finalmente, el estado de *Die* al morir, que representa la animación de muerte. *Attack* no es un estado, sino que es un subgrafo con los diferentes ataques. Lo bueno es que estos grafos suelen ser bastantes más simples que el del jugador ya que no hay tantas dependencias al tener muchas menos acciones.

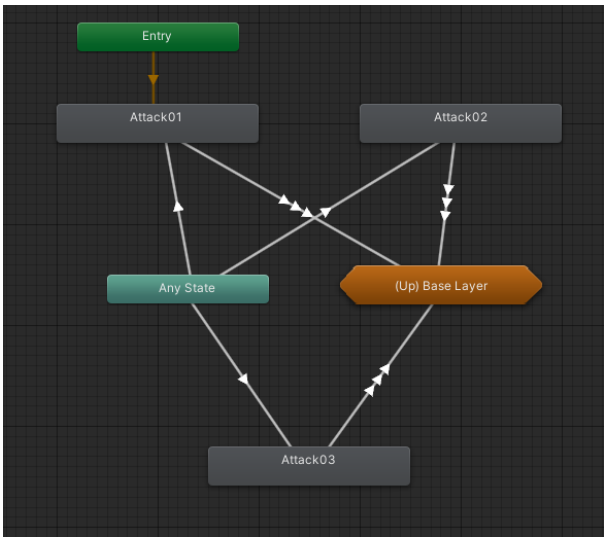


Ilustración 115. Subgrafo de ataque de un enemigo cualquiera

```
7 referencias
public override void monsterBehaviour()
{
    if (!isAttacking())
    {
        if (playerDetected())
        {
            if (Vector3.Distance(monsterPos.position, playerPos.position) > attackRadius)
            {
                agent.speed = speed;
                agent.SetDestination(playerPos.position);
                thisAnim.SetFloat(Constants.SPEED_CONDITION_FLOAT, speed);
            }
            else
            {
                agent.speed = 0.0f;
                thisAnim.SetFloat(Constants.SPEED_CONDITION_FLOAT, 0.0f);

                Vector3 forward = (playerPos.position - monsterPos.position).normalized;
                float targetAngle = Mathf.Atan2(forward.x, forward.z) * Mathf.Rad2Deg;
                float angle = Mathf.SmoothDampAngle(monsterPos.transform.eulerAngles.y, targetAngle,
                    ref turnSmoothVelocity, turnSmoothTime);
                monsterPos.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);
            }
        }
    }
}
```

Ilustración 116. Método monsterBehaviour de NormalMonster

```
7 referencias
public override IEnumerator attackCoroutine()
{
    while (true)
    {
        float time = Random.Range(0.1f, awaitTime);

        if (Vector3.Distance(monsterPos.position, playerPos.position) <= attackRadius && !isAttacking())
        {
            if (playerDetected())
            {
                Vector3 forward = (playerPos.position - monsterPos.position).normalized;
                float targetAngle = Mathf.Atan2(forward.x, forward.z) * Mathf.Rad2Deg;
                float angle = Mathf.SmoothDampAngle(monsterPos.transform.eulerAngles.y, targetAngle,
                    ref turnSmoothVelocity, turnSmoothTime);
                monsterPos.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

                yield return new WaitForSeconds(time);

                chooseAttack();
            }
        }

        yield return new WaitForSeconds(time);
    }
}
```

Ilustración 117. Corrutina de ataque de NormalMonster

En las imágenes de arriba se puede ver este proceso, donde la velocidad del enemigo se vuelve 0 cuando el jugador está dentro del radio de ataque y recupera la velocidad cuando no lo está. Además, se usa el mismo código de la tangente y el *SmoothDampAngle* para suavizar los giros del enemigo y hacer que este esté siempre mirando hacia el jugador tanto para cuando lo está siguiendo como cuando está atacando. Nótese también que el tiempo de espera entre ataque y ataque es aleatorio y que *Monster::chooseAttack* es un método que elige aleatoriamente un ataque de entre los que tiene el monstruo. Por lo demás, el fundamento de los ataques es idéntico al del arma del jugador: cada arma enemiga tiene uno o más *Colliders* asociados que se encuentran desactivados hasta que el enemigo ataca, momento en el que se activan, y se comprueba la colisión con el jugador por medio de un *OnTriggerEnter*.

Tendremos, como se han mencionado antes, múltiples subclases de *Monster*; una de ellas, ***NormalMonster***, representa el tipo de monstruo más común, que es el que funciona de esta manera; no obstante, hay otros tipos de monstruos que funcionan de forma diferente. Vamos a repasar ahora todos los diferentes tipos de enemigos que nos podemos encontrar, cuyos modelos y animaciones provienen de los paquetes *RPG Monster Wave PBR* y el *RPG Monster Wave 2 PBR* de la tienda de *assets* de *Unity*.

- **Ferrus.** Es un enemigo común, sin ningún tipo de ataque especial, pero con una dificultad mayor de la habitual.

Ilustración 118. Imagen del Ferrus

Posee 3 tipos de ataques que realiza con las cuchillas de sus manos. Por lo demás, funciona básicamente de la forma que se ha explicado más arriba sin mucho más misterio.

- **Ghini.** Nuevamente, es un enemigo común que funciona con un *NormalMonster*. Posee 2 ataques diferentes con la espada.

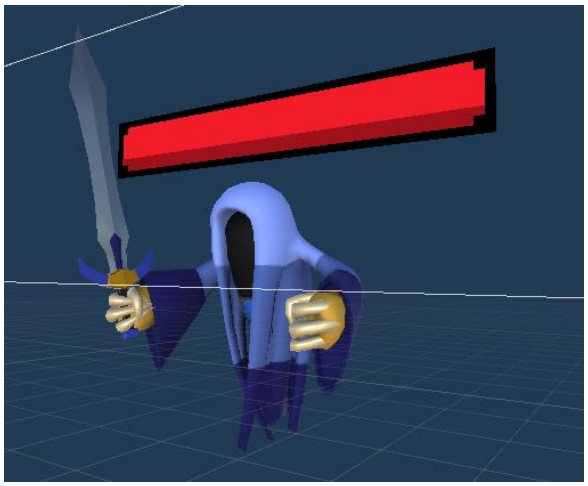


Ilustración 119. Imagen del Ghini

- **Goriya.** Otro enemigo común que puede realizar 2 ataques diferentes con la daga.

Ilustración 120. Imagen del Goriya

- **Keese.** Un enemigo normal más con un solo ataque de mordisco.

Ilustración 121. Imagen del Keese

- **Lanmola.** Enemigo común que posee 3 ataques diferentes con su pinza.

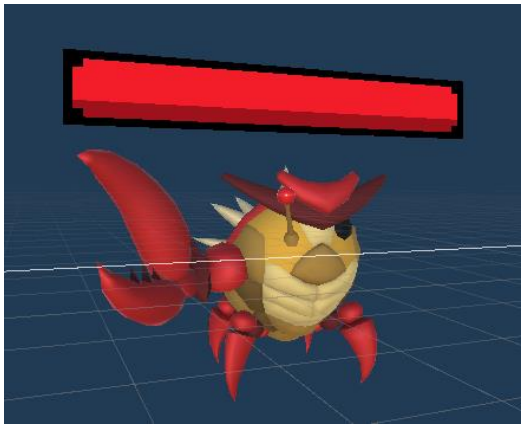


Ilustración 122. Imagen del Lanmola

- **Leever/Manhandla.** Otro enemigo más con la misma funcionalidad que los anteriores que solo tiene un único ataque que consiste en un mordisco. El *Manhadla* es el mismo enemigo, pero más grande y fuerte, pudiendo ser un *boss.*

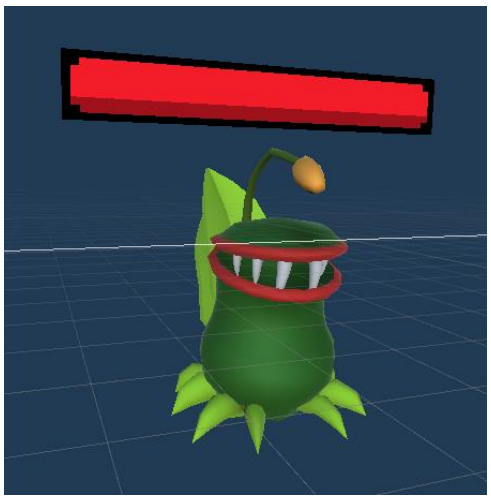


Ilustración 123. Imagen del Leever/Manhandla

- **Moblin.** De nuevo, es un enemigo con la funcionalidad básica que posee dos ataques diferentes, un golpe con la maza y un ataque giratorio, aunque es algo más duro de lo habitual y es usado como *mini boss*.

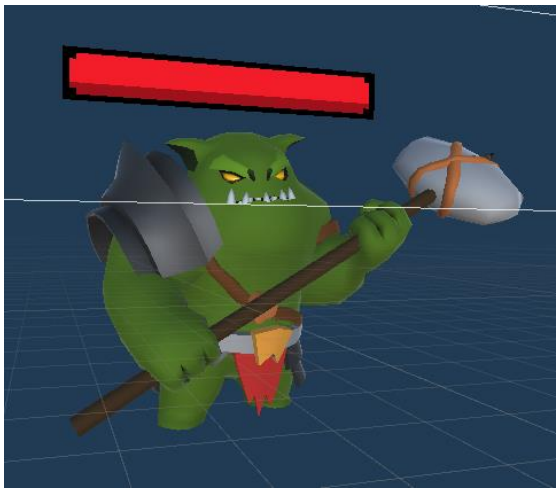


Ilustración 124. Imagen del Moblin

Hay un detalle con este enemigo: el *Collider* del arma no debería ser desactivado cuando realiza el ataque giratorio e impacta con el jugador, porque queremos que impacte cuantas veces haga falta mientras esté girando. Esta excepción se especifica en el código de *Player::OnTriggerEnter*.

```
// This if is for an special case where the moblin uses a circular attack,
// in which case its weapon collider isn´t disabled.
if(other.GetComponent<Enemy>() != null)
{
    if (!other.gameObject.CompareTag(Constants.MOBLIN_NAME) &&
    other.GetComponent<Enemy>().getLastAttack() != Constants.ATTACK02_TRIGGER)
    {
        other.enabled = false;
    }
}
else
{
    other.enabled = false;
}
```

Ilustración 125. Parte del código del Player::OnTriggerEnter

- **Vire.** Un enemigo común algo más duro de lo habitual que tiene solo un ataque de mordisco.

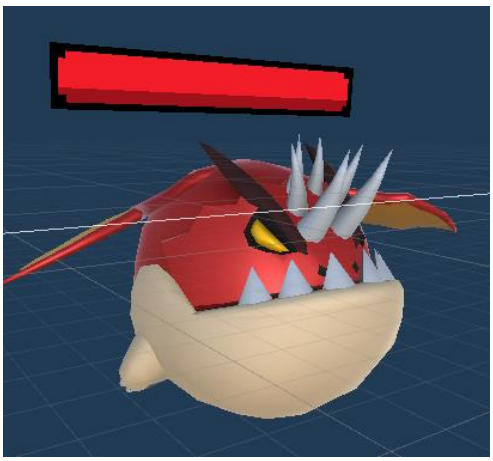


Ilustración 126. Imagen del Vire

De nuevo, este enemigo tiene un detalle: al morir, en lugar de dejar *loot*, hace aparecer 3 *keeses*. Esto se hace por medio del método *Enemy::instatiateChildMonster*, el cual se llama al morir el monstruo. El método hace aparecer los monstruos por medio de *Resources.Load*, los coloca de una cierta forma y los inicializa.

```
/// <summary>
/// Some monsters spawn other monsters the moment they die, and
/// this method does that.
/// </summary>
/// <param name="childMonsterName"></param>
2 referencias
private void instatiateChildMonster(string childMonsterName)
{
    GameObject childMonster1 = Instantiate((GameObject)Resources.Load("Prefabs/Enemies/" + childMonsterName, typeof(GameObject)));
    GameObject childMonster2 = Instantiate((GameObject)Resources.Load("Prefabs/Enemies/" + childMonsterName, typeof(GameObject)));
    GameObject childMonster3 = Instantiate((GameObject)Resources.Load("Prefabs/Enemies/" + childMonsterName, typeof(GameObject)));

    childMonster1.transform.position = transform.position + new Vector3(0.0f, 0.0f, 2.0f);
    childMonster2.transform.position = transform.position + new Vector3(1.5f, 0.0f, -0.5f);
    childMonster3.transform.position = transform.position + new Vector3(-1.5f, 0.0f, -0.5f);

    childMonster1.GetComponent<Enemy>().setPlayer(player);
    childMonster2.GetComponent<Enemy>().setPlayer(player);
    childMonster3.GetComponent<Enemy>().setPlayer(player);

    childMonster1.GetComponent<Enemy>().setPlayerPosition(playerPosition);
    childMonster2.GetComponent<Enemy>().setPlayerPosition(playerPosition);
    childMonster3.GetComponent<Enemy>().setPlayerPosition(playerPosition);
}
```

Ilustración 127. Código de Enemy::instatiateChildMonster

- **Aquamentus.** Se trata de uno de los posibles jefes del juego; por ello, es un enemigo particularmente duro.

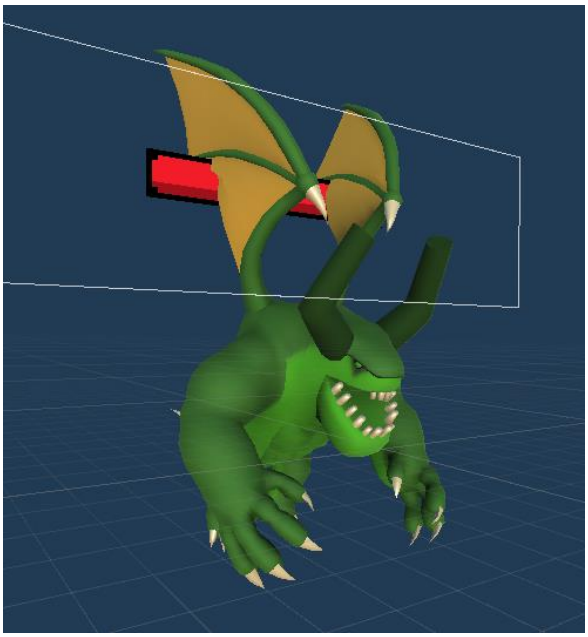


Ilustración 128. Imagen del Aquamentus

Posee 4 posibles ataques: un mordisco, un ataque con la garra, una llamarada y una bola de fuego. Para este enemigo se usa la clase ***NormalBallMonster***. Esta clase engloba enemigos que persiguen al jugador con normalidad y usan ataques a meleé, pero que además tienen algún ataque que consiste en escupir algún tipo de **bola de energía** o similar.

```
/// <summary>
/// The normal ball monsters throw a ball if the player is outside the attack radius
/// but inside the detection radius.
/// </summary>
/// <returns></returns>
7 referencias
public override IEnumerator attackCoroutine()
{
    while (true)
    {
        float time = Random.Range(0.1f, awaitTime);

        if (playerDetected())
        {
            if (Vector3.Distance(monsterPos.position, playerPos.position) <= attackRadius && !isAttacking())
            {
                Vector3 forward = (playerPos.position - monsterPos.position).normalized;
                float targetAngle = Mathf.Atan2(forward.x, forward.z) * Mathf.Rad2Deg;
                float angle = Mathf.SmoothDampAngle(monsterPos.transform.eulerAngles.y, targetAngle,
                    ref turnSmoothVelocity, turnSmoothTime);
                monsterPos.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

                yield return new WaitForSeconds(time);

                chooseAttack();
            }
            else if (!isAttacking())
            {
                yield return new WaitForSeconds(time);

                float chance = Random.Range(0.0f, 1.0f);

                if (chance <= energyBallChance)
                {
                    thisAnim.SetBool(Constants.IS_ATTACKING_CONDITION_BOOL, true);
                    thisAnim.SetTrigger(Constants.ATTACK03_TRIGGER);
                    lastAttack = Constants.ATTACK03_TRIGGER;
                }

            }
        }

        yield return new WaitForSeconds(time);
    }
}
```

Ilustración 129. Método attackCoroutine de NormalBallMonster

Como se puede ver en la imagen, la diferencia con respecto a *NormalMonster::attackCoroutine* es que, si el jugador está fuera del radio de ataque, hay una probabilidad de que el enemigo suelte una bola de energía.

Aparte, también tenemos un *script* llamado ***EnergyBall***. Esta clase se encarga del funcionamiento de la bola, de manera que se encarga de moverla en la dirección del jugador y de gestionar el daño que hace y su sistema de partículas.

```
// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    lifetime += Time.deltaTime;

    if (lifetime >= maxLifetime)
        Destroy(gameObject);

    if (isFinished)
    {
        if(ballHit == null)
        {
            Destroy(gameObject);
        }
    }
    else
    {
        transform.position += direction * speed * Time.deltaTime;
    }
}
```

Ilustración 130. Método Update de EnergyBall

Como se puede ver en la imagen, tiene un tiempo de vida y, cuando este llega al máximo, se destruye; por lo demás, avanza en la dirección del jugador o, mejor dicho, en la dirección que tenía hacia el jugador cuando fue lanzada, pues no queremos que lo persiga exactamente, solo lanzarla hacia donde estaba antes el jugador. La bola no se destruye inmediatamente al impactar con el jugador; cuando lo hace, empieza a reproducirse un sistema de partículas que representa el impacto de la bola y cuando ese sistema para de emitir partículas es cuando se destruye el objeto. Este es un mecanismo que se usa en diversos ataques enemigos.

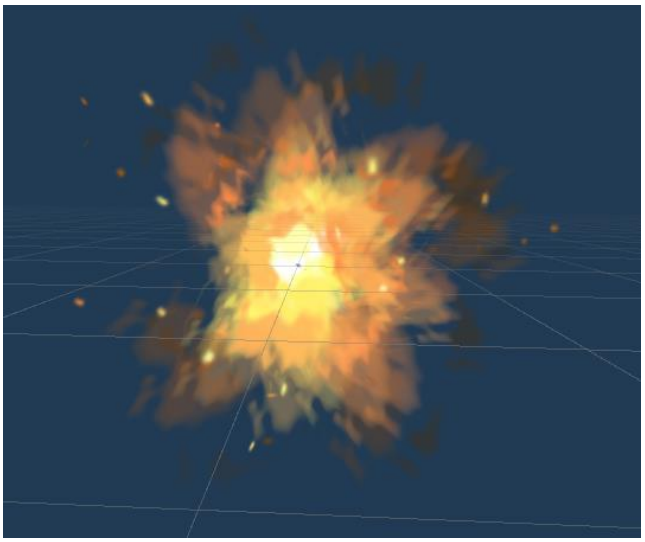


Ilustración 131. Asset de la bola de fuego del Aquamentus

Todavía no hemos acabado con el *Aquamentus*, ya que necesitamos de un cierto *script*, ***AquamentusBehaviour***, para que contenga algunos eventos llamados por sus animaciones. En concreto, la animación tanto de la bola de fuego como del lanzallamas es la misma, con la salvedad de que la animación se ralentiza para el lanzallamas ya que queremos que el *Aquamentus* lo mantenga durante un cierto tiempo.

```
1 referencia
public void startFireBall()
{
    ballCopy = Instantiate(fireBall);
    ballCopy.transform.position = fireBall.transform.position;
    ballCopy.GetComponent<ParticleSystem>().Play();
    ballCopy.GetComponent<SphereCollider>().enabled = true;
    ballCopy.GetComponent<EnergyBall>().enabled = true;
}

1 referencia
public void startFireBreath()
{
    fireBreath.GetComponent<ParticleSystem>().Play();
    fireBreath.GetComponent<CapsuleCollider>().enabled = true;
    GetComponent<Animator>().speed = breathAnimatorSpeed;
}
```

Ilustración 132. Eventos para la bola de fuego y el lanzallamas de AquamentusBehaviour

Cuando la corrutina de ataque de *NormalBallMonster* hace que el *Aquamentus* realice el ataque correspondiente tanto a la bola como al lanzallamas, que tienen la misma animación, dicha animación llama al evento *AquamentusBehaviour::startFire* el cual decide aleatoriamente cuál usar, por lo que llama o a *startFireBall* o a *startFireBreath*. Nótese que *startFireBall* instancia una copia de la bola de fuego que está dentro de la jerarquía del *prefab* de *Aquamentus* y activa el *script* de *EnergyBall*, con lo cual la bola ya va directa al jugador. Por su parte, *startFireBreath* ralentiza la animación de ataque para mantenerlo un tiempo y activa tanto el sistema de partículas de la llamarada como su *Collider*.

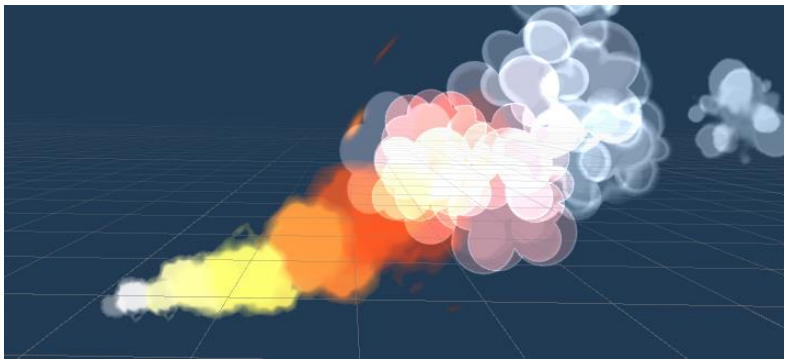


Ilustración 133. Asset de la llamarada del Aquamentus

Un último apunte: en *Player* se usa además un *OnTriggerStay* (es lo mismo que *OnTriggerEnter*, salvo por el hecho de que se llama cada *frame* que los *Colliders* permanecen colisionando en lugar de solo el primero) para estos ataques de tipo llamarada. La razón es que queremos que el jugador se haga daño durante todo el tiempo que su *Collider* permanezca en contacto con el *Collider* de la llamarada.

Los sistemas de partículas del lanzallamas y la bola de fuego provienen del paquete *Cartoon FX Remaster Free*.

- **Armos.** Otro de los *bosses* del juego. Es un *NormalMonster* que posee dos ataques diferentes: un puñetazo y un golpe fuerte contra el suelo.

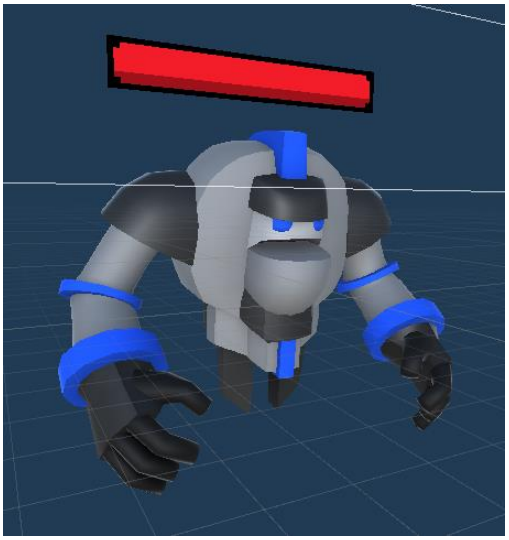


Ilustración 134. Imagen del Armos

Pese a funcionar casi enteramente como un enemigo normal, hay un detalle a tener en cuenta: posee también como el *Aquamentus* un *script* llamado ***ArmosBehaviour***. Fundamentalmente, lo que hace este *script* es capturar

el evento de la animación (es la principal motivación para todos los *scripts* tipo *Behaviour* en realidad, al no poder llamar a los eventos desde las subclases de *Monster* por no ser *MonoBehaviour*) del golpe fuerte en el suelo que dicta cuándo tiene que empezar el sistema de partículas y cuándo comienza el ataque a hacer daño, pues la idea es que el *Collider* del ataque esté activado durante una cierta ventana de tiempo.

```
// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    if (isExploding)
    {
        if (explosionCollider.enabled)
        {
            actualDamageTime += Time.deltaTime;

            if (actualDamageTime >= damageTime)
            {
                explosionCollider.enabled = false;
                actualDamageTime = 0.0f;
                isExploding = false;
            }
        }
    }

}

/// <summary>
/// Event that is called by the attack animation for starting the explosion.
/// </summary>
0 referencias
private void explode()
{
    explosionCopy = Instantiate(explosion);
    explosionCopy.transform.position = explosion.transform.position;
    explosionCopy.SetActive(true);
    explosionCopy.GetComponent<ParticleSystem>().Play();
    explosionSound.Play();
    explosionCollider.enabled = true;
    isExploding = true;
}
```

Ilustración 135. Update y el evento llamado por la animación, ambos de ArmosBehaviour

Como se puede ver, el *Update* se encarga de abrir y cerrar dicha ventana de tiempo en la que el jugador puede sufrir daño, mientras que el *explode* sencillamente activa el sistema de partículas y avisa al *Update* de abrir la ventana de tiempo por medio de una *flag*.

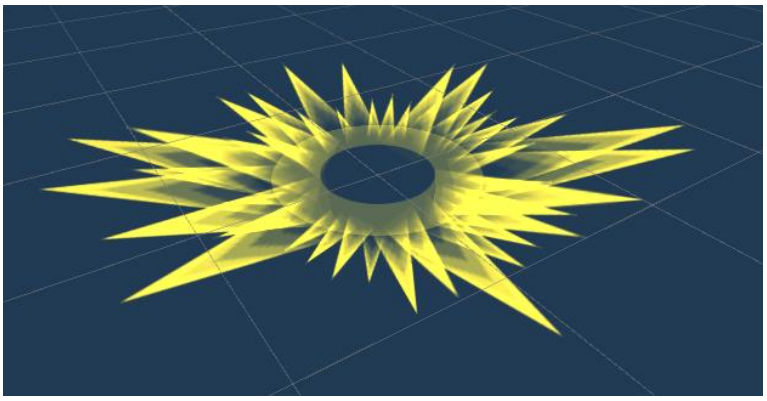


Ilustración 136. Sistema de partículas usado para el impacto del golpe del Armos

De nuevo, para este sistema de partículas se ha usado el paquete antes indicado.

- **Digdogger.** Este bicho, que es uno de los *bosses*, ya no implementa ninguna de las subclases de *Monster* antes mencionadas, sino *StaticMonster*, otra subclase que se relaciona con monstruos que ya no siguen al jugador, sino que se quedan fijos en el sitio (como es el caso de este) o que se teletransportan y no se desplazan de forma habitual.

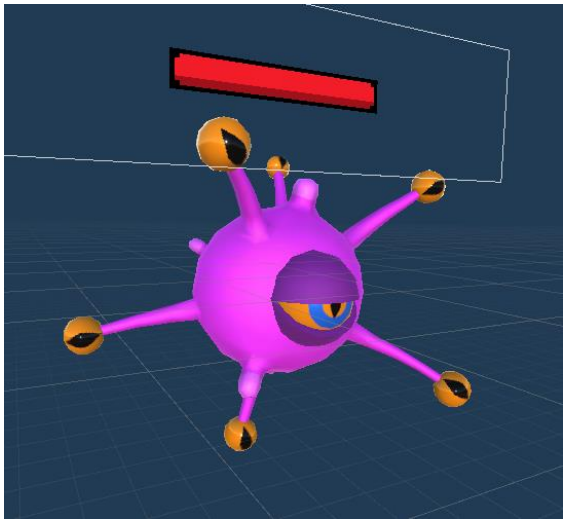


Ilustración 137. Imagen del Digdogger

```
/// <summary>
/// Every frame, the monster looks at the player and nothing else.
/// </summary>
7 referencias
public override void monsterBehaviour()
{
    agent.speed = 0.0f;

    if (!isAttacking())
    {
        if (playerDetected())
        {
            Vector3 forward = (playerPos.position - monsterPos.position).normalized;
            float targetAngle = Mathf.Atan2(forward.x, forward.z) * Mathf.Rad2Deg;
            float angle = Mathf.SmoothDampAngle(monsterPos.transform.eulerAngles.y, targetAngle,
                ref turnSmoothVelocity, turnSmoothTime);
            monsterPos.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);
        }
    }
}
```

Ilustración 138. Código de StaticMonster::monsterBehaviour

Mientras que la corrutina de ataque es idéntica a la de *NormalMonster* salvo por el hecho de que estos bichos no usan radio de ataque al ser atacantes fundamentalmente a distancia, el *monsterBehaviour*, como vemos, solo gira al monstruo en la dirección del jugador.

El *Digdogger* tiene además dos ataques bien diferenciados: una **bola de energía** y un **rayo láser**. Es una situación muy similar a la del *Aquamentus*, donde tenemos un ataque de bola y uno que está mantenido en el tiempo como es el rayo láser. Por ello, vamos a usar a su vez un *script* muy similar a *AquamentusBehaviour*, ***DiggdogerBehaviour***.

```
0 referencias
public void chargingLaser()
{
    chargingSound.Play();
    GetComponent<Animator>().speed = laserAnimationSpeed;
    laserStart.Play();
}

0 referencias
public void chargingBall()
{
    shootSound.Play();
    energyBall.GetComponent<ParticleSystem>().Play();
}

0 referencias
public void shootLaser()
{
    shootSound.Play();
    origPlayerPos = playerTransform.position;

    laserRenderer.enabled = true;
    laserCollider.enabled = true;
}

0 referencias
public void shootBall()
{
    ballCopy = Instantiate(energyBall);
    energyBall.GetComponent<ParticleSystem>().Stop();
    ballCopy.transform.position = energyBall.transform.position;
    ballCopy.GetComponent<ParticleSystem>().Play();
    ballCopy.GetComponent<SphereCollider>().enabled = true;
    ballCopy.GetComponent<EnergyBall>().enabled = true;
}
```

Ilustración 139. Algunos métodos de DigdoggerBehaviour

Como podemos ver, es algo distinto al *Aquamentus* en el sentido de que ahora tenemos eventos para cargar la bola de energía y el rayo láser, pero por lo demás es muy similar; cuando se da el tiempo correcto en la animación, se llama primero al método *charging* correspondiente y luego al método *shoot* adecuado. Aquí, a diferencia del *Aquamentus*, no hay una única animación para ambos ataques, sino que hay dos animaciones y cada una llama a unos eventos u otros.

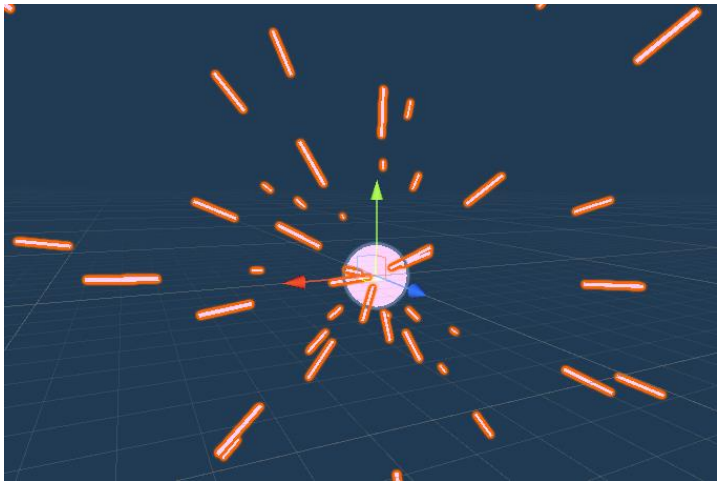


Ilustración 140. Asset de la bola de energia

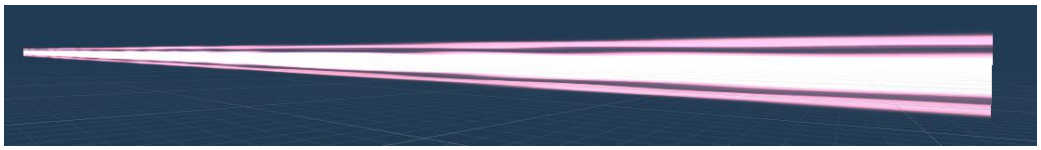


Ilustración 141. Asset del rayo láser

Para estos *assets* se han usado los paquetes *15 – Magic Laser* y el *Sherbb's Particle Collection*.

- **Dodongo.** La idea de este monstruo es muy similar al de un *NormalMonster*; no obstante, posee una diferencia: puede **bloquear** ataques.

Ilustración 142. Imagen del Dodongo

Por ello, tiene asociado una subclase de *Monster* diferente: ***NormalBlockingMonster***, que es a su vez subclase de ***BlockingMonster***. La razón de que se necesite otra clase abstracta más como *BlockingMonster* es que hay monstruos que añaden funcionalidad al comportamiento básico de seguir al jugador, aunque no es el caso de este enemigo. El *BlockingMonster::monsterBehaviour* es esencialmente el mismo que el de *NormalMonster*, salvo que si está bloqueando un ataque no se mueve. Por otra parte, cuenta con un método *BlockingMonster::block* que decide cuándo el enemigo debe bloquear.

```
/// <summary>
/// Method that decides whether the monster has to block or not.
/// </summary>
1 referencia
public void block()
{
    if (player.isAttacking())
    {
        monsterPos.LookAt(playerPos);

        if (firstAttack)
        {
            float chance = Random.Range(0.0f, 1.0f);
            if (chance <= blockChance)
            {
                agent.speed = 0.0f;
                thisAnim.SetBool(Constants.IS_BLOCKING_CONDITION_BOOL, true);
            }

            firstAttack = false;
        }

    }
    else
    {
        firstAttack = true;
        thisAnim.SetBool(Constants.IS_BLOCKING_CONDITION_BOOL, false);
    }
}
```

Ilustración 143. Método BlockingMonster::block

Como se puede ver, el enemigo bloquea si el personaje está atacando y con una cierta probabilidad, porque por supuesto puede no bloquear. Hay que tener en cuenta que este método es llamado por *monsterBehaviour* cada *frame*, de forma que necesitamos que la decisión de bloquear o no se tome en el primer *frame* que el jugador esté atacando y no que se decida cada *frame*, para eso es el *flag firstAttack*.

NormalBlockingMonster en realidad únicamente implementa el método abstracto *specializedAttackCoroutine* heredado de *BlockingMonster* y es fundamentalmente la misma corrutina de ataque de *NormalMonster*.

También conviene mencionar que los escudos de los enemigos con escudo tienen un *script* muy similar al que ya vimos de *Shield* llamado ***EnemyShield***.

- **Ganon.** Si antes decíamos que *BlockingMonster* era una clase abstracta por ciertos enemigos que no se comportan normalmente, este enemigo es el motivo.

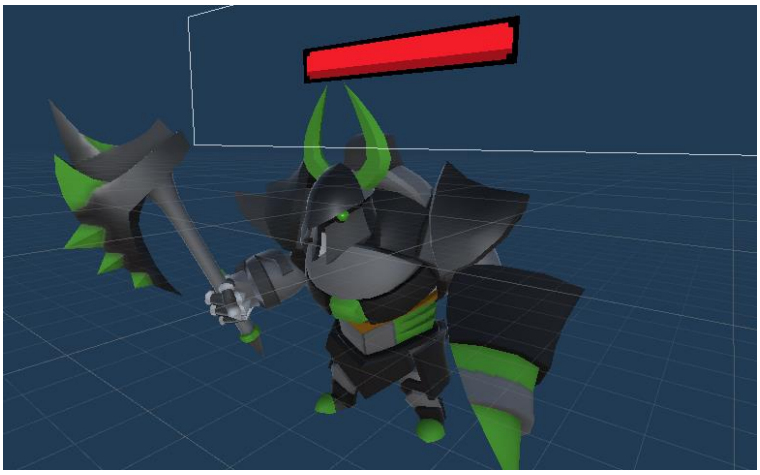


Ilustración 144. Imagen del Ganon

En realidad, es un enemigo que sigue funcionando en su mayoría de una forma muy similar al *Dodongo*: persigue al jugador, le ataca y bloquea de vez en cuando. No obstante, este monstruo es capaz de lanzar una bola de energía especial, y lo hace sin ser un evento de animación, pues en realidad no tiene animación alguna para ello. En su lugar, a través de la clase ***Ganon*** que hereda de *BlockingMonster*, dentro de la corrutina de ataque, si le toca lanzar una bola de energía llama a *GanonBehaviour::shoot* para que lo haga.

```
/// <summary>
/// Ganon will chase the player and, if it isn´t already shooting, it will shoot an energy ball
/// while attacking if necessary.
/// </summary>
/// <returns></returns>
3 referencias
public override IEnumerator specializedAttackCoroutine()
{
    while (true)
    {
        float time = Random.Range(0.1f, awaitTime);

        if (playerDetected())
        {
            if (Vector3.Distance(monsterPos.position, playerPos.position) <= attackRadius && !isAttacking())
            {
                Vector3 forward = (playerPos.position - monsterPos.position).normalized;
                float targetAngle = Mathf.Atan2(forward.x, forward.z) * Mathf.Rad2Deg;
                float angle = Mathf.SmoothDampAngle(monsterPos.transform.eulerAngles.y, targetAngle,
                    ref turnSmoothVelocity, turnSmoothTime);
                monsterPos.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

                yield return new WaitForSeconds(time);

                chooseAttack();
            }

            if (!ganonShoot.getIsShooting())
            {
                float random = Random.Range(0.0f, 1.0f);
                if (random <= ballChance)
                {
                    lastAttack = Constants.ATTACK04_TRIGGER;
                    ganonShoot.shoot();
                }
            }
        }

        yield return new WaitForSeconds(time);
    }
}
```

Ilustración 145. Corrutina de ataque del Ganon

Eso es precisamente lo que se observa en el código: de normal, persigue y ataca a meleé al jugador, pero si lo decide lanza una bola de energía. La bola de energía en sí misma también es diferente y no hacemos uso aquí del *script* de *EnergyBall*, sino de ***GanonBall***.

```
// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    // This first if is to check if the game object needs to be destroyed.
    if (secondSparksLong.time >= 1.0f)
        Destroy(gameObject);

    // This second if is to know whether the ball has already exploded, in which
    // case we disable the collider to not damage the player but we maintain the game
    // object alive until the particles have all appeared and perished.
    if (secondSparksLong.time > 0.0f)
        sphereCollider.enabled = false;

    if (!isFinished)
    {
        // This code is for chasing the player.
        Vector3 target = new Vector3(playerTransform.position.x, playerTransform.position.y + 1.0f, playerTransform.position.z);
        direction = (target - transform.position).normalized;
        transform.position += direction * speed * Time.deltaTime;

        sphereCollider.radius += sizeStep;
    }
    else
    {
        // Here we disable the collider in case that the ball has impacted the player.
        sphereCollider.enabled = false;
    }
}
```

Ilustración 146. Método Update de GanonBall

La bola de energía de *Ganon* tiene varias particularidades: sigue siempre al jugador hasta que explota y crece de tamaño conforme más tiempo lleva existiendo. Esto se ve en el código, el cual está debidamente comentado y explicado.

Ilustración 147. El asset de la bola del Ganon, proveniente del Sherbb's Particle Collection antes mencionado

- **Gel.** Se trata de un enemigo que usa **JumpingMonster**, la cual es una subclase de *Monster*.

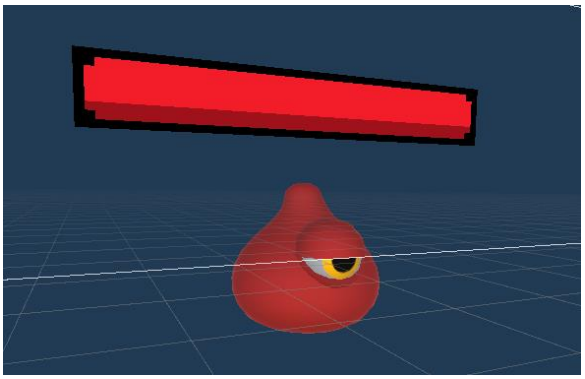


Ilustración 148. Imagen del Gel

Como su nombre indica, *JumpingMonster* es un tipo de enemigo que ataca saltando. El comportamiento del monstruo, esto es, el *monsterBehaviour*, es exactamente el mismo que para los enemigos comunes: el monstruo persigue al jugador cuando es detectado y lo ataca si está en su radio de ataque. No obstante, la forma de atacar es diferente.

```
/// <summary>
/// Basically this method applies a force to the monster rigidbody, towards the player and
/// also in the direction of the up vector.
/// </summary>
1 referencia
protected void jump()
{
    Vector3 direction = new Vector3(playerPos.position.x - monsterPos.position.x,
        0.0f, playerPos.position.z - monsterPos.position.z);

    agent.enabled = false;
    rigidbody.isKinematic = false;
    weapons[0].enabled = true;

    thisAnim.SetBool(Constants.IS_ATTACKING_CONDITION_BOOL, true);
    rigidbody.AddForce(Vector3.up * jumpForce);
    rigidbody.AddForce(direction * jumpForce);
}
```

Ilustración 149. Método jump de JumpingMonster

Como se puede ver en el código, lo que se hace es (aparte de iniciar la animación de ataque correspondiente) añadirle una fuerza vertical y otra horizontal al *RigidBody* del enemigo para que describa una parábola hacia el jugador, de manera que lo impacte con todo su cuerpo.

- **Gleeok.** Es uno de los *bosses* del juego y, además, un *StaticMonster* como el *Digdogger*. No obstante, hay una diferencia importante: el *Gleeok* se puede **teletransportar**.

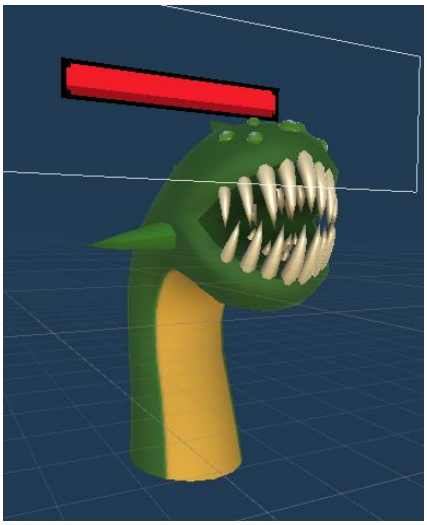


Ilustración 150. Imagen del Gleeok

Para ello, el *Gleeok* cuenta con un *script* llamado ***GleeokBehaviour***. Este *script* no solo tiene esa funcionalidad, sino que este monstruo será capaz, como el *Aquamentus*, de echar bolas y lanzar llamas de una forma idéntica.

```
0 referencias
public void teleport()
{
    teleportSound.Play();

    float newX = teleportCenter.position.x + Random.Range(-1.0f * teleportRadius, teleportRadius);
    float newZ = teleportCenter.position.z + Random.Range(-1.0f * teleportRadius, teleportRadius);
    Vector3 newPos = new Vector3(newX, transform.position.y, newZ);

    transform.position = newPos;
    GetComponent<Animator>().SetBool(Constants.IS_ATTACKING_CONDITION_BOOL, false);
}
```

Ilustración 151. Método teleport de GleeokBehaviour

Como se puede ver en la imagen, es tan fácil como que se elige un punto aleatorio dentro de un cuadrado con un centro determinado. Se considera además que teletransportarse es un ataque, por lo que puede ser elegido por la corrutina de ataque. Cuenta además con animaciones para excavar la tierra y aparecer de ella, pues no se teletransporta estrictamente hablando, sino que se sumerge en la tierra y luego sale.

- **Gohma.** Este monstruo es un *NormalBallMonster*, lo cual quiere decir que su comportamiento es perseguir al jugador y lanzar tanto ataques a meleé como un ataque con una bola de energía.

Ilustración 152. Imagen del Gohma

- **Stalfos.** Sencillamente es un *NormalBlockingMonster* al estilo del *Dodongo*.

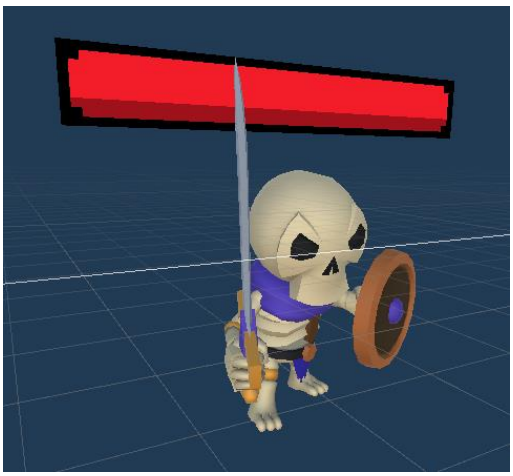


Ilustración 153. Imagen del Stalfos

- **Tektite.** Un *JumpingMonster* al estilo del *Gel*.

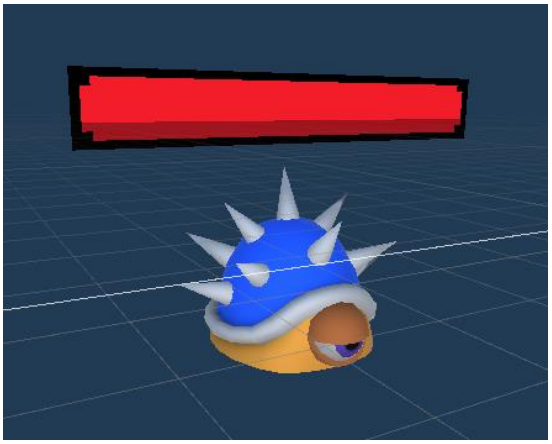


Ilustración 154. Imagen del Tektite

- **Wizzrobe.** Pese a que en esencia este enemigo es un monstruo estático como el *Gleeok* y es capaz de teletransportarse, vamos a usar una subclase de *Monster* diferente llamada como el enemigo: ***Wizzrobe***.

Ilustración 155. Imagen del Wizzrobe

La diferencia con *StaticMonster* radica en la corrutina de ataque. Puesto que, en este caso, el teletransporte del enemigo no es un ataque propiamente dicho pues no tiene animación de teletransporte como tal y, por tanto, no puede ser llamada como evento por la animación, el teletransporte ha de llamarse en el propio código.

```
7 referencias
public override IEnumerator attackCoroutine()
{
    while (true)
    {
        float time = Random.Range(0.1f, awaitTime);

        if (!isAttacking())
        {
            if (playerDetected())
            {
                Vector3 forward = (playerPos.position - monsterPos.position).normalized;
                float targetAngle = Mathf.Atan2(forward.x, forward.z) * Mathf.Rad2Deg;
                float angle = Mathf.SmoothDampAngle(monsterPos.transform.eulerAngles.y, targetAngle,
                    ref turnSmoothVelocity, turnSmoothTime);
                monsterPos.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

                yield return new WaitForSeconds(time);

                thisAnim.SetBool(Constants.IS_ATTACKING_CONDITION_BOOL, true);

                int attack = Random.Range(0, numAttacks);
                switch (attack)
                {
                    case 0:
                        thisAnim.SetTrigger(Constants.ATTACK01_TRIGGER);
                        lastAttack = Constants.ATTACK01_TRIGGER;
                        break;
                    case 1:
                        teleport();
                        break;
                }
            }
        }

        yield return new WaitForSeconds(time);
    }
}
```

Ilustración 156. Corrutina de ataque de la clase Wizzrobe

Como podemos ver, es muy similar a la corrutina habitual, salvo por el hecho de que se llama a *teleport* en ella. Este método, *Wizzrobe::teleport*, llama a su vez *WizzrobeBehaviour::teleport*, el cual es un tanto distinto al de *GleeokBehaviour*.

```
1 referencia
public void teleport()
{
    teleportCopy = Instantiate(teleportParticles);
    teleportCopy.transform.position = teleportParticles.transform.position;
    teleportCopy.GetComponent<ParticleSystem>().Play();
    teleportSound.Play();
    isTeleporting = true;
}
```

Ilustración 157. Método teleport de WizzrobeBehaviour

```
// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    if (isTeleporting)
    {
        if (teleportCopy.GetComponent<ParticleSystem>().time >= teleportTime)
        {
            float newX = teleportCenter.position.x + Random.Range(-1.0f * teleportRadius, teleportRadius);
            float newZ = teleportCenter.position.z + Random.Range(-1.0f * teleportRadius, teleportRadius);
            Vector3 newPos = new Vector3(newX, transform.position.y, newZ);

            transform.position = newPos;
            teleportParticles.GetComponent<ParticleSystem>().Play();
            GetComponent<Animator>().SetBool(Constants.IS_ATTACKING_CONDITION_BOOL, false);
            isTeleporting = false;
        }
    }

    if (teleportCopy != null)
    {
        if (!teleportCopy.GetComponent<ParticleSystem>().isEmitting)
        {
            Destroy(teleportCopy);
        }
    }
}
```

Ilustración 158. Método Update de WizzrobeBehaviour

La diferencia con respecto al teletransporte del *Gleeok* es que aquí, aparte de no contar con animaciones dedicadas, usamos un sistema de partículas (proveniente del *Sherbb's Particle Collection* mencionado antes), y de él depende cuándo se teletransporta el *Wizzrobe*. El método *teleport* básicamente inicia ese sistema de partículas y, cuando lleva un cierto tiempo reproducido, el monstruo efectúa el cambio de posición.

- **Zol.** Es el último enemigo que nos queda por repasar. Se trata de un *Gel* más grande y fuerte y que, al igual que el *Vire*, al morir hace que aparezcan una serie de enemigos, en este caso *Gels* más pequeños.

3.2.7 Mecánica de fijar

Esta mecánica consiste en que el personaje sea capaz de **fijar** al enemigo que tiene más cerca, lo cual quiere decir que la cámara se quedará fija mirando a ese enemigo y nuestro personaje rotará en torno a dicho enemigo. Esta mecánica hace que tengamos que añadir ciertas funcionalidades extra a mecánicas anteriores que hemos visto, fundamentalmente a **rodar** y **andar**.

Como ya hemos visto, el personaje al rodar y al andar se desplazaba únicamente hacia delante en cualquier caso. No quiere decir esto que el personaje únicamente se desplazase hacia delante, sino que siempre se giraba hacia la dirección a la que tenía que desplazarse y andar hacia delante en esa dirección. Por tanto, solo teníamos una única animación para andar y rodar.

Sin embargo, al fijar al enemigo el personaje siempre está mirando hacia él, por lo que ahora necesitamos una animación para cada nueva dirección: hacia atrás, izquierda y derecha. Por ello, el grafo de animaciones de andar se complica y ahora es el siguiente:

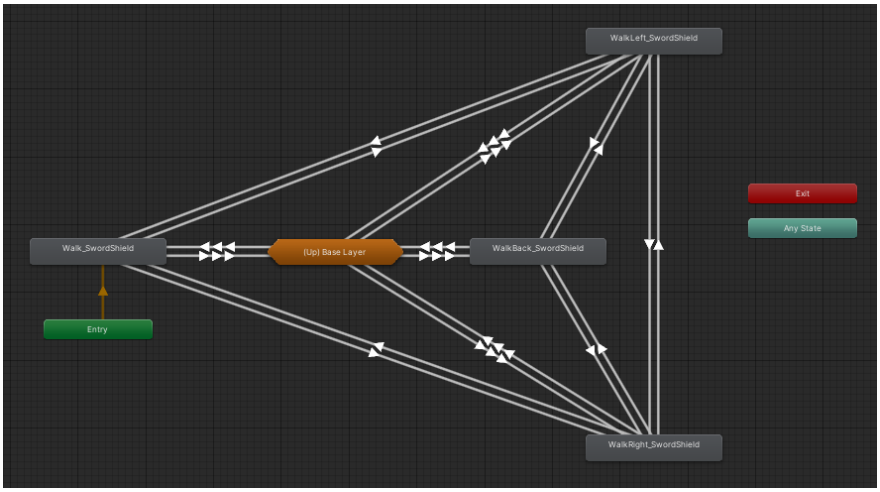


Ilustración 159. Grafo de animaciones del jugador al moverse

Puede resultar complicado, pero en realidad no lo es tanto; sencillamente, ahora vamos a tener una *flag* llamada ***isTargeting*** que nos indica si el jugador está fijando. Evidentemente, si el jugador no está fijando no se podrá acceder a otra animación que no sea la de ir hacia delante. No obstante, si lo está haciendo, entonces hay que discernir en qué dirección está yendo para saber qué animación usar en cada caso. Para ello, tenemos una serie de *flags* en el *Animator* que nos lo indica:

isMovingForward, isMovingBack, isMovingLeft y *isMovingRight*. Además, para saber qué *flag* activar en cada momento, nos vamos a la clase *Move* y creamos un nuevo método: ***setMovingDirection***.

El código es farragoso, pero la idea es muy simple: calculamos las distancias entre el vector dirección que está siguiendo el jugador y las cuatro direcciones cardinales, y nos quedamos con la **dirección cardinal más cercana**, de manera que activamos su *flag* correspondiente.

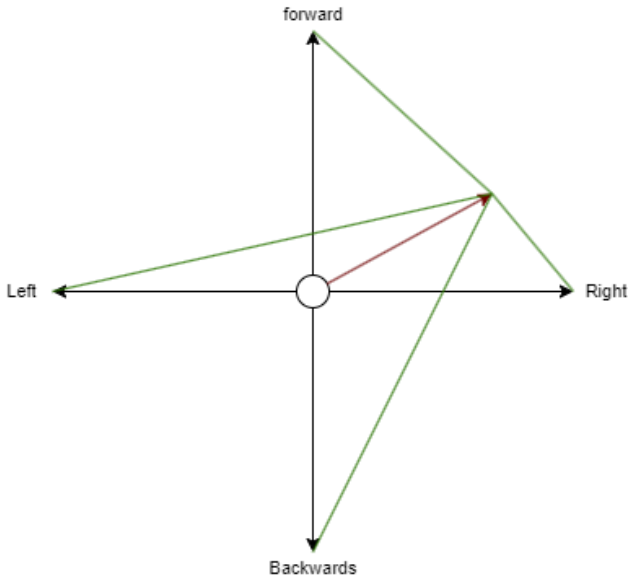


Ilustración 160. Cálculo de las distancias entre el vector dirección y las direcciones cardinales

En la imagen de arriba, el origen de coordenadas sería el jugador y las direcciones cardinales estarían relativas a donde esté mirando, por lo que el *forward* mirará hacia la posición del enemigo.

```
13 referencias
public override void execute()
{
    // Direction of movement.
    Vector3 direction = new Vector3(playerMove.x, 0.0f, playerMove.y).normalized;

    // Setting of the moving direction (for the different types of rolling).
    if (!isRolling() && !isBeaten())
    {
        setMovingDirection();
    }

    if (!isAttacking() && !isRolling() && !isBeaten())
    {
        // Setting of the speed depending on if the player is blocking, walking or running.
        if (isBlocking())
            actualSpeed = 0.0f;
        else if (!isRunning())
            actualSpeed = player.getPlayerSpeed();
        else
            actualSpeed = player.getRunningSpeed();

        // We tell the animator controller to execute the walking animation via the float speed.
        animator.SetFloat(Constants.SPEED_CONDITION_FLOAT, direction.magnitude);

        /* Two cases of calculating the movement direction:
            1. The player isn't targeting, in which case we, first, calculate the target angle we want
               the player to be rotated by, which will be the direction angle given by the input rotated by the
               rotation of the camera itself; that way, if we move forward, that forward direction will be the
               forward of the camera. We don't want the player to rotate instantly, so we use SmoothDampAngle,
               which is a function that gradually changes an angle given in degrees towards a desired goal angle over time,
               and we rotate the player by that angle until we reach that goal angle.
            2. The player is targeting, in which case the player will always be looking at the enemy, so we just move
               the player considering the targetTransform forward their own forward. That transform always look at the enemy
               target. Note that in this case the player does not rotate; that's because while they are targeting, they don't
               actually always look at the enemy: if they are running, they won't.*/

        if (direction.magnitude >= 0.1f)
        {
            Vector3 moveDirection;
            if (!isTargeting())
            {
                float targetAngle = Mathf.Atan2(direction.x, direction.z) * Mathf.Rad2Deg + Camera.main.transform.eulerAngles.y;
                float angle = Mathf.SmoothDampAngle(player.transform.eulerAngles.y, targetAngle,
                    ref turnSmoothVelocity, player.getTurnSmoothTime());
                player.transform.rotation = Quaternion.Euler(0.0f, angle, 0.0f);

                moveDirection = Quaternion.Euler(0.0f, targetAngle, 0.0f) * Vector3.forward;
            }
            else
            {
                moveDirection = Quaternion.Euler(0.0f, player.getTargetTransform().rotation.eulerAngles.y, 0.0f) * direction;
            }

            controller.Move(moveDirection.normalized * actualSpeed * Time.deltaTime);
        }
    }
}
```

Ilustración 161. Método execute de Move

Como se puede ver en el código, el jugador se moverá de diferentes formas **dependiendo de si está fijando** o no. En este primer caso, la dirección de movimiento original se rotará de acuerdo a la rotación de *targetTransform*, que es un *Transform* situado en la jerarquía de objetos del personaje del jugador y en una posición trasera que siempre está mirando hacia el enemigo mientras el jugador esté fijando; por tanto, hacemos que la dirección de movimiento esté relativa a la dirección entre el jugador y el enemigo, haciendo que esa dirección sea su *forward*, por así decirlo.

Para el rodar es muy similar. Contamos con un grafo de animaciones muy parecido donde se dan unas animaciones u otras dependiendo de la dirección cardinal que nos dicta *Move::setMovingDirection* y dependiendo de si el jugador está fijando o no (ver grafo de animaciones de rodar, presentado con anterioridad). Con respecto al código de *Roll*, los cambios van orientados también en esta dirección: sencillamente, ahora se hace una cosa si el jugador está fijando y otra si no lo está haciendo.

Toca hablar ahora de todo lo concerniente a la **cámara**. El fundamento es hacer que la cámara se mueva de la posición que ocupa a una posición que se encuentre detrás del personaje para que el jugador tenga una visión más o menos clara. Para ello, en realidad vamos a cambiar entre la cámara principal y otra cámara que se encargue del fijado. La idea es que este cambio no sea brusco y que se vea el movimiento de la cámara de la posición inicial a la final.

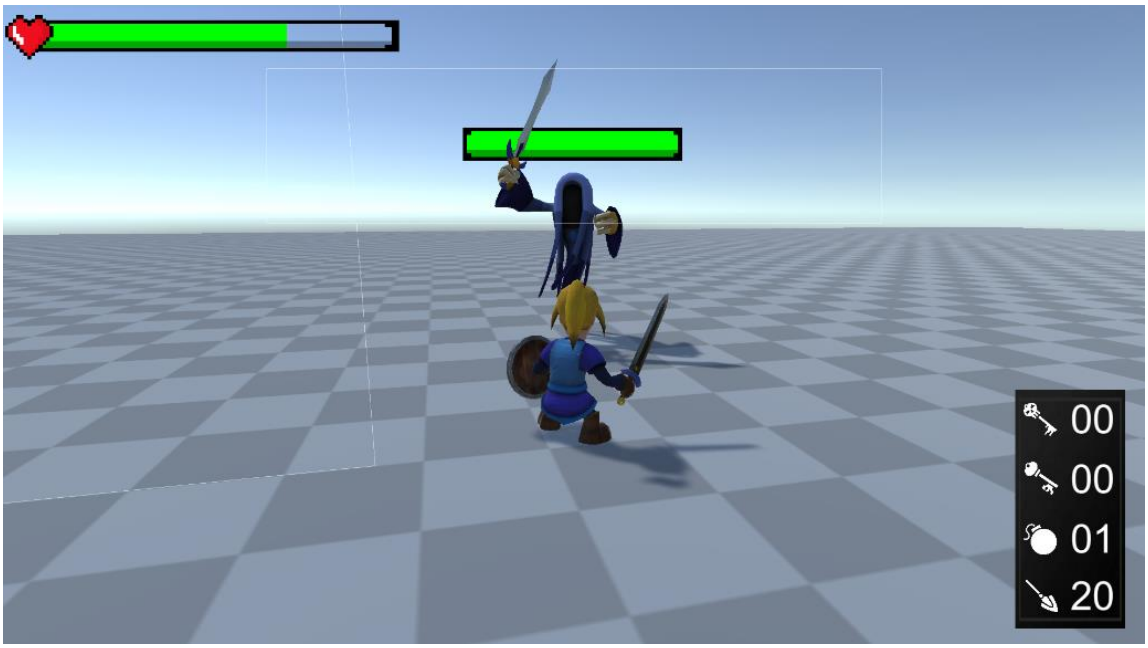


Ilustración 162. Vista del juego cuando el personaje está fijando un enemigo

Se creó un *script* subclase de *Command* llamado ***Target*** para llevar a cabo la acción de fijar, de manera que cuenta con el evento denominado *Target::setTarget* y llamado por *InputHandler* (a través de *Player*, como el resto de hijos de *Command* que abstraen alguna acción del jugador).

```
/// <summary>
/// Callback that is called by the Input Handler for targeting an enemy. It finds the colliders
/// inside the detection radius, sorts it and tells the camera to start the traveling.
/// </summary>
1 referencia
public void setTarget()
{
    findTargets();

    if (targetColliders.Count > 0)
    {
        sortTargets();

        if (enemyTarget == null)
            enemyTarget = sortedColliders[0].Item2.transform;

        if (isTargeting())
        {
            setIsTargeting(false);
            player.getPlayerCamera().reverseTargetTraveling();
        }
        else
        {
            setIsTargeting(true);
            player.getPlayerCamera().startTargetTraveling(enemyTarget.position);
        }
    }
}
```

Ilustración 163. Método setTarget de Target

Dentro de este método podemos ver que, primero, se buscan los posibles *targets*, es decir, los posibles enemigos que se encuentran dentro del radio de detección del jugador por medio del método ***findTargets***. La idea de esto es simple: se usa el método *Physics.OverlapSphere* el cual devuelve todos los *Colliders* que hay dentro de una cierta esfera, en este caso con radio el radio de detección del jugador y centro el propio jugador. Posteriormente, se comprueba si ese *Collider* es de un enemigo y se realiza un *raycast* con el objetivo de saber si el jugador tiene al enemigo visible.

Una vez hemos buscado posibles enemigos a los que fijar, los ordenamos según su distancia con respecto al jugador por medio del método ***sortTargets*** y cogemos el más cercano al jugador. Dependiendo de si ya estábamos fijando o no, el jugador bien sale del modo fijar y vuelve a la cámara normal (porque se sale de este modo pulsando el mismo botón que para fijar), bien empieza a fijar al enemigo más cercano llamando al método *FreeLookCam::startTargetTraveling*. En esta misma clase es donde está todo el código concerniente al ***traveling*** que realiza la cámara desde su posición de origen hasta su posición final.

```
/// <summary>
/// Function called when the player wants to start targeting an enemy and the camera
/// performs a traveling until it reaches the target transform position. It initializes
/// the target camera with the same parameters of the main camera and the initial positions
/// and rotations.
/// </summary>
/// <param name="enemyPosition"></param>
1 referencia
public void startTargetTraveling(Vector3 enemyPosition)
{
    isTargeting = true;

    targetCameraTransform.position = cameraTransform.position;
    targetCameraTransform.forward = cameraTransform.forward;

    cameraTransform.gameObject.SetActive(false);
    targetCameraTransform.gameObject.SetActive(true);

    // This calculations below are here because the target transform don't follow the
    // player correctly even if it's in the hierarchy when the player runs, so we basically
    // place the target transform in a line that we can draw between the enemy position
    // and the player, so the target camera is placed always behind the player and looking
    // at the enemy.
    float distancePlayer = Vector3.Distance(cameraTransform.position, playerTransform.position);

    targetTransform.position = playerTransform.position + Vector3.up * 2;
    targetTransform.position += (playerTransform.position - enemyPosition).normalized * distancePlayer;
    targetTransform.LookAt(enemyPosition);

    startRotation = targetCameraTransform.rotation;
    endRotation = targetTransform.rotation;
    startPosition = targetCameraTransform.position;
    endPosition = targetTransform.position;

    // Keep a note of the time the movement started.
    startTime = Time.time;
    Debug.Log("Start traveling");
}
```

Ilustración 164. Método FreeLookCam::startTargetTraveling

Este método inicializa el *traveling* de la cámara, de forma que desactivamos la cámara principal y activamos la cámara para fijar, y la inicializamos con los mismos parámetros que la principal para que empiece desde la misma posición con la misma rotación y demás, para que no se note el salto de una cámara a otra. A partir de aquí, se inicializa además la posición y rotación finales que tendrá la cámara una vez complete el *traveling*, las cuales vienen dadas por el *targetTransform* antes mencionado.

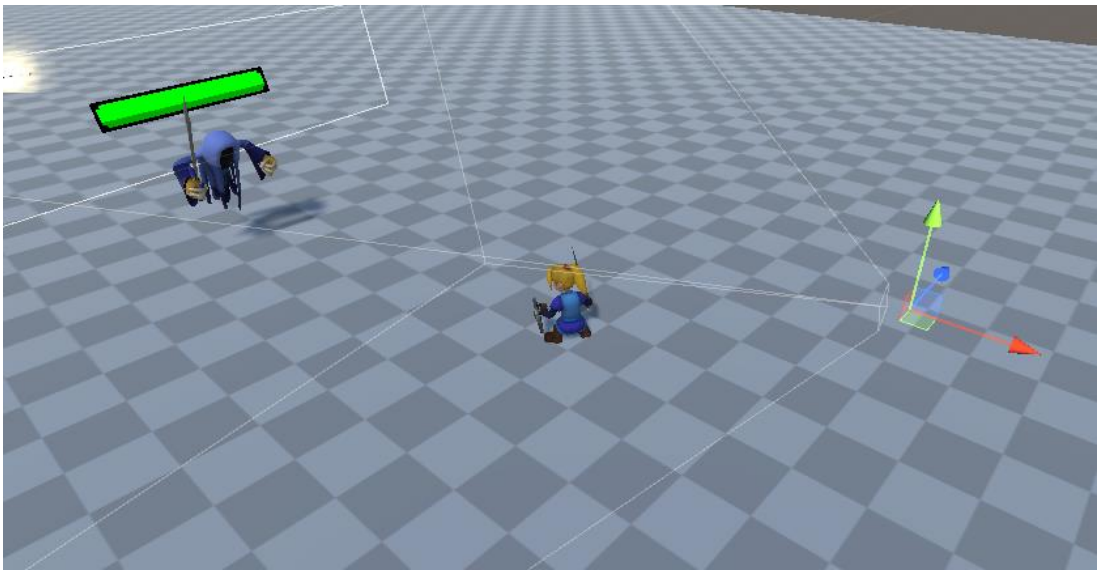


Ilustración 165. Posición de la cámara cuando el jugador está fijando

De esta forma, una vez que se ha llamado a este método, cada *frame* se llamará a su vez a *FreeLookCam::targetTraveling*. Este método realiza las interpolaciones necesarias para que la cámara llegue a su posición final. Estas interpolaciones son, primero, una **interpolación lineal** entre la posición inicial de la cámara y la final marcada por la posición del *targetTransform* que se realiza por medio de la función *Vector3.Lerp* y que hace uso de la fórmula $p_i = p_0(1-t) + p_n t$ tal que $i \in [0, n]$ (U. Technologies, n.d.-f), donde n es el número de fotogramas que ocurren durante el tiempo t, p_0 es la posición inicial, p_n la final y p_i las posiciones interpoladas entre las dos. Esta es la interpolación más sencilla, cuya intuición es que mientras mayor sea la t, el punto interpolado estará más cerca de la posición final y más lejos de la inicial (de ahí que se use $1-t$ para la inicial y t para la final). Luego, tenemos una **interpolación lineal esférica** realizada a través de *Quaternion.Slerp* para que la cámara vaya rotando de su rotación inicial a la final. Las interpolaciones esféricas son mucho más complejas y requieren de cuaterniones, por lo que lo único que vamos a destacar es que comenzamos con el cuaternión inicial que representa la rotación de la cámara al inicio y, por cada *frame*, la función nos devuelve un cuaternión interpolado, hasta llegar al cuaternión final.

```
/// <summary>
/// This function performs the interpolation during the camera animation from
/// the original position to the target camera final position.
/// </summary>
/// <param name="enemyPosition"></param>
1 referencia
public void targetTraveling(Vector3 enemyPosition)
{
    // This calculation is the same as the startTargetTraveling.
    float distancePlayer = Vector3.Distance(cameraTransform.position, playerTransform.position);

    targetTransform.position = playerTransform.position + Vector3.up * 2;
    targetTransform.position += (playerTransform.position - enemyPosition).normalized * distancePlayer;
    targetTransform.LookAt(enemyPosition);

    endRotation = targetTransform.rotation;
    endPosition = targetTransform.position;

    // Set our position as a fraction of the distance between the markers.
    targetCameraTransform.rotation = Quaternion.Slerp(startRotation, endRotation, (Time.time - startTime) * 2.0f);
    targetCameraTransform.position = Vector3.Lerp(startPosition, endPosition, (Time.time - startTime) * 2.0f);
}
```

Ilustración 166. Método FreeLookCam::targetTraveling

Una vez el jugador quiere dejar de fijar o mata al enemigo o este sale del radio de detección el sistema llama a *FreeLookCam::reverseTargetTraveling*, que no hace más que desactivar la cámara de fijado y activar la cámara normal.

3.2.8 Mecánica de correr

Esta mecánica es muy sencilla ya que es sencillamente que el personaje se desplace a **mayor velocidad** y solamente añade una nueva animación. En la ilustración del método *Move::execute* se puede ver que la velocidad cambia dependiendo de si se está corriendo o no, y en el *Animator* únicamente se añade un nuevo estado.

No obstante, conviene recalcar el detalle de que el jugador no mira al enemigo fijado mientras está corriendo, sino que lo hace hacia la dirección de movimiento, ya que solo tenemos una animación de correr hacia delante y no hacia ninguna otra dirección.

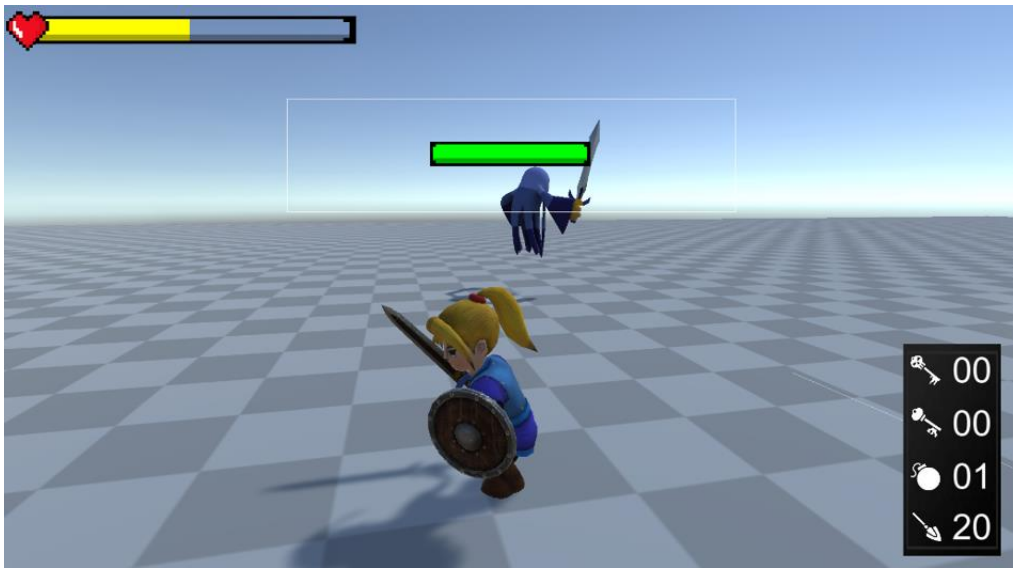


Ilustración 167. El personaje del jugador corriendo alrededor del enemigo mientras lo tiene fijado

3.2.9 Mecánicas del arco

A continuación, se implementó todo lo concerniente a la **mecánica de usar el arco**. Para ello, lo primero de todo fue añadir la posibilidad de cambiar de armas. Con ese objetivo se añadió a *Player* el método ***changeWeapon***, que fundamentalmente cambia el *Animator* del personaje del de espada y escudo al del arco. Por tanto, esto también supone la creación de un grafo de animaciones enteramente nuevo, pero en realidad es una copia prácticamente exacta del *Animator* de espada y escudo, con la única diferencia de que solo tiene un ataque en vez de un combo de ataques y que las animaciones son equivalentes (animación de andar, de rodar…) pero con la postura del arco en vez de la de la espada y el escudo.

```
/// <summary>
/// Function that is called whenever we change of weapon. It will change the current using animator
/// and will activate the corresponding weapons.
/// </summary>
2 referencias
private void changeWeapon()
{
    bool isTargeting;
    switch (weaponsSelector.Current())
    {
        case Constants.WeaponSelector.SwordShield:
            activateWeaponRight(Constants.SWORD1_NAME);
            activateWeaponLeft(Constants.SHIELD1_NAME);

            isTargeting = animator.GetBool(Constants.IS_TARGETING_CONDITION_BOOL);
            animator.runtimeAnimatorController =
                (RuntimeAnimatorController)Resources.Load("Animators/SwordShield", typeof(RuntimeAnimatorController));
            animator.SetBool(Constants.IS_TARGETING_CONDITION_BOOL, isTargeting);
            break;
        case Constants.WeaponSelector.Bow:
            activateWeaponRight(Constants.ARROW1_NAME);
            activateWeaponLeft(Constants.BOW1_NAME);

            isTargeting = animator.GetBool(Constants.IS_TARGETING_CONDITION_BOOL);
            animator.runtimeAnimatorController =
                (RuntimeAnimatorController)Resources.Load("Animators/Bow", typeof(RuntimeAnimatorController));
            animator.SetBool(Constants.IS_TARGETING_CONDITION_BOOL, isTargeting);
            break;
    }
}
```

Ilustración 168. Método Player::changeWeapon

Ilustración 169. Personaje con el arco

Una vez tenemos la posibilidad de cambiar de arma, toca **hacer que el arma dispare flechas**. Para ello, se creó **BowAttack**, una subclase de *Attack*, cuyo evento de ataque (*BowAttack::attack*) consiste en instanciar una flecha de manera que apunte bien hacia delante cuando el personaje no esté fijando a nadie, bien hacia el enemigo que esté fijando. Una vez se crea la flecha, se activa el *script* ***Arrow***, que básicamente se encarga de su movimiento, el cual se realiza añadiéndole una **fuerza** a su *RigidBody*. Hay que destacar que el método *BowAttack::attack* no se llama, como sería habitual, en el *Update* de *Player*, sino que es un evento que llama la animación

de disparo del arco a través de *Player::throwArrow*, de forma que la flecha se dispara justo cuando en la animación el personaje la suelta.

3.2.10 Generación procedural del castillo

El algoritmo ha sido fundamentalmente explicado en anteriores apartados, pero aquí vamos a echarle un vistazo rápido a la implementación. Partimos de la clase ***Dungeon***, que es la clase central para la generación de la mazmorra. Vistas las etapas del algoritmo en el apartado 2.2.2.1, cada una de ellas coincide con una de las funciones de *Dungeon*:

- ***generateRooms*** es la función que inicia el proceso recursivo de subdivisión por medio del método ***splitRect*** que es la función recursiva que divide un rectángulo en dos mitades y esas dos mitades a su vez en otras dos y así.

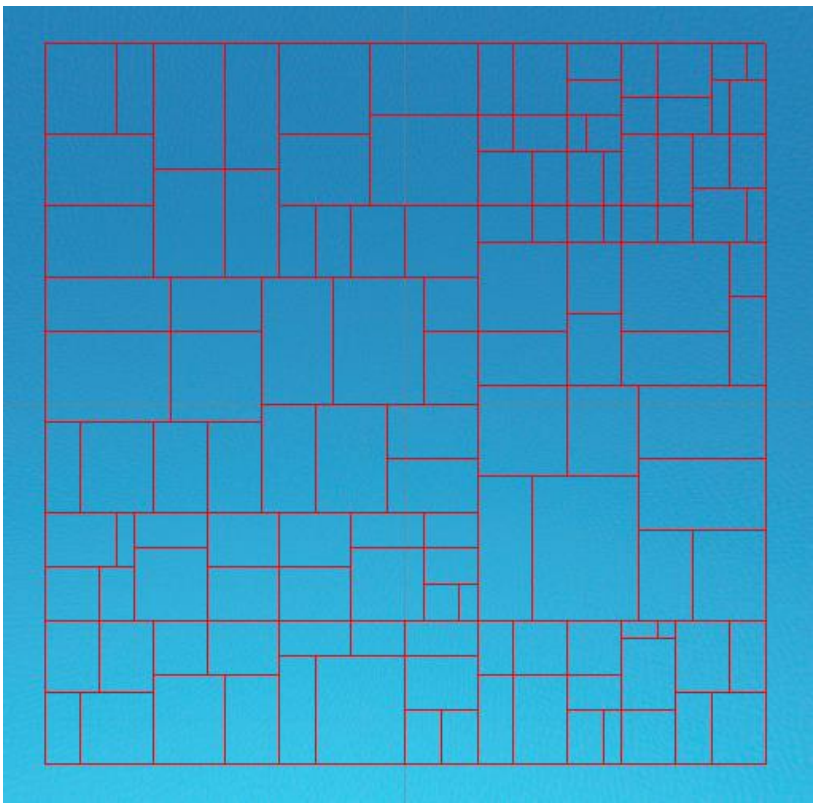


Ilustración 170. Ejemplo real de generación de habitaciones usando la subdivisión binaria

- La función encargada de asignar las habitaciones principales tal y como se explicó en anteriores apartados se denomina ***assignMainRooms***.

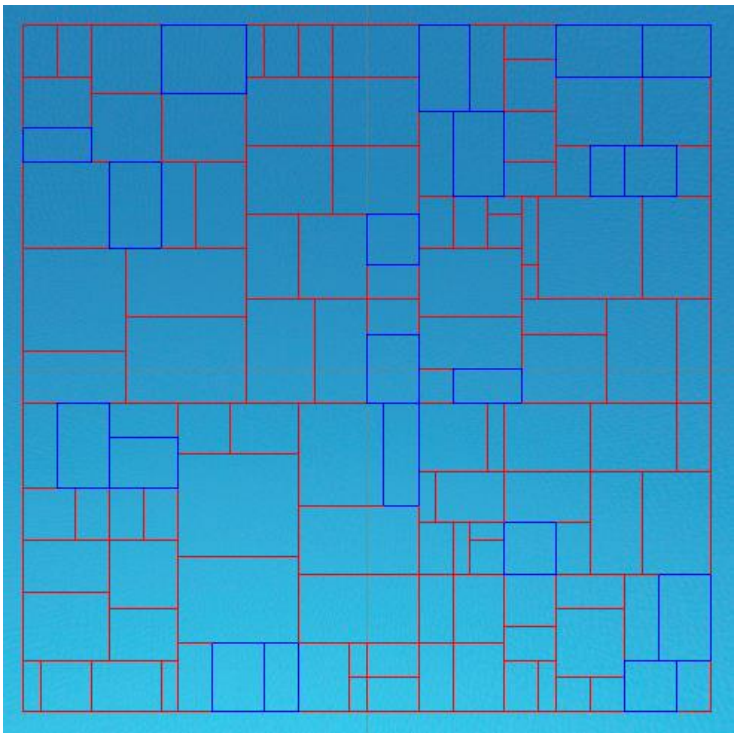


Ilustración 171. Ejemplo real de asignación de habitaciones principales (en azul)

- El paso siguiente del algoritmo es el cálculo de la triangulación de *Delaunay*. Para este paso, se ha usado el método ***computeDelaunay*** y la librería *Delaunator*.

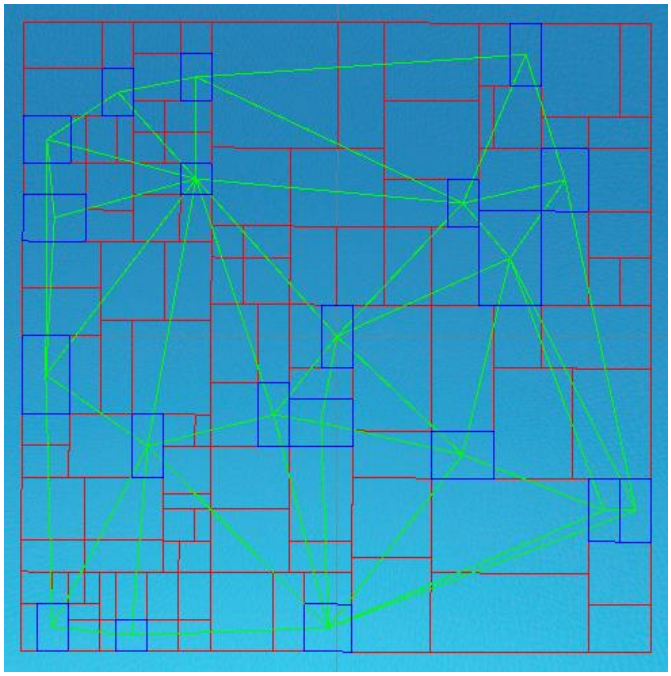


Ilustración 172. Ejemplo real de cálculo del Delaunay (en verde)

- A continuación, se realizaba el cálculo del MST por medio de la función ***computeDungeonGraph*** que además usa la clase ***MinimumSpanningTree***; esta, a su vez, se encarga del cálculo del Kruskal.

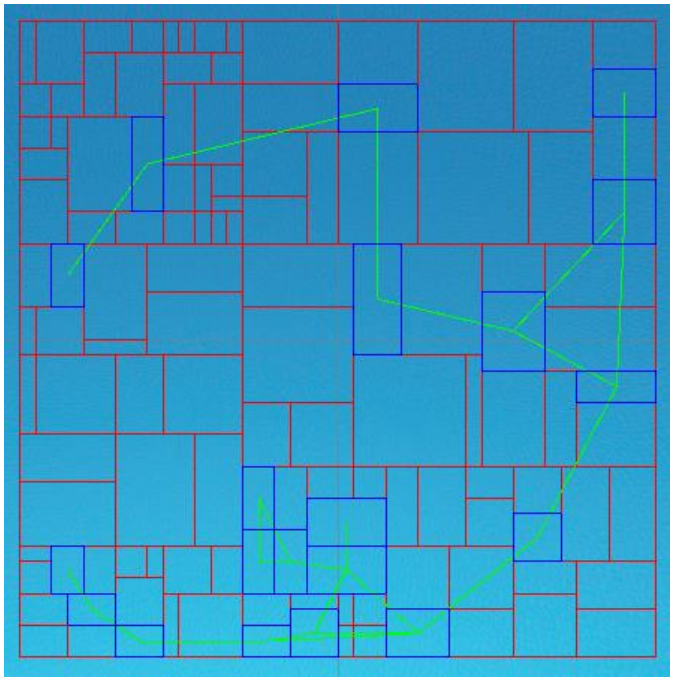


Ilustración 173. Ejemplo real de cálculo del MST

- El siguiente paso (que es el cálculo de pasillos) es realizado por la función ***computeHallways***.

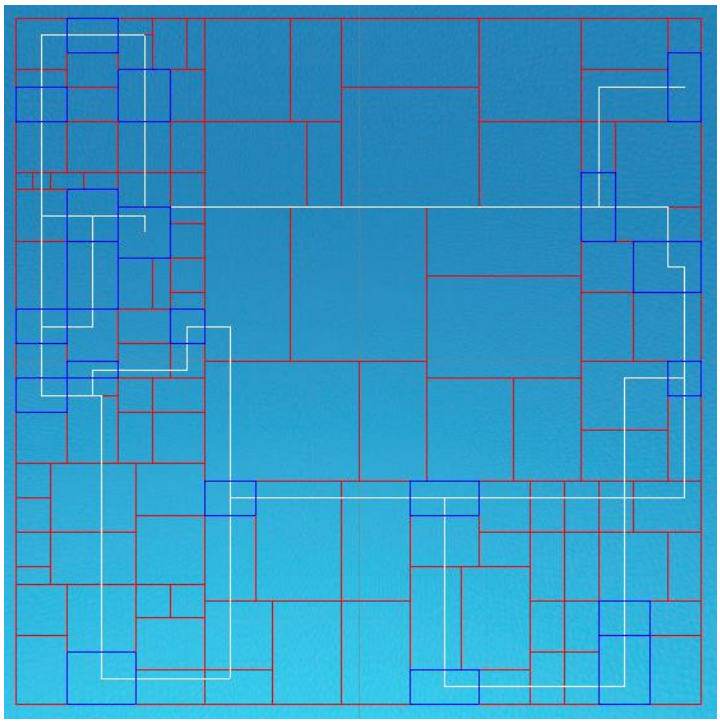


Ilustración 174. Ejemplo real de generación de pasillos (en blanco)

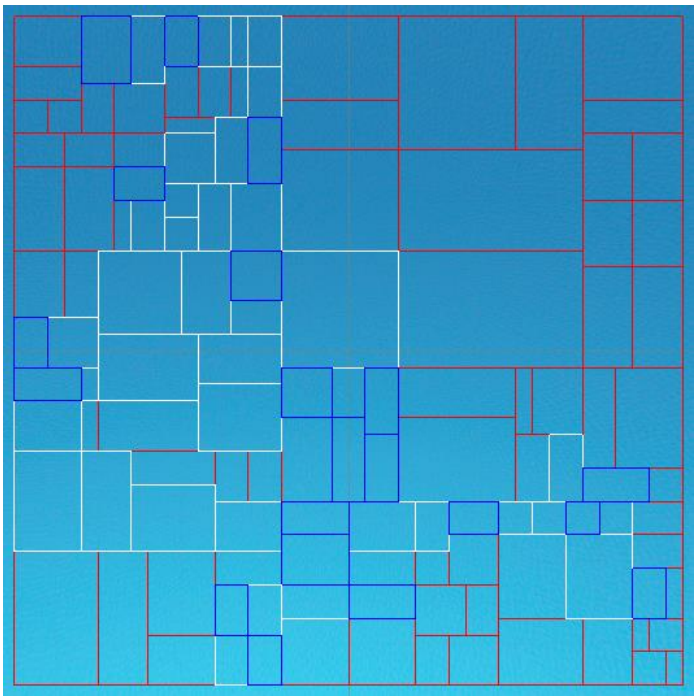


Ilustración 175. Ejemplo real de asignación de habitaciones como pasillos entre las habitaciones principales (en blanco)

- A continuación, venía un paso sencillo: la eliminación de habitaciones por defecto, es decir, las que no se usan para la mazmorra. Para ello usamos la función ***clearDefaultRooms***.

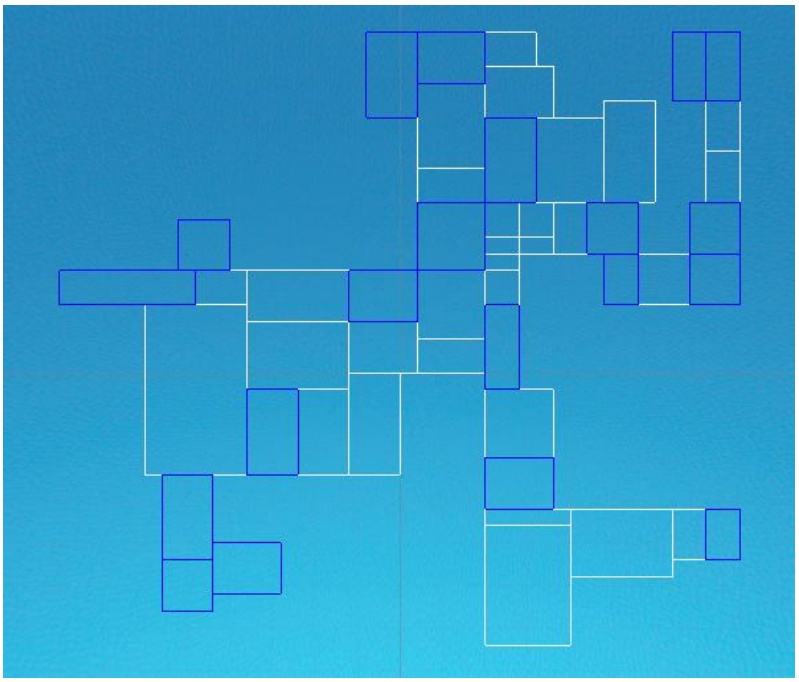


Ilustración 176. Ejemplo real de una mazmorra sin habitaciones por defecto

- Tocaba ahora añadir las habitaciones especiales, lo cual se hace por medio del método ***addSpecialRooms***.

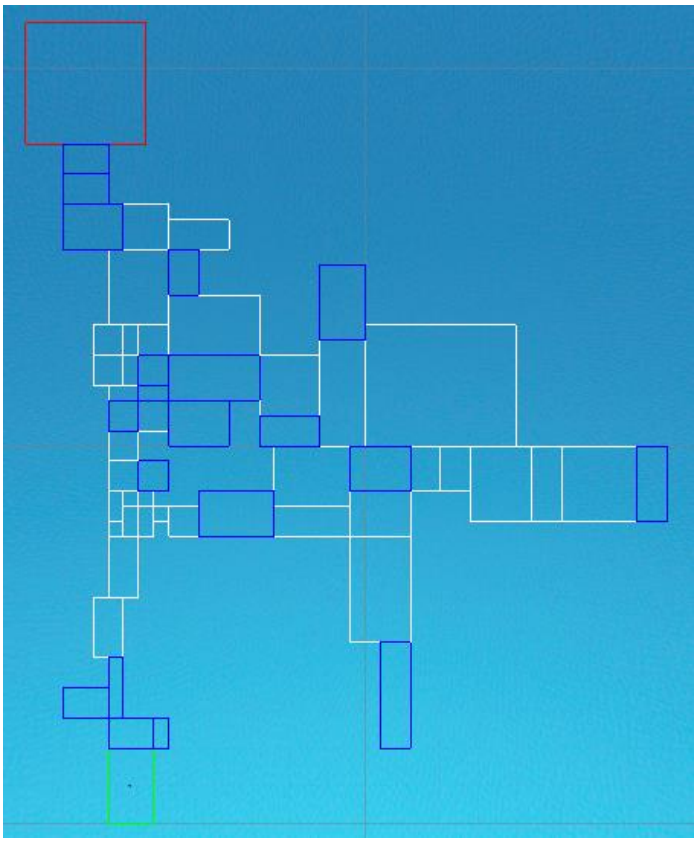


Ilustración 177. Ejemplo real de mazmorra con las habitaciones de spawn (verde) y del boss (rojo)

- Siendo de los últimos pasos, calcular los vecindarios y las puertas es algo que realiza la función ***computeNeighborhoods*** por medio de las funciones ***Room::isNeighbor*** y ***Room::computeDoor***, que usan los algoritmos explicados en apartados anteriores.

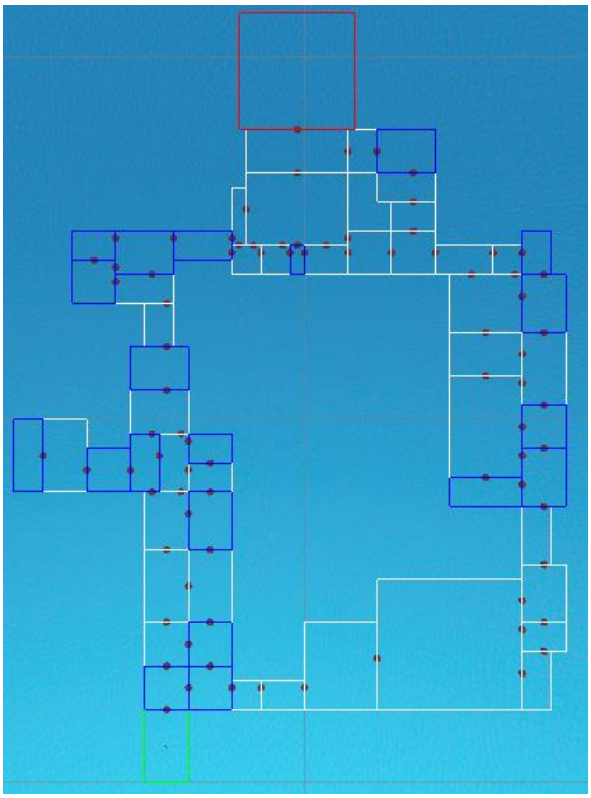


Ilustración 178. Ejemplo real de cálculo de puertas para una mazmorra, representadas por los puntos rojos

- Por último, queda la tarea de visualizar la mazmorra. Para ello, se ha usado la función ***visualizeRooms*** que a su vez tiene tres etapas: la generación del suelo de cada habitación, de lo cual se encarga ***createBasicRoom***; después la creación de cada muro, para lo cual se hace uso de ***instatiateWall***; y, por último, la creación de la utillería y, al mismo tiempo, de los enemigos de cada habitación, para lo cual tenemos ***instatiateProps***.

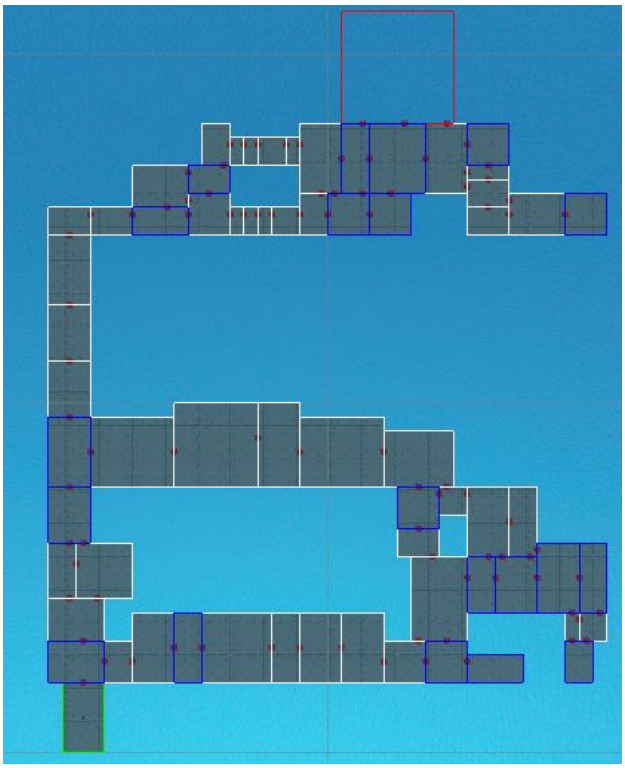


Ilustración 179. Ejemplo real de colocación del suelo, lo cual se hace para todas las habitaciones excepto la del boss

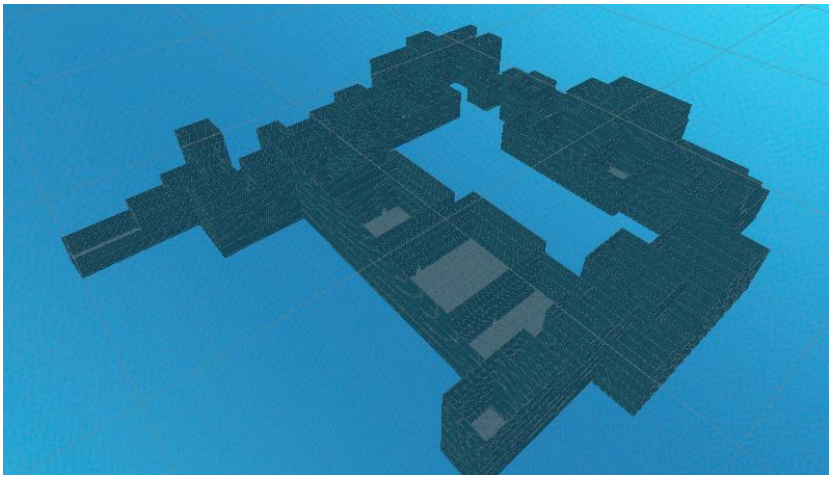


Ilustración 180. Ejemplo real de creación de los muros

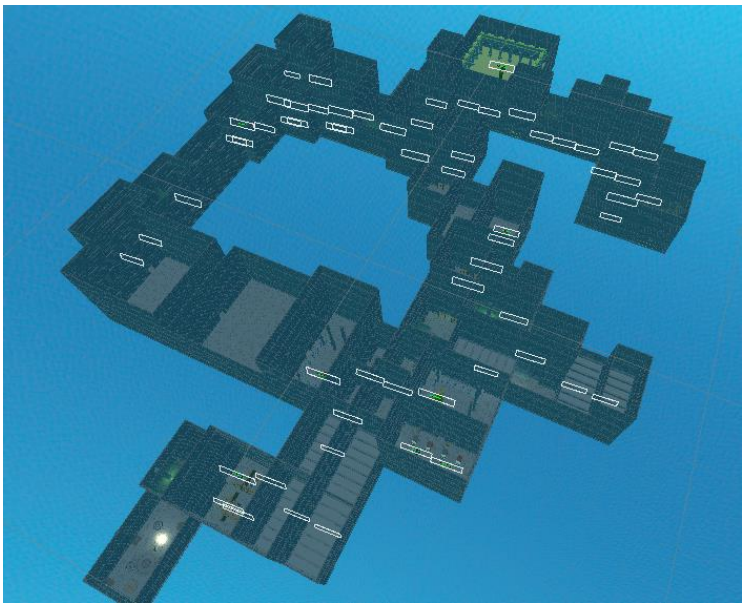


Ilustración 181. Creación de los props de cada habitación

Además, convendría ver cuáles son las diferentes utillerías que se han usado para cada tamaño de habitación, provenientes del paquete *Low Poly Dungeons* y del *Pure Nature*:

Ilustración 182. Props de la habitación del boss

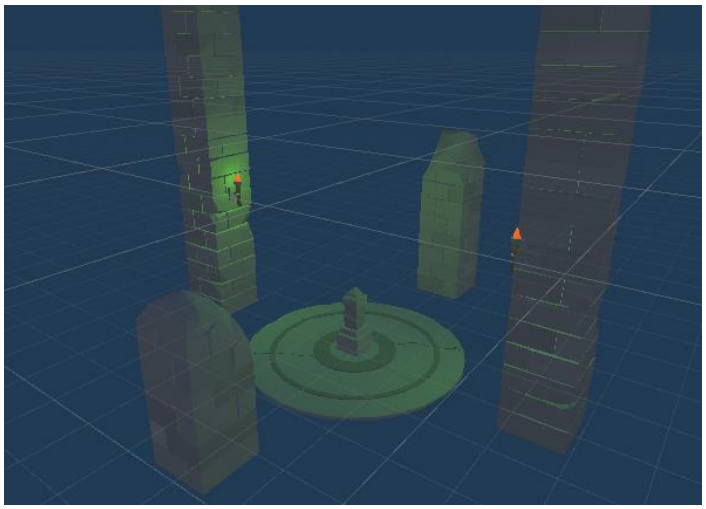


Ilustración 183. Props de las habitaciones 2x2

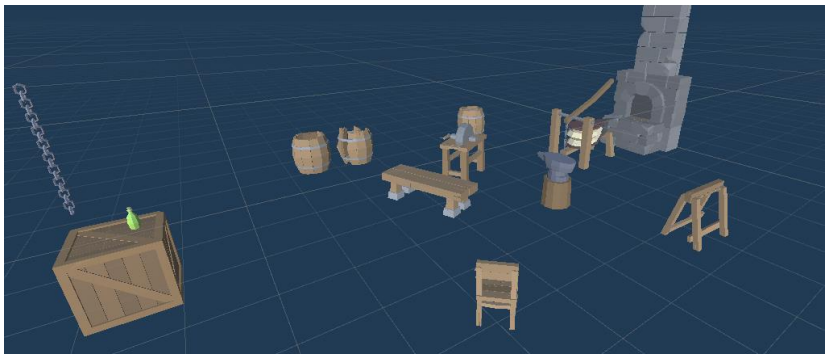


Ilustración 184. Props de las habitaciones 3x2/2x3

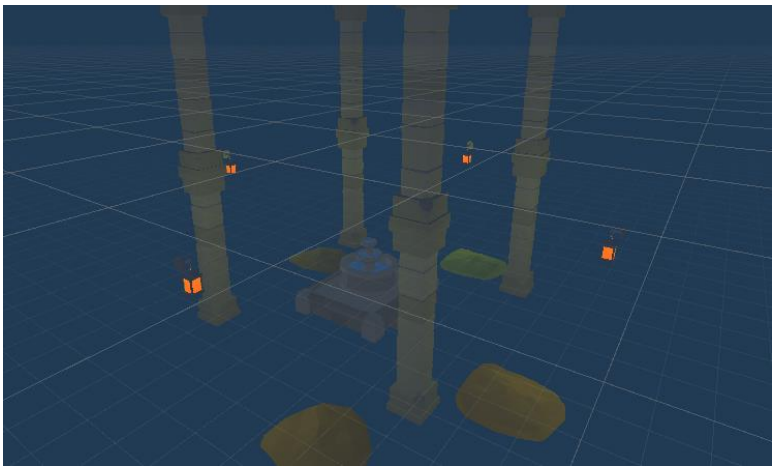


Ilustración 185. Props de las habitaciones 4x4

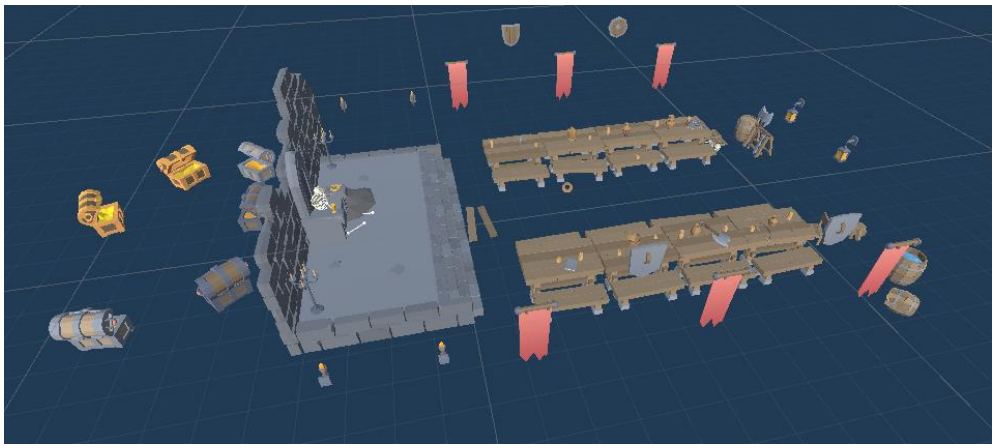


Ilustración 186. Props de las habitaciones 3x6/6x3

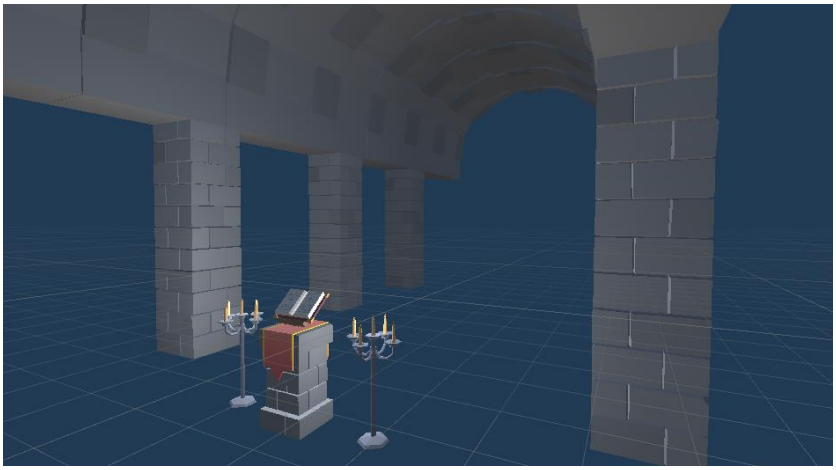


Ilustración 187. Props de las habitaciones 4x2/2x4

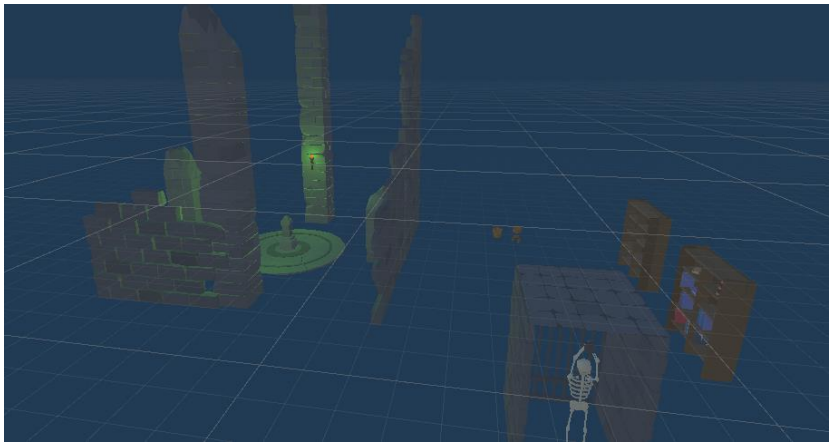


Ilustración 188. Props de las habitaciones 4x3/3x4

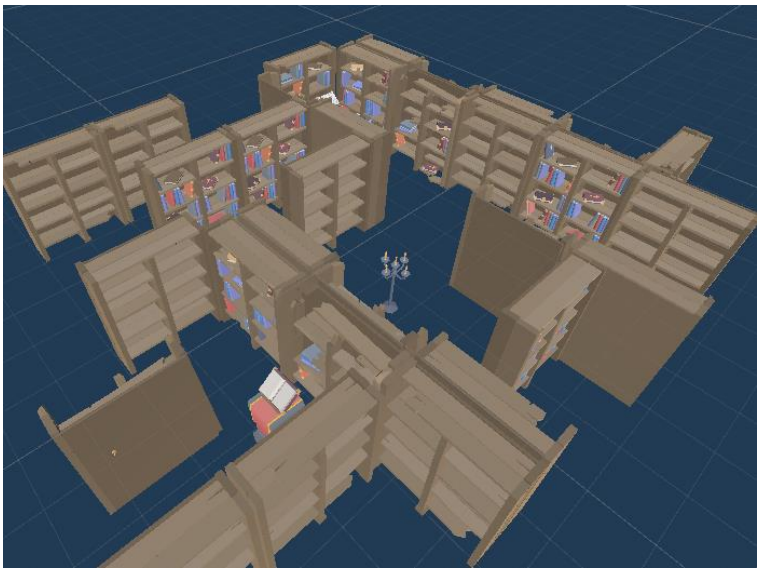


Ilustración 189. Props de las habitaciones 4x4

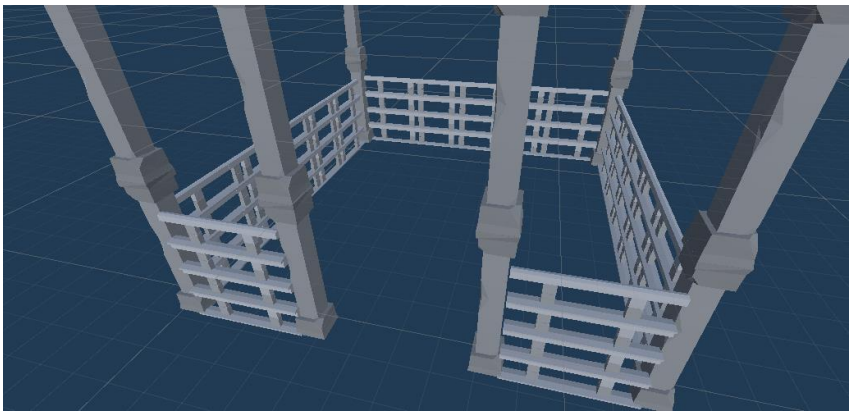


Ilustración 190. Props de las habitaciones 4x5/5x4

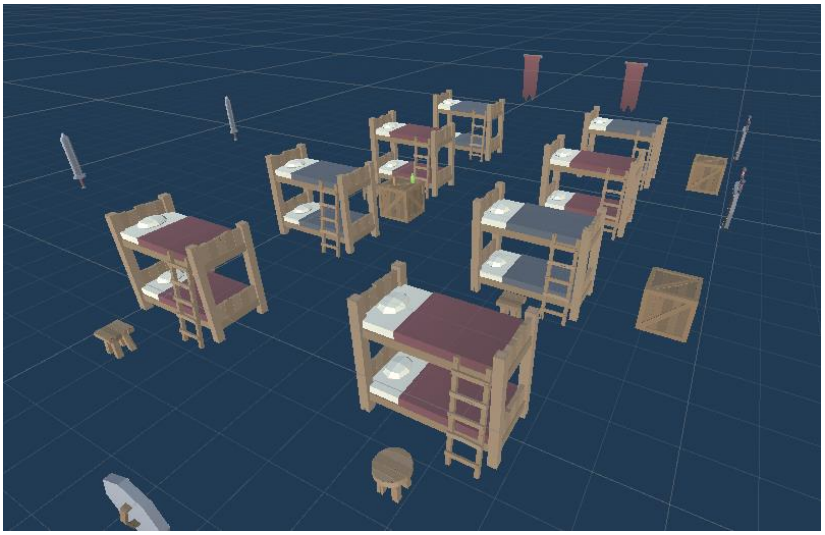


Ilustración 191. Props de las habitaciones 5x3/3x5

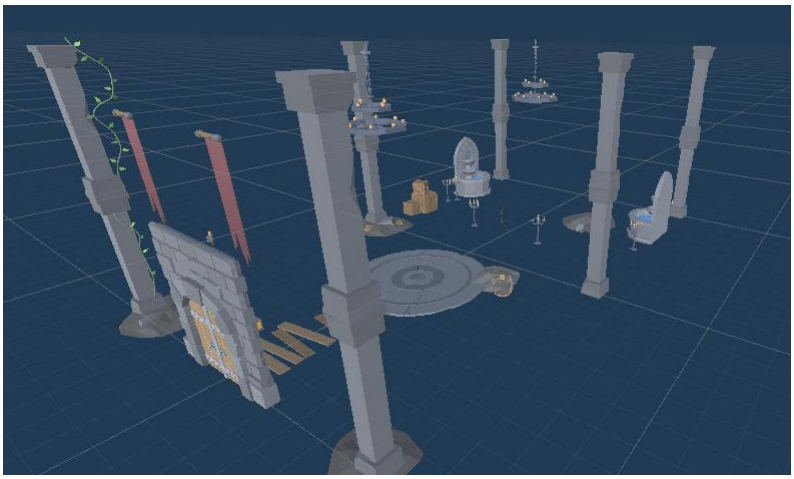


Ilustración 192. Props de la habitación inicial

Los *props* de las habitaciones 2x5/5x2, 2x6/6x2, 2x7/7x2 y 2x8/8x2 son idénticas a las del 4x2/2x4 pero más grandes. Lo mismo aplica a las habitaciones 3x7/6x7 y 3x8/8x3, que coinciden con la 3x6/6x3 pero más grandes, y para las habitaciones 4x6/6x4, 4x7/7x4 y 4x8/8x4, que usan de igual forma la habitación 4x5/5x4. Para habitaciones que no se han citado, se dejan vacías y ya. En cuanto a los enemigos, la *dungeon* tiene una **probabilidad** asociada de que haya enemigos o un objeto en cada habitación. Si se cumple la probabilidad, hay enemigos; si no, hay un objeto. Mencionar también que el *final boss* se **elige aleatoriamente** entre los disponibles, de la misma forma que también se hace para la habitación 4x5/5x4, la cual es en realidad el escenario de combate de un *mini boss*, el cual se elige también aleatoriamente (aunque estas habitaciones se ven sometidas también a la probabilidad de los enemigos).

3.2.11 Mecánica de la bomba

La **mecánica de la bomba** es implementada de forma sencilla. Básicamente, tenemos un *script* llamado ***Bomb*** que es usado como componente de un *prefab*. Una vez que el jugador pulsa el botón correspondiente se realiza una llamada a *Player::throwBomb*, función que sencillamente se encarga de instanciar una bomba de forma que se inicia el *script* de *Bomb* asociado. Una vez inicializado, se pone en marcha una cuenta atrás y, cuando esta llega a 0, la bomba explota, mostrando un sistema de partículas de explosión y abriendo una ventana de tiempo en la cual un *Collider* se activa para infringir daño a quien pase encima de ella.

Ilustración 193. Modelo de la bomba

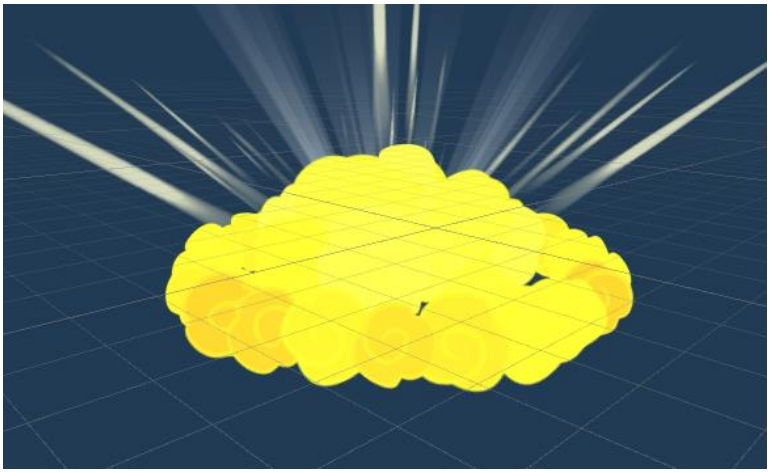


Ilustración 194. Explosión de la bomba

El modelo de la bomba se ha obtenido del paquete *3D Monster Bomb* y la explosión es un sistema de partículas del ya mencionado *Cartoon FX Remaster Free*.

3.2.12 Generación procedural de las islas

Para llevar a cabo la **generación de las islas** en el cielo lo primero que fue necesario es crear las clases correspondientes a la gramática generativa. Tenemos para empezar la clase que abstrae precisamente la gramática, que es ***MissionGrammar***; además, también tenemos la que abstrae los nodos del grafo generado por la gramática, ***GrammarNode***. No obstante, lo primero de todo que hay que tener en cuenta es que nuestra implementación consiste en tener almacenado en disco en formato ***json*** las reglas de la gramática, de forma que sean fácilmente editables.

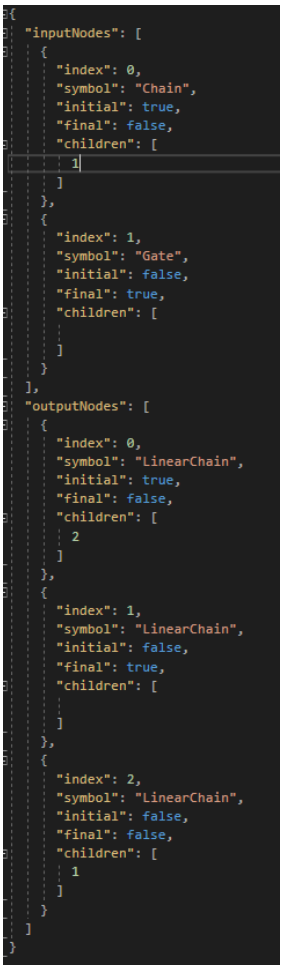


Ilustración 195. Ejemplo de regla en formato json

El formato *json* es muy sencillo. Simplemente tenemos una clase serializable llamada ***Rule*** la cual tiene de atributos nada más que dos vectores de la clase ***RuleNode***, *inputNodes* y *outputNodes*, de forma que estos dos atributos se especifican entre llaves ({}) y, como son vectores, su contenido a su vez se encuentra dentro de corchetes ([]). Una vez dentro de los corchetes, cada uno de los objetos que compone el vector se enumera cada uno dentro de llaves a su vez, donde se especifican los atributos de ese objeto. Sin entrar en mucho detalle, por medio del índice y el vector de hijos sabemos cuál es la estructura del subgrafo que representa la regla, en este ejemplo esta regla de aquí:

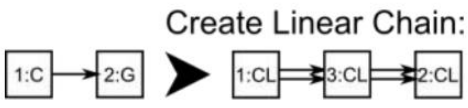


Ilustración 196. Grafo de la regla especificada arriba en json

De esta forma, tenemos el método ***MissionGrammar::loadRules***, el cual sencillamente carga todas las reglas en formato *json* del directorio *[Project Directory]/Resources/GrammarRules*. Una vez cargadas en memoria, *MissionGrammar* crea el grafo de misión aplicándolas nodo a nodo a través de ***MissionGrammar::applyRules***, una función recursiva que va generando los nodos de la forma que se explicó en apartados anteriores.

Cabe destacar que la versión del alfabeto de J. Dormans (2010) es ligeramente distinta a la nuestra ya que, en la primera, los nodos de *boss* y el final son diferentes, mientras que nosotros los hemos agrupado en un único nodo.

Una vez tenemos el grafo generado tenemos que **crear las islas** propiamente dichas. Para ello, contamos con la clase ***Sky***, la cual es la que realiza todo el proceso (y es la que crea también el grafo por medio de *MissionGrammar*). Tras la creación del grafo de misión por medio de la gramática, el siguiente paso era calcular las posiciones de las islas, lo cual se realiza tal y como se ha explicado en el apartado 2.2.2.2 y por medio de la función ***Sky::computeIslands***, que asigna además cada isla con un nodo del grafo de misiones de la forma ya discutida con anterioridad.

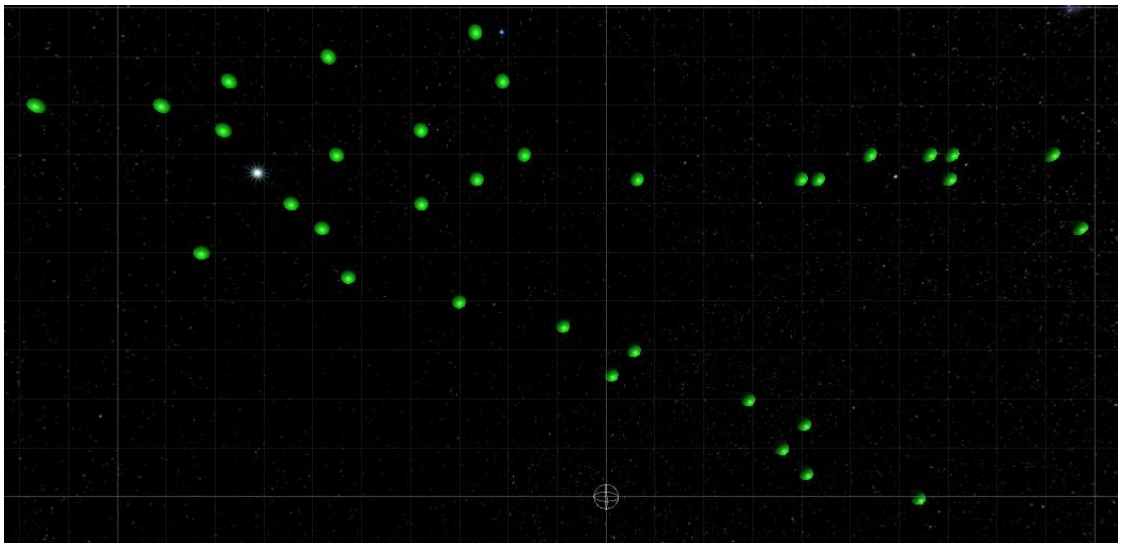


Ilustración 197. Ejemplo real de las posiciones de las islas generadas junto con la circunferencia inicial abajo

La siguiente etapa era la creación propiamente dicha de las islas, para lo cual tenemos la función ***Sky::visualizeIslands*** que, como podemos deducir, se encarga de crear la utillería y las propias islas según el nodo del grafo que tenga asignado.

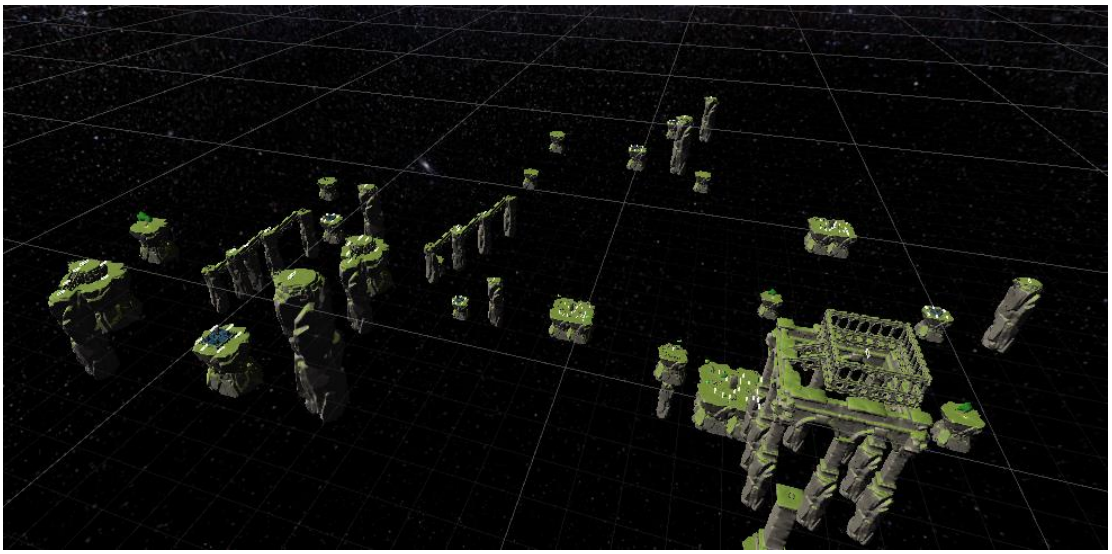


Ilustración 198. Ejemplo real de generación de islas

Es importante mencionar también que las islas en realidad pueden estar formadas por varias y que algunas de ellas pueden actuar de “entrada” en el sentido de que reciben los puentes del padre y otras de “salida”, de manera que de ahí solo salen los puentes entre esa isla y sus hijas. Vamos a hacer un repaso además por las diferentes islas, que se han realizado casi por completo con la utilería proporcionada por el paquete *Pure Nature* antes mencionado, aunque hay ciertos elementos provenientes de *Ancient Ruins and Plants*, como el modelo del teletransporte.

- **Isla del boss.** Se trata de la isla final donde nos enfrentaremos al *boss*. Posee un teletransporte abajo el cual nos lleva a la parte superior donde está el *boss* y el cual recibe los puentes de entrada.

Ilustración 199. Isla del boss

- **Isla inicial.** Es una isla simple en la que aparece el personaje del jugador.

Ilustración 200. Isla inicial

- **Isla de item.** Es la isla en la que aparecerán todos los tipos de *items* indicados por la gramática: *bonus item, key, final key y multi key*. Dependiendo del tipo que sea habrá unos *items* u otros.

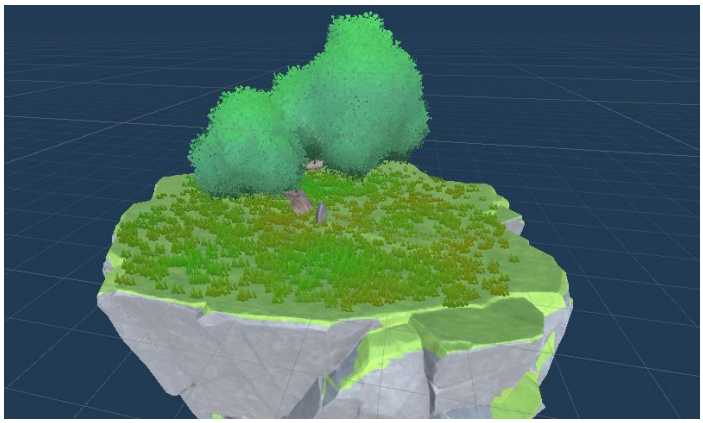


Ilustración 201. Isla de item

- **Isla del mini boss.** Como pasaba en el castillo, y tal y como se puede ver en el alfabeto de la gramática, hay un *mini boss* (a diferencia de los varios que puede haber en el castillo) y esta es su isla, que funciona de manera similar a la del *boss*, con un teletransporte abajo.

Ilustración 202. Isla del mini boss

- **Islas de enemigos.** Por último, nos queda las islas que contienen enemigos. Estas se corresponden con los nodos *nothing, test item, secret test* y *test*. Como podemos ver, en principio, estos nodos estaban hechos para ser distintos y contar con diferentes mecánicas, pero al final optamos por aglutinarlos, de forma que, si una isla tiene uno de estos nodos, se escoge aleatoriamente una de las siguientes islas (salvo para *nothing*, para la cual se escoge siempre la primera).

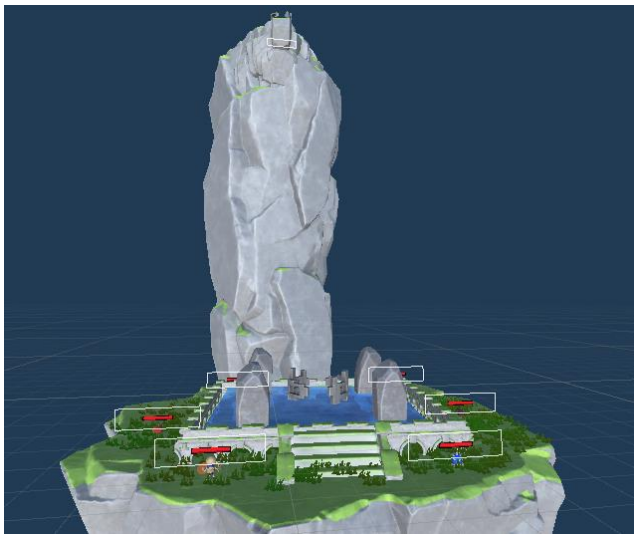


Ilustración 203. Isla de nothing

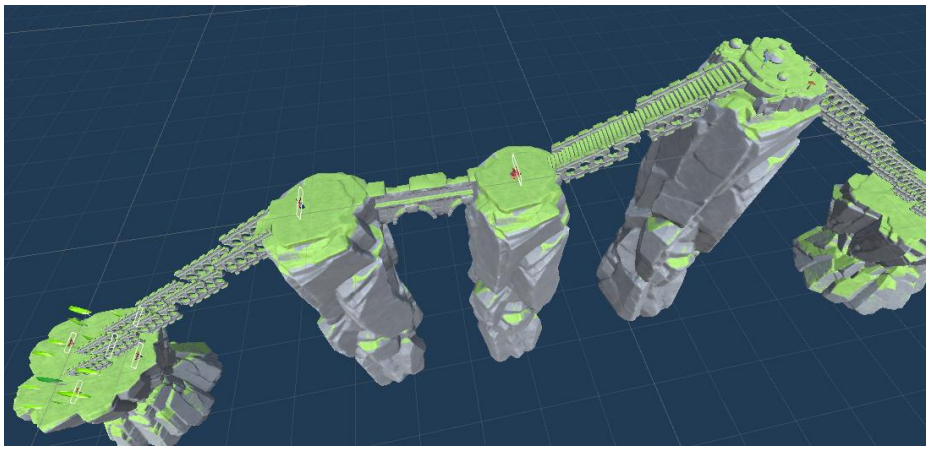


Ilustración 204. Isla de enemigos 1

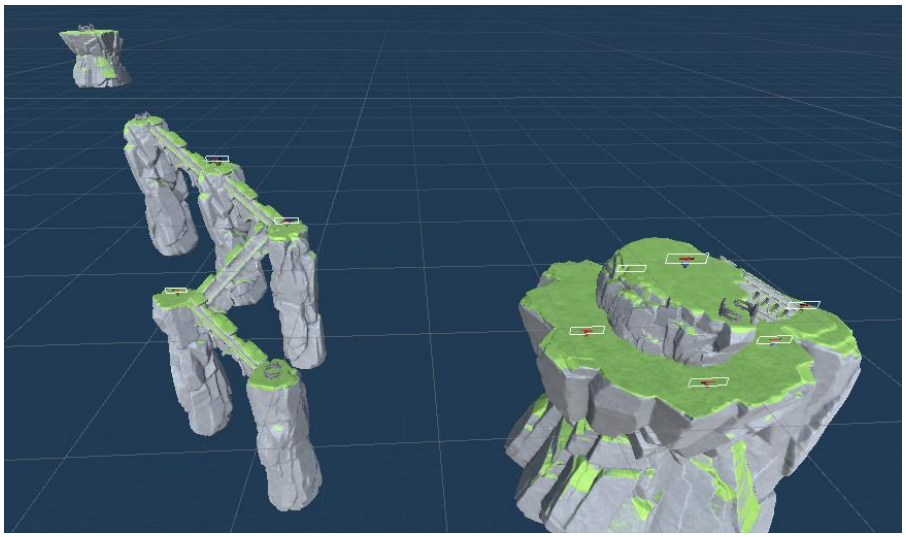


Ilustración 205. Isla de enemigos 2

Ilustración 206. Isla de enemigos 3

Como se puede ver en las imágenes, las islas con enemigos suelen estar formadas en realidad por más de una isla y, por tanto, tienen diferentes entradas y salidas de puentes.

Por último, nos queda conectar las islas a través de puentes, para lo cual tenemos ***Sky::createBridges***. Como ya vimos, los puentes no son tal sino más bien teletransportadores entre islas.

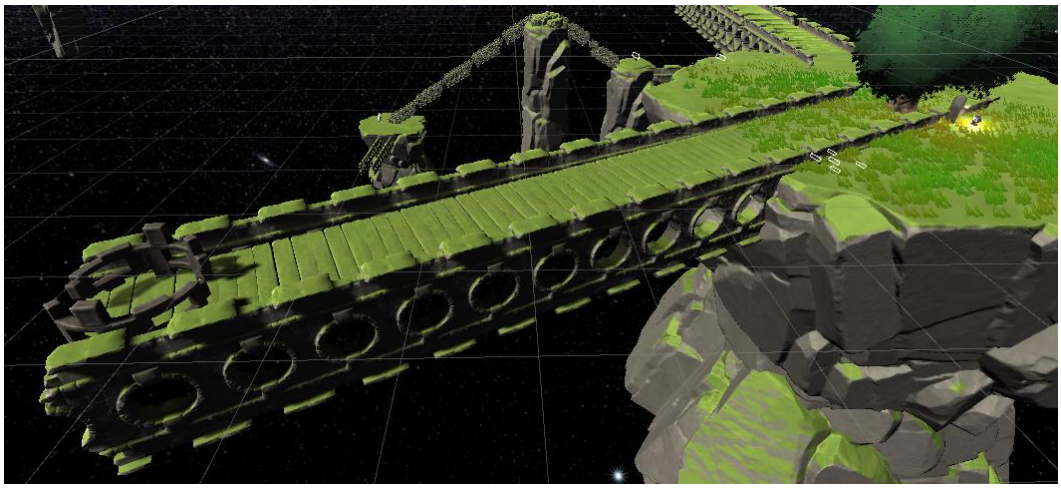


Ilustración 207. Ejemplo de puente generado para una cierta isla

Como podemos observar en esta ilustración, el puente tiene un **teletransportador** al final del mismo. Estos teletransportadores funcionan a través de un *script* llamado ***Teleport***. En esencia, funciona de una forma muy similar al teletransporte de enemigos como el *Wizzrobe* (de hecho, comparten sistema de partículas). Una vez el jugador entra dentro de uno aparece al que está conectado en otra isla, haciendo efectivo el transporte entre islas.

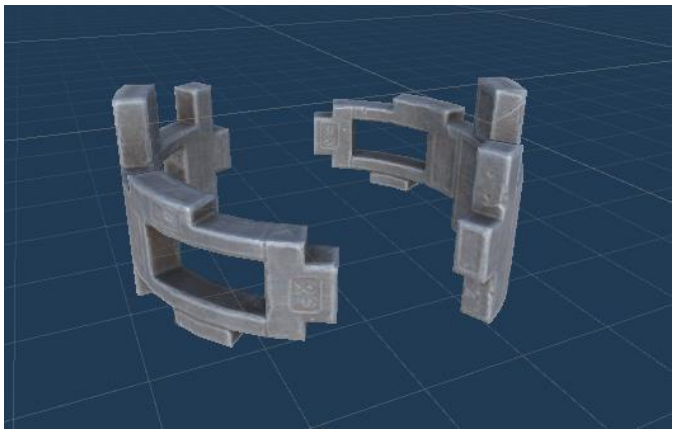


Ilustración 208. Modelo de teletransportador

A diferencia de en la generación del castillo, aquí los enemigos sí están siempre en las islas que les corresponden, pero también se escoge aleatoriamente al *boss* y al *mini boss* de entre los que hay disponibles.

3.2.13 Objetos interactuables

A lo largo del videojuego ya hemos mencionado que el jugador debe encontrarse una serie de **recursos**. Dichos recursos vienen representados por la clase ***Loot***, que simplemente se encarga de estar pendiente por medio de un *OnTriggerEnter* de que el jugador colisione con dicho objeto y, por tanto, lo obtenga. Estos recursos además contarán con un sistema de partículas de brillo para hacerlos notar. Dichos recursos son los siguientes:

- **Flechas.** Como es lógico, las flechas del arco deberían de estar limitadas y que el jugador tenga una cierta cantidad y cada vez que dispare se gaste una. Por tanto, necesita ir encontrando recursos de este tipo a lo largo del juego.

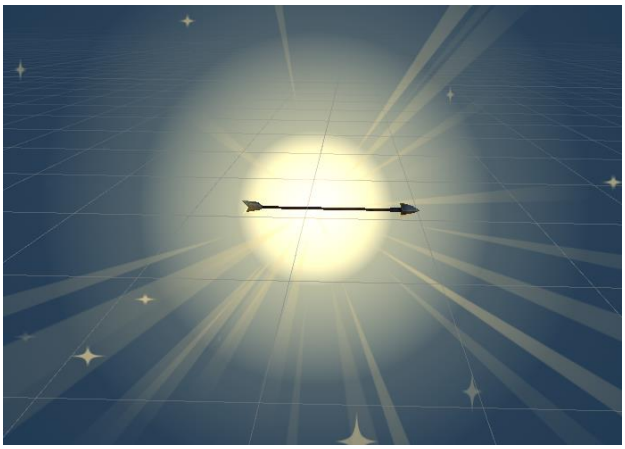


Ilustración 209. Prefab del recurso de flecha

- **Bombas.** Las bombas a su vez serán limitadas también y el jugador deberá de ir consiguiéndolas a lo largo de la partida.

Ilustración 210. Prefab del recurso de bomba

- **Arco.** El jugador no empezará con el arco, sino que se lo tendrá que encontrar a lo largo de la partida.

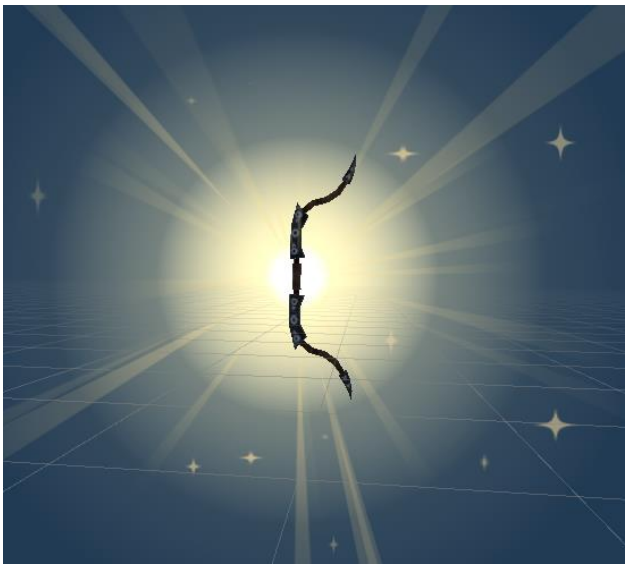


Ilustración 211. Prefab del recurso del arco

- **Vida.** El jugador además tendrá la posibilidad de recuperar vida si se encuentra con este objeto.

Ilustración 212. Prefab del recurso de vida

- **Llave normal.** A lo largo del mapa de las islas flotantes habrá determinadas puertas cerradas tal y como indica el alfabeto de la gramática generativa. Por tanto, también necesitaremos llaves, las cuales vienen también indicadas en dicho alfabeto.

Ilustración 213. Prefab de la llave normal

- **Llave final.** De la misma forma, la gramática prevé la presencia de una puerta antes del *final boss* que debe ser abierta con una llave especial.

Ilustración 214. Prefab de la llave final

El modelo de las llaves se ha obtenido de *Handpainted Keys* y el del corazón de *Simple Gems Ultimate Animated Customizable Pack*.

En cuanto a las propias puertas, estas poseen un *script* llamado ***Door*** que se encarga de abrirlas si el jugador está lo suficientemente cerca de ellas y tiene las suficientes llaves indicadas. Las puertas se abren por medio de una interpolación lineal esférica que se realiza sobre sus bisagras. Están localizadas en ciertos puentes.

Ilustración 215. Prefab de la puerta

3.2.14 Interfaz de usuario

Como ya vimos anteriormente en el apartado 2.2.3.3 donde se proponía unos *sketchs* de cómo debían ser las diferentes interfaces, las cuales son la **interfaz del menú principal**, la del **menú de pausa** y la interfaz principal de juego o ***Head-Up Display*** (HUD), que es como se suele conocer en la industria. Además, se ha usado para la tipografía de ciertas letras *TextMeshPro*, que es uno de los paquetes que ofrece *Unity* para *rendering* de texto avanzado. Aparte, destacar que se ha seguido Brackeys (2020b), Brackeys (2017a), Brackeys (2017b) y Brackeys (2017c) como referencias para la elaboración de la interfaz.

Para el primero de ellos, creamos la clase ***MainMenu***, la cual abstrae el menú y se encarga de estar a la escucha del *input*, de forma que hay determinados elementos de la interfaz que llaman a *callbacks* de esta clase.

Ilustración 216. Interfaz del menú principal

Como podemos ver, existen varios botones y cada uno de ellos tiene una funcionalidad. Los dos primeros nos llevan a otras interfaces, lo cual se hace sencillamente desactivando el *GameObject* que representa esta interfaz y activando el de las otras. ¿Cómo se hace esto? Pues por medio de los *callbacks* mencionados anteriormente. En concreto, usamos *MainMenu::openSelectScenes* para el botón de *start*, *MainMenu::openOptions* para el de *settings* y *MainMenu::quitGame*. Si pulsamos el botón de opciones se nos lleva a la siguiente interfaz:

Ilustración 217. Interfaz del menú de opciones principal

Como podemos ver, en el menú de opciones se nos presentan diferentes parámetros que podemos ajustar. Lo primero de todo es la **resolución** de la ventana

del videojuego, que se nos presenta con un *dropdown*. Si pulsamos sobre ese *dropdown* se nos muestran diferentes opciones dependiendo de las resoluciones que tengamos disponibles en nuestro sistema.

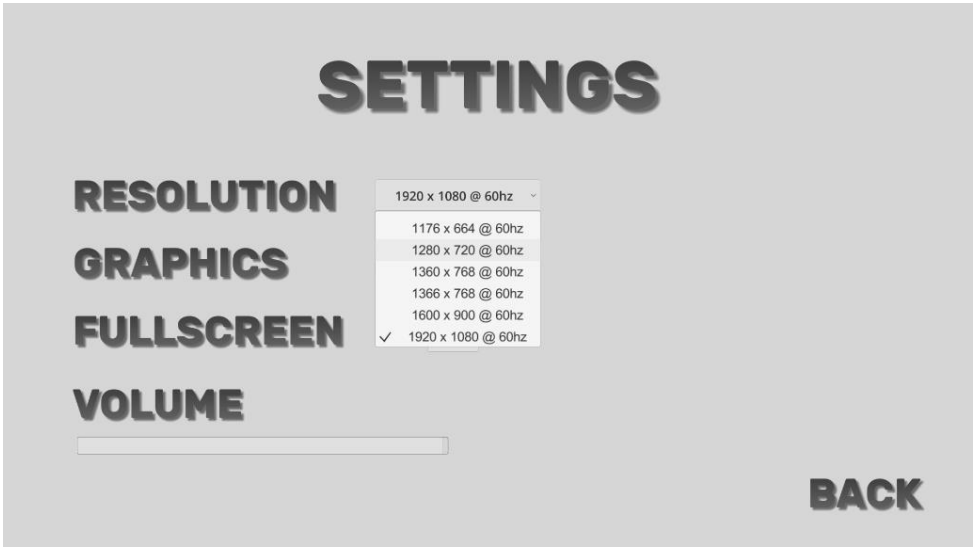


Ilustración 218. Dropdown de resoluciones

Para que el *dropdown* se rellene de las resoluciones disponibles y el resto del funcionamiento de esta interfaz tenemos el *script* ***SettingsMenu***. Al igual que *MainMenu* contiene los *callbacks* propios de cada elemento de la interfaz, de forma que seleccionar una nueva resolución llama a *SettingsMenu::setResolution*. Asimismo, tenemos otro *dropdown*, el de **gráficos**:

Ilustración 219. Dropdown de gráficos

Al elegir un nivel de calidad de los gráficos, este *dropdown* llama a *SettingsMenu::setQuality*, función que efectúa este cambio. Los niveles de calidad gráfica se pueden modificar en *Edit -> Project Settings -> Quality*. Se pueden añadir los niveles que se quieran y además modificar qué atributos gráficos posee cada nivel. En nuestro caso, lo hemos dejado por defecto.

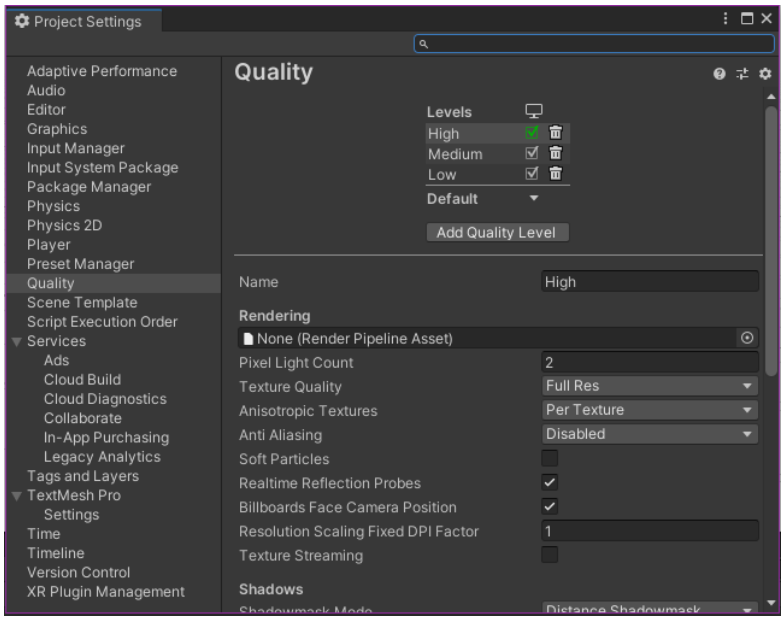


Ilustración 220. Ventana de calidades gráficas del proyecto

El siguiente parámetro que se puede tocar es un *toggle* que nos indica si queremos jugar en **pantalla completa** o en modo ventana. Dicho *toggle* está asociado con el *callback SettingsMenu::setFullscreen*.

Como último parámetro tenemos el **volumen del sonido** de nuestro juego. Para ello, necesitamos hacer uso de un *asset* de *Unity* llamado *Audio Mixer*, de manera que creamos uno llamado *MainMixer* el cual está a cargo del volumen maestro del videojuego. Lo que hacemos, por tanto, en el *callback SettingsMenu::setVolume* es establecer el volumen maestro del juego a través de una referencia a este *Audio Mixer*.

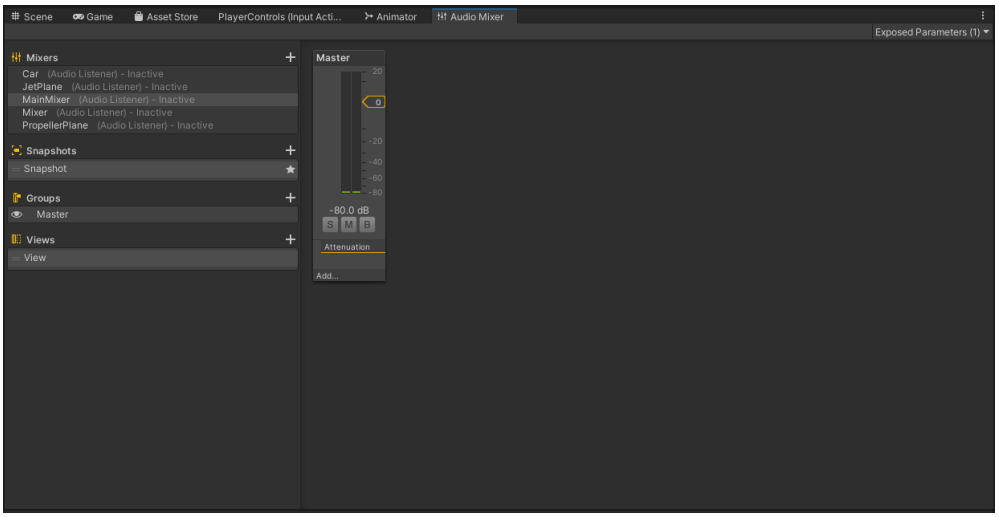


Ilustración 221. Audio Mixer principal del juego

Como último elemento de la interfaz del menú de opciones, tenemos el botón de *back* que sencillamente nos devuelve a la pantalla anterior. Volviendo al menú principal, de una forma similar, el botón *quit* simplemente nos saca de la aplicación. El botón *start*, como hemos mencionado antes, llama al *callback* *MainMenu::openSelectScenes* que hace que se nos muestre la siguiente interfaz:

Ilustración 222. Interfaz para seleccionar el modo de juego

Un botón nos carga uno de los **dos escenarios posibles**, el cielo con las islas flotantes, y el otro nos carga la mazmorra. Para ello, vamos a usar sencillamente *SceneManager.LoadScene* y eso nos llevará directamente a las escenas correspondientes (cabe destacar que en nuestro proyecto hay 4 escenas disponibles,

la del menú principal que es en la que nos encontramos, la de prueba que esa no está incluida en el juego final y se ha usado únicamente como “campo de pruebas”, la del cielo y la de la mazmorra). Para ocultar los tiempos de carga y evitar que el jugador se desespere (aunque no son en realidad muy elevados) vamos a usar una **pantalla de carga**, de forma que antes de cargar la escena cambiamos a una pantalla de carga y, cuando finalmente se ha cargado la escena, se quita.

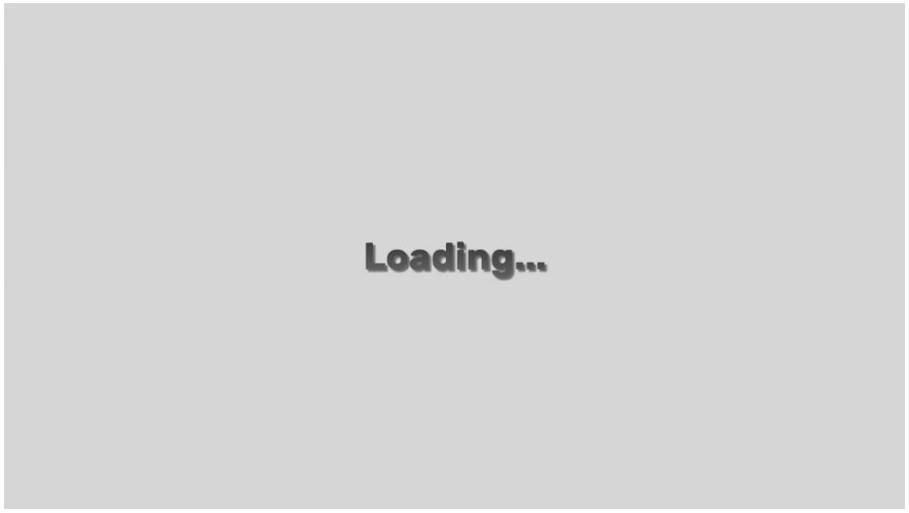


Ilustración 223. Pantalla de carga

Por otra parte, tenemos el HUD del juego, que sirve para **informar** al jugador de su estado a nivel de vida y recursos. Para el caso de la vida de su personaje cuenta con una barra de vida la cual se actualiza por medio del *script HealthBar*, el cual se comunica con *Player* que es el que lleva la cuenta de la vida que le queda al jugador. La barra de vida no es más que un *slider* cuyo valor máximo es el máximo de vida que puede tener el jugador y el mínimo 0. Cuenta además con un gradiente de color para hacer que, mientras más vida tenga el personaje, más cercano al verde sea el contenido de la barra de vida y, mientras menos vida, más cercano al rojo. Los *assets* de la barra de vida se han conseguido de Brackeys (2020b).

Ilustración 224. HUD del videojuego

Como podemos ver, además contamos con una serie de **indicadores** de la cantidad de recursos de los que dispone el jugador, cada uno con su propio **icono** que actúa como metáfora visual, los cuales se han conseguido de https://game-icons.net. Estos indicadores van a su vez actualizándose a través de *Player*, el cual es el encargado en este caso de actualizar tanto el conteo interno como la interfaz. La barra de los enemigos a su vez funciona de una forma idéntica a la de *Player*, salvo por el hecho de que se encuentra dentro de la geometría del propio juego, pero fuera de la narrativa; esto es, son **elementos espaciales**, mientras que la barra de vida y el inventario son **elementos no diegéticos** porque se encuentran fuera de la narrativa y de la geometría del videojuego. Lo que queremos lograr con las barras de vida de los enemigos es que se queden sobre sus cabezas, que se muevan junto a ellos y que siempre estén mirando hacia la cámara para que el jugador pueda verlos. Para ello, tenemos el *script* ***Billboard***, que sencillamente se encarga de girar estas barras de vida para que estén permanentemente mirando hacia cámara.

Ilustración 225. Barra de vida de un enemigo

La única parte que nos queda por discutir de la interfaz es el **menú de pausa**. Este menú es muy sencillo y simplemente consiste de los botones de *resume*, que reanuda el juego, *options*, que muestra un menú para ajustar parámetros durante el juego, y el botón de *quit*, que nos devuelve al menú principal. Cabe destacar además que el juego pausado se puede ver al fondo, pero con una capa oscura transparente para que los botones puedan verse adecuadamente.

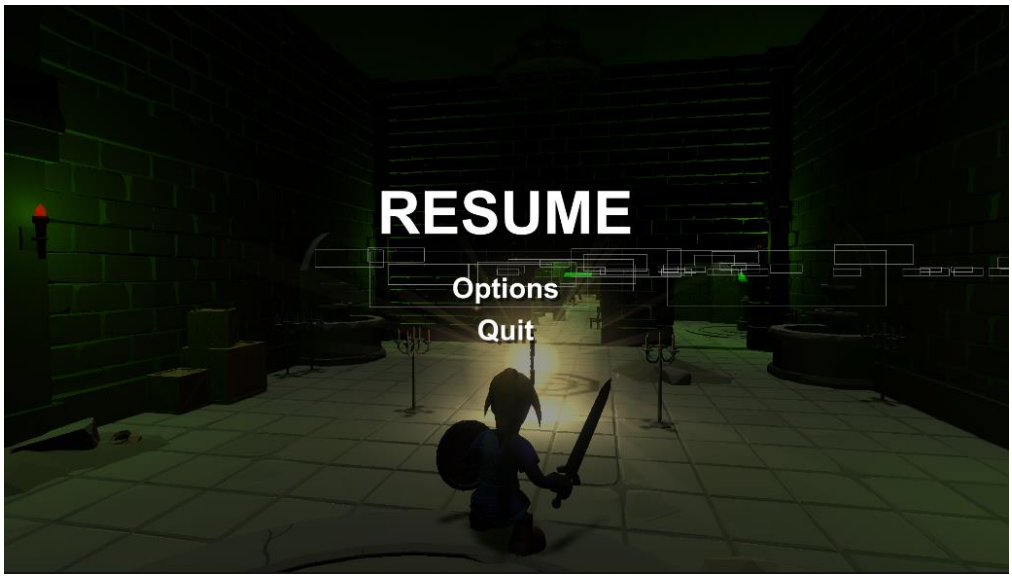


Ilustración 226. Menú de pausa

Este menú de pausa está abstraído por la clase ***GameMenu***, la cual es la responsable, entre otras cosas, de pausar el propio juego y enseñarnos este menú llamando al *callback GameMenu::changePauseState*, que cambia el estado del juego

de pausado a reanudado dependiendo de cómo estuviera antes. Pausar el juego es simple y requiere solamente cambiar el atributo *Time.timeScale* a 0 y a 1 para reanudar el juego. Por otra parte, si pulsamos el botón de opciones nos saltará la siguiente interfaz:

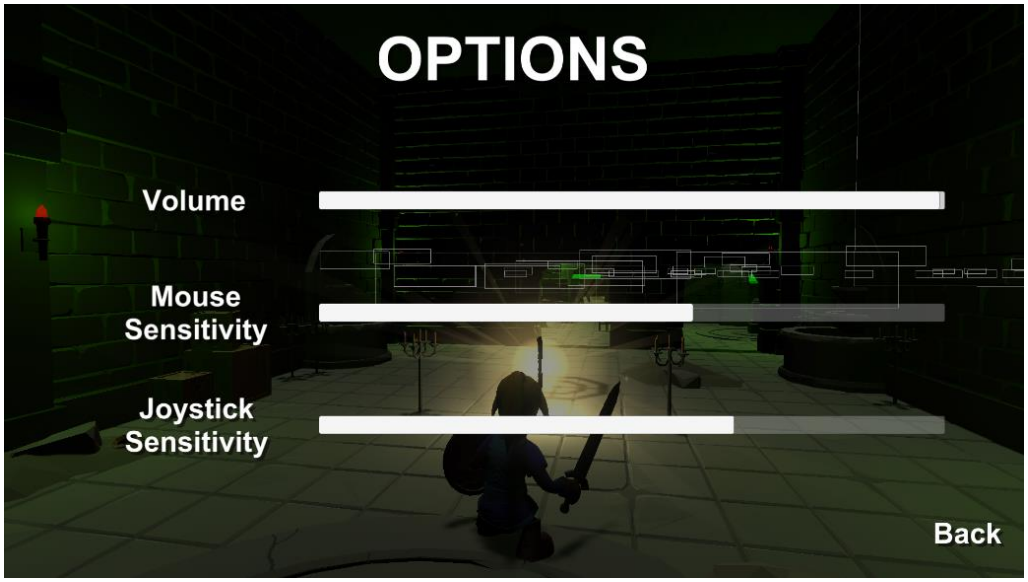


Ilustración 227. Menú de opciones del menú de pausa

Como en el menú de opciones principal podemos ajustar el volumen del juego, lo cual se hace de una forma idéntica por medio del *callback GameMenu::setVolume*. Para cambiar las sensibilidades de ratón y mando tenemos una referencia a *FreeLookCam* y cambiamos dichas sensibilidades a través de los *callbacks GameMenu::setGamepadSensitivity* y *GameMenu::setMouseSensitivity*. El botón *back* nos devuelve al menú de pausa.

En realidad, quedan otras dos interfaces más, pero esas se verán en el apartado siguiente.

3.2.15 Condiciones de victoria y derrota

Este apartado básicamente consiste en las formas que tiene el jugador de **perder** y **ganar** el juego, las cuales son sencillas: el jugador lo pierde si su personaje se queda sin vida y muere, con lo cual pierde todo el progreso y tiene que empezar de 0, y gana si acaba con el *final boss*, el cual suelta al morir una **corona** y, al conseguirla, el jugador gana.

Para la muerte del personaje del jugador contamos con el método ***Player::die***, el cual es llamado cuando el personaje pierde toda su vida. No obstante, no muere inmediatamente por así decirlo, sino que se activa la animación de morir y esta a su vez llama como evento de la animación a ***Player::dead***, que es el método encargado de enseñar la interfaz de fin del juego que llama a su vez a *GameMenu::loadDeathScreen*. Esta es una interfaz muy sencilla que pausa el juego y únicamente permite al jugador volver al menú principal.

Ilustración 228. Interfaz de game over

Por otra parte, tenemos la situación en la que el jugador gana el juego, para lo cual tenemos el método ***Player::win*** que sencillamente pausa el juego y activa la interfaz de victoria llamando a *GameMenu::loadWinScreen*. Para que se llame a esta función, el jugador deberá haber recogido el siguiente *item*, que funciona de forma idéntica a los recursos antes citados:

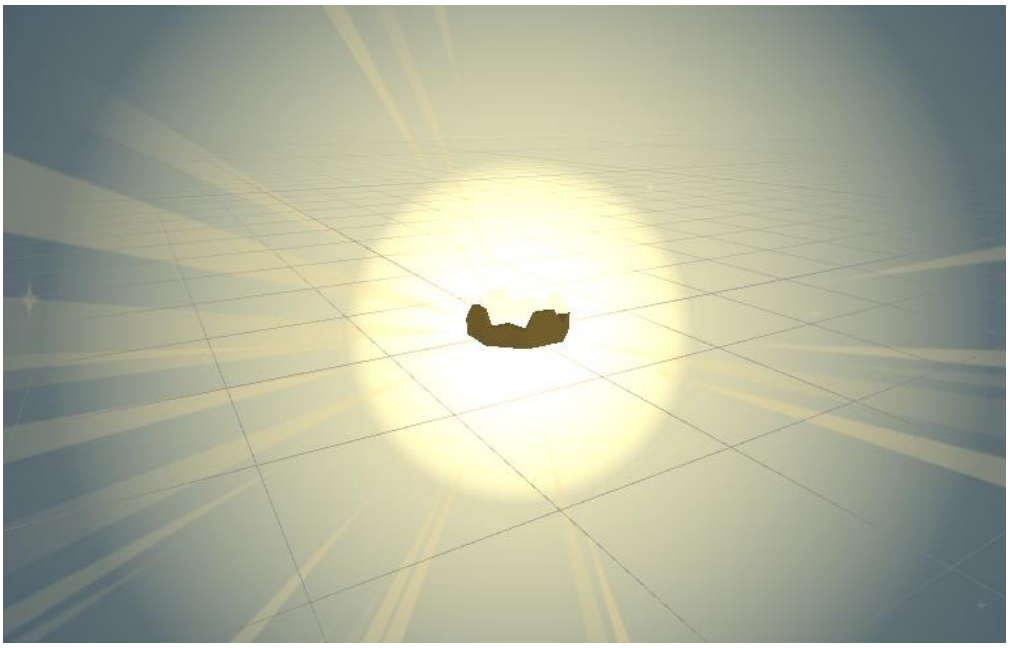


Ilustración 229. Corona que se necesita para ganar el juego

Ilustración 230. Interfaz de victoria

3.2.16 Efectos audiovisuales

Pese a que a lo largo del desarrollo ya se han discutido varios efectos audiovisuales, en este apartado vamos a comentar algunos de ellos que se añadieron hacia el final del proyecto. Para empezar, se añadieron **pistas de música** y **sonidos** para enriquecer la experiencia del usuario, lo cual se hizo por medio de un *asset* de *Unity* llamado ***Audio Source***, los cuales sencillamente se encargan de reproducir algún sonido o música. Por otra parte, tenemos los ***Audio Listeners***, los cuales, como

su nombre indica, escuchan los sonidos, de forma que mientras más alejado esté de la fuente, más atenuado estará el sonido, como en el mundo real. Los *Audio Listeners* se encuentran exclusivamente en las cámaras. Vamos a ver qué pistas de sonido se añadieron:

1. **Música del menú principal**. Como se puede ver, está en *loop* y se reproduce en *Awake*, lo cual quiere decir que está en bucle y se reproduce desde que se carga esta escena. Procedente del paquete *Action RPG Music Free*.

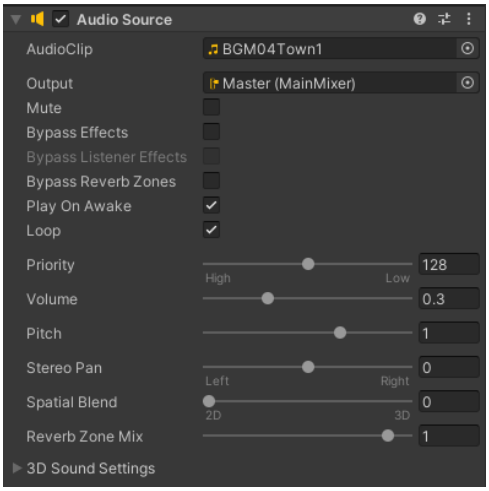


Ilustración 231. Audio Source de la música del menú principal, ubicado en la propia cámara

2. **Música de ambiente** de cada escenario, la cual varía según si estamos en el escenario del cielo o del castillo. También está en bucle y se reproducen desde el principio, y también se encuentran como componentes de la cámara. Procedente del mismo paquete que el anterior.

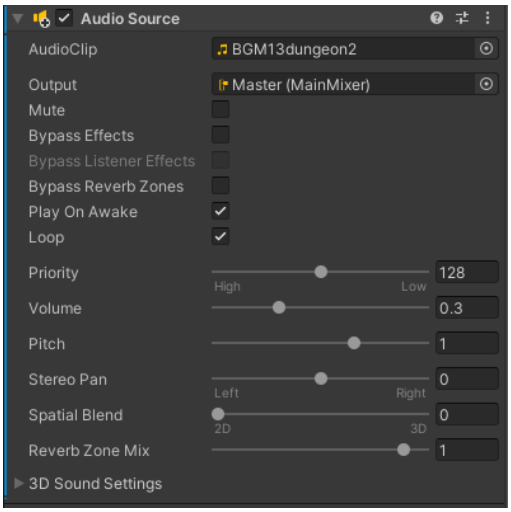


Ilustración 232. Audio Source de la música de ambiente del mapa del cielo

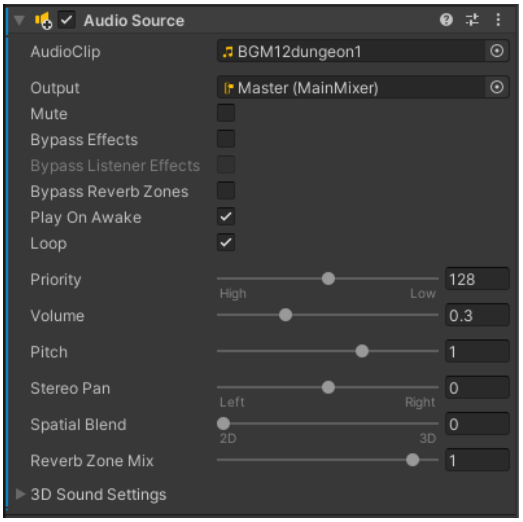


Ilustración 233. Audio Source de la música de ambiente del mapa de la mazmorra

3. **Sonido del personaje sufriendo daño**, el cual se reproduce cada vez que el personaje es dañado por un enemigo. Ya no se reproduce ni al principio ni en bucle, únicamente cuando se le dice que se reproduzca a través de *Player::takeDamage*. Se encuentra como componente del *GameObject* de jugador. Procedente del paquete *RPG Essentials Sound Effects*.

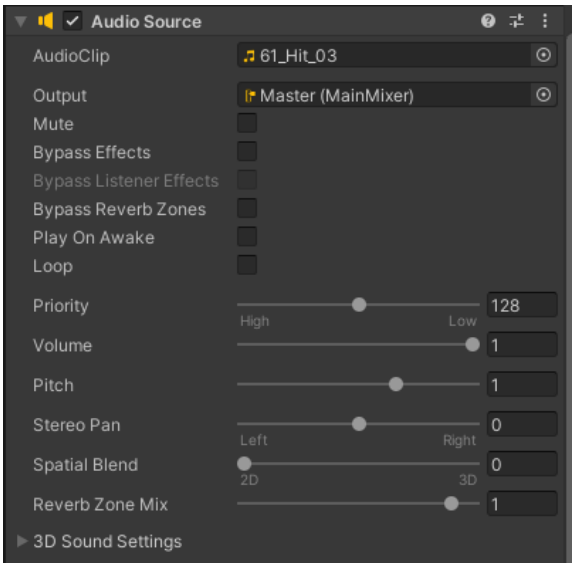


Ilustración 234. Audio Source del jugador sufriendo daño

4. **Sonido del enemigo sufriendo daño**, el cual funciona de una forma idéntica al anterior y del mismo paquete.

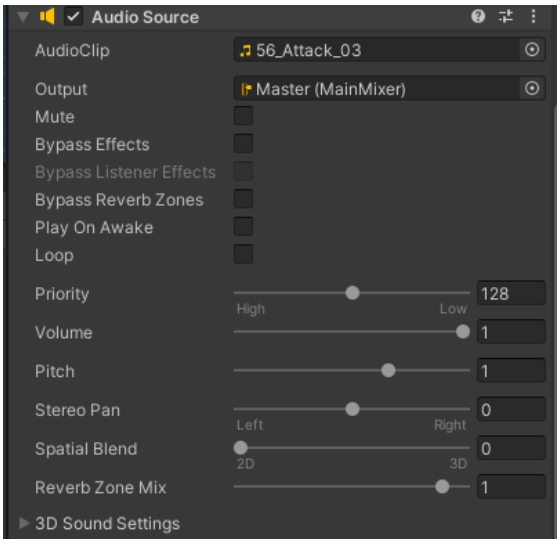


Ilustración 235. Audio Source del enemigo sufriendo daño

5. **Sonido del jugador/enemigo bloqueando** con el escudo. De nuevo, muy similar a los anteriores y del mismo paquete, solo que ahora está como componente en las jerarquías de los escudos.

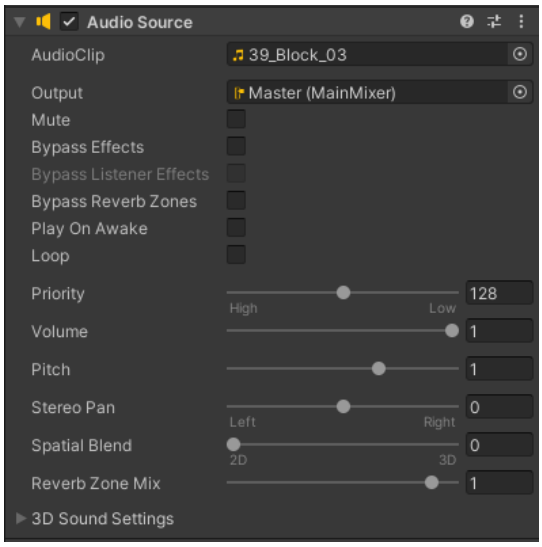


Ilustración 236. Audio Source del bloqueo del escudo

6. **Sonido de la muerte del jugador**, que sirve para indicar que ha muerto. Se encuentra también en la jerarquía del objeto de jugador. Procedente de *Fantasy Arcade RPG Freedom World Sounds*.

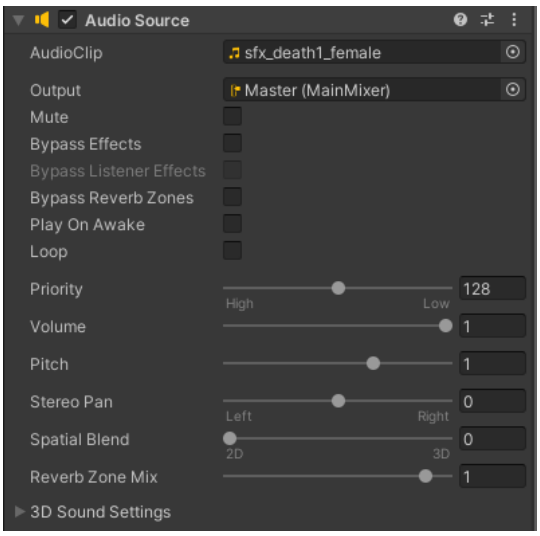


Ilustración 237. Audio Source del sonido de muerte del jugador

7. **Sonidos de ciertas acciones de los enemigos**. Como ya hemos visto, ciertos enemigos son capaces de ataques a distancia o de teletransportarse y estos tienen también sonidos asociados. Por ello, estos sonidos están asociados con los *scripts* de tipo *[Nombre del monstruo]Behaviour*, ya que se les hace reproducirse con las mismas funciones con las que se ataca. En general, provienen del paquete ya mencionado *RPG Essentials Sound Effects*.

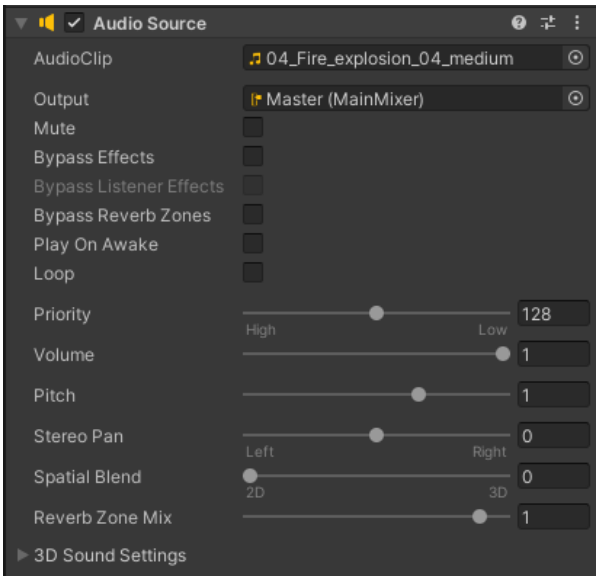


Ilustración 238. Audio Source de los ataques de fuego

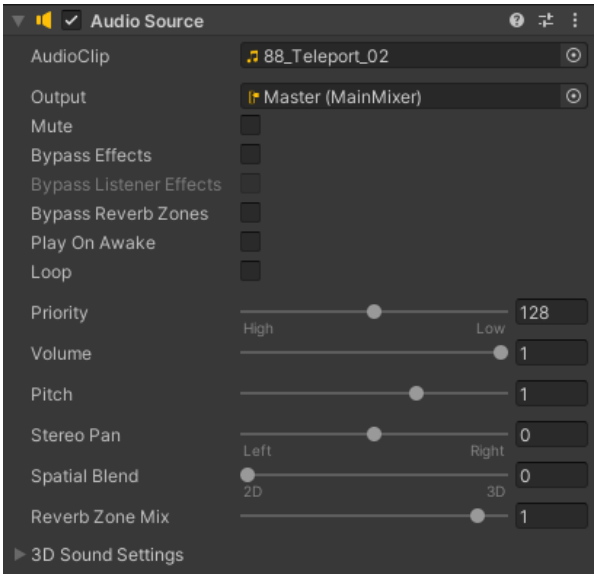


Ilustración 239. Audio Source para los sonidos de teletransporte tanto enemigo como aliado

8. **Sonidos de la bomba**. A la bomba se le han añadido los sonidos tanto de la explosión como del *tictac* provenientes del paquete *Grenade Sound FX*.

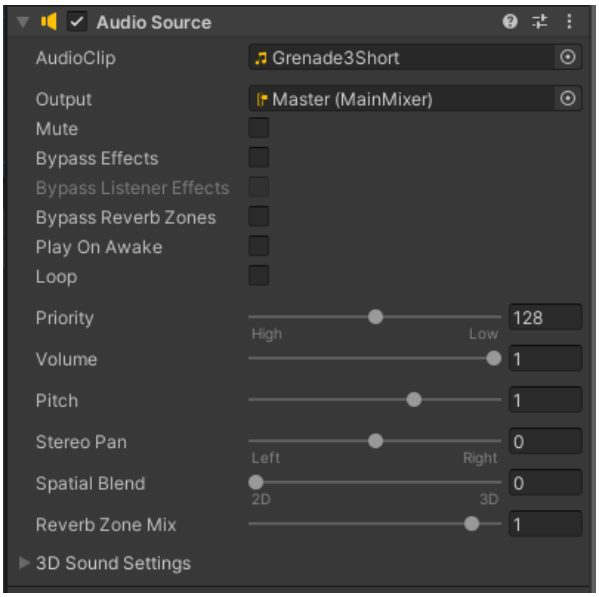


Ilustración 240. Audio Source del sonido de la explosión de la bomba

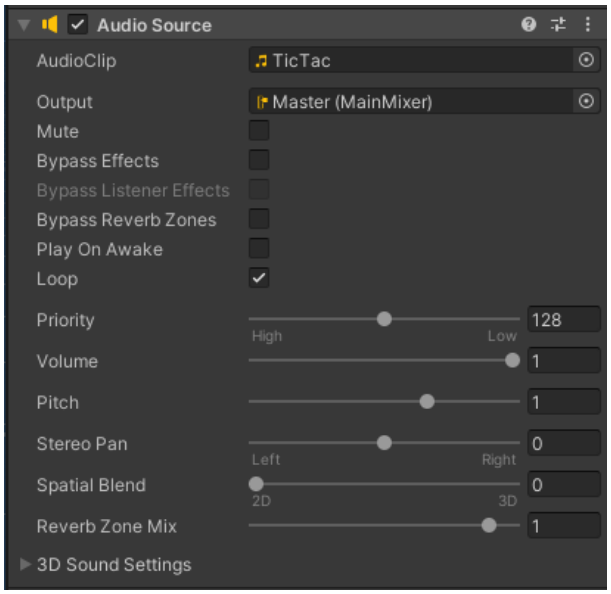


Ilustración 241. Audio Source del sonido del tic tac

9. **Música del *final boss***. El *boss* final tendrá también una música única asociada a su enfrentamiento, la cual está cogida del paquete antes mencionado *Action RPG Music Free*.

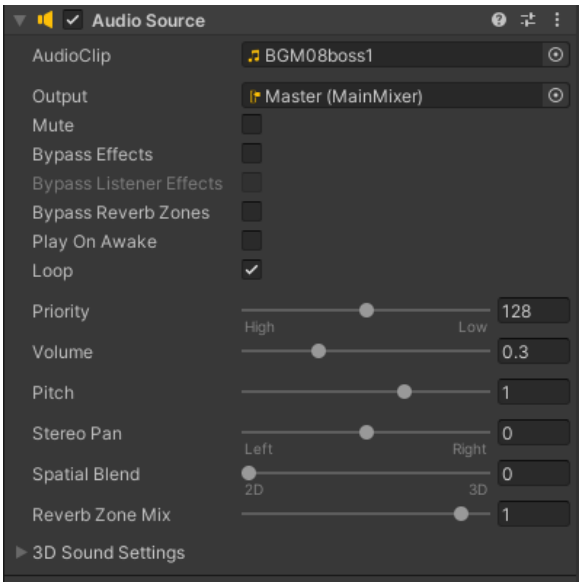


Ilustración 242. Audio Source del sonido del final boss

10. **Música de victoria**. Finalmente, al ganar el juego sonará también una melodía de victoria, sacada del mismo paquete que el anterior sonido. Este *Audio Source* está en el *prefab* de la corona.

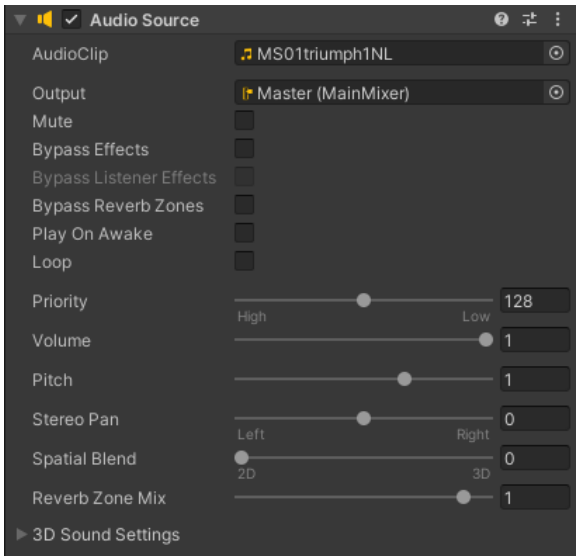


Ilustración 243. Audio Source de la música de victoria

Queda otro efecto audiovisual a destacar, que es el de **indicar que un enemigo o un personaje ha sufrido daño visualmente**. Para ello, vamos a usar una convención extendida en los RPG de acción que es pintar al personaje dañado de rojo durante un momento. Para ello, vamos a contar con unas corrutinas llamadas *Player::damageMaterialCoroutine* (para el daño del jugador) y *Enemy::damageMaterialCoroutine* (para cuando sea el enemigo el dañado). La idea

es cambiar el material de la(s) malla(s) del personaje/enemigo a uno rojizo en *Player::takeDamage* (o en su defecto *Enemy::takeDamage*) durante un período muy corto de tiempo, de manera que hacemos que la corrutina espere ese tiempo para después volverlo a cambiar.

```
// We change the player mesh materials with the damage material and launch a coroutine.
for (int i = 0; i < meshes.Length; i++)
{
    meshes[i].material = damageMaterial;
}

StartCoroutine(damageMaterialCoroutine());
```

Ilustración 244. Parte del código de Player::takeDamage

```
/// <summary>
/// Coroutine that is used to maintain the damage material during a short period of
/// time until the player gets back their original materials.
/// </summary>
/// <returns></returns>
1 referencia
IEnumerator damageMaterialCoroutine()
{
    yield return new WaitForSeconds(0.15f);

    for(int i = 0; i < meshes.Length; i++)
    {
        meshes[i].material = originalMaterials[i];
    }
}
```

Ilustración 245. Corrutina de cambio de material de Player

3.3 Pruebas finales

Al tratarse este proyecto del desarrollo de un videojuego, las pruebas que se han realizado están orientadas a **jugar nosotros mismos** algunas partidas para comprobar que no existe ningún *bug* grave ni nada por el estilo y que la experiencia de juego es satisfactoria. En concreto, se han comprobado las siguientes cosas:

- Que el personaje del jugador responda correctamente y se mueva de acorde al *input*.
- Que el personaje del jugador ataque correctamente con la espada y de acuerdo al *input*, de forma que sea capaz de dañar a los enemigos. Ídem para el arco y las bombas.

- Que el manejo de los recursos se lleve adecuadamente, de forma que el jugador gaste flechas cuando dispara y bombas cuando las lanza.
- Que los enemigos sean capaces de atacar al jugador, de inflingirle daño y de actuar acorde al algoritmo de inteligencia artificial implementado.
- Que el jugador sea capaz de completar una partida, es decir, de recorrer de principio a fin el mapa para poder ganar.
- Que el jugador pueda fijar a los enemigos sin problemas.
- Que se capaz de recoger objetos del suelo.
- Que sea capaz de abrir las puertas.
- Que los enemigos suelten objetos al morir adecuadamente.
- Que los mapas se generen de una forma coherente de forma que no haya habitaciones inaccesibles ni nada así.
- Que, en general, la jugabilidad no resulte frustrante.

Existen determinados *bugs*, más visuales que otra cosa, que no se han podido solucionar a lo largo del desarrollo: enemigos atravesando parcialmente objetos, el jugador montándose encima de los enemigos… Hay que entender que esto no es un gran proyecto y no poseo la pericia técnica para ajustar adecuadamente este tipo de cosas, más aún cuando muchas veces son consecuencia de los *assets* usados o siquiera los grandes estudios de videojuegos son capaces de arreglarlos. No obstante, no consideramos que estos problemas rompan la experiencia de juego.

3.3.1 Validación de la usabilidad del sistema

Para que un software sea usable debe cumplir que sea **fácil de aprender** y **fácil de utilizar**, esto es, que permita realizar las tareas rápidamente y sin errores y que el usuario use la herramienta de la manera más natural posible, de manera que permita al usuario centrarse en su tarea y no en la aplicación. En concreto, vamos a explorar los **principios generales de la usabilidad** y ver si nuestra aplicación los cumple:

1. **Facilidad de aprendizaje.** El tiempo requerido desde el desconocimiento de una aplicación hasta su uso productivo debe ser mínimo. Para ello, la

aplicación debe ser **sintetizable**, es decir, que cuando se produce un cambio en el sistema el usuario debe ser capaz de captarlo, y **familiar**, es decir, que debe existir una correlación entre los conocimientos que el usuario ya posee y los requeridos para usar el software. Como hemos ido viendo, nuestra aplicación es sintetizable porque existen diversos recursos para que el jugador se percate de ciertos cambios (el color rojo al sufrir/ejercer daño, ayudas sonoras para cuando hace daño a un enemigo, para cuando un enemigo hace un cierto ataque, las propias animaciones…), y es familiar porque, entre otras cosas, sigue muchas convenciones del género de los RPG de acción y otros videojuegos en cuanto a su esquema de controles (el ataque se hace con el click izquierdo del ratón/botón oeste del mando, el jugador se mueve con un esquema WASD/*joystick*…) y el de su interfaz (barra de vida, menú principal, pantallas de carga, menú de opciones, etc…).

2. **Flexibilidad.** Es la multiplicidad de maneras en que el usuario y el sistema pueden intercambiar información. Entre otras cosas, que un sistema sea flexible requiere dotar al usuario de **control** sobre el mismo. En nuestro software, esto se traduce en permitirle ajustar ciertos parámetros a través de los menús de opciones como el volumen del sonido, la resolución de pantalla, el nivel de los gráficos o la sensibilidad de la cámara, lo cual permite al sistema adecuarse a las necesidades del usuario (por ejemplo, a su equipo, que puede ser mejor o peor) y no al revés.
3. **Consistencia.** Un sistema es consistente si todos los mecanismos que se utilizan son siempre usados de la misma manera, siempre que se utilicen y sea cual sea el momento en el que se haga. Puede resultar evidente, pero a veces no lo es tanto, y en nuestra aplicación hemos intentado que así sea, haciendo que la interfaz guarde en general un cierto estilo compartido, por ejemplo.
4. **Robustez.** El sistema debe permitir al usuario **conseguir sus objetivos** sin problemas. Como hemos visto anteriormente, el jugador es capaz de ganar la partida sin problemas; además, también es capaz de navegar entre los menús, atacar a los enemigos, etc…

5. **Recuperabilidad.** El sistema debe permitir al usuario **corregir una acción** una vez que esta ha sido reconocida como errónea. Este principio no se aplica tanto en principio a nuestra aplicación por su naturaleza, aunque se podría decir que la cumple al darle la posibilidad al jugador de retroceder al menú anterior una vez se ha metido en uno en el que no quería estar a través de botones de *back*, o cuando el jugador, mientras está jugando, es capaz de interrumpir un ataque rodando, por ejemplo. No obstante, hay algo curioso a tener en cuenta, y es que en un videojuego a veces queremos **castigar** al jugador de manera que no es capaz de corregir una acción; es lógico, el juego debe tener un factor de riesgo para interesar al jugador. De esta forma, el jugador no es capaz de corregir, por ejemplo, haber recibido un ataque del enemigo y haber perdido vida.
6. **Tiempo de respuesta.** Es el tiempo que necesita el sistema para expresar los cambios de estado al usuario, los cuales deben ser soportables. Esto es esencial en un videojuego: el personaje del jugador debe poder reaccionar inmediatamente al *input* que se le suministre, lo cual es fácilmente comprobable que nuestro sistema cumple y para ello se han implementado cosas como la memoria de los ataques. Por otra parte, esto también se aplica a los **tiempos de carga** del sistema. Como hemos visto, los ocultamos con una pantalla de carga para evitar desesperar al jugador y, además, nos hemos asegurado de que no sean prohibitivamente largos.
7. **Adecuación de las tareas.** El sistema debe permitir todas las tareas que el usuario quiere hacer y en la forma en que las quiere hacer. Evidentemente, esto es algo que se cumple, pues el usuario es capaz de ejecutar las acciones que quiere que haga su personaje, por ejemplo; ahora bien, esto está, como es lógico, limitado por la naturaleza de las mecánicas y, como hemos visto, no siempre queremos que el jugador sea capaz de hacer todo cuanto desea ya que en los videojuegos el castigo es importante.
8. **Disminución de la carga cognitiva.** Este principio está asociado con la **complejidad de las interfaces**: debe favorecerse en los usuarios el reconocimiento mediante el recuerdo, de manera que no tengan que recordar abreviaturas o códigos complicados. En otras palabras, las

interfaces deben ser **minimalistas**. Esto es algo que se puede ver a simple vista que la interfaz de nuestro juego cumple.

4 CONCLUSIONES Y TRABAJOS FUTUROS

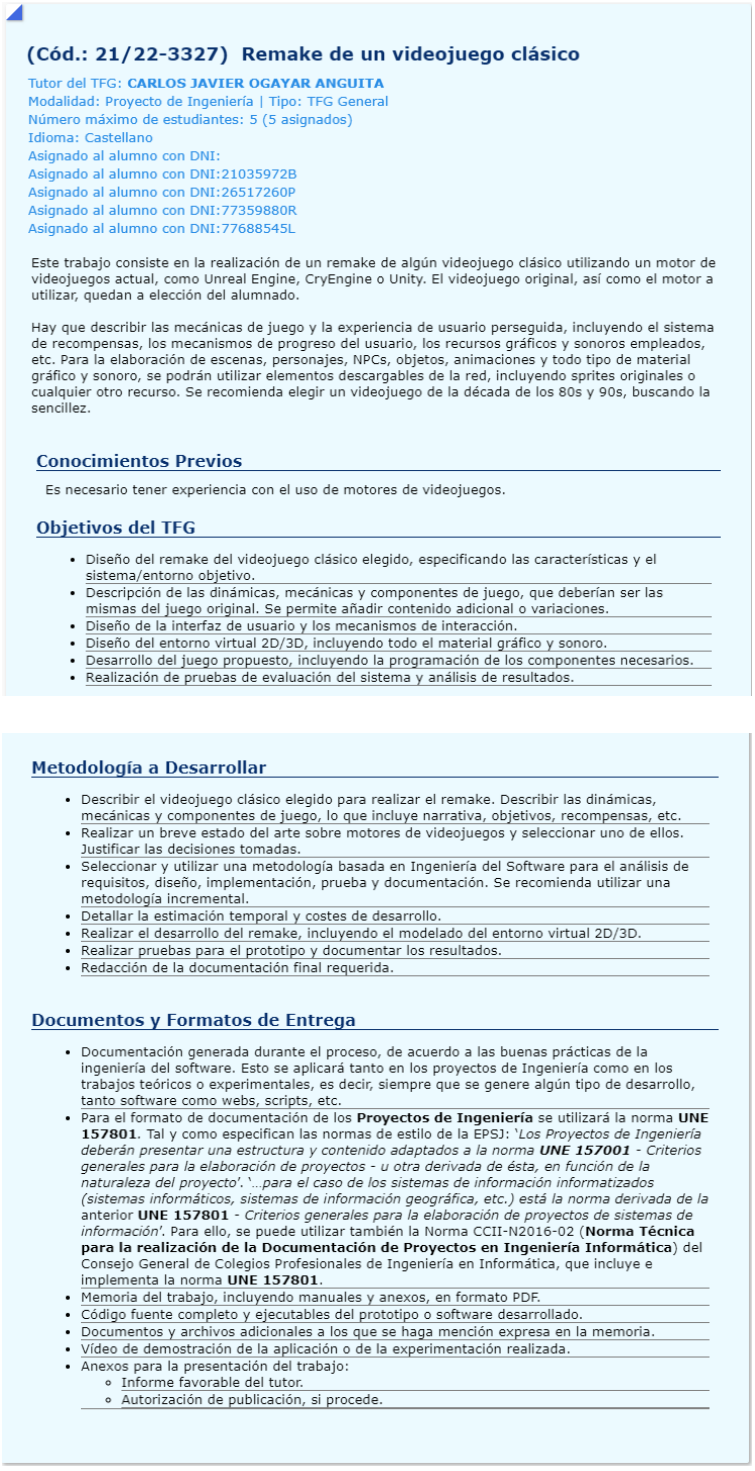
Tras haber finalizado el proyecto creemos que se han **cumplido los objetivos propuestos** con creces: usar no uno sino varios algoritmos de generación procedural e implementar las mecánicas típicas de un RPG de acción con un gran énfasis en la experiencia de usuario, haciendo uso además de interfaces persona-ordenador, todo ello empleando una gama muy variada de los conocimientos que se han adquirido a lo largo de la carrera tales como conocimientos en algoritmos geométricos, búsquedas de caminos óptimos en grafos, gramáticas generativas, técnicas avanzadas de programación, informática gráfica, técnicas de animación, desarrollo de videojuegos usando *Unity*, ingeniería del software y gestión de proyectos, entre otros.

No obstante, creemos también que el proyecto **se podría expandir** de diversas formas. Para empezar, se echa en falta una mayor cantidad de *assets* para cosas como por ejemplo los *props* y enemigos del juego, así como un diseño más cuidado en la progresión de los niveles, una mayor variedad y personalización en las armas y armaduras que puede usar el jugador al estilo de lo que suelen ofrecer los RPG, un sistema de subida de niveles, un inventario más complejo, más tipos de objetos, interfaces algo más atractivas y con más opciones, gráficos de mayor calidad, más niveles diferentes a los dos que se han propuesto o, incluso, añadirle narrativa con algún sistema de diálogos y NPCs que no sean enemigos. No obstante, creemos que muchas de estas expansiones quedan fuera del ámbito de la ingeniería informática y se adentra más en el del trabajo artístico y, por tanto, son de menor interés para nosotros, por no hablar de que requerirían de una cantidad de recursos y tiempo de la que estamos muy lejos de disponer.

5 APÉNDICES

5.1 Guía original del Trabajo Fin de Título

Este proyecto ha sufrido una serie de modificaciones a lo largo del tiempo. Primero, se trataba de un trabajo genérico:



(Cód.: 21/22-3327) Remake de un videojuego clásico

Tutor del TFG: **CARLOS JAVIER OGAYAR ANGUITA**
Modalidad: Proyecto de Ingeniería | Tipo: TFG General
Número máximo de estudiantes: 5 (5 asignados)
Idioma: Castellano
Asignado al alumno con DNI:
Asignado al alumno con DNI:21035972B
Asignado al alumno con DNI:26517260P
Asignado al alumno con DNI:77359880R
Asignado al alumno con DNI:77688545L

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

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

Conocimientos Previos

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

Objetivos del TFG

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

Metodología a Desarrollar

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

Documentos y Formatos de Entrega

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

Ilustración 246. Guía original del TFG

Posteriormente, se cambió el título a “Remake de The Legend of Zelda (1986)” y, finalmente, se volvió a cambiar a “Desarrollo de un RPG de acción usando técnicas de generación procedural de mazmorras” y se presentó la guía que se ha usado finalmente:

PROPUESTA DE TRABAJO DE FIN DE GRADO

GRADO EN:
Grado en Ingeniería Informática

ESPECIALIDAD:
Tecnologías de la Información

MENCIÓN:
Sistemas Gráficos

TÍTULO DEL TRABAJO:
Desarrollo de un RPG de acción usando técnicas de generación procedural de mazmorras

Idioma:
Castellano

DESCRIPCION CORTA DEL TFG:
Este trabajo consiste en la realización de un RPG de acción usando técnicas de generación procedural de mazmorras utilizando un motor de videojuegos actual, como Unreal Engine, CryEngine o Unity. La temática del videojuego, así como el motor a utilizar, quedan a elección del alumnado.

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

DATOS DEL TRABAJO FIN DE GRADO:

Modalidad del TFG
Proyecto de Ingeniería

Tipo de TFG
- ○ TFG General
- ◉ TFG Específico

TFG en equipo
- ☐ TFG en Equipo

(Debe ajuntarse memoria justificativa)

Número máximo de estudiantes
1

TUTOR DEL TRABAJO FIN DE GRADO:

Tutor/a

Nombre:
CARLOS JAVIER OGAYAR ANGUITA

Departamento:
Informática

A. Conocimiento:
LENGUAJES Y SISTEMAS INFORMÁTICOS

☐ Este TFG tiene un segundo tutor

DATOS DEL ALUMNO ASIGNADO

Alumno/a asignado/a

Nombre:
Ramón Cruz Blanco
DNI:
77688545L
Dirección:

Teléfono:

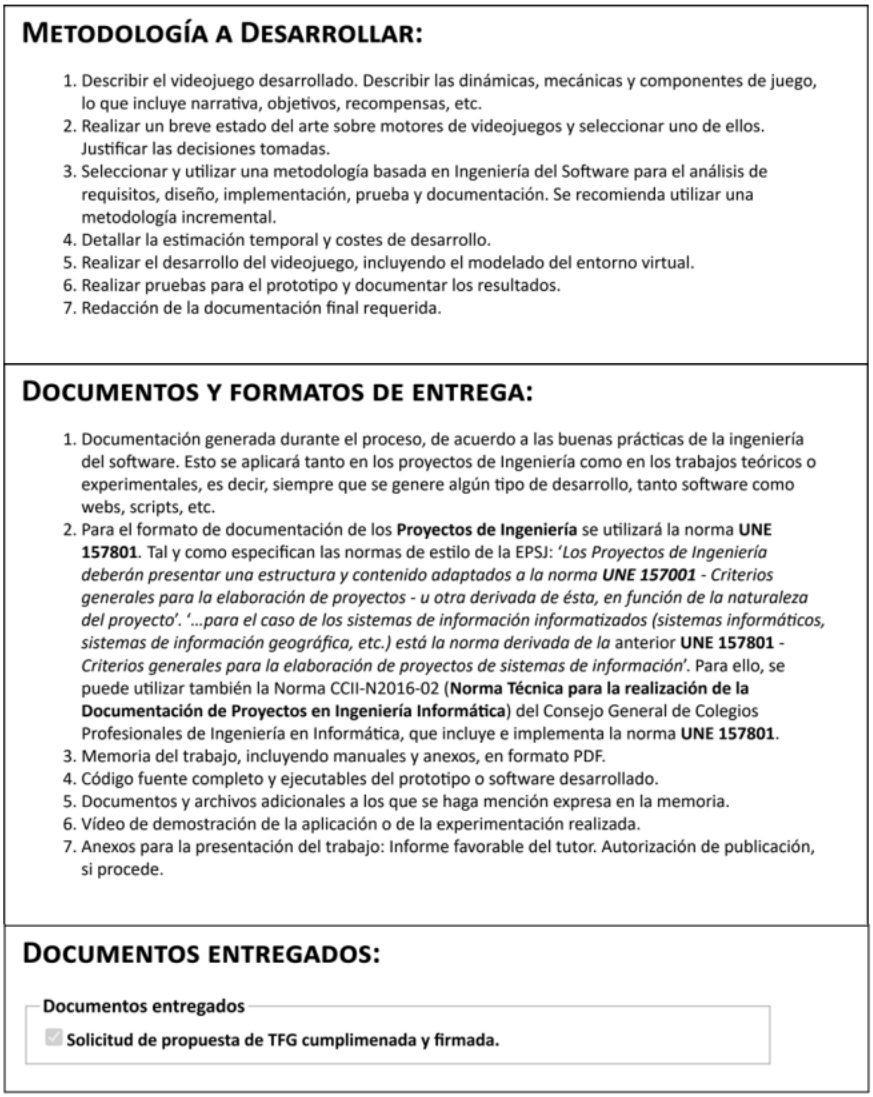
Correo-E:
rcb00029@red.ujaen.es

CONOCIMIENTOS/REQUISITOS PREVIOS RECOMENDADOS

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

OBJETIVOS DEL TFG:

1. Diseño del videojuego tipo RPG, especificando las características y el sistema/entorno objetivo.
2. Diseño del método de generación procedural de mazmorras.
3. Descripción de las dinámicas, mecánicas y componentes de juego, que deberían ser las mismas del juego original. Se permite añadir contenido adicional o variaciones.
4. Diseño de la interfaz de usuario y los mecanismos de interacción.
5. Diseño del entorno virtual 2D/3D, incluyendo todo el material gráfico y sonoro.
6. Desarrollo del juego propuesto, incluyendo la programación de los componentes necesarios.
7. Realización de pruebas de evaluación del sistema y análisis de resultados.



METODOLOGÍA A DESARROLLAR:

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

DOCUMENTOS Y FORMATOS DE ENTREGA:

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

DOCUMENTOS ENTREGADOS:

Documentos entregados

☑ **Solicitud de propuesta de TFG cumplimenada y firmada.**

Ilustración 247. Guía del TFG usada finalmente

5.2 Instalación y configuración del sistema

Para ejecutar el videojuego únicamente tenemos que irnos a la raíz del proyecto y dentro encontraremos la carpeta *Build*. Dentro de esa carpeta nos encontraremos a su vez con un archivo *.exe* llamado *TFG.exe* y únicamente tenemos que ejecutarlo. Mucho cuidado porque el ejecutable hay que mantenerlo dentro de la carpeta *Build*; mientas que esta carpeta puede estar en cualquier parte (no necesariamente dentro de la carpeta del proyecto), el ejecutable si debe permanecer dentro de la carpeta.

5.3 Manuales de usuario

Tenemos dos contextos donde se puede encontrar el usuario: en los **menús** y en el **mundo del juego** propiamente dicho. Para el primer caso, tenemos las siguientes acciones que puede realizar:

1. Desplazarse de un elemento de la interfaz a otro.
2. Pulsar en el elemento de la interfaz.

Con teclado y ratón es sencillo, se puede controlar bien íntegramente con el ratón, de manera que se posa el cursor sobre el elemento y se clicka, bien con las flechas de teclado y se pulsa el elemento con el intro.

Con mando, en cambio, se puede usar bien *joystick*, bien las flechas para pasar de un elemento a otro y se pulsa con el botón sur, la X con mandos de *PlayStation*.

Ahora, en el contexto del mundo de juego disponemos de las siguientes acciones que el jugador puede realizar:

1. Mover la cámara.
2. Mover al jugador.
3. Atacar.
4. Bloquear.
5. Rodar.
6. Fijar.
7. Correr.
8. Cambiar arma derecha.
9. Cambiar arma izquierda.
10. Lanzar bomba.
11. Pausar el juego.

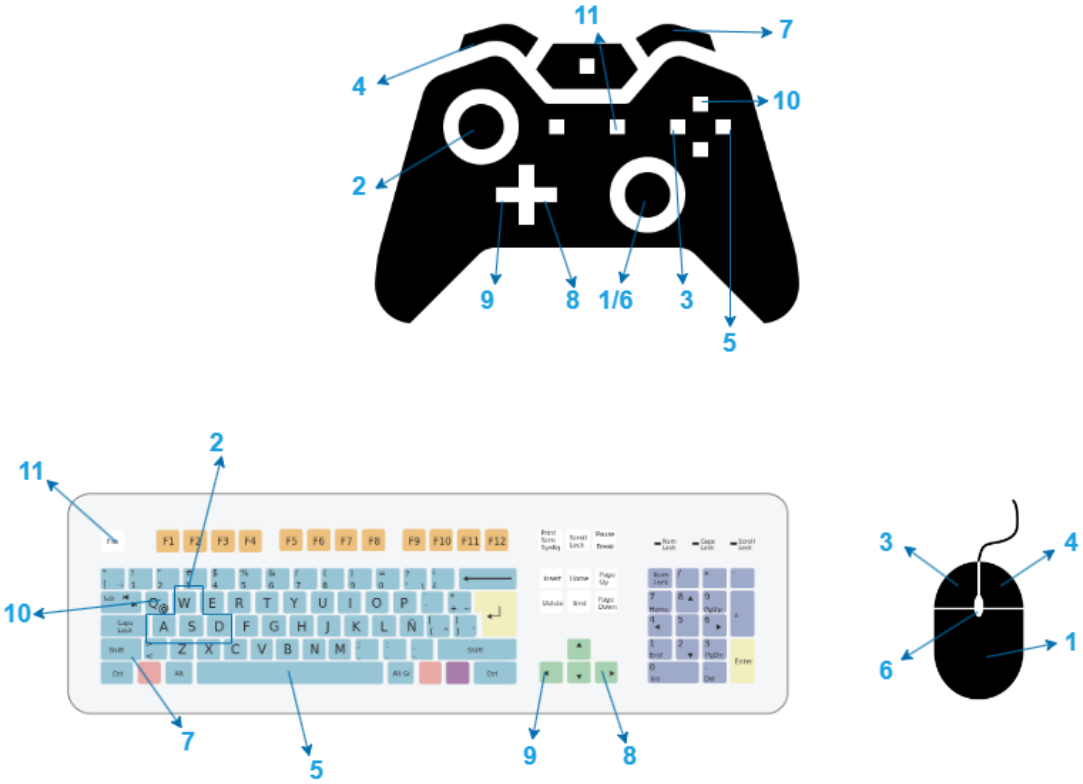


Ilustración 248. Diagrama de los controles del juego

Existen algunas aclaraciones que cabe hacerse con respecto al diagrama:

- En el caso del ratón, la acción 1 (mover la cámara) se hace moviendo el propio ratón.
- En el mando, el *joystick* derecho está asociado a dos acciones: mover la cámara, que se hace moviendo el propio *joystick*, y fijar, que se hace pulsándolo.
- En el teclado, la acción 2 se realiza siguiendo un esquema WASD, de manera que el W hace que el personaje vaya hacia delante, A hacia la izquierda, D hacia la derecha y S hacia atrás.
- Para bloquear con el mando se pulsa el gatillo correspondiente al L1 para mandos de *PlayStation*. Tanto los mandos *PlayStation* como los *Xbox* tienen gatillos en la cara que mira hacia el frente, 2 filas de 2 cada una para ser exactos. Pues bien, para bloquear se pulsa el gatillo de arriba a la izquierda.

- Ídem para la acción de correr, solo que en este caso se deja pulsado el botón de abajo a la derecha, el equivalente al R2 en *PlayStation*, y teniendo en cuenta que tanto para mando como para teclado hay que dejar pulsado R2/Shift mientras se mueve al personaje; de otra forma, no correrá.

6 DEFINICIONES Y ABREVIATURAS

- **RPG (Role-playing game).** Se trata de un género de videojuegos donde se interpreta a un personaje inmerso en algún tipo de mundo detallado, generalmente de fantasía o ciencia ficción. La nomenclatura proviene de los juegos de rol de mesa, de los cuales los videojuegos han heredado elementos como el progreso estadístico de los personajes, terminología, escenarios o mecánicas de juego, así como el énfasis en una narrativa detallada, aunque a día de hoy la definición es muy amplia y abarca videojuegos de muy diversa índole (Wikipedia, 2023a).
- **Nivel de videojuego.** Es el área o escenario total disponible para el jugador a la hora de completar un objetivo específico. Generalmente, se trata solamente de un solo escenario de los muchos que conforman un videojuego, y normalmente está caracterizado porque el jugador debe completar algún tipo de tarea antes de llegar al siguiente nivel. Los niveles de los videojuegos suelen tener una dificultad que aumenta progresivamente para atraer la atención de jugadores de todos los tipos. Cada nivel muestra nuevo contenido y desafíos para mantener alto el interés de los jugadores (Wikipedia, 2023b).
- **Mecánica de juego.** Este es un concepto cuya definición está sujeta a intenso debate y diversos autores proponen diferentes definiciones. Nosotros, en este proyecto, nos guiaremos por la siguiente: las mecánicas son el conjunto de acciones que el jugador puede llevar a cabo que cambian el *game state*, esto es, un cambio en la posición y características concretas de todos los objetos y entornos de un videojuego en un momento preciso en el tiempo (L. Nigromante, 2018); un cambio, en definitiva, en su **modelo lógico**, que usualmente produce

una reacción del sistema (al realizar la acción de saltar, el personaje del jugador se desplaza hacia arriba y luego regresa al suelo, siguiendo algo similar a las leyes de la gravedad). Estas acciones están constreñidas por una serie de **reglas** y el jugador no puede hacer nada más allá de ellas, pues están implícitas en el mismo código del videojuego.

- **Sistema de combate.** Parte del modelo lógico de un videojuego que se encarga de implementar las mecánicas concernientes con las batallas o enfrentamientos entre el personaje del jugador y una serie de enemigos, tales como atacar con una espada o esquivar un ataque.
- **Asset.** En desarrollo de videojuegos, un *asset* viene a ser una representación de cualquier objeto que puede ser utilizado en un juego o proyecto (U. Technologies, n.d.-a). Por lo general, se tratan de modelos 3D, animaciones, texturas… pero en realidad es cualquier recurso usado para la creación de dicho videojuego, incluido el código u otros elementos como los que puede ofrecer el propio motor de videojuegos (un *audio mixer* por ejemplo, en el caso de Unity).
- **Combate a tiempo real.** Se trata del tipo del combate en el cual el tiempo transcurre de la misma forma tanto para el jugador como para los personajes y enemigos, es decir, aquel que más se acerca al combate en la vida real.
- **Combate por turnos.** Inspirado en los juegos de mesa como el ajedrez, en el combate por turnos el jugador (o jugadores) disponen de un turno durante el cual se "congela" el tiempo para pensar su próxima acción, seguidamente del turno del contrincante.
- **Rogue-like.** Es un género de videojuegos cuyo elemento definitorio es la creación aleatoria de mazmorras y la muerte permanente del personaje, de forma que la idea del juego es hacer que el jugador realice varias *runs* o partidas hasta que pueda completarlo. En cada partida además al jugador se le pueden proporcionar algunas ventajas que le pueden ayudar a avanzar en su objetivo de completar el videojuego (R. Cervantes, 2022).

- **Número pseudo-aleatorio.** Se trata de un número generado en un proceso que, a simple vista, parece generar números aleatorios, pero en realidad no lo hace. Las secuencias de números pseudo-aleatorios no muestran ningún patrón o regularidad aparente desde un punto de vista estadístico (si el generador es de cierta calidad), a pesar de haber sido generados por un algoritmo totalmente determinista que, dadas las mismas entradas, producirá las mismas salidas (Wikipedia, 2023c).
- **Semilla.** Número que se usa como entrada de un generador de números pseudo-aleatorios. Si le damos a un generador de este tipo la misma semilla, el generador comenzará a devolver la misma secuencia de números (Wikipedia, 2023c).
- **Non Playable Character (NPC).** Se trata de todos aquellos personajes, generalmente dotados con cierta inteligencia, que pueblan el mundo del videojuego. Pueden ser tanto amistosos como enemigos.
- **Tasas de frames.** Se trata de la cantidad de imágenes consecutivas que un producto audiovisual nos enseña por pantalla medidas por unidad de tiempo, generalmente en segundos, puesto que, cuando vemos un video o jugamos a un videojuego, lo que vemos en realidad es una secuencia muy rápida de imágenes que al ojo humano le producen una sensación de movimiento (Y. Fernández, 2020).
- **Rendering pipeline.** En informática gráfica, una *pipeline* de renderizado es la secuencia de pasos que se deben dar para que un determinado software sea capaz de renderizar objetos, es decir, de mostrar dichos objetos por pantalla a partir de los datos que los representan. En *OpenGL* por ejemplo, la secuencia típica es *Vertex Specification -> Vertex Shader -> Tesselation -> Geometry Shader -> Vertex Post-Processing -> Primitive Assembly -> Rasterization -> Fragment Shader -> Per-Sample Operations* (OpenGL Wiki, n.d.).
- **Game object.** Los *GameObjects* son objetos fundamentales en Unity que representan personajes, props, y el escenario. Estos no logran nada por sí mismos pero funcionan como contenedores para *Components*, que implementan la verdadera funcionalidad (U. Technologies, n.d.-b).

- **Props.** En videojuegos, los *props* son la utillería, fundamentalmente decorado para escenarios y otros objetos que podemos encontrar en un videojuego.
- **Números aleatorios normalizados.** Decimos que los números aleatorios normalizados son los que proporciona una variable aleatoria normal $N(\mu, \sigma)$ cuya función de densidad de probabilidad viene dada por:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

  (A. Hernández, n.d.). Las distribuciones normales tienen la particularidad de que los valores que puede adquirir la variable se concentran en el centro, de manera que es poco probable que aparezcan valores extremos (pero no imposible).
- **Isomorfismo.** En teoría de grafos, el isomorfismo entre dos grafos G y H se define como una biyección f entre los conjuntos de sus vértices tal que $f: V(G) \to V(H)$ que preserva la relación de adyacencia. Es decir, cualesquiera dos vértices u y v de G son adyacentes si y solo si lo son sus imágenes, $f(u)$ y $f(v)$ en H (Wikipedia, 2023e). De forma simplificada, significa que, en esencia, son el mismo grafo.
- **Ángulos de Euler.** Los ángulos de Euler asociados a un determinado cuerpo se definen como:
  - ϕ es el ángulo entre $\hat{\imath}$ e $\hat{\imath}'$.
  - θ es el ángulo entre $\hat{k}$ y $\widehat{k_c}$.
  - ψ es el ángulo entre $\hat{\imath}'$ y $\widehat{\imath_c}$.

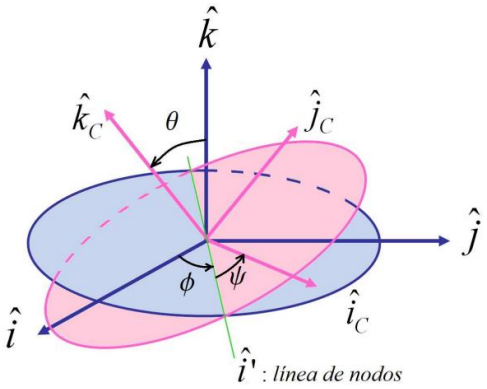


Ilustración 249. Sistema de referencia usado para mostrar los ángulos de Euler (R. Ferraro, 2021)

Pese a que la formulación matemática es farragosa, en la práctica en *Unity* simplemente los podemos pensar como ángulos alrededor de cada uno de los ejes del sistema de referencia xyz.

- **Collider.** Los componentes *Collider* en *Unity* definen la forma de un objeto para los propósitos de colisiones físicas, es decir, son superficies cuya forma escoge el desarrollador para que los objetos sean capaces de interactuar entre ellos por medio del sistema de físicas (U. Technologies, n.d.-c).
- **Raycast.** Proceso por el cual *Unity castea* un rayo para comprobar con qué *colliders* interseccionа en su trayectoria. Útil para cosas como detectar si un enemigo tiene visión directa con el jugador, por ejemplo.
- **Corrutina.** Una corrutina es una función que tiene la habilidad de pausar su ejecución y devolver el control a *Unity* para luego continuar donde lo dejó en el siguiente *frame* (U. Technologies, n.d.-d).
- **Rigidbody.** Los *Rigidbodies* le permite a los *GameObjects* actuar bajo el control de la física. El *Rigidbody* puede recibir fuerza y torque para hacer que los objetos se muevan en una manera realista (U. Technologies, n.d.-e).
- **Interpolación.** En el subcampo matemático del análisis numérico, se denomina interpolación a obtención de nuevos puntos partiendo del conocimiento de un conjunto de puntos (Wikipedia, 2020).

- **Cuaternión.** Se tratan de vectores de 4 componentes $Q(s, v) = (s, x, y, z)$ donde $v = (x, y, z)$ designa un eje de rotación y s representa la cantidad de rotación, esto es, un ángulo. Resumidamente, los cuaterniones son una forma de representar rotaciones.
- **Metáfora visual.** Una metáfora visual en el contexto de una interfaz de usuario es una imagen que nos permite representar alguna cosa de tal manera que el usuario pueda reconocer lo que representa y conocer su propósito. Estas metáforas deben ser lo más universales posibles para hacer que la mayor cantidad de usuarios posibles sean capaces de reconocerlas.

7 BIBLIOGRAFÍA

- Colaboradores de Wikipedia. (2023a). Videojuego de rol. *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/Videojuego_de_rol
- Colaboradores de Wikipedia. (2023b). Nivel (videojuegos). *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/Nivel_(videojuegos)
- Fdez, I. (2016). Procedurally Generated Content: la revolución de los videojuegos es ahora (aunque llevamos 40 años creándola). *Xataka*. https://www.xataka.com/videojuegos/procedurally-generated-content-la-revolucion-de-los-videojuegos-es-ahora-aunque-llevamos-40-anos-creandola
- Nigromante, L. (2018). ¿Qué son las mecánicas de juego? Una aproximación al concepto. *Todas Gamers*. https://todasgamers.com/2017/07/09/las-mecanicas-juego-una-aproximacion-al-concepto/
- Technologies, U. (n.d.-a). *Unity - Manual: Flujo de trabajo de los Assets (Asset Workflow)*. https://docs.unity3d.com/es/530/Manual/AssetWorkflow.html#:~:text=Un%20asset%20es%20una%20representaci%C3%B3n,de%20archivos%20que%20Unity%20soporta.
- Cervantes, R. (2022). ¿Qué es un roguelike? Aquí tienes los 5 mejores ejemplos del género. *3DJuegos Guías*. https://www.3djuegosguias.com/otros-generos/que-roguelike-aqui-tienes-5-mejores-ejemplos-genero
- Colaboradores de Wikipedia. (2023c). Número pseudoaleatorio. *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/N%C3%BAmero_pseudoaleatorio
- Gregory, J. M. (2009). Game Engine Architecture. In *A K Peters/CRC Press eBooks*. https://doi.org/10.1201/b10681
- Arora, S. K. (n.d.). Unity vs Unreal: Which Game Engine Should You Choose? *Hackr.io*. https://hackr.io/blog/unity-vs-unreal-engine#:~:text=Graphics%3A%20Both%20tools%20produce%20excellent,post-processing%20and%20game%20development

- *Rendering Pipeline Overview - OpenGL Wiki*. (n.d.). https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
- Technologies, U. (n.d.-b). *GameObject - Unity Manual*. https://docs.unity3d.com/es/2018.4/Manual/class-GameObject.html
- Mendicute Arminio, J. R. (2018). *Generación Automática de Mazmorras* [Trabajo de Fin de Grado]. Universidad de Cantabria. https://repositorio.unican.es/xmlui/bitstream/handle/10902/15267/Mendicute%20Arminio%20Juan%20Ramon.pdf?sequence=1&isAllowed=y
- Kun, J. (2017). *The Cellular Automaton Method for Cave Generation*. Math ∩ Programming. https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/
- Eiben, A. E., & Smith, J. G. (2003). Introduction to Evolutionary Computing. In *Natural computing series*. Springer Science+Business Media. https://doi.org/10.1007/978-3-662-05094-1
- Pressman, R. S. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science, Engineering & Mathematics.
- Martins, J. (2022). Diagrama de Gantt: qué es y cómo crear uno con ejemplos [2022] • Asana. *Asana*. https://asana.com/es/resources/gantt-chart-basics
- Adams, E. (2013). *Fundamentals of Game Design*. New Riders.
- Adonaac, A. (2015). Procedural Dungeon Generation Algorithm. *Game Developer*. https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm
- Hernández, A. (n.d). *Distribución Normal*. (n.d.). http://www4.ujaen.es/~rmfernan/Distribucionesprobabilidad/Normal.html
- Okabe, A., Boots, B., Sugihara, K., & Chiu, S. N. (2009). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons.
- Colaboradores de Wikipedia. (2022). Grafo ponderado. *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/Grafo_ponderado
- Colaboradores de Wikipedia. (2023d). Minimum spanning tree. *Wikipedia*. https://en.wikipedia.org/wiki/Minimum_spanning_tree

- Brassard, G., & Bratley, P. (1996). *Fundamentals of Algorithmics*. Englewood, N.J. : Prentice Hall.
- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to Automata Theory, Languages, and Computation*. Pearson.
- J. Dormans (2010). Adventures in level design: Generating missions and spaces for action adventure games. Workshop on Procedural Content Generation in Games, PCGames. http://doi.acm.org/10.1145/1814256.1814257
- Colaboradores de Wikipedia. (2023e). Isomorfismo de grafos. *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/Isomorfismo_de_grafos
- French, J. (2022). Input in Unity made easy (complete guide to the new system). *Game Dev Beginner*. https://gamedevbeginner.com/input-in-unity-made-easy-complete-guide-to-the-new-system/#input_system
- Brackeys. (2020a). *THIRD PERSON MOVEMENT in Unity* [Video]. YouTube. https://www.youtube.com/watch?v=4HpC--2iowE
- Rodney. (2021). LEY DE LOS SENOS. *Leyes Del Universo*. https://leyesdeluniverso.es/funcion-trigonometrica-ley-de-los-senos/
- Ferraro, R. (2021). *Mecánica Clásica, Clase 14 (Cinemática del cuerpo rígido)* [Apuntes de clase]. Universidad de Buenos Aires. http://materias.df.uba.ar/mcaa2021c1/files/2019/02/Clase-141.pdf
- Technologies, U. (n.d.-c). *Colliders - Unity Manual*. https://docs.unity3d.com/es/2018.4/Manual/CollidersOverview.html
- Technologies, U. (n.d.-d). *Unity - Manual: Corrutinas*. https://docs.unity3d.com/es/530/Manual/Coroutines.html
- Technologies, U. (n.d.-e). *Rigidbody - Unity Manual*. https://docs.unity3d.com/es/2019.4/Manual/class-Rigidbody.html
- Colaboradores de Wikipedia. (2020). Interpolación. *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/Interpolaci%C3%B3n
- Technologies, U. (n.d.-f). *Unity - Scripting API: Vector3.Lerp*. https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html

- Brackeys. (2020b). *How to make a HEALTH BAR in Unity!* [Video]. YouTube. https://www.youtube.com/watch?v=BLfNP4Sc_iA
- Brackeys. (2017a). *START MENU in Unity* [Video]. YouTube. https://www.youtube.com/watch?v=zc8ac_qUXQY
- Brackeys. (2017b). *SETTINGS MENU in Unity* [Video]. YouTube. https://www.youtube.com/watch?v=YOaYQrN1oYQ
- Brackeys. (2017c). *PAUSE MENU in Unity* [Video]. YouTube. https://www.youtube.com/watch?v=JivuXdrIHK0